

Fundamentals-Of-Algorithms Archive

Analysis of Algorithms | Set 1 (Asymptotic Analysis)

Why performance analysis?

There are many important things that should be taken care of, like user friendliness, modularity, security, maintainability, etc. Why to worry about performance?

The answer to this is simple, we can have all the above things only if we have performance. So performance is like currency through which we can buy all the above things. Another reason for studying performance is – speed is fun!

Given two algorithms for a task, how do we find out which one is better?

One naive way of doing this is – implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time. There are many problems with this approach for analysis of algorithms.

1) It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better.

2) It might also be possible that for some inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs.

Asymptotic Analysis is the big idea that handles above issues in analyzing algorithms. In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time). We calculate, how does the time (or space) taken by an algorithm increases with the input size.

For example, let us consider the search problem (searching a given item) in a sorted array. One way to search is Linear Search (order of growth is linear) and other way is Binary Search (order of growth is logarithmic). To understand how Asymptotic Analysis solves the above mentioned problems in analyzing algorithms, let us say we run the Linear Search on a fast computer and Binary Search on a slow computer. For small values of input array size n, the fast computer may take less time. But, after certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine. The reason is the order of growth of Binary Search with respect to input size logarithmic while the order of growth of Linear Search is linear. So the machine dependent constants can always be ignored after certain values of input size.

Does Asymptotic Analysis always work?

Asymptotic Analysis is not perfect, but that's the best way available for analyzing algorithms. For example, say there are two sorting algorithms that take $1000n\log n$ and $2n\log n$ time respectively on a machine. Both of these algorithms are asymptotically same (order of growth is $n\log n$). So, With Asymptotic Analysis, we can't judge which one is better as we ignore constants in Asymptotic Analysis.

Also, in Asymptotic analysis, we always talk about input sizes larger than a constant value. It might be possible that those large inputs are never given to your software and an algorithm which is asymptotically slower, always performs better for your particular situation. So, you may end up choosing an algorithm that is Asymptotically slower but faster for your software.

Next – [Analysis of Algorithms | Set 2 \(Worst, Average and Best Cases\)](#)

References:

[MIT's Video lecture 1 on Introduction to Algorithms](#).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Analysis
Articles

Analysis of Algorithms | Set 2 (Worst, Average and Best Cases)

In the [previous post](#), we discussed how Asymptotic analysis overcomes the problems of naive way of analyzing algorithms. In this post, we will take an example of Linear Search and analyze it using Asymptotic analysis.

We can have three cases to analyze an algorithm:

- 1) Worst Case
- 2) Average Case
- 3) Best Case

Let us consider the following implementation of Linear Search.

```
#include <stdio.h>

// Linearly search x in arr[]. If x is present then return the index,
// otherwise return -1
int search(int arr[], int n, int x)
{
    int i;
    for (i=0; i<n; i++)
    {
        if (arr[i] == x)
            return i;
    }
    return -1;
}

/* Driver program to test above functions*/
int main()
{
    int arr[] = {1, 10, 30, 15};
    int x = 30;
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("%d is present at index %d", x, search(arr, n, x));

    getchar();
    return 0;
}
```

Worst Case Analysis (Usually Done)

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x in the above code) is not present in the array. When x is not present, the search() function compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be $\Theta(n)$.

Average Case Analysis (Sometimes done)

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are [uniformly distributed](#) (including the case of x not being present in array). So we sum all the cases and divide the sum by $(n+1)$. Following is the value of average case time complexity.

Average Case Time = ?

= ?

= $\Theta(n)$

Best Case Analysis (Bogus)

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be $\Theta(1)$. Most of the times, we do worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information.

The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs. The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

For some algorithms, all the cases are asymptotically same, i.e., there are no worst and best cases. For example, Merge Sort. Merge Sort does $\Theta(n\log n)$ operations in all cases. Most of the other sorting algorithms have worst and best cases. For example, in the typical implementation of Quick Sort (where pivot is chosen as a corner element), the worst occurs when the input array is already sorted and the best occur when the pivot elements always divide array in two halves. For insertion sort, the worst case occurs when the array is reverse sorted and the best case occurs when the array is sorted in the same order as output.

Next – [Analysis of Algorithms | Set 3 \(Asymptotic Notations\)](#)

References:

MIT's Video lecture 1 on Introduction to Algorithms.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Analysis
Articles

Analysis of Algorithms | Set 3 (Asymptotic Notations)

We have discussed [Asymptotic Analysis](#), and [Worst, Average and Best Cases of Algorithms](#). The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, and doesn't require algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.

thetanotation



1) **Θ Notation:** The theta notation bounds a function from above and below, so it defines exact asymptotic behavior.

A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression.

$$3n^3 + 6n^2 + 6000 = \Theta(n^3)$$

Dropping lower order terms is always fine because there will always be a n_0 after which $\Theta(n^3)$ has higher values than $\Theta(n^2)$ irrespective of the constants involved.

For a given function $g(n)$, we denote $\Theta(g(n))$ is following set of functions.

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$

The above definition means, if $f(n)$ is theta of $g(n)$, then the value $f(n)$ is always between $c_1 g(n)$ and $c_2 g(n)$ for large values of n ($n \geq n_0$). The definition of theta also requires that $f(n)$ must be non-negative for values of n greater than n_0 .

BigO



2) **Big O Notation:** The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is $O(n^2)$. Note that $O(n^2)$ also covers linear time.

If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:

1. The worst case time complexity of Insertion Sort is $\Theta(n^2)$.
2. The best case time complexity of Insertion Sort is $\Theta(n)$.

The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

BigOmega



3) **Ω Notation:** Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

Ω Notation can be useful when we have lower bound on time complexity of an algorithm. As discussed in the previous post, the [best case performance of an algorithm is generally not useful](#), the Omega notation is the least used notation among all three.

For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions.

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as $\Omega(n)$, but it is not a very useful information about insertion sort, as we are generally interested in worst case and sometimes in average case.

Exercise:

Which of the following statements is/are valid?

1. Time Complexity of QuickSort is $\Theta(n^2)$

2. Time Complexity of QuickSort is $O(n^2)$
3. For any two functions $f(n)$ and $g(n)$, we have $f(n) = O(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
4. Time complexity of all computer algorithms can be written as $\Omega(1)$

Next → [Analysis of Algorithms | Set 4 \(Analysis of Loops\)](#)

References:

[Lec 1 | MIT \(Introduction to Algorithms\)](#)

[Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](#)

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Analysis
Articles

Analysis of Algorithms | Set 4 (Analysis of Loops)

We have discussed [Asymptotic Analysis](#), [Worst](#), [Average](#) and [Best Cases](#) and [Asymptotic Notations](#) in previous posts. In this post, analysis of iterative programs with simple examples is discussed.

- 1) **$O(1)$** : Time complexity of a function (or set of statements) is considered as $O(1)$ if it doesn't contain loop, recursion and call to any other non-constant time function.

```
// set of non-recursive and non-loop statements
```

For example `swap()` function has $O(1)$ time complexity.

A loop or recursion that runs a constant number of times is also considered as $O(1)$. For example the following loop is $O(1)$.

```
// Here c is a constant
for (int i = 1; i <= c; i++) {
    // some O(1) expressions
}
```

- 2) **$O(n)$** : Time Complexity of a loop is considered as $O(n)$ if the loop variables is incremented / decremented by a constant amount. For example following functions have $O(n)$ time complexity.

```
// Here c is a positive integer constant
for (int i = 1; i <= n; i += c) {
    // some O(1) expressions
}

for (int i = n; i > 0; i -= c) {
    // some O(1) expressions
}
```

- 3) **$O(n^2)$** : Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example the following sample loops have $O(n^2)$ time complexity

```
for (int i = 1; i <= n; i += c) {
    for (int j = 1; j <= n; j += c) {
        // some O(1) expressions
    }
}

for (int i = n; i > 0; i += c) {
    for (int j = i+1; j <= n; j += c) {
        // some O(1) expressions
    }
}
```

For example [Selection sort](#) and [Insertion Sort](#) have $O(n^2)$ time complexity.

- 4) **$O(\log n)$** Time Complexity of a loop is considered as $O(\log n)$ if the loop variables is divided / multiplied by a constant amount.

```
for (int i = 1; i <= n; i *= c) {
    // some O(1) expressions
}

for (int i = n; i > 0; i /= c) {
    // some O(1) expressions
}
```

For example [Binary Search](#)(refer [iterative implementation](#)) has $O(\log n)$ time complexity.

- 5) **$O(\log \log n)$** Time Complexity of a loop is considered as $O(\log \log n)$ if the loop variables is reduced / increased exponentially by a constant amount.

```
// Here c is a constant greater than 1
for (int i = 2; i <= n; i = pow(i, c)) {
    // some O(1) expressions
}

// Here fun is sqrt or cuberoot or any other constant root
for (int i = n; i > 0; i = fun(i)) {
    // some O(1) expressions
}
```

See [this](#) for more explanation.

How to combine time complexities of consecutive loops?

When there are consecutive loops, we calculate time complexity as sum of time complexities of individual loops.

```
for (int i = 1; i <= m; i += c) {
    // some O(1) expressions
}
for (int i = 1; i <= n; i += c) {
    // some O(1) expressions
}

Time complexity of above code is  $O(m) + O(n)$  which is  $O(m+n)$ .
If  $m = n$ , the time complexity becomes  $O(2n)$  which is  $O(n)$ .
```

How to calculate time complexity when there are many if, else statements inside loops?

As discussed [here](#), worst case time complexity is the most useful among best, average and worst. Therefore we need to consider worst case. We evaluate the situation when values in if-else conditions cause maximum number of statements to be executed.

For example consider the [linear search function](#) where we consider the case when element is present at the end or not present at all.

When the code is too complex to consider all if-else cases, we can get an upper bound by ignoring if else and other complex control statements.

How to calculate time complexity of recursive functions?

Time complexity of a recursive function can be written as a mathematical recurrence relation. To calculate time complexity, we must know how to solve recurrences. We will soon be discussing recurrence solving techniques as a separate post.

Quiz on Analysis of Algorithms

Next – Analysis of Algorithm | Set 4 (Solving Recurrences)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Analysis
Articles

Analysis of Algorithm | Set 4 (Solving Recurrences)

In the previous post, we discussed [analysis of loops](#). Many algorithms are recursive in nature. When we analyze them, we get a recurrence relation for time complexity. We get running time on an input of size n as a function of n and the running time on inputs of smaller sizes. For example in [Merge Sort](#), to sort a given array, we divide it in two halves and recursively repeat the process for the two halves. Finally we merge the results. Time complexity of Merge Sort can be written as $T(n) = 2T(n/2) + cn$. There are many other algorithms like Binary Search, Tower of Hanoi, etc.

There are mainly three ways for solving recurrences.

1) Substitution Method: We make a guess for the solution and then we use mathematical induction to prove the the guess is correct or incorrect.

For example consider the recurrence $T(n) = 2T(n/2) + n$

We guess the solution as $T(n) = O(n\log n)$. Now we use induction to prove our guess.

We need to prove that $T(n) \leq cn\log n$. We can assume that it is true for values smaller than n .

```
T(n) = 2T(n/2) + n
<= cn/2Log(n/2) + n
= cnLog(n/2) + cn
= cnLog(n) + cn
<= cnLog n
```

2) Recurrence Tree Method: In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically a arithmetic or geometric series.

For example consider the recurrence relation
 $T(n) = T(n/4) + T(n/2) + cn^2$

```
cn^2
/
T(n/4)  T(n/2)
```

If we further break down the expression $T(n/4)$ and $T(n/2)$, we get following recursion tree.

```
cn^2
/
c(n^2)/16  c(n^2)/4
/   \   /   \
T(n/16)  T(n/8)  T(n/8)  T(n/4)
```

Breaking down further gives us following

```
cn^2
/
c(n^2)/16  c(n^2)/4
/   \   /   \
c(n^2)/256  c(n^2)/64  c(n^2)/64  c(n^2)/16
/   \   /   \
/   \   /   \
```

To know the value of $T(n)$, we need to calculate sum of tree nodes level by level. If we sum the above tree level by level, we get the following series

$T(n) = c(n^2) + 5(n^2)/16 + 25(n^2)/256 + \dots$

The above series is geometrical progression with ratio 5/16.

To get an upper bound, we can sum the infinite series.

We get the sum as $(n^2)/(1 - 5/16)$ which is $O(n^2)$

3) Master Method:

Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type.

$T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b > 1$

There are following three cases:

1. If $f(n) = \Theta(n^c)$ where $c < \log_b a$ then $T(n) = \Theta(n^{\log_b a})$

2. If $f(n) = \Theta(n^c)$ where $c = \log_b a$ then $T(n) = \Theta(n^c \log n)$

3. If $f(n) = \Theta(n^c)$ where $c > \log_b a$ then $T(n) = \Theta(f(n))$

How does this work?

Master method is mainly derived from recurrence tree method. If we draw recurrence tree of $T(n) = aT(n/b) + f(n)$, we can see that the work done at root is $f(n)$ and work done at all leaves is $\Theta(n^c)$ where c is $\log_b a$. And the height of recurrence tree is $\log_b n$.



In recurrence tree method, we calculate total work done. If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1). If work done at leaves and root is asymptotically same, then our result becomes height multiplied by work done at any level (Case 2). If work done at root is asymptotically more, then our result becomes work done at root (Case 3).

Examples of some standard algorithms whose time complexity can be evaluated using Master Method

Merge Sort: $T(n) = 2T(n/2) + \Theta(n)$. It falls in case 2 as $c = 1$ and $\log_b a = 1$ is also 1. So the solution is $\Theta(n \log n)$

Binary Search: $T(n) = T(n/2) + \Theta(1)$. It also falls in case 2 as $c = 0$ and $\log_b a = 0$ is also 0. So the solution is $\Theta(\log n)$

Notes:

1) It is not necessary that a recurrence of the form $T(n) = aT(n/b) + f(n)$ can be solved using Master Theorem. The given three cases have some gaps between them. For example, the recurrence $T(n) = 2T(n/2) + n/\log n$ cannot be solved using master method.

2) Case 2 can be extended for $f(n) = \Theta(n^c \log^k n)$

If $f(n) = \Theta(n^c \log^k n)$ for some constant $k \geq 0$ and $c = \log_b a$, then $T(n) = \Theta(n^c \log^{k+1} n)$

Practice Problems and Solutions on Master Theorem.

Next – [Analysis of Algorithm | Set 5 \(Amortized Analysis Introduction\)](#)

References:

http://en.wikipedia.org/wiki/Master_theorem

MIT Video Lecture on Asymptotic Notation | Recurrences | Substitution, Master Method

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Analysis

Analysis of Algorithm | Set 5 (Amortized Analysis Introduction)

Amortized Analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyze a sequence of operations and guarantee a worst case average time which is lower than the worst case time of a particular expensive operation.

The example data structures whose operations are analyzed using Amortized Analysis are Hash Tables, Disjoint Sets and Splay Trees.

Let us consider an example of a simple hash table insertions. How do we decide table size? There is a trade-off between space and time, if we make hash-table size big, search time becomes fast, but space required becomes high.

Dynamic Table



The solution to this trade-off problem is to use **Dynamic Table (or Arrays)**. The idea is to increase size of table whenever it becomes full. Following are the steps to follow when table becomes full.

- 1) Allocate memory for a larger table of size, typically twice the old table.
- 2) Copy the contents of old table to new table.
- 3) Free the old table.

If the table has space available, we simply insert new item in available space.

What is the time complexity of n insertions using the above scheme?

If we use simple analysis, the worst case cost of an insertion is $O(n)$. Therefore, worst case cost of n inserts is $n * O(n)$ which is $O(n^2)$. This analysis gives an upper bound, but not a tight upper bound for n insertions as all insertions don't take $O(n)$ time.

AmortizedAnalysis



So using Amortized Analysis, we could prove that the Dynamic Table scheme has $O(1)$ insertion time which is a great result used in hashing. Also, the concept of dynamic table is used in vectors in C++.

ArrayList in Java.

Following are few important notes.

1) Amortized cost of a sequence of operations can be seen as expenses of a salaried person. The average monthly expense of the person is less than or equal to the salary, but the person can spend more money in a particular month by buying a car or something. In other months, he or she saves money for the expensive month.

2) The above Amortized Analysis done for Dynamic Array example is called **Aggregate Method**. There are two more powerful ways to do Amortized analysis called **Accounting Method** and **Potential Method**. We will be discussing the other two methods in separate posts.

3) The amortized analysis doesn't involve probability. There is also another different notion of average case running time where algorithms use randomization to make them faster and expected running time is faster than the worst case running time. These algorithms are analyzed using Randomized Analysis. Examples of these algorithms are Randomized Quick Sort, Quick Select and Hashing. We will soon be covering Randomized analysis in a different post.

Sources:

Berkeley Lecture 35: Amortized Analysis

MIT Lecture 13: Amortized Algorithms, Table Doubling, Potential Method

<http://www.cs.cornell.edu/courses/cs3110/2011sp/lectures/lec20-amortized/amortized.htm>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Analysis

What does 'Space Complexity' mean?

Space Complexity:

The term Space Complexity is misused for Auxiliary Space at many places. Following are the correct definitions of Auxiliary Space and Space Complexity.

Auxiliary Space is the extra space or temporary space used by an algorithm.

Space Complexity of an algorithm is total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

For example, if we want to compare standard sorting algorithms on the basis of space, then Auxiliary Space would be a better criteria than Space Complexity. Merge Sort uses $O(n)$ auxiliary space, Insertion sort and Heap Sort use $O(1)$ auxiliary space. Space complexity of all these sorting algorithms is $O(n)$ though.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Analysis

Pseudo-polynomial Algorithms

What is Pseudo-polynomial?

An algorithm whose worst case time complexity depends on numeric value of input (not number of inputs) is called Pseudo-polynomial algorithm.

For example, consider the problem of counting frequencies of all elements in an array of positive numbers. A pseudo-polynomial time solution for this is to first find the maximum value, then iterate from 1 to maximum value and for each value, find its frequency in array. This solution requires time according to maximum value in input array, therefore pseudo-polynomial. On the other hand, an algorithm whose time complexity is only based on number of elements in array (not value) is considered as polynomial time algorithm.

Pseudo-polynomial and NP-Completeness

Some NP-Complete problems have Pseudo Polynomial time solutions. For example, Dynamic Programming Solutions of 0-1 Knapsack, Subset-Sum and Partition problems are Pseudo-Polynomial. NP complete problems that can be solved using a pseudo-polynomial time algorithms are called weakly NP-complete.

Reference:

https://en.wikipedia.org/wiki/Pseudo-polynomial_time

This article is contributed by **Dheeraj Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Analysis

NP-Completeness | Set 1 (Introduction)

We have been writing about efficient algorithms to solve complex problems, like shortest path, Euler graph, minimum spanning tree, etc. Those were all success stories of algorithm designers. In this post, failure stories of computer science are discussed.

Can all computational problems be solved by a computer? There are computational problems that can not be solved by algorithms even with unlimited time. For example Turing Halting problem (Given a program and an input, whether the program will eventually halt when run with that input, or will run forever). Alan Turing proved that general algorithm to solve the halting problem for all possible program-input pairs cannot exist. A key part of the proof is, Turing machine was used as a mathematical definition of a computer and program (Source [Halting Problem](#)).

Status of **NP Complete** problems is another failure story, NP complete problems are problems whose status is unknown. No polynomial time algorithm has yet been discovered for any NP complete problem, nor has anybody yet been able to prove that no polynomial-time algorithm exists for any of them. The interesting part is, if any one of the NP complete problems can be solved in polynomial time, then all of them can be solved.

What are NP, P, NP-complete and NP-Hard problems?

P is set of problems that can be solved by a deterministic Turing machine in Polynomial time.

NP is set of decision problems that can be solved by a Non-deterministic Turing Machine in Polynomial time. P is subset of NP (any problem that can be solved by deterministic machine in polynomial time can also be solved by non-deterministic machine in polynomial time).

Informally, NP is set of decision problems which can be solved by a polynomial time via a "Lucky Algorithm", a magical algorithm that always makes a right guess among the given set of choices (Source [Ref 1](#)).

NP-complete problems are the hardest problems in **NP** set. A decision problem L is NP-complete if:

- 1) L is in NP (Any given solution for NP-complete problems can be verified quickly, but there is no efficient known solution).
- 2) Every problem in NP is reducible to L in polynomial time (Reduction is defined below).

A problem is **NP-Hard** if it follows property 2 mentioned above, doesn't need to follow property 1. Therefore, NP-Complete set is also a subset of NP-Hard set.

**Decision vs Optimization Problems**

NP-completeness applies to the realm of decision problems. It was set up this way because it's easier to compare the difficulty of decision problems than that of optimization problems. In reality, though, being able to solve a decision problem in polynomial time will often permit us to solve the corresponding optimization problem in polynomial time (using a polynomial number of calls to the decision problem). So, discussing the difficulty of decision problems is often really equivalent to discussing the difficulty of optimization problems. (Source Ref 2).

For example, consider the [vertex cover problem](#) (Given a graph, find out the minimum sized vertex set that covers all edges). It is an optimization problem. Corresponding decision problem is, given undirected graph G and k, is there a vertex cover of size k?

What is Reduction?

Let L_1 and L_2 be two decision problems. Suppose algorithm A_2 solves L_2 . That is, if y is an input for L_2 then algorithm A_2 will answer Yes or No depending upon whether y belongs to L_2 or not.

The idea is to find a transformation from L_1 to L_2 so that the algorithm A_2 can be part of an algorithm A_1 to solve L_1 .

reduction



Learning reduction in general is very important. For example, if we have library functions to solve certain problem and if we can reduce a new problem to one of the solved problems, we save a lot of time. Consider the example of a problem where we have to find minimum product path in a given directed graph where product of path is multiplication of weights of edges along the path. If we have code for Dijkstra's algorithm to find shortest path, we can take log of all weights and use Dijkstra's algorithm to find the minimum product path rather than writing a fresh code for this new problem.

How to prove that a given problem is NP complete?

From the definition of NP-complete, it appears impossible to prove that a problem L is NP-Complete. By definition, it requires us to show every problem in NP is polynomial time reducible to L . Fortunately, there is an alternate way to prove it. The idea is to take a known NP-Complete problem and reduce it to L . If polynomial time reduction is possible, we can prove that L is NP-Complete by transitivity of reduction (If a NP-Complete problem is reducible to L in polynomial time, then all problems are reducible to L in polynomial time).

What was the first problem proved as NP-Complete?

There must be some first NP-Complete problem proved by definition of NP-Complete problems. [SAT \(Boolean satisfiability problem\)](#) is the first NP-Complete problem proved by Cook (See CLRS book for proof).

It is always useful to know about NP-Completeness even for engineers. Suppose you are asked to write an efficient algorithm to solve an extremely important problem for your company. After a lot of thinking, you can only come up exponential time approach which is impractical. If you don't know about NP-Completeness, you can only say that I could not come with an efficient algorithm. If you know about NP-Completeness and prove that the problem is NP-complete, you can proudly say that the polynomial time solution is unlikely to exist. If there is a polynomial time solution possible, then that solution solves a big problem of computer science many scientists have been trying for years.

We will soon be discussing more NP-Complete problems and their proof for NP-Completeness.

References:

[MIT Video Lecture on Computational Complexity](#)

[Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](#)

<http://www.ics.uci.edu/~eppstein/161/960312.html>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Analysis
Articles
NPHard

Polynomial Time Approximation Scheme

It is a very well known fact that there is no known polynomial time solution for [NP Complete problems](#) and these problems occur a lot in real world (See [this](#), [this](#) and [this](#) for example). So there must be a way to handle them. We have seen algorithms to these problems which are ϵ approximate (For example [2 approximate for Travelling Salesman](#)). Can we do better?

Polynomial Time Approximation Scheme (PTAS) is a type of approximate algorithms that provide user to control over accuracy which is a desirable feature. These algorithms take an additional parameter $\epsilon > 0$ and provide a solution that is $(1 + \epsilon)$ approximate for minimization and $(1 - \epsilon)$ for maximization. For example consider a minimization problem, if ϵ is 0.5, then the solution provided by the PTAS algorithm is 1.5 approximate. The running time of PTAS must be polynomial in terms of n , however it can be exponential in terms of ϵ .

In PTAS algorithms, the exponent of the polynomial can increase dramatically as ϵ reduces, for example if the runtime is $O(n^{(1/\epsilon)})$ which is a problem. There is a stricter scheme, **Fully Polynomial Time Approximation Scheme (FPTAS)**. In FPTAS, algorithm need to polynomial in both the problem size n and $1/\epsilon$.

Example (0-1 knapsack problem):

We know that [0-1 knapsack](#) is NP Complete. There is a DP based [pseudo polynomial solution](#) for this. But if input values are high, then the solution becomes infeasible and there is a need of approximate solution. One approximate solution is to use Greedy Approach (compute value per kg for all items and put the highest value per kg first if it is smaller than W), but Greedy approach is not PTAS, so we don't have control over accuracy.

Below is a FPTAS solution for 0-1 Knapsack problem:

Input:

W (Capacity of Knapsack)

$val[0..n-1]$ (Values of Items)

$wt[0..n-1]$ (Weights of Items)

- Find the maximum valued item, i.e., find maximum value in $val[]$. Let this maximum value be $maxVal$.

- Compute adjustment factor k for all values

$$k = (maxVal * \epsilon) / n$$

- Adjust all values, i.e., create a new array $val'[]$ that values divided by k . Do following for every value $val[i]$.

$$val'[i] = \text{floor}(val[i] / k)$$

- Run DP based solution for reduced values, i.e, $val'[0..n-1]$ and all other parameter same.

The above solution works in polynomial time in terms of both n and ϵ . The solution provided by this FPTAS is $(1 - \epsilon)$ approximate. The idea is to round off some of the least significant digits of values then they will be bounded by a polynomial and $1/\epsilon$.

Example:

```
val[] = {12, 16, 4, 8}
wt[] = {3, 4, 5, 2}
W = 10
epsilon = 0.5

maxVal = 20 [maximum value in val[]]
Adjustment factor, k = (16 * 0.5)/4 = 2.0
```

Now we apply DP based solution on below modified instance of problem.

```
val'[] = {6, 8, 2, 4} [ val'[i] = floor(val[i]/k) ]
wt[] = {3, 4, 5, 2}
W = 10
```

How is the solution $(1 - \epsilon) * OPT$?

Here OPT is the optimal value. Let S be the set produced by above FPTAS algorithm and total value of S be $val(S)$. We need to show that

$$val(S) \geq (1 - \epsilon) * OPT$$

Let O be the set produced by optimal solution (the solution with total value OPT), i.e., $val(O) = OPT$.

$$val(O) - k * val'(O)$$

After the dynamic programming step, we get a set that is optimal for the scaled instance

and therefore must be at least as good as choosing the set S with the smaller profits. From that, we can conclude,

$$\begin{aligned} val(S) &\geq k \cdot val'(O) \\ &\geq val(O) - k * val'(O) \\ &\geq OPT - \epsilon * maxVal \\ &\geq OPT - \epsilon * OPT [OPT \geq maxVal] \\ &\geq (1 - \epsilon) * OPT \end{aligned}$$

Sources:

<http://math.mit.edu/~goemans/18434S06/knapsack-katherine.pdf>

https://en.wikipedia.org/wiki/Polynomial-time_approximation_scheme

This article is contributed by Dheeraj Gupta. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the G

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner

Company Wise Coding Practice



A Time Complexity Question

What is the time complexity of following function fun()? Assume that $\log(x)$ returns log value in base 2.

```
void fun()
{
    int i, j;
    for (i=1; i<=n; i++)
        for (j=1; j<=log(i); j++)
            printf("GeeksforGeeks");
}
```

Time Complexity of the above function can be written as $\Theta(\log 1) + \Theta(\log 2) + \Theta(\log 3) + \dots + \Theta(\log n)$ which is $\Theta(\log n!)$

Order of growth of ' $\log n!$ ' and ' $n \log n$ ' is same for large values of n , i.e., $\Theta(\log n!) = \Theta(n \log n)$. So time complexity of fun() is $\Theta(n \log n)$.

The expression $\Theta(\log n!) = \Theta(n \log n)$ can be easily derived from following [Stirling's approximation](#) (or Stirling's formula).

$$\log n! = n \log n - n + O(\log(n))$$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Sources:

http://en.wikipedia.org/wiki/Stirling%27s_approximation

GATE CS Corner Company Wise Coding Practice

Analysis

Time Complexity of building a heap

Consider the following algorithm for building a Heap of an input array A.

```
BUILD-HEAP(A)
  heapsize := size(A);
  for i := floor(heapsize/2) down to 1
    do HEAPIFY(A, i);
  end for
END
```

What is the worst case time complexity of the above algo?

Although the worst case complexity looks like $O(n\log n)$, upper bound of time complexity is $O(n)$. See following links for the proof of time complexity.

<http://www.cse.iitk.ac.in/users/sbaswana/Courses/ESO211/heap.pdf>
http://www.cs.sfu.ca/CourseCentral/307/petra/2009/SLN_2.pdf

GATE CS Corner Company Wise Coding Practice

Analysis
Heap

Time Complexity where loop variable is incremented by 1, 2, 3, 4 ..

What is the time complexity of below code?

```
void fun(int n)
{
  int j = 1, i = 0;
  while (i < n)
  {
    // Some O(1) task
    i = i + j;
    j++;
  }
}
```

The loop variable 'i' is incremented by 1, 2, 3, 4, ... until i becomes greater than or equal to n.

The value of i is $x(x+1)/2$ after x iterations. So if loop runs x times, then $x(x+1)/2$

This article is contributed by **Piyush Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Analysis
Articles

Time Complexity of Loop with Powers

What is the time complexity of below function?

```
void fun(int n, int k)
{
  for (int i=1; i<=n; i++)
  {
    int p = pow(i, k);
    for (int j=1; j<=p; j++)
    {
      // Some O(1) work
    }
  }
}
```

Time complexity of above function can be written as $1^k + 2^k + 3^k + \dots n^k$.

Let us try few examples:

```
k=1
Sum = 1 + 2 + 3 ... n
= n(n+1)/2
= n2 + n/2

k=2
Sum = 12 + 22 + 32 + ... n2.
= n(n+1)(2n+1)/6
= n3/3 + n2/2 + n/6

k=3
Sum = 13 + 23 + 33 + ... n3.
= n2(n+1)2/4
= n4/4 + n3/2 + n2/4
```

In general, asymptotic value can be written as $(n^{k+1})/(k+1) + \Theta(n^k)$

Note that, in asymptotic notations like Θ we can always ignore lower order terms. So the time complexity is $\Theta(n^{k+1} / (k+1))$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Performance of loops (A caching question)

Consider below two C language functions to compute sum of elements in a 2D array. Ignoring the compiler optimizations, which of the two is better implementation of sum?

```
// Function 1
int fun1(int arr[R][C])
{
    int sum = 0;
    for (int i=0; i<R; i++)
        for (int j=0; j<C; j++)
            sum += arr[i][j];
}

// Function 2
int fun2(int arr[R][C])
{
    int sum = 0;
    for (int j=0; j<C; j++)
        for (int i=0; i<R; i++)
            sum += arr[i][j];
}
```

In C/C++, elements are stored in Row-Major order. So the first implementation has better spatial locality (nearby memory locations are referenced in successive iterations). Therefore, first implementation should always be preferred for iterating multidimensional arrays.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Algorithms

Linear Search

Problem: Given an array arr[] of n elements, write a function to search a given element x in arr[].

A simple approach is to do **linear search**, i.e

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
- If x matches with an element, return the index.
- If x doesn't match with any of elements, return -1.

linear-search1



Example:

C/C++

```
// Linearly search x in arr[]. If x is present then return its
// location, otherwise return -1
int search(int arr[], int n, int x)
{
    int i;
    for (i=0; i<n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}
```

Python

```
# Searching an element in a list/array in python
# can be simply done using 'in' operator
# Example:
# if x in arr:
```

```

# print arr.index(x)

# If you want to implement Linear Search in python

# Linearly search x in arr[]
# If x is present then return its location
# else return -1

def search(arr, x):

    for i in range(len(arr)):

        if arr[i] == x:
            return i

    return -1

```

Java

```

class LinearSearch
{
    // This function returns index of element x in arr[]
    static int search(int arr[], int n, int x)
    {
        for (int i = 0; i < n; i++)
        {
            // Return the index of the element if the element
            // is found
            if (arr[i] == x)
                return i;
        }

        // return -1 if the element is not found
        return -1;
    }
}

```

The time complexity of above algorithm is $O(n)$.

Linear search is rarely used practically because other search algorithms such as the binary search algorithm and hash tables allow significantly faster searching comparison to Linear search.

Also See – [Binary Search](#)

Image Source : http://www.nyckidd.com/bob/Linear%20Search%20and%20Binary%20Search_WorkingCopy.pdf

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Searching and Sorting

Binary Search

Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].

A simple approach is to do [linear search](#). The time complexity of above algorithm is $O(n)$. Another approach to perform the same task is using Binary Search.

Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

[binary-search1](#)



Example:

Image Source : http://www.nyckidd.com/bob/Linear%20Search%20and%20Binary%20Search_WorkingCopy.pdf

The idea of binary search is to use the information that the array is sorted and reduce the time complexity to $O(\log n)$.

We strongly recommend that you click here and practice it, before moving on to the solution.

We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.

[Recursive implementation of Binary Search](#)

C/C++

```
#include <stdio.h>

// A recursive binary search function. It returns location of x in
// given array arr[l..r] is present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l)
    {
        int mid = l + (r - l)/2;

        // If the element is present at the middle itself
        if (arr[mid] == x) return mid;

        // If element is smaller than mid, then it can only be present
        // in left subarray
        if (arr[mid] > x) return binarySearch(arr, l, mid-1, x);

        // Else the element can only be present in right subarray
        return binarySearch(arr, mid+1, r, x);
    }

    // We reach here when element is not present in array
    return -1;
}

int main(void)
{
    int arr[] = {2, 3, 4, 10, 40};
    int n = sizeof(arr)/ sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n-1, x);
    (result == -1)? printf("Element is not present in array")
                  : printf("Element is present at index %d", result);
    return 0;
}
```

Python

```
# Python Program for recursive binary search.

# Returns index of x in arr if present, else -1
def binarySearch (arr, l, r, x):

    # Check base case
    if r >= l:

        mid = l + (r - l)/2

        # If element is present at the middle itself
        if arr[mid] == x:
            return mid

        # If element is smaller than mid, then it can only
        # be present in left subarray
        elif arr[mid] > x:
            return binarySearch(arr, l, mid-1, x)

        # Else the element can only be present in right subarray
        else:
            return binarySearch(arr, mid+1, r, x)

    else:
        # Element is not present in the array
        return -1

# Test array
arr = [ 2, 3, 4, 10, 40 ]
x = 10

# Function call
result = binarySearch(arr, 0, len(arr)-1, x)

if result != -1:
    print "Element is present at index %d" % result
else:
    print "Element is not present in array"
```

Java

```
// Java implementation of recursive Binary Search
class BinarySearch
{
    // Returns index of x if it is present in arr[l..r], else
    // return -1
    int binarySearch(int arr[], int l, int r, int x)
    {
        if (r >= l)
        {
            int mid = l + (r - l)/2;

            // If the element is present at the middle itself
            if (arr[mid] == x)
                return mid;

            // If element is smaller than mid, then it can only
            // be present in left subarray
            if (arr[mid] > x)
                return binarySearch(arr, l, mid-1, x);

            // Else the element can only be present in right
            // subarray
        }
    }
}
```

```

        return binarySearch(arr, mid+1, r, x);
    }

    // We reach here when element is not present in array
    return -1;
}

// Driver method to test above
public static void main(String args[])
{
    BinarySearch ob = new BinarySearch();
    int arr[] = {2,3,4,10,40};
    int n = arr.length;
    int x = 10;
    int result = ob.binarySearch(arr,0,n-1,x);
    if (result == -1)
        System.out.println("Element not present");
    else
        System.out.println("Element found at index "+result);
}
}

/* This code is contributed by Rajat Mishra */

```

Output:

Element is present at index 3

Iterative implementation of Binary Search

C/C++

```

#include <stdio.h>

// A iterative binary search function. It returns location of x in
// given array arr[l..r] if present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r)
    {
        int m = l + (r-l)/2;

        // Check if x is present at mid
        if (arr[m] == x)
            return m;

        // If x is greater, ignore left half
        if (arr[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // If we reach here, then element was not present
    return -1;
}

int main(void)
{
    int arr[] = {2, 3, 4, 10, 40};
    int n = sizeof(arr)/ sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n-1, x);
    (result == -1)? printf("Element is not present in array")
                  : printf("Element is present at index %d", result);
    return 0;
}

```

Python

```

# Iterative Binary Search Function
# It returns location of x in given array arr if present,
# else returns -1
def binarySearch(arr, l, r, x):

    while l <= r:

        mid = l + (r - l)/2;

        # Check if x is present at mid
        if arr[mid] == x:
            return mid

        # If x is greater, ignore left half
        elif arr[mid] < x:
            l = mid + 1

        # If x is smaller, ignore right half
        else:
            r = mid - 1

    # If we reach here, then the element was not present
    return -1

# Test array
arr = [ 2, 3, 4, 10, 40 ]
x = 10

# Function call

```

```

result = binarySearch(arr, 0, len(arr)-1, x)

if result != -1:
    print "Element is present at index %d" % result
else:
    print "Element is not present in array"

```

Java

```

// Java implementation of iterative Binary Search
class BinarySearch
{
    // Returns index of x if it is present in arr[], else
    // return -1
    int binarySearch(int arr[], int x)
    {
        int l = 0, r = arr.length - 1;
        while (l <= r)
        {
            int m = l + (r-l)/2;

            // Check if x is present at mid
            if (arr[m] == x)
                return m;

            // If x greater, ignore left half
            if (arr[m] < x)
                l = m + 1;

            // If x is smaller, ignore right half
            else
                r = m - 1;
        }

        // if we reach here, then element was not present
        return -1;
    }

    // Driver method to test above
    public static void main(String args[])
    {
        BinarySearch ob = new BinarySearch();
        int arr[] = {2, 3, 4, 10, 40};
        int n = arr.length;
        int x = 10;
        int result = ob.binarySearch(arr, x);
        if (result == -1)
            System.out.println("Element not present");
        else
            System.out.println("Element found at index "+result);
    }
}

```

Output:

Element is present at index 3

Time Complexity:

The time complexity of Binary Search can be written as

$$T(n) = T(n/2) + c$$

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is $\Theta(\log n)$.

Auxiliary Space: $O(1)$ in case of iterative implementation. In case of recursive implementation, $O(\log n)$ recursion call stack space.

Algorithmic Paradigm: Divide and Conquer

Interesting articles based on Binary Search.

- [The Ubiquitous Binary Search](#)
- [Interpolation search vs Binary search](#)
- [Find the minimum element in a sorted and rotated array](#)
- [Find a peak element](#)
- [Find a Fixed Point in a given array](#)
- [Count the number of occurrences in a sorted array](#)
- [Median of two sorted arrays](#)
- [Floor and Ceiling in a sorted array](#)
- [Find the maximum element in an array which is first increasing and then decreasing](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Searching and Sorting

Jump Search

Like [Binary Search](#), Jump Search is a searching algorithm for sorted arrays. The basic idea is to check fewer elements (than [linear search](#)) by jumping ahead by fixed steps or skipping some elements in place of searching all elements.

For example, suppose we have an array arr[] of size n and block (to be jumped) size m. Then we search at the indexes arr[0], arr[m], arr[2m].....arr[km] and so on. Once we find the interval (arr[km] < x < arr[(k+1)m]), we perform a linear search operation from the index km to find the element x.

Let's consider the following array: (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610). Length of the array is 16. Jump search will find the value of 55 with the following steps assuming that the block size to be jumped is 4.

STEP 1: Jump from index 0 to index 4;

STEP 2: Jump from index 4 to index 8;

STEP 3: Jump from index 8 to index 16;

STEP 4: Since the element at index 16 is greater than 55 we will jump back a step to come to index 9.

STEP 5: Perform linear search from index 9 to get the element 55.

What is the optimal block size to be skipped?

In the worst case, we have to do n/m jumps and if the last checked value is greater than the element to be searched for, we perform $m-1$ comparisons more for linear search. Therefore the total number of comparisons in the worst case will be $((n/m) + m-1)$. The value of the function $((n/m) + m-1)$ will be minimum when $m = \sqrt{n}$. Therefore, the best step size is $m = \sqrt{n}$.

Java Implementation :

```
// Java program to implement Jump Search.
public class JumpSearch
{
    public static int jumpSearch(int[] arr, int x)
    {
        int n = arr.length;

        // Finding block size to be jumped
        int step = (int)Math.floor(Math.sqrt(n));

        // Finding the block where element is
        // present (if it is present)
        int prev = 0;
        while (arr[Math.min(step, n)-1] < x)
        {
            prev = step;
            step += (int)Math.floor(Math.sqrt(n));
            if (prev >= n)
                return -1;
        }

        // Doing a linear search for x in block
        // beginning with prev.
        while (arr[prev] < x)
        {
            prev++;
        }

        // If we reached next block or end of
        // array, element is not present.
        if (prev == Math.min(step, n))
            return -1;
    }

    // If element is found
    if (arr[prev] == x)
        return prev;

    return -1;
}

// Driver program to test function
public static void main(String [] args)
{
    int arr[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21,
            34, 55, 89, 144, 233, 377, 610};
    int x = 55;

    // Find the index of 'x' using Jump Search
    int index = jumpSearch(arr, x);

    // Print the index where 'x' is located
    System.out.println("Number " + x +
            " is at index " + index);
}
}
```

Output:

Number 55 is at index 10

Time Complexity : $O(\sqrt{n})$

Auxiliary Space : $O(1)$

Important points:

- Works only sorted arrays.
- The optimal size of a block to be jumped is $O(\sqrt{n})$. This makes the time complexity of Jump Search $O(\sqrt{n})$.
- The time complexity of Jump Search is between Linear Search ($O(n)$) and Binary Search ($O(\log n)$).
- Binary Search is better than Jump Search, but Jump search has an advantage that we traverse back only once (Binary Search may require up to $O(\log n)$ jumps, consider a situation where the element to be search is the smallest element or smaller than the smallest). So in a systems where jumping back is costly, we use Jump Search.

References:

https://en.wikipedia.org/wiki/Jump_search

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Searching

Interpolation Search

Given a sorted array of n uniformly distributed values $\text{arr}[]$, write a function to search for a particular element x in the array.

Linear Search finds the element in $O(n)$ time, **Jump Search** takes $O(\sqrt{n})$ time and **Binary Search** take $O(\log n)$ time.

The Interpolation Search is an improvement over **Binary Search** for instances, where the values in a sorted array are uniformly distributed. Binary Search always goes to middle element to check. On the other hand interpolation search may go to different locations according the value of key being searched. For example if the value of key is closer to the last element, interpolation search is likely to start search toward the end side.

To find the position to be searched, it uses following formula.

```
// The idea of formula is to return higher value of pos
// when element to be searched is closer to arr[hi]. And
// smaller value when closer to arr[lo]
pos = lo + [(x - arr[lo]) * (hi - lo) / (arr[hi] - arr[lo])]

arr[] ==> Array where elements need to be searched
x ==> Element to be searched
lo ==> Starting index in arr[]
hi ==> Ending index in arr[]
```

Algorithm

Rest of the Interpolation algorithm is same except the above partition logic.

Step1: In a loop, calculate the value of "pos" using the probe position formula.

Step2: If it is a match, return the index of the item, and exit.

Step3: If the item is less than $\text{arr}[\text{pos}]$, calculate the probe position of the left sub-array. Otherwise calculate the same in the right sub-array.

Step4: Repeat until a match is found or the sub-array reduces to zero.

Below is C implementation of algorithm.

```
// C program to implement interpolation search
#include<stdio.h>

// If x is present in arr[0..n-1], then returns
// index of it, else returns -1.
int interpolationSearch(int arr[], int n, int x)
{
    // Find indexes of two corners
    int lo = 0, hi = (n - 1);

    // Since array is sorted, an element present
    // in array must be in range defined by corner
    while (lo <= hi && x >= arr[lo] && x <= arr[hi])
    {
        // Probing the position with keeping
        // uniform distribution in mind.
        int pos = lo + (((double)(hi - lo) /
        (arr[hi] - arr[lo])) * (x - arr[lo]));

        // Condition of target found
        if (arr[pos] == x)
            return pos;

        // If x is larger, x is in upper part
        if (arr[pos] < x)
            lo = pos + 1;

        // If x is smaller, x is in lower part
        else
            hi = pos - 1;
    }
    return -1;
}

// Driver Code
int main()
{
    // Array of items on which search will
    // be conducted.
    int arr[] = {10, 12, 13, 16, 18, 19, 20, 21, 22, 23,
                24, 33, 35, 42, 47};
    int n = sizeof(arr)/sizeof(arr[0]);

    int x = 18; // Element to be searched
    int index = interpolationSearch(arr, n, x);

    // If element was found
    if (index != -1)
        printf("Element found at index %d", index);
    else
        printf("Element not found.");
    return 0;
}
```

Output :

Element found at index 4

Time Complexity : If elements are uniformly distributed, then $O(\log \log n)$. In worst case it can take upto $O(n)$.

Auxiliary Space : $O(1)$

This article is contributed by **Aayu sachdev**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Searching
Technical Scripter

Exponential Search

The name of this searching algorithm may be misleading as it works in $O(\log n)$ time. The name comes from the way it searches an element.

Given a sorted array an element x to be searched, find position of x in the array.

Input: arr[] = {10, 20, 40, 45, 55}
x = 45

Output: Element found at index 3

Input: arr[] = {10, 15, 25, 45, 55}
x = 15

Output: Element found at index 1

We have discussed, [linear search](#), [binary search](#) for this problem.

Exponential search involves two steps:

1. Find range where element is present
2. Do Binary Search in above found range.

How to find the range where element may be present?

The idea is to start with subarray size 1 compare its last element with x, then try size 2, then 4 and so on until last element of a subarray is not greater.

Once we find an index i (after repeated doubling of i), we know that the element must be present between $i/2$ and i (Why $i/2$? because we could not find a greater value in previous iteration)

Below is C++ implementation of above steps.

```
// C++ program to find an element x in a
// sorted array using Exponential search.
#include <bits/stdc++.h>
using namespace std;

int binarySearch(int arr[], int l, int r, int x);

// Returns position of first occurrence of
// x in array
int exponentialSearch(int arr[], int n, int x)
{
    // If x is present at first location itself
    if (arr[0] == x)
        return 0;

    // Find range for binary search by
    // repeated doubling
    int i = 1;
    while (i < n && arr[i] <= x)
        i = i*2;

    // Call binary search for the found range.
    return binarySearch(arr, i/2, min(i, n), x);
}

// A recursive binary search function. It returns
// location of x in given array arr[l..r] is
// present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l)
    {
        int mid = l + (r - l)/2;

        // If the element is present at the middle
        // itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then it
        // can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid-1, x);

        // Else the element can only be present
        // in right subarray
        return binarySearch(arr, mid+1, r, x);
    }

    // We reach here when element is not present
    // in array
    return -1;
}

// Driver code
int main(void)
```

```

{
    int arr[] = {2, 3, 4, 10, 40};
    int n = sizeof(arr)/ sizeof(arr[0]);
    int x = 10;
    int result = exponentialSearch(arr, n, x);
    (result == -1)? printf("Element is not present in array")
        : printf("Element is present at index %d",
            result);
    return 0;
}

```

Output :

Element is present at index 3

Time Complexity : $O(\log n)$

Auxiliary Space : The above implementation of Binary Search is recursive and requires $O(\log n)$ space. With iterative Binary Search, we need only $O(1)$ space.

Applications of Exponential Search:

1. Exponential Binary Search is particularly useful for unbounded searches, where size of array is infinite. Please refer [Unbounded Binary Search](#) for an example.
2. It works better than Binary Search for bounded arrays also when the element to be searched is closer to the first element.

Reference:

https://en.wikipedia.org/wiki/Exponential_search

This article is contributed by **Pankaj Sharma**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Searching
Binary-Search

Why is Binary Search preferred over Ternary Search?

The following is a simple recursive **Binary Search** function in C++ taken from [here](#).

```

// A recursive binary search function. It returns location of x in
// given array arr[l..r] is present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l)
    {
        int mid = l + (r - l)/2;

        // If the element is present at the middle itself
        if (arr[mid] == x) return mid;

        // If element is smaller than mid, then it can only be present
        // in left subarray
        if (arr[mid] > x) return binarySearch(arr, l, mid-1, x);

        // Else the element can only be present in right subarray
        return binarySearch(arr, mid+1, r, x);
    }

    // We reach here when element is not present in array
    return -1;
}

```

The following is a simple recursive **Ternary Search** function in C++.

```

// A recursive ternary search function. It returns location of x in
// given array arr[l..r] is present, otherwise -1
int ternarySearch(int arr[], int l, int r, int x)
{
    if (r >= l)
    {
        int mid1 = l + (r - l)/3;
        int mid2 = mid1 + (r - l)/3;

        // If x is present at the mid1
        if (arr[mid1] == x) return mid1;

        // If x is present at the mid2
        if (arr[mid2] == x) return mid2;

        // If x is present in left one-third
        if (arr[mid1] > x) return ternarySearch(arr, l, mid1-1, x);

        // If x is present in right one-third
        if (arr[mid2] < x) return ternarySearch(arr, mid2+1, r, x);

        // If x is present in middle one-third
        return ternarySearch(arr, mid1+1, mid2-1, x);
    }

    // We reach here when element is not present in array
    return -1;
}

```

Which of the above two does less comparisons in worst case?

From the first look, it seems the ternary search does less number of comparisons as it makes $\log_3 n$ recursive calls, but binary search makes $\log_2 n$ recursive calls. Let us take a closer look.

The following is recursive formula for counting comparisons in worst case of Binary Search.

$$T(n) = T(n/2) + 2, \quad T(1) = 1$$

The following is recursive formula for counting comparisons in worst case of Ternary Search.

$T(n) = T(n/3) + 4$, $T(1) = 1$

In binary search, there are $2\log_2 n + 1$ comparisons in worst case. In ternary search, there are $4\log_3 n + 1$ comparisons in worst case.

Time Complexity for Binary search = $2c\log_2 n + O(1)$
Time Complexity for Ternary search = $4c\log_3 n + O(1)$

Therefore, the comparison of Ternary and Binary Searches boils down the comparison of expressions $2\log_3 n$ and $\log_2 n$. The value of $2\log_3 n$ can be written as $(2 / \log_2 3) * \log_2 n$. Since the value of $(2 / \log_2 3)$ is more than one, Ternary Search does more comparisons than Binary Search in worst case.

Exercise:

Why Merge Sort divides input array in two halves, why not in three or more parts?

This article is contributed by **Anmol**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Searching

Selection Sort

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

- 1) The subarray which is already sorted.
- 2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Following example explains the above steps:

```
arr[] = 64 25 12 22 11  
  
// Find the minimum element in arr[0...4]  
// and place it at beginning  
11 25 12 22 64  
  
// Find the minimum element in arr[1...4]  
// and place it at beginning of arr[1...4]  
11 12 25 22 64  
  
// Find the minimum element in arr[2...4]  
// and place it at beginning of arr[2...4]  
11 12 22 25 64  
  
// Find the minimum element in arr[3...4]  
// and place it at beginning of arr[3...4]  
11 12 22 25 64
```

We strongly recommend that you click here and practice it, before moving on to the solution.

C/C++

```
// C program for implementation of selection sort  
#include <stdio.h>  
  
void swap(int *xp, int *yp)  
{  
    int temp = *xp;  
    *xp = *yp;  
    *yp = temp;  
}  
  
void selectionSort(int arr[], int n)  
{  
    int i, j, min_idx;  
  
    // One by one move boundary of unsorted subarray  
    for (i = 0; i < n-1; i++)  
    {  
        // Find the minimum element in unsorted array  
        min_idx = i;  
        for (j = i+1; j < n; j++)  
            if (arr[j] < arr[min_idx])  
                min_idx = j;  
  
        // Swap the found minimum element with the first element  
        swap(&arr[min_idx], &arr[i]);  
    }  
  
    /* Function to print an array */  
    void printArray(int arr[], int size)  
{  
        int i;  
        for (i=0; i < size; i++)  
            printf("%d ", arr[i]);  
        printf("\n");  
    }  
  
    // Driver program to test above functions  
int main()  
{  
    int arr[] = {64, 25, 12, 22, 11};  
    int n = sizeof(arr)/sizeof(arr[0]);  
    selectionSort(arr, n);  
    printf("Sorted array: \n");  
    printArray(arr, n);  
    return 0;  
}
```

```
}
```

Python

```
# Python program for implementation of Selection
# Sort
import sys
A = [64, 25, 12, 22, 11]

# Traverse through all array elements
for i in range(len(A)):

    # Find the minimum element in remaining
    # unsorted array
    min_idx = i
    for j in range(i+1, len(A)):
        if A[min_idx] > A[j]:
            min_idx = j

    # Swap the found minimum element with
    # the first element
    A[i], A[min_idx] = A[min_idx], A[i]

# Driver code to test above
print ("Sorted array")
for i in range(len(A)):
    print("%d" %A[i]),
```

Java

```
// Java program for implementation of Selection Sort
class SelectionSort
{
    void sort(int arr[])
    {
        int n = arr.length;

        // One by one move boundary of unsorted subarray
        for (int i = 0; i < n-1; i++)
        {
            // Find the minimum element in unsorted array
            int min_idx = i;
            for (int j = i+1; j < n; j++)
                if (arr[j] < arr[min_idx])
                    min_idx = j;

            // Swap the found minimum element with the first
            // element
            int temp = arr[min_idx];
            arr[min_idx] = arr[i];
            arr[i] = temp;
        }
    }

    // Prints the array
    void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i]+" ");
        System.out.println();
    }
}

// Driver code to test above
public static void main(String args[])
{
    SelectionSort ob = new SelectionSort();
    int arr[] = {64,25,12,22,11};
    ob.sort(arr);
    System.out.println("Sorted array");
    ob.printArray(arr);
}
/* This code is contributed by Rajat Mishra*/
```

Output:

```
Sorted array:
11 12 22 25 64
```

Time Complexity: $O(n^2)$ as there are two nested loops.

Auxiliary Space: $O(1)$

The good thing about selection sort is it never makes more than $O(n)$ swaps and can be useful when memory write is a costly operation.

Snapshots:

scene00721



scene00937



scene01153



scene01441



scene01729



scene02089

**Quiz on Selection Sort****Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz:**

- Bubble Sort
- Insertion Sort
- Merge Sort
- Heap Sort
- QuickSort
- Radix Sort
- Counting Sort
- Bucket Sort
- ShellSort

Coding practice for sorting.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Searching and Sorting

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Example:

First Pass:

(5 1 4 2 8) \rightarrow (1 5 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 5 4 2 8) \rightarrow (1 4 5 2 8), Swap since 5 > 4
(1 4 5 2 8) \rightarrow (1 4 2 5 8), Swap since 5 > 2
(1 4 2 5 8) \rightarrow (1 4 2 5 8), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(1 4 2 5 8) \rightarrow (1 4 2 5 8)
(1 4 2 5 8) \rightarrow (1 2 4 5 8), Swap since 4 > 2
(1 2 4 5 8) \rightarrow (1 2 4 5 8)
(1 2 4 5 8) \rightarrow (1 2 4 5 8)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(1 2 4 5 8) \rightarrow (1 2 4 5 8)
(1 2 4 5 8) \rightarrow (1 2 4 5 8)
(1 2 4 5 8) \rightarrow (1 2 4 5 8)
(1 2 4 5 8) \rightarrow (1 2 4 5 8)

Following are C/C++, Python and Java implementations of Bubble Sort.

C/C++

```
// C program for implementation of Bubble sort
#include <stdio.h>

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i++)
        // Last i elements are already in place
        for (j = 0; j < n-i-1; j++)
            if (arr[j] > arr[j+1])
                swap(&arr[j], &arr[j+1]);
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    for (int i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program to test above functions
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Java

```
// Java program for implementation of Bubble Sort
class BubbleSort
{
    void bubbleSort(int arr[])
    {
        int n = arr.length;
        for (int i = 0; i < n-1; i++)
            for (int j = 0; j < n-i-1; j++)
                if (arr[j] > arr[j+1])
                {
                    // swap temp and arr[i]
                    int temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
    }
}
```

```

/* Prints the array */
void printArray(int arr[])
{
    int n = arr.length;
    for (int i=0; i<n; i++)
        System.out.print(arr[i] + " ");
    System.out.println();
}

// Driver method to test above
public static void main(String args[])
{
    BubbleSort ob = new BubbleSort();
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    ob.bubbleSort(arr);
    System.out.println("Sorted array");
    ob.printArray(arr);
}
/* This code is contributed by Rajat Mishra */

```

Python

```

# Python program for implementation of Bubble Sort

def bubbleSort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):

        # Last i elements are already in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

    # Driver code to test above
    arr = [64, 34, 25, 12, 22, 11, 90]

    bubbleSort(arr)

    print ("Sorted array is:")
    for i in range(len(arr)):
        print ("%d" %arr[i]),

```

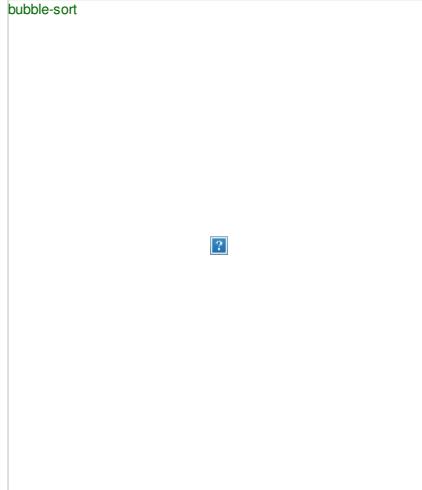
Output:

```

Sorted array:
11 12 22 25 34 64 90

```

Illustration :



Optimized Implementation:

The above function always runs $O(n^2)$ time even if the array is sorted. It can be optimized by stopping the algorithm if inner loop didn't cause any swap.

```

// Optimized implementation of Bubble sort
#include <stdio.h>

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// An optimized version of Bubble Sort
void bubbleSort(int arr[], int n)
{
    int i, j;
    bool swapped;
    for (i = 0; i < n-1; i++)
    {
        swapped = false;

```

```

for (j = 0; j < n-i-1; j++)
{
    if (arr[j] > arr[j+1])
    {
        swap(&arr[j], &arr[j+1]);
        swapped = true;
    }
}

// IF no two elements were swapped by inner loop, then break
if (swapped == false)
    break;
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program to test above functions
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}

```

Output:

```

Sorted array:
11 12 22 25 34 64 90

```

Worst and Average Case Time Complexity: $O(n^2)$. Worst case occurs when array is reverse sorted.

Best Case Time Complexity: $O(n)$. Best case occurs when array is already sorted.

Auxiliary Space: $O(1)$

Boundary Cases: Bubble sort takes minimum time (Order of n) when elements are already sorted.

Sorting In Place: Yes

Stable: Yes

Due to its simplicity, bubble sort is often used to introduce the concept of a sorting algorithm.

In computer graphics it is popular for its capability to detect a very small error (like swap of just two elements) in almost-sorted arrays and fix it with just linear complexity ($2n$). For example, it is used in a polygon filling algorithm, where bounding lines are sorted by their x coordinate at a specific scan line (a line parallel to x axis) and with incrementing y their order changes (two elements are swapped) only at intersections of two lines (Source: [Wikipedia](#))

Snapshots:

scene00361



scene00433





Quiz on Bubble Sort

Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz:

- Selection Sort
- Insertion Sort
- Merge Sort
- Heap Sort
- QuickSort
- Radix Sort
- Counting Sort
- Bucket Sort
- ShellSort

Recursive Bubble Sort

Coding practice for sorting.

Reference:

- Wikipedia – Bubble Sort
- Image Source

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Searching and Sorting

Insertion Sort

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.



Algorithm

```
// Sort an arr[] of size n  
insertionSort(arr, n)  
Loop from i = 1 to n-1.
```

.....a) Pick element arr[i] and insert it into sorted sequence arr[0..i-1]

Example:

insertion-sort



Another Example:

12, 11, 13, 5, 6

Let us loop for i = 1 (second element of the array) to 5 (Size of input array)

i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12

11, 12, 13, 5, 6

i = 2. 13 will remain at its position as all elements in A[0..i-1] are smaller than 13

11, 12, 13, 5, 6

i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.

5, 11, 12, 13, 6

i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.

5, 6, 11, 12, 13

We strongly recommend that you click here and practice it, before moving on to the solution.

C/C++

```
// C program for insertion sort
#include <stdio.h>
#include <math.h>

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i-1;

        /* Move elements of arr[0..i-1], that are
        greater than key, to one position ahead
        of their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j+1] = arr[j];
            j = j-1;
        }
        arr[j+1] = key;
    }
}

// A utility function to print an array of size n
void printArray(int arr[], int n)
{
    int i;
    for (i=0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

/* Driver program to test insertion sort */
int main()
{
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr)/sizeof(arr[0]);

    insertionSort(arr, n);
    printArray(arr, n);

    return 0;
}
```

Python

```
# Python program for implementation of Insertion Sort

# Function to do insertion sort
def insertionSort(arr):
```

```

# Traverse through 1 to len(arr)
for i in range(1, len(arr)):

    key = arr[i]

    # Move elements of arr[0..i-1], that are
    # greater than key, to one position ahead
    # of their current position
    j = i-1
    while j >=0 and key < arr[j] :
        arr[j+1] = arr[j]
        j -= 1
    arr[j+1] = key

# Driver code to test above
arr = [12, 11, 13, 5, 6]
insertionSort(arr)
print ("Sorted array is:")
for i in range(len(arr)):
    print ("%d" %arr[i])

# This code is contributed by Mohit Kumra

```

Java

```

// Java program for implementation of Insertion Sort
class InsertionSort
{
    /*Function to sort array using insertion sort*/
    void sort(int arr[])
    {
        int n = arr.length;
        for (int i=1; i<n; i++)
        {
            int key = arr[i];
            int j = i-1;

            /* Move elements of arr[0..i-1], that are
               greater than key, to one position ahead
               of their current position */
            while (j>=0 && arr[j] > key)
            {
                arr[j+1] = arr[j];
                j = j-1;
            }
            arr[j+1] = key;
        }
    }

    /* A utility function to print array of size n*/
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; i++)
            System.out.print(arr[i] + " ");

        System.out.println();
    }

    // Driver method
    public static void main(String args[])
    {
        int arr[] = {12, 11, 13, 5, 6};

        InsertionSort ob = new InsertionSort();
        ob.sort(arr);

        printArray(arr);
    }
} /* This code is contributed by Rajat Mishra. */

```

Output:

```
5 6 11 12 13
```

Time Complexity: $O(n^2)$

Auxiliary Space: $O(1)$

Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

Algorithmic Paradigm: Incremental Approach

Sorting In Place: Yes

Stable: Yes

Online: Yes

Uses: Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

What is Binary Insertion Sort?

We can use binary search to reduce the number of comparisons in normal insertion sort. Binary Insertion Sort find use binary search to find the proper location to insert the selected item at each iteration. In normal insertion, sort it takes $O(i)$ (at i th iteration) in worst case. we can reduce it to $O(\log i)$ by using binary search. The algorithm as a whole still has a running worst case running time of $O(n^2)$ because of the series of swaps required for each insertion. Refer [this](#) for implementation.

How to implement Insertion Sort for Linked List?

Below is simple insertion sort algorithm for linked list.

- 1) Create an empty sorted (or result) list
- 2) Traverse the given list, do following for every node.
-a) Insert current node in sorted way in sorted or result list.

3) Change head of given linked list to head of sorted (or result) list.

Refer [this](#) for implementation.

Snapshots:

scene00793



scene01009



scene01225



scene01729



scene01801



scene01873



Quiz on Insertion Sort

Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz

Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Heap Sort, QuickSort, Radix Sort, Counting Sort, Bucket Sort, ShellSort, Comb Sort,

Coding practice for sorting.

Image Source: http://www.just.edu.jo/~basel/algorithms/Algo%20Slides/algo_ch2_getting_started.pdf

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Searching and Sorting

Merge Sort

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

```
MergeSort(arr[], l, r)
If r > l
    1. Find the middle point to divide the array into two halves:
        middle m = (l+r)/2
    2. Call mergeSort for first half:
        Call mergeSort(arr, l, m)
    3. Call mergeSort for second half:
        Call mergeSort(arr, m+1, r)
    4. Merge the two halves sorted in step 2 and 3:
        Call merge(arr, l, m, r)
```

The following diagram from wikipedia shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



We strongly recommend that you click here and practice it, before moving on to the solution.

C/C++

```
/* C program for Merge Sort */
#include<stdlib.h>
#include<stdio.h>

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    /* Create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[]
     * Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    /* Merge the temp arrays back into arr[l..r]
     * Initial index of first subarray
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copy the remaining elements of L[], if there
     * are any */
    while (i < n1)
    {
```

```

        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy the remaining elements of R[], if there
     * are any */
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

/* l is for left index and r is right index of the
 * sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l+(r-l)/2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}

/* UTILITY FUNCTIONS */
/* Function to print an array */
void printArray(int A[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr)/sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    printf("\nSorted array is \n");
    printArray(arr, arr_size);
    return 0;
}

```

Java

```

/* Java program for Merge Sort */
class MergeSort
{
    // Merges two subarrays of arr[].
    // First subarray is arr[l..m]
    // Second subarray is arr[m+1..r]
    void merge(int arr[], int l, int m, int r)
    {
        // Find sizes of two subarrays to be merged
        int n1 = m - l + 1;
        int n2 = r - m;

        /* Create temp arrays */
        int L[] = new int [n1];
        int R[] = new int [n2];

        /*Copy data to temp arrays*/
        for (int i=0; i<n1; ++i)
            L[i] = arr[l + i];
        for (int j=0; j<n2; ++j)
            R[j] = arr[m + 1 + j];

        /* Merge the temp arrays */

        // Initial indexes of first and second subarrays
        int i = 0, j = 0;

        // Initial index of merged subarry array
        int k = l;
        while (i < n1 && j < n2)
        {
            if (L[i] <= R[j])
            {
                arr[k] = L[i];
                i++;
            }
            else
            {
                arr[k] = R[j];
                j++;
            }
            k++;
        }
    }
}

```

```

}

/* Copy remaining elements of L[] if any */
while (i < n1)
{
    arr[k] = L[i];
    i++;
    k++;
}

/* Copy remaining elements of L[] if any */
while (j < n2)
{
    arr[k] = R[j];
    j++;
    k++;
}

// Main function that sorts arr[l..r] using
// merge()
void sort(int arr[], int l, int r)
{
    if (l < r)
    {
        // Find the middle point
        int m = (l+r)/2;

        // Sort first and second halves
        sort(arr, l, m);
        sort(arr, m+1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}

/* A utility function to print array of size n */
static void printArray(int arr[])
{
    int n = arr.length;
    for (int i=0; i<n; ++i)
        System.out.print(arr[i] + " ");
    System.out.println();
}

// Driver method
public static void main(String args[])
{
    int arr[] = {12, 11, 13, 5, 6, 7};

    System.out.println("Given Array");
    printArray(arr);

    MergeSort ob = new MergeSort();
    ob.sort(arr, 0, arr.length-1);

    System.out.println("\nSorted array");
    printArray(arr);
}
*/
/* This code is contributed by Rajat Mishra */

```

Python

```

# Python program for implementation of MergeSort

# Merges two subarrays of arr[].
# First subarray is arr[l..m]
# Second subarray is arr[m+1..r]
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m

    # create temp arrays
    L = [0] * (n1)
    R = [0] * (n2)

    # Copy data to temp arrays L[] and R[]
    for i in range(0 ,n1):
        L[i] = arr[l + i]

    for j in range(0 ,n2):
        R[j] = arr[m + 1 + j]

    # Merge the temp arrays back into arr[l..r]
    i = 0      # Initial index of first subarray
    j = 0      # Initial index of second subarray
    k = l      # Initial index of merged subarray

    while i < n1 and j < n2 :
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    # Copy the remaining elements of L[], if there
    # are any
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1

    # Copy the remaining elements of R[], if there
    # are any
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

```

```

k += 1

# Copy the remaining elements of R[], if there
# are any
while j < n2:
    arr[k] = R[j]
    j += 1
    k += 1

# l is for left index and r is right index of the
# sub-array of arr to be sorted
def mergeSort(arr,l,r):
    if l < r:

        # Same as (l+r)/2, but avoids overflow for
        # large l and h
        m = (l+(r-1))/2

        # Sort first and second halves
        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)

# Driver code to test above
arr = [12, 11, 13, 5, 6, 7]
n = len(arr)
print ("Given array is")
for i in range(n):
    print ("%d" %arr[i]),

mergeSort(arr,0,n-1)
print ("\n\nSorted array is")
for i in range(n):
    print ("%d" %arr[i]),

# This code is contributed by Mohit Kumra

```

Output:

```

Given array is
12 11 13 5 6 7

Sorted array is
5 6 7 11 12 13

```

Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$$T(n) = 2T(n/2) + \boxed{\text{?}}$$

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is $\boxed{\text{?}}$.

Time complexity of Merge Sort is $\boxed{\text{?}}$ in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.

Auxiliary Space: $O(n)$

Algorithmic Paradigm: Divide and Conquer

Sorting In Place: No in a typical implementation

Stable: Yes

Applications of Merge Sort

1. Merge Sort is useful for sorting linked lists in $O(n\log n)$ time. In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in $O(1)$ extra space and $O(1)$ time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.
In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of $A[0]$ be x then to access $A[i]$, we can directly access the memory at $(x + i * 4)$. Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i 'th index, we have to travel each and every node from the head to i 'th node as we don't have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.
2. Inversion Count Problem
3. Used in External Sorting

Snapshots:



Quiz on Merge Sort

Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz:

Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Heap Sort, QuickSort, Radix Sort, Counting Sort, Bucket Sort, ShellSort, Comb Sort

Coding practice for sorting.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Searching and Sorting

Heap Sort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining elements.

What is Binary Heap?

Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible (Source Wikipedia)

A [Binary Heap](#) is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater (or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

Why array based representation for Binary Heap?

Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index i , the left child can be calculated by $2 * i + 1$ and right child by $2 * i + 2$ (assuming the indexing starts at 0).

Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

How to build the heap?

Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.

Lets understand with the help of an example:

```
Input data: 4, 10, 3, 5, 1
        4(0)
       / \
      10(1) 3(2)
     / \
    5(3) 1(4)
```

The numbers in bracket represent the indices in the array representation of data.

Applying heapify procedure to index 1:

```
        4(0)
       / \
      10(1) 3(2)
     / \
    5(3) 1(4)
```

Applying heapify procedure to index 0:

```
        10(0)
       / \
      5(1) 3(2)
     / \
    4(3) 1(4)
```

The heapify procedure calls itself recursively to build heap in top down manner.

C++

```
// C++ program for implementation of Heap Sort
#include <iostream>
using namespace std;
```

```
// To heapify a subtree rooted with node i which is
// an index in arr[], n is size of heap
void heapify(int arr[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int l = 2*i + 1; // left = 2*i + 1
    int r = 2*i + 2; // right = 2*i + 2
```

```
    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;
```

```
    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;
```

```
    // If largest is not root
    if (largest != i)
    {
        swap(arr[i], arr[largest]);
        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}
```

```
// main function to do heap sort
void heapSort(int arr[], int n)
{
```

```
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
```

```
    // One by one extract an element from heap
    for (int i=n-1; i>=0; i--)
    {
        // Move current root to end
        swap(arr[0], arr[i]);
```

```
        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

```
/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
```

```
    for (int i=0; i<n; ++i)
        cout << arr[i] << " ";
    cout << "\n";
```

```
// Driver program
```

```
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);
```

```
    heapSort(arr, n);
```

```
    cout << "Sorted array is \n";
    printArray(arr, n);
}
```

```
}
```

Java

```
// Java program for implementation of Heap Sort
public class HeapSort
{
    public void sort(int arr[])
    {
        int n = arr.length;

        // Build heap (rearrange array)
        for (int i = n / 2 - 1; i >= 0; i--)
            heapify(arr, n, i);

        // One by one extract an element from heap
        for (int i=n-1; i>=0; i--)
        {
            // Move current root to end
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            // call max heapify on the reduced heap
            heapify(arr, i, 0);
        }

        // To heapify a subtree rooted with node i which is
        // an index in arr[], n is size of heap
        void heapify(int arr[], int n, int i)
        {
            int largest = i; // Initialize largest as root
            int l = 2*i + 1; // left = 2*i + 1
            int r = 2*i + 2; // right = 2*i + 2

            // If left child is larger than root
            if (l < n && arr[l] > arr[largest])
                largest = l;

            // If right child is larger than largest so far
            if (r < n && arr[r] > arr[largest])
                largest = r;

            // If largest is not root
            if (largest != i)
            {
                int swap = arr[i];
                arr[i] = arr[largest];
                arr[largest] = swap;

                // Recursively heapify the affected sub-tree
                heapify(arr, n, largest);
            }
        }

        /* A utility function to print array of size n */
        static void printArray(int arr[])
        {
            int n = arr.length;
            for (int i=0; i<n; i++)
                System.out.print(arr[i]+ " ");
            System.out.println();
        }

        // Driver program
        public static void main(String args[])
        {
            int arr[] = {12, 11, 13, 5, 6, 7};
            int n = arr.length;

            HeapSort ob = new HeapSort();
            ob.sort(arr);

            System.out.println("Sorted array is");
            printArray(arr);
        }
    }
}
```

Python

```
# Python program for implementation of heap Sort

# To heapify subtree rooted at index i.
# n is size of heap
def heapify(arr, n, i):
    largest = i # Initialize largest as root
    l = 2 * i + 1 # left = 2*i + 1
    r = 2 * i + 2 # right = 2*i + 2

    # See if left child of root exists and is
    # greater than root
    if l < n and arr[l] > arr[largest]:
        largest = l

    # See if right child of root exists and is
    # greater than root
    if r < n and arr[largest] < arr[r]:
        largest = r

    # Change root, if needed
    if largest != i:
```

```

arr[i],arr[largest] = arr[largest],arr[i] # swap

# Heapify the root.
heapify(arr, n, largest)

# The main function to sort an array of given size
def heapSort(arr):
    n = len(arr)

    # Build a maxheap.
    for i in range(n, -1, -1):
        heapify(arr, n, i)

    # One by one extract elements
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i] # swap
        heapify(arr, i, 0)

# Driver code to test above
arr = [ 12, 11, 13, 5, 6, 7]
heapSort(arr)
n = len(arr)
print ("Sorted array is")
for i in range(n):
    print ("%d" %arr[i]),
# This code is contributed by Mohit Kumra

```

Output:

```

Sorted array is
5 6 7 11 12 13

```

[Here](#) is previous C code for reference.

Notes:

Heap sort is an in-place algorithm.

Its typical implementation is not stable, but can be made stable (See [this](#))

Time Complexity: Time complexity of heapify is $O(\log n)$. Time complexity of createAndBuildHeap() is $O(n)$ and overall time complexity of Heap Sort is $O(n\log n)$.

Applications of HeapSort

1. Sort a nearly sorted (or K sorted) array
2. k largest(or smallest) elements in an array

Heap sort algorithm has limited uses because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used. See [Applications of Heap Data Structure](#)

Snapshots:

scene00505



scene00793



scene01081





Quiz on Heap Sort

Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz:

QuickSort, Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Heap Sort, QuickSort, Radix Sort, Counting Sort, Bucket Sort, ShellSort, Comb Sort, Pigeonhole Sort

Coding practice for sorting.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Searching and Sorting

QuickSort

Like [Merge Sort](#), QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Pseudo Code for recursive QuickSort function :

```
/* low --> Starting index, high --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
        quicksort
```



Partition Algorithm

There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i.

```
/* low --> Starting index, high --> Ending index */
quickSort(arr[], low, high)
{
```

```
if (low
```

Pseudo code for partition()

```
/* This function takes last element as pivot, places  
the pivot element at its correct position in sorted  
array, and places all smaller (smaller than pivot)  
to left of pivot and all greater elements to right  
of pivot */  
partition (arr[], low, high)  
{  
    // pivot (Element to be placed at right position)  
    pivot = arr[high];  
  
    i = (low - 1) // Index of smaller element  
  
    for (j = low; j <= high - 1; j++)  
        if (arr[j] <= pivot)  
            swap(arr[i], arr[j]);  
  
    swap(arr[i + 1], arr[high]);  
    return i + 1;  
}
```

Illustration of partition() :

```
arr[] = {10, 80, 30, 90, 40, 50, 70};  
Indexes: 0 1 2 3 4 5 6
```

```
low = 0, high = 6, pivot = arr[high] = 70  
Initialize index of smaller element, i = -1
```

```
Traverse elements from j = low to high-1  
j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])  
i = 0  
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j  
// are same
```

```
j = 1 : Since arr[j] > pivot, do nothing  
// No change in i and arr[]
```

```
j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])  
i = 1  
arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30
```

```
j = 3 : Since arr[j] > pivot, do nothing  
// No change in i and arr[]
```

```
j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])  
i = 2  
arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped  
j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]  
i = 3  
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped
```

```
We come out of loop because j is now equal to high-1.  
Finally we place pivot at correct position by swapping  
arr[i+1] and arr[high] (or pivot)  
arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped
```

```
Now 70 is at its correct place. All elements smaller than  
70 are before it and all elements greater than 70 are after  
it.
```

We strongly recommend that you click here and practice it, before moving on to the solution.

Implementation:

Following are C++, Java and Python implementations of QuickSort.

C/C++

```
/* C implementation QuickSort */  
#include<stdio.h>  
  
// A utility function to swap two elements  
void swap(int* a, int* b)  
{  
    int t = *a;  
    *a = *b;  
    *b = t;  
}  
  
/* This function takes last element as pivot, places  
the pivot element at its correct position in sorted  
array, and places all smaller (smaller than pivot)  
to left of pivot and all greater elements to right  
of pivot */  
int partition (int arr[], int low, int high)  
{  
    int pivot = arr[high]; // pivot  
    int i = (low - 1); // Index of smaller element  
  
    for (int j = low; j <= high- 1; j++)  
    {  
        // If current element is smaller than or  
        // equal to pivot  
        if (arr[j] <= pivot)  
        {  
            i++; // increment index of smaller element  
            swap(&arr[i], &arr[j]);  
        }  
    }  
    swap(&arr[i + 1], &arr[high]);  
    return i + 1;  
}
```

```

        swap(&arr[i], &arr[j]);
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
         * at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program to test above functions
int main()
{
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    quickSort(arr, 0, n-1);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}

```

Java

```

// Java program for implementation of QuickSort
class QuickSort
{
    /* This function takes last element as pivot,
     places the pivot element at its correct
     position in sorted array, and places all
     smaller (smaller than pivot) to left of
     pivot and all greater elements to right
     of pivot */
    int partition(int arr[], int low, int high)
    {
        int pivot = arr[high];
        int i = (low-1); // index of smaller element
        for (int j=low; j<=high-1; j++)
        {
            // If current element is smaller than or
            // equal to pivot
            if (arr[j] <= pivot)
            {
                i++;
                // swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        // swap arr[i+1] and arr[high] (or pivot)
        int temp = arr[i+1];
        arr[i+1] = arr[high];
        arr[high] = temp;

        return i+1;
    }

    /* The main function that implements QuickSort()
    arr[] --> Array to be sorted,
    low --> Starting index,
    high --> Ending index */
    void sort(int arr[], int low, int high)
    {
        if (low < high)
        {
            /* pi is partitioning index, arr[pi] is
             * now at right place */
            int pi = partition(arr, low, high);

            // Recursively sort elements before
            // partition and after partition

```

```

        sort(arr, low, pi-1);
        sort(arr, pi+1, high);
    }

/* A utility function to print array of size n */
static void printArray(int arr[])
{
    int n = arr.length;
    for (int i=0; i<n; ++i)
        System.out.print(arr[i]+" ");
    System.out.println();
}

// Driver program
public static void main(String args[])
{
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = arr.length;

    QuickSort ob = new QuickSort();
    ob.sort(arr, 0, n-1);

    System.out.println("sorted array");
    printArray(arr);
}
}

/*This code is contributed by Rajat Mishra */

```

Python

```

# Python program for implementation of Quicksort Sort

# This function takes last element as pivot, places
# the pivot element at its correct position in sorted
# array, and places all smaller (smaller than pivot)
# to left of pivot and all greater elements to right
# of pivot
def partition(arr,low,high):
    i = ( low-1 )      # index of smaller element
    pivot = arr[high]   # pivot

    for j in range(low , high):

        # If current element is smaller than or
        # equal to pivot
        if arr[j] <= pivot:

            # increment index of smaller element
            i = i+1
            arr[i],arr[j] = arr[j],arr[i]

    arr[i+1],arr[high] = arr[high],arr[i+1]
    return ( i+1 )

# The main function that implements QuickSort
# arr[] --> Array to be sorted,
# low --> Starting index,
# high --> Ending index

# Function to do Quick sort
def quickSort(arr,low,high):
    if low < high:

        # pi is partitioning index, arr[p] is now
        # at right place
        pi = partition(arr,low,high)

        # Separately sort elements before
        # partition and after partition
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)

# Driver code to test above
arr = [10, 7, 8, 9, 1, 5]
n = len(arr)
quickSort(arr,0,n-1)
print ("Sorted array is:")
for i in range(n):
    print ("%d" %arr[i]),

# This code is contributed by Mohit Kumra

```

Output:

```

Sorted array:
1 5 7 8 9 10

```

Analysis of QuickSort

Time taken by QuickSort in general can be written as following.

$$T(n) = T(k) + T(n-k-1) + \dots + T(1)$$

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.

The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case recurrence relation is

$$T(n) = T(0) + T(n-1) + \Theta(n)$$

which is equivalent to

$$T(n) = T(n-1) + \Theta(n)$$

The solution of above recurrence is $\Theta(n^2)$.

Best Case: The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

$$T(n) = 2T(n/2) + \Theta(n)$$

The solution of above recurrence is $\Theta(n\log n)$. It can be solved using case 2 of [Master Theorem](#).

Average Case:

To do average case analysis, we need to [consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy](#).

We can get an idea of average case by considering the case when partition puts $O(n/9)$ elements in one set and $O(9n/10)$ elements in other set. Following is recurrence for this case.

$$T(n) = T(n/9) + T(9n/10) + \Theta(n)$$

Solution of above recurrence is also $O(n\log n)$

Although the worst case time complexity of QuickSort is $O(n^2)$ which is more than many other sorting algorithms like [Merge Sort](#) and [Heap Sort](#), QuickSort is faster in practice, because its inner loop can be efficient.

What is 3-Way QuickSort?

In simple QuickSort algorithm, we select an element as pivot, partition the array around pivot and recur for subarrays on left and right of pivot.

Consider an array which has many redundant elements. For example, {1, 4, 2, 4, 2, 4, 1, 2, 4, 1, 2, 2, 2, 2, 4, 1, 4, 4, 4}. If 4 is picked as pivot in Simple QuickSort, we fix only one 4 and recursively process remaining elements.

- a) arr[..i] elements less than pivot.
- b) arr[i+1..j-1] elements equal to pivot.
- c) arr[j..r] elements greater than pivot.

See [this](#) for implementation.

How to implement QuickSort for Linked Lists?

[QuickSort on Singly Linked List](#)

[QuickSort on Doubly Linked List](#)

Can we implement QuickSort Iteratively?

Yes, please refer [Iterative Quick Sort](#).

Why Quick Sort is preferred over MergeSort for sorting Arrays

Quick Sort in its general form is an in-place sort (i.e. it doesn't require any extra storage) whereas merge sort requires $O(N)$ extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating $O(N)$ extra space can be time consuming.

Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of $O(n\log n)$. The worst case is possible in randomized version also, but worst case does not occur very often.

Quick Sort is also a cache friendly sorting algorithm as it has good locality of reference when used for arrays.

Quick Sort is also tail recursive, therefore tail call optimizations is done.

Why MergeSort is preferred over QuickSort for Linked Lists?

In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can't access elements using random access operator.

In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at address x + i * 4.

How to optimize QuickSort so that it takes $O(\log n)$ extra space in worst case?

Please see [QuickSort Tail Call Optimization \(Reducing worst case space to Log n\)](#)

Snapshots:

scene00865



scene01369



scene01801



scene02377



scene02881



scene03025



scene03385



scene03889

?

Quiz on QuickSort

References:

<http://en.wikipedia.org/wiki/Quicksort>

Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz:

Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Heap Sort, QuickSort, Radix Sort, Counting Sort, Bucket Sort, ShellSort, Comb Sort, Pigeonhole Sort

Coding practice for sorting.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Searching and Sorting

Radix Sort

The lower bound for Comparison based sorting algorithm (Merge Sort, Heap Sort, Quick-Sort .. etc) is $\Omega(n\log n)$, i.e., they cannot do better than $n\log n$.

Counting sort is a linear time sorting algorithm that sorts in $O(n+k)$ time when elements are in range from 1 to k .

What if the elements are in range from 1 to n^2 ?

We can't use counting sort because counting sort will take $O(n^2)$ which is worse than comparison based sorting algorithms. Can we sort such an array in linear time?

Radix Sort is the answer. The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort.

The Radix Sort Algorithm

1) Do following for each digit i where i varies from least significant digit to the most significant digit.

.....a) Sort input array using counting sort (or any stable sort) according to the i 'th digit.

Example:

Original, unsorted list:

170, 45, 75, 90, 802, 24, 2, 66

Sorting by least significant digit (1s place) gives: [*Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.]

170, 90, 802, 2, 24, 45, 75, 66

Sorting by next digit (10s place) gives: [*Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.]

802, 2, 24, 45, 66, 170, 75, 90

Sorting by most significant digit (100s place) gives:

2, 24, 45, 66, 75, 90, 170, 802

What is the running time of Radix Sort?

Let there be d digits in input integers. Radix Sort takes $O(d*(n+b))$ time where b is the base for representing numbers, for example, for decimal system, b is 10. What is the value of d ? If k is the maximum possible value, then d would be $O(\log_b(k))$. So overall time complexity is $O((n+b) * \log_b(k))$. Which looks more than the time complexity of comparison based sorting algorithms for a large k . Let us first limit k . Let $k c$ where c is a constant. In that case, the complexity becomes $O(n\log_b(n))$. But it still doesn't beat comparison based sorting algorithms.

What if we make value of b larger? What should be the value of b to make the time complexity linear? If we set b as n , we get the time complexity as $O(n)$. In other words, we can sort an array of integers with range from 1 to n^c if the numbers are represented in base n (or every digit takes $\log_2(n)$ bits).

Is Radix Sort preferable to Comparison based sorting algorithms like Quick-Sort?

If we have $\log_2 n$ bits for every digit, the running time of Radix Sort appears to be better than Quick Sort for a wide range of input numbers. The constant factors hidden in asymptotic notation are higher for Radix Sort and Quick-Sort uses hardware caches more effectively. Also, Radix sort uses counting sort as a subroutine and counting sort takes extra space to sort numbers.

Implementation of Radix Sort

Following is a simple C++ implementation of Radix Sort. For simplicity, the value of d is assumed to be 10. We recommend you to see [Counting Sort](#) for details of countSort() function in below code.

C/C++

```
// C++ implementation of Radix Sort
#include<iostream>
using namespace std;

// A utility function to get maximum value in arr[]
int getMax(int arr[], int n)
{
    int mx = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > mx)
            mx = arr[i];
    return mx;
}

// A function to do counting sort of arr[] according to
// the digit represented by exp.
void countSort(int arr[], int n, int exp)
{
    int output[n]; // output array
    int i, count[10] = {0};

    // Store count of occurrences in count[]
    for (i = 0; i < n; i++)
        count[(arr[i]/exp)%10]++;
    
    // Change count[i] so that count[i] now contains actual
    // position of this digit in output[]
    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Build the output array
    for (i = n - 1; i >= 0; i--)
    {
        output[count[(arr[i]/exp)%10] - 1] = arr[i];
        count[(arr[i]/exp)%10]--;
    }

    // Copy the output array to arr[], so that arr[] now
    // contains sorted numbers according to current digit
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}

// The main function to that sorts arr[] of size n using
// Radix Sort
void radixsort(int arr[], int n)
{
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);

    // Do counting sort for every digit. Note that instead
    // of passing digit number, exp is passed. exp is 10^i
    // where i is current digit number
    for (int exp = 1; m/exp > 0; exp *= 10)
        countSort(arr, n, exp);
}

// A utility function to print an array
void print(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
}

// Driver program to test above functions
int main()
{
    int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
    int n = sizeof(arr)/sizeof(arr[0]);
    radixsort(arr, n);
    print(arr, n);
    return 0;
}
```

Java

```
// Radix sort Java implementation
import java.io.*;
import java.util.*;

class Radix {
```

```

// A utility function to get maximum value in arr[]
static int getMax(int arr[], int n)
{
    int mx = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > mx)
            mx = arr[i];
    return mx;
}

// A function to do counting sort of arr[] according to
// the digit represented by exp.
static void countSort(int arr[], int n, int exp)
{
    int output[] = new int[n]; // output array
    int i;
    int count[] = new int[10];
    Arrays.fill(count, 0);

    // Store count of occurrences in count[]
    for (i = 0; i < n; i++)
        count[(arr[i]/exp)%10]++;
}

// Change count[i] so that count[i] now contains
// actual position of this digit in output[]
for (i = 1; i < 10; i++)
    count[i] += count[i - 1];

// Build the output array
for (i = n - 1; i >= 0; i--)
{
    output[count[(arr[i]/exp)%10] - 1] = arr[i];
    count[(arr[i]/exp)%10]--;
}

// Copy the output array to arr[], so that arr[] now
// contains sorted numbers according to current digit
for (i = 0; i < n; i++)
    arr[i] = output[i];
}

// The main function to that sorts arr[] of size n using
// Radix Sort
static void radixsort(int arr[], int n)
{
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);

    // Do counting sort for every digit. Note that instead
    // of passing digit number, exp is passed. exp is 10^i
    // where i is current digit number
    for (int exp = 1; m/exp > 0; exp *= 10)
        countSort(arr, n, exp);
}

// A utility function to print an array
static void print(int arr[], int n)
{
    for (int i=0; i<n; i++)
        System.out.print(arr[i] + " ");
}

/*Driver function to check for above function*/
public static void main (String[] args)
{
    int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
    int n = arr.length;
    radixsort(arr, n);
    print(arr, n);
}
/* This code is contributed by Devesh Agrawal */

```

Python

```

# Python program for implementation of Radix Sort

# A function to do counting sort of arr[] according to
# the digit represented by exp.
def countingSort(arr, exp1):

    n = len(arr)

    # The output array elements that will have sorted arr
    output = [0] * (n)

    # initialize count array as 0
    count = [0] * (10)

    # Store count of occurrences in count[]
    for i in range(0, n):
        index = (arr[i]/exp1)
        count[(index)%10] += 1

    # Change count[i] so that count[i] now contains actual
    # position of this digit in output array
    for i in range(1, 10):
        count[i] += count[i-1]

    # Build the output array
    i = n-1
    while i>=0:

```

```

index = (arr[i]/exp1)
output[ count[ (index)%10 ] - 1 ] = arr[i]
count[ (index)%10 ] -= 1
i -= 1

# Copying the output array to arr[], 
# so that arr now contains sorted numbers
i = 0
for i in range(0,len(arr)):
    arr[i] = output[i]

# Method to do Radix Sort
def radixSort(arr):

    # Find the maximum number to know number of digits
    max1 = max(arr)

    # Do counting sort for every digit. Note that instead
    # of passing digit number, exp is passed. exp is 10^i
    # where i is current digit number
    exp = 1
    while max1/exp > 0:
        countingSort(arr,exp)
        exp *= 10

    # Driver code to test above
    arr = [ 170, 45, 75, 90, 802, 24, 2, 66]
    radixSort(arr)

for i in range(len(arr)):
    print(arr[i])

# This code is contributed by Mohit Kumra

```

Output:

2 24 45 66 75 90 170 802

Snapshots:
scene00577



scene00649



scene00793



scene01009



scene01225



Quiz on Radix Sort

Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz:

- Selection Sort
- Bubble Sort
- Insertion Sort
- Merge Sort
- Heap Sort
- QuickSort
- Counting Sort
- Bucket Sort
- ShellSort

References:

http://en.wikipedia.org/wiki/Radix_sort

<http://alg12.wikischolars.columbia.edu/file/view/RADIX.pdf>

MIT Video Lecture

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

GATE CS Corner Company Wise Coding Practice

Sorting

Counting Sort

Counting sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence.

Let us understand it with the help of an example.

```
For simplicity, consider the data in the range 0 to 9.  
Input data: 1, 4, 1, 2, 7, 5, 2  
1) Take a count array to store the count of each unique object.  
Index: 0 1 2 3 4 5 6 7 8 9  
Count: 0 2 2 0 1 1 0 1 0 0
```

```
2) Modify the count array such that each element at each index  
stores the sum of previous counts.  
Index: 0 1 2 3 4 5 6 7 8 9  
Count: 0 2 4 4 5 6 6 7 7 7
```

The modified count array indicates the position of each object in the output sequence.

```
3) Output each object from the input sequence followed by  
decreasing its count by 1.  
Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2.  
Put data 1 at index 2 in output. Decrease count by 1 to place  
next data 1 at an index 1 smaller than this index.
```

Following is C implementation of counting sort.

C/C++

```
// C Program for counting sort  
#include <stdio.h>  
#include <string.h>  
#define RANGE 255  
  
// The main function that sort the given string arr[] in  
// alphabetical order  
void countSort(char arr[]){  
    // The output character array that will have sorted arr  
    char output[strlen(arr)];  
  
    // Create a count array to store count of individual  
    // characters and initialize count array as 0  
    int count[RANGE + 1], i;  
    memset(count, 0, sizeof(count));  
  
    // Store count of each character  
    for(i = 0; arr[i]; ++i)  
        ++count[arr[i]];  
  
    // Change count[i] so that count[i] now contains actual  
    // position of this character in output array  
    for (i = 1; i <= RANGE; ++i)  
        count[i] += count[i-1];  
  
    // Build the output character array  
    for (i = 0; arr[i]; ++i)  
    {  
        output[count[arr[i]]-1] = arr[i];  
        --count[arr[i]];  
    }  
  
    // Copy the output array to arr, so that arr now  
    // contains sorted characters  
    for (i = 0; arr[i]; ++i)  
        arr[i] = output[i];  
}  
  
// Driver program to test above function  
int main()  
{  
    char arr[] = "geeksforgeeks"/"applepp";  
  
    countSort(arr);  
  
    printf("Sorted character array is %s\n", arr);  
    return 0;  
}
```

Java

```
// Java implementation of Counting Sort  
class CountingSort  
{  
    void sort(char arr[]){  
        int n = arr.length;  
  
        // The output character array that will have sorted arr  
        char output[] = new char[n];
```

```

// Create a count array to store count of individual
// characters and initialize count array as 0
int count[] = new int[256];
for (int i=0; i<256; ++i)
    count[i] = 0;

// store count of each character
for (int i=0; i<n; ++i)
    ++count[arr[i]];

// Change count[i] so that count[i] now contains actual
// position of this character in output array
for (int i=1; i<=255; ++i)
    count[i] += count[i-1];

// Build the output character array
for (int i = 0; i<n; ++i)
{
    output[count[arr[i]]-1] = arr[i];
    --count[arr[i]];
}

// Copy the output array to arr, so that arr now
// contains sorted characters
for (int i = 0; i<n; ++i)
    arr[i] = output[i];
}

// Driver method
public static void main(String args[])
{
    CountingSort ob = new CountingSort();
    char arr[] = {'g', 'e', 'e', 'k', 's', 'f', 'o',
                 'r', 'g', 'e', 'e', 'k', 's'};
    ob.sort(arr);
    System.out.print("Sorted character array is ");
    for (int i=0; i<arr.length; ++i)
        System.out.print(arr[i]);
}
}

/*This code is contributed by Rajat Mishra */

```

Python

```

# Python program for counting sort

# The main function that sort the given string arr[] in
# alphabetical order
def countSort(arr):

    # The output character array that will have sorted arr
    output = [0 for i in range(256)]

    # Create a count array to store count of individual
    # characters and initialize count array as 0
    count = [0 for i in range(256)]

    # For storing the resulting answer since the
    # string is immutable
    ans = [""] * len(arr)

    # Store count of each character
    for i in arr:
        count[ord(i)] += 1

    # Change count[i] so that count[i] now contains actual
    # position of this character in output array
    for i in range(256):
        count[i] += count[i-1]

    # Build the output character array
    for i in range(len(arr)):
        output[count[ord(arr[i])]-1] = arr[i]
        count[ord(arr[i])] -= 1

    # Copy the output array to arr, so that arr now
    # contains sorted characters
    for i in range(len(arr)):
        ans[i] = output[i]
    return ans

# Driver program to test above function
arr = "geeksforgeeks"
ans = countSort(arr)
print "Sorted character array is %s" %("".join(ans))

# This code is contributed by Nikhil Kumar Singh

```

Output:

Sorted character array is eeeeefggkkors

Time Complexity: O(n+k) where n is the number of elements in input array and k is the range of input.

Auxiliary Space: O(n+k)

Points to be noted:

1. Counting sort is efficient if the range of input data is not significantly greater than the number of objects to be sorted. Consider the situation where the input sequence is between range 1 to 10K and the data is 10, 5, 10K, 5K.
2. It is not a comparison based sorting. Its running time complexity is O(n) with space proportional to the range of data.

3. It is often used as a sub-routine to another sorting algorithm like radix sort.
4. Counting sort uses a partial hashing to count the occurrence of the data object in $O(1)$.
5. Counting sort can be extended to work for negative inputs also.

Exercise:

1. Modify above code to sort the input data in the range from M to N.
2. Modify above code to sort negative input data.
3. Is counting sort stable and online?
4. Thoughts on parallelizing the counting sort algorithm.

Snapshots:
scene00721



scene00865



scene01153



scene01297



scene01369



scene02521



scene02881



Quiz on Counting Sort

Coding Practice for Sorting

Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz

Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Heap Sort, QuickSort, Radix Sort, Counting Sort, Bucket Sort, ShellSort, Comb Sort, PigeonHole Sorting

This article is compiled by Aashish Barnwal. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Sorting

Bucket Sort

Bucket sort is mainly useful when input is uniformly distributed over a range. For example, consider the following problem.

Sort a large set of floating point numbers which are in range from 0.0 to 1.0 and are uniformly distributed across the range. How do we sort the numbers efficiently?

A simple way is to apply a comparison based sorting algorithm. The lower bound for Comparison based sorting algorithm (Merge Sort, Heap Sort, Quick-Sort .. etc) is $\Omega(n \log n)$, i.e., they cannot do better than $n \log n$.

Can we sort the array in linear time? Counting sort can not be applied here as we use keys as index in counting sort. Here keys are floating point numbers.

The idea is to use bucket sort. Following is bucket algorithm.

```
bucketSort(arr[], n)
1) Create n empty buckets (Or lists).
2) Do following for every array element arr[i].
.....a) Insert arr[i] into bucket[n*array[i]]
3) Sort individual buckets using insertion sort.
4) Concatenate all sorted buckets.
```

Following diagram (taken from CLRS book) demonstrates working of bucket sort.



Time Complexity: If we assume that insertion in a bucket takes $O(1)$ time then steps 1 and 2 of the above algorithm clearly take $O(n)$ time. The $O(1)$ is easily possible if we use a linked list to represent a bucket (In the following code, C++ vector is used for simplicity). Step 4 also takes $O(n)$ time as there will be n items in all buckets.

The main step to analyze is step 3. This step also takes $O(n)$ time on average if all numbers are uniformly distributed (please refer CLRS book for more details)

Following is C++ implementation of the above algorithm.

```
// C++ program to sort an array using bucket sort
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

// Function to sort arr[] of size n using bucket sort
void bucketSort(float arr[], int n)
{
    // 1) Create n empty buckets
    vector<float> b[n];

    // 2) Put array elements in different buckets
    for (int i=0; i<n; i++)
    {
        int bi = n*arr[i]; // Index in bucket
        b[bi].push_back(arr[i]);
    }

    // 3) Sort individual buckets
    for (int i=0; i<n; i++)
        sort(b[i].begin(), b[i].end());

    // 4) Concatenate all buckets into arr[]
    int index = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < b[i].size(); j++)
            arr[index++] = b[i][j];
}

/* Driver program to test above function */
int main()
{
    float arr[] = {0.897, 0.565, 0.656, 0.1234, 0.665, 0.3434};
    int n = sizeof(arr)/sizeof(arr[0]);
    bucketSort(arr, n);

    cout << "Sorted array is \n";
    for (int i=0; i<n; i++)
        cout << arr[i] << " ";
    return 0;
}
```

Output:

```
Sorted array is
0.1234 0.3434 0.565 0.656 0.665 0.897
```

Bucket Sort To Sort an Array with Negative Numbers

References:

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

http://en.wikipedia.org/wiki/Bucket_sort

Snapshots:
scene00505



scene01009



scene01513



scene01729



scene01801



scene01945



scene02017



scene02521



Quiz on Bucket Sort

Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz:

- Selection Sort
- Bubble Sort
- Insertion Sort
- Merge Sort
- Heap Sort
- QuickSort
- Radix Sort
- Counting Sort
- ShellSort

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

ShellSort

ShellSort is mainly a variation of **Insertion Sort**. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of shellSort is to allow exchange of far items. In shellSort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sublists of every h'th element is sorted.

Following is C++ implementation of ShellSort.

C++

```
// C++ implementation of Shell Sort
#include <iostream>
using namespace std;

/* function to sort arr using shellSort */
int shellSort(int arr[], int n)
{
    // Start with a big gap, then reduce the gap
    for (int gap = n/2; gap > 0; gap /= 2)
    {
        // Do a gapped insertion sort for this gap size.
        // The first gap elements a[0..gap-1] are already in gapped order
        // keep adding one more element until the entire array is
        // gap sorted
        for (int i = gap; i < n; i += 1)
        {
            // add a[i] to the elements that have been gap sorted
            // save a[i] in temp and make a hole at position i
            int temp = arr[i];

            // shift earlier gap-sorted elements up until the correct
            // location for a[i] is found
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
                arr[j] = arr[j - gap];

            // put temp (the original a[i]) in its correct location
            arr[j] = temp;
        }
    }
    return 0;
}

void printArray(int arr[], int n)
{
    for (int i=0; i<n; i++)
        cout << arr[i] << " ";
}

int main()
{
    int arr[] = {12, 34, 54, 2, 3}, i;
    int n = sizeof(arr)/sizeof(arr[0]);

    cout << "Array before sorting: \n";
    printArray(arr, n);

    shellSort(arr, n);

    cout << "\nArray after sorting: \n";
    printArray(arr, n);

    return 0;
}
```

Java

```
// Java implementation of ShellSort
class ShellSort
{
    /* An utility function to print array of size n*/
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; i++)
            System.out.print(arr[i] + " ");
        System.out.println();
    }

    /* function to sort arr using shellSort */
    int sort(int arr[])
    {
        int n = arr.length;

        // Start with a big gap, then reduce the gap
        for (int gap = n/2; gap > 0; gap /= 2)
        {
            // Do a gapped insertion sort for this gap size.
            // The first gap elements a[0..gap-1] are already
            // in gapped order keep adding one more element
            // until the entire array is gap sorted
            for (int i = gap; i < n; i += 1)
            {
                // add a[i] to the elements that have been gap
                // sorted save a[i] in temp and make a hole at
                // position i
                int temp = arr[i];

                // shift earlier gap-sorted elements up until
```

```

// the correct location for a[i] is found
int j;
for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
    arr[j] = arr[j - gap];

// put temp (the original a[i]) in its correct
// location
arr[j] = temp;
}
return 0;
}

// Driver method
public static void main(String args[])
{
    int arr[] = {12, 34, 54, 2, 3};
    System.out.println("Array before sorting");
    printArray(arr);

    ShellSort ob = new ShellSort();
    ob.sort(arr);

    System.out.println("Array after sorting");
    printArray(arr);
}
/*This code is contributed by Rajat Mishra */

```

Python

```

# Python program for implementation of Shell Sort

def shellSort(arr):

    # Start with a big gap, then reduce the gap
    n = len(arr)
    gap = n//2

    # Do a gapped insertion sort for this gap size.
    # The first gap elements a[0..gap-1] are already in gapped
    # order keep adding one more element until the entire array
    # is gap sorted
    while gap > 0:

        for i in range(gap,n):

            # add arr[i] to the elements that have been gap sorted
            # save arr[i] in temp and make a hole at position i
            temp = arr[i]

            # shift earlier gap-sorted elements up until the correct
            # location for arr[i] is found
            j = i
            while j >= gap and arr[j-gap] > temp:
                arr[j] = arr[j-gap]
                j -= gap

            # put temp (the original arr[i]) in its correct location
            arr[j] = temp
            gap //= 2

    # Driver code to test above
    arr = [ 12, 34, 54, 2, 3]

    n = len(arr)
    print ("Array before sorting:")
    for i in range(n):
        print(arr[i]),

    shellSort(arr)

    print ("\nArray after sorting:")
    for i in range(n):
        print(arr[i]),

# This code is contributed by Mohit Kumra

```

Output:

```

Array before sorting:
12 34 54 2 3
Array after sorting:
2 3 12 34 54

```

Time Complexity: Time complexity of above implementation of shellsort is $O(n^2)$. In the above implementation gap is reduce by half in every iteration. There are many other ways to reduce gap which lead to better time complexity. See [this](#) for more details.

References:

<https://www.youtube.com/watch?v=pGhazjsFW28>
<http://en.wikipedia.org/wiki/Shellsort>

Snapshots:
scene00721



scene00793



scene00937



scene01009



scene01801



scene02305



Quiz on Shell Sort

Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz:

- Selection Sort
- Bubble Sort
- Insertion Sort
- Merge Sort
- Heap Sort
- QuickSort
- Radix Sort
- Counting Sort
- Bucket Sort

Coding practice for sorting.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Searching and Sorting

Comb Sort

Comb Sort is mainly an improvement over Bubble Sort. Bubble sort always compares adjacent values. So all inversions are removed one by one. Comb Sort improves on Bubble Sort by using gap of size more than 1. The gap starts with a large value and shrinks by a factor of 1.3 in every iteration until it reaches the value 1. Thus Comb Sort removes more than one inversion counts with one swap and performs better than Bubble Sort.

The shrink factor has been empirically found to be 1.3 (by testing Combsort on over 200,000 random lists) [Source: Wiki]

Although, it works better than Bubble Sort on average, worst case remains $O(n^2)$.

Below is C++ implementation.

C++

```
// C++ implementation of Comb Sort
#include<bits/stdc++.h>
using namespace std;

// To find gap between elements
int getNextGap(int gap)
{
    // Shrink gap by Shrink factor
    gap = (gap*10)/13;

    if (gap < 1)
        return 1;
    return gap;
}

// Function to sort a[0..n-1] using Comb Sort
void combSort(int a[], int n)
{
    // Initialize gap
    int gap = n;

    // Initialize swapped as true to make sure that
    // loop runs
    bool swapped = true;

    // Keep running while gap is more than 1 and last
    // iteration caused a swap
    while (gap != 1 || swapped == true)
    {
        // Find next gap
        gap = getNextGap(gap);

        // Initialize swapped as false so that we can
        // check if swap happened or not
        swapped = false;

        // Compare all elements with current gap
        for (int i=0; i<n-gap; i++)
        {
            if (a[i] > a[i+gap])
            {
                swap(a[i], a[i+gap]);
                swapped = true;
            }
        }
    }

    // Driver program
    int main()
    {
        int a[] = {8, 4, 1, 56, 3, -44, 23, -6, 28, 0};
        int n = sizeof(a)/sizeof(a[0]);

        combSort(a, n);

        printf("Sorted array: \n");
        for (int i=0; i<n; i++)
            printf("%d ", a[i]);

        return 0;
    }
}
```

Java

```
// Java program for implementation of Comb Sort
class CombSort
{
    // To find gap between elements
    int getNextGap(int gap)
    {
        // Shrink gap by Shrink factor
        gap = (gap*10)/13;
        if (gap < 1)
```

```

        return 1;
    return gap;
}

// Function to sort arr[] using Comb Sort
void sort(int arr[])
{
    int n = arr.length;

    // initialize gap
    int gap = n;

    // Initialize swapped as true to make sure that
    // loop runs
    boolean swapped = true;

    // Keep running while gap is more than 1 and last
    // iteration caused a swap
    while (gap != 1 || swapped == true)
    {
        // Find next gap
        gap = getNextGap(gap);

        // Initialize swapped as false so that we can
        // check if swap happened or not
        swapped = false;

        // Compare all elements with current gap
        for (int i=0; i<n-gap; i++)
        {
            if (arr[i] > arr[i+gap])
            {
                // Swap arr[i] and arr[i+gap]
                int temp = arr[i];
                arr[i] = arr[i+gap];
                arr[i+gap] = temp;

                // Set swapped
                swapped = true;
            }
        }
    }

    // Driver method
    public static void main(String args[])
    {
        CombSort ob = new CombSort();
        int arr[] = {8, 4, 1, 56, 3, -44, 23, -6, 28, 0};
        ob.sort(arr);

        System.out.println("sorted array");
        for (int i=0; i<arr.length; i++)
            System.out.print(arr[i] + " ");

    }
}
/* This code is contributed by Rajat Mishra */

```

Python

```

# Python program for implementation of CombSort

# To find next gap from current
def getNextGap(gap):

    # Shrink gap by Shrink factor
    gap = (gap * 10)/13
    if gap < 1:
        return 1
    return gap

# Function to sort arr[] using Comb Sort
def combSort(arr):
    n = len(arr)

    # Initialize gap
    gap = n

    # Initialize swapped as true to make sure that
    # loop runs
    swapped = True

    # Keep running while gap is more than 1 and last
    # iteration caused a swap
    while gap != 1 or swapped == 1:

        # Find next gap
        gap = getNextGap(gap)

        # Initialize swapped as false so that we can
        # check if swap happened or not
        swapped = False

        # Compare all elements with current gap
        for i in range(0, n-gap):
            if arr[i] > arr[i + gap]:
                arr[i], arr[i + gap] = arr[i + gap], arr[i]
                swapped = True

# Driver code to test above
arr = [8, 4, 1, 56, 3, -44, 23, -6, 28, 0]
combSort(arr)

```

```
print ("Sorted array:")
for i in range(len(arr)):
    print (arr[i])

# This code is contributed by Mohit Kumra
```

Output :

```
Sorted array:
-44 -6 0 1 3 4 8 23 28 56
```

Illustration:

Let the array elements be

```
8, 4, 1, 56, 3, -44, 23, -6, 28, 0
```

Initially gap value = 10

After shrinking gap value => $10/1.3 = 7$:

```
8 4 1 56 3 -44 23 -6 28 0
-6 4 1 56 3 -44 23 8 28 0
-6 4 0 56 3 -44 23 8 28 1
```

New gap value => $7/1.3 = 5$:

```
-44 4 0 56 3 -6 23 8 28 1
-44 4 0 28 3 -6 23 8 56 1
-44 4 0 28 1 -6 23 8 56 3
```

New gap value => $5/1.3 = 3$:

```
-44 1 0 28 4 -6 23 8 56 3
-44 1 -6 28 4 0 23 8 56 3
-44 1 -6 23 4 0 28 8 56 3
-44 1 -6 23 4 0 3 8 56 28
```

New gap value => $3/1.3 = 2$:

```
-44 1 -6 0 4 23 3 8 56 28
-44 1 -6 0 3 23 4 8 56 28
-44 1 -6 0 3 8 4 23 56 28
```

New gap value => $2/1.3 = 1$:

```
-44 -6 1 0 3 8 4 23 56 28
-44 -6 0 1 3 8 4 23 56 28
-44 -6 0 1 3 4 8 23 56 28
-44 -6 0 1 3 4 8 23 28 56
```

no more swaps required (Array sorted)

Time Complexity : Worst case complexity of this algorithm is $O(n^2)$ and the Best Case complexity is $O(n)$.

Auxiliary Space : $O(1)$.

Quiz on Comb Sort

This article is contributed by **Rahul Agrawal**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Snapshots:

scene00505



scene00577



scene00721



scene02305



scene02809



scene02881



Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz

Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Heap Sort, QuickSort, Radix Sort, Counting Sort, Bucket Sort, ShellSort, Pigeonhole Sort

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Sorting

Pigeonhole Sort

Pigeonhole sorting is a sorting algorithm that is suitable for sorting lists of elements where the number of elements and the number of possible key values are approximately the same. It requires $O(n + Range)$ time where n is number of elements in input array and 'Range' is number of possible values in array.

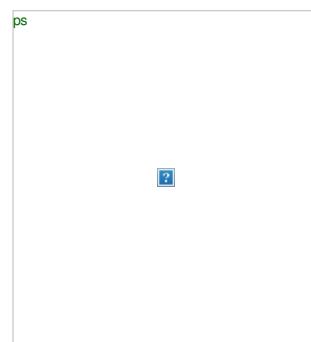
Working of Algorithm :

1. Find minimum and maximum values in array. Let the minimum and maximum values be 'min' and 'max' respectively. Also find range as 'max-min-1'.
2. Set up an array of initially empty "pigeonholes" the same size as of the range.

3. Visit each element of the array and then put each element in its pigeonhole. An element arr[i] is put in hole at index arr[i] – min.
4. Start the loop all over the pigeonhole array in order and put the elements from non- empty holes back into the original array.

Comparison with Counting Sort :

It is similar to [counting sort](#), but differs in that it "moves items twice: once to the bucket array and again to the final destination".



Below is C++ implementation of Pigeonhole Sort.

```
/* C program to implement Pigeonhole Sort */
#include <bits/stdc++.h>
using namespace std;

/* Sorts the array using pigeonhole algorithm */
void pigeonholeSort(int arr[], int n)
{
    // Find minimum and maximum values in arr[]
    int min = arr[0], max = arr[0];
    for (int i = 1; i < n; i++)
    {
        if (arr[i] < min)
            min = arr[i];
        if (arr[i] > max)
            max = arr[i];
    }
    int range = max - min + 1; // Find range

    // Create an array of vectors. Size of array
    // range. Each vector represents a hole that
    // is going to contain matching elements.
    vector<int> holes(range);

    // Traverse through input array and put every
    // element in its respective hole
    for (int i = 0; i < n; i++)
        holes[arr[i]-min].push_back(arr[i]);

    // Traverse through all holes one by one. For
    // every hole, take its elements and put in
    // array.
    int index = 0; // index in sorted array
    for (int i = 0; i < range; i++)
    {
        vector<int>::iterator it;
        for (it = holes[i].begin(); it != holes[i].end(); ++it)
            arr[index++] = *it;
    }
}

// Driver program to test the above function
int main()
{
    int arr[] = {8, 3, 2, 7, 4, 6, 8};
    int n = sizeof(arr)/sizeof(arr[0]);

    pigeonholeSort(arr, n);

    printf("Sorted order is : ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

Output:

```
Sorted order is : 2 3 4 6 7 8 8
```

Pigeonhole sort has limited use as requirements are rarely met. For arrays where range is much larger than n , bucket sort is a generalization that is more efficient in space and time.

References:

https://en.wikipedia.org/wiki/Pigeonhole_sort

This article is contributed **Ayush Govil**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz

Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Heap Sort, QuickSort, Radix Sort, Counting Sort, Bucket Sort, ShellSort, Comb Sort,

GATE CS Corner Company Wise Coding Practice

Sorting

Cycle Sort

Cycle sort is an in-place sorting Algorithm, [unstable sorting algorithm](#), a comparison sort that is theoretically optimal in terms of the total number of writes to the original array.

- It is optimal in terms of number of memory writes. It [minimizes the number of memory writes](#) to sort (Each value is either written zero times, if it's already in its correct position, or written one time to its correct position.)
- It is based on the idea that array to be sorted can be divided into cycles. Cycles can be visualized as a graph. We have n nodes and an edge directed from node i to node j if the element at i-th index must be present at j-th index in the sorted array.

Cycle in arr[] = {4, 5, 2, 1, 5}

cycle-sort



Cycle in arr[] = {4, 3, 2, 1}

cyclic-sort2



We one by one consider all cycles. We first consider the cycle that includes first element. We find correct position of first element, place it at its correct position, say j. We consider old value of arr[j] and find its correct position, we keep doing this till all elements of current cycle are placed at correct position, i.e., we don't come back to cycle starting point.

Explanation :

```
arr[] = {10, 5, 2, 3}
index = 0 1 2 3
cycle_start = 0
item = 10 = arr[0]
```

Find position where we put the item
pos = cycle_start
while (arr[i] < item)
 pos++;

We put 10 at arr[3] and change item to
old value of arr[3].
arr[] = {10, 5, 2, **10**}
item = 3

Again rotate rest cycle that start with **index '0'**
Find position where we put the item = 3
we swap item with element at arr[1] now
arr[] = {10, **3**, 2, **10**}
item = 5

Again rotate rest cycle that start with index '0' and item = 5
we swap item with element at arr[2].
arr[] = {10, **3**, **5**, **10**}
item = 2

Again rotate rest cycle that start with index '0' and item = 2
arr[] = {**2**, **3**, **5**, **10**}

Above is one iteration for cycle_start = 0.
Repeat above steps for cycle_start = 1, 2, ..n-2

Below is C++ implementation of above Cycle Sort.

```
// C++ program to implement cycle sort
#include <iostream>
using namespace std;

// Function sort the array using Cycle sort
void cycleSort (int arr[], int n)
{
    // count number of memory writes
    int writes = 0;

    // traverse array elements and put it to on
    // the right place
    for (int cycle_start=0; cycle_start<=n-2; cycle_start++)
    {
        // initialize item as starting point
        int item = arr[cycle_start];

        // Find position where we put the item. We basically
        // count all smaller elements on right side of item.
        int pos = cycle_start;
        for (int i = cycle_start+1; i<n; i++)
            if (arr[i] < item)
                pos++;

        // If item is already in correct position
        if (pos == cycle_start)
            continue;

        // ignore all duplicate elements
        while (item == arr[pos])
            pos += 1;

        // put the item to it's right position
        if (pos != cycle_start)
            swap(arr[pos], arr[cycle_start]);
    }
}
```

```

    {
        swap(item, arr[pos]);
        writes++;
    }

    // Rotate rest of the cycle
    while (pos != cycle_start)
    {
        pos = cycle_start;

        // Find position where we put the element
        for (int i = cycle_start+1; i<n; i++)
            if (arr[i] < item)
                pos += 1;

        // ignore all duplicate elements
        while (item == arr[pos])
            pos += 1;

        // put the item to its right position
        if (item != arr[pos])
        {
            swap(item, arr[pos]);
            writes++;
        }
    }

    // Number of memory writes or swaps
    // cout << writes << endl;
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 8, 3, 9, 10, 10, 2, 4 };
    int n = sizeof(arr)/sizeof(arr[0]);
    cycleSort(arr, n);

    cout << "After sort : " << endl;
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    return 0;
}

```

Output:

1 3 4 8 10 10

Time Complexity : $O(n^2)$

Worst Case : $O(n^2)$

Average Case: $O(n^2)$

Best Case : $O(n^2)$

This sorting algorithm is best suited for situations where memory write or swap operations are costly.

Reference:

https://en.wikipedia.org/wiki/Cycle_sort

This article is contributed by **Nishant Singh**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Sorting

Interpolation search vs Binary search

[Interpolation search](#) works better than Binary Search for a sorted and uniformly distributed array.

On average the interpolation search makes about $\log(\log(n))$ comparisons (if the elements are uniformly distributed), where n is the number of elements to be searched. In the worst case (for instance where the numerical values of the keys increase exponentially) it can make up to $O(n)$ comparisons.

[Interpolation Search Article](#)

Sources:

http://en.wikipedia.org/wiki/Interpolation_search

GATE CS Corner Company Wise Coding Practice

Searching
Binary-Search

Stability in sorting algorithms

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array. Some sorting algorithms are stable by nature like Insertion sort, Merge Sort, Bubble Sort, etc. And some sorting algorithms are not, like Heap Sort, Quick Sort, etc.

However, any given sorting algo which is not stable can be modified to be stable. There can be sorting algo specific ways to make it stable, but in general, any comparison based sorting algorithm which is not stable by nature can be modified to be stable by changing the key comparison operation so that the comparison of two keys considers position as a factor for objects with equal keys.

References:

<http://www.math.uic.edu/~leon/cs-mcs401-s08/handouts/stability.pdf>

http://en.wikipedia.org/wiki/Sorting_algorithm#Stability

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

When does the worst case of Quicksort occur?

The answer depends on strategy for choosing pivot. In early versions of Quick Sort where leftmost (or rightmost) element is chosen as pivot, the worst occurs in following cases.

- 1) Array is already sorted in same order.
- 2) Array is already sorted in reverse order.
- 3) All elements are same (special case of case 1 and 2)

Since these cases are very common use cases, the problem was easily solved by choosing either a random index for the pivot, choosing the middle index of the partition or (especially for longer partitions) choosing the median of the first, middle and last element of the partition for the pivot. With these modifications, the worst case of Quick sort has less chances to occur, but worst case can still occur if the input array is such that the maximum (or minimum) element is always chosen as pivot.

References:

<http://en.wikipedia.org/wiki/Quicksort>

GATE CS Corner Company Wise Coding Practice

Lower bound for comparison based sorting algorithms

The problem of sorting can be viewed as following.

Input: A sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence such that $a'_1 < a'_2 < \dots < a'_n$.

A sorting algorithm is comparison based if it uses comparison operators to find the order between two numbers. Comparison sorts can be viewed abstractly in terms of decision trees. A decision tree is a full binary tree that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size. The execution of the sorting algorithm corresponds to tracing a path from the root of the decision tree to a leaf. At each internal node, a comparison $a_i < a_j$ is made. The left subtree then dictates subsequent comparisons for $a_i < a_k$, and the right subtree dictates subsequent comparisons for $a_j > a_k$. When we come to a leaf, the sorting algorithm has established the ordering. So we can say following about the decision tree.

- 1) Each of the $n!$ permutations on n elements must appear as one of the leaves of the decision tree for the sorting algorithm to sort properly.
- 2) Let x be the maximum number of comparisons in a sorting algorithm. The maximum height of the decision tree would be x . A tree with maximum height x has at most 2^x leaves.

After combining the above two facts, we get following relation.

$$n! \leq 2^x$$

Taking Log on both sides.

$$\log_2(n!) \leq x$$

Since $\log_2(n!) = \Theta(n\log n)$, we can say

$$x = \Omega(n\log_2 n)$$

Therefore, any comparison based sorting algorithm must make at least $n\log_2 n$ comparisons to sort the input array, and Heapsort and merge sort are asymptotically optimal comparison sorts.

References:

Introduction to Algorithms, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein

GATE CS Corner Company Wise Coding Practice

Which sorting algorithm makes minimum number of memory writes?

Minimizing the number of writes is useful when making writes to some huge data set is very expensive, such as with EEPROMs or Flash memory, where each write reduces the lifespan of the memory.

Among the sorting algorithms that we generally study in our data structure and algorithm courses, Selection Sort makes least number of writes (it makes $O(n)$ swaps). But, Cycle Sort almost always makes less number of writes compared to Selection Sort. In Cycle Sort, each value is either written zero times, if it's already in its correct position, or written one time to its correct position. This matches the minimal number of overwrites required for a completed in-place sort.

Sources:

http://en.wikipedia.org/wiki/Cycle_sort

http://en.wikipedia.org/wiki/Selection_sort

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Find the Minimum length Unsorted Subarray, sorting which makes the complete array sorted

Given an unsorted array $arr[0..n-1]$ of size n , find the minimum length subarray $arr[s..e]$ such that sorting this subarray makes the whole array sorted.

Examples:

- 1) If the input array is [10, 12, 20, 30, 25, 40, 32, 31, 35, 50, 60], your program should be able to find that the subarray lies between the indexes 3 and 8.
- 2) If the input array is [0, 1, 15, 25, 6, 7, 30, 40, 50], your program should be able to find that the subarray lies between the indexes 2 and 5.

We strongly recommend that you click here and practice it, before moving on to the solution.

Solution:

- 1) Find the candidate unsorted subarray

- a) Scan from left to right and find the first element which is greater than the next element. Let s be the index of such an element. In the above example 1, s is 3 (index of 30).
 b) Scan from right to left and find the first element (first in right to left order) which is smaller than the next element (next in right to left order). Let e be the index of such an element. In the above example 1, e is 7 (index of 31).

2) Check whether sorting the candidate unsorted subarray makes the complete array sorted or not. If not, then include more elements in the subarray.

- a) Find the minimum and maximum values in $\text{arr}[s..e]$. Let minimum and maximum values be min and max . min and max for [30, 25, 40, 32, 31] are 25 and 40 respectively.
 b) Find the first element (if there is any) in $\text{arr}[0..s-1]$ which is greater than min , change s to index of this element. There is no such element in above example 1.
 c) Find the last element (if there is any) in $\text{arr}[e+1..n-1]$ which is smaller than max , change e to index of this element. In the above example 1, e is changed to 8 (index of 35)

3) Print s and e .

Implementation:

```
#include<stdio.h>

void printUnsorted(int arr[], int n)
{
    int s = 0, e = n-1, i, max, min;

    // step 1(a) of above algo
    for (s = 0; s < n-1; s++)
    {
        if (arr[s] > arr[s+1])
            break;
    }
    if (s == n-1)
    {
        printf("The complete array is sorted");
        return;
    }

    // step 1(b) of above algo
    for(e = n - 1; e > 0; e--)
    {
        if(arr[e] < arr[e-1])
            break;
    }

    // step 2(a) of above algo
    max = arr[s]; min = arr[s];
    for(i = s + 1; i <= e; i++)
    {
        if(arr[i] > max)
            max = arr[i];
        if(arr[i] < min)
            min = arr[i];
    }

    // step 2(b) of above algo
    for( i = 0; i < s; i++)
    {
        if(arr[i] > min)
        {
            s = i;
            break;
        }
    }

    // step 2(c) of above algo
    for( i = n - 1; i >= e+1; i--)
    {
        if(arr[i] < max)
        {
            e = i;
            break;
        }
    }

    // step 3 of above algo
    printf(" The unsorted subarray which makes the given array "
           " sorted lies between the indees %d and %d", s, e);
    return;
}

int main()
{
    int arr[] = {10, 12, 20, 30, 25, 40, 32, 31, 35, 50, 60};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    printUnsorted(arr, arr_size);
    getchar();
    return 0;
}
```

Time Complexity: O(n)

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Sorting

Merge Sort for Linked Lists

Merge sort is often preferred for sorting a linked list. The slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

Let head be the first node of the linked list to be sorted and headRef be the pointer to head. Note that we need a reference to head in MergeSort() as the below implementation changes next links to sort the linked lists (not data at the nodes), so head node has to be changed if the data at original head is not the smallest value in linked list.

```
MergeSort(headRef)
1) If head is NULL or there is only one element in the Linked List
   then return.
```

2) Else divide the linked list into two halves.
 FrontBackSplit(head, &a, &b); /* a and b are two halves */
 3) Sort the two halves a and b.
 MergeSort(a);
 MergeSort(b);
 4) Merge the sorted a and b (using SortedMerge() discussed [here](#))
 and update the head pointer using headRef.
 *headRef = SortedMerge(a, b);

```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* function prototypes */
struct node* SortedMerge(struct node* a, struct node* b);
void FrontBackSplit(struct node* source,
                    struct node** frontRef, struct node** backRef);

/* sorts the linked list by changing next pointers (not data) */
void MergeSort(struct node** headRef)
{
    struct node* head = *headRef;
    struct node* a;
    struct node* b;

    /* Base case -- length 0 or 1 */
    if ((head == NULL) || (head->next == NULL))
    {
        return;
    }

    /* Split head into 'a' and 'b' sublists */
    FrontBackSplit(head, &a, &b);

    /* Recursively sort the sublists */
    MergeSort(&a);
    MergeSort(&b);

    /* answer = merge the two sorted lists together */
    *headRef = SortedMerge(a, b);
}

/* See http://geeksforgeeks.org/?p=3622 for details of this
function */
struct node* SortedMerge(struct node* a, struct node* b)
{
    struct node* result = NULL;

    /* Base cases */
    if (a == NULL)
        return(b);
    else if (b==NULL)
        return(a);

    /* Pick either a or b, and recur */
    if (a->data <= b->data)
    {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
    else
    {
        result = b;
        result->next = SortedMerge(a, b->next);
    }
    return(result);
}

/* UTILITY FUNCTIONS */
/* Split the nodes of the given list into front and back halves,
and return the two lists using the reference parameters.
If the length is odd, the extra node should go in the front list.
Uses the fast/slow pointer strategy. */
void FrontBackSplit(struct node* source,
                    struct node** frontRef, struct node** backRef)
{
    struct node* fast;
    struct node* slow;
    if (source==NULL || source->next==NULL)
    {
        /* length < 2 cases */
        *frontRef = source;
        *backRef = NULL;
    }
    else
    {
        slow = source;
        fast = source->next;

        /* Advance 'fast' two nodes, and advance 'slow' one node */
        while (fast != NULL)
        {
            fast = fast->next;
            if (fast != NULL)
            {
                slow = slow->next;
                fast = fast->next;
            }
        }
    }
}
```

```

/* 'slow' is before the midpoint in the list, so split it in two
at that point.*/
*frontRef = source;
*backRef = slow->next;
slow->next = NULL;
}

}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
while(node!=NULL)
{
printf("%d ",node->data);
node = node->next;
}
}

/* Function to insert a node at the beginning of the linked list */
void push(struct node** head_ref, int new_data)
{
/* allocate node */
struct node* new_node =
(struct node*) malloc(sizeof(struct node));

/* put in the data */
new_node->data = new_data;

/* link the old list off the new node */
new_node->next = (*head_ref);

/* move the head to point to the new node */
(*head_ref) = new_node;
}

/* Driver program to test above functions*/
int main()
{
/* Start with the empty list */
struct node* res = NULL;
struct node* a = NULL;

/* Let us create a unsorted linked lists to test the functions
Created lists shall be a: 2->3->20->5->10->15 */
push(&a, 15);
push(&a, 10);
push(&a, 5);
push(&a, 20);
push(&a, 3);
push(&a, 2);

/* Sort the above created Linked List */
MergeSort(&a);

printf("\n Sorted Linked List is: \n");
printList(a);

getchar();
return 0;
}

```

Time Complexity: $O(n\log n)$

Sources:

http://en.wikipedia.org/wiki/Merge_sort

<http://cslibrary.stanford.edu/105/LinkedListProblems.pdf>

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Linked Lists
Sorting
Merge Sort

Sort a nearly sorted (or K sorted) array

Given an array of n elements, where each element is at most k away from its target position, devise an algorithm that sorts in $O(n \log k)$ time.

For example, let us consider k is 2, an element at index 7 in the sorted array, can be at indexes 5, 6, 7, 8, 9 in the given array.

Source: [Nearly sorted algorithm](#)

We can **use Insertion Sort** to sort the elements efficiently. Following is the C code for standard Insertion Sort.

```

/* Function to sort an array using insertion sort/
void insertionSort(int A[], int size)
{
int i, key, j;
for (i = 1; i < size; i++)
{
    key = A[i];
    j = i-1;

    /* Move elements of A[0..i-1], that are greater than key, to one
       position ahead of their current position.
       This loop will run at most k times */
    while (j >= 0 && A[j] > key)
    {
        A[j+1] = A[j];
        j = j-1;
    }
    A[j+1] = key;
}
}

```

The inner loop will run at most k times. To move every element to its correct place, at most k elements need to be moved. So overall complexity will be $O(nk)$

We can sort such arrays **more efficiently with the help of Heap data structure**. Following is the detailed process that uses Heap.

1) Create a Min Heap of size $k+1$ with first $k+1$ elements. This will take $O(k)$ time (See this [GFact](#))

2) One by one remove min element from heap, put it in result array, and add a new element to heap from remaining elements.

Removing an element and adding a new element to min heap will take LogK time. So overall complexity will be $O(k) + O((n-k)*\log K)$

```
#include<iostream>
using namespace std;

// Prototype of a utility function to swap two integers
void swap(int *x, int *y);

// A class for Min Heap
class MinHeap
{
    int *harr; // pointer to array of elements in heap
    int heap_size; // size of min heap
public:
    // Constructor
    MinHeap(int arr[], int size);

    // to heapify a subtree with root at given index
    void MinHeapify(int i);

    // to get index of left child of node at index i
    int left(int i) { return (2*i + 1); }

    // to get index of right child of node at index i
    int right(int i) { return (2*i + 2); }

    // to remove min (or root), add a new value x, and return old root
    int replaceMin(int x);

    // to extract the root which is the minimum element
    int extractMin();
};

// Given an array of size n, where every element is k away from its target
// position, sorts the array in  $O(n \log k)$  time.
int sortK(int arr[], int n, int k)
{
    // Create a Min Heap of first ( $k+1$ ) elements from
    // input array
    int *harr = new int[k+1];
    for (int i = 0; i <= k && i < n; i++) // i < n condition is needed when k > n
        harr[i] = arr[i];
    MinHeap hp(harr, k+1);

    // i is index for remaining elements in arr[] and ti
    // is target index of current minimum element in
    // Min Heap 'hp'.
    for (int i = k+1, ti = 0; ti < n; i++, ti++)
    {
        // If there are remaining elements, then place
        // root of heap at target index and add arr[]
        // to Min Heap
        if (i < n)
            arr[ti] = hp.replaceMin(arr[i]);

        // Otherwise place root at its target index and
        // reduce heap size
        else
            arr[ti] = hp.extractMin();
    }
}

// FOLLOWING ARE IMPLEMENTATIONS OF STANDARD MIN HEAP METHODS FROM CORMEN BOOK
// Constructor: Builds a heap from a given array arr[] of given size
MinHeap::MinHeap(int arr[], int size)
{
    heap_size = size;
    harr = arr; // store address of array
    int i = (heap_size - 1)/2;
    while (i >= 0)
    {
        MinHeapify(i);
        i--;
    }
}

// Method to remove minimum element (or root) from min heap
int MinHeap::extractMin()
{
    int root = harr[0];
    if (heap_size > 1)
    {
        harr[0] = harr[heap_size-1];
        heap_size--;
        MinHeapify(0);
    }
    return root;
}

// Method to change root with given value x, and return the old root
int MinHeap::replaceMin(int x)
{
    int root = harr[0];
    harr[0] = x;
    if (root < x)
        MinHeapify(0);
    return root;
}

// A recursive method to heapify a subtree with root at given index
// This method assumes that the subtrees are already heapified
```

```

void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l] < harr[i])
        smallest = l;
    if (r < heap_size && harr[r] < harr[smallest])
        smallest = r;
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}

// A utility function to swap two elements
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// A utility function to print array elements
void printArray(int arr[], int size)
{
    for (int i=0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    int k = 3;
    int arr[] = {2, 6, 3, 12, 56, 8};
    int n = sizeof(arr)/sizeof(arr[0]);
    sortK(arr, n, k);

    cout << "Following is sorted array\n";
    printArray (arr, n);

    return 0;
}

```

Output:

```

Following is sorted array
2 3 6 8 12 56

```

The Min Heap based method takes O(nLogk) time and uses O(k) auxiliary space.

We can also use a **Balanced Binary Search Tree** instead of Heap to store K+1 elements. The **insert** and **delete** operations on Balanced BST also take O(Logk) time. So Balanced BST based method will also take O(nLogk) time, but the Heap based method seems to be more efficient as the minimum element will always be at root. Also, Heap doesn't need extra space for left and right pointers.

Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Heap
Sorting
Insertion Sort

Iterative Quick Sort

Following is a typical recursive implementation of **Quick Sort** that uses last element as pivot.

C++

```

/* A typical recursive C/C++ implementation of QuickSort */

/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
   array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
int partition (int arr[], int l, int h)
{
    int x = arr[h];
    int i = (l - 1);

    for (int j = l; j <= h - 1; j++)
    {
        if (arr[j] <= x)
        {
            i++;
            swap (&arr[i], &arr[j]);
        }
    }
    swap (&arr[i + 1], &arr[h]);
    return (i + 1);
}

/* A[] --> Array to be sorted,
   l --> Starting index,
   h --> Ending index */
void quickSort(int A[], int l, int h)
{
    if (l < h)
    {
        /* Partitioning index */

```

```

    int p = partition(A, l, h);
    quickSort(A, l, p - 1);
    quickSort(A, p + 1, h);
}
}

```

Python

```

# A typical recursive Python implementation of QuickSort *

# This function takes last element as pivot, places
# the pivot element at its correct position in sorted
# array, and places all smaller (smaller than pivot)
# to left of pivot and all greater elements to right
# of pivot
def partition(arr,low,high):
    i = (low-1)      # index of smaller element
    pivot = arr[high] # pivot

    for j in range(low , high):

        # If current element is smaller than or
        # equal to pivot
        if arr[j] <= pivot:

            # increment index of smaller element
            i = i+1
            arr[i],arr[j] = arr[j],arr[i]

    arr[i+1],arr[high] = arr[high],arr[i+1]
    return ( i+1 )

# The main function that implements QuickSort
# arr[] --> Array to be sorted,
# low --> Starting index,
# high --> Ending index

# Function to do Quick sort
def quickSort(arr,low,high):
    if low < high:

        # pi is partitioning index, arr[p] is now
        # at right place
        pi = partition(arr,low,high)

        # Separately sort elements before
        # partition and after partition
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)

# This code is contributed by Mohit Kumra

```

Java

```

// Java program for implementation of QuickSort
class QuickSort {
    /* This function takes last element as pivot,
     * places the pivot element at its correct
     * position in sorted array, and places all
     * smaller (smaller than pivot) to left of
     * pivot and all greater elements to right
     * of pivot */
    int partition(int arr[], int low, int high)
    {
        int pivot = arr[high];
        int i = (low-1); // index of smaller element
        for (int j=low; j<=high-1; j++)
        {
            // If current element is smaller than or
            // equal to pivot
            if (arr[j] <= pivot)
            {
                i++;
                // swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        // swap arr[i+1] and arr[high] (or pivot)
        int temp = arr[i+1];
        arr[i+1] = arr[high];
        arr[high] = temp;

        return i+1;
    }

    /* The main function that implements QuickSort()
     * arr[] --> Array to be sorted,
     * low --> Starting index,
     * high --> Ending index */
    void qSort(int arr[], int low, int high)
    {
        if (low < high)
        {
            /* pi is partitioning index, arr[pi] is
             * now at right place */
            int pi = partition(arr, low, high);

            // Separately sort elements before
            // partition and after partition
            qSort(arr, low, pi-1);
            qSort(arr, pi+1, high);
        }
    }
}

```

```

    // Recursively sort elements before
    // partition and after partition
    qSort(arr, low, pi-1);
    qSort(arr, pi+1, high);
}
}
}

```

The above implementation can be optimized in many ways

1) The above implementation uses last index as pivot. This causes worst-case behavior on already sorted arrays, which is a commonly occurring case. The problem can be solved by choosing either a random index for the pivot, or choosing the middle index of the partition or choosing the median of the first, middle and last element of the partition for the pivot. (See [this](#) for details)

2) To reduce the recursion depth, recur first for the smaller half of the array, and use a tail call to recurse into the other.

3) Insertion sort works better for small subarrays. Insertion sort can be used for invocations on such small arrays (i.e. where the length is less than a threshold t determined experimentally). For example, [this](#) library implementation of qsort uses insertion sort below size 7.

Despite above optimizations, the function remains recursive and uses [function call stack](#) to store intermediate values of l and h. The function call stack stores other bookkeeping information together with parameters. Also, function calls involve overheads like storing activation record of the caller function and then resuming execution.

The above function can be easily converted to iterative version with the help of an auxiliary stack. Following is an iterative implementation of the above recursive code.

C/C++

```

// An iterative implementation of quick sort
#include <stdio.h>

```

```

// A utility function to swap two elements
void swap ( int* a, int* b )
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* This function is same in both iterative and recursive*/
int partition ( int arr[], int l, int h )
{
    int x = arr[h];
    int i = (l - 1);

    for ( int j = l; j <= h - 1; j++ )
    {
        if ( arr[j] <= x )
        {
            i++;
            swap ( &arr[i], &arr[j] );
        }
    }
    swap ( &arr[i + 1], &arr[h] );
    return ( i + 1 );
}

/* A[] --> Array to be sorted,
   l --> Starting index,
   h --> Ending index */
void quickSortIterative ( int arr[], int l, int h )
{
    // Create an auxiliary stack
    int stack[ h - l + 1 ];

    // initialize top of stack
    int top = -1;

    // push initial values of l and h to stack
    stack[ ++top ] = l;
    stack[ ++top ] = h;

    // Keep popping from stack while is not empty
    while ( top >= 0 )
    {
        // Pop h and l
        h = stack[ top-- ];
        l = stack[ top-- ];

        // Set pivot element at its correct position
        // in sorted array
        int p = partition( arr, l, h );

        // If there are elements on left side of pivot,
        // then push left side to stack
        if ( p-1 > l )
        {
            stack[ ++top ] = l;
            stack[ ++top ] = p - 1;
        }

        // If there are elements on right side of pivot,
        // then push right side to stack
        if ( p+1 < h )
        {
            stack[ ++top ] = p + 1;
            stack[ ++top ] = h;
        }
    }

    // A utility function to print contents of arr
    void printArr( int arr[], int n )
    {
        int i;
        for ( i = 0; i < n; ++i )
            printf( "%d ", arr[i] );
    }
}

```

```
// Driver program to test above functions
int main()
{
    int arr[] = {4, 3, 5, 2, 1, 3, 2, 3};
    int n = sizeof( arr ) / sizeof( *arr );
    quickSortIterative( arr, 0, n - 1 );
    printArr( arr, n );
    return 0;
}
```

Python

```
# Python program for implementation of Quicksort

# This function is same in both iterative and recursive
def partition(arr,l,h):
    i = ( l - 1 )
    x = arr[h]

    for j in range(l , h):
        if arr[j] <= x:
            # increment index of smaller element
            i = i+1
            arr[i],arr[j] = arr[j],arr[i]

    arr[i+1],arr[h] = arr[h],arr[i+1]
    return (i+1)

# Function to do Quick sort
# arr[] --> Array to be sorted,
# l --> Starting index,
# h --> Ending index
def quickSortIterative(arr,l,h):

    # Create an auxiliary stack
    size = h - l + 1
    stack = [0] * (size)

    # initialize top of stack
    top = -1

    # push initial values of l and h to stack
    top = top + 1
    stack[top] = l
    top = top + 1
    stack[top] = h

    # Keep popping from stack while is not empty
    while top >= 0:

        # Pop h and l
        h = stack[top]
        top = top - 1
        l = stack[top]
        top = top - 1

        # Set pivot element at its correct position in
        # sorted array
        p = partition( arr, l, h )

        # If there are elements on left side of pivot,
        # then push left side to stack
        if p-1 > l:
            top = top + 1
            stack[top] = l
            top = top + 1
            stack[top] = p - 1

        # If there are elements on right side of pivot,
        # then push right side to stack
        if p+1 < h:
            top = top + 1
            stack[top] = p + 1
            top = top + 1
            stack[top] = h

    # Driver code to test above
    arr = [4, 3, 5, 2, 1, 3, 2, 3]
    n = len(arr)
    quickSortIterative(arr, 0, n-1)
    print ("Sorted array is:")
    for i in range(n):
        print ("%d" %arr[i]),

# This code is contributed by Mohit Kumra
```

Java

```
// Java implementation of iterative quick sort
class IterativeQuickSort
{
    void swap(int arr[],int i,int j)
    {
        int t = arr[i];
        arr[i] = arr[j];
        arr[j] = t;
    }

    /* This function is same in both iterative and
    recursive*/
}
```

```

int partition (int arr[], int l, int h)
{
    int x = arr[h];
    int i = (l - 1);

    for (int j = l; j <= h - 1; j++)
    {
        if (arr[j] <= x)
        {
            i++;
            // swap arr[i] and arr[j]
            swap(arr,i,j);
        }
    }
    // swap arr[i+1] and arr[h]
    swap(arr,i+1,h);
    return (i + 1);
}

// Sorts arr[l..h] using iterative QuickSort
void QuickSort(int arr[], int l, int h)
{
    // create auxiliary stack
    int stack[] = new int[h-l+1];

    // initialize top of stack
    int top = -1;

    // push initial values in the stack
    stack[++top] = l;
    stack[++top] = h;

    // keep popping elements until stack is not empty
    while (top >= 0)
    {
        // pop h and l
        h = stack[top--];
        l = stack[top--];

        // set pivot element at it's proper position
        int p = partition(arr, l, h);

        // If there are elements on left side of pivot,
        // then push left side to stack
        if (p-1 > l)
        {
            stack[ ++top ] = l;
            stack[ ++top ] = p - 1;
        }

        // If there are elements on right side of pivot,
        // then push right side to stack
        if (p+1 < h)
        {
            stack[ ++top ] = p + 1;
            stack[ ++top ] = h;
        }
    }
}

// A utility function to print contents of arr
void printArr( int arr[], int n )
{
    int i;
    for ( i = 0; i < n; ++i )
        System.out.print(arr[i]+ " ");
}

// Driver code to test above
public static void main(String args[])
{
    IterativeQuickSort ob = new IterativeQuickSort();
    int arr[] = {4, 3, 5, 2, 1, 3, 2, 3};
    ob.QuickSort(arr, 0, arr.length-1);
    ob.printArr(arr, arr.length);
}
*/
/*This code is contributed by Rajat Mishra */

```

Output:

1 2 2 3 3 3 4 5

The above mentioned optimizations for recursive quick sort can also be applied to iterative version.

- 1) Partition process is same in both recursive and iterative. The same techniques to choose optimal pivot can also be applied to iterative version.
- 2) To reduce the stack size, first push the indexes of smaller half.
- 3) Use insertion sort when the size reduces below a experimentally calculated threshold.

References:

<http://en.wikipedia.org/wiki/Quicksort>

This article is compiled by **Aashish Barnwal** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

QuickSort on Doubly Linked List is discussed [here](#). QuickSort on Singly linked list was given as an exercise. Following is C++ implementation for same. The important things about implementation are, it changes pointers rather swapping data and time complexity is same as the implementation for Doubly Linked List.

In `partition()`, we consider last element as pivot. We traverse through the current list and if a node has value greater than pivot, we move it after tail. If the node has smaller value, we keep it at its current position.

In `QuickSortRecur()`, we first call `partition()` which places pivot at correct position and returns pivot. After pivot is placed at correct position, we find tail node of left side (list before pivot) and recur for left list. Finally, we recur for right list.

```
// C++ program for Quick Sort on Singly Linled List
#include <iostream>
#include <csdio>
using namespace std;

/* a node of the singly linked list */
struct node
{
    int data;
    struct node *next;
};

/* A utility function to insert a node at the beginning of linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node = new node;

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* A utility function to print linked list */
void printList(struct node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
    printf("\n");
}

// Returns the last node of the list
struct node *getTail(struct node *cur)
{
    while (cur != NULL && cur->next != NULL)
        cur = cur->next;
    return cur;
}

// Partitions the list taking the last element as the pivot
struct node *partition(struct node *head, struct node *end,
                      struct node **newHead, struct node **newEnd)
{
    struct node *pivot = end;
    struct node *prev = NULL, *cur = head, *tail = pivot;

    // During partition, both the head and end of the list might change
    // which is updated in the newHead and newEnd variables
    while (cur != pivot)
    {
        if (cur->data < pivot->data)
        {
            // First node that has a value less than the pivot - becomes
            // the new head
            if ((*newHead) == NULL)
                (*newHead) = cur;

            prev = cur;
            cur = cur->next;
        }
        else // If cur node is greater than pivot
        {
            // Move cur node to next of tail, and change tail
            if (prev)
                prev->next = cur->next;
            struct node *tmp = cur->next;
            cur->next = NULL;
            tail->next = cur;
            tail = cur;
            cur = tmp;
        }
    }

    // If the pivot data is the smallest element in the current list,
    // pivot becomes the head
    if ((*newHead) == NULL)
        (*newHead) = pivot;

    // Update newEnd to the current last node
    (*newEnd) = tail;

    // Return the pivot node
    return pivot;
}

//here the sorting happens exclusive of the end node
struct node *quickSortRecur(struct node *head, struct node *end)
{
    // base condition
```

```

if (!head || head == end)
    return head;

node *newHead = NULL, *newEnd = NULL;

// Partition the list, newHead and newEnd will be updated
// by the partition function
struct node *pivot = partition(head, end, &newHead, &newEnd);

// If pivot is the smallest element - no need to recur for
// the left part.
if (newHead != pivot)
{
    // Set the node before the pivot node as NULL
    struct node *tmp = newHead;
    while (tmp->next != pivot)
        tmp = tmp->next;
    tmp->next = NULL;

    // Recur for the list before pivot
    newHead = quickSortRecur(newHead, tmp);

    // Change next of last node of the left half to pivot
    tmp = getTail(newHead);
    tmp->next = pivot;
}

// Recur for the list after the pivot element
pivot->next = quickSortRecur(pivot->next, newEnd);

return newHead;
}

// The main function for quick sort. This is a wrapper over recursive
// function quickSortRecur()
void quickSort(struct node **headRef)
{
    (*headRef) = quickSortRecur(*headRef, getTail(*headRef));
    return;
}

// Driver program to test above functions
int main()
{
    struct node *a = NULL;
    push(&a, 5);
    push(&a, 20);
    push(&a, 4);
    push(&a, 3);
    push(&a, 30);

    cout << "Linked List before sorting \n";
    printList(a);

    quickSort(&a);

    cout << "Linked List after sorting \n";
    printList(a);

    return 0;
}

```

Output:

```

Linked List before sorting
30 3 4 20 5
Linked List after sorting
3 4 5 20 30

```

This article is contributed by [Balasubramanian.N](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

[Linked Lists](#)
[Sorting](#)
[Quick Sort](#)

QuickSort on Doubly Linked List

Following is a typical recursive implementation of [QuickSort](#) for arrays. The implementation uses last element as pivot.

```

/* A typical recursive implementation of Quicksort for array*/

/* This function takes last element as pivot, places the pivot element at its
correct position in sorted array, and places all smaller (smaller than
pivot) to left of pivot and all greater elements to right of pivot */
int partition (int arr[], int l, int h)
{
    int x = arr[h];
    int i = (l - 1);

    for (int j = l; j <= h - 1; j++)
    {
        if (arr[j] <= x)
        {
            i++;
            swap (&arr[i], &arr[j]);
        }
    }
    swap (&arr[i + 1], &arr[h]);
    return (i + 1);
}

/* A[] --> Array to be sorted, l --> Starting index, h --> Ending index */

```

```

void quickSort(int A[], int l, int h)
{
    if (l < h)
    {
        int p = partition(A, l, h); /* Partitioning index */
        quickSort(A, l, p - 1);
        quickSort(A, p + 1, h);
    }
}

```

Can we use same algorithm for Linked List?

Following is C++ implementation for doubly linked list. The idea is simple, we first find out pointer to last node. Once we have pointer to last node, we can recursively sort the linked list using pointers to first and last nodes of linked list, similar to the above recursive function where we pass indexes of first and last array elements. The partition function for linked list is also similar to partition for arrays. Instead of returning index of the pivot element, it returns pointer to the pivot element. In the following implementation, quickSort() is just a wrapper function, the main recursive function is _quickSort() which is similar to quickSort() for array implementation.



C++

```

// A C++ program to sort a linked list using Quicksort
#include <iostream>
#include <stdio.h>
using namespace std;

/* a node of the doubly linked list */
struct node
{
    int data;
    struct node *next;
    struct node *prev;
};

/* A utility function to swap two elements */
void swap ( int* a, int* b )
{ int t = *a; *a = *b; *b = t; }

/* A utility function to find last node of linked list */
struct node *lastNode(node *root)
{
    while (root && root->next)
        root = root->next;
    return root;
}

/* Considers last element as pivot, places the pivot element at its
   correct position in sorted array, and places all smaller (smaller than
   pivot) to left of pivot and all greater elements to right of pivot */
node* partition(node *l, node *h)
{
    // set pivot as h element
    int x = h->data;

    // similar to i = l-1 for array implementation
    node *i = l->prev;

    // Similar to "for (int j = l; j <= h- 1; j++)"
    for (node *j = l; j != h; j = j->next)
    {
        if (j->data <= x)
        {
            // Similar to i++ for array
            i = (i == NULL)? l : i->next;

            swap(&(i->data), &(j->data));
        }
    }
    i = (i == NULL)? l : i->next; // Similar to i++
    swap(&(i->data), &(h->data));
    return i;
}

/* A recursive implementation of quicksort for linked list */
void _quickSort(struct node* l, struct node *h)
{
    if (h != NULL && l != h && l != h->next)
    {
        struct node *p = partition(l, h);
        _quickSort(l, p->prev);
        _quickSort(p->next, h);
    }
}

// The main function to sort a linked list. It mainly calls _quickSort()
void quickSort(struct node *head)
{
    // Find last node
    struct node *h = lastNode(head);

    // Call the recursive QuickSort
    _quickSort(head, h);
}

// A utility function to print contents of arr
void printList(struct node *head)
{
    while (head)
    {
        cout << head->data << " ";
        head = head->next;
    }
}

```

```

    }
    cout << endl;
}

/* Function to insert a node at the begining of the Doubly Linked List */
void push(struct node** head_ref, int new_data)
{
    struct node* new_node = new node; /* allocate node */
    new_node->data = new_data;

    /* since we are adding at the begining, prev is always NULL */
    new_node->prev = NULL;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* change prev of head node to new node */
    if ((*head_ref) != NULL) (*head_ref)->prev = new_node;

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Driver program to test above function */
int main()
{
    struct node *a = NULL;
    push(&a, 5);
    push(&a, 20);
    push(&a, 4);
    push(&a, 3);
    push(&a, 30);

    cout << "Linked List before sorting \n";
    printList(a);

    quickSort(a);

    cout << "Linked List after sorting \n";
    printList(a);

    return 0;
}

```

Java

```

// A Java program to sort a linked list using Quicksort
class QuickSort_Using_Doubly_LinkedList{
    Node head;

    /* a node of the doubly linked list */
    static class Node{
        private int data;
        private Node next;
        private Node prev;

        Node(int d){
            data = d;
            next = null;
            prev = null;
        }
    }

    // A utility function to find last node of linked list
    Node lastNode(Node node){
        while(node.next!=null)
            node = node.next;
        return node;
    }

    /* Considers last element as pivot, places the pivot element at its
    correct position in sorted array, and places all smaller (smaller than
    pivot) to left of pivot and all greater elements to right of pivot */
    Node partition(Node l,Node h)
    {
        // set pivot as h element
        int x = h.data;

        // similar to i = l-1 for array implementation
        Node i = l.prev;

        // Similar to "for (int j = l; j <= h- 1; j++)"
        for(Node j=l; j!=h; j=j.next)
        {
            if(j.data <= x)
            {
                // Similar to i++ for array
                i = (i==null) ? l : i.next;
                int temp = i.data;
                i.data = j.data;
                j.data = temp;
            }
            i = (i==null) ? l : i.next; // Similar to i++
            int temp = i.data;
            i.data = h.data;
            h.data = temp;
        }
        return i;
    }

    /* A recursive implementation of quicksort for linked list */
    void _quickSort(Node l,Node h)
    {

```

```

if(h==null && l==h && l==h.next){
    Node temp = partition(l,h);
    _quickSort(l,temp.prev);
    _quickSort(temp.next,h);
}
}

// The main function to sort a linked list. It mainly calls _quickSort()
public void quickSort(Node node)
{
    // Find last node
    Node head = lastNode(node);

    // Call the recursive QuickSort
    _quickSort(node.head);
}

// A utility function to print contents of arr
public void printList(Node head)
{
    while(head!=null){
        System.out.print(head.data+" ");
        head = head.next;
    }
}

/* Function to insert a node at the beginning of the Doubly Linked List */
void push(int new_Data)
{
    Node new_Node = new Node(new_Data); /* allocate node */

    // if head is null, head = new_Node
    if(head==null){
        head = new_Node;
        return;
    }

    /* link the old list off the new node */
    new_Node.next = head;

    /* change prev of head node to new node */
    head.prev = new_Node;

    /* since we are adding at the begining, prev is always NULL */
    new_Node.prev = null;

    /* move the head to point to the new node */
    head = new_Node;
}

/* Driver program to test above function */
public static void main(String[] args){
    QuickSort_using_Doubly_LinkedList list = new QuickSort_using_Doubly_LinkedList();

    list.push(5);
    list.push(20);
    list.push(4);
    list.push(3);
    list.push(30);

    System.out.println("Linked List before sorting ");
    list.printList(list.head);
    System.out.println("\nLinked List after sorting");
    list.quickSort(list.head);
    list.printList(list.head);

}
}

// This code has been contributed by Amit Khandelwal

```

Output :

```

Linked List before sorting
30 3 4 20 5
Linked List after sorting
3 4 5 20 30

```

Time Complexity: Time complexity of the above implementation is same as time complexity of QuickSort() for arrays. It takes $O(n^2)$ time in worst case and $O(n\log n)$ in average and best cases. The worst case occurs when the linked list is already sorted.

Can we implement random quick sort for linked list?

Quicksort can be implemented for Linked List only when we can pick a fixed point as pivot (like last element in above implementation). Random QuickSort cannot be efficiently implemented for Linked Lists by picking random pivot.

Exercise:

The above implementation is for doubly linked list. Modify it for singly linked list. Note that we don't have prev pointer in singly linked list.

Refer [QuickSort on Singly Linked List](#) for solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Linked Lists
Sorting
Quick Sort

Find k closest elements to a given value

Given a sorted array arr[] and a value X, find the k closest elements to X in arr[].

Examples:

```
Input: K = 4, X = 35
arr[] = {12, 16, 22, 30, 35, 39, 42,
        45, 48, 50, 53, 55, 56}
Output: 30 39 42 45
```

Note that if the element is present in array, then it should not be in output, only the other closest elements are required.

In the following solutions, it is assumed that all elements of array are distinct.

A **simple solution** is to do linear search for k closest elements.

- 1) Start from the first element and search for the crossover point (The point before which elements are smaller than or equal to X and after which elements are greater). This step takes O(n) time.
- 2) Once we find the crossover point, we can compare elements on both sides of crossover point to print k closest elements. This step takes O(k) time.

The time complexity of the above solution is O(n).

An **Optimized Solution** is to find k elements in O(Logn + k) time. The idea is to use [Binary Search](#) to find the crossover point. Once we find index of crossover point, we can print k closest elements in O(k) time.

C/C++

```
#include<stdio.h>

/* Function to find the cross over point (the point before
   which elements are smaller than or equal to x and after
   which greater than x)*/
int findCrossOver(int arr[], int low, int high, int x)
{
    // Base cases
    if (arr[high] <= x) // x is greater than all
        return high;
    if (arr[low] > x) // x is smaller than all
        return low;

    // Find the middle point
    int mid = (low + high)/2; /* low + (high - low)/2 */

    /* If x is same as middle element, then return mid */
    if (arr[mid] <= x && arr[mid+1] > x)
        return mid;

    /* If x is greater than arr[mid], then either arr[mid + 1]
       is ceiling of x or ceiling lies in arr[mid+1...high] */
    if (arr[mid] < x)
        return findCrossOver(arr, mid+1, high, x);

    return findCrossOver(arr, low, mid - 1, x);
}

// This function prints k closest elements to x in arr[].
// n is the number of elements in arr[]
void printKclosest(int arr[], int x, int k, int n)
{
    // Find the crossover point
    int l = findCrossOver(arr, 0, n-1, x);
    int r = l+1; // Right index to search
    int count = 0; // To keep track of count of elements already printed

    // If x is present in arr[], then reduce left index
    // Assumption: all elements in arr[] are distinct
    if (arr[l] == x) l--;

    // Compare elements on left and right of crossover
    // point to find the k closest elements
    while (l >= 0 && r < n && count < k)
    {
        if (x - arr[l] < arr[r] - x)
            printf("%d ", arr[l--]);
        else
            printf("%d ", arr[r++]);
        count++;
    }

    // If there are no more elements on right side, then
    // print left elements
    while (count < k && l >= 0)
        printf("%d ", arr[l--]), count++;

    // If there are no more elements on left side, then
    // print right elements
    while (count < k && r < n)
        printf("%d ", arr[r++]), count++;
}

/* Driver program to check above functions */
int main()
{
    int arr[] = {12, 16, 22, 30, 35, 39, 42,
                45, 48, 50, 53, 55, 56};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x = 35, k = 4;
    printKclosest(arr, x, 4, n);
    return 0;
}
```

Java

```
// Java program to find k closest elements to a given value
class KClosest
{
    /* Function to find the cross over point (the point before
```

```

which elements are smaller than or equal to x and after
which greater than x*/
int findCrossOver(int arr[], int low, int high, int x)
{
    // Base cases
    if (arr[high] <= x) // x is greater than all
        return high;
    if (arr[low] > x) // x is smaller than all
        return low;

    // Find the middle point
    int mid = (low + high)/2; /* low + (high - low)/2 */

    /* If x is same as middle element, then return mid */
    if (arr[mid] <= x && arr[mid+1] > x)
        return mid;

    /* If x is greater than arr[mid], then either arr[mid + 1]
     * is ceiling of x or ceiling lies in arr[mid+1...high] */
    if(arr[mid] < x)
        return findCrossOver(arr, mid+1, high, x);

    return findCrossOver(arr, low, mid - 1, x);
}

// This function prints k closest elements to x in arr[].
// n is the number of elements in arr[]
void printKclosest(int arr[], int x, int k, int n)
{
    // Find the crossover point
    int l = findCrossOver(arr, 0, n-1, x);
    int r = l+1; // Right index to search
    int count = 0; // To keep track of count of elements
    // already printed

    // If x is present in arr[], then reduce left index
    // Assumption: all elements in arr[] are distinct
    if (arr[l] == x) l--;

    // Compare elements on left and right of crossover
    // point to find the k closest elements
    while (l >= 0 && r < n && count < k)
    {
        if (x - arr[l] < arr[r] - x)
            System.out.print(arr[l--] + " ");
        else
            System.out.print(arr[r++] + " ");
        count++;
    }

    // If there are no more elements on right side, then
    // print left elements
    while (count < k && l >= 0)
    {
        System.out.print(arr[l--] + " ");
        count++;
    }
}

// If there are no more elements on left side, then
// print right elements
while (count < k && r < n)
{
    System.out.print(arr[r++] + " ");
    count++;
}

/* Driver program to check above functions */
public static void main(String args[])
{
    KCloset ob = new KCloset();
    int arr[] = {12, 16, 22, 30, 35, 39, 42,
                45, 48, 50, 53, 55, 56
               };
    int n = arr.length;
    int x = 35, k = 4;
    ob.printKclosest(arr, x, 4, n);
}
*/
/* This code is contributed by Rajat Mishra */

```

Output:

39 30 42 45

The time complexity of this method is O(Logn + k).

Exercise: Extend the optimized solution to work for duplicates also, i.e., to work for arrays where elements don't have to be distinct.

This article is contributed by **Rahul Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Searching

Sort n numbers in range from 0 to $n^2 - 1$ in linear time

Given an array of numbers of size n. It is also given that the array elements are in range from 0 to $n^2 - 1$. Sort the given array in linear time.

Examples:

Since there are 5 elements, the elements can be from 0 to 24.

Input: arr[] = {0, 23, 14, 12, 9}

Output: arr[] = {0, 9, 12, 14, 23}

Since there are 3 elements, the elements can be from 0 to 8.

Input: arr[] = {7, 0, 2}

Output: arr[] = {0, 2, 7}

We strongly recommend to minimize the browser and try this yourself first.

Solution: If we use Counting Sort, it would take $O(n^2)$ time as the given range is of size n^2 . Using any comparison based sorting like Merge Sort, Heap Sort, .. etc would take $O(n \log n)$ time.

Now question arises how to do this in $O(n)$? Firstly, is it possible? Can we use data given in question? n numbers in range from 0 to $n^2 - 1$?

The idea is to use Radix Sort. Following is standard Radix Sort algorithm.

- 1) Do following for each digit i where i varies from least significant digit to the most significant digit.
.....
.....a) Sort input array using counting sort (or any stable sort) according to the i 'th digit

Let there be d digits in input integers. Radix Sort takes $O(d*(n+b))$ time where b is the base for representing numbers, for example, for decimal system, b is 10. Since n^2-1 is the maximum possible value, the value of d would be $O(\log_b(n))$. So overall time complexity is $O((n+b)*O(\log_b(n)))$. Which looks more than the time complexity of comparison based sorting algorithms for a large k . The idea is to change base b . If we set b as n , the value of $O(\log_b(n))$ becomes $O(1)$ and overall time complexity becomes $O(n)$.

arr[] = {0, 10, 13, 12, 7}

Let us consider the elements in base 5. For example 13 in base 5 is 23, and 7 in base 5 is 12.

arr[] = {00(0), 20(10), 23(13), 22(12), 12(7)}

After first iteration (Sorting according to the last digit in base 5), we get.

arr[] = {00(0), 20(10), 12(7), 22(12), 23(13)}

After second iteration, we get

arr[] = {00(0), 12(7), 20(10), 22(12), 23(13)}

Following is C++ implementation to sort an array of size n where elements are in range from 0 to $n^2 - 1$.

C++

```
#include<iostream>
using namespace std;

// A function to do counting sort of arr[] according to
// the digit represented by exp.
int countSort(int arr[], int n, int exp)
{
    int output[n]; // output array
    int i, count[n];
    for (int i=0; i < n; i++)
        count[i] = 0;

    // Store count of occurrences in count[]
    for (i = 0; i < n; i++)
        count[(arr[i]/exp)%n]++;
    
    // Change count[i] so that count[i] now contains actual
    // position of this digit in output[]
    for (i = 1; i < n; i++)
        count[i] += count[i - 1];

    // Build the output array
    for (i = n - 1; i >= 0; i--)
    {
        output[count[(arr[i]/exp)%n] - 1] = arr[i];
        count[(arr[i]/exp)%n]--;
    }

    // Copy the output array to arr[], so that arr[] now
    // contains sorted numbers according to current digit
    for (i = 0; i < n; i++)
        arr[i] = output[i];
    }

// The main function to that sorts arr[] of size n using Radix Sort
void sort(int arr[], int n)
{
    // Do counting sort for first digit in base n. Note that
    // instead of passing digit number, exp (n^0 = 0) is passed.
    countSort(arr, n, 1);

    // Do counting sort for second digit in base n. Note that
    // instead of passing digit number, exp (n^1 = n) is passed.
    countSort(arr, n, n);
}

// A utility function to print an array
void printArr(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
}

// Driver program to test above functions
int main()
{
    // Since array size is 7, elements should be from 0 to 48
    int arr[] = {40, 12, 45, 32, 31, 1, 22};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Given array is \n";
    printArr(arr, n);

    sort(arr, n);
}
```

```

cout << "nSorted array is 'n";
printArr(arr, n);
return 0;
}

```

Java

```

// Java program to sort an array of size n where elements are
// in range from 0 to n^2 - 1.
class Sort1ToN2
{
    // A function to do counting sort of arr[] according to
    // the digit represented by exp.
    void countSort(int arr[], int n, int exp)
    {
        int output[] = new int[n]; // output array
        int i, count[] = new int[n];
        for (i=0; i < n; i++)
            count[i] = 0;

        // Store count of occurrences in count[]
        for (i = 0; i < n; i++)
            count[(arr[i]/exp)%n]++;
        
        // Change count[i] so that count[i] now contains actual
        // position of this digit in output[]
        for (i = 1; i < n; i++)
            count[i] += count[i - 1];

        // Build the output array
        for (i = n - 1; i >= 0; i--)
        {
            output[count[(arr[i]/exp)%n] - 1] = arr[i];
            count[(arr[i]/exp)%n]--;
        }

        // Copy the output array to arr[], so that arr[] now
        // contains sorted numbers according to current digit
        for (i = 0; i < n; i++)
            arr[i] = output[i];
    }

    // The main function to that sorts arr[] of size n using Radix Sort
    void sort(int arr[], int n)
    {
        // Do counting sort for first digit in base n. Note that
        // instead of passing digit number, exp (n^0 = 0) is passed.
        countSort(arr, n, 1);

        // Do counting sort for second digit in base n. Note that
        // instead of passing digit number, exp (n^1 = n) is passed.
        countSort(arr, n, n);
    }

    // A utility function to print an array
    void printArr(int arr[], int n)
    {
        for (int i = 0; i < n; i++)
            System.out.print(arr[i] + " ");
    }

    // Driver program to test above functions
    public static void main(String args[])
    {
        Sort1ToN2 ob = new Sort1ToN2();

        // Since array size is 7, elements should be from 0 to 48
        int arr[] = {40, 12, 45, 32, 33, 1, 22};
        int n = arr.length;
        System.out.println("Given array");
        ob.printArr(arr, n);

        ob.sort(arr, n);

        System.out.println("Sorted array");
        ob.printArr(arr, n);
    }
}

/*This code is contributed by Rajat Mishra */

```

Output:

```

Given array
40 12 45 32 33 1 22
Sorted array is
1 12 22 32 33 40 45

```

How to sort if range is from 1 to n²?

If range is from 1 to n^2 , the above process can not be directly applied, it must be changed. Consider $n = 100$ and range from 1 to 10000. Since the base is 100, a digit must be from 0 to 99 and there should be 2 digits in the numbers. But the number 10000 has more than 2 digits. So to sort numbers in a range from 1 to n^2 , we can use following process.

- 1) Subtract all numbers by 1.
- 2) Since the range is now 0 to n^2 , do counting sort twice as done in the above implementation.
- 3) After the elements are sorted, add 1 to all numbers to obtain the original numbers.

How to sort if range is from 0 to $n^3 - 1$?

Since there can be 3 digits in base n, we need to call counting sort 3 times.

This article is contributed by **Bateesh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

A Problem in Many Binary Search Implementations

Consider the following C implementation of [Binary Search](#) function, is there anything wrong in this?

```
// A iterative binary search function. It returns location of x in
// given array arr[l..r] if present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r)
    {
        // find index of middle element
        int m = (l+r)/2;

        // Check if x is present at mid
        if (arr[m] == x) return m;

        // If x greater, ignore left half
        if (arr[m] < x) l = m + 1;

        // If x is smaller, ignore right half
        else r = m - 1;
    }

    // if we reach here, then element was not present
    return -1;
}
```

The above looks fine except one subtle thing, the expression “ $m = (l+r)/2$ ”. It fails for large values of l and r . Specifically, it fails if the sum of low and high is greater than the maximum positive int value ($2^{31} - 1$). The sum overflows to a negative value, and the value stays negative when divided by two. In C this causes an array index out of bounds with unpredictable results.

What is the way to resolve this problem?

Following is one way:

```
int mid = low + ((high - low) / 2);
```

Probably faster, and arguably as clear is (works only in Java, refer [this](#)):

```
int mid = (low + high) >>> 1;
```

In C and C++ (where you don't have the $>>>$ operator), you can do this:

```
mid = ((unsigned int)low + (unsigned int)high)) >> 1
```

The similar problem appears in [Merge Sort](#) as well.

The above content is taken from [google research blog](#).

Please refer [this](#) as well, it points out that the above solutions may not always work.

The above problem occurs when array length is 2^{30} or greater and the search repeatedly moves to second half of the array. This much size of array is not likely to appear most of the time. For example, when we try the below program with 32 bit [Code Blocks](#) compiler, we get compiler error.

```
int main()
{
    int arr[1<<30];
    return 0;
}
```

Output:

```
error: size of array 'arr' is too large
```

Even when we try boolean array, the program compiles fine, but crashes when run in Windows 7.0 and [Code Blocks](#) 32 bit compiler

```
#include <stdbool.h>
int main()
{
    bool arr[1<<30];
    return 0;
}
```

Output: No compiler error, but crashes at run time.

Sources:

<http://googleresearch.blogspot.in/2006/06/extra-extra-read-all-about-it-nearly.html>
http://locklessinc.com/articles/binary_search/

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Search in an almost sorted array

Given an array which is sorted, but after sorting some elements are moved to either of the adjacent positions, i.e., $arr[i]$ may be present at $arr[i+1]$ or $arr[i-1]$. Write an efficient function to search an element in this array. Basically the element $arr[i]$ can only be swapped with either $arr[i+1]$ or $arr[i-1]$.

For example consider the array {2, 3, 10, 4, 40}, 4 is moved to next position and 10 is moved to previous position.

Example:

```
Input: arr[] = {10, 3, 40, 20, 50, 80, 70}, key = 40
```

```

Output: 2
Output is index of 40 in given array

Input: arr[] = {10, 3, 40, 20, 50, 80, 70}, key = 90
Output: -1
-1 is returned to indicate element is not present

```

A simple solution is to linearly search the given key in given array. Time complexity of this solution is $O(n)$. We can modify binary search to do it in $O(\log n)$ time.

The idea is to compare the key with middle 3 elements, if present then return the index. If not present, then compare the key with middle element to decide whether to go in left half or right half. Comparing with middle element is enough as all the elements after mid+2 must be greater than element mid and all elements before mid-2 must be smaller than mid element.

Following is C++ implementation of this approach.

C++

```

// C++ program to find an element in an almost sorted
// array
#include <stdio.h>

// A recursive binary search based function. It returns
// index of x in given array arr[l..r] is present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l)
    {
        int mid = l + (r - l)/2;

        // If the element is present at one of the middle
        // 3 positions
        if (arr[mid] == x) return mid;
        if (mid > 1 && arr[mid-1] == x) return (mid - 1);
        if (mid < r && arr[mid+1] == x) return (mid + 1);

        // If element is smaller than mid, then it can only
        // be present in left subarray
        if (arr[mid] > x) return binarySearch(arr, l, mid-2, x);

        // Else the element can only be present in right subarray
        return binarySearch(arr, mid+2, r, x);
    }

    // We reach here when element is not present in array
    return -1;
}

// Driver program to test above function
int main(void)
{
    int arr[] = {3, 2, 10, 4, 40};
    int n = sizeof(arr)/ sizeof(arr[0]);
    int x = 4;
    int result = binarySearch(arr, 0, n-1, x);
    (result == -1)? printf("Element is not present in array")
                  : printf("Element is present at index %d", result);
    return 0;
}

```

Java

```

// Java program to find an element in an almost sorted array
class SearchAlmost
{
    // A recursive binary search based function. It returns
    // index of x in given array arr[l..r] is present,
    // otherwise -1
    int binarySearch(int arr[], int l, int r, int x)
    {
        if (r >= l)
        {
            int mid = l + (r - l)/2;

            // If the element is present at one of the middle
            // 3 positions
            if (arr[mid] == x) return mid;
            if (mid > 1 && arr[mid-1] == x) return (mid - 1);
            if (mid < r && arr[mid+1] == x) return (mid + 1);

            // If element is smaller than mid, then it can
            // only be present in left subarray
            if (arr[mid] > x) return binarySearch(arr, l, mid-2, x);

            // Else the element can only be present in right
            // subarray
            return binarySearch(arr, mid+2, r, x);
        }

        // We reach here when element is not present in array
        return -1;
    }

    // Driver method
    public static void main(String args[])
    {
        abc ob = new abc();
        int arr[] = {3, 2, 10, 4, 40};
        int n = arr.length;
        int x = 4;
        int result = ob.binarySearch(arr, 0, n-1, x);
        if(result == -1)
            System.out.println("Element is not present in array");
    }
}

```

```

        else
            System.out.println("Element is present at index " +
                               result);
    }
}

/*This code is contributed by Rajat Mishra*/

```

Output:

```
Element is present at index 3
```

Time complexity of the above function is O(Logn).

This article is contributed by **Abhishek**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Searching

Sort an array in wave form

Given an unsorted array of integers, sort the array into a wave like array. An array 'arr[0..n-1]' is sorted in wave form if arr[0] >= arr[1] = arr[3] =

Examples:

```

Input: arr[] = {10, 5, 6, 3, 2, 20, 100, 80}
Output: arr[] = {10, 5, 6, 2, 20, 3, 100, 80} OR
        {20, 5, 10, 2, 80, 6, 100, 3} OR
        any other array that is in wave form

Input: arr[] = {20, 10, 8, 6, 4, 2}
Output: arr[] = {20, 8, 10, 4, 6, 2} OR
        {10, 8, 20, 2, 6, 4} OR
        any other array that is in wave form

Input: arr[] = {2, 4, 6, 8, 10, 20}
Output: arr[] = {4, 2, 8, 6, 20, 10} OR
        any other array that is in wave form

Input: arr[] = {3, 6, 5, 10, 7, 20}
Output: arr[] = {6, 3, 10, 5, 20, 7} OR
        any other array that is in wave form

```

A Simple Solution is to use sorting. First sort the input array, then swap all adjacent elements.

For example, let the input array be {3, 6, 5, 10, 7, 20}. After sorting, we get {3, 5, 6, 7, 10, 20}. After swapping adjacent elements, we get {5, 3, 7, 6, 20, 10}.

Below are implementations of this simple approach.

C++

```

// A C++ program to sort an array in wave form using
// a sorting function
#include<iostream>
#include<algorithm>
using namespace std;

// A utility method to swap two numbers.
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// This function sorts arr[0..n-1] in wave form, i.e.,
// arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= arr[5]...
void sortInWave(int arr[], int n)
{
    // Sort the input array
    sort(arr, arr+n);

    // Swap adjacent elements
    for (int i=0; i<n-1; i+= 2)
        swap(&arr[i], &arr[i+1]);
}

// Driver program to test above function
int main()
{
    int arr[] = {10, 90, 49, 2, 1, 5, 23};
    int n = sizeof(arr)/sizeof(arr[0]);
    sortInWave(arr, n);
    for (int i=0; i<n; i++)
        cout << arr[i] << " ";
    return 0;
}

```

Python

```

# Python function to sort the array arr[0..n-1] in wave form,
# i.e., arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= arr[5]
def sortInWave(arr, n):

    #sort the array
    arr.sort()

    # Swap adjacent elements
    for i in range(0,n-1,2):

```

```

arr[i], arr[i+1] = arr[i+1], arr[i]

# Driver program
arr = [10, 90, 49, 2, 1, 5, 23]
sortInWave(arr, len(arr))
for i in range(0,len(arr)):
    print arr[i],

```

This code is contributed by __Devesh Agrawal__

Java

```

// Java implementation of naive method for sorting
// an array in wave form.
import java.util.*;

class SortWave
{
    // A utility method to swap two numbers.
    void swap(int arr[], int a, int b)
    {
        int temp = arr[a];
        arr[a] = arr[b];
        arr[b] = temp;
    }

    // This function sorts arr[0..n-1] in wave form, i.e.,
    // arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4]...
    void sortInWave(int arr[], int n)
    {
        // Sort the input array
        Arrays.sort(arr);

        // Swap adjacent elements
        for (int i=0; i<n-1; i+= 2)
            swap(arr, i, i+1);
    }

    // Driver method
    public static void main(String args[])
    {
        Test ob = new Test();
        int arr[] = {10, 90, 49, 2, 1, 5, 23};
        int n = arr.length;
        ob.sortInWave(arr, n);
        for (int i : arr)
            System.out.print(i + " ");
    }
}

```

/*This code is contributed by Rajat Mishra*/

Output:

```
2 1 10 5 49 23 90
```

The time complexity of the above solution is $O(n\log n)$ if a $O(n\log n)$ sorting algorithm like [Merge Sort](#), [Heap Sort](#), .. etc is used.

This can be done in **$O(n)$ time by doing a single traversal** of given array. The idea is based on the fact that if we make sure that all even positioned (at index 0, 2, 4, ..) elements are greater than their adjacent odd elements, we don't need to worry about odd positioned element. Following are simple steps.

1) Traverse all even positioned elements of input array, and do following.

....a) If current element is smaller than previous odd element, swap previous and current.

....b) If current element is smaller than next odd element, swap next and current.

Below are implementations of above simple algorithm.

C++

```

// A O(n) program to sort an input array in wave form
#include<iostream>
using namespace std;

// A utility method to swap two numbers.
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// This function sorts arr[0..n-1] in wave form, i.e., arr[0] >=
// arr[1] <= arr[2] >= arr[3] <= arr[4] >= arr[5] ....
void sortInWave(int arr[], int n)
{
    // Traverse all even elements
    for (int i = 0; i < n; i+=2)
    {
        // If current even element is smaller than previous
        if ((i>0 && arr[i-1] > arr[i]))
            swap(&arr[i], &arr[i-1]);

        // If current even element is smaller than next
        if (i<n-1 && arr[i] < arr[i+1])
            swap(&arr[i], &arr[i + 1]);
    }
}

// Driver program to test above function
int main()
{
    int arr[] = {10, 90, 49, 2, 1, 5, 23};
    int n = sizeof(arr)/sizeof(arr[0]);
}

```

```

sortInWave(arr, n);
for (int i=0;i<n;i++)
    cout << arr[i] << " ";
return 0;
}

```

Python

```

# Python function to sort the array arr[0..n-1] in wave form,
# i.e., arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= arr[5]
def sortInWave(arr, n):

    # Traverse all even elements
    for i in range(0, n, 2):

        # If current even element is smaller than previous
        if ((i>0 and arr[i]<arr[i-1])):
            arr[i],arr[i-1] = arr[i-1],arr[i]

        # If current even element is smaller than next
        if ((i < n-1 and arr[i]<arr[i+1])):
            arr[i],arr[i+1] = arr[i+1],arr[i]

    # Driver program
arr = [10,90,49,2,1,5,23]
sortInWave(arr, len(arr))
for i in range(0,len(arr)):
    print arr[i],

```

This code is contributed by __Devesh Agrawal__

Java

```

// A O(n) Java program to sort an input array in wave form
class SortWave
{
    // A utility method to swap two numbers.
    void swap(int arr[], int a, int b)
    {
        int temp = arr[a];
        arr[a] = arr[b];
        arr[b] = temp;
    }

    // This function sorts arr[0..n-1] in wave form, i.e.,
    // arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4]....
    void sortInWave(int arr[], int n)
    {
        // Traverse all even elements
        for (int i = 0; i < n; i+=2)
        {
            // If current even element is smaller
            // than previous
            if ((i>0 && arr[i-1] > arr[i]))
                swap(arr, i-1, i);

            // If current even element is smaller
            // than next
            if ((i<n-1 && arr[i] < arr[i+1]))
                swap(arr, i, i + 1);
        }
    }

    // Driver program to test above function
    public static void main(String args[])
    {
        SortWave ob = new SortWave();
        int arr[] = {10, 90, 49, 2, 1, 5, 23};
        int n = arr.length;
        ob.sortInWave(arr, n);
        for (int i : arr)
            System.out.print(i+" ");
    }
}
/*This code is contributed by Rajat Mishra*/

```

Output:

```
90 10 49 1 5 2 23
```

This article is contributed by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Sorting
array-rearrange

Why is Binary Search preferred over Ternary Search?

The following is a simple recursive **Binary Search** function in C++ taken from [here](#).

```

// A recursive binary search function. It returns location of x in
// given array arr[l..r] is present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l)
    {
        int mid = l + (r - l)/2;

```

```

// If the element is present at the middle itself
if (arr[mid] == x) return mid;

// If element is smaller than mid, then it can only be present
// in left subarray
if (arr[mid] > x) return binarySearch(arr, l, mid-1, x);

// Else the element can only be present in right subarray
return binarySearch(arr, mid+1, r, x);
}

// We reach here when element is not present in array
return -1;
}

```

The following is a simple recursive **Ternary Search** function in C++.

```

// A recursive ternary search function. It returns location of x in
// given array arr[l..r] is present, otherwise -1
int ternarySearch(int arr[], int l, int r, int x)
{
    if (r >= l)
    {
        int mid1 = l + (r - l)/3;
        int mid2 = mid1 + (r - l)/3;

        // If x is present at the mid1
        if (arr[mid1] == x) return mid1;

        // If x is present at the mid2
        if (arr[mid2] == x) return mid2;

        // If x is present in left one-third
        if (arr[mid1] > x) return ternarySearch(arr, l, mid1-1, x);

        // If x is present in right one-third
        if (arr[mid2] < x) return ternarySearch(arr, mid2+1, r, x);

        // If x is present in middle one-third
        return ternarySearch(arr, mid1+1, mid2-1, x);
    }
    // We reach here when element is not present in array
    return -1;
}

```

Which of the above two does less comparisons in worst case?

From the first look, it seems the ternary search does less number of comparisons as it makes $\log_3 n$ recursive calls, but binary search makes $\log_2 n$ recursive calls. Let us take a closer look. The following is recursive formula for counting comparisons in worst case of Binary Search.

$$T(n) = T(n/2) + 2, \quad T(1) = 1$$

The following is recursive formula for counting comparisons in worst case of Ternary Search.

$$T(n) = T(n/3) + 4, \quad T(1) = 1$$

In binary search, there are $2\log_2 n + 1$ comparisons in worst case. In ternary search, there are $4\log_3 n + 1$ comparisons in worst case.

Time Complexity for Binary search = $2c\log_2 n + O(1)$
 Time Complexity for Ternary search = $4c\log_3 n + O(1)$

Therefore, the comparison of Ternary and Binary Searches boils down the comparison of expressions $2\log_3 n$ and $\log_2 n$. The value of $2\log_3 n$ can be written as $(2 / \log_2 3) * \log_2 n$. Since the value of $(2 / \log_2 3)$ is more than one, Ternary Search does more comparisons than Binary Search in worst case.

Exercise:

Why Merge Sort divides input array in two halves, why not in three or more parts?

This article is contributed by **Anmol**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Searching

K'th Smallest/Largest Element in Unsorted Array | Set 2 (Expected Linear Time)

We recommend to read following post as a prerequisite of this post.

K'th Smallest/Largest Element in Unsorted Array | Set 1

Given an array and a number k where k is smaller than size of array, we need to find the k'th smallest element in the given array. It is given that all array elements are distinct.

Examples:

```

Input: arr[] = {7, 10, 4, 3, 20, 15}
k = 3
Output: 7

Input: arr[] = {7, 10, 4, 3, 20, 15}
k = 4
Output: 10

```

We have discussed three different solutions [here](#).

In this post method 4 is discussed which is mainly an extension of method 3 (QuickSelect) discussed in the [previous](#) post. The idea is to randomly pick a pivot element. To implement randomized partition, we use a random function, [rand\(\)](#) to generate index between l and r, swap the element at randomly generated index with the last element, and finally call the standard partition process which uses last element as pivot.

Following is implementation of above Randomized QuickSelect.

C/C++

```

// C++ implementation of randomized quickSelect
#include<iostream>
#include<climits>
#include<cstdlib>
using namespace std;

int randomPartition(int arr[], int l, int r);

// This function returns k'th smallest element in arr[l..r] using
// QuickSort based method. ASSUMPTION: ELEMENTS IN ARR[] ARE DISTINCT
int kthSmallest(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k > 0 && k <= r - l + 1)
    {
        // Partition the array around a random element and
        // get position of pivot element in sorted array
        int pos = randomPartition(arr, l, r);

        // If position is same as k
        if (pos - l == k - 1)
            return arr[pos];
        if (pos - l > k - 1) // If position is more, recur for left subarray
            return kthSmallest(arr, l, pos - 1, k);

        // Else recur for right subarray
        return kthSmallest(arr, pos + 1, r, k - pos + l - 1);
    }

    // If k is more than number of elements in array
    return INT_MAX;
}

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Standard partition process of QuickSort(). It considers the last
// element as pivot and moves all smaller element to left of it and
// greater elements to right. This function is used by randomPartition()
int partition(int arr[], int l, int r)
{
    int x = arr[r], i = l;
    for (int j = l; j <= r - 1; j++)
    {
        if (arr[j] <= x)
        {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }
    swap(&arr[i], &arr[r]);
    return i;
}

// Picks a random pivot element between l and r and partitions
// arr[l..r] around the randomly picked element using partition()
int randomPartition(int arr[], int l, int r)
{
    int n = r - l + 1;
    int pivot = rand() % n;
    swap(&arr[l + pivot], &arr[r]);
    return partition(arr, l, r);
}

// Driver program to test above methods
int main()
{
    int arr[] = {12, 3, 5, 7, 4, 19, 26};
    int n = sizeof(arr)/sizeof(arr[0]), k = 3;
    cout << "K'th smallest element is " << kthSmallest(arr, 0, n-1, k);
    return 0;
}

```

Java

```

// Java program to find k'th smallest element in expected
// linear time
class KthSmallest
{
    // This function returns k'th smallest element in arr[l..r]
    // using QuickSort based method. ASSUMPTION: ALL ELEMENTS
    // IN ARR[] ARE DISTINCT
    int kthSmallest(int arr[], int l, int r, int k)
    {
        // If k is smaller than number of elements in array
        if (k > 0 && k <= r - l + 1)
        {
            // Partition the array around a random element and
            // get position of pivot element in sorted array
            int pos = randomPartition(arr, l, r);

            // If position is same as k
            if (pos - l == k - 1)
                return arr[pos];

            // If position is more, recur for left subarray
            if (pos - l > k - 1)
                return kthSmallest(arr, l, pos - 1, k);

            // Else recur for right subarray
        }
    }
}

```

```

        return kthSmallest(arr, pos+1, r, k-pos+1);
    }

    // If k is more than number of elements in array
    return Integer.MAX_VALUE;
}

// Utility method to swap arr[i] and arr[j]
void swap(int arr[], int i, int j)
{
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

// Standard partition process of QuickSort(). It considers
// the last element as pivot and moves all smaller element
// to left of it and greater elements to right. This function
// is used by randomPartition()
int partition(int arr[], int l, int r)
{
    int x = arr[r], i = l;
    for (int j = l; j <= r - 1; j++)
    {
        if (arr[j] <= x)
        {
            swap(arr, i, j);
            i++;
        }
    }
    swap(arr, i, r);
    return i;
}

// Picks a random pivot element between l and r and
// partitions arr[l..r] around the randomly picked
// element using partition()
int randomPartition(int arr[], int l, int r)
{
    int n = r-l+1;
    int pivot = (int)(Math.random()) % n;
    swap(arr, l + pivot, r);
    return partition(arr, l, r);
}

// Driver method to test above
public static void main(String args[])
{
    KthSmallest ob = new KthSmallest();
    int arr[] = {12, 3, 5, 7, 4, 19, 26};
    int n = arr.length,k = 3;
    System.out.println("K'th smallest element is "+
        ob.kthSmallest(arr, 0, n-1, k));
}
}
/*This code is contributed by Rajat Mishra*/

```

Output:

K'th smallest element is 5

Time Complexity:

The worst case time complexity of the above solution is still $O(n^2)$. In worst case, the randomized function may always pick a corner element. The expected time complexity of above randomized QuickSelect is $\Theta(n)$, see [CLRS book](#) or [MIT video lecture](#) for proof. The assumption in the analysis is, random number generator is equally likely to generate any number in the input range.

Sources:

[MIT Video Lecture on Order Statistics, Median](#)

[Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.](#)

This article is contributed by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Randomized
Searching
Order-Statistics
Quick Sort

K'th Smallest/Largest Element in Unsorted Array | Set 2 (Expected Linear Time)

We recommend to read following post as a prerequisite of this post.

K'th Smallest/Largest Element in Unsorted Array | Set 1

Given an array and a number k where k is smaller than size of array, we need to find the k 'th smallest element in the given array. It is given that all array elements are distinct.

Examples:

Input: arr[] = {7, 10, 4, 3, 20, 15}
 $k = 3$

Output: 7

Input: arr[] = {7, 10, 4, 3, 20, 15}
 $k = 4$

Output: 10

We have discussed three different solutions [here](#).

In this post method 4 is discussed which is mainly an extension of method 3 (QuickSelect) discussed in the [previous](#) post. The idea is to randomly pick a pivot element. To implement randomized partition, we use a random function, [rand\(\)](#) to generate index between l and r , swap the element at randomly generated index with the last element, and finally call the standard partition process which uses last element as pivot.

Following is implementation of above Randomized QuickSelect.

C/C++

```
// C++ implementation of randomized quickSelect
#include<iostream>
#include<climits>
#include<cstdlib>
using namespace std;

int randomPartition(int arr[], int l, int r);

// This function returns k'th smallest element in arr[l..r] using
// QuickSort based method. ASSUMPTION: ELEMENTS IN ARR[] ARE DISTINCT
int kthSmallest(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k > 0 && k <= r - l + 1)
    {
        // Partition the array around a random element and
        // get position of pivot element in sorted array
        int pos = randomPartition(arr, l, r);

        // If position is same as k
        if (pos - l == k - 1)
            return arr[pos];
        if (pos - l > k - 1) // If position is more, recur for left subarray
            return kthSmallest(arr, l, pos-1, k);

        // Else recur for right subarray
        return kthSmallest(arr, pos+1, r, k - pos + l - 1);
    }

    // If k is more than number of elements in array
    return INT_MAX;
}

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Standard partition process of QuickSort(). It considers the last
// element as pivot and moves all smaller element to left of it and
// greater elements to right. This function is used by randomPartition()
int partition(int arr[], int l, int r)
{
    int x = arr[r], i = l;
    for (int j = l; j <= r - 1; j++)
    {
        if (arr[j] <= x)
        {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }
    swap(&arr[i], &arr[r]);
    return i;
}

// Picks a random pivot element between l and r and partitions
// arr[l..r] around the randomly picked element using partition()
int randomPartition(int arr[], int l, int r)
{
    int n = r - l + 1;
    int pivot = rand() % n;
    swap(&arr[l + pivot], &arr[r]);
    return partition(arr, l, r);
}

// Driver program to test above methods
int main()
{
    int arr[] = {12, 3, 5, 7, 4, 19, 26};
    int n = sizeof(arr)/sizeof(arr[0]), k = 3;
    cout << "K'th smallest element is " << kthSmallest(arr, 0, n-1, k);
    return 0;
}
```

Java

```
// Java program to find k'th smallest element in expected
// linear time
class KthSmallest
{
    // This function returns k'th smallest element in arr[l..r]
    // using QuickSort based method. ASSUMPTION: ALL ELEMENTS
    // IN ARR[] ARE DISTINCT
    int kthSmallest(int arr[], int l, int r, int k)
    {
        // If k is smaller than number of elements in array
        if (k > 0 && k <= r - l + 1)
        {
            // Partition the array around a random element and
            // get position of pivot element in sorted array
            int pos = randomPartition(arr, l, r);

            // If position is same as k
            if (pos - l == k - 1)
                return arr[pos];

            // If position is more, recur for left subarray
            if (pos - l > k - 1)
```

```

        return kthSmallest(arr, l, pos-1, k);

    // Else recur for right subarray
    return kthSmallest(arr, pos+1, r, k-pos+l-1);
}

// If k is more than number of elements in array
return Integer.MAX_VALUE;
}

// Utility method to swap arr[i] and arr[j]
void swap(int arr[], int i, int j)
{
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

// Standard partition process of QuickSort(). It considers
// the last element as pivot and moves all smaller element
// to left of it and greater elements to right. This function
// is used by randomPartition()
int partition(int arr[], int l, int r)
{
    int x = arr[r], i = l;
    for (int j = l; j <= r - 1; j++)
    {
        if (arr[j] <= x)
        {
            swap(arr, i, j);
            i++;
        }
    }
    swap(arr, i, r);
    return i;
}

// Picks a random pivot element between l and r and
// partitions arr[l..r] around the randomly picked
// element using partition()
int randomPartition(int arr[], int l, int r)
{
    int n = r-l+1;
    int pivot = (int)(Math.random()) % n;
    swap(arr, l + pivot, r);
    return partition(arr, l, r);
}

// Driver method to test above
public static void main(String args[])
{
    KthSmallest ob = new KthSmallest();
    int arr[] = {12, 3, 5, 7, 4, 19, 26};
    int n = arr.length, k = 3;
    System.out.println("K'th smallest element is "+ob.kthSmallest(arr, 0, n-1, k));
}
}

/*This code is contributed by Rajat Mishra*/

```

Output:

K'th smallest element is 5

Time Complexity:

The worst case time complexity of the above solution is still $O(n^2)$. In worst case, the randomized function may always pick a corner element. The expected time complexity of above randomized QuickSelect is $\Theta(n)$, see [CLRS book](#) or [MIT video lecture](#) for proof. The assumption in the analysis is, random number generator is equally likely to generate any number in the input range.

Sources:

[MIT Video Lecture on Order Statistics, Median](#)

[Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.](#)

This article is contributed by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Randomized
Searching
Order-Statistics
Quick Sort

K'th Smallest/Largest Element in Unsorted Array | Set 3 (Worst Case Linear Time)

We recommend to read following posts as a prerequisite of this post.

[K'th Smallest/Largest Element in Unsorted Array | Set 1](#)

[K'th Smallest/Largest Element in Unsorted Array | Set 2 \(Expected Linear Time\)](#)

Given an array and a number k where k is smaller than size of array, we need to find the k 'th smallest element in the given array. It is given that all array elements are distinct.

Examples:

Input: arr[] = {7, 10, 4, 3, 20, 15}
 $k = 3$
Output: 7

Input: arr[] = {7, 10, 4, 3, 20, 15}
 $k = 4$
Output: 10

In previous post, we discussed an expected linear time algorithm. In this post, a worst case linear time method is discussed. The idea in this new method is similar to quickSelect(), we get worst case linear time by selecting a pivot that divides array in a balanced way (there are not very few elements on one side and many on other side). After the array is divided in a balanced way, we apply the same steps as used in quickSelect() to decide whether to go left or right of pivot.

Following is complete algorithm.

```
kthSmallest(int arr[0..n-1], k)
1) Divide arr[] into [n/5ceil] groups where size of each group is 5
   except possibly the last group which may have less than 5 elements.

2) Sort the above created [n/5] groups and find median
   of all groups. Create an auxiliary array 'median[]' and store medians
   of all [n/5] groups in this median array.

// Recursively call this method to find median of median[0..[n/5]-1]
3) medOfMed = kthSmallest(median[0..[n/5]-1], [n/10])

4) Partition arr[] around medOfMed and obtain its position.
   pos = partition(arr, n, medOfMed)

5) If pos == k return medOfMed
6) If pos < k return kthSmallest(arr[..pos-1], k)
7) If pos > k return kthSmallest(arr[pos+1..r], k-pos+l-1)
```

In above algorithm, last 3 steps are same as algorithm in [previous post](#). The first four steps are used to obtain a good point for partitioning the array (to make sure that there are not too many elements either side of pivot).

Following is C++ implementation of above algorithm.

```
// C++ implementation of worst case linear time algorithm
// to find k'th smallest element
#include<iostream>
#include<algorithm>
#include<climits>
using namespace std;

int partition(int arr[], int l, int r, int k);

// A simple function to find median of arr[]. This is called
// only for an array of size 5 in this program.
int findMedian(int arr[], int n)
{
    sort(arr, arr+n); // Sort the array
    return arr[n/2]; // Return middle element
}

// Returns k'th smallest element in arr[l..r] in worst case
// linear time. ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
int kthSmallest(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k > 0 && k <= r - l + 1)
    {
        int n = r - l + 1; // Number of elements in arr[l..r]

        // Divide arr[] in groups of size 5, calculate median
        // of every group and store it in median[] array.
        int i, median[(n+4)/5]; // There will be floor((n+4)/5) groups;
        for (i=0; i<n/5; i++)
            median[i] = findMedian(arr+l+i*5, 5);
        if (i*5 < n) //For last group with less than 5 elements
        {
            median[i] = findMedian(arr+l+i*5, n%5);
            i++;
        }

        // Find median of all medians using recursive call.
        // If median[] has only one element, then no need
        // of recursive call
        int medOfMed = (i == 1)? median[i-1]:
            kthSmallest(median, 0, i-1, i/2);

        // Partition the array around a random element and
        // get position of pivot element in sorted array
        int pos = partition(arr, l, r, medOfMed);

        // If position is same as k
        if (pos-l == k-1)
            return arr[pos];
        if (pos-l > k-1) // If position is more, recur for left
            return kthSmallest(arr, l, pos-1, k);

        // Else recur for right subarray
        return kthSmallest(arr, pos+1, r, k-pos+l-1);
    }

    // If k is more than number of elements in array
    return INT_MAX;
}

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// It searches for x in arr[l..r], and partitions the array
// around x.
int partition(int arr[], int l, int r, int x)
{
    // Search for x in arr[l..r] and move it to end
    int i;
    for (i=l; i<r; i++)
        if (arr[i] == x)
            break;
    swap(&arr[i], &arr[r]);

    // Standard partition algorithm
    i = l;
```

```

for (int j = l; j <= r - 1; j++)
{
    if (arr[j] <= x)
    {
        swap(&arr[i], &arr[j]);
        i++;
    }
}
swap(&arr[i], &arr[r]);
return i;
}

// Driver program to test above methods
int main()
{
    int arr[] = {12, 3, 5, 7, 4, 19, 26};
    int n = sizeof(arr)/sizeof(arr[0]), k = 3;
    cout << "K'th smallest element is "
         << kthSmallest(arr, 0, n-1, k);
    return 0;
}

```

Output:

K'th smallest element is 5

Time Complexity:

The worst case time complexity of the above algorithm is O(n). Let us analyze all steps.

The steps 1) and 2) take O(n) time as finding median of an array of size 5 takes O(1) time and there are n/5 arrays of size 5.

The step 3) takes T(n/5) time. The step 4 is standard partition and takes O(n) time.

The interesting steps are 6) and 7). At most, one of them is executed. These are recursive steps. What is the worst case size of these recursive calls. The answer is maximum number of elements greater than medOfMed (obtained in step 3) or maximum number of elements smaller than medOfMed.

How many elements are greater than medOfMed and how many are smaller?

At least half of the medians found in step 2 are greater than or equal to medOfMed. Thus, at least half of the n/5 groups contribute 3 elements that are greater than medOfMed, except for the one group that has fewer than 5 elements. Therefore, the number of elements greater than medOfMed is at least.



Similarly, the number of elements that are less than medOfMed is at least 3n/10 - 6. In the worst case, the function recurs for at most n - (3n/10 - 6) which is 7n/10 + 6 elements.

Note that 7n/10 + 6 > 20 and that any input of 80 or fewer elements requires O(1) time. We can therefore obtain the recurrence



We show that the running time is linear by substitution. Assume that T(n) $\leq cn$ for some constant c and all $n > 80$. Substituting this inductive hypothesis into the right-hand side of the recurrence yields

T(n)

since we can pick c large enough so that $c(n/10 - 7)$ is larger than the function described by the O(n) term for all $n > 80$. The worst-case running time of is therefore linear (Source: <http://staff.ustc.edu.cn/~csli/graduate/algo.html>)

Note that the above algorithm is linear in worst case, but the constants are very high for this algorithm. Therefore, this algorithm doesn't work well in practical situations, **randomized quickSelect** works much better and pre

Sources:

[MIT Video Lecture on Order Statistics, Median](#)

[Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.](#)

<http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap10.htm>

This article is contributed by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner

Company Wise Coding Practice

Searching
Order-Statistics

Find the closest pair from two sorted arrays

Given two sorted arrays and a number x, find the pair whose sum is closest to x and the pair has an element from each array.

We are given two arrays $ar1[0..m-1]$ and $ar2[0..n-1]$ and a number x, we need to find the pair $ar1[i] + ar2[j]$ such that absolute value of $(ar1[i] + ar2[j] - x)$ is minimum.

Example:

```
Input: ar1[] = {1, 4, 5, 7};  
       ar2[] = {10, 20, 30, 40};  
       x = 32  
Output: 1 and 30  
  
Input: ar1[] = {1, 4, 5, 7};  
       ar2[] = {10, 20, 30, 40};  
       x = 50  
Output: 7 and 40
```

We strongly recommend to minimize your browser and try this yourself first.

A **Simple Solution** is to run two loops. The outer loop considers every element of first array and inner loop checks for the pair in second array. We keep track of minimum difference between $ar1[i] + ar2[j]$ and x .

We can do it in $O(n)$ time using following steps.

1) Merge given two arrays into an auxiliary array of size $m+n$ using merge process of merge sort. While merging keep another boolean array of size $m+n$ to indicate whether the current element in merged array is from $ar1[]$ or $ar2[]$.

2) Consider the merged array and use the linear time algorithm to find the pair with sum closest to x . One extra thing we need to consider only those pairs which have one element from $ar1[]$ and other from $ar2[]$, we use the boolean array for this purpose.

Can we do it in a single pass and $O(1)$ extra space?

The idea is to start from left side of one array and right side of another array, and use the algorithm same as step 2 of above approach. Following is detailed algorithm.

- 1) Initialize a variable diff as infinite (Diff is used to store the difference between pair and x). We need to find the minimum diff.
- 2) Initialize two index variables l and r in the given sorted array.
 - (a) Initialize first as the leftmost index in $ar1$: $l = 0$
 - (b) Initialize second as the rightmost index in $ar2$: $r = n-1$
- 3) Loop while $l < r$
 - (a) If $|ar1[l] + ar2[r] - x| < diff$

Following is C++ implementation of this approach.

C++

```
// C++ program to find the pair from two sorted arrays such  
// that the sum of pair is closest to a given number x  
#include <iostream>  
#include <climits>  
#include <cstdlib>  
using namespace std;  
  
// ar1[0..m-1] and ar2[0..n-1] are two given sorted arrays  
// and x is given number. This function prints the pair from  
// both arrays such that the sum of the pair is closest to x.  
void printClosest(int ar1[], int ar2[], int m, int n, int x)  
{  
    // Initialize the diff between pair sum and x.  
    int diff = INT_MAX;  
  
    // res_l and res_r are result indexes from ar1[] and ar2[]  
    // respectively  
    int res_l, res_r;  
  
    // Start from left side of ar1[] and right side of ar2[]  
    int l = 0, r = n-1;  
    while (l < m && r >= 0)  
    {  
        // If this pair is closer to x than the previously  
        // found closest, then update res_l, res_r and diff  
        if (abs(ar1[l] + ar2[r] - x) < diff)  
        {  
            res_l = l;  
            res_r = r;  
            diff = abs(ar1[l] + ar2[r] - x);  
        }  
  
        // If sum of this pair is more than x, move to smaller  
        // side  
        if (ar1[l] + ar2[r] > x)  
            r--;  
        else // move to the greater side  
            l++;  
    }  
  
    // Print the result  
    cout << "The closest pair is [" << ar1[res_l] << ", "  
        << ar2[res_r] << "] \n";  
}  
  
// Driver program to test above functions  
int main()  
{  
    int ar1[] = {1, 4, 5, 7};  
    int ar2[] = {10, 20, 30, 40};  
    int m = sizeof(ar1)/sizeof(ar1[0]);  
    int n = sizeof(ar2)/sizeof(ar2[0]);  
    int x = 38;  
    printClosest(ar1, ar2, m, n, x);  
    return 0;  
}
```

Java

```
// Java program to find closest pair in an array
class ClosestPair
{
    // ar1[0..m-1] and ar2[0..n-1] are two given sorted
    // arrays/ and x is given number. This function prints
    // the pair from both arrays such that the sum of the
    // pair is closest to x.
    void printClosest(int ar1[], int ar2[], int m, int n, int x)
    {
        // Initialize the diff between pair sum and x.
        int diff = Integer.MAX_VALUE;

        // res_l and res_r are result indexes from ar1[] and ar2[]
        // respectively
        int res_l = 0, res_r = 0;

        // Start from left side of ar1[] and right side of ar2[]
        int l = 0, r = n-1;
        while (l < m && r >= 0)
        {
            // If this pair is closer to x than the previously
            // found closest, then update res_l, res_r and diff
            if (Math.abs(ar1[l] + ar2[r] - x) < diff)
            {
                res_l = l;
                res_r = r;
                diff = Math.abs(ar1[l] + ar2[r] - x);
            }

            // If sum of this pair is more than x, move to smaller
            // side
            if (ar1[l] + ar2[r] > x)
                r--;
            else // move to the greater side
                l++;
        }

        // Print the result
        System.out.print("The closest pair is [" + ar1[res_l] +
                        ", " + ar2[res_r] + "]");
    }

    // Driver program to test above functions
    public static void main(String args[])
    {
        ClosestPair ob = new ClosestPair();
        int ar1[] = {1, 4, 5, 7};
        int ar2[] = {10, 20, 30, 40};
        int m = ar1.length;
        int n = ar2.length;
        int x = 38;
        ob.printClosest(ar1, ar2, m, n, x);
    }
}
/*This code is contributed by Rajat Mishra */
```

Output:

```
The closest pair is [7, 30]
```

This article is contributed by Harsh. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner

Company Wise Coding Practice

Searching

Find common elements in three sorted arrays

Given three arrays sorted in non-decreasing order, print all common elements in these arrays.

Examples:

```
ar1[] = {1, 5, 10, 20, 40, 80}
ar2[] = {6, 7, 20, 80, 100}
ar3[] = {3, 4, 15, 20, 30, 70, 80, 120}
Output: 20, 80

ar1[] = {1, 5, 5}
ar2[] = {3, 4, 5, 5, 10}
ar3[] = {5, 5, 10, 20}
Output: 5, 5
```

A simple solution is to first find intersection of two arrays and store the intersection in a temporary array, then find the intersection of third array and temporary array. Time complexity of this solution is $O(n_1 + n_2 + n_3)$ where n_1 , n_2 and n_3 are sizes of $ar1[]$, $ar2[]$ and $ar3[]$ respectively.

The above solution requires extra space and two loops, we can find the common elements using a single loop and without extra space. The idea is similar to [intersection of two arrays](#). Like two arrays loop, we run a loop and traverse three arrays.

Let the current element traversed in $ar1[]$ be x , in $ar2[]$ be y and in $ar3[]$ be z . We can have following cases inside the loop.

- 1) If x , y and z are same, we can simply print any of them as common element and move ahead in all three arrays.
- 2) Else If x and $y > z$, we can simply move ahead in $ar3[]$ as z cannot be a common element.

Following are implementations of the above idea.

C++

```
// C++ program to print common elements in three arrays
#include <iostream>
using namespace std;

// This function prints common elements in ar1
void findCommon(int ar1[], int ar2[], int ar3[], int n1, int n2, int n3)
{
    // Initialize starting indexes for ar1[], ar2[] and ar3[]
    int i = 0, j = 0, k = 0;

    // Iterate through three arrays while all arrays have elements
    while (i < n1 && j < n2 && k < n3)
    {
        // If x = y and y = z, print any of them and move ahead
        // in all arrays
        if (ar1[i] == ar2[j] && ar2[j] == ar3[k])
        { cout << ar1[i] << " "; i++; j++; k++; }

        // x < y
        else if (ar1[i] < ar2[j])
            i++;

        // y < z
        else if (ar2[j] < ar3[k])
            j++;

        // We reach here when x > y and z < y, i.e., z is smallest
        else
            k++;
    }

    // Driver program to test above function
    int main()
    {
        int ar1[] = {1, 5, 10, 20, 40, 80};
        int ar2[] = {6, 7, 20, 80, 100};
        int ar3[] = {3, 4, 15, 20, 30, 70, 80, 120};
        int n1 = sizeof(ar1)/sizeof(ar1[0]);
        int n2 = sizeof(ar2)/sizeof(ar2[0]);
        int n3 = sizeof(ar3)/sizeof(ar3[0]);

        cout << "Common Elements are ";
        findCommon(ar1, ar2, ar3, n1, n2, n3);
        return 0;
    }
}
```

Python

```
# Python function to print common elements in three sorted arrays
def findCommon(ar1, ar2, ar3, n1, n2, n3):

    # Initialize starting indexes for ar1[], ar2[] and ar3[]
    i, j, k = 0, 0, 0

    # Iterate through three arrays while all arrays have elements
    while (i < n1 and j < n2 and k < n3):

        # If x = y and y = z, print any of them and move ahead
        # in all arrays
        if (ar1[i] == ar2[j] and ar2[j] == ar3[k]):
            print(ar1[i]),
            i += 1
            j += 1
            k += 1

        # x < y
        elif ar1[i] < ar2[j]:
            i += 1

        # y < z
        elif ar2[j] < ar3[k]:
            j += 1

        # We reach here when x > y and z < y, i.e., z is smallest
        else:
            k += 1
```

```

k += 1

# Driver program to check above function
ar1 = [1, 5, 10, 20, 40, 80]
ar2 = [6, 7, 20, 80, 100]
ar3 = [3, 4, 15, 20, 30, 70, 80, 120]
n1 = len(ar1)
n2 = len(ar2)
n3 = len(ar3)
print "Common elements are",
findCommon(ar1, ar2, ar3, n1, n2, n3)

# This code is contributed by __Devesh Agrawal__

```

Java

```

// Java program to find common elements in three arrays
class FindCommon
{
    // This function prints common elements in ar1
    void findCommon(int ar1[], int ar2[], int ar3[])
    {
        // Initialize starting indexes for ar1[], ar2[] and ar3[]
        int i = 0, j = 0, k = 0;

        // Iterate through three arrays while all arrays have elements
        while (i < ar1.length && j < ar2.length && k < ar3.length)
        {
            // If x = y and y = z, print any of them and move ahead
            // in all arrays
            if (ar1[i] == ar2[j] && ar2[j] == ar3[k])
            { System.out.print(ar1[i] + " "); i++; j++; k++; }

            // x < y
            else if (ar1[i] < ar2[j])
                i++;

            // y < z
            else if (ar2[j] < ar3[k])
                j++;

            // We reach here when x > y and z < y, i.e., z is smallest
            else
                k++;
        }
    }

    // Driver code to test above
    public static void main(String args[])
    {
        FindCommon ob = new FindCommon();

        int ar1[] = {1, 5, 10, 20, 40, 80};
        int ar2[] = {6, 7, 20, 80, 100};
        int ar3[] = {3, 4, 15, 20, 30, 70, 80, 120};

        System.out.print("Common elements are ");
        ob.findCommon(ar1, ar2, ar3);
    }
}
/*This code is contributed by Rajat Mishra */

```

Output:

```
Common Elements are 20 80
```

Time complexity of the above solution is $O(n_1 + n_2 + n_3)$. In worst case, the largest sized array may have all small elements and middle sized array has all middle elements.

This article is compiled by **Rahul Gupta** Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Arrays

Given a sorted array and a number x, find the pair in array whose sum is closest to x

Given a sorted array and a number x, find a pair in array whose sum is closest to x.

Examples:

```
Input: arr[] = {10, 22, 28, 29, 30, 40}, x = 54
Output: 22 and 30
```

```
Input: arr[] = {1, 3, 4, 7, 10}, x = 15
Output: 4 and 10
```

A simple solution is to consider every pair and keep track of closest pair (absolute difference between pair sum and x is minimum). Finally print the closest pair. Time complexity of this solution is $O(n^2)$

An efficient solution can find the pair in $O(n)$ time. The idea is similar to method 2 of [this post](#). Following is detailed algorithm.

- 1) Initialize a variable diff as infinite (Diff is used to store the difference between pair and x). We need to find the minimum diff.
- 2) Initialize two index variables l and r in the given sorted array.
 - (a) Initialize first to the leftmost index: l = 0
 - (b) Initialize second the rightmost index: r = n-1
- 3) Loop while l < r

Following is C++ implementation of above algorithm.

C++

```
// Simple C++ program to find the pair with sum closest to a given no.
#include <iostream>
#include <climits>
#include <cstdlib>
using namespace std;

// Prints the pair with sum closest to x
void printClosest(int arr[], int n, int x)
{
    int res_l, res_r; // To store indexes of result pair

    // Initialize left and right indexes and difference between
    // pair sum and x
    int l = 0, r = n-1, diff = INT_MAX;

    // While there are elements between l and r
    while (r > l)
    {
        // Check if this pair is closer than the closest pair so far
        if (abs(arr[l] + arr[r] - x) < diff)
        {
            res_l = l;
            res_r = r;
            diff = abs(arr[l] + arr[r] - x);
        }

        // If this pair has more sum, move to smaller values.
        if (arr[l] + arr[r] > x)
            r--;
        else // Move to larger values
            l++;
    }

    cout << "The closest pair is " << arr[res_l] << " and " << arr[res_r];
}

// Driver program to test above functions
int main()
{
    int arr[] = {10, 22, 28, 29, 30, 40}, x = 54;
    int n = sizeof(arr)/sizeof(arr[0]);
    printClosest(arr, n, x);
    return 0;
}
```

Java

```
// Java program to find pair with sum closest to x
import java.io.*;
import java.util.*;
import java.lang.Math;

class CloseSum {

    // Prints the pair with sum closest to x
    static void printClosest(int arr[], int n, int x)
    {
        int res_l=0, res_r=0; // To store indexes of result pair

        // Initialize left and right indexes and difference between
        // pair sum and x
        int l = 0, r = n-1, diff = Integer.MAX_VALUE;

        // While there are elements between l and r
        while (r > l)
        {
            // Check if this pair is closer than the closest pair so far
            if (Math.abs(arr[l] + arr[r] - x) < diff)
            {
                res_l = l;
                res_r = r;
                diff = Math.abs(arr[l] + arr[r] - x);
            }

            // If this pair has more sum, move to smaller values.
            if (arr[l] + arr[r] > x)
                r--;
            else // Move to larger values
                l++;
        }

        System.out.println("The closest pair is "+arr[res_l]+" and "+arr[res_r]);
    }

    // Driver program to test above function
    public static void main(String[] args)
    {
        int arr[] = {10, 22, 28, 29, 30, 40}, x = 54;
        int n = arr.length;
        printClosest(arr, n, x);
    }
}
```

```
}
```

```
/*This code is contributed by Devesh Agrawal*/
```

Output:

```
The closest pair is 22 and 30
```

This article is contributed by **Harsh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: [Algorithms](#)

Count 1's in a sorted binary array

Given a binary array sorted in non-increasing order, count the number of 1's in it.

Examples:

```
Input: arr[] = {1, 1, 0, 0, 0, 0, 0}  
Output: 2
```

```
Input: arr[] = {1, 1, 1, 1, 1, 1, 1}  
Output: 7
```

```
Input: arr[] = {0, 0, 0, 0, 0, 0, 0}  
Output: 0
```

A simple solution is to linearly traverse the array. The time complexity of the simple solution is $O(n)$. We can use [Binary Search](#) to find count in $O(\log n)$ time. The idea is to look for last occurrence of 1 using Binary Search. Once we find the index last occurrence, we return index + 1 as count.

The following is C++ implementation of above idea.

C++

```
// C++ program to count one's in a boolean array  
#include <iostream>  
using namespace std;  
  
/* Returns counts of 1's in arr[low..high]. The array is  
assumed to be sorted in non-increasing order */  
int countOnes(bool arr[], int low, int high)  
{  
    if (high >= low)  
    {  
        // get the middle index  
        int mid = low + (high - low)/2;  
  
        // check if the element at middle index is last 1  
        if ((mid == high || arr[mid+1] == 0) && (arr[mid] == 1))  
            return mid+1;  
  
        // If element is not last 1, recur for right side  
        if (arr[mid] == 1)  
            return countOnes(arr, (mid + 1), high);  
  
        // else recur for left side  
        return countOnes(arr, low, (mid -1));  
    }  
    return 0;  
}
```

```

/* Driver program to test above functions */
int main()
{
    bool arr[] = {1, 1, 1, 1, 0, 0, 0};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Count of 1's in given array is " << countOnes(arr, 0, n-1);
    return 0;
}

```

Python

```

# Python program to count one's in a boolean array

# Returns counts of 1's in arr[low..high]. The array is
# assumed to be sorted in non-increasing order
def countOnes(arr,low,high):

    if high>=low:

        # get the middle index
        mid = low + (high-low)/2

        # check if the element at middle index is last 1
        if ((mid == high or arr[mid+1]==0) and (arr[mid]==1)):
            return mid+1

        # If element is not last 1, recur for right side
        if arr[mid]==1:
            return countOnes(arr, (mid+1), high)

        # else recur for left side
        return countOnes(arr, low, mid-1)

    return 0

# Driver function
arr=[1, 1, 1, 1, 0, 0, 0]
print("Count of 1's in given array is",countOnes(arr, 0 , len(arr)-1))

# This code is contributed by _Devesh Agrawal_

```

Java

```

// Java program to count 1's in a sorted array
class CountOnes
{
    /* Returns counts of 1's in arr[low..high]. The
    array is assumed to be sorted in non-increasing
    order */
    int countOnes(int arr[], int low, int high)
    {
        if (high >= low)
        {
            // get the middle index
            int mid = low + (high - low)/2;

            // check if the element at middle index is last 1
            if ((mid == high || arr[mid+1] == 0) &&
                (arr[mid] == 1))
                return mid+1;

            // If element is not last 1, recur for right side
            if (arr[mid] == 1)
                return countOnes(arr, (mid + 1), high);

            // else recur for left side
            return countOnes(arr, low, (mid - 1));
        }
        return 0;
    }

    /* Driver program to test above functions */
    public static void main(String args[])
    {
        CountOnes ob = new CountOnes();
        int arr[] = {1, 1, 1, 1, 0, 0, 0};
        int n = arr.length;
        System.out.println("Count of 1's in given array is " +
                           ob.countOnes(arr, 0, n-1));
    }
}

/* This code is contributed by Rajat Mishra */

```

Output:

```
Count of 1's in given array is 4
```

Time complexity of the above solution is O(Logn)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

Binary Insertion Sort

We can use binary search to reduce the number of comparisons in [normal insertion sort](#). Binary Insertion Sort find use binary search to find the proper location to insert the selected item at each iteration. In normal insertion, sort it takes $O(i)$ (at ith iteration) in worst case. we can reduce it to $O(\log i)$ by using [binary search](#).

```
// C program for implementation of binary insertion sort
#include <stdio.h>

// A binary search based function to find the position
// where item should be inserted in a[low..high]
int binarySearch(int a[], int item, int low, int high)
{
    if (high <= low)
        return (item > a[low]) ? (low + 1) : low;

    int mid = (low + high)/2;

    if(item == a[mid])
        return mid+1;

    if(item > a[mid])
        return binarySearch(a, item, mid+1, high);
    return binarySearch(a, item, low, mid-1);
}

// Function to sort an array a[] of size 'n'
void insertionSort(int a[], int n)
{
    int i, loc, j, k, selected;

    for (i = 1; i < n; ++i)
    {
        j = i - 1;
        selected = a[i];

        // find location where selected could be inserted
        loc = binarySearch(a, selected, 0, j);

        // Move all elements after location to create space
        while (j >= loc)
        {
            a[j+1] = a[j];
            j--;
        }
        a[j+1] = selected;
    }
}

// Driver program to test above function
int main()
{
    int a[] = {37, 23, 0, 17, 12, 72, 31,
              46, 100, 88, 54};
    int n = sizeof(a)/sizeof(a[0]), i;

    insertionSort(a, n);

    printf("Sorted array: \n");
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);

    return 0;
}
```

Output:

```
Sorted array:
0 12 17 23 31 37 46 54 72 88 100
```

Time Complexity: The algorithm as a whole still has a running worst case running time of $O(n^2)$ because of the series of swaps required for each insertion.

This article is contributed by **Amit Auddy**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Algorithms Searching and Sorting

Insertion Sort for Singly Linked List

We have discussed [Insertion Sort](#) for arrays. In this article same for linked list is discussed.

Below is simple insertion sort algorithm for linked list.

- 1) Create an empty sorted (or result) list
- 2) Traverse the given list, do following for every node.
 -a) Insert current node in sorted way in sorted or result list.
 - 3) Change head of given linked list to head of sorted (or result) list.

The main step is (2.a) which has been covered in below post.

Sorted Insert for Singly Linked List

Below is C implementation of above algorithm

```
/* C program for insertion sort on a linked list */
#include<stdio.h>
#include<stdlib.h>
```

```

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

// Function to insert a given node in a sorted linked list
void sortedInsert(struct node**, struct node*);

// function to sort a singly linked list using insertion sort
void insertionSort(struct node **head_ref)
{
    // Initialize sorted linked list
    struct node *sorted = NULL;

    // Traverse the given linked list and insert every
    // node to sorted
    struct node *current = *head_ref;
    while (current != NULL)
    {
        // Store next for next iteration
        struct node *next = current->next;

        // insert current in sorted linked list
        sortedInsert(&sorted, current);

        // Update current
        current = next;
    }

    // Update head_ref to point to sorted linked list
    *head_ref = sorted;
}

/* function to insert a new_node in a list. Note that this
function expects a pointer to head_ref as this can modify the
head of the input linked list (similar to push())*/
void sortedInsert(struct node** head_ref, struct node* new_node)
{
    struct node* current;
    /* Special case for the head end */
    if (*head_ref == NULL || (*head_ref)->data >= new_node->data)
    {
        new_node->next = *head_ref;
        *head_ref = new_node;
    }
    else
    {
        /* Locate the node before the point of insertion */
        current = *head_ref;
        while (current->next!=NULL &&
               current->next->data < new_node->data)
        {
            current = current->next;
        }
        new_node->next = current->next;
        current->next = new_node;
    }
}

/* BELOW FUNCTIONS ARE JUST UTILITY TO TEST sortedInsert */

/* Function to print linked list */
void printList(struct node *head)
{
    struct node *temp = head;
    while(temp != NULL)
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}

/* A utility function to insert a node at the beginning of linked list */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node = new node;

    /* put in the data */
    new_node->data = new_data;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

// Driver program to test above functions
int main()
{
    struct node *a = NULL;
    push(&a, 5);
    push(&a, 20);
    push(&a, 4);
    push(&a, 3);
    push(&a, 30);

    printf("Linked List before sorting \n");
    printList(a);

    insertionSort(&a);
}

```

```
printf("\nLinked List after sorting \n");
printList(a);

return 0;
}
```

```
Linked List before sorting
30 3 4 20 5
Linked List after sorting
3 4 5 20 30
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: [Linked List](#)

Why Quick Sort preferred for Arrays and Merge Sort for Linked Lists?

Why is Quick Sort preferred for arrays?

Below are recursive and iterative implementations of Quick Sort and Merge Sort for arrays.

Recursive Quick Sort for array.
Iterative Quick Sort for arrays.
Recursive Merge Sort for arrays
Iterative Merge Sort for arrays

Quick Sort in its general form is an in-place sort (i.e. it doesn't require any extra storage) whereas merge sort requires $O(N)$ extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. Comparing average complexity we find that both type of sorts have $O(N\log N)$ average complexity but the constants differ. For arrays, merge sort loses due to the use of extra $O(N)$ storage space.

Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of $O(n\log n)$. The worst case is possible in randomized version also, but worst case doesn't occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.

Quick Sort is also a cache friendly sorting algorithm as it has good [locality of reference](#) when used for arrays.

Quick Sort is also [tail recursive](#), therefore tail call optimizations is done.

Why is Merge Sort preferred for Linked Lists?

Below are implementations of Quicksort and Mergesort for singly and doubly linked lists.

Quick Sort for Doubly Linked List
Quick Sort for Singly Linked List
Merge Sort for Singly Linked List
Merge Sort for Doubly Linked List

In case of [linked lists](#) the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in $O(1)$ extra space and $O(1)$ time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.

In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of $A[0]$ be x then to access $A[i]$, we can directly access the memory at $(x + i * 4)$. Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i 'th index, we have to travel each and every node from the head to i 'th node as we don't have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

Related Articles:

[Know Your Sorting Algorithm | Set 1 \(Sorting Weapons used by Programming Languages\)](#)
[Iterative Merge Sort](#)
[Iterative Quick Sort](#)

Thanks to [Sayan Mukhopadhyay](#) for providing initial draft for above article. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

[Sorting](#)
[Merge Sort](#)
[Quick Sort](#)

Merge Sort for Doubly Linked List

Given a doubly linked list, write a function to sort the doubly linked list in increasing order using merge sort.

For example, the following doubly linked list should be changed to 24810



We strongly recommend to minimize your browser and try this yourself first.

Merge sort for singly linked list is already discussed. The important change here is to modify the previous pointers also when merging two lists.

Below is the implementation of merge sort for doubly linked list.

C

```
// C program for merge sort on doubly linked list
#include<stdio.h>
#include<stdlib.h>
struct node
{
```

```

int data;
struct node *next, *prev;
};

struct node *split(struct node *head);

// Function to merge two linked lists
struct node *merge(struct node *first, struct node *second)
{
    // If first linked list is empty
    if (!first)
        return second;

    // If second linked list is empty
    if (!second)
        return first;

    // Pick the smaller value
    if (first->data < second->data)
    {
        first->next = merge(first->next,second);
        first->next->prev = first;
        first->prev = NULL;
        return first;
    }
    else
    {
        second->next = merge(first,second->next);
        second->next->prev = second;
        second->prev = NULL;
        return second;
    }
}

// Function to do merge sort
struct node *mergeSort(struct node *head)
{
    if (!head || !head->next)
        return head;
    struct node *second = split(head);

    // Recur for left and right halves
    head = mergeSort(head);
    second = mergeSort(second);

    // Merge the two sorted halves
    return merge(head,second);
}

// A utility function to insert a new node at the
// beginning of doubly linked list
void insert(struct node **head, int data)
{
    struct node *temp =
        (struct node *)malloc(sizeof(struct node));
    temp->data = data;
    temp->next = temp->prev = NULL;
    if (!(*head))
        (*head) = temp;
    else
    {
        temp->next = *head;
        (*head)->prev = temp;
        (*head) = temp;
    }
}

// A utility function to print a doubly linked list in
// both forward and backward directions
void print(struct node *head)
{
    struct node *temp = head;
    printf("Forward Traversal using next pointer\n");
    while (head)
    {
        printf("%d ",head->data);
        temp = head;
        head = head->next;
    }
    printf("\nBackward Traversal using prev pointer\n");
    while (temp)
    {
        printf("%d ", temp->data);
        temp = temp->prev;
    }
}

// Utility function to swap two integers
void swap(int *A, int *B)
{
    int temp = *A;
    *A = *B;
    *B = temp;
}

// Split a doubly linked list (DLL) into 2 DLLs of
// half sizes
struct node *split(struct node *head)
{
    struct node *fast = head,*slow = head;
    while (fast->next && fast->next->next)
    {
        fast = fast->next->next;
        slow = slow->next;
    }
    struct node *temp = slow->next;
    slow->next = NULL;
}

```

```

    return temp;
}

// Driver program
int main(void)
{
    struct node *head = NULL;
    insert(&head,5);
    insert(&head,20);
    insert(&head,4);
    insert(&head,3);
    insert(&head,30);
    insert(&head,10);
    head = mergeSort(head);
    printf("\nUnLinked List after sorting\n");
    print(head);
    return 0;
}

```

Java

```

// Java program to implement merge sort in singly linked list

// Linked List Class
class LinkedList {

    static Node head; // head of list

    /* Node Class */
    static class Node {

        int data;
        Node next, prev;

        // Constructor to create a new node
        Node(int d) {
            data = d;
            next = prev = null;
        }
    }

    void print(Node node) {
        Node temp = node;
        System.out.println("Forward Traversal using next pointer");
        while (node != null) {
            System.out.print(node.data + " ");
            temp = node;
            node = node.next;
        }
        System.out.println("\nBackward Traversal using prev pointer");
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.prev;
        }
    }

    // Split a doubly linked list (DLL) into 2 DLLs of
    // half sizes
    Node split(Node head) {
        Node fast = head, slow = head;
        while (fast.next != null && fast.next.next != null) {
            fast = fast.next.next;
            slow = slow.next;
        }
        Node temp = slow.next;
        slow.next = null;
        return temp;
    }

    Node mergeSort(Node node) {
        if (node == null || node.next == null) {
            return node;
        }
        Node second = split(node);

        // Recur for left and right halves
        node = mergeSort(node);
        second = mergeSort(second);

        // Merge the two sorted halves
        return merge(node, second);
    }

    // Function to merge two linked lists
    Node merge(Node first, Node second) {
        // If first linked list is empty
        if (first == null) {
            return second;
        }

        // If second linked list is empty
        if (second == null) {
            return first;
        }

        // Pick the smaller value
        if (first.data < second.data) {
            first.next = merge(first.next, second);
            first.next.prev = first;
            first.prev = null;
            return first;
        } else {
            second.next = merge(first, second.next);
            second.next.prev = second;
        }
    }
}

```

```

        second.prev = null;
        return second;
    }

    // Driver program to test above functions
    public static void main(String[] args) {

        LinkedList list = new LinkedList();
        list.head = new Node(10);
        list.head.next = new Node(30);
        list.head.next.next = new Node(3);
        list.head.next.next.next = new Node(4);
        list.head.next.next.next.next = new Node(20);
        list.head.next.next.next.next.next = new Node(5);

        Node node = null;
        node = list.mergeSort(head);
        System.out.println("Linked list after sorting :");
        list.print(node);

    }
}

// This code has been contributed by Mayank Jaiswal

```

Python

```

# Program for merge sort on doubly linked list

# A node of the doubly linked list
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:

    # Constructor for empty Doubly Linked List
    def __init__(self):
        self.head = None

    # Function to merge two linked list
    def merge(self, first, second):

        # If first linked list is empty
        if first is None:
            return second

        # If secon linked list is empty
        if second is None:
            return first

        # Pick the smaller value
        if first.data < second.data:
            first.next = self.merge(first.next, second)
            first.next.prev = first
            first.prev = None
            return first
        else:
            second.next = self.merge(first, second.next)
            second.next.prev = second
            second.prev = None
            return second

    # Function to do merge sort
    def mergeSort(self, tempHead):
        if tempHead is None:
            return tempHead
        if tempHead.next is None:
            return tempHead

        second = self.split(tempHead)

        # Recur for left and right halves
        tempHead = self.mergeSort(tempHead)
        second = self.mergeSort(second)

        # Merge the two sorted halves
        return self.merge(tempHead, second)

    # Split the doubly linked list (DLL) into two DLLs
    # of half sizes
    def split(self, tempHead):
        fast = slow = tempHead
        while(True):
            if fast.next is None:
                break
            if fast.next.next is None:
                break
            fast = fast.next.next
            slow = slow.next

        temp = slow.next
        slow.next = None
        return temp

```

```

# Given a reference to the head of a list and an
# integer, inserts a new node on the front of list
def push(self, new_data):

    # 1. Allocates node
    # 2. Put the data in it
    new_node = Node(new_data)

    # 3. Make next of new node as head and
    # previous as None (already None)
    new_node.next = self.head

    # 4. change prev of head node to new_node
    if self.head is not None:
        self.head.prev = new_node

    # 5. move the head to point to the new node
    self.head = new_node

def printList(self, node):
    temp = node
    print "Forward Traversal using next pointer"
    while(node is not None):
        print node.data,
        temp = node
        node = node.next
    print "\nBackward Traversal using prev pointer"
    while(temp):
        print temp.data,
        temp = temp.prev

# Driver program to test the above functions
dll = DoublyLinkedList()
dll.push(5)
dll.push(20);
dll.push(4);
dll.push(3);
dll.push(30)
dll.push(10);
dll.head = dll.mergeSort(dll.head)
print "Linked List after sorting"
dll.printList(dll.head)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

Linked List after sorting
Forward Traversal using next pointer
3 4 5 10 20 30
Backward Traversal using prev pointer
30 20 10 5 4 3

```

Thanks to Goku for providing above implementation in a comment here.

Time Complexity: Time complexity of the above implementation is same as time complexity of MergeSort for arrays. It takes $\Theta(n \log n)$ time.

You may also like to see [QuickSort for doubly linked list](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

[Linked Lists](#)
[Sorting](#)
[Merge Sort](#)

Greedy Algorithms | Set 1 (Activity Selection Problem)

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Greedy algorithms are used for optimization problems. An optimization problem can be solved using Greedy if the problem has the following property: *At every step, we can make a choice that looks best at the moment, and we get the optimal solution of the complete problem.*

If a Greedy Algorithm can solve a problem, then it generally becomes the best method to solve that problem as the Greedy algorithms are in general more efficient than other techniques like Dynamic Programming. But Greedy algorithms cannot always be applied. For example, Fractional Knapsack problem (See [this](#)) can be solved using Greedy, but [0-1 Knapsack](#) cannot be solved using Greedy.

Following are some standard algorithms that are Greedy algorithms.

1) Kruskal's Minimum Spanning Tree (MST): In Kruskal's algorithm, we create a MST by picking edges one by one. The Greedy Choice is to pick the smallest weight edge that doesn't cause a cycle in the MST constructed so far.

2) Prim's Minimum Spanning Tree: In Prim's algorithm also, we create a MST by picking edges one by one. We maintain two sets: set of the vertices already included in MST and the set of the vertices not yet included. The Greedy Choice is to pick the smallest weight edge that connects the two sets.

3) Dijkstra's Shortest Path: The Dijkstra's algorithm is very similar to Prim's algorithm. The shortest path tree is built up, edge by edge. We maintain two sets: set of the vertices already included in the tree and the set of the vertices not yet included. The Greedy Choice is to pick the edge that connects the two sets and is on the smallest weight path from source to the set that contains not yet included vertices.

4) Huffman Coding: Huffman Coding is a loss-less compression technique. It assigns variable length bit codes to different characters. The Greedy Choice is to assign least bit length code to the most frequent character.

The greedy algorithms are sometimes also used to get an approximation for Hard optimization problems. For example, [Traveling Salesman Problem](#) is a NP Hard problem. A Greedy choice for this problem is to pick the nearest unvisited city from the current city at every step. This solutions doesn't always produce the best optimal solution, but can be used to get an approximate optimal solution.

Let us consider the [Activity Selection problem](#) as our first example of Greedy algorithms. Following is the problem statement.

You are given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

Example:

Example 1 : Consider the following 3 activities sorted by
by finish time.
start[] = {10, 12, 20};
finish[] = {20, 25, 30};
A person can perform at most **two** activities. The

```
maximum set of activities that can be executed  
is {0, 2} [ These are indexes in start[] and  
finish[] ]
```

Example 2 : Consider the following 6 activities

sorted by their finish time.

```
start[] = {1, 3, 0, 5, 8, 5};
```

```
finish[] = {2, 4, 6, 7, 9, 9};
```

A person can perform at most **four** activities. The maximum set of activities that can be executed is {0, 1, 3, 4} [These are indexes in start[] and finish[]]

The greedy choice is to always pick the next activity whose finish time is least among the remaining activities and the start time is more than or equal to the finish time of previously selected activity. We can sort the activities according to their finishing time so that we always consider the next activity as minimum finishing time activity.

- 1) Sort the activities according to their finishing time
- 2) Select the first activity from the sorted array and print it.
- 3) Do the same for remaining activities in the sorted array.

.....a) If the start time of this activity is greater than or equal to the finish time of previously selected activity then select this activity and print it.

In the following C implementation, it is assumed that the activities are already sorted according to their finish time.

C++

```
// C++ program for activity selection problem.  
// The following implementation assumes that the activities  
// are already sorted according to their finish time  
#include<stdio.h>  
  
// Prints a maximum set of activities that can be done by a single  
// person, one at a time.  
// n -> Total number of activities  
// s[] -> An array that contains start time of all activities  
// f[] -> An array that contains finish time of all activities  
void printMaxActivities(int s[], int f[], int n)  
{  
    int i, j;  
  
    printf("Following activities are selected \n");  
  
    // The first activity always gets selected  
    i = 0;  
    printf("%d ", i);  
  
    // Consider rest of the activities  
    for (j = 1; j < n; j++)  
    {  
        // If this activity has start time greater than or  
        // equal to the finish time of previously selected  
        // activity, then select it  
        if (s[j] >= f[i])  
        {  
            printf(" %d ", j);  
            i = j;  
        }  
    }  
  
    // driver program to test above function  
int main()  
{  
    int s[] = {1, 3, 0, 5, 8, 5};  
    int f[] = {2, 4, 6, 7, 9, 9};  
    int n = sizeof(s)/sizeof(s[0]);  
    printMaxActivities(s, f, n);  
    return 0;  
}
```

Java

```
// The following implementation assumes that the activities  
// are already sorted according to their finish time  
import java.util.*;  
import java.lang.*;  
import java.io.*;  
  
class ActivitySelection  
{  
    // Prints a maximum set of activities that can be done by a single  
    // person, one at a time.  
    // n -> Total number of activities  
    // s[] -> An array that contains start time of all activities  
    // f[] -> An array that contains finish time of all activities  
    public static void printMaxActivities(int s[], int f[], int n)  
    {  
        int i, j;  
  
        System.out.print("Following activities are selected : \n");  
  
        // The first activity always gets selected  
        i = 0;  
        System.out.print(i+" ");  
  
        // Consider rest of the activities  
        for (j = 1; j < n; j++)  
        {  
            // If this activity has start time greater than or  
            // equal to the finish time of previously selected  
            // activity, then select it  
            if (s[j] >= f[i])
```

```

    {
        System.out.print(j+" ");
        i = j;
    }
}

// driver program to test above function
public static void main(String[] args)
{
    int s[] = {1, 3, 0, 5, 8, 5};
    int f[] = {2, 4, 6, 7, 9, 9};
    int n = s.length;

    printMaxActivities(s, f, n);
}
}

```

Python

```

"""The following implementation assumes that the activities
are already sorted according to their finish time"""

"""Prints a maximum set of activities that can be done by a
single person, one at a time"""
# n -> Total number of activities
# s[] -> An array that contains start time of all activities
# f[] -> An array that contains finish time of all activities

def printMaxActivities(s, f):
    n = len(f)
    print "The following activities are selected"

    # The first activity is always selected
    i = 0
    print i,

    # Consider rest of the activities
    for j in xrange(n):

        # If this activity has start time greater than
        # or equal to the finish time of previously
        # selected activity, then select it
        if s[j] >= f[i]:
            print j,
            i = j

# Driver program to test above function
s = [1, 3, 0, 5, 8, 5]
f = [2, 4, 6, 7, 9, 9]
printMaxActivities(s, f)

# This code is contributed by Nikhil Kumar Singh

```

Output:

```

Following activities are selected
0 1 3 4

```

How does Greedy Choice work for Activities sorted according to finish time?

Let the give set of activities be $S = \{1, 2, 3, \dots, n\}$ and activities be sorted by finish time. The greedy choice is to always pick activity 1. How come the activity 1 always provides one of the optimal solutions. We can prove it by showing that if there is another solution B with first activity other than 1, then there is also a solution A of same size with activity 1 as first activity. Let the first activity selected by B be k , then there always exist $A = \{B - \{k\}\} \cup \{1\}$. (Note that the activities in B are independent and k has smallest finishing time among all. Since k is not 1, $\text{finish}(k) \geq \text{finish}(1)$).

How to implement when given activities are not sorted?

We create a structure/class for activities. We sort all activities by finish time (Refer sort in C++ STL). Once we have activities sorted, we apply same above algorithm.

```

// C++ program for activity selection problem
// when input activities may not be sorted.
#include <iostream>
using namespace std;

// A job has start time, finish time and profit.
struct Activity
{
    int start, finish;
};

// A utility function that is used for sorting
// activities according to finish time
bool activityCompare(Activity s1, Activity s2)
{
    return (s1.finish < s2.finish);
}

// Returns count of maximum set of activities that can
// be done by a single person, one at a time.
void printMaxActivities(Activity arr[], int n)
{
    // Sort jobs according to finish time
    sort(arr, arr+n, activityCompare);

    cout << "Following activities are selected \n";

    // The first activity always gets selected
    int i = 0;
    cout << "(" << arr[i].start << ", " << arr[i].finish << ")";

    // Consider rest of the activities
    for (int j = 1; j < n; j++)
    {

```

```

// If this activity has start time greater than or
// equal to the finish time of previously selected
// activity, then select it
if (arr[j].start >= arr[i].finish)
{
    cout << "(" << arr[j].start << ", "
        << arr[j].finish << ")";
    i = j;
}
}

// Driver program
int main()
{
    Activity arr[] = {{5, 9}, {1, 2}, {3, 4}, {0, 6},
                      {5, 7}, {8, 9}};
    int n = sizeof(arr)/sizeof(arr[0]);
    printMaxActivities(arr, n);
    return 0;
}

```

Output:

Following activities are selected
(1, 2), (3, 4), (5, 7), (8, 9).

Time Complexity : It takes O(n log n) time if input activities may not be sorted. It takes O(n) time when it is given that input activities are always sorted.

References:

Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
http://en.wikipedia.org/wiki/Greedy_algorithm

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Greedy
Activity Selection Problem
Greedy
Greedy Algorithm

Greedy Algorithms | Set 2 (Kruskal's Minimum Spanning Tree Algorithm)

What is Minimum Spanning Tree?

Given a connected and undirected graph, a *spanning tree* of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A *minimum spanning tree (MST)* or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

How many edges does a minimum spanning tree has?

A minimum spanning tree has $(V - 1)$ edges where V is the number of vertices in the given graph.

What are the applications of Minimum Spanning Tree?

See this for applications of MST.

Below are the steps for finding MST using Kruskal's algorithm

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree

formed so far. If cycle is not formed, include this edge. Else, discard it.

3. Repeat step#2 until there are $(V-1)$ edges in the spanning tree.

The step#2 uses [Union-Find algorithm](#) to detect cycle. So we recommend to read following post as a prerequisite.

[Union-Find Algorithm | Set 1 \(Detect Cycle in a Graph\)](#)

[Union-Find Algorithm | Set 2 \(Union By Rank and Path Compression\)](#)

The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example: Consider the below input graph.



The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having $(9 - 1) = 8$ edges.

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

Now pick all edges one by one from sorted list of edges

1. Pick edge 7-6: No cycle is formed, include it.



2. Pick edge 8-2: No cycle is formed, include it.



3. Pick edge 6-5: No cycle is formed, include it.



4. Pick edge 0-1: No cycle is formed, include it.



5. Pick edge 2-5: No cycle is formed, include it.



6. Pick edge 8-6: Since including this edge results in cycle, discard it.

7. Pick edge 2-3: No cycle is formed, include it.



8. Pick edge 7-8: Since including this edge results in cycle, discard it.

9. Pick edge 0-7: No cycle is formed, include it.



10. Pick edge 1-2: Since including this edge results in cycle, discard it.

11. Pick edge 3-4: No cycle is formed, include it.



Since the number of edges included equals ($V - 1$), the algorithm stops here.

C/C++

```
// C++ program for Kruskal's algorithm to find Minimum Spanning Tree
// of a given connected, undirected and weighted graph
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// a structure to represent a weighted edge in graph
struct Edge
{
    int src, dest, weight;
};

// a structure to represent a connected, undirected and weighted graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges. Since the graph is
    // undirected, the edge from src to dest is also edge from dest
    // to src. Both are counted as 1 edge here.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = (struct Graph*) malloc( sizeof(struct Graph) );
    graph->V = V;
    graph->E = E;

    graph->edge = (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );

    return graph;
}

// A structure to represent a subset for union-find
struct subset
{
    int parent;
    int rank;
};

// A utility function to find set of an element i
// (uses path compression technique)
int find(struct subset subsets[], int i)
{
    // find root and make root as parent of i (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high rank tree
    // (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and increment
    // its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Compare two edges according to their weights.
// Used in qsort() for sorting an array of edges
int myComp(const void* a, const void* b)
{
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
    return a1->weight > b1->weight;
}

// The main function to construct MST using Kruskal's algorithm
void KruskalMST(struct Graph* graph)
{
    int V = graph->V; // This will store the resultant MST
    int e = 0; // An index variable, used for result[]
    int i = 0; // An index variable, used for sorted edges
```

```

// Step 1: Sort all the edges in non-decreasing order of their weight
// If we are not allowed to change the given graph, we can create a copy of
// array of edges
qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

// Allocate memory for creating V ssubsets
struct subset *subsets = 
    (struct subset*) malloc( V * sizeof(struct subset) );

// Create V subsets with single elements
for (int v = 0; v < V; ++v)
{
    subsets[v].parent = v;
    subsets[v].rank = 0;
}

// Number of edges to be taken is equal to V-1
while (e < V - 1)
{
    // Step 2: Pick the smallest edge. And increment the index
    // for next iteration
    struct Edge next_edge = graph->edge[i++];

    int x = find(subsets, next_edge.src);
    int y = find(subsets, next_edge.dest);

    // If including this edge doesn't cause cycle, include it
    // in result and increment the index of result for next edge
    if (x != y)
    {
        result[e++] = next_edge;
        Union(subsets, x, y);
    }
    // Else discard the next_edge
}

// print the contents of result[] to display the built MST
printf("Following are the edges in the constructed MST\n");
for (i = 0; i < e; ++i)
    printf("%d - %d == %d\n", result[i].src, result[i].dest,
           result[i].weight);
return;
}

// Driver program to test above functions
int main()
{
    /* Let us create following weighted graph
       10
      0-----1
      | \   |
      6|  5\ |15
      |   \  |
      2-----3
          4   */
    int V = 4; // Number of vertices in graph
    int E = 5; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 10;

    // add edge 0-2
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 6;

    // add edge 0-3
    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;
    graph->edge[2].weight = 5;

    // add edge 1-3
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 15;

    // add edge 2-3
    graph->edge[4].src = 2;
    graph->edge[4].dest = 3;
    graph->edge[4].weight = 4;

    KruskalMST(graph);

    return 0;
}

```

Java

```

// Java program for Kruskal's algorithm to find Minimum Spanning Tree
// of a given connected, undirected and weighted graph
import java.util.*;
import java.lang.*;
import java.io.*;

class Graph
{
    // A class to represent a graph edge
    class Edge implements Comparable<Edge>
    {

```

```

int src, dest, weight;

// Comparator function used for sorting edges based on
// their weight
public int compareTo(Edge compareEdge)
{
    return this.weight - compareEdge.weight;
}

// A class to represent a subset for union-find
class subset
{
    int parent, rank;
};

int V, E; // V-> no. of vertices & E->no.of edges
Edge edge[]; // collection of all edges

// Creates a graph with V vertices and E edges
Graph(int v, int e)
{
    V = v;
    E = e;
    edge = new Edge[E];
    for (int i=0; i<e; ++i)
        edge[i] = new Edge();
}

// A utility function to find set of an element i
// (uses path compression technique)
int find(subset subsets[], int i)
{
    // find root and make root as parent of i (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high rank tree
    // (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and increment
    // its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// The main function to construct MST using Kruskal's algorithm
void KruskalMST()
{
    Edge result[] = new Edge[V]; // This will store the resultant MST
    int e = 0; // An index variable, used for result[]
    int i = 0; // An index variable, used for sorted edges
    for (i=0; i<V; ++i)
        result[i] = new Edge();

    // Step 1: Sort all the edges in non-decreasing order of their
    // weight. If we are not allowed to change the given graph, we
    // can create a copy of array of edges
    Arrays.sort(edge);

    // Allocate memory for creating V subsets
    subset subsets[] = new subset[V];
    for(i=0; i<V; ++i)
        subsets[i] = new subset();

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    i = 0; // Index used to pick next edge

    // Number of edges to be taken is equal to V-1
    while (e < V - 1)
    {
        // Step 2: Pick the smallest edge. And increment the index
        // for next iteration
        Edge next_edge = new Edge();
        next_edge = edge[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        // If including this edge doesn't cause cycle, include it
        // in result and increment the index of result for next edge
        if (x != y)
        {
            result[e++] = next_edge;
        }
    }
}

```

```

        Union(subsets, x, y);
    }
    // Else discard the next_edge
}

// print the contents of result[] to display the built MST
System.out.println("Following are the edges in the constructed MST");
for (i = 0; i < e; ++i)
    System.out.println(result[i].src + " -- " + result[i].dest + " == " +
                       result[i].weight);
}

// Driver Program
public static void main (String[] args)
{
    /* Let us create following weighted graph
       10
      0-----1
      | \   |
      6| 5\  |15
      | \  |
      2-----3
      4-----*/
    int V = 4; // Number of vertices in graph
    int E = 5; // Number of edges in graph
    Graph graph = new Graph(V, E);

    // add edge 0-1
    graph.edge[0].src = 0;
    graph.edge[0].dest = 1;
    graph.edge[0].weight = 10;

    // add edge 0-2
    graph.edge[1].src = 0;
    graph.edge[1].dest = 2;
    graph.edge[1].weight = 6;

    // add edge 0-3
    graph.edge[2].src = 0;
    graph.edge[2].dest = 3;
    graph.edge[2].weight = 5;

    // add edge 1-3
    graph.edge[3].src = 1;
    graph.edge[3].dest = 3;
    graph.edge[3].weight = 15;

    // add edge 2-3
    graph.edge[4].src = 2;
    graph.edge[4].dest = 3;
    graph.edge[4].weight = 4;

    graph.KruskalMST();
}
}

//This code is contributed by Aakash Hasija

```

Python

```

# Python program for Kruskal's algorithm to find Minimum Spanning Tree
# of a given connected, undirected and weighted graph

from collections import defaultdict

#Class to represent a graph
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = [] # default dictionary to store graph

        # function to add an edge to graph
    def addEdge(self,u,v,w):
        self.graph.append([u,v,w])

    # A utility function to find set of an element i
    # (uses path compression technique)
    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    # A function that does union of two sets of x and y
    # (uses union by rank)
    def union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)

        # Attach smaller rank tree under root of high rank tree
        # (Union by Rank)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        #If ranks are same, then make one as root and increment
        # its rank by one
        else :
            parent[yroot] = xroot
            rank[xroot] += 1

    # The main function to construct MST using Kruskal's algorithm
    def KruskalMST(self):

```

```

result =[] #This will store the resultant MST

i = 0 # An index variable, used for sorted edges
e = 0 # An index variable, used for result[]

#Step 1: Sort all the edges in non-decreasing order of their
# weight. If we are not allowed to change the given graph, we
# can create a copy of graph
self.graph = sorted(self.graph,key=lambda item: item[2])
#print self.graph

parent = [] ; rank = []

# Create V subsets with single elements
for node in range(self.V):
    parent.append(node)
    rank.append(0)

# Number of edges to be taken is equal to V-1
while e < self.V - 1 :

    # Step 2: Pick the smallest edge and increment the index
    # for next iteration
    u,v,w = self.graph[i]
    i = i + 1

    x = self.find(parent, u)
    y = self.find(parent, v)

    # If including this edge doesn't cause cycle, include it
    # in result and increment the index of result for next edge
    if x != y:
        e = e + 1
        result.append([u,v,w])
        self.union(parent, rank, x, y)
        # Else discard the edge

    # print the contents of result[] to display the built MST
    print "Following are the edges in the constructed MST"
    for u,v,weight in result:
        print str(u) + " -- " + str(v) + " == " + str(weight)
    print ("%"d -- %"d == %"d" % (u,v,weight))

g = Graph(4)
g.addEdge(0, 1, 10)
g.addEdge(0, 2, 6)
g.addEdge(0, 3, 5)
g.addEdge(1, 3, 15)
g.addEdge(2, 3, 4)

g.KruskalMST()

#This code is contributed by Neelam Yadav

```

Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10

Time Complexity: $O(E\log E)$ or $O(E\log V)$. Sorting of edges takes $O(E\log E)$ time. After sorting, we iterate through all edges and apply find-union algorithm. The find and union operations can take atmost $O(\log V)$ time. So overall complexity is $O(E\log E + E\log V)$ time. The value of E can be atmost $O(V^2)$, so $O(\log V)$ are $O(\log E)$ same. Therefore, overall time complexity is $O(E\log E)$ or $O(E\log V)$

References:

<http://www.ics.uci.edu/~eppstein/161/960206.html>
http://en.wikipedia.org/wiki/Minimum_spanning_tree

This article is compiled by Aashish Barnwal and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Graph
Greedy
Graph
Greedy Algorithm
Kruskal
Kruskal'sAlgorithm
MST

Greedy Algorithms | Set 3 (Huffman Coding)

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.

The variable-length codes assigned to input characters are **Prefix Codes**, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.

Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be "cccd" or "ccb" or "acd" or "ab".

See [this](#) for applications of Huffman Coding.

There are mainly two major parts in Huffman Coding

- 1) Build a Huffman Tree from input characters.
- 2) Traverse the Huffman Tree and assign codes to characters.

Steps to build Huffman Tree

Input is array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)

2. Extract two nodes with the minimum frequency from the min heap.

3. Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the

min heap.

4. Repeat steps #2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Let us understand the algorithm with an example:

character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

Step 1: Build a min heap that contains 6 nodes where each node represents root of a tree with single node.

Step 2 Extract two minimum frequency nodes from min heap. Add a new internal node with frequency $5 + 9 = 14$.



Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

character	Frequency
c	12
d	13
Internal Node	14
e	16
f	45

Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency $12 + 13 = 25$



Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes.

character	Frequency
Internal Node	14
e	16
Internal Node	25
f	45

Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency $14 + 16 = 30$



Now min heap contains 3 nodes.

character	Frequency
Internal Node	25
Internal Node	30
f	45

Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency $25 + 30 = 55$



Now min heap contains 2 nodes.

character	Frequency
f	45
Internal Node	55

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency $45 + 55 = 100$



Now min heap contains only one node.

character	Frequency
Internal Node	100

Since the heap contains only one node, the algorithm stops here.

Steps to print codes from Huffman Tree:

Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



The codes are as follows:

```
character code-word
f    0
c   100
d   101
a   1100
b   1101
e   111
```

C

```
// C program for Huffman Coding
#include <stdio.h>
#include <stdlib.h>

// This constant can be avoided by explicitly calculating height of Huffman Tree
#define MAX_TREE_HT 100

// A Huffman tree node
struct MinHeapNode
{
    char data; // One of the input characters
    unsigned freq; // Frequency of the character
    struct MinHeapNode *left, *right; // Left and right child of this node
};

// A Min Heap: Collection of min heap (or Huffman tree) nodes
struct MinHeap
{
    unsigned size; // Current size of min heap
    unsigned capacity; // Capacity of min heap
    struct MinHeapNode **array; // Array of minheap node pointers
};

// A utility function allocate a new min heap node with given character
// and frequency of the character
struct MinHeapNode* newNode(char data, unsigned freq)
{
    struct MinHeapNode* temp =
        (struct MinHeapNode*) malloc(sizeof(struct MinHeapNode));
    temp->left = temp->right = NULL;
    temp->data = data;
    temp->freq = freq;
    return temp;
}

// A utility function to create a min heap of given capacity
struct MinHeap* createMinHeap(unsigned capacity)
{
    struct MinHeap* minHeap =
        (struct MinHeap*) malloc(sizeof(struct MinHeap));
    minHeap->size = 0; // current size is 0
    minHeap->capacity = capacity;
    minHeap->array =
        (struct MinHeapNode**)malloc(minHeap->capacity * sizeof(struct MinHeapNode*));
    return minHeap;
}

// A utility function to swap two min heap nodes
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b)
{
    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

// The standard minHeapify function.
void minHeapify(struct MinHeap* minHeap, int idx)
{
    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;

    if (left < minHeap->size &&
        minHeap->array[left]->freq < minHeap->array[smallest]->freq)
        smallest = left;

    if (right < minHeap->size &&
        minHeap->array[right]->freq < minHeap->array[smallest]->freq)
        smallest = right;

    if (smallest != idx)
    {
        swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
}

// A utility function to check if size of heap is 1 or not
int isSizeOne(struct MinHeap* minHeap)
{
    return (minHeap->size == 1);
}
```

```

// A standard function to extract minimum value node from heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap)
{
    struct MinHeapNode* temp = minHeap->array[0];
    minHeap->array[0] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeapify(minHeap, 0);
    return temp;
}

// A utility function to insert a new node to Min Heap
void insertMinHeap(struct MinHeap* minHeap, struct MinHeapNode* minHeapNode)
{
    ++minHeap->size;
    int i = minHeap->size - 1;
    while (i && minHeapNode->freq < minHeap->array[(i - 1)/2]->freq)
    {
        minHeap->array[i] = minHeap->array[(i - 1)/2];
        i = (i - 1)/2;
    }
    minHeap->array[i] = minHeapNode;
}

// A standard funvtion to build min heap
void buildMinHeap(struct MinHeap* minHeap)
{
    int n = minHeap->size - 1;
    int i;
    for (i = (n - 1) / 2; i >= 0; --i)
        minHeapify(minHeap, i);
}

// A utility function to print an array of size n
void printArr(int arr[], int n)
{
    int i;
    for (i = 0; i < n; ++i)
        printf("%d", arr[i]);
    printf("\n");
}

// Utility function to check if this node is leaf
int isLeaf(struct MinHeapNode* root)
{
    return !(root->left) && !(root->right) ;
}

// Creates a min heap of capacity equal to size and inserts all character of
// data[] in min heap. Initially size of min heap is equal to capacity
struct MinHeap* createAndBuildMinHeap(char data[], int freq[], int size)
{
    struct MinHeap* minHeap = createMinHeap(size);
    for (int i = 0; i < size; ++i)
        minHeap->array[i] = newNode(data[i], freq[i]);
    minHeap->size = size;
    buildMinHeap(minHeap);
    return minHeap;
}

// The main function that builds Huffman tree
struct MinHeapNode* buildHuffmanTree(char data[], int freq[], int size)
{
    struct MinHeapNode *left, *right, *top;

    // Step 1: Create a min heap of capacity equal to size. Initially, there are
    // modes equal to size.
    struct MinHeap* minHeap = createAndBuildMinHeap(data, freq, size);

    // Iterate while size of heap doesn't become 1
    while (!isSizeOne(minHeap))
    {
        // Step 2: Extract the two minimum freq items from min heap
        left = extractMin(minHeap);
        right = extractMin(minHeap);

        // Step 3: Create a new internal node with frequency equal to the
        // sum of the two nodes frequencies. Make the two extracted node as
        // left and right children of this new node. Add this node to the min heap
        // '$' is a special value for internal nodes, not used
        top = newNode('$', left->freq + right->freq);
        top->left = left;
        top->right = right;
        insertMinHeap(minHeap, top);
    }

    // Step 4: The remaining node is the root node and the tree is complete.
    return extractMin(minHeap);
}

// Prints huffman codes from the root of Huffman Tree. It uses arr[] to
// store codes
void printCodes(struct MinHeapNode* root, int arr[], int top)
{
    // Assign 0 to left edge and recur
    if (root->left)
    {
        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }

    // Assign 1 to right edge and recur
    if (root->right)
    {
        arr[top] = 1;
        printCodes(root->right, arr, top + 1);
    }
}

```

```

// If this is a leaf node, then it contains one of the input
// characters, print the character and its code from arr[]
if (isLeaf(root))
{
    printf("%c:", root->data);
    printArr(arr, top);
}
}

// The main function that builds a Huffman Tree and print codes by traversing
// the built Huffman Tree
void HuffmanCodes(char data[], int freq[], int size)
{
    // Construct Huffman Tree
    struct MinHeapNode* root = buildHuffmanTree(data, freq, size);

    // Print Huffman codes using the Huffman tree built above
    int arr[MAX_TREE_HT], top = 0;
    printCodes(root, arr, top);
}

// Driver program to test above functions
int main()
{
    char arr[] = {'a', 'b', 'c', 'd', 'e', 'f'};
    int freq[] = {5, 9, 12, 13, 16, 45};
    int size = sizeof(arr)/sizeof(arr[0]);
    HuffmanCodes(arr, freq, size);
    return 0;
}

```

C++ using STL

```

// C++ program for Huffman Coding
#include <bits/stdc++.h>
using namespace std;

// A Huffman tree node
struct MinHeapNode
{
    char data;           // One of the input characters
    unsigned freq;       // Frequency of the character
    MinHeapNode *left, *right; // Left and right child

    MinHeapNode(char data, unsigned freq)
    {
        left = right = NULL;
        this->data = data;
        this->freq = freq;
    }
};

// For comparison of two heap nodes (needed in min heap)
struct compare
{
    bool operator()(MinHeapNode* l, MinHeapNode* r)
    {
        return (l->freq > r->freq);
    }
};

// Prints huffman codes from the root of Huffman Tree.
void printCodes(struct MinHeapNode* root, string str)
{
    if (!root)
        return;

    if (root->data != '$')
        cout << root->data << ":" << str << "\n";

    printCodes(root->left, str + "0");
    printCodes(root->right, str + "1");
}

// The main function that builds a Huffman Tree and
// print codes by traversing the built Huffman Tree
void HuffmanCodes(char data[], int freq[], int size)
{
    struct MinHeapNode *left, *right, *top;

    // Create a min heap & inserts all characters of data[]
    priority_queue<MinHeapNode*, vector<MinHeapNode*>, compare> minHeap;
    for (int i = 0; i < size; ++i)
        minHeap.push(new MinHeapNode(data[i], freq[i]));

    // Iterate while size of heap doesn't become 1
    while (minHeap.size() != 1)
    {
        // Extract the two minimum freq items from min heap
        left = minHeap.top();
        minHeap.pop();

        minHeap.pop();

        right = minHeap.top();
        minHeap.pop();

        // Create a new internal node with frequency equal to the
        // sum of the two nodes frequencies. Make the two extracted
        // node as left and right children of this new node. Add
        // this node to the min heap
        // '$' is a special value for internal nodes, not used
        top = new MinHeapNode('$', left->freq + right->freq);
        top->left = left;
        top->right = right;
    }
}

```

```

        minHeap.push(top);
    }

    // Print Huffman codes using the Huffman tree built above
    printCodes(minHeap.top(), "");
}

// Driver program to test above functions
int main()
{
    char arr[] = {'a', 'b', 'c', 'd', 'e', 'f'};
    int freq[] = {5, 9, 12, 13, 16, 45};
    int size = sizeof(arr) / sizeof(arr[0]);

    HuffmanCodes(arr, freq, size);

    return 0;
}
// This code is contributed by Aditya Goel

```

```

f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111

```

Time complexity: $O(n\log n)$ where n is the number of unique characters. If there are n nodes, $\text{extractMin}()$ is called $2*(n - 1)$ times. $\text{extractMin}()$ takes $O(\log n)$ time as it calls $\text{minHeapify}()$. So, overall complexity is $O(n\log n)$.

If the input array is sorted, there exists a linear time algorithm. We will soon be discussing in our next post.

Reference:

http://en.wikipedia.org/wiki/Huffman_coding

This article is compiled by Aashish Barnwal and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Greedy
Greedy Algorithm
Huffman Coding

Greedy Algorithms | Set 4 (Efficient Huffman Coding for Sorted Input)

We recommend to read following post as a prerequisite for this.

Greedy Algorithms | Set 3 (Huffman Coding)

Time complexity of the algorithm discussed in above post is $O(n\log n)$. If we know that the given array is sorted (by non-decreasing order of frequency), we can generate Huffman codes in $O(n)$ time. Following is a $O(n)$ algorithm for sorted input.

1. Create two empty queues.
2. Create a leaf node for each unique character and Enqueue it to the first queue in non-decreasing order of frequency. Initially second queue is empty.
3. Dequeue two nodes with the minimum frequency by examining the front of both queues. Repeat following steps two times
 -a) If second queue is empty, dequeue from first queue.
 -b) If first queue is empty, dequeue from second queue.
 -c) Else, compare the front of two queues and dequeue the minimum.
4. Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first Dequeued node as its left child and the second Dequeued node as right child. Enqueue this node to second queue.
5. Repeat steps#3 and #4 until there is more than one node in the queues. The remaining node is the root node and the tree is complete.

```

// C Program for Efficient Huffman Coding for Sorted input
#include <stdio.h>
#include <stdlib.h>

// This constant can be avoided by explicitly calculating height of Huffman Tree
#define MAX_TREE_HT 100

// A node of huffman tree
struct QueueNode
{
    char data;
    unsigned freq;
    struct QueueNode *left, *right;
};

// Structure for Queue: collection of Huffman Tree nodes (or QueueNodes)
struct Queue
{
    int front, rear;
    int capacity;
    struct QueueNode **array;
};

// A utility function to create a new QueueNode
struct QueueNode* newNode(char data, unsigned freq)
{
    struct QueueNode* temp =
        (struct QueueNode*) malloc(sizeof(struct QueueNode));
    temp->left = temp->right = NULL;
    temp->data = data;
    temp->freq = freq;
    return temp;
}

// A utility function to create a Queue of given capacity
struct Queue* createQueue(int capacity)
{

```

```

struct Queue* queue = (struct Queue*) malloc(sizeof(struct Queue));
queue->front = queue->rear = -1;
queue->capacity = capacity;
queue->array =
    (struct QueueNode**) malloc(queue->capacity * sizeof(struct QueueNode*));
return queue;
}

// A utility function to check if size of given queue is 1
int isSizeOne(struct Queue* queue)
{
    return queue->front == queue->rear && queue->front != -1;
}

// A utility function to check if given queue is empty
int isEmpty(struct Queue* queue)
{
    return queue->front == -1;
}

// A utility function to check if given queue is full
int isFull(struct Queue* queue)
{
    return queue->rear == queue->capacity - 1;
}

// A utility function to add an item to queue
void enqueue(struct Queue* queue, struct QueueNode* item)
{
    if (isFull(queue))
        return;
    queue->array[queue->rear] = item;
    if (queue->front == -1)
        ++queue->front;
}

// A utility function to remove an item from queue
struct QueueNode* dequeue(struct Queue* queue)
{
    if (isEmpty(queue))
        return NULL;
    struct QueueNode* temp = queue->array[queue->front];
    if (queue->front == queue->rear) // If there is only one item in queue
        queue->front = queue->rear = -1;
    else
        ++queue->front;
    return temp;
}

// A utility function to get front of queue
struct QueueNode* getFront(struct Queue* queue)
{
    if (isEmpty(queue))
        return NULL;
    return queue->array[queue->front];
}

/* A function to get minimum item from two queues */
struct QueueNode* findMin(struct Queue* firstQueue, struct Queue* secondQueue)
{
    // Step 3.a: If second queue is empty, dequeue from first queue
    if (isEmpty(firstQueue))
        return dequeue(secondQueue);

    // Step 3.b: If first queue is empty, dequeue from second queue
    if (isEmpty(secondQueue))
        return dequeue(firstQueue);

    // Step 3.c: Else, compare the front of two queues and dequeue minimum
    if (getFront(firstQueue)->freq < getFront(secondQueue)->freq)
        return dequeue(firstQueue);

    return dequeue(secondQueue);
}

// Utility function to check if this node is leaf
int isLeaf(struct QueueNode* root)
{
    return !(root->left) && !(root->right) ;
}

// A utility function to print an array of size n
void printArr(int arr[], int n)
{
    int i;
    for (i = 0; i < n; ++i)
        printf("%d", arr[i]);
    printf("\n");
}

// The main function that builds Huffman tree
struct QueueNode* buildHuffmanTree(char data[], int freq[], int size)
{
    struct QueueNode *left, *right, *top;

    // Step 1: Create two empty queues
    struct Queue* firstQueue = createQueue(size);
    struct Queue* secondQueue = createQueue(size);

    // Step 2: Create a leaf node for each unique character and Enqueue it to
    // the first queue in non-decreasing order of frequency. Initially second
    // queue is empty
    for (int i = 0; i < size; ++i)
        enqueue(firstQueue, newNode(data[i], freq[i]));

    // Run while Queues contain more than one node. Finally, first queue will
    // be empty and second queue will contain only one node

```

```

while (!isEmpty(firstQueue) && isSizeOne(secondQueue))
{
    // Step 3: Dequeue two nodes with the minimum frequency by examining
    // the front of both queues
    left = findMin(firstQueue, secondQueue);
    right = findMin(firstQueue, secondQueue);

    // Step 4: Create a new internal node with frequency equal to the sum
    // of the two nodes frequencies. Enqueue this node to second queue.
    top = newNode('$', left->freq + right->freq);
    top->left = left;
    top->right = right;
    enQueue(secondQueue, top);
}

return deQueue(secondQueue);
}

// Prints huffman codes from the root of Huffman Tree. It uses arr[] to
// store codes
void printCodes(struct QueueNode* root, int arr[], int top)
{
    // Assign 0 to left edge and recur
    if (root->left)
    {
        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }

    // Assign 1 to right edge and recur
    if (root->right)
    {
        arr[top] = 1;
        printCodes(root->right, arr, top + 1);
    }

    // If this is a leaf node, then it contains one of the input
    // characters, print the character and its code from arr[]
    if (isLeaf(root))
    {
        printf("%c: ", root->data);
        printArr(arr, top);
    }
}

// The main function that builds a Huffman Tree and print codes by traversing
// the built Huffman Tree
void HuffmanCodes(char data[], int freq[], int size)
{
    // Construct Huffman Tree
    struct QueueNode* root = buildHuffmanTree(data, freq, size);

    // Print Huffman codes using the Huffman tree built above
    int arr[MAX_TREE_HT], top = 0;
    printCodes(root, arr, top);
}

// Driver program to test above functions
int main()
{
    char arr[] = {'a', 'b', 'c', 'd', 'e', 'f'};
    int freq[] = {5, 9, 12, 13, 16, 45};
    int size = sizeof(arr)/sizeof(arr[0]);
    HuffmanCodes(arr, freq, size);
    return 0;
}

```

Output:

```

f:0
c:100
d:101
a:1100
b:1101
e:111

```

Time complexity: O(n)

If the input is not sorted, it need to be sorted first before it can be processed by the above algorithm. Sorting can be done using heap-sort or merge-sort both of which run in Theta(nlogn). So, the overall time complexity becomes O(nlogn) for unsorted input.

Reference:

http://en.wikipedia.org/wiki/Huffman_coding

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Greedy
Greedy Algorithm
Huffman Coding

Greedy Algorithms | Set 5 (Prim's Minimum Spanning Tree (MST))

We have discussed [Kruskal's algorithm](#) for [Minimum Spanning Tree](#). Like Kruskal's algorithm, Prim's algorithm is also a [Greedy algorithm](#). It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

A group of edges that connects two set of vertices in a graph is called [cut in graph theory](#). So, at every step of Prim's algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the vertices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).

How does Prim's Algorithm Work? The idea behind Prim's algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a [Spanning Tree](#). And they must be connected with the minimum weight edge to make it a [Minimum Spanning Tree](#).

Algorithm

- 1) Create a set *mstSet* that keeps track of vertices already included in MST.
- 2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
- 3) While *mstSet* doesn't include all vertices
 -a) Pick a vertex *u* which is not there in *mstSet* and has minimum key value.
 -b) Include *u* to *mstSet*.
 -c) Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u-v* is less than the previous key value of *v*, update the key value as weight of *u-v*

The idea of using key values is to pick the minimum weight edge from *cut*. The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

Let us understand with the following example:



0

The set *mstSet* is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum key value. The vertex 0 is picked, include it in *mstSet*. So *mstSet* becomes {0}. After including to *mstSet*, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.



0
1

Pick the vertex with minimum key value and not already included in MST (not in *mstSET*). The vertex 1 is picked and added to *mstSet*. So *mstSet* now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.



0
1
2

Pick the vertex with minimum key value and not already included in MST (not in *mstSET*). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So *mstSet* now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (7 and 1 respectively).



0
1
6
7

Pick the vertex with minimum key value and not already included in MST (not in *mstSET*). Vertex 6 is picked. So *mstSet* now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.



0
1
5
6
7

We repeat the above steps until *mstSet* includes all vertices of given graph. Finally, we get the following graph.



0
1
2
3
4
5
6
7

How to implement the above algorithm?

We use a boolean array *mstSet[]* to represent the set of vertices included in MST. If a value *mstSet[v]* is true, then vertex *v* is included in MST, otherwise not. Array *key[]* is used to store key values of all vertices. Another array *parent[]* to store indexes of parent nodes in MST. The parent array is the output array which is used to show the constructed MST.

C/C++

```
// A C / C++ program for Prim's Minimum Spanning Tree (MST) algorithm.  
// The program is for adjacency matrix representation of the graph  
  
#include <stdio.h>  
#include <limits.h>  
  
// Number of vertices in the graph  
#define V 5  
  
// A utility function to find the vertex with minimum key value, from  
// the set of vertices not yet included in MST  
int minKey(int key[], bool mstSet[])  
{  
    // Initialize min value  
    int min = INT_MAX, min_index;  
  
    for (int v = 0; v < V; v++)  
        if (mstSet[v] == false && key[v] < min)
```

```

min = key[v], min_index = v;
return min_index;
}

// A utility function to print the constructed MST stored in parent[]
int printMST(int parent[], int n, int graph[V][V])
{
    printf("Edge  Weight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d  %d\n", parent[i], i, graph[i][parent[i]]);

}

// Function to construct and print MST for a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
    int parent[V]; // Array to store constructed MST
    int key[V]; // Key values used to pick minimum weight edge in cut
    bool mstSet[V]; // To represent set of vertices not yet included in MST

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    key[0] = 0; // Make key 0 so that this vertex is picked as first vertex
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V-1; count++)
    {
        // Pick the minimum key vertex from the set of vertices
        // not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of the adjacent vertices of
        // the picked vertex. Consider only those vertices which are not yet
        // included in MST
        for (int v = 0; v < V; v++)
            if (graph[u][v] != 0 && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // print the constructed MST
    printMST(parent, V, graph);
}

// driver program to test above function
int main()
{
    /* Let us create the following graph
     *   2   3
     * (0)--(1)--(2)
     * |   / \   |
     * 6| 8 /   \ 5 |7
     * |   \   |
     * (3)-----(4)
     * 9           *
     */
    int graph[V][V] = {{0, 2, 0, 6, 0},
                       {2, 0, 3, 8, 5},
                       {0, 3, 0, 0, 7},
                       {6, 8, 0, 0, 9},
                       {0, 5, 7, 9, 0},
                       };

    // Print the solution
    primMST(graph);

    return 0;
}

```

Java

```

// A Java program for Prim's Minimum Spanning Tree (MST) algorithm.
// The program is for adjacency matrix representation of the graph

import java.util.*;
import java.lang.*;
import java.io.*;

class MST
{
    // Number of vertices in the graph
    private static final int V=5;

    // A utility function to find the vertex with minimum key
    // value, from the set of vertices not yet included in MST
    int minKey(int key[], Boolean mstSet[])
    {
        // Initialize min value
        int min = Integer.MAX_VALUE, min_index=-1;

        for (int v = 0; v < V; v++)
            if (mstSet[v] == false && key[v] < min)
                {

```

```

        min = key[v];
        min_index = v;
    }

    return min_index;
}

// A utility function to print the constructed MST stored in
// parent[]
void printMST(int parent[], int n, int graph[][])
{
    System.out.println("Edge Weight");
    for (int i = 1; i < V; i++)
        System.out.println(parent[i] + " - " + i + " " +
                           graph[i][parent[i]]);
}

// Function to construct and print MST for a graph represented
// using adjacency matrix representation
void primMST(int graph[][])
{
    // Array to store constructed MST
    int parent[] = new int[V];

    // Key values used to pick minimum weight edge in cut
    int key[] = new int [V];

    // To represent set of vertices not yet included in MST
    Boolean mstSet[] = new Boolean[V];

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
    {
        key[i] = Integer.MAX_VALUE;
        mstSet[i] = false;
    }

    // Always include first 1st vertex in MST.
    key[0] = 0; // Make key 0 so that this vertex is
                // picked as first vertex
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V-1; count++)
    {
        // Pick thd minimum key vertex from the set of vertices
        // not yet included in MST
        int u = minKey(key, mstSet);

        // Add the picked vertex to the MST Set
        mstSet[u] = true;

        // Update key value and parent index of the adjacent
        // vertices of the picked vertex. Consider only those
        // vertices which are not yet included in MST
        for (int v = 0; v < V; v++)

            // graph[u][v] is non zero only for adjacent vertices of m
            // mstSet[v] is false for vertices not yet included in MST
            // Update the key only if graph[u][v] is smaller than key[v]
            if (graph[u][v]>0 && mstSet[v] == false &&
                graph[u][v] < key[v])
            {
                parent[v] = u;
                key[v] = graph[u][v];
            }
    }

    // print the constructed MST
    printMST(parent, V, graph);
}
}

public static void main (String[] args)
{
    /* Let us create the following graph
     2 3
     (0)-(1)-(2)
     | \ |
     6 8 / 5 | 7
     | / \ |
     (3)-----(4)
     9         *
    MST t = new MST();
    int graph[][] = new int[][] {{0, 2, 0, 6, 0},
                                {2, 0, 3, 8, 5},
                                {0, 3, 0, 0, 7},
                                {6, 8, 0, 0, 9},
                                {0, 5, 7, 9, 0},
                                };
    */

    // Print the solution
    t.primMST(graph);
}
}

// This code is contributed by Aakash Hasija

```

Output:

Edge	Weight
0 - 1	2
1 - 2	3
0 - 3	6
0 - 4	5

Time Complexity of the above program is $O(V^2)$. If the input graph is represented using adjacency list, then the time complexity of Prim's algorithm can be reduced to $O(E \log V)$ with the help of binary

heap. Please see [Prim's MST for Adjacency List Representation](#) for more details.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Graph
Greedy
Greedy Algorithm
Minimum Spanning Tree
Prim's Algorithm.MST

Greedy Algorithms | Set 6 (Prim's MST for Adjacency List Representation)

We recommend to read following two posts as a prerequisite of this post.

1. [Greedy Algorithms | Set 5 \(Prim's Minimum Spanning Tree \(MST\)\)](#)
2. [Graph and its representations](#)

We have discussed Prim's algorithm and its implementation for adjacency matrix representation of graphs. The time complexity for the matrix representation is $O(V^2)$. In this post, $O(E\log V)$ algorithm for adjacency list representation is discussed.

As discussed in the previous post, in Prim's algorithm, two sets are maintained, one set contains list of vertices already included in MST, other set contains vertices not yet included. With adjacency list representation, all vertices of a graph can be traversed in $O(V+E)$ time using BFS. The idea is to traverse all vertices of graph using BFS and use a Min Heap to store the vertices not yet included in MST. Min Heap is used as a priority queue to get the minimum weight edge from the cut. Min Heap is used as time complexity of operations like extracting minimum element and decreasing key value is $O(\log V)$ in Min Heap.

Following are the detailed steps.

- 1) Create a Min Heap of size V where V is the number of vertices in the given graph. Every node of min heap contains vertex number and key value of the vertex.
- 2) Initialize Min Heap with first vertex as root (the key value assigned to first vertex is 0). The key value assigned to all other vertices is INF (infinite).
- 3) While Min Heap is not empty, do following
 -a) Extract the min value node from Min Heap. Let the extracted vertex be u.
 -b) For every adjacent vertex v of u, check if v is in Min Heap (not yet included in MST). If v is in Min Heap and its key value is more than weight of u-v, then update the key value of v as weight of u-v.

Let us understand the above algorithm with the following example:



Initially, key value of first vertex is 0 and INF (infinite) for all other vertices. So vertex 0 is extracted from Min Heap and key values of vertices adjacent to 0 (1 and 7) are updated. Min Heap contains all vertices except vertex 0.

The vertices in green color are the vertices included in MST.



Since key value of vertex 1 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 1 are updated (Key is updated if the a vertex is not in Min Heap and previous key value is greater than the weight of edge from 1 to the adjacent). Min Heap contains all vertices except vertex 0 and 1.



Since key value of vertex 7 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 7 are updated (Key is updated if the a vertex is not in Min Heap and previous key value is greater than the weight of edge from 7 to the adjacent). Min Heap contains all vertices except vertex 0, 1 and 7.



Since key value of vertex 6 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 6 are updated (Key is updated if the a vertex is not in Min Heap and previous key value is greater than the weight of edge from 6 to the adjacent). Min Heap contains all vertices except vertex 0, 1, 7 and 6.



The above steps are repeated for rest of the nodes in Min Heap till Min Heap becomes empty



```
// C / C++ program for Prim's MST for adjacency list representation of graph
```

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a node in adjacency list
struct AdjListNode
{
    int dest;
    int weight;
    struct AdjListNode* next;
};

// A structure to represent an adjacency list
struct AdjList
{
    struct AdjListNode *head; // pointer to head node of list
};

// A structure to represent a graph. A graph is an array of adjacency lists.
// Size of array will be V (number of vertices in graph)
struct Graph
{
    int V;
    struct AdjList* array;
};

// A utility function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest, int weight)
{
    struct AdjListNode* newNode =
        (struct AdjListNode*) malloc(sizeof(struct AdjListNode));
    newNode->dest = dest;
    newNode->weight = weight;
    newNode->next = NULL;
    return newNode;
}

// A utility function that creates a graph of V vertices
struct Graph* createGraph(int V)
{
    struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
    graph->V = V;

    // Create an array of adjacency lists. Size of array will be V
    graph->array = (struct AdjList*) malloc(V * sizeof(struct AdjList));

    // Initialize each adjacency list as empty by making head as NULL
    for (int i = 0; i < V; ++i)
        graph->array[i].head = NULL;

    return graph;
}

// Adds an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest, int weight)
{
    // Add an edge from src to dest. A new node is added to the adjacency
    // list of src. The node is added at the beginning
    struct AdjListNode* newNode = newAdjListNode(dest, weight);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;

    // Since graph is undirected, add an edge from dest to src also
    newNode = newAdjListNode(src, weight);
    newNode->next = graph->array[dest].head;
    graph->array[dest].head = newNode;
}

// Structure to represent a min heap node
struct MinHeapNode
{
    int v;
    int key;
};

// Structure to represent a min heap
struct MinHeap
{
    int size; // Number of heap nodes present currently
    int capacity; // Capacity of min heap
    int *pos; // This is needed for decreaseKey()
    struct MinHeapNode **array;
};

// A utility function to create a new Min Heap Node
struct MinHeapNode* newMinHeapNode(int v, int key)
```

```

{
    struct MinHeapNode* minHeapNode =
        (struct MinHeapNode*) malloc(sizeof(struct MinHeapNode));
    minHeapNode->v = v;
    minHeapNode->key = key;
    return minHeapNode;
}

// A utilit function to create a Min Heap
struct MinHeap* createMinHeap(int capacity)
{
    struct MinHeap* minHeap =
        (struct MinHeap*) malloc(sizeof(struct MinHeap));
    minHeap->pos = (int *)malloc(capacity * sizeof(int));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array =
        (struct MinHeapNode**) malloc(capacity * sizeof(struct MinHeapNode*));
    return minHeap;
}

// A utility function to swap two nodes of min heap. Needed for min heapify
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b)
{
    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

// A standard function to heapify at given idx
// This function also updates position of nodes when they are swapped.
// Position is needed for decreaseKey()
void minHeapify(struct MinHeap* minHeap, int idx)
{
    int smallest, left, right;
    smallest = idx;
    left = 2 * idx + 1;
    right = 2 * idx + 2;

    if (left < minHeap->size &&
        minHeap->array[left]->key < minHeap->array[smallest]->key )
        smallest = left;

    if (right < minHeap->size &&
        minHeap->array[right]->key < minHeap->array[smallest]->key )
        smallest = right;

    if (smallest != idx)
    {
        // The nodes to be swapped in min heap
        MinHeapNode *smallestNode = minHeap->array[smallest];
        MinHeapNode *idxNode = minHeap->array[idx];

        // Swap positions
        minHeap->pos[smallestNode->v] = idx;
        minHeap->pos[idxNode->v] = smallest;

        // Swap nodes
        swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

        minHeapify(minHeap, smallest);
    }
}

// A utility function to check if the given minHeap is ampty or not
int isEmpty(struct MinHeap* minHeap)
{
    return minHeap->size == 0;
}

// Standard function to extract minimum node from heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap)
{
    if (isEmpty(minHeap))
        return NULL;

    // Store the root node
    struct MinHeapNode* root = minHeap->array[0];

    // Replace root node with last node
    struct MinHeapNode* lastNode = minHeap->array[minHeap->size - 1];
    minHeap->array[0] = lastNode;

    // Update position of last node
    minHeap->pos[root->v] = minHeap->size-1;
    minHeap->pos[lastNode->v] = 0;

    // Reduce heap size and heapify root
    --minHeap->size;
    minHeapify(minHeap, 0);

    return root;
}

// Function to decreasy key value of a given vertex v. This function
// uses pos[] of min heap to get the current index of node in min heap
void decreaseKey(struct MinHeap* minHeap, int v, int key)
{
    // Get the index of v in heap array
    int i = minHeap->pos[v];

    // Get the node and update its key value
    minHeap->array[i]->key = key;

    // Travel up while the complete tree is not heepified.
    // This is a O(Logn) loop
    while (i && minHeap->array[i]->key < minHeap->array[(i - 1) / 2]->key)
}

```

```

{
    // Swap this node with its parent
    minHeap->pos[minHeap->array[i]->v] = (i-1)/2;
    minHeap->pos[minHeap->array[(i-1)/2]->v] = i;
    swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

    // move to parent index
    i = (i - 1) / 2;
}
}

// A utility function to check if a given vertex
// 'v' is in min heap or not
bool isInMinHeap(struct MinHeap *minHeap, int v)
{
    if (minHeap->pos[v] < minHeap->size)
        return true;
    return false;
}

// A utility function used to print the constructed MST
void printArr(int arr[], int n)
{
    for (int i = 1; i < n; ++i)
        printf("%d - %d\n", arr[i], i);
}

// The main function that constructs Minimum Spanning Tree (MST)
// using Prim's algorithm
void PrimMST(struct Graph* graph)
{
    int V = graph->V;// Get the number of vertices in graph
    int parent[V]; // Array to store constructed MST
    int key[V]; // Key values used to pick minimum weight edge in cut

    // minHeap represents set E
    struct MinHeap* minHeap = createMinHeap(V);

    // Initialize min heap with all vertices. Key value of
    // all vertices (except 0th vertex) is initially infinite
    for (int v = 1; v < V; ++v)
    {
        parent[v] = -1;
        key[v] = INT_MAX;
        minHeap->array[v] = newMinHeapNode(v, key[v]);
        minHeap->pos[v] = v;
    }

    // Make key value of 0th vertex as 0 so that it
    // is extracted first
    key[0] = 0;
    minHeap->array[0] = newMinHeapNode(0, key[0]);
    minHeap->pos[0] = 0;

    // Initially size of min heap is equal to V
    minHeap->size = V;

    // In the followin loop, min heap contains all nodes
    // not yet added to MST.
    while (!isEmpty(minHeap))
    {
        // Extract the vertex with minimum key value
        struct MinHeapNode* minHeapNode = extractMin(minHeap);
        int u = minHeapNode->v; // Store the extracted vertex number

        // Traverse through all adjacent vertices of u (the extracted
        // vertex) and update their key values
        struct AdjListNode* pCrawl = graph->array[u].head;
        while (pCrawl != NULL)
        {
            int v = pCrawl->dest;

            // If v is not yet included in MST and weight of u-v is
            // less than key value of v, then update key value and
            // parent of v
            if (isInMinHeap(minHeap, v) && pCrawl->weight < key[v])
            {
                key[v] = pCrawl->weight;
                parent[v] = u;
                decreaseKey(minHeap, v, key[v]);
            }
            pCrawl = pCrawl->next;
        }
    }

    // print edges of MST
    printArr(parent, V);
}

// Driver program to test above functions
int main()
{
    // Let us create the graph given in above fugure
    int V = 9;
    struct Graph* graph = createGraph(V);
    addEdge(graph, 0, 1, 4);
    addEdge(graph, 0, 7, 8);
    addEdge(graph, 1, 2, 8);
    addEdge(graph, 1, 7, 11);
    addEdge(graph, 2, 3, 7);
    addEdge(graph, 2, 8, 2);
    addEdge(graph, 2, 5, 4);
    addEdge(graph, 3, 4, 9);
    addEdge(graph, 3, 5, 14);
    addEdge(graph, 4, 5, 10);
    addEdge(graph, 5, 6, 2);
    addEdge(graph, 6, 7, 1);
}

```

```

    addEdge(graph, 6, 8, 6);
    addEdge(graph, 7, 8, 7);

    PrimMST(graph);

    return 0;
}

```

Output:

```

0 - 1
5 - 2
2 - 3
3 - 4
6 - 5
7 - 6
0 - 7
2 - 8

```

Time Complexity: The time complexity of the above code/algorithm looks $O(V^2)$ as there are two nested while loops. If we take a closer look, we can observe that the statements in inner loop are executed $O(V+E)$ times (similar to BFS). The inner loop has `decreaseKey()` operation which takes $O(\log V)$ time. So overall time complexity is $O(E+V) \cdot O(\log V)$ which is $O((E+V) \cdot \log V) = O(E \log V)$ (For a connected graph, $V = O(E)$)

References:

Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

http://en.wikipedia.org/wiki/Prim's_algorithm

This article is compiled by Aashish Barnwal and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Graph
Greedy
Graph
Greedy Algorithm
MST

Greedy Algorithms | Set 7 (Dijkstra's shortest path algorithm)

Given a graph and a source vertex in graph, find shortest paths from source to all vertices in the given graph.

Dijkstra's algorithm is very similar to [Prim's algorithm for minimum spanning tree](#). Like Prim's MST, we generate a *SPT (shortest path tree)* with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has minimum distance from source.

Below are the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Algorithm

1) Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

3) While *sptSet* doesn't include all vertices

....a) Pick a vertex *u* which is not there in *sptSet* and has minimum distance value.

....b) Include *u* to *sptSet*.

....c) Update distance value of all adjacent vertices of *u*. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex *v*, if sum of distance value of *u* (from source) and weight of edge *u-v*, is less than the distance value of *v*, then update the distance value of *v*.

Let us understand with the following example:



The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green color.



Pick the vertex with minimum distance value and not already included in SPT (not in *sptSET*). The vertex 1 is picked and added to *sptSet*. So *sptSet* now becomes {0, 1}. Update the distance values of

adjacent vertices of 1. The distance value of vertex 2 becomes 12.



Pick the vertex with minimum distance value and not already included in SPT (not in sptSet). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance value and not already included in SPT (not in sptSet). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until sptSet doesn't include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).



How to implement the above algorithm?

We use a boolean array sptSet[] to represent the set of vertices included in SPT. If a value sptSet[v] is true, then vertex v is included in SPT, otherwise not. Array dist[] is used to store shortest distance values of all vertices.

C/C++

```
// A C / C++ program for Dijkstra's single source shortest path algorithm.  
// The program is for adjacency matrix representation of the graph
```

```
#include <stdio.h>  
#include <limits.h>  
  
// Number of vertices in the graph  
#define V 9  
  
// A utility function to find the vertex with minimum distance value, from  
// the set of vertices not yet included in shortest path tree  
int minDistance(int dist[], bool sptSet[])  
{  
    // Initialize min value  
    int min = INT_MAX, min_index;  
  
    for (int v = 0; v < V; v++)  
        if (sptSet[v] == false && dist[v] <= min)  
            min = dist[v], min_index = v;  
  
    return min_index;  
}  
  
// A utility function to print the constructed distance array  
int printSolution(int dist[], int n)  
{  
    printf("Vertex Distance from Source\n");  
    for (int i = 0; i < V; i++)  
        printf("%d \t %d\n", i, dist[i]);  
}  
  
// Function that implements Dijkstra's single source shortest path algorithm  
// for a graph represented using adjacency matrix representation  
void dijkstra(int graph[V][V], int src)  
{  
    int dist[V]; // The output array. dist[i] will hold the shortest  
    // distance from src to i  
  
    bool sptSet[V]; // sptSet[i] will true if vertex i is included in shortest  
    // path tree or shortest distance from src to i is finalized  
  
    // Initialize all distances as INFINITE and sptSet[] as false  
    for (int i = 0; i < V; i++)  
        dist[i] = INT_MAX, sptSet[i] = false;  
  
    // Distance of source vertex from itself is always 0  
    dist[src] = 0;  
  
    // Find shortest path for all vertices  
    for (int count = 0; count < V-1; count++)  
    {  
        // Pick the minimum distance vertex from the set of vertices not  
        // yet processed. u is always equal to src in first iteration.  
        int u = minDistance(dist, sptSet);
```

```

// Mark the picked vertex as processed
sptSet[u] = true;

// Update dist value of the adjacent vertices of the picked vertex.
for (int v = 0; v < V; v++)

    // Update dist[v] only if it is not in sptSet, there is an edge from
    // u to v, and total weight of path from src to v through u is
    // smaller than current value of dist[v]
    if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
        && dist[u]+graph[u][v] < dist[v])
        dist[v] = dist[u] + graph[u][v];
    }

// print the constructed distance array
printSolution(dist, V);
}

// driver program to test above function
int main()
{
    /* Let us create the example graph discussed above */
    int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},
    {4, 0, 8, 0, 0, 0, 0, 11, 0},
    {0, 8, 0, 7, 0, 4, 0, 0, 2},
    {0, 0, 7, 0, 9, 14, 0, 0, 0},
    {0, 0, 0, 9, 0, 10, 0, 0, 0},
    {0, 0, 4, 14, 10, 0, 2, 0, 0},
    {0, 0, 0, 0, 2, 0, 1, 6},
    {8, 11, 0, 0, 0, 0, 1, 0, 7},
    {0, 0, 2, 0, 0, 0, 6, 7, 0}
    };

    dijkstra(graph, 0);

    return 0;
}

```

Java

```

// A Java program for Dijkstra's single source shortest path algorithm.
// The program is for adjacency matrix representation of the graph
import java.util.*;
import java.lang.*;
import java.io.*;

class ShortestPath
{
    // A utility function to find the vertex with minimum distance value,
    // from the set of vertices not yet included in shortest path tree
    static final int V=9;
    int minDistance(int dist[], Boolean sptSet[])
    {
        // Initialize min value
        int min = Integer.MAX_VALUE, min_index=-1;

        for (int v = 0; v < V; v++)
            if (sptSet[v] == false && dist[v] <= min)
            {
                min = dist[v];
                min_index = v;
            }

        return min_index;
    }

    // A utility function to print the constructed distance array
    void printSolution(int dist[], int n)
    {
        System.out.println("Vertex Distance from Source");
        for (int i = 0; i < V; i++)
            System.out.println(i+" "+dist[i]);
    }

    // Function that implements Dijkstra's single source shortest path
    // algorithm for a graph represented using adjacency matrix
    // representation
    void dijkstra(int graph[][], int src)
    {
        int dist[] = new int[V]; // The output array. dist[i] will hold
                               // the shortest distance from src to i

        // sptSet[i] will true if vertex i is included in shortest
        // path tree or shortest distance from src to i is finalized
        Boolean sptSet[] = new Boolean[V];

        // Initialize all distances as INFINITE and sptSet[] as false
        for (int i = 0; i < V; i++)
        {
            dist[i] = Integer.MAX_VALUE;
            sptSet[i] = false;
        }

        // Distance of source vertex from itself is always 0
        dist[src] = 0;

        // Find shortest path for all vertices
        for (int count = 0; count < V-1; count++)
        {
            // Pick the minimum distance vertex from the set of vertices
            // not yet processed. u is always equal to src in first
            // iteration.
            int u = minDistance(dist, sptSet);

```

```

// Mark the picked vertex as processed
spSet[u] = true;

// Update dist value of the adjacent vertices of the
// picked vertex.
for (int v = 0; v < V; v++)

    // Update dist[v] only if is not in spSet, there is an
    // edge from u to v, and total weight of path from src to
    // v through u is smaller than current value of dist[v]
    if (!spSet[v] && graph[u][v]==0 &&
        dist[u]==Integer.MAX_VALUE &&
        dist[u]+graph[u][v] < dist[v])
        dist[v] = dist[u] + graph[u][v];
    }

    // print the constructed distance array
    printSolution(dist, V);
}

// Driver method
public static void main (String[] args)
{
/* Let us create the example graph discussed above */
int graph[][] = new int[][]{{0, 4, 0, 0, 0, 0, 8, 0},
                           {4, 0, 8, 0, 0, 0, 0, 11, 0},
                           {0, 8, 0, 7, 0, 4, 0, 0, 2},
                           {0, 0, 7, 0, 9, 14, 0, 0, 0},
                           {0, 0, 0, 9, 0, 10, 0, 0, 0},
                           {0, 0, 4, 14, 10, 0, 2, 0, 0},
                           {0, 0, 0, 0, 2, 0, 1, 6},
                           {8, 11, 0, 0, 0, 0, 1, 0, 7},
                           {0, 0, 2, 0, 0, 0, 6, 7, 0}
                           };
ShortestPath t = new ShortestPath();
t.dijkstra(graph, 0);
}
}

//This code is contributed by Aakash Hasija

```

Output:

Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14

Notes:

- 1) The code calculates shortest distance, but doesn't calculate the path information. We can create a parent array, update the parent array when distance is updated (like prim's implementation) and use it to show the shortest path from source to different vertices.
 - 2) The code is for undirected graph, same dijkstra function can be used for directed graphs also.
 - 3) The code finds shortest distances from source to all vertices. If we are interested only in shortest distance from source to a single target, we can break the for loop when the picked minimum distance vertex is equal to target (Step 3.a of algorithm).
 - 4) Time Complexity of the implementation is $O(V^2)$. If the input graph is represented using adjacency list, it can be reduced to $O(E \log V)$ with the help of binary heap. Please see [Dijkstra's Algorithm for Adjacency List Representation](#) for more details.
 - 5) Dijkstra's algorithm doesn't work for graphs with negative weight edges. For graphs with negative weight edges, [Bellman-Ford algorithm](#) can be used, we will soon be discussing it as a separate post.
- [Dijkstra's Algorithm for Adjacency List Representation](#)
- [Printing Paths in Dijkstra's Shortest Path Algorithm](#)
- [Dijkstra's shortest path algorithm using set in STL](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Graph
Greedy
Dijkstra
Graph
Greedy Algorithm
shortest path

Greedy Algorithms | Set 8 (Dijkstra's Algorithm for Adjacency List Representation)

We recommend to read following two posts as a prerequisite of this post.

1. Greedy Algorithms | Set 7 (Dijkstra's shortest path algorithm)
2. Graph and its representations

We have discussed Dijkstra's algorithm and its implementation for adjacency matrix representation of graphs. The time complexity for the matrix representation is $O(V^2)$. In this post, $O(E \log V)$ algorithm for adjacency list representation is discussed.

As discussed in the previous post, in Dijkstra's algorithm, two sets are maintained, one set contains list of vertices already included in SPT (Shortest Path Tree), other set contains vertices not yet included. With adjacency list representation, all vertices of a graph can be traversed in $O(V+E)$ time using BFS. The idea is to traverse all vertices of graph using BFS and use a Min Heap to store the vertices not yet included in SPT (or the vertices for which shortest distance is not finalized yet). Min Heap is used as a priority queue to get the minimum distance vertex from set of not yet included vertices. Time complexity of operations like extract-min and decrease-key value is $O(\log V)$ for Min Heap.

Following are the detailed steps.

- 1) Create a Min Heap of size V where V is the number of vertices in the given graph. Every node of min heap contains vertex number and distance value of the vertex.
- 2) Initialize Min Heap with source vertex as root (the distance value assigned to source vertex is 0). The distance value assigned to all other vertices is INF (infinite).
- 3) While Min Heap is not empty, do following
.....a) Extract the vertex with minimum distance value node from Min Heap. Let the extracted vertex be u.
.....b) For every adjacent vertex v of u, check if v is in Min Heap. If v is in Min Heap and distance value is more than weight of u-v plus distance value of u, then update the distance value of v.

Let us understand with the following example. Let the given source vertex be 0



Initially, distance value of source vertex is 0 and INF (infinite) for all other vertices. So source vertex is extracted from Min Heap and distance values of vertices adjacent to 0 (1 and 7) are updated. Min Heap contains all vertices except vertex 0.

The vertices in green color are the vertices for which minimum distances are finalized and are not in Min Heap



Since distance value of vertex 1 is minimum among all nodes in Min Heap, it is extracted from Min Heap and distance values of vertices adjacent to 1 are updated (distance is updated if the a vertex is not in Min Heap and distance through 1 is shorter than the previous distance). Min Heap contains all vertices except vertex 0 and 1.



Pick the vertex with minimum distance value from min heap. Vertex 7 is picked. So min heap now contains all vertices except 0, 1 and 7. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance from min heap. Vertex 6 is picked. So min heap now contains all vertices except 0, 1, 7 and 6. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



Above steps are repeated till min heap doesn't become empty. Finally, we get the following shortest path tree.



```
// C / C++ program for Dijkstra's shortest path algorithm for adjacency
// list representation of graph

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a node in adjacency list
struct AdjListNode
{
    int dest;
    int weight;
    struct AdjListNode* next;
};

// A structure to represent an adjacency list
struct AdjList
{
    struct AdjListNode* head;
};
```

```

struct AdjList
{
    struct AdjListNode *head; // pointer to head node of list
};

// A structure to represent a graph. A graph is an array of adjacency lists.
// Size of array will be V (number of vertices in graph)
struct Graph
{
    int V;
    struct AdjList* array;
};

// A utility function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest, int weight)
{
    struct AdjListNode* newNode =
        (struct AdjListNode*) malloc(sizeof(struct AdjListNode));
    newNode->dest = dest;
    newNode->weight = weight;
    newNode->next = NULL;
    return newNode;
}

// A utility function that creates a graph of V vertices
struct Graph* createGraph(int V)
{
    struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
    graph->V = V;

    // Create an array of adjacency lists. Size of array will be V
    graph->array = (struct AdjList*) malloc(V * sizeof(struct AdjList));

    // Initialize each adjacency list as empty by making head as NULL
    for (int i = 0; i < V; ++i)
        graph->array[i].head = NULL;

    return graph;
}

// Adds an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest, int weight)
{
    // Add an edge from src to dest. A new node is added to the adjacency
    // list of src. The node is added at the beginning
    struct AdjListNode* newNode = newAdjListNode(dest, weight);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;

    // Since graph is undirected, add an edge from dest to src also
    newNode = newAdjListNode(src, weight);
    newNode->next = graph->array[dest].head;
    graph->array[dest].head = newNode;
}

// Structure to represent a min heap node
struct MinHeapNode
{
    int v;
    int dist;
};

// Structure to represent a min heap
struct MinHeap
{
    int size; // Number of heap nodes present currently
    int capacity; // Capacity of min heap
    int *pos; // This is needed for decreaseKey()
    struct MinHeapNode **array;
};

// A utility function to create a new Min Heap Node
struct MinHeapNode* newMinHeapNode(int v, int dist)
{
    struct MinHeapNode* minHeapNode =
        (struct MinHeapNode*) malloc(sizeof(struct MinHeapNode));
    minHeapNode->v = v;
    minHeapNode->dist = dist;
    return minHeapNode;
}

// A utility function to create a Min Heap
struct MinHeap* createMinHeap(int capacity)
{
    struct MinHeap* minHeap =
        (struct MinHeap*) malloc(sizeof(struct MinHeap));
    minHeap->pos = (int *)malloc(capacity * sizeof(int));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array =
        (struct MinHeapNode**) malloc(capacity * sizeof(struct MinHeapNode)));
    return minHeap;
}

// A utility function to swap two nodes of min heap. Needed for min heapify
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b)
{
    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

// A standard function to heapify at given idx
// This function also updates position of nodes when they are swapped.
// Position is needed for decreaseKey()
void minHeapify(struct MinHeap* minHeap, int idx)
{
}

```

```

int smallest, left, right;
smallest = idx;
left = 2 * idx + 1;
right = 2 * idx + 2;

if (left < minHeap->size &&
    minHeap->array[left]->dist < minHeap->array[smallest]->dist)
    smallest = left;

if (right < minHeap->size &&
    minHeap->array[right]->dist < minHeap->array[smallest]->dist)
    smallest = right;

if (smallest != idx)
{
    // The nodes to be swapped in min heap
    MinHeapNode *smallestNode = minHeap->array[smallest];
    MinHeapNode *idxNode = minHeap->array[idx];

    // Swap positions
    minHeap->pos[smallestNode->v] = idx;
    minHeap->pos[idxNode->v] = smallest;

    // Swap nodes
    swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

    minHeapify(minHeap, smallest);
}
}

// A utility function to check if the given minHeap is ampty or not
int isEmpty(struct MinHeap* minHeap)
{
    return minHeap->size == 0;
}

// Standard function to extract minimum node from heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap)
{
    if (isEmpty(minHeap))
        return NULL;

    // Store the root node
    struct MinHeapNode* root = minHeap->array[0];

    // Replace root node with last node
    struct MinHeapNode* lastNode = minHeap->array[minHeap->size - 1];
    minHeap->array[0] = lastNode;

    // Update position of last node
    minHeap->pos[root->v] = minHeap->size-1;
    minHeap->pos[lastNode->v] = 0;

    // Reduce heap size and heapify root
    --minHeap->size;
    minHeapify(minHeap, 0);

    return root;
}

// Function to decrease dist value of a given vertex v. This function
// uses pos[] of min heap to get the current index of node in min heap
void decreaseKey(struct MinHeap* minHeap, int v, int dist)
{
    // Get the index of v in heap array
    int i = minHeap->pos[v];

    // Get the node and update its dist value
    minHeap->array[i]->dist = dist;

    // Travel up while the complete tree is not heified.
    // This is a O(Logn) loop
    while (i && minHeap->array[i]->dist < minHeap->array[(i - 1) / 2]->dist)
    {
        // Swap this node with its parent
        minHeap->pos[minHeap->array[i]->v] = (i-1)/2;
        minHeap->pos[minHeap->array[(i-1)/2]->v] = i;
        swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

        // move to parent index
        i = (i - 1) / 2;
    }
}

// A utility function to check if a given vertex
// 'v' is in min heap or not
bool isInMinHeap(struct MinHeap *minHeap, int v)
{
    if (minHeap->pos[v] < minHeap->size)
        return true;
    return false;
}

// A utility function used to print the solution
void printArr(int dist[], int n)
{
    printf("Vertex Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t %d\n", i, dist[i]);
}

// The main function that calculates distances of shortest paths from src to all
// vertices. It is a O(ELogV) function
void dijkstra(struct Graph* graph, int src)
{
    int V = graph->V;// Get the number of vertices in graph
    int dist[V]; // dist values used to pick minimum weight edge in cut
}

```

```

// minHeap represents set E
struct MinHeap* minHeap = createMinHeap(V);

// Initialize min heap with all vertices. dist value of all vertices
for (int v = 0; v < V; ++v)
{
    dist[v] = INT_MAX;
    minHeap->array[v] = newMinHeapNode(v, dist[v]);
    minHeap->pos[v] = v;
}

// Make dist value of src vertex as 0 so that it is extracted first
minHeap->array[src] = newMinHeapNode(src, dist[src]);
minHeap->pos[src] = src;
dist[src] = 0;
decreaseKey(minHeap, src, dist[src]);

// Initially size of min heap is equal to V
minHeap->size = V;

// In the followin loop, min heap contains all nodes
// whose shortest distance is not yet finalized.
while (!isEmpty(minHeap))
{
    // Extract the vertex with minimum distance value
    struct MinHeapNode* minHeapNode = extractMin(minHeap);
    int u = minHeapNode->v; // Store the extracted vertex number

    // Traverse through all adjacent vertices of u (the extracted
    // vertex) and update their distance values
    struct AdjListNode* pCrawl = graph->array[u].head;
    while (pCrawl != NULL)
    {
        int v = pCrawl->dest;

        // If shortest distance to v is not finalized yet, and distance to v
        // through u is less than its previously calculated distance
        if (isInMinHeap(minHeap, v) && dist[u] != INT_MAX &&
            pCrawl->weight + dist[u] < dist[v])
        {
            dist[v] = dist[u] + pCrawl->weight;

            // update distance value in min heap also
            decreaseKey(minHeap, v, dist[v]);
        }
        pCrawl = pCrawl->next;
    }

    // print the calculated shortest distances
    printArr(dist, V);
}

// Driver program to test above functions
int main()
{
    // create the graph given in above figure
    int V = 9;
    struct Graph* graph = createGraph(V);
    addEdge(graph, 0, 1, 4);
    addEdge(graph, 0, 7, 8);
    addEdge(graph, 1, 2, 8);
    addEdge(graph, 1, 7, 11);
    addEdge(graph, 2, 3, 7);
    addEdge(graph, 2, 8, 2);
    addEdge(graph, 2, 5, 4);
    addEdge(graph, 3, 4, 9);
    addEdge(graph, 3, 5, 14);
    addEdge(graph, 4, 5, 10);
    addEdge(graph, 5, 6, 2);
    addEdge(graph, 6, 7, 1);
    addEdge(graph, 6, 8, 6);
    addEdge(graph, 7, 8, 7);

    dijkstra(graph, 0);

    return 0;
}

```

Output:

Vertex	Distance from Source
0	0
1	4
2	12
3	19
4	21
5	11
6	9
7	8
8	14

Time Complexity: The time complexity of the above code/algorithm looks $O(V^2)$ as there are two nested while loops. If we take a closer look, we can observe that the statements in inner loop are executed $O(V+E)$ times (similar to BFS). The inner loop has `decreaseKey()` operation which takes $O(\log V)$ time. So overall time complexity is $O(E+V)*O(\log V)$ which is $O((E+V)*\log V) = O(E\log V)$. Note that the above code uses Binary Heap for Priority Queue implementation. Time complexity can be reduced to $O(E + V\log V)$ using Fibonacci Heap. The reason is, Fibonacci Heap takes $O(1)$ time for decrease-key operation while Binary Heap takes $O(\log n)$ time.

Notes:

- 1) The code calculates shortest distance, but doesn't calculate the path information. We can create a parent array, update the parent array when distance is updated (like prim's implementation) and use it to show the shortest path from source to different vertices.
- 2) The code is for undirected graph, same dijkstra function can be used for directed graphs also.
- 3) The code finds shortest distances from source to all vertices. If we are interested only in shortest distance from source to a single target, we can break the for loop when the picked minimum distance

vertex is equal to target (Step 3.a of algorithm).

4) Dijkstra's algorithm doesn't work for graphs with negative weight edges. For graphs with negative weight edges, [Bellman–Ford algorithm](#) can be used, we will soon be discussing it as a separate post.

[Printing Paths in Dijkstra's Shortest Path Algorithm](#)

Dijkstra's shortest path algorithm using set in STL

References:

Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

Algorithms by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Graph
Greedy
Dijkstra
shortest path

Job Sequencing Problem | Set 1 (Greedy Algorithm)

Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes single unit of time, so the minimum possible deadline for any job is 1. How to maximize total profit if only one job can be scheduled at a time.

Examples:

Input: Four Jobs with following deadlines and profits

JobID	Deadline	Profit
a	4	20
b	1	10
c	1	40
d	1	30

Output: Following is maximum profit sequence of jobs

c, a

Input: Five Jobs with following deadlines and profits

JobID	Deadline	Profit
a	2	100
b	1	19
c	2	27
d	1	25
e	3	15

Output: Following is maximum profit sequence of jobs

c, a, e

We strongly recommend to minimize your browser and try this yourself first.

A **Simple Solution** is to generate all subsets of given set of jobs and check individual subset for feasibility of jobs in that subset. Keep track of maximum profit among all feasible subsets. The time complexity of this solution is exponential.

This is a standard [Greedy Algorithm](#) problem. Following is algorithm.

- 1) Sort all jobs in decreasing order of profit.
- 2) Initialize the result sequence as first job in sorted jobs.
- 3) Do following for remaining n-1 jobs
 -a) If the current job can fit in the current result sequence
 - without missing the deadline, add current job to the result.
 - Else ignore the current job.

The Following is C++ implementation of above algorithm.

```
// Program to find the maximum profit job sequence from a given array
// of jobs with deadlines and profits
#include<iostream>
#include<algorithm>
using namespace std;

// A structure to represent a job
struct Job
{
    char id; // Job Id
    int dead; // Deadline of job
    int profit; // Profit if job is over before or on deadline
};

// This function is used for sorting all jobs according to profit
bool comparison(Job a, Job b)
{
    return (a.profit > b.profit);
}

// Returns minimum number of platforms required
void printJobScheduling(Job arr[], int n)
{
    // Sort all jobs according to decreasing order of profit
    sort(arr, arr+n, comparison);

    int result[n]; // To store result (Sequence of jobs)
    bool slot[n]; // To keep track of free time slots

    // Initialize all slots to be free
    for (int i=0; i<n; i++)
        slot[i] = false;

    // Iterate through all given jobs
    for (int i=0; i<n; i++)
    {
        // Find a free slot for this job (Note that we start
        // from the last possible slot)
        for (int j=min(n, arr[i].dead)-1; j>=0; j--)
        {
            // Free slot found
            if (slot[j] == false)
            {
                slot[j] = true;
                result[i] = j;
                break;
            }
        }
    }
}
```

```

        if (slot[i]==false)
    {
        result[i] = i; // Add this job to result
        slot[i] = true;// Make this slot occupied
        break;
    }
}

// Print the result
for (int i=0; i<n; i++)
{
    if (slot[i])
        cout << arr[result[i]].id << " ";
}

// Driver program to test methods
int main()
{
    Job arr[] = { {a', 2, 100}, {b', 1, 19}, {c', 2, 27},
                 {d', 1, 25}, {e', 3, 15} };
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Following is maximum profit sequence of jobs\n";
    printJobScheduling(arr, n);
    return 0;
}

```

Output:

```

Following is maximum profit sequence of jobs
c a e

```

Time Complexity of the above solution is $O(n^2)$. It can be optimized using Disjoint Set Data Structure. Please refer below post for details.

Job Sequencing Problem | Set 2 (Using Disjoint Set)

Sources:

http://ocw.mit.edu/courses/civil-and-environmental-engineering/1-204-computer-algorithms-in-systems-engineering-spring-2010/lecture-notes/MIT1_204S10_lec10.pdf

This article is contributed by **Shubham**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Greedy
Greedy Algorithm

Question 1

Which of the following standard algorithms is not a Greedy algorithm?

ADijkstra's shortest path algorithm

BPrim's algorithm

CKruskal algorithm

DHuffman Coding

EBellmen Ford Shortest path algorithm

Greedy Algorithms

Discuss it

Question 1 Explanation:

See <http://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/> and <http://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/>

Question 2

Suppose we run Dijkstra's single source shortest-path algorithm on the following edge weighted directed graph with vertex P as the source. In what order do the nodes get included into the set of vertices for gate2004



which the shortest path distances are finalized? (GATE CS 2004)

AP, Q, R, S, T, U

BP, Q, R, U, S, T

CP, Q, R, U, T, S

DP, Q, T, R, U, S

Greedy Algorithms

Discuss it

Question 2 Explanation:

Refer <http://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/>

Question 3

A networking company uses a compression technique to encode the message before transmitting over the network. Suppose the message contains the following characters with their frequency:

character	Frequency
a	5
b	9
c	12

d	13
e	16
f	45

Note : Each character in input message takes 1 byte. If the compression technique used is Huffman Coding, how many bits will be saved in the message?

A224

B800

C576

D324

Greedy Algorithms

Discuss it

Question 3 Explanation:

Total number of characters in the message = 100.

Each character takes 1 byte. So total number of bits needed = 800.

After *Huffman Coding*, the characters can be represented with:

f: 0

c: 100

d: 101

a: 1100

b: 1101

e: 111

Total number of bits needed = 224

Hence, number of bits saved = 800 - 224 = 576

See [here](#) for complete explanation and algorithm.

Question 4

What is the time complexity of Huffman Coding?

AO(N)

BO(NlogN)

CO(N(logN)^2)

DO(N^2)

Greedy Algorithms

Discuss it

Question 4 Explanation:

O(nlogn) where n is the number of unique characters. If there are n nodes, extractMin() is called $2*(n - 1)$ times. extractMin() takes O(logn) time as it calls minHeapify(). So, overall complexity is O(nlogn).

See [here](#) for more details of the algorithm.

Question 5

In question #2, which of the following represents the word "dead"?

A1011111100101

B0100000011010

CBoth A and B

DNone of these

Greedy Algorithms

Discuss it

Question 5 Explanation:

The *Huffman Tree* generated is:

character code-word

f 0

c 100

d 101

a 1100

b 1101

e 111

The word *dead* can be represented as: 101 111 1100 101 However, the alternative codeword can also be found by assigning 1 to the left edge and 0 to the right edge of the tree, i.e. *dead* can also be represented as: 010 000 0011 010 See [here](#) for more details of the algorithm.

Question 6

Which of the following is true about Kruskal and Prim MST algorithms? Assume that Prim is implemented for adjacency list representation using Binary Heap and Kruskal is implemented using union by rank.

AWorst case time complexity of both algorithms is same.

BWorst case time complexity of Kruskal is better than Prim

CWorst case time complexity of Prim is better than Kruskal

Greedy Algorithms

Discuss it

Question 6 Explanation:

See following [Kruskal's Minimum Spanning Tree Algorithm](#) [Prim's MST for Adjacency List Representation](#)

Question 7

Which of the following is true about Huffman Coding.

AHuffman coding may become lossy in some cases

BHuffman Codes may not be optimal lossless codes in some cases

CIIn Huffman coding, no code is prefix of any other code.

DAII of the above

Greedy Algorithms

Discuss it

Question 7 Explanation:

Huffman coding is a lossless data compression algorithm. The codes assigned to input characters are Prefix Codes, means the codes are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding.

Question 8

Suppose the letters a, b, c, d, e, f have probabilities 1/2, 1/4, 1/8, 1/16, 1/32, 1/32 respectively. Which of the following is the Huffman code for the letter a, b, c, d, e, f?

A0, 10, 110, 11110, 11110, 11111

B11, 10, 011, 010, 001, 000

C11, 10, 01, 001, 0001, 0000

D110, 100, 010, 000, 001, 111

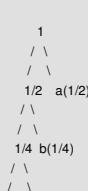
Greedy Algorithms GATE-CS-2007

Discuss it

Question 8 Explanation:

We get the following Huffman Tree after applying [Huffman Coding Algorithm](#). The idea is to keep the least probable characters as low as possible by picking them first.

The letters a, b, c, d, e, f have probabilities
1/2, 1/4, 1/8, 1/16, 1/32, 1/32 respectively.



```

1/8 c(1/8)
 / \
 / \
1/16 d(1/16)
 / \
 e f

```

Question 9

Suppose the letters a, b, c, d, e, f have probabilities 1/2, 1/4, 1/8, 1/16, 1/32, 1/32 respectively. What is the average length of Huffman codes?

- A3
B2.1875
C2.25
D1.9375

Greedy Algorithms GATE-CS-2007

Discuss it

Question 9 Explanation:

We get the following Huffman Tree after applying [Huffman Coding Algorithm](#). The idea is to keep the least probable characters as low as possible by picking them first.

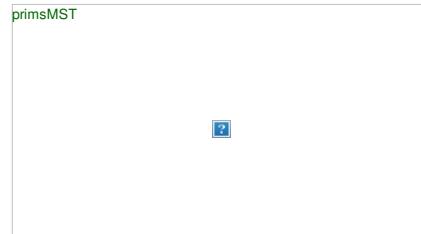
The letters a, b, c, d, e, f have probabilities
1/2, 1/4, 1/8, 1/16, 1/32, 1/32 respectively.

```

1
 / \
 / \
1/2 a(1/2)
 / \
 / \
1/4 b(1/4)
 / \
 / \
1/8 c(1/8)
 / \
 / \
1/16 d(1/16)
 / \
e f
The average length = (1*1/2 + 2*1/4 + 3*1/8 + 4*1/16 + 5*1/32 + 5*1/32)
= 1.9375

```

Question 10



Consider the undirected graph below:

Using Prim's algorithm to construct a minimum spanning tree starting with node A, which one of the following sequences of edges represents a possible order in which the edges would be added to construct the minimum spanning tree?

- A(E, G), (C, F), (F, G), (A, D), (A, B), (A, C)
B(A, D), (A, B), (A, C), (C, F), (G, E), (F, G)
C(A, B), (A, D), (D, F), (F, G), (G, E), (F, C)
D(A, D), (A, B), (D, F), (F, C), (F, G), (G, E)

Greedy Algorithms GATE-IT-2004

Discuss it

Question 10 Explanation:

A and B are **False** : The idea behind Prim's algorithm is to construct a spanning tree - **means all vertices must be connected** but here vertices are disconnected C. **False**. Prim's is a **greedy algorithm** and At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. In this option weight of AB<AD so must be picked up first. D.**TRUE**. It represents a possible order in which the edges would be added to construct the minimum spanning tree. Prim's algorithm is also a **Greedy algorithm**. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST. Read more at: <http://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-minimum-spanning-tree-mst-2/>

There are 10 questions to complete.

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.

Greedy Algorithm to find Minimum number of Coins

Given a value V, if we want to make change for V Rs, and we have infinite supply of each of the denominations in Indian currency, i.e., we have infinite supply of { 1, 2, 5, 10, 20, 50, 100, 500, 1000} valued coins/notes, what is the minimum number of coins and/or notes needed to make the change?

Examples:

Input: V = 70
Output: 2
We need a 50 Rs note and a 20 Rs note.

Input: V = 121
Output: 3
We need a 100 Rs note, a 20 Rs note and a 1 Rs coin.

We strongly recommend you to minimize your browser and try this yourself first.

The idea is simple Greedy Algorithm. Start from largest possible denomination and keep adding denominations while remaining value is greater than 0. Below is complete algorithm.

- 1) Initialize result as empty.
- 2) find the largest denomination that is smaller than V.
- 3) Add found denomination to result. Subtract value of found denomination from V.
- 4) If V becomes 0, then print result.
Else repeat steps 2 and 3 for new value of V

Below is C++ implementation of above algorithm.

```
// C++ program to find minimum number of denominations
#include <bits/stdc++.h>
using namespace std;

// All denominations of Indian Currency
int deno[] = {1, 2, 5, 10, 20, 50, 100, 500, 1000};
int n = sizeof(deno)/sizeof(deno[0]);

// Driver program
void findMin(int V)
{
    // Initialize result
    vector<int> ans;

    // Traverse through all denomination
    for (int i=n-1; i>=0; i--)
    {
        // Find denominations
        while (V >= deno[i])
        {
            V -= deno[i];
            ans.push_back(deno[i]);
        }
    }

    // Print result
    for (int i = 0; i < ans.size(); i++)
        cout << ans[i] << " ";
}

// Driver program
int main()
{
    int n = 93;
    cout << "Following is minimal number of change for " << n << " is ";
    findMin(n);
    return 0;
}
```

Output:

```
Following is minimal number of change for 93 is 50 20 20 2 1
```

Note that above approach may not work for all denominations. For example, it doesn't work for denominations {9, 6, 5, 1} and V = 11. The above approach would print 9, 1 and 1. But we can use 2 denominations 5 and 6.

For general input, we use below dynamic programming approach.

[Find minimum number of coins that make a given value](#)

Thanks to [Utkarsh](#) for providing above solution here.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: [Algorithms](#)

K Centers Problem | Set 1 (Greedy Approximate Algorithm)

Given n cities and distances between every pair of cities, select k cities to place warehouses (or ATMs or Cloud Server) such that the maximum distance of a city to a warehouse (or ATM or Cloud Server) is minimized.

For example consider the following four cities, 0, 1, 2 and 3 and distances between them, how do place 2 ATMs among these 4 cities so that the maximum distance of a city to an ATM is minimized.



There is no polynomial time solution available for this problem as the problem is a known NP-Hard problem. There is a polynomial time Greedy approximate algorithm, the greedy algorithm provides a solution which is never worse than twice the optimal solution. The greedy solution works only if the distances between cities follow [Triangular Inequality](#) (Distance between two points is always smaller than sum of distances through a third point).

The 2-Approximate Greedy Algorithm:

- 1) Choose the first center arbitrarily.
- 2) Choose remaining k-1 centers using the following criteria.
Let c1, c2, c3, ... ci be the already chosen centers. Choose (i+1)'th center by picking the city which is farthest from already selected centers, i.e, the point p which has following value as maximum
$$\text{Min}[\text{dist}(p, c1), \text{dist}(p, c2), \text{dist}(p, c3), \dots, \text{dist}(p, ci)]$$

The following diagram taken from [here](#) illustrates above algorithm.

**Example (k = 3 in the above shown Graph)**

- a) Let the first arbitrarily picked vertex be 0.
- b) The next vertex is 1 because 1 is the farthest vertex from 0.
- c) Remaining cities are 2 and 3. Calculate their distances from already selected centers (0 and 1). The greedy algorithm basically calculates following values.

Minimum of all distanced from 2 to already considered centers

$\text{Min}[\text{dist}(2, 0), \text{dist}(2, 1)] = \text{Min}[7, 8] = 7$

Minimum of all distanced from 3 to already considered centers

$\text{Min}[\text{dist}(3, 0), \text{dist}(3, 1)] = \text{Min}[6, 5] = 5$

After computing the above values, the city 2 is picked as the value corresponding to 2 is maximum.

Note that the greedy algorithm doesn't give best solution for $k = 2$ as this is just an approximate algorithm with bound as twice of optimal.

Proof that the above greedy algorithm is 2 approximate.

Let OPT be the maximum distance of a city from a center in the Optimal solution. We need to show that the maximum distance obtained from Greedy algorithm is $2 \cdot \text{OPT}$.

The proof can be done using contradiction.

- a) Assume that the distance from the furthest point to all centers is $> 2 \cdot \text{OPT}$.
- b) This means that distances between all centers are also $> 2 \cdot \text{OPT}$.
- c) We have $k + 1$ points with distances $> 2 \cdot \text{OPT}$ between every pair.
- d) Each point has a center of the optimal solution with distance $\leq \text{OPT}$ to it.
- e) There exists a pair of points with the same center X in the optimal solution (pigeonhole principle: k optimal centers, $k+1$ points)
- f) The distance between them is at most $2 \cdot \text{OPT}$ (triangle inequality) which is a contradiction.

Source:

<http://algo2.itit.kit.edu/vansteek/courses/kcenter.pdf>

This article is contributed by **Harshit**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Graph
Greedy
Greedy Algorithm

Dynamic Programming | Set 1 (Overlapping Subproblems Property)

Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again. Following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming.

In this post, we will discuss first property (Overlapping Subproblems) in detail. The second property of Dynamic programming is discussed in next post i.e. Set 2.

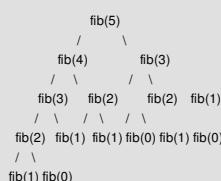
- 1) Overlapping Subproblems
- 2) Optimal Substructure

1) Overlapping Subproblems:

Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don't have to be recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again. For example, **Binary Search** doesn't have common subproblems. If we take example of following recursive program for Fibonacci Numbers, there are many subproblems which are solved again and again.

```
/* simple recursive program for Fibonacci numbers */
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

Recursion tree for execution of *fib(5)*



We can see that the function *f(3)* is being called 2 times. If we would have stored the value of *f(3)*, then instead of computing it again, we could have reused the old stored value. There are following two different ways to store the values so that these values can be reused:

- a) Memoization (Top Down)
- b) Tabulation (Bottom Up)

a) Memoization (Top Down): The memoized program for a problem is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions. We initialize a lookup array with all initial values as NIL. Whenever we need solution to a subproblem, we first look into the lookup table. If the precomputed value is there then we return that value, otherwise we calculate the value and put the result in lookup table so that it can be reused later.

Following is the memoized version for nth Fibonacci Number.

```

/* C/C++ program for Memoized version for nth Fibonacci number */
#include<stdio.h>
#define NIL -1
#define MAX 100

int lookup[MAX];

/* Function to initialize NIL values in lookup table */
void _initialize()
{
    int i;
    for (i = 0; i < MAX; i++)
        lookup[i] = NIL;
}

/* function for nth Fibonacci number */
int fib(int n)
{
    if (lookup[n] == NIL)
    {
        if (n <= 1)
            lookup[n] = n;
        else
            lookup[n] = fib(n-1) + fib(n-2);
    }

    return lookup[n];
}

int main ()
{
    int n = 40;
    _initialize();
    printf("Fibonacci number is %d ", fib(n));
    return 0;
}

```

Python

```

# Python program for Memoized version of nth Fibonacci number

# Function to calculate nth Fibonacci number
def fib(n, lookup):

    # Base case
    if n == 0 or n == 1 :
        lookup[n] = n

    # If the value is not calculated previously then calculate it
    if lookup[n] is None:
        lookup[n] = fib(n-1 ,lookup) + fib(n-2 ,lookup)

    # return the value corresponding to that value of n
    return lookup[n]
# end of function

# Driver program to test the above function
def main():
    n = 34
    # Declaration of lookup table
    # Handles till n = 100
    lookup = [None]*(101)
    print "Fibonacci Number is ",fib(n,lookup)

if __name__=="__main__":
    main()

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

b) Tabulation (Bottom Up): The tabulated program for a given problem builds a table in bottom up fashion and returns the last entry from table. For example, for the same Fibonacci number, we first calculate fib(0) then fib(1) then fib(2) then fib(3) and so on. So literally, we are building the solutions of subproblems bottom-up.

Following is the tabulated version for nth Fibonacci Number.

C/C++

```

/* C program for Tabulated version */
#include<stdio.h>
int fib(int n)
{
    int f[n+1];
    int i;
    f[0] = 0; f[1] = 1;
    for (i = 2; i <= n; i++)
        f[i] = f[i-1] + f[i-2];

    return f[n];
}

int main ()
{
    int n = 9;
    printf("Fibonacci number is %d ", fib(n));
    return 0;
}

```

Python

```
# Python program Tabulated (bottom up) version
```

```

def fib(n):

    # array declaration
    f = [0] * (n+1)

    # base case assignment
    f[1] = 1

    # calculating the fibonacci and storing the values
    for i in range(2, n+1):
        f[i] = f[i-1] + f[i-2]
    return f[n]

# Driver program to test the above function
def main():

    n = 9
    print "Fibonacci number is ", fib(n)

if __name__ == "__main__":
    main()

# This code is contributed by Nikhil Kumar Singh (nickzuck_007)

```

Output:

Fibonacci number is 34

Both Tabulated and Memoized store the solutions of subproblems. In Memoized version, table is filled on demand while in Tabulated version, starting from the first entry, all entries are filled one by one. Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version. For example, [Memoized solution](#) of the [LCS problem](#) doesn't necessarily fill all entries.

To see the optimization achieved by Memoized and Tabulated solutions over the basic Recursive solution, see the time taken by following runs for calculating 40th Fibonacci number:

[Recursive solution](#)
[Memoized solution](#)
[Tabulated solution](#)

Time taken by Recursion method is much more than the two Dynamic Programming techniques mentioned above – Memoization and Tabulation!

Also see method 2 of [Ugly Number post](#) for one more simple example where we have overlapping subproblems and we store the results of subproblems.

We will be covering Optimal Substructure Property and some more example problems in future posts on Dynamic Programming.

Try following questions as an exercise of this post.

- 1) Write a Memoized solution for LCS problem. Note that the Tabular solution is given in the CLRS book.
- 2) How would you choose between Memoization and Tabulation?

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:

<http://www.youtube.com/watch?v=V5hZoJ6uK-s>

GATE CS Corner Company Wise Coding Practice

[Dynamic Programming](#)
[Dynamic Programming](#)
[Fibonacci](#)

Dynamic Programming | Set 2 (Optimal Substructure Property)

As we discussed in [Set 1](#), following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming:

- 1) Overlapping Subproblems
- 2) Optimal Substructure

We have already discussed Overlapping Subproblem property in the [Set 1](#). Let us discuss Optimal Substructure property here.

2) Optimal Substructure: A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

For example, the Shortest Path problem has following optimal substructure property:

If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v . The standard All Pair Shortest Path algorithms like [Floyd-Warshall](#) and [Bellman-Ford](#) are typical examples of Dynamic Programming.

On the other hand, the Longest Path problem doesn't have the Optimal Substructure property. Here by Longest Path we mean longest simple path (path without cycle) between two nodes. Consider the following unweighted graph given in the [CLRS book](#). There are two longest paths from q to t : $q \rightarrow r \rightarrow t$ and $q \rightarrow s \rightarrow t$. Unlike shortest paths, these longest paths do not have the optimal substructure property. For example, the longest path $q \rightarrow r \rightarrow t$ is not a combination of longest path from q to r and longest path from r to t , because the longest path from q to r is $q \rightarrow s \rightarrow t \rightarrow r$ and the longest path from r to t is $r \rightarrow q \rightarrow s \rightarrow t$.



We will be covering some example problems in future posts on [Dynamic Programming](#).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:

http://en.wikipedia.org/wiki/Optimal_substructure
[CLRS book](#)

GATE CS Corner Company Wise Coding Practice

[Dynamic Programming](#)
[Dynamic Programming](#)

Dynamic Programming | Set 3 (Longest Increasing Subsequence)

We have discussed Overlapping Subproblems and Optimal Substructure properties in [Set 1](#) and [Set 2](#) respectively.

Let us discuss Longest Increasing Subsequence (LIS) problem as an example problem that can be solved using Dynamic Programming.

The Longest Increasing Subsequence (LIS) problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order. For example, the length of LIS for $\{10, 22, 9, 33, 21, 50, 41, 60, 80\}$ is 6 and LIS is $\{10, 22, 33, 50, 60, 80\}$.

longest-increasing-subsequence



More Examples:

```
Input : arr[] = {3, 10, 2, 1, 20}
Output : Length of LIS = 3
The longest increasing subsequence is 3, 10, 20
```

```
Input : arr[] = {3, 2}
Output : Length of LIS = 1
The longest increasing subsequences are {3} and {2}
```

```
Input : arr[] = {50, 3, 10, 7, 40, 80}
Output : Length of LIS = 4
The longest increasing subsequence is {3, 7, 40, 80}
```

We strongly recommend that you click here and practice it, before moving on to the solution.

Optimal Substructure:

Let $\text{arr}[0..n-1]$ be the input array and $L(i)$ be the length of the LIS ending at index i such that $\text{arr}[i]$ is the last element of the LIS.

Then, $L(i)$ can be recursively written as:

$L(i) = 1 + \max(L(j))$ where $0 < j < i$ and $\text{arr}[j] < \text{arr}[i]$; or

$L(i) = 1$, if no such j exists.

To find the LIS for a given array, we need to return $\max(L(i))$ where $0 < i < n$.

Thus, we see the LIS problem satisfies the optimal substructure property as the main problem can be solved using solutions to subproblems.

Following is a simple recursive implementation of the LIS problem. It follows the recursive structure discussed above.

C/C++

```
// A naive C/C++ based recursive implementation of LIS problem
#include<stdio.h>
#include<stdlib.h>

// Recursive implementation for calculating the LIS
int _lis(int arr[], int n, int *max_lis_length)
{
    // Base case
    if (n == 1)
        return 1;

    int current_lis_length = 1;
    for (int i=0; i<n-1; i++)
    {
        // Recursively calculate the length of the LIS
        // ending at arr[i]
        int subproblem_lis_length = _lis(arr, i, max_lis_length);

        // Check if appending arr[n-1] to the LIS
        // ending at arr[i] gives us an LIS ending at
        // arr[n-1] which is longer than the previously
        // calculated LIS ending at arr[n-1]
        if (arr[i] < arr[n-1] &&
            current_lis_length < (1+subproblem_lis_length))
            current_lis_length = 1+subproblem_lis_length;
    }

    // Check if currently calculated LIS ending at
    // arr[n-1] is longer than the previously calculated
    // LIS and update max_lis_length accordingly
    if (*max_lis_length < current_lis_length)
        *max_lis_length = current_lis_length;

    return current_lis_length;
}

// The wrapper function for _lis()
int lis(int arr[], int n)
{
    int max_lis_length = 1; // stores the final LIS

    // max_lis_length is passed as a reference below
    // so that it can maintain its value
    // between the recursive calls
    _lis(arr, n, &max_lis_length);

    return max_lis_length;
}

// Driver program to test the functions above
int main()
{
    int arr[] = {10, 22, 9, 33, 21, 50, 41, 60};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Length of LIS is %d\n", lis(arr, n));
    return 0;
}
```

Java

```
// A naive Java based recursive implementation of LIS problem
class LIS
{
    static int max_lis_length; // stores the final LIS
```

```

// Recursive implementation for calculating the LIS
static int _lis(int arr[], int n)
{
    // base case
    if (n == 1)
        return 1;

    int current_lis_length = 1;
    for (int i=0; i<n-1; i++)
    {
        // Recursively calculate the length of the LIS
        // ending at arr[i]
        int subproblem_lis_length = _lis(arr, i);

        // Check if appending arr[n-1] to the LIS
        // ending at arr[i] gives us an LIS ending at
        // arr[n-1] which is longer than the previously
        // calculated LIS ending at arr[n-1]
        if (arr[i] < arr[n-1] &&
            current_lis_length < (1+subproblem_lis_length))
            current_lis_length = 1+subproblem_lis_length;
    }

    // Check if currently calculated LIS ending at
    // arr[n-1] is longer than the previously calculated
    // LIS and update max_lis_length accordingly
    if (max_lis_length < current_lis_length)
        max_lis_length = current_lis_length;
}

// The wrapper function for _lis()
static int lis(int arr[], int n)
{
    max_lis_length = 1; // stores the final LIS

    // max_lis_length is declared static above
    // so that it can maintain its value
    // between the recursive calls of _lis()
    _lis( arr, n );

    return max_lis_length;
}

// Driver program to test the functions above
public static void main(String args[])
{
    int arr[] = {10, 22, 9, 33, 21, 50, 41, 60};
    int n = arr.length;
    System.out.println("Length of LIS is " + lis( arr, n ));
}
} // End of LIS class.

// This code is contributed by Rajat Mishra

```

Python

```

# A naive Python based recursive implementation of LIS problem

global max_lis_length # stores the final LIS

# Recursive implementation for calculating the LIS
def _lis(arr, n):
    # Following declaration is needed to allow modification
    # of the global copy of max_lis_length in _lis()
    global max_lis_length

    # Base Case
    if n == 1:
        return 1

    current_lis_length = 1

    for i in xrange(0, n-1):
        # Recursively calculate the length of the LIS
        # ending at arr[i]
        subproblem_lis_length = _lis(arr, i)

        # Check if appending arr[n-1] to the LIS
        # ending at arr[i] gives us an LIS ending at
        # arr[n-1] which is longer than the previously
        # calculated LIS ending at arr[n-1]
        if arr[i] < arr[n-1] and \
           current_lis_length < (1+subproblem_lis_length):
            current_lis_length = (1+subproblem_lis_length)

    # Check if currently calculated LIS ending at
    # arr[n-1] is longer than the previously calculated
    # LIS and update max_lis_length accordingly
    if (max_lis_length < current_lis_length):
        max_lis_length = current_lis_length

    return current_lis_length

# The wrapper function for _lis()
def lis(arr, n):

    # Following declaration is needed to allow modification
    # of the global copy of max_lis_length in lis()
    global max_lis_length

    max_lis_length = 1 # stores the final LIS

```

```

# max_lis_length is declared global at the top
# so that it can maintain its value
# between the recursive calls of _lis()
_lis(arr, n)

return max_lis_length

# Driver program to test the functions above
def main():
    arr = [10, 22, 9, 33, 21, 50, 41, 60]
    n = len(arr)
    print "Length of LIS is", lis(arr, n)

if __name__=="__main__":
    main()

# This code is contributed by NIKHIL KUMAR SINGH

```

Output:

Length of LIS is 5

Overlapping Subproblems:

Considering the above implementation, following is recursion tree for an array of size 4. lis(n) gives us the length of LIS for arr[].

```

    lis(4)
    /   |   \
lis(3) lis(2) lis(1)
/ \   /
lis(2) lis(1) lis(1)
/
lis(1)

```

We can see that there are many subproblems which are solved again and again. So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization or Tabulation. Following is a tabulated implementation for the LIS problem.

C/C++

```

/* Dynamic Programming C/C++ implementation of LIS problem */
#include<stdio.h>
#include<stdlib.h>

/* lis() returns the length of the longest increasing
subsequence in arr[] of size n */
int lis( int arr[], int n )
{
    int *lis, i, j, max = 0;
    lis = (int*) malloc ( sizeof( int ) * n );

    /* Initialize LIS values for all indexes */
    for ( i = 0; i < n; i++ )
        lis[i] = 1;

    /* Compute optimized LIS values in bottom up manner */
    for ( i = 1; i < n; i++ )
        for ( j = 0; j < i; j++ )
            if ( arr[i] > arr[j] && lis[i] < lis[j] + 1 )
                lis[i] = lis[j] + 1;

    /* Pick maximum of all LIS values */
    for ( i = 0; i < n; i++ )
        if ( max < lis[i] )
            max = lis[i];

    /* Free memory to avoid memory leak */
    free(lis);

    return max;
}

/* Driver program to test above function */
int main()
{
    int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Length of lis is %d\n", lis( arr, n ) );
    return 0;
}

```

Java

```

/* Dynamic Programming Java implementation of LIS problem */

class LIS
{
    /* lis() returns the length of the longest increasing
    subsequence in arr[] of size n */
    static int lis(int arr[],int n)
    {
        int lis[] = new int[n];
        int i,j,max = 0;

        /* Initialize LIS values for all indexes */
        for ( i = 0; i < n; i++ )
            lis[i] = 1;

        /* Compute optimized LIS values in bottom up manner */
        for ( i = 1; i < n; i++ )
            for ( j = 0; j < i; j++ )

```

```
        if ( arr[i] > arr[j] && lis[i] < lis[j] + 1 )
```

```
            lis[i] = lis[j] + 1;
```

```
    /* Pick maximum of all LIS values */
```

```
    for ( i = 0; i < n; i++ )
```

```
        if ( max < lis[i] )
```

```
            max = lis[i];
```

```
    return max;
```

```
}
```

```
public static void main(String args[])
```

```
{
```

```
    int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
```

```
    int n = arr.length;
```

```
    System.out.println("Length of lis is " + lis(arr) + "\n");
```

```
}
```

```
}
```

```
/* This code is contributed by Rajat Mishra */
```

Python

```
# Dynamic programming Python implementation of LIS problem
```

```
# lis returns length of the longest increasing subsequence
```

```
# in arr of size n
```

```
def lis(arr):
```

```
    n = len(arr)
```

```
    # Declare the list (array) for LIS and initialize LIS
```

```
    # values for all indexes
```

```
    lis = [1]*n
```

```
    # Compute optimized LIS values in bottom up manner
```

```
    for i in range(1, n):
```

```
        for j in range(0, i):
```

```
            if arr[i] > arr[j] and lis[i] < lis[j] + 1 :
```

```
                lis[i] = lis[j]+1
```

```
    # Initialize maximum to 0 to get the maximum of all
```

```
# LIS
```

```
maximum = 0
```

```
    # Pick maximum of all LIS values
```

```
    for i in range(n):
```

```
        maximum = max(maximum , lis[i])
```

```
    return maximum
```

```
# end of lis function
```

```
# Driver program to test above function
```

```
arr = [10, 22, 9, 33, 21, 50, 41, 60]
```

```
print "Length of lis is",lis(arr)
```

```
# This code is contributed by Nikhil Kumar Singh
```

Output:

```
Length of lis is 5
```

Note that the time complexity of the above Dynamic Programming (DP) solution is $O(n^2)$ and there is a $O(n \log n)$ solution for the LIS problem. We have not discussed the $O(n \log n)$ solution here as the purpose of this post is to explain Dynamic Programming with a simple example. See below post for $O(n \log n)$ solution.

Longest Increasing Subsequence Size ($N \log N$)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming
LIS

Dynamic Programming | Set 4 (Longest Common Subsequence)

We have discussed Overlapping Subproblems and Optimal Substructure properties in [Set 1](#) and [Set 2](#) respectively. We also discussed one example problem in [Set 3](#). Let us discuss Longest Common Subsequence (LCS) problem as one more example problem that can be solved using Dynamic Programming.

LCS Problem Statement: Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, "abc", "abg", "bd", "aeg", "acefg", .. etc are subsequences of "abcdefg". So a string of length n has 2^n different possible subsequences.

It is a classic computer science problem, the basis of `diff` (a file comparison program that outputs the differences between two files), and has applications in bioinformatics.

Examples:

LCS for input Sequences "ABCDGH" and "AEDFHR" is "ADH" of length 3.

LCS for input Sequences "AGGTAB" and "GXTXAYB" is "GTAB" of length 4.

We strongly recommend that you click here and practice it, before moving on to the solution.

The naive solution for this problem is to generate all subsequences of both given sequences and find the longest matching subsequence. This solution is exponential in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem.

1) Optimal Substructure:

Let the input sequences be $X[0..m-1]$ and $Y[0..n-1]$ of lengths m and n respectively. And let $L(X[0..m-1], Y[0..n-1])$ be the length of LCS of the two sequences X and Y . Following is the recursive definition of $L(X[0..m-1], Y[0..n-1])$.

If last characters of both sequences match (or $X[m-1] == Y[n-1]$) then

$$L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])$$

If last characters of both sequences do not match (or $X[m-1] != Y[n-1]$) then

$$L(X[0..m-1], Y[0..n-1]) = \text{MAX} (L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2]))$$

Examples:

1) Consider the input strings "AGGTAB" and "GXTXAYB". Last characters match for the strings. So length of LCS can be written as:

$$L("AGGTAB", "GXTXAYB") = 1 + L("AGGTA", "GXTXAY")$$

longest-common-subsequence



2) Consider the input strings "ABCDGH" and "AEDFHR". Last characters do not match for the strings. So length of LCS can be written as:

$$L("ABCDGH", "AEDFHR") = \text{MAX} (L("ABCDG", "AEDFHR"), L("ABCDGH", "AEDFH"))$$

So the LCS problem has optimal substructure property as the main problem can be solved using solutions to subproblems.

2) Overlapping Subproblems:

Following is simple recursive implementation of the LCS problem. The implementation simply follows the recursive structure mentioned above.

C/C++

```
/* A Naive recursive implementation of LCS problem */
#include<bits/stdc++.h>

int max(int a, int b);

/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n )
{
    if (m == 0 || n == 0)
        return 0;
    if (X[m-1] == Y[n-1])
        return 1 + lcs(X, Y, m-1, n-1);
    else
        return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
}

/* Utility function to get max of 2 integers */
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Driver program to test above function */
int main()
{
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";

    int m = strlen(X);
    int n = strlen(Y);

    printf("Length of LCS is %d\n", lcs( X, Y, m, n ) );

    return 0;
}
```

Python

```
# A Naive recursive Python implementation of LCS problem

def lcs(X, Y, m, n):

    if m == 0 or n == 0:
        return 0;
    elif X[m-1] == Y[n-1]:
        return 1 + lcs(X, Y, m-1, n-1);
    else:
        return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));

# Driver program to test the above function
```

```
X = "AGGTAB"
Y = "GXTXAYB"
print("Length of LCS is ", lcs(X , Y, len(X), len(Y))
```

Output:

```
Length of LCS is 4
```

Time complexity of the above naive recursive approach is $O(2^n)$ in worst case and worst case happens when all characters of X and Y mismatch i.e., length of LCS is 0. Considering the above implementation, following is a partial recursion tree for input strings "AXYT" and "AYZX"

```
lcs("AXYT", "AYZX")
  /   \
lcs("AXY", "AYZ")    lcs("AXYT", "AYZ")
 / \ / \
lcs("AX", "AYZ") lcs("AXY", "AYZ") lcs("AXY", "AYZ") lcs("AXYT", "AY")
```

In the above partial recursion tree, $\text{lcs}(\text{AXY}, \text{AYZ})$ is being solved twice. If we draw the complete recursion tree, then we can see that there are many subproblems which are solved again and again. So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization or Tabulation. Following is a tabulated implementation for the LCS problem.

C/C++

```
/* Dynamic Programming C/C++ implementation of LCS problem */
#include<bits/stdc++.h>

int max(int a, int b);

/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n )
{
    int L[m+1][n+1];
    int i, j;

    /* Following steps build L[m+1][n+1] in bottom up fashion. Note
       that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1] */
    for (i=0; i<=m; i++)
    {
        for (j=0; j<=n; j++)
        {
            if (i == 0 || j == 0)
                L[i][j] = 0;
            else if (X[i-1] == Y[j-1])
                L[i][j] = L[i-1][j-1] + 1;
            else
                L[i][j] = max(L[i-1][j], L[i][j-1]);
        }
    }

    /* L[m][n] contains length of LCS for X[0..n-1] and Y[0..m-1] */
    return L[m][n];
}

/* Utility function to get max of 2 integers */
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Driver program to test above function */
int main()
{
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";

    int m = strlen(X);
    int n = strlen(Y);

    printf("Length of LCS is %d\n", lcs( X, Y, m, n ) );

    return 0;
}
```

Python

```
# Dynamic Programming implementation of LCS problem

def lcs(X , Y):
    # find the length of the strings
    m = len(X)
    n = len(Y)

    # declaring the array for storing the dp values
    L = [[None]*(n+1) for i in xrange(m+1)]

    """Following steps build L[m+1][n+1] in bottom up fashion
    Note: L[i][j] contains length of LCS of X[0..i-1]
    and Y[0..j-1]"""
    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0 :
                L[i][j] = 0
            elif X[i-1] == Y[j-1]:
                L[i][j] = L[i-1][j-1]+1
            else:
                L[i][j] = max(L[i-1][j] , L[i][j-1])

    # L[m][n] contains the length of LCS of X[0..n-1] & Y[0..m-1]
```

```

return L[m][n]
#endif function lcs

# Driver program to test the above function
X = "AGGTAB"
Y = "GXTXAYB"
print "Length of LCS is ", lcs(X, Y)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Time Complexity of the above implementation is $O(mn)$ which is much better than the worst case time complexity of Naive Recursive implementation.

The above algorithm/code returns only length of LCS. Please see the following post for printing the LCS.

Printing Longest Common Subsequence

You can also check the space optimized version of LCS at

[Space Optimized Solution of LCS](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:

<http://www.youtube.com/watch?v=V5hZoJ6uK-s>
http://www.algorithmist.com/index.php/Longest_Common_Subsequence
<http://www.ics.uci.edu/~eppstein/161/960229.html>
http://en.wikipedia.org/wiki/Longest_common_subsequence_problem

GATE CS Corner Company Wise Coding Practice

[Dynamic Programming](#)
[Dynamic Programming](#)
[LCS](#)
[subsequence](#)

Dynamic Programming | Set 5 (Edit Distance)

Given two strings str1 and str2 and below operations that can be performed on str1. Find minimum number of edits (operations) required to convert 'str1' into 'str2'.

- a. Insert
- b. Remove
- c. Replace

All of the above operations are of equal cost.

Examples:

Input: str1 = "geek", str2 = "gesek"
Output: 1
We can convert str1 into str2 by inserting a 's'.

Input: str1 = "cat", str2 = "cut"
Output: 1
We can convert str1 into str2 by replacing 'a' with 'u'.

Input: str1 = "sunday", str2 = "saturday"
Output: 3
Last three and first characters are same. We basically need to convert "un" to "atur". This can be done using below three operations.
Replace 'n' with 'r', insert t, insert a

What are the subproblems in this case?

The idea is process all characters one by one starting from either from left or right sides of both strings.

Let us traverse from right corner, there are two possibilities for every pair of character being traversed.

m: Length of str1 (first string)
n: Length of str2 (second string)

1. If last characters of two strings are same, nothing much to do. Ignore last characters and get count for remaining strings. So we recur for lengths m-1 and n-1.
2. Else (If last characters are not same), we consider all operations on 'str1', consider all three operations on last character of first string, recursively compute minimum cost for all three operations and take minimum of three values.
 - a. Insert: Recur for m and n-1
 - b. Remove: Recur for m-1 and n
 - c. Replace: Recur for m-1 and n-1

Below is C++ implementation of above Naive recursive solution.

C++

```
// A Naive recursive C++ program to find minimum number
// operations to convert str1 to str2
#include<bits/stdc++.h>
using namespace std;

// Utility function to find minimum of three numbers
int min(int x, int y, int z)
{
    return min(min(x, y), z);
}

int editDist(string str1 , string str2 , int m ,int n)
{
    // If first string is empty, the only option is to
    // insert all characters of second string into first
    if (m == 0) return n;

    // If second string is empty, the only option is to
    // remove all characters of first string
    if (n == 0) return m;

    // If last characters of two strings are same, nothing
    // much to do. Ignore last characters and get count for
    // remaining strings.
    if (str1[m-1] == str2[n-1])
        return editDist(str1, str2, m-1, n-1);

    // If last characters are not same, consider all three
    // operations on last character of first string, recursively
    // compute minimum cost for all three operations and take
    // minimum of three values.
    return 1 + min ( editDist(str1, str2, m, n-1), // Insert
                    editDist(str1, str2, m-1, n), // Remove
                    editDist(str1, str2, m-1, n-1) // Replace
                  );
}

// Driver program
int main()
{
    // your code goes here
    string str1 = "sunday";
    string str2 = "saturday";

    cout << editDist( str1 , str2 , str1.length(), str2.length());

    return 0;
}
```

Java

```
// A Naive recursive Java program to find minimum number
// operations to convert str1 to str2
class EDIST
{
    static int min(int x,int y,int z)
    {
        if (x<y && x<z) return x;
        if (y<x && y<z) return y;
        else return z;
    }

    static int editDist(String str1 , String str2 , int m ,int n)
    {
        // If first string is empty, the only option is to
        // insert all characters of second string into first
        if (m == 0) return n;

        // If second string is empty, the only option is to
        // remove all characters of first string
        if (n == 0) return m;

        // If last characters of two strings are same, nothing
        // much to do. Ignore last characters and get count for
        // remaining strings.
        if (str1.charAt(m-1) == str2.charAt(n-1))
            return editDist(str1, str2, m-1, n-1);

        // If last characters are not same, consider all three
        // operations on last character of first string, recursively
        // compute minimum cost for all three operations and take
        // minimum of three values.
        return 1 + min ( editDist(str1, str2, m, n-1), // Insert
                        editDist(str1, str2, m-1, n), // Remove
                        editDist(str1, str2, m-1, n-1) // Replace
                      );
    }

    public static void main(String args[])
    {
        String str1 = "sunday";
        String str2 = "saturday";

        System.out.println( editDist( str1 , str2 , str1.length(), str2.length() ) );
    }
}
/*This code is contributed by Rajat Mishra*/
```

Python

```

# A Naive recursive Python program to fin minimum number
# operations to convert str1 to str2
def editDistance(str1, str2, m ,n):

    # If first string is empty, the only option is to
    # insert all characters of second string into first
    if m==0:
        return n

    # If second string is empty, the only option is to
    # remove all characters of first string
    if n==0:
        return m

    # If last characters of two strings are same, nothing
    # much to do. Ignore last characters and get count for
    # remaining strings.
    if str1[m-1]==str2[n-1]:
        return editDistance(str1,str2,m-1,n-1)

    # If last characters are not same, consider all three
    # operations on last character of first string, recursively
    # compute minimum cost for all three operations and take
    # minimum of three values.
    return 1 + min(editDistance(str1, str2, m, n-1), # Insert
                   editDistance(str1, str2, m-1, n), # Remove
                   editDistance(str1, str2, m-1, n-1) # Replace
                 )

# Driver program to test the above function
str1 = "sunday"
str2 = "saturday"
print(editDistance(str1, str2,len(str1), len(str2)))

# This code is contributed by Bhavya Jain

```

Output:

3

The time complexity of above solution is exponential. In worst case, we may end up doing $O(3^m)$ operations. The worst case happens when none of characters of two strings match. Below is a recursive call diagram for worst case.

EditDistance



We can see that many subproblems are solved again and again, for example eD(2,2) is called three times. Since same subproblems are called again, this problem has Overlapping Subproblems property. So Edit Distance problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array that stores results of subproblems.

C++

```

// A Dynamic Programming based C++ program to find minimum
// number operations to convert str1 to str2
#include<bits/stdc++.h>
using namespace std;

// Utility function to find minimum of three numbers
int min(int x, int y, int z)
{
    return min(min(x, y), z);
}

int editDistDP(string str1, string str2, int m, int n)
{
    // Create a table to store results of subproblems
    int dp[m+1][n+1];

    // Fill d[][] in bottom up manner
    for (int i=0; i<=m; i++)
    {
        for (int j=0; j<=n; j++)
        {
            // If first string is empty, only option is to
            // insert all characters of second string
            if (i==0)
                dp[i][j] = j; // Min. operations = j

            // If second string is empty, only option is to
            // remove all characters of second string
            else if (j==0)
                dp[i][j] = i; // Min. operations = i

            // If last characters are same, ignore last char
            // and recur for remaining string
            else if (str1[i-1] == str2[j-1])
                dp[i][j] = dp[i-1][j-1];

            // If last characters are different, consider all
            // three operations on last character of first string,
            // recursively compute minimum cost for all three
            // operations and take minimum of three values.
            else
                dp[i][j] = 1 + min(dp[i-1][j], // Insert
                                   dp[i][j-1], // Remove
                                   dp[i-1][j-1]); // Replace
        }
    }

    return dp[m][n];
}

```

```

    // If last character are different, consider all
    // possibilities and find minimum
    else
        dp[i][j] = 1 + min(dp[i][j-1], // Insert
                            dp[i-1][j]), // Remove
                            dp[i-1][j-1]); // Replace
    }
}

return dp[m][n];
}

// Driver program
int main()
{
    // your code goes here
    string str1 = "sunday";
    string str2 = "saturday";

    cout << editDistDP(str1, str2, str1.length(), str2.length());

    return 0;
}

```

Java

```

// A Dynamic Programming based Java program to find minimum
// number operations to convert str1 to str2
class EDIST
{
    static int min(int x,int y,int z)
    {
        if (x < y && x < z) return x;
        if (y < x && y < z) return y;
        else return z;
    }

    static int editDistDP(String str1, String str2, int m, int n)
    {
        // Create a table to store results of subproblems
        int dp[][] = new int[m+1][n+1];

        // Fill d[][] in bottom up manner
        for (int i=0; i<=m; i++)
        {
            for (int j=0; j<=n; j++)
            {
                // If first string is empty, only option is to
                // insert all characters of second string
                if (i==0)
                    dp[i][j] = j; // Min. operations = j

                // If second string is empty, only option is to
                // remove all characters of second string
                else if (j==0)
                    dp[i][j] = i; // Min. operations = i

                // If last characters are same, ignore last char
                // and recur for remaining string
                else if (str1.charAt(i-1) == str2.charAt(j-1))
                    dp[i][j] = dp[i-1][j-1];

                // If last character are different, consider all
                // possibilities and find minimum
                else
                    dp[i][j] = 1 + min(dp[i][j-1], // Insert
                                      dp[i-1][j]), // Remove
                                      dp[i-1][j-1]); // Replace
            }
        }

        return dp[m][n];
    }
}

```

```

public static void main(String args[])
{
    String str1 = "sunday";
    String str2 = "saturday";
    System.out.println( editDistDP( str1 , str2 , str1.length(), str2.length()) );
}

/*This code is contributed by Rajat Mishra*/

```

Python

```

# A Dynamic Programming based Python program for edit
# distance problem
def editDistDP(str1, str2, m, n):
    # Create a table to store results of subproblems
    dp = [[0 for x in range(n+1)] for x in range(m+1)]

    # Fill d[][] in bottom up manner
    for i in range(m+1):
        for j in range(n+1):

            # If first string is empty, only option is to
            # insert all characters of second string
            if i == 0:
                dp[i][j] = j # Min. operations = j

            # If second string is empty, only option is to

```

```

# remove all characters of second string
elf1 == 0:
    dp[i][j] = i # Min. operations = i

# If last characters are same, ignore last char
# and recur for remaining string
elif str1[i-1] == str2[j-1]:
    dp[i][j] = dp[i-1][j-1]

# If last character are different, consider all
# possibilities and find minimum
else:
    dp[i][j] = 1 + min(dp[i][j-1], # Insert
                        dp[i-1][j], # Remove
                        dp[i-1][j-1]) # Replace

return dp[m][n]

# Driver program
str1 = "sunday"
str2 = "saturday"

print(editDistDP(str1, str2, len(str1), len(str2)))
# This code is contributed by Bhavya Jain

```

Output:

3

Applications: There are many practical applications of edit distance algorithm, refer [Lucene API](#) for sample. Another example, display all the words in a dictionary that are near proximity to a given

REFERENCES AND NOTES

GATE CS Corner | Company Wise Coding Practice

Dynamic Programming
About Venki
Software Engineer
www.allaboutvenki.com

Dynamic Programming | Set 6 (Min Cost Path)

Given a cost matrix $\text{cost}[\cdot][\cdot]$ and a position (m, n) in $\text{cost}[\cdot][\cdot]$, write a function that returns cost of minimum cost path to reach (m, n) from $(0, 0)$. Each cell of the matrix represents a cost to traverse through that cell. Total cost of a path to reach (m, n) is sum of all the costs on that path (including both source and destination). You can only traverse down, right and diagonally lower cells from a given cell, i.e., from a given cell (i, j) , cells $(i+1, j)$, $(i, j+1)$ and $(i+1, j+1)$ can be traversed. You may assume that all costs are positive integers.

For example, in the following figure, what is the minimum cost path to (2, 2)?

The path with minimum cost is highlighted in the following figure. The path is $(0, 0) \rightarrow (0, 1) \rightarrow (1, 2) \rightarrow (2, 2)$. The cost of the path is 8 ($1 + 2 + 2 + 3$).

1) Optimal Substructure

The path to reach (m, n) must be through one of the 3 cells: $(m-1, n-1)$ or $(m-1, n)$ or $(m, n-1)$. So minimum cost to reach (m, n) can be written as "minimum of the 3 cells plus $\text{cost}[m][n]$ ".

$$\text{minCost}(m, n) = \min(\text{minCost}(m-1, n-1), \text{minCost}(m-1, n), \text{minCost}(m, n-1)) + \text{cost}[m][n]$$

2) Overlapping Subproblems

Following is simple recursive implementation of the MCP (Minimum Cost Path) problem. The implementation simply follows the recursive structure mentioned above.

```

/* A utility function that returns minimum of 3 integers */
int min(int x, int y, int z)
{
    if (x < y)
        return (x < z)? x : z;
    else
        return (y < z)? y : z;
}

/* Driver program to test above functions */
int main()
{
    int cost[R][C] = { {1, 2, 3},
                      {4, 8, 2},
                      {1, 5, 3} };
    printf("%d ", minCost(cost, 2, 2));
    return 0;
}

```

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree, there are many nodes which appear more than once. Time complexity of this naive recursive solution is exponential and it is terribly slow.

```

mC refers to minCost()
      mC(2, 2)
      /   |   \
      /   |   \
  mC(1, 1)   mC(1, 2)   mC(2, 1)
  / | \ / | \ / | \
 / | \ / | \ / | \
mC(0,0) mC(0,1) mC(1,0) mC(0,1) mC(0,2) mC(1,1) mC(1,0) mC(1,1) mC(2,0)

```

So the MCP problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array `tc[][]` in bottom up manner.

C++

```

/* Dynamic Programming implementation of MCP problem */
#include<stdio.h>
#include<limits.h>
#define R 3
#define C 3

int min(int x, int y, int z);

int minCost(int cost[R][C], int m, int n)
{
    int i, j;

    // Instead of following line, we can use int tc[m+1][n+1]
    // dynamically allocate memory to save space. The following line is
    // used to keep the program simple and make it working on all compilers.
    int tc[R][C];

    tc[0][0] = cost[0][0];

    /* Initialize first column of total cost(tc) array */
    for (i = 1; i <= m; i++)
        tc[i][0] = tc[i-1][0] + cost[i][0];

    /* Initialize first row of tc array */
    for (j = 1; j <= n; j++)
        tc[0][j] = tc[0][j-1] + cost[0][j];

    /* Construct rest of the tc array */
    for (i = 1; i <= m; i++)
        for (j = 1; j <= n; j++)
            tc[i][j] = min(tc[i-1][j-1],
                           tc[i-1][j],
                           tc[i][j-1]) + cost[i][j];

    return tc[m][n];
}

/* A utility function that returns minimum of 3 integers */
int min(int x, int y, int z)
{
    if (x < y)
        return (x < z)? x : z;
    else
        return (y < z)? y : z;
}

/* Driver program to test above functions */
int main()
{
    int cost[R][C] = { {1, 2, 3},
                      {4, 8, 2},
                      {1, 5, 3} };
    printf("%d ", minCost(cost, 2, 2));
    return 0;
}

```

Java

```

/* Java program for Dynamic Programming implementation
of Min Cost Path problem */
import java.util.*;

class MinimumCostPath
{
    /* A utility function that returns minimum of 3 integers */

```

```

private static int min(int x, int y, int z)
{
    if (x < y)
        return (x < z) ? x : z;
    else
        return (y < z) ? y : z;
}

private static int minCost(int cost[][], int m, int n)
{
    int i, j;
    int tc[][]=new int[m+1][n+1];

    tc[0][0] = cost[0][0];

    /* Initialize first column of total cost(tc) array */
    for (i = 1; i <= m; i++)
        tc[i][0] = tc[i-1][0] + cost[i][0];

    /* Initialize first row of tc array */
    for (j = 1; j <= n; j++)
        tc[0][j] = tc[0][j-1] + cost[0][j];

    /* Construct rest of the tc array */
    for (i = 1; i <= m; i++)
        for (j = 1; j <= n; j++)
            tc[i][j] = min(tc[i-1][j]-1,
                           tc[i-1][j],
                           tc[i][j-1] + cost[i][j]);

    return tc[m][n];
}

/* Driver program to test above functions */
public static void main(String args[])
{
    int cost[][]={{1, 2, 3},
                  {4, 8, 2},
                  {1, 5, 3}};
    System.out.println("minimum cost to reach (2,2) = " +
                       minCost(cost,2,2));
}
}

// This code is contributed by Pankaj Kumar

```

Python

```

# Dynamic Programming Python implementation of Min Cost Path
# problem
R = 3
C = 3

def minCost(cost, m, n):

    # Instead of following line, we can use int tc[m+1][n+1] or
    # dynamically allocate memory to save space. The following
    # line is used to keep the program simple and make it working
    # on all compilers.
    tc = [[0 for x in range(C)] for x in range(R)]

    tc[0][0] = cost[0][0]

    # Initialize first column of total cost(tc) array
    for i in range(1, m+1):
        tc[i][0] = tc[i-1][0] + cost[i][0]

    # Initialize first row of tc array
    for j in range(1, n+1):
        tc[0][j] = tc[0][j-1] + cost[0][j]

    # Construct rest of the tc array
    for i in range(1, m+1):
        for j in range(1, n+1):
            tc[i][j] = min(tc[i-1][j]-1, tc[i-1][j], tc[i][j-1]) + cost[i][j]

    return tc[m][n]

# Driver program to test above functions
cost = [[1, 2, 3],
        [4, 8, 2],
        [1, 5, 3]]
print(minCost(cost, 2, 2))

# This code is contributed by Bhavya Jain

```

Output:

8

Time Complexity of the DP implementation is O(mn) which is much better than Naive Recursive implementation.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Matrix
Dynamic Programming
Matrix

Given a value N, if we want to make change for N cents, and we have infinite supply of each of S = { S1, S2, .. , Sm} valued coins, how many ways can we make the change? The order of coins doesn't matter.

For example, for N = 4 and S = {1,2,3}, there are four solutions: {1,1,1,1},{1,1,2},{2,2},{1,3}. So output should be 4. For N = 10 and S = {2, 5, 3, 6}, there are five solutions: {2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}. So the output should be 5.

1) Optimal Substructure

To count total number solutions, we can divide all set solutions in two sets.

1) Solutions that do not contain mth coin (or Sm).

2) Solutions that contain at least one Sm.

Let count(S[], m, n) be the function to count the number of solutions, then it can be written as sum of count(S[], m-1, n) and count(S[], m, n-Sm).

Therefore, the problem has optimal substructure property as the problem can be solved using solutions to subproblems.

2) Overlapping Subproblems

Following is a simple recursive implementation of the Coin Change problem. The implementation simply follows the recursive structure mentioned above.

```
#include<stdio.h>

// Returns the count of ways we can sum S[0...m-1] coins to get sum n
int count( int S[], int m, int n )
{
    // If n is 0 then there is 1 solution (do not include any coin)
    if (n == 0)
        return 1;

    // If n is less than 0 then no solution exists
    if (n < 0)
        return 0;

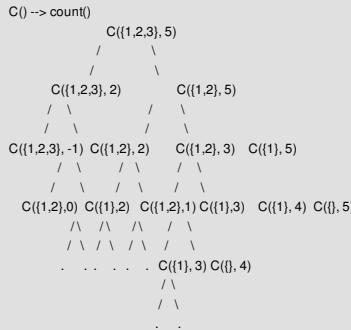
    // If there are no coins and n is greater than 0, then no solution exist
    if (m <=0 && n >= 1)
        return 0;

    // count is sum of solutions (i) including S[m-1] (ii) excluding S[m-1]
    return count( S, m - 1, n ) + count( S, m, n-S[m-1] );
}

// Driver program to test above function
int main()
{
    int i, j;
    int arr[] = {1, 2, 3};
    int m = sizeof(arr)/sizeof(arr[0]);
    printf("%d ", count(arr, m, 4));
    getchar();
    return 0;
}
```

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for S = {1, 2, 3} and n = 5.

The function C({1}, 3) is called two times. If we draw the complete tree, then we can see that there are many subproblems being called more than once.



Since same subproblems are called again, this problem has Overlapping Subproblems property. So the Coin Change problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem. Like other typical [Dynamic Programming\(DP\) problems](#), recomputations of same subproblems can be avoided by constructing a temporary array table[][] in bottom up manner.

Dynamic Programming Solution

C

```
#include<stdio.h>

int count( int S[], int m, int n )
{
    int i, j, x, y;

    // We need n+1 rows as the table is constructed in bottom up manner using
    // the base case 0 value case (n = 0)
    int table[n+1][m];

    // Fill the entries for 0 value case (n = 0)
    for (i=0; i<m; i++)
        table[0][i] = 1;

    // Fill rest of the table entries in bottom up manner
    for (i = 1; i < n+1; i++)
    {
        for (j = 0; j < m; j++)
        {
            // Count of solutions including S[i]
            x = (i-S[j] >= 0)? table[i - S[j]][j]: 0;

            // Count of solutions excluding S[i]
            y = (j >= 1)? table[i][j-1]: 0;

            // total count
            table[i][j] = x + y;
        }
    }
}
```

```

        table[i][j] = x + y;
    }
}
return table[n][m-1];
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 2, 3};
    int m = sizeof(arr)/sizeof(arr[0]);
    int n = 4;
    printf(" %d ", count(arr, m, n));
    return 0;
}

```

Java

```

/* Dynamic Programming Java implementation of Coin
   Change problem */
import java.util.Arrays;

class CoinChange
{
    static long countWays(int S[], int m, int n)
    {
        //Time complexity of this function: O(mn)
        //Space Complexity of this function: O(n)

        // table[i] will be storing the number of solutions
        // for value i. We need n+1 rows as the table is
        // constructed in bottom up manner using the base
        // case (n = 0)
        long[] table = new long[n+1];

        // Initialize all table values as 0
        Arrays.fill(table, 0); //O(n)

        // Base case (If given value is 0)
        table[0] = 1;

        // Pick all coins one by one and update the table[]
        // values after the index greater than or equal to
        // the value of the picked coin
        for (int i=0; i<m; i++)
            for (int j=S[i]; j<=n; j++)
                table[j] += table[j-S[i]];

        return table[n];
    }

    // Driver Function to test above function
    public static void main(String args[])
    {
        int arr[] = {1, 2, 3};
        int m = arr.length;
        int n = 4;
        System.out.println(countWays(arr, m, n));
    }
}
// This code is contributed by Pankaj Kumar

```

Python

```

# Dynamic Programming Python implementation of Coin Change problem
def count(S, m, n):
    # We need n+1 rows as the table is constructed in bottom up
    # manner using the base case 0 value case (n = 0)
    table = [[0 for x in range(m)] for x in range(n+1)]

    # Fill the entries for 0 value case (n = 0)
    for i in range(m):
        table[0][i] = 1

    # Fill rest of the table entries in bottom up manner
    for i in range(1, n+1):
        for j in range(m):
            # Count of solutions including S[i]
            x = table[i - S[j]][j] if i-S[j] >= 0 else 0

            # Count of solutions excluding S[i]
            y = table[i][j-1] if j >= 1 else 0

            # total count
            table[i][j] = x + y

    return table[n][m-1]

# Driver program to test above function
arr = [1, 2, 3]
m = len(arr)
n = 4
print(count(arr, m, n))

# This code is contributed by Bhavya Jain

```

Output:

4

Time Complexity: O(mn)

Following is a simplified version of method 2. The auxiliary space required here is O(n) only.

C

```
int count( int S[], int m, int n )  
{  
    // table[i] will be storing the number of solutions for  
    // value i. We need n+1 rows as the table is constructed  
    // in bottom up manner using the base case (n = 0)  
    int table[n+1];  
  
    // Initialize all table values as 0  
    memset(table, 0, sizeof(table));  
  
    // Base case (If given value is 0)  
    table[0] = 1;  
  
    // Pick all coins one by one and update the table[] values  
    // after the index greater than or equal to the value of the  
    // picked coin  
    for(int i=0; i<m; i++)  
        for(int j=S[i]; j<=n; j++)  
            table[j] += table[j-S[i]];  
  
    return table[n];  
}
```

Python

```
# Dynamic Programming Python implementation of Coin  
# Change problem  
def count(S, m, n):  
  
    # table[i] will be storing the number of solutions for  
    # value i. We need n+1 rows as the table is constructed  
    # in bottom up manner using the base case (n = 0)  
    # Initialize all table values as 0  
    table = [0 for k in range(n+1)]  
  
    # Base case (If given value is 0)  
    table[0] = 1  
  
    # Pick all coins one by one and update the table[] values  
    # after the index greater than or equal to the value of the  
    # picked coin  
    for i in range(m):  
        for j in range(S[i], n+1):  
            table[j] += table[j-S[i]]  
  
    return table[n]  
  
# Driver program to test above function  
arr = [1, 2, 3]  
m = len(arr)  
n = 4  
x = count(arr, m, n)  
print(x)  
  
# This code is contributed by Afzal Ansari
```

Thanks to Rohan Laishram for suggesting this space optimized version.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:

http://www.algorithmist.com/index.php/Coin_Change

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming

Dynamic Programming | Set 8 (Matrix Chain Multiplication)

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:

$$(ABC)D = (AB)(CD) = A(BCD) = \dots$$

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a 10×30 matrix, B is a 30×5 matrix, and C is a 5×60 matrix. Then,

$$\begin{aligned}(AB)C &= (10 \times 30 \times 5) + (10 \times 5 \times 60) = 1500 + 3000 = 4500 \text{ operations} \\ A(BC) &= (30 \times 5 \times 60) + (10 \times 30 \times 60) = 9000 + 18000 = 27000 \text{ operations.}\end{aligned}$$

Clearly the first parenthesization requires less number of operations.

Given an array p[] which represents the chain of matrices such that the ith matrix A_i is of dimension $p[i-1] \times p[i]$. We need to write a function MatrixChainOrder() that should return the minimum number of multiplications needed to multiply the chain.

Input: p[] = {40, 20, 30, 10, 30}
Output: 26000

There are 4 matrices of dimensions 40×20 , 20×30 , 30×10 and 10×30 .
Let the input 4 matrices be A, B, C and D. The minimum number of multiplications are obtained by putting parenthesis in following way

```
(A(BC))D -> 20*30*10 + 40*20*10 + 40*10*30
```

Input: p[] = {10, 20, 30, 40, 30}

Output: 30000

There are 4 matrices of dimensions 10x20, 20x30, 30x40 and 40x30.
Let the input 4 matrices be A, B, C and D. The minimum number of
multiplications are obtained by putting parenthesis in following way
(AB)CD -> 10*20*30 + 10*30*40 + 10*40*30

Input: p[] = {10, 20, 30}

Output: 6000

There are only two matrices of dimensions 10x20 and 20x30. So there
is only one way to multiply the matrices, cost of which is 10*20*30

1) Optimal Substructure:

A simple solution is to place parenthesis at all possible places, calculate the cost for each placement and return the minimum value. In a chain of matrices of size n, we can place the first set of parenthesis in n-1 ways. For example, if the given chain is of 4 matrices. let the chain be ABCD, then there are 3 ways to place first set of parenthesis outer side: (A)(BCD), (AB)(CD) and (ABC)(D). So when we place set of parenthesis, we divide the problem into subproblems of smaller size. Therefore, the problem has optimal substructure property and can be easily solved using recursion.

Minimum number of multiplication needed to multiply a chain of size n = Minimum of all n-1 placements (these placements create subproblems of smaller size)

2) Overlapping Subproblems

Following is a recursive implementation that simply follows the above optimal substructure property.

C

```
/* A naive recursive implementation that simply
follows the above optimal substructure property */
#include<stdio.h>
#include<limits.h>

// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
int MatrixChainOrder(int p[], int i, int j)
{
    if(i == j)
        return 0;
    int k;
    int min = INT_MAX;
    int count;

    // place parenthesis at different places between first
    // and last matrix, recursively calculate count of
    // multiplications for each parenthesis placement and
    // return the minimum count
    for (k = i; k <j; k++)
    {
        count = MatrixChainOrder(p, i, k) +
            MatrixChainOrder(p, k+1, j) +
            p[i-1]*p[k]*p[j];

        if (count < min)
            min = count;
    }

    // Return minimum count
    return min;
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 2, 3, 4, 3};
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("Minimum number of multiplications is %d ",
           MatrixChainOrder(arr, 1, n-1));

    getchar();
    return 0;
}
```

Java

```
/* A naive recursive implementation that simply follows
the above optimal substructure property */
class MatrixChainMultiplication
{
    // Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
    static int MatrixChainOrder(int p[], int i, int j)
    {
        if (i == j)
            return 0;

        int min = Integer.MAX_VALUE;

        // place parenthesis at different places between first
        // and last matrix, recursively calculate count of
        // multiplications for each parenthesis placement and
        // return the minimum count
        for (int k=i; k <j; k++)
        {
            int count = MatrixChainOrder(p, i, k) +
                MatrixChainOrder(p, k+1, j) +
                p[i-1]*p[k]*p[j];

            if (count < min)
                min = count;
        }

        // Return minimum count
        return min;
    }
}
```

```

// Driver program to test above function
public static void main(String args[])
{
    int arr[] = new int[] { 1, 2, 3, 4, 3 };
    int n = arr.length;

    System.out.println("Minimum number of multiplications is "+
        MatrixChainOrder(arr, 1, n-1));

}
/* This code is contributed by Rajat Mishra*/

```

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for a matrix chain of size 4. The function MatrixChainOrder(p, 3, 4) is called two times. We can see that there are many subproblems being called more than once.



Since same subproblems are called again, this problem has Overlapping Subproblems property. So Matrix Chain Multiplication problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem. Like other typical [Dynamic Programming\(DP\) problems](#), recomputations of same subproblems can be avoided by constructing a temporary array m[][] in bottom up manner.

Dynamic Programming Solution

Following is C/C++ implementation for Matrix Chain Multiplication problem using Dynamic Programming.

C

```

// See the Cormen book for details of the following algorithm
#include<stdio.h>
#include<limits.h>

// Matrix A[i] has dimension p[i-1] x p[i] for i = 1..n
int MatrixChainOrder(int p[], int n)
{

    /* For simplicity of the program, one extra row and one
       extra column are allocated in m[][]. 0th row and 0th
       column of m[][] are not used */
    int m[n][n];

    int i, j, k, L, q;

    /* m[i][j] = Minimum number of scalar multiplications needed
       to compute the matrix A[i]A[i+1]...A[j] = A[i..j] where
       dimension of A[i] is p[i-1] x p[i] */

    // cost is zero when multiplying one matrix.
    for (i=1; i<n; i++)
        m[i][i] = 0;

    // L is chain length.
    for (L=2; L<n; L++)
    {
        for (i=1; i<n-L+1; i++)
        {
            j = i+L-1;
            m[i][j] = INT_MAX;
            for (k=i; k<=j-1; k++)
            {
                // q = cost/scalar multiplications
                q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }

    return m[1][n-1];
}

int main()
{
    int arr[] = {1, 2, 3, 4};
    int size = sizeof(arr)/sizeof(arr[0]);

    printf("Minimum number of multiplications is %d ",
        MatrixChainOrder(arr, size));

    getchar();
    return 0;
}

```

Java

```

// Dynamic Programming Python implementation of Matrix
// Chain Multiplication.
// See the Cormen book for details of the following algorithm
class MatrixChainMultiplication
{
    // Matrix A[i] has dimension p[i-1] x p[i] for i = 1..n
    static int MatrixChainOrder(int p[], int n)
    {

        /* For simplicity of the program, one extra row and one
           extra column are allocated in m[][]. 0th row and 0th
           column of m[][] are not used */

```

```

extra column are allocated in m[][], 0th row and 0th
column of m[][] are not used */
int m[][] = new int[n][n];

int i, j, k, L, q;

/* m[i][j] = Minimum number of scalar multiplications needed
to compute the matrix A[i]A[i+1]..A[j] = A[i..j] where
dimension of A[i] is p[i-1] x p[i] */

// cost is zero when multiplying one matrix.
for (i = 1; i < n; i++)
    m[i][i] = 0;

// L is chain length.
for (L=2; L<n; L++)
{
    for (i=1; i<n-L+1; i++)
    {
        j = i+L-1;
        if(j == n) continue;
        m[i][j] = Integer.MAX_VALUE;
        for (k=i; k<=j-1; k++)
        {
            // q = cost/scalar multiplications
            q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
            if (q < m[i][j])
                m[i][j] = q;
        }
    }
}

return m[1][n-1];
}

// Driver program to test above function
public static void main(String args[])
{
    int arr[] = new int[] {1, 2, 3, 4};
    int size = arr.length;

    System.out.println("Minimum number of multiplications is "+
        MatrixChainOrder(arr, size));
}
*/
/* This code is contributed by Rajat Mishra*/

```

Python

```

# Dynamic Programming Python implementation of Matrix
# Chain Multiplication. See the Cormen book for details
# of the following algorithm
import sys

# Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
def MatrixChainOrder(p, n):
    # For simplicity of the program, one extra row and one
    # extra column are allocated in m[][][], 0th row and 0th
    # column of m[][][] are not used
    m = [[0 for x in range(n)] for x in range(n)]

    # m[i][j] = Minimum number of scalar multiplications needed
    # to compute the matrix A[i]A[i+1]..A[j] = A[i..j] where
    # dimension of A[i] is p[i-1] x p[i]

    # cost is zero when multiplying one matrix.
    for i in range(1, n):
        m[i][i] = 0

    # L is chain length.
    for L in range(2, n):
        for i in range(1, n-L+1):
            j = i+L-1
            m[i][j] = sys.maxint
            for k in range(i, j):

                # q = cost/scalar multiplications
                q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j]
                if q < m[i][j]:
                    m[i][j] = q

    return m[1][n-1]

# Driver program to test above function
arr = [1, 2, 3, 4]
size = len(arr)

print("Minimum number of multiplications is " +
    str(MatrixChainOrder(arr, size)))
# This Code is contributed by Bhavya Jain

```

Output:

Minimum number of multiplications is 18

Time Complexity: O(n^3)
Auxiliary Space: O(n^2)

Printing brackets in Matrix Chain Multiplication Problem

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Applications:

Minimum and Maximum values of an expression with * and +

References:

http://en.wikipedia.org/wiki/Matrix_chain_multiplication

<http://www.personal.kent.edu/~muhamma/Algorithms/MyAlgorithms/Dynamic/chainMatrixMult.htm>

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming
Matrix

Dynamic Programming | Set 9 (Binomial Coefficient)

Following are common definition of Binomial Coefficients.

1) A binomial coefficient $C(n, k)$ can be defined as the coefficient of X^k in the expansion of $(1 + X)^n$.

2) A binomial coefficient $C(n, k)$ also gives the number of ways, disregarding order, that k objects can be chosen from among n objects; more formally, the number of k -element subsets (or k -combinations) of an n -element set.

The Problem

Write a function that takes two parameters n and k and returns the value of Binomial Coefficient $C(n, k)$. For example, your function should return 6 for $n = 4$ and $k = 2$, and it should return 10 for $n = 5$ and $k = 2$.

1) Optimal Substructure

The value of $C(n, k)$ can be recursively calculated using following standard formula for Binomial Coefficients.

$$\begin{aligned} C(n, k) &= C(n-1, k-1) + C(n-1, k) \\ C(n, 0) &= C(n, n) = 1 \end{aligned}$$

Following is a simple recursive implementation that simply follows the recursive structure mentioned above.

C/C++

```
// A Naive Recursive Implementation
#include<stdio.h>

// Returns value of Binomial Coefficient C(n, k)
int binomialCoeff(int n, int k)
{
    // Base Cases
    if (k==0 || k==n)
        return 1;

    // Recur
    return binomialCoeff(n-1, k-1) + binomialCoeff(n-1, k);
}

/* Driver program to test above function*/
int main()
{
    int n = 5, k = 2;
    printf("Value of C(%d,%d) is %d ", n, k, binomialCoeff(n, k));
    return 0;
}
```

Python

```
# A naive recursive Python implementation

def binomialCoeff(n , k):

    if k==0 or k ==n :
        return 1

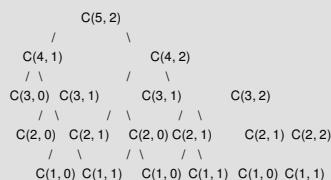
    # Recursive Call
    return binomialCoeff(n-1 , k-1) + binomialCoeff(n-1 , k)

# Driver Program to test ht above function
n = 5
k = 2
print "Value of C(%d,%d) is (%d)" %(n , k , binomialCoeff(n , k))

# This code is contributed by Nikhil Kumar Singh (nickzuck_007)
```

2) Overlapping Subproblems

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for $n = 5$ and $k = 2$. The function $C(3, 1)$ is called two times. For large values of n , there will be many common subproblems.



Since same subproblems are called again, this problem has Overlapping Subproblems property. So the Binomial Coefficient problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, re-computations of same subproblems can be avoided by constructing a temporary array $C[i][j]$ in bottom up manner. Following is Dynamic Programming based implementation.

C

```
// A Dynamic Programming based solution that uses table C[][] to
```

```

// calculate the Binomial Coefficient
#include<stdio.h>

// Prototype of a utility function that returns minimum of two integers
int min(int a, int b);

// Returns value of Binomial Coefficient C(n, k)
int binomialCoeff(int n, int k)
{
    int C[n+1][k+1];
    int i, j;

    // Calculate value of Binomial Coefficient in bottom up manner
    for (i = 0; i <= n; i++)
    {
        for (j = 0; j <= min(i, k); j++)
        {
            // Base Cases
            if (j == 0 || j == i)
                C[i][j] = 1;

            // Calculate value using previously stored values
            else
                C[i][j] = C[i-1][j-1] + C[i-1][j];
        }
    }

    return C[n][k];
}

// A utility function to return minimum of two integers
int min(int a, int b)
{
    return (a < b) ? a : b;
}

/* Driver program to test above function*/
int main()
{
    int n = 5, k = 2;
    printf ("Value of C(%d, %d) is %d ", n, k, binomialCoeff(n, k));
    return 0;
}

```

Java

```

// A Dynamic Programming based solution that uses table C[][] to
// calculate the Binomial Coefficient

class BinomialCoefficient
{
    // Returns value of Binomial Coefficient C(n, k)
    static int binomialCoeff(int n, int k)
    {
        int C[][] = new int[n+1][k+1];
        int i, j;

        // Calculate value of Binomial Coefficient in bottom up manner
        for (i = 0; i <= n; i++)
        {
            for (j = 0; j <= min(i, k); j++)
            {
                // Base Cases
                if (j == 0 || j == i)
                    C[i][j] = 1;

                // Calculate value using previously stored values
                else
                    C[i][j] = C[i-1][j-1] + C[i-1][j];
            }
        }

        return C[n][k];
    }

    // A utility function to return minimum of two integers
    static int min(int a, int b)
    {
        return (a < b) ? a : b;
    }

    /* Driver program to test above function*/
    public static void main(String args[])
    {
        int n = 5, k = 2;
        System.out.println("Value of C("+n+","+k+") is "+binomialCoeff(n, k));
    }
}
/*This code is contributed by Rajat Mishra*/

```

Python

```

# A Dynamic Programming based Python Program that uses table C[][]
# to calculate the Binomial Coefficient

# Returns value of Binomial Coefficient C(n, k)
def binomialCoeff(n, k):
    C = [[0 for x in range(k+1)] for x in range(n+1)]

    # Calculate value of Binomial Coefficient in bottom up manner
    for i in range(n+1):

```

```

for j in range(min(i, k)+1):
    # Base Cases
    if j == 0 or j == i:
        C[i][j] = 1

    # Calculate value using previously stored values
    else:
        C[i][j] = C[i-1][j-1] + C[i-1][j]

return C[n][k]

# Driver program to test above function
n = 5
k = 2
print("Value of C[" + str(n) + "][" + str(k) + "] is "
+ str(binomialCoeff(n,k)))

# This code is contributed by Bhavya Jain

```

Output:

Value of C[5][2] is 10

Time Complexity: O(n*k)

Auxiliary Space: O(n*k)

Following is a space optimized version of the above code. The following code only uses O(k). Thanks to [AK](#) for suggesting this method.

C/C++

```

// C++ program for space optimized Dynamic Programming
// Solution of Binomial Coefficient
#include<bits/stdc++.h>
using namespace std;

int binomialCoeff(int n, int k)
{
    int C[k+1];
    memset(C, 0, sizeof(C));

    C[0] = 1; // nC0 is 1

    for (int i = 1; i <= n; i++)
    {
        // Compute next row of pascal triangle using
        // the previous row
        for (int j = min(i, k); j > 0; j--)
            C[j] = C[j] + C[j-1];
    }
    return C[k];
}

/* Drier program to test above function*/
int main()
{
    int n = 5, k = 2;
    printf ("Value of C(%d,%d) is %d ",
           n, k, binomialCoeff(n, k));
    return 0;
}

```

Python

```

# Python program for Optimized Dynamic Programming solution to
# Binomial Coefficient. This one uses the concept of pascal
# Triangle and less memory

def binomialCoeff(n ,k):

    # Declaring an empty array
    C = [0 for i in xrange(k+1)]
    C[0] = 1 #since nC0 is 1

    for i in range(1,n+1):

        # Compute next row of pascal triangle using
        # the previous row
        j = min(i ,k)
        while (j>0):
            C[j] = C[j] + C[j-1]
            j -= 1

    return C[k]

# Driver Program to test the above function
n = 5
k = 2
print "Value of C(%d,%d) is %d" %(n,k,binomialCoeff(n,k))

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

Value of C[5][2] is 10

Time Complexity: O(n*k)

Auxiliary Space: O(k)

Explanation:

1=====>> n = 0, C(0,0) = 1

```

1-1=====>> n = 1, C(1,0) = 1, C(1,1) = 1
1-2-1=====>> n = 2, C(2,0) = 1, C(2,1) = 2, C(2,2) = 1
1-3-3-1=====>> n = 3, C(3,0) = 1, C(3,1) = 3, C(3,2) = 3, C(3,3)=1
1-4-6-4-1=====>> n = 4, C(4,0) = 1, C(4,1) = 4, C(4,2) = 6, C(4,3)=4, C(4,4)=1
So here every loop on i, builds i'th row of pascal triangle, using (i-1)th row

```

At any time, every element of array C will have some value (ZERO or more) and in next iteration, value for those elements comes from previous iteration.

In statement,

$C[i] = C[i] + C[j-1]$

Right hand side represents the value coming from previous iteration (A row of Pascal's triangle depends on previous row). Left Hand side represents the value of current iteration which will be obtained by this statement.

Let's say we want to calculate $C(4, 3)$,
i.e. n=4, k=3:

All elements of array C of size 4 (k+1) are initialized to ZERO.

i.e. $C[0] = C[1] = C[2] = C[3] = C[4] = 0$;
Then $C[0]$ is set to 1

For i = 1:
 $C[1] = C[1] + C[0] = 0 + 1 = 1 ==> C(1,1) = 1$

For i = 2:
 $C[2] = C[2] + C[1] = 0 + 1 = 1 ==> C(2,2) = 1$
 $C[1] = C[1] + C[0] = 1 + 1 = 2 ==> C(2,2) = 2$

For i=3:
 $C[3] = C[3] + C[2] = 0 + 1 = 1 ==> C(3,3) = 1$
 $C[2] = C[2] + C[1] = 1 + 2 = 3 ==> C(3,2) = 3$
 $C[1] = C[1] + C[0] = 2 + 1 = 3 ==> C(3,1) = 3$

For i=4:
 $C[4] = C[4] + C[3] = 0 + 1 = 1 ==> C(4,4) = 1$
 $C[3] = C[3] + C[2] = 1 + 3 = 4 ==> C(4,3) = 4$
 $C[2] = C[2] + C[1] = 3 + 3 = 6 ==> C(4,2) = 6$
 $C[1] = C[1] + C[0] = 3 + 1 = 4 ==> C(4,1) = 4$

$C(4,3) = 4$ is would be the answer in our example.

See this for Space and time efficient Binomial Coefficient

References:

<http://www.csl.mtu.edu/cs4321/www/Lectures/Lecture%2015%20-%20Dynamic%20Programming%20Binomial%20Coefficients.htm>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Mathematical
Dynamic Programming

Dynamic Programming | Set 10 (0-1 Knapsack Problem)

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don't pick it (0-1 property).

A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.

1) Optimal Substructure:

To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set.

Therefore, the maximum value that can be obtained from n items is max of following two values.

1) Maximum value obtained by n-1 items and W weight (excluding nth item).

2) Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

2) Overlapping Subproblems

Following is recursive implementation that simply follows the recursive structure mentioned above.

C/C++

```
/* A Naive recursive implementation of 0-1 Knapsack problem */
#include<stdio.h>

// A utility function that returns maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }

// Returns the maximum value that can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{
    // Base Case
    if (n == 0 || W == 0)
        return 0;

    // If weight of the nth item is more than Knapsack capacity W, then
    // this item cannot be included in the optimal solution
    if (wt[n-1] > W)
        return knapSack(W, wt, val, n-1);

    // Return the maximum of two cases:
    // (1) nth item included
    // (2) not included
    else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
                     knapSack(W, wt, val, n-1)
                 );
}

// Driver program to test above function
int main()
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val)/sizeof(val[0]);
    printf("%d", knapSack(W, wt, val, n));
    return 0;
}
```

Java

```
/* A Naive recursive implementation of 0-1 Knapsack problem */
class Knapsack
{

    // A utility function that returns maximum of two integers
    static int max(int a, int b) { return (a > b)? a : b; }

    // Returns the maximum value that can be put in a knapsack of capacity W
    static int knapSack(int W, int wt[], int val[], int n)
    {
        // Base Case
        if (n == 0 || W == 0)
            return 0;

        // If weight of the nth item is more than Knapsack capacity W, then
        // this item cannot be included in the optimal solution
        if (wt[n-1] > W)
            return knapSack(W, wt, val, n-1);

        // Return the maximum of two cases:
        // (1) nth item included
        // (2) not included
        else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
                         knapSack(W, wt, val, n-1)
                     );
    }

    // Driver program to test above function
    public static void main(String args[])
    {
        int val[] = new int[]{60, 100, 120};
        int wt[] = new int[]{10, 20, 30};
        int W = 50;
        int n = val.length;
        System.out.println(knapSack(W, wt, val, n));
    }
}
/*This code is contributed by Rajat Mishra */
```

Python

```
#A naive recursive implementation of 0-1 Knapsack Problem
```

```

# Returns the maximum value that can be put in a knapsack of
# capacity W
def knapSack(W , wt , val , n):

    # Base Case
    if n == 0 or W == 0 :
        return 0

    # If weight of the nth item is more than Knapsack of capacity
    # W, then this item cannot be included in the optimal solution
    if (wt[n-1] > W):
        return knapSack(W , wt , val , n-1)

    # return the maximum of two cases:
    # (1) nth item included
    # (2) not included
    else:
        return max(val[n-1] + knapSack(W-wt[n-1] , wt , val , n-1),
                   knapSack(W , wt , val , n-1))

# end of function knapSack

# To test above function
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print knapSack(W , wt , val , n)

# This code is contributed by Nikhil Kumar Singh

```

Output:

220

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree, $K(1, 1)$ is being evaluated twice. Time complexity of this naive recursive solution is exponential (2^n).

In the following recursion tree, $K()$ refers to `knapSack()`. The two parameters indicated in the following recursion tree are n and W .

The recursion tree is for following sample inputs.

$wt[] = \{1, 1, 1\}$, $W = 2$, $val[] = \{10, 20, 30\}$

```

      K(3, 2)      -----> K(n, W)
      /         \
     /           \
K(2,2)       K(2,1)
 /   \       /   \
 /     \   /   \
K(1,2)   K(1,1)   K(1,1)   K(1,0)
 / \   / \   / \
 / \   / \   / \
K(0,2) K(0,1) K(0,0) K(0,1) K(0,0)
Recursion tree for Knapsack capacity 2 units and 3 items of 1 unit weight.

```

Since suproblems are evaluated again, this problem has Overlapping Subproblems property. So the 0-1 Knapsack problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array $K[][]$ in bottom up manner. Following is Dynamic Programming based implementation.

C++

```

// A Dynamic Programming based solution for 0-1 Knapsack problem
#include<stdio.h>

// A utility function that returns maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }

// Returns the maximum value that can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{
    int i, w;
    int K[n+1][W+1];

    // Build table K[][] in bottom up manner
    for (i = 0; i <= n; i++)
    {
        for (w = 0; w <= W; w++)
        {
            if (i==0 || w==0)
                K[i][w] = 0;
            else if (wt[i-1] <= w)
                K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);
            else
                K[i][w] = K[i-1][w];
        }
    }

    return K[n][W];
}

int main()
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val)/sizeof(val[0]);
    printf("%d", knapSack(W, wt, val, n));
    return 0;
}

```

Java

```
// A Dynamic Programming based solution for 0-1 Knapsack problem
class Knapsack
{
    // A utility function that returns maximum of two integers
    static int max(int a, int b) { return (a > b)? a : b; }

    // Returns the maximum value that can be put in a knapsack of capacity W
    static int knapSack(int W, int wt[], int val[], int n)
    {
        int i, w;
        int K[][] = new int[n+1][W+1];

        // Build table K[][] in bottom up manner
        for (i = 0; i <= n; i++)
        {
            for (w = 0; w <= W; w++)
            {
                if (i==0 || w==0)
                    K[i][w] = 0;
                else if (wt[i-1] <= w)
                    K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);
                else
                    K[i][w] = K[i-1][w];
            }
        }

        return K[n][W];
    }

    // Driver program to test above function
    public static void main(String args[])
    {
        int val[] = new int[]{60, 100, 120};
        int wt[] = new int[]{10, 20, 30};
        int W = 50;
        int n = val.length;
        System.out.println(knapSack(W, wt, val, n));
    }
}
/*This code is contributed by Rajat Mishra */
```

Python

```
# A Dynamic Programming based Python Program for 0-1 Knapsack problem
# Returns the maximum value that can be put in a knapsack of capacity W
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W+1)] for x in range(n+1)]

    # Build table K[][] in bottom up manner
    for i in range(n+1):
        for w in range(W+1):
            if i==0 or w==0:
                K[i][w] = 0
            elif wt[i-1] <= w:
                K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w])
            else:
                K[i][w] = K[i-1][w]

    return K[n][W]

# Driver program to test above function
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print(knapSack(W, wt, val, n))

# This code is contributed by Bhavya Jain
```

Output:

220

Time Complexity: O(nW) where n is the number of items and W is the capacity of knapsack.

References:

<http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf>

<http://www.cse.unl.edu/~goddard/Courses/CSCE310J/Lectures/Lecture8-DynamicProgramming.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
knapsack

Dynamic Programming | Set 11 (Egg Dropping Puzzle)

The following is a description of the instance of this famous puzzle involving n=2 eggs and a building with k=36 floors.

Suppose that we wish to know which stories in a 36-story building are safe to drop eggs from, and which will cause the eggs to break on landing. We make a few assumptions:

-An egg that survives a fall can be used again.
-A broken egg must be discarded.
-The effect of a fall is the same for all eggs.
-If an egg breaks when dropped, then it would break if dropped from a higher floor.
-If an egg survives a fall then it would survive a shorter fall.
-It is not ruled out that the first-floor windows break eggs, nor is it ruled out that the 36th-floor do not cause an egg to break.

If only one egg is available and we wish to be sure of obtaining the right result, the experiment can be carried out in only one way. Drop the egg from the first-floor window; if it survives, drop it from the second floor window. Continue upward until it breaks. In the worst case, this method may require 36 droppings. Suppose 2 eggs are available. What is the least number of egg-droppings that is guaranteed to work in all cases?

The problem is not actually to find the critical floor, but merely to decide floors from which eggs should be dropped so that total number of trials are minimized.

Source: [Wiki for Dynamic Programming](#)

In this post, we will discuss solution to a general problem with n eggs and k floors. The solution is to try dropping an egg from every floor (from 1 to k) and recursively calculate the minimum number of droppings needed in worst case. The floor which gives the minimum value in worst case is going to be part of the solution.

In the following solutions, we return the minimum number of trials in worst case; these solutions can be easily modified to print floor numbers of every trials also.

1) Optimal Substructure:

When we drop an egg from a floor x, there can be two cases (1) The egg breaks (2) The egg doesn't break.

- 1) If the egg breaks after dropping from xth floor, then we only need to check for floors lower than x with remaining eggs; so the problem reduces to x-1 floors and n-1 eggs
- 2) If the egg doesn't break after dropping from the xth floor, then we only need to check for floors higher than x; so the problem reduces to k-x floors and n eggs.

Since we need to minimize the number of trials in worst case, we take the maximum of two cases. We consider the max of above two cases for every floor and choose the floor which yields minimum number of trials.

```
k ==> Number of floors
n ==> Number of Eggs
eggDrop(n, k) ==> Minimum number of trials needed to find the critical
                     floor in worst case.
eggDrop(n, k) = 1 + min(max(eggDrop(n - 1, x - 1), eggDrop(n, k - x));
x in {1, 2, ..., k})
```

2) Overlapping Subproblems

Following is recursive implementation that simply follows the recursive structure mentioned above.

```
# include <stdio.h>
# include <limits.h>

// A utility function to get maximum of two integers
int max(int a, int b) { return (a > b)? a: b; }

/* Function to get minimum number of trials needed in worst
   case with n eggs and k floors */
int eggDrop(int n, int k)
{
    // If there are no floors, then no trials needed. OR if there is
    // one floor, one trial needed.
    if (k == 1 || k == 0)
        return k;

    // We need k trials for one egg and k floors
    if (n == 1)
        return k;

    int min = INT_MAX, res;

    // Consider all droppings from 1st floor to kth floor and
    // return the minimum of these values plus 1.
    for (x = 1; x <= k; x++)
    {
        res = max(eggDrop(n-1, x-1), eggDrop(n, k-x));
        if (res < min)
            min = res;
    }

    return min + 1;
}
```

```

}

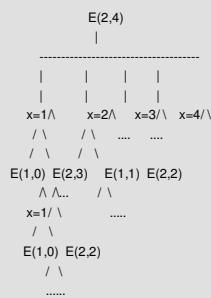
/* Driver program to test to printDups*/
int main()
{
    int n = 2, k = 10;
    printf ("\nMinimum number of trials in worst case with %d eggs and "
           "%d floors is %d \n", n, k, eggDrop(n, k));
    return 0;
}

```

Output:

Minimum number of trials in worst case with 2 eggs and 10 floors is 4

It should be noted that the above function computes the same subproblems again and again. See the following partial recursion tree, E(2, 2) is being evaluated twice. There will many repeated subproblems when you draw the complete recursion tree even for small values of n and k.



Partial recursion tree for 2 eggs and 4 floors.

Since same subproblems are called again, this problem has Overlapping Subproblems property. So Egg Dropping Puzzle has both properties (see [this](#) and [this](#)) of a dynamic programming problem. Like other typical [Dynamic Programming\(DP\) problems](#), recomputations of same subproblems can be avoided by constructing a temporary array eggFloor[][] in bottom up manner.

Dynamic Programming Solution

Following are C++ and Python implementations for Egg Dropping problem using Dynamic Programming.

C++

```

# A Dynamic Programming based C++ Program for the Egg Dropping Puzzle
# include <stdio.h>
# include <limits.h>

// A utility function to get maximum of two integers
int max(int a, int b) { return (a > b)? a: b; }

/* Function to get minimum number of trials needed in worst
case with n eggs and k floors */
int eggDrop(int n, int k)
{
    /* A 2D table where entry eggFloor[i][j] will represent minimum
    number of trials needed for i eggs and j floors. */
    int eggFloor[n+1][k+1];
    int res;
    int i, j, x;

    // We need one trial for one floor and 0 trials for 0 floors
    for (i = 1; i <= n; i++)
    {
        eggFloor[i][1] = 1;
        eggFloor[i][0] = 0;
    }

    // We always need j trials for one egg and j floors.
    for (j = 1; j <= k; j++)
        eggFloor[1][j] = j;

    // Fill rest of the entries in table using optimal substructure
    // property
    for (i = 2; i <= n; i++)
    {
        for (j = 2; j <= k; j++)
        {
            for (x = 1; x <= j; x++)
            {
                res = 1 + max(eggFloor[i-1][x-1], eggFloor[i][j-x]);
                if (res < eggFloor[i][j])
                    eggFloor[i][j] = res;
            }
        }
    }

    // eggFloor[n][k] holds the result
    return eggFloor[n][k];
}

/* Driver program to test to printDups*/
int main()
{
    int n = 2, k = 36;
    printf ("\nMinimum number of trials in worst case with %d eggs and "
           "%d floors is %d \n", n, k, eggDrop(n, k));
    return 0;
}

```

Java

```
//A Dynamic Programming based Python Program for the Egg Dropping Puzzle
class EggDrop
{
    // A utility function to get maximum of two integers
    static int max(int a, int b) { return (a > b)? a: b; }

    /* Function to get minimum number of trials needed in worst
    case with n eggs and k floors */
    static int eggDrop(int n, int k)
    {
        /* A 2D table where entry eggFloor[i][j] will represent minimum
        number of trials needed for i eggs and j floors.*/
        int eggFloor[][] = new int[n+1][k+1];
        int res;
        int i, j, x;

        // We need one trial for one floor and0 trials for 0 floors
        for (i = 1; i <= n; i++)
        {
            eggFloor[i][1] = 1;
            eggFloor[i][0] = 0;
        }

        // We always need j trials for one egg and j floors.
        for (j = 1; j <= k; j++)
            eggFloor[1][j] = j;

        // Fill rest of the entries in table using optimal substructure
        // property
        for (i = 2; i <= n; i++)
        {
            for (j = 2; j <= k; j++)
            {
                eggFloor[i][j] = Integer.MAX_VALUE;
                for (x = 1; x <= j; x++)
                {
                    res = 1 + max(eggFloor[i-1][x-1], eggFloor[i][j-x]);
                    if (res < eggFloor[i][j])
                        eggFloor[i][j] = res;
                }
            }
        }

        // eggFloor[n][k] holds the result
        return eggFloor[n][k];
    }

    /* Driver program to test to print printDups*/
    public static void main(String args[])
    {
        int n = 2, k = 10;
        System.out.println("Minimum number of trials in worst case with "+n+" eggs and "+k+
                           " floors is "+eggDrop(n, k));
    }
}

/*This code is contributed by Rajat Mishra*/
```

Python

```
# A Dynamic Programming based Python Program for the Egg Dropping Puzzle
INT_MAX = 32767

# Function to get minimum number of trials needed in worst
# case with n eggs and k floors
def eggDrop(n, k):
    # A 2D table where entry eggFloor[i][j] will represent minimum
    # number of trials needed for i eggs and j floors.
    eggFloor = [[0 for x in range(k+1)] for x in range(n+1)]

    # We need one trial for one floor and0 trials for 0 floors
    for i in range(1, n+1):
        eggFloor[i][1] = 1
        eggFloor[i][0] = 0

    # We always need j trials for one egg and j floors.
    for j in range(1, k+1):
        eggFloor[1][j] = j

    # Fill rest of the entries in table using optimal substructure
    # property
    for i in range(2, n+1):
        for j in range(2, k+1):
            eggFloor[i][j] = INT_MAX
            for x in range(1, j+1):
                res = 1 + max(eggFloor[i-1][x-1], eggFloor[i][j-x])
                if res < eggFloor[i][j]:
                    eggFloor[i][j] = res

    # eggFloor[n][k] holds the result
    return eggFloor[n][k]

# Driver program to test to print printDups
n = 2
k = 36
print("Minimum number of trials in worst case with " + str(n) + " eggs and " +
      + str(k) + " floors is " + str(eggDrop(n, k)))

# This code is contributed by Bhavya Jain
```

Output:
Minimum number of trials in worst case with 2 eggs and 36 floors is 8

Time Complexity: O(nk^2)
Auxiliary Space: O(nk)

As an exercise, you may try modifying the above DP solution to print all intermediate floors (The floors used for minimum trial solution).

2 Eggs and 100 Floor Puzzle

References:

<http://archive.ite.journal.informs.org/Vol4No1/Sniedovich/index.php>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming

Dynamic Programming | Set 12 (Longest Palindromic Subsequence)

Given a sequence, find the length of the longest palindromic subsequence in it. For example, if the given sequence is "BBABCBCAB", then the output should be 7 as "BACBABA" is the longest palindromic subsequence in it. "BBBBB" and "BBCBB" are also palindromic subsequences of the given sequence, but not the longest ones.

The naive solution for this problem is to generate all subsequences of the given sequence and find the longest palindromic subsequence. This solution is exponential in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem and can efficiently solved using Dynamic Programming.

1) Optimal Substructure:

Let X[0..n-1] be the input sequence of length n and L(0, n-1) be the length of the longest palindromic subsequence of X[0..n-1].

If last and first characters of X are same, then L(0, n-1) = L(1, n-2) + 2.

Else L(0, n-1) = MAX (L(1, n-1), L(0, n-2)).

Following is a general recursive solution with all cases handled.

```
// Every single character is a palindrome of length 1
L(i, i) = 1 for all indexes i in given sequence

// If first and last characters are not same
If (X[i] != X[j]) L(i, j) = max{L(i + 1, j), L(i, j - 1)}

// If there are only 2 characters and both are same
Else if (j == i + 1) L(i, j) = 2

// If there are more than two characters, and first and last
// characters are same
Else L(i, j) = L(i + 1, j - 1) + 2
```

2) Overlapping Subproblems

Following is simple recursive implementation of the LPS problem. The implementation simply follows the recursive structure mentioned above.

```
#include<stdio.h>
#include<string.h>

// A utility function to get max of two integers
int max (int x, int y) { return (x > y)? x : y; }

// Returns the length of the longest palindromic subsequence in seq
int lps(char *seq, int i, int j)
{
    // Base Case 1: If there is only 1 character
    if (i == j)
        return 1;

    // Base Case 2: If there are only 2 characters and both are same
    if (seq[i] == seq[j] && i + 1 == j)
        return 2;

    // If the first and last characters match
    if (seq[i] == seq[j])
        return lps (seq, i+1, j-1) + 2;

    // If the first and last characters do not match
    return max( lps(seq, i, j-1), lps(seq, i+1, j) );
}

/* Driver program to test above functions */
int main()
{
    char seq[] = "GEEKSFORGEEKS";
    int n = strlen(seq);
```

```

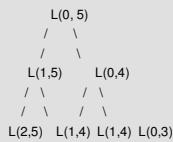
printf ("The length of the LPS is %d", lps(seq, 0, n-1));
getchar();
return 0;
}

```

Output:

```
The length of the LPS is 5
```

Considering the above implementation, following is a partial recursion tree for a sequence of length 6 with all different characters.



In the above partial recursion tree, $L(1, 4)$ is being solved twice. If we draw the complete recursion tree, then we can see that there are many subproblems which are solved again and again. Since same subproblems are called again, this problem has Overlapping Subproblems property. So LPS problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array $L[i][j]$ in bottom up manner.

Dynamic Programming Solution

C++

```

// A Dynamic Programming based Python program for LPS problem
// Returns the length of the longest palindromic subsequence in seq
#include<stdio.h>
#include<string.h>

// A utility function to get max of two integers
int max (int x, int y) { return (x > y)? x : y; }

// Returns the length of the longest palindromic subsequence in seq
int lps(char *str)
{
    int n = strlen(str);
    int i, j, cl;
    int L[n][n]; // Create a table to store results of subproblems

    // Strings of length 1 are palindrome of length 1
    for (i = 0; i < n; i++)
        L[i][i] = 1;

    // Build the table. Note that the lower diagonal values of table are
    // useless and not filled in the process. The values are filled in a
    // manner similar to Matrix Chain Multiplication DP solution (See
    // http://www.geeksforgeeks.org/archives/15553). cl is length of
    // substring
    for (cl=2; cl<=n; cl++)
    {
        for (i=0; i<n-cl+1; i++)
        {
            j = i+cl-1;
            if (str[i] == str[j] && cl == 2)
                L[i][j] = 2;
            else if (str[i] == str[j])
                L[i][j] = L[i+1][j-1] + 2;
            else
                L[i][j] = max(L[i][j-1], L[i+1][j]);
        }
    }

    return L[0][n-1];
}

/* Driver program to test above functions */
int main()
{
    char seq[] = "GEEKS FOR GEEKS";
    int n = strlen(seq);
    printf ("The length of the LPS is %d", lps(seq));
    getchar();
    return 0;
}

```

Java

```

//A Dynamic Programming based Python Program for the Egg Dropping Puzzle
class LPS
{

    // A utility function to get max of two integers
    static int max (int x, int y) { return (x > y)? x : y; }

    // Returns the length of the longest palindromic subsequence in seq
    static int lps(String seq)
    {
        int n = seq.length();
        int i, j, cl;
        int L[][] = new int[n][n]; // Create a table to store results of subproblems

        // Strings of length 1 are palindrome of length 1
        for (i = 0; i < n; i++)
            L[i][i] = 1;

        // Build the table. Note that the lower diagonal values of table are
        // useless and not filled in the process. The values are filled in a
        // manner similar to Matrix Chain Multiplication DP solution (See
        // http://www.geeksforgeeks.org/archives/15553). cl is length of
        // substring
        for (cl=2; cl<=n; cl++)
        {
            for (i=0; i<n-cl+1; i++)
            {
                j = i+cl-1;
                if (str[i] == str[j] && cl == 2)
                    L[i][j] = 2;
                else if (str[i] == str[j])
                    L[i][j] = L[i+1][j-1] + 2;
                else
                    L[i][j] = max(L[i][j-1], L[i+1][j]);
            }
        }

        return L[0][n-1];
    }
}

```

```

// manner similar to Matrix Chain Multiplication DP solution (See
// http://www.geeksforgeeks.org/archives/15553). cl is length of
// substring
for (cl=2; cl<=n; cl++)
{
    for (i=0; i<n-cl+1; i++)
    {
        j = i+cl-1;
        if (seq.charAt(i) == seq.charAt(j) && cl == 2)
            L[i][j] = 2;
        else if (seq.charAt(i) == seq.charAt(j))
            L[i][j] = L[i+1][j-1] + 2;
        else
            L[i][j] = max(L[i][j-1], L[i+1][j]);
    }
}

return L[0][n-1];
}

/* Driver program to test above functions */
public static void main(String args[])
{
    String seq = "GEEKSFORGEEKS";
    int n = seq.length();
    System.out.println("The length of the LPS is "+ lps(seq));
}
}

/* This code is contributed by Rajat Mishra */

```

Python

```

# A Dynamic Programming based Python program for LPS problem
# Returns the length of the longest palindromic subsequence in seq
def lps(str):
    n = len(str)

    # Create a table to store results of subproblems
    L = [[0 for x in range(n)] for x in range(n)]

    # Strings of length 1 are palindrome of length 1
    for i in range(n):
        L[i][i] = 1

    # Build the table. Note that the lower diagonal values of table are
    # useless and not filled in the process. The values are filled in a
    # manner similar to Matrix Chain Multiplication DP solution (See
    # http://www.geeksforgeeks.org/dynamic-programming-set-8-matrix-chain-multiplication/
    # cl is length of substring
    for cl in range(2, n+1):
        for i in range(n-cl+1):
            j = i+cl-1
            if str[i] == str[j] and cl == 2:
                L[i][j] = 2
            elif str[i] == str[j]:
                L[i][j] = L[i+1][j-1] + 2
            else:
                L[i][j] = max(L[i][j-1], L[i+1][j]);

    return L[0][n-1]

# Driver program to test above functions
seq = "GEEKS FOR GEEKS"
n = len(seq)
print("The length of the LPS is "+ str(lps(seq)))

# This code is contributed by Bhavya Jain

```

Output:

The length of the LPS is 7

Time Complexity of the above implementation is $O(n^2)$ which is much better than the worst case time complexity of Naive Recursive implementation.

This problem is close to the [Longest Common Subsequence \(LCS\) problem](#). In fact, we can use LCS as a subroutine to solve this problem. Following is the two step solution that uses LCS.

- 1) Reverse the given sequence and store the reverse in another array say rev[0..n-1]
 - 2) LCS of the given sequence and rev[] will be the longest palindromic sequence.
- This solution is also a $O(n^2)$ solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:

<http://users.eecs.northwestern.edu/~dda902/336/hw6-sol.pdf>

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming
palindrome
subsequence

Dynamic Programming | Set 13 (Cutting a Rod)

Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n . Determine the maximum value obtainable by cutting up the rod and selling the pieces. For example, if length of the rod is 8 and the values of different pieces are given as following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6)

length	1	2	3	4	5	6	7	8

price	1	5	8	9	10	17	17	20

And if the prices are as following, then the maximum obtainable value is 24 (by cutting in eight pieces of length 1)

length	1	2	3	4	5	6	7	8
<hr/>								
price	3	5	8	9	10	17	17	20

The naive solution for this problem is to generate all configurations of different pieces and find the highest priced configuration. This solution is exponential in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem and can efficiently solved using Dynamic Programming.

1) Optimal Substructure:

We can get the best price by making a cut at different positions and comparing the values obtained after a cut. We can recursively call the same function for a piece obtained after a cut.

Let $\text{cutRod}(n)$ be the required (best possible price) value for a rod of length n . $\text{cutRod}(n)$ can be written as following.

$$\text{cutRod}(n) = \max(\text{price}[i] + \text{cutRod}(n-i)) \text{ for all } i \in \{0, 1 \dots n-1\}$$

2) Overlapping Subproblems

Following is simple recursive implementation of the Rod Cutting problem. The implementation simply follows the recursive structure mentioned above.

C++

```
// A Naive recursive solution for Rod cutting problem
#include<stdio.h>
#include<limits.h>

// A utility function to get the maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }

/* Returns the best obtainable price for a rod of length n and
   price[] as prices of different pieces */
int cutRod(int price[], int n)
{
    if (n <= 0)
        return 0;
    int max_val = INT_MIN;

    // Recursively cut the rod in different pieces and compare different
    // configurations
    for (int i = 0; i < n; i++)
        max_val = max(max_val, price[i] + cutRod(price, n-i));

    return max_val;
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {1, 5, 8, 9, 10, 17, 17, 20};
    int size = sizeof(arr)/sizeof(arr[0]);
    printf("Maximum Obtainable Value is %d\n", cutRod(arr, size));
    getchar();
    return 0;
}
```

Java

```
/* // A Naive recursive solution for Rod cutting problem
class RodCutting
{
    /* Returns the best obtainable price for a rod of length
       n and price[] as prices of different pieces */
    static int cutRod(int price[], int n)
    {
        if (n <= 0)
            return 0;
        int max_val = Integer.MIN_VALUE;

        // Recursively cut the rod in different pieces and
        // compare different configurations
        for (int i = 0; i < n; i++)
            max_val = Math.max(max_val,
                               price[i] + cutRod(price, n-i));

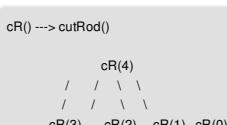
        return max_val;
    }

    /* Driver program to test above functions */
    public static void main(String args[])
    {
        int arr[] = new int[] {1, 5, 8, 9, 10, 17, 17, 20};
        int size = arr.length;
        System.out.println("Maximum Obtainable Value is "+
                           cutRod(arr, size));
    }
}
/* This code is contributed by Rajat Mishra */
```

Output:

```
Maximum Obtainable Value is 22
```

Considering the above implementation, following is recursion tree for a Rod of length 4.



```

    / | \   / \   |
    / \ \ / \ \ |
cR(2) cR(1) cR(0) cR(1) cR(0) cR(0)
/\   |   |
/ \   |   |
cR(1) cR(0) cR(0)   cR(0)
/
/
CR(0)

```

In the above partial recursion tree, $cR(2)$ is being solved twice. We can see that there are many subproblems which are solved again and again. Since same subproblems are called again, this problem has Overlapping Subproblems property. So the Rod Cutting problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array $val[]$ in bottom up manner.

C++

```

// A Dynamic Programming solution for Rod cutting problem
#include<stdio.h>
#include<limits.h>

// A utility function to get the maximum of two integers
int max(int a, int b) { return (a > b)? a : b; }

/* Returns the best obtainable price for a rod of length n and
   price[] as prices of different pieces */
int cutRod(int price[], int n)
{
    int val[n+1];
    val[0] = 0;
    int i, j;

    // Build the table val[] in bottom up manner and return the last entry
    // from the table
    for (i = 1; i<=n; i++)
    {
        int max_val = INT_MIN;
        for (j = 0; j < i; j++)
            max_val = max(max_val, price[j] + val[i-j-1]);
        val[i] = max_val;
    }

    return val[n];
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {1, 5, 8, 9, 10, 17, 17, 20};
    int size = sizeof(arr)/sizeof(arr[0]);
    printf("Maximum Obtainable Value is %d\n", cutRod(arr, size));
    getchar();
    return 0;
}

```

Java

```

// A Dynamic Programming solution for Rod cutting problem
class RodCutting
{
    /* Returns the best obtainable price for a rod of
       length n and price[] as prices of different pieces */
    static int cutRod(int price[],int n)
    {
        int val[] = new int[n+1];
        val[0] = 0;

        // Build the table val[] in bottom up manner and return
        // the last entry from the table
        for (int i = 1; i<=n; i++)
        {
            int max_val = Integer.MIN_VALUE;
            for (int j = 0; j < i; j++)
                max_val = Math.max(max_val,
                                   price[j] + val[i-j-1]);
            val[i] = max_val;
        }

        return val[n];
    }

    /* Driver program to test above functions */
    public static void main(String args[])
    {
        int arr[] = new int[] {1, 5, 8, 9, 10, 17, 17, 20};
        int size = arr.length;
        System.out.println("Maximum Obtainable Value is " +
                           cutRod(arr, size));
    }
}
/* This code is contributed by Rajat Mishra */

```

Python

```

# A Dynamic Programming solution for Rod cutting problem
INT_MIN = -32767

# Returns the best obtainable price for a rod of length n and
# price[] as prices of different pieces
def cutRod(price, n):

```

```

val = [0 for x in range(n+1)]
val[0] = 0

# Build the table val[] in bottom up manner and return
# the last entry from the table
for i in range(1, n+1):
    max_val = INT_MIN
    for j in range(i):
        max_val = max(max_val, price[j] + val[i-j-1])
    val[i] = max_val

return val[n]

# Driver program to test above functions
arr = [1, 5, 8, 9, 10, 17, 17, 20]
size = len(arr)
print("Maximum Obtainable Value is " + str(cutRod(arr, size)))

# This code is contributed by Bhavya Jain

```

Output:

Maximum Obtainable Value is 22

Time Complexity of the above implementation is $O(n^2)$ which is much better than the worst case time complexity of Naive Recursive implementation.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming

Dynamic Programming | Set 14 (Maximum Sum Increasing Subsequence)

Given an array of n positive integers. Write a program to find the sum of maximum sum subsequence of the given array such that the integers in the subsequence are sorted in increasing order. For example, if input is {1, 101, 2, 3, 100, 4, 5}, then output should be 106 (1 + 2 + 3 + 100), if the input array is {3, 4, 5, 10}, then output should be 22 (3 + 4 + 5 + 10) and if the input array is {10, 5, 4, 3}, then output should be 10

Solution

This problem is a variation of standard [Longest Increasing Subsequence \(LIS\) problem](#). We need a slight change in the Dynamic Programming solution of LIS problem. All we need to change is to use sum as a criteria instead of length of increasing subsequence.

Following are C/C++ and Python implementations for Dynamic Programming solution of the problem.

C/C++

```

/* Dynamic Programming implementation of Maximum Sum Increasing
Subsequence (MSIS) problem */
#include<stdio.h>

/* maxSumIS() returns the maximum sum of increasing subsequence
in arr[] of size n */
int maxSumIS( int arr[], int n )
{
    int i, j, max = 0;
    int msis[n];

    /* Initialize msis values for all indexes */
    for ( i = 0; i < n; i++ )
        msis[i] = arr[i];

    /* Compute maximum sum values in bottom up manner */
    for ( i = 1; i < n; i++ )
        for ( j = 0; j < i; j++ )
            if ( arr[i] > arr[j] && msis[j] < msis[i] + arr[i] )
                msis[i] = msis[i] + arr[i];

    /* Pick maximum of all msis values */
    for ( i = 0; i < n; i++ )
        if ( max < msis[i] )
            max = msis[i];

    return max;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {1, 101, 2, 3, 100, 4, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Sum of maximum sum increasing subsequence is %d\n",
           maxSumIS( arr, n ) );
    return 0;
}

```

Java

```

/* Dynamic Programming Java implementation of Maximum Sum
Increasing Subsequence (MSIS) problem */
class MSIS
{
    /* maxSumIS() returns the maximum sum of increasing
    subsequence in arr[] of size n */
    static int maxSumIS( int arr[], int n )
    {
        int i, j, max = 0;
        int msis[] = new int[n];

```

```

/* Initialize msis values for all indexes */
for ( i = 0; i < n; i++ )
    msis[i] = arr[i];

/* Compute maximum sum values in bottom up manner */
for ( i = 1; i < n; i++ )
    for ( j = 0; j < i; j++ )
        if ( arr[j] > arr[i] ) &&
            msis[i] < msis[j] + arr[i])
            msis[i] = msis[j] + arr[i];

/* Pick maximum of all msis values */
for ( i = 0; i < n; i++ )
    if ( max < msis[i] )
        max = msis[i];

return max;
}

/* Driver program to test above function */
public static void main(String args[])
{
    int arr[] = new int[]{11, 101, 2, 3, 100, 4, 5};
    int n = arr.length;
    System.out.println("Sum of maximum sum increasing "+
        "subsequence is "+maxSumIS( arr, n ));
}
*/
/* This code is contributed by Rajat Mishra */

```

Python

```

# Dynamic Programming based Python implementation of Maximum Sum Increasing
# Subsequence (MSIS) problem

# maxSumIS() returns the maximum sum of increasing subsequence in arr[] of
# size n
def maxSumIS(arr, n):
    max = 0
    msis = [0 for x in range(n)]

    # Initialize msis values for all indexes
    for i in range(n):
        msis[i] = arr[i]

    # Compute maximum sum values in bottom up manner
    for i in range(1, n):
        for j in range(i):
            if arr[j] > arr[i] and msis[i] < msis[j] + arr[i]:
                msis[i] = msis[j] + arr[i]

    # Pick maximum of all msis values
    for i in range(n):
        if max < msis[i]:
            max = msis[i]

    return max

# Driver program to test above function
arr = [1, 101, 2, 3, 100, 4, 5]
n = len(arr)
print("Sum of maximum sum increasing subsequence is " +
      str(maxSumIS(arr, n)))

# This code is contributed by Bhavya Jain

```

Output:

Sum of maximum sum increasing subsequence is 106

Time Complexity: O(n^2)

Source: Maximum Sum Increasing Subsequence Problem

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming
LIS

Dynamic Programming | Set 15 (Longest Bitonic Subsequence)

Given an array arr[0 ... n-1] containing n positive integers, a **subsequence** of arr[] is called Bitonic if it is first increasing, then decreasing. Write a function that takes an array as argument and returns the length of the longest bitonic subsequence.

A sequence, sorted in increasing order is considered Bitonic with the decreasing part as empty. Similarly, decreasing order sequence is considered Bitonic with the increasing part as empty.

Examples:

Input arr[] = {1, 11, 2, 10, 4, 5, 2, 1};
Output: 6 (A Longest Bitonic Subsequence of length 6 is 1, 2, 10, 4, 2, 1)

Input arr[] = {12, 11, 40, 5, 3, 1}
Output: 5 (A Longest Bitonic Subsequence of length 5 is 12, 11, 5, 3, 1)

Input arr[] = {80, 60, 30, 40, 20, 10}
Output: 5 (A Longest Bitonic Subsequence of length 5 is 80, 60, 30, 20, 10)

Source: Microsoft Interview Question

Solution

This problem is a variation of standard [Longest Increasing Subsequence \(LIS\) problem](#). Let the input array be arr[] of length n. We need to construct two arrays lis[] and lds[] using Dynamic Programming solution of [LIS problem](#). lis[i] stores the length of the Longest Increasing subsequence ending with arr[i]. lds[i] stores the length of the longest Decreasing subsequence starting from arr[i]. Finally, we need to return the max value of lis[i] + lds[i] - 1 where i is from 0 to n-1.

Following is C++ implementation of the above Dynamic Programming solution.

C++

```
/* Dynamic Programming implementation of longest bitonic subsequence problem */
#include<stdio.h>
#include<stdlib.h>

/* lbs() returns the length of the Longest Bitonic Subsequence in
arr[] of size n. The function mainly creates two temporary arrays
lis[] and lds[] and returns the maximum lis[i] + lds[i] - 1.

lis[i] ==> Longest Increasing subsequence ending with arr[i]
lds[i] ==> Longest decreasing subsequence starting with arr[i]

*/
int lbs( int arr[], int n )
{
    int i, j;

    /* Allocate memory for LIS[] and initialize LIS values as 1 for
    all indexes */
    int *lis = new int[n];
    for (i = 0; i < n; i++)
        lis[i] = 1;

    /* Compute LIS values from left to right */
    for (i = 1; i < n; i++)
        for (j = 0; j < i; j++)
            if (arr[i] > arr[j] && lis[i] < lis[j] + 1)
                lis[i] = lis[j] + 1;

    /* Allocate memory for lds and initialize LDS values for
    all indexes */
    int *lds = new int[n];
    for (i = 0; i < n; i++)
        lds[i] = 1;

    /* Compute LDS values from right to left */
    for (i = n-2; i >= 0; i--)
        for (j = n-1; j > i; j--)
            if (arr[i] > arr[j] && lds[i] < lds[j] + 1)
                lds[i] = lds[j] + 1;

    /* Return the maximum value of lis[i] + lds[i] - 1 */
    int max = lis[0] + lds[0] - 1;
    for (i = 1; i < n; i++)
        if (lis[i] + lds[i] - 1 > max)
            max = lis[i] + lds[i] - 1;
    return max;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5,
                13, 3, 11, 7, 15};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Length of LBS is %d\n", lbs( arr, n ) );
    return 0;
}
```

Java

```
/* Dynamic Programming implementation in Java for longest bitonic
subsequence problem */
import java.util.*;
import java.lang.*;
import java.io.*;

class LBS
{
    /* lbs() returns the length of the Longest Bitonic Subsequence in
    arr[] of size n. The function mainly creates two temporary arrays
    lis[] and lds[] and returns the maximum lis[i] + lds[i] - 1.

    lis[i] ==> Longest Increasing subsequence ending with arr[i]
    lds[i] ==> Longest decreasing subsequence starting with arr[i]

    */
    static int lbs( int arr[], int n )
    {
        int i, j;

        /* Allocate memory for LIS[] and initialize LIS values as 1 for
        all indexes */
        int[] lis = new int[n];
        for (i = 0; i < n; i++)
            lis[i] = 1;

        /* Compute LIS values from left to right */
        for (i = 1; i < n; i++)
            for (j = 0; j < i; j++)
                if (arr[i] > arr[j] && lis[i] < lis[j] + 1)
                    lis[i] = lis[j] + 1;
```

```

/* Allocate memory for lds and initialize LDS values for
all indexes */
int[] lds = new int[n];
for (i = 0; i < n; i++)
    lds[i] = 1;

/* Compute LDS values from right to left */
for (i = n-2; i >= 0; i--)
    for (j = n-1; j > i; j--)
        if (arr[i] > arr[j] && lds[i] < lds[j] + 1)
            lds[i] = lds[j] + 1;

/* Return the maximum value of lis[i] + lds[i] - 1 */
int max = lis[0] + lds[0] - 1;
for (i = 1; i < n; i++)
    if ((lis[i] + lds[i] - 1) > max)
        max = lis[i] + lds[i] - 1;

return max;
}

public static void main (String[] args)
{
    int arr[] = {0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5,
                13, 3, 11, 7, 15};
    int n = arr.length;
    System.out.println("Length of LBS is "+ lbs( arr, n ));
}

```

Python

```

# Dynamic Programming implementation of longest bitonic subsequence problem
"""

lbs() returns the length of the Longest Bitonic Subsequence in
arr[] of size n. The function mainly creates two temporary arrays
lis[] and lds[] and returns the maximum lis[i] + lds[i] - 1.

lis[i] ==> Longest Increasing subsequence ending with arr[i]
lds[i] ==> Longest decreasing subsequence starting with arr[i]
"""

def lbs(arr):
    n = len(arr)

    # allocate memory for LIS[] and initialize LIS values as 1
    # for all indexes
    lis = [1 for i in range(n+1)]

    # Compute LIS values from left to right
    for i in range(1 ,n):
        for j in range(0 ,i):
            if ((arr[i] > arr[j]) and (lis[i] < lis[j] +1)):
                lis[i] = lis[j] + 1

    # allocate memory for LDS and initialize LDS values for
    # all indexes
    lds = [1 for i in range(n+1)]

    # Compute LDS values from right to left
    for i in reversed(range(n-1)): #loop from n-2 downto 0
        for j in reversed(range(-1 ,i)): #loop from n-1 downto i-1
            if((arr[i] > arr[j]) and lds[i] < lds[j] + 1):
                lds[i] = lds[j] + 1

    # Return the maximum value of (lis[i] + lds[i] - 1)
    maximum = lis[0] + lds[0] - 1
    for i in range(1 ,n):
        maximum = max((lis[i] + lds[i]-1), maximum)

    return maximum

# Driver program to test the above function
arr = [0, 8, 4, 12, 2, 10 ,6 ,14 ,1 ,9 ,5 ,13,
       3, 11 ,7 ,15]
print "Length of LBS is",lbs(arr)

# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

Length of LBS is 7

Time Complexity: O(n^2)
Auxiliary Space: O(n)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming
subsequence

Dynamic Programming | Set 16 (Floyd Warshall Algorithm)

The [Floyd Warshall Algorithm](#) is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

Example:

Input:
graph[0][0] = { 0, 5, INF, 10},
{INF, 0, 3, INF},
{INF, INF, 0, 1},
{INF, INF, INF, 0} }
which represents the following graph



Note that the value of $graph[i][j]$ is 0 if i is equal to j .
And $graph[i][j]$ is INF (infinite) if there is no edge from vertex i to j .

Output:

Shortest distance matrix
0 5 8 9
INF 0 3 4
INF INF 0 1
INF INF INF 0

Floyd Warshall Algorithm

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices $\{0, 1, 2, \dots, k-1\}$ as intermediate vertices. For every pair (i, j) of source and destination vertices respectively, there are two possible cases.

1) k is not an intermediate vertex in shortest path from i to j . We keep the value of $dist[i][j]$ as it is.

2) k is an intermediate vertex in shortest path from i to j . We update the value of $dist[i][j]$ as $dist[i][k] + dist[k][j]$.

The following figure is taken from the Cormen book. It shows the above optimal substructure property in the all-pairs shortest path problem.



Following is implementations of the Floyd Warshall algorithm.

C/C++

```
// C Program for Floyd Warshall Algorithm  
#include<stdio.h>  
  
// Number of vertices in the graph  
#define V 4  
  
/* Define Infinite as a large enough value. This value will be used  
for vertices not connected to each other */  
#define INF 99999  
  
// A function to print the solution matrix  
void printSolution(int dist[V][V]);  
  
// Solves the all-pairs shortest path problem using Floyd Warshall algorithm  
void floydWarshall (int graph[V][V])  
{  
    /* dist[][] will be the output matrix that will finally have the shortest  
    distances between every pair of vertices */  
    int dist[V][V], i, j, k;  
  
    /* Initialize the solution matrix same as input graph matrix. Or  
    we can say the initial values of shortest distances are based  
    on shortest paths considering no intermediate vertex */  
    for (i = 0; i < V; i++)  
        for (j = 0; j < V; j++)  
            dist[i][j] = graph[i][j];  
  
    /* Add all vertices one by one to the set of intermediate vertices.  
    --> Before start of a iteration, we have shortest distances between all  
    pairs of vertices such that the shortest distances consider only the  
    vertices in set  $\{0, 1, 2, \dots, k-1\}$  as intermediate vertices.  
    ----> After the end of a iteration, vertex no.  $k$  is added to the set of  
    intermediate vertices and the set becomes  $\{0, 1, 2, \dots, k\}$  */  
    for (k = 0; k < V; k++)  
    {  
        /* Pick all vertices as source one by one  
        for (i = 0; i < V; i++)  
        {  
            /* Pick all vertices as destination for the  
            // above picked source  
            for (j = 0; j < V; j++)  
            {  
                /* If vertex k is on the shortest path from  
                // i to j, then update the value of dist[i][j]  
                if (dist[i][k] + dist[k][j] < dist[i][j])  
                    dist[i][j] = dist[i][k] + dist[k][j];  
            }  
        }  
  
        // Print the shortest distance matrix  
        printSolution(dist);  
    }  
}
```

```

/* A utility function to print solution */
void printSolution(int dist[][V])
{
    printf ("Following matrix shows the shortest distances
           "between every pair of vertices \n");
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf ("%7d", dist[i][j]);
        }
        printf("\n");
    }
}

// driver program to test above function
int main()
{
    /* Let us create the following weighted graph
       10
       (0)----->(3)
       |     /\
      5 |     |
       |     | 1
      \  |     |
       (1)----->(2)
       3     *
    int graph[V][V] = { {0, 5, INF, 10},
                        {INF, 0, 3, INF},
                        {INF, INF, 0, 1},
                        {INF, INF, INF, 0}
                      };

    // Print the solution
    floydWarshall(graph);
    return 0;
}

```

Java

```

// A Java program for Floyd Warshall All Pairs Shortest
// Path algorithm.
import java.util.*;
import java.lang.*;
import java.io.*;

class AllPairShortestPath
{
    final static int INF = 99999, V = 4;

    void floydWarshall(int graph[][])
    {
        int dist[][] = new int[V][V];
        int i, j, k;

        /* Initialize the solution matrix same as input graph matrix.
         Or we can say the initial values of shortest distances
         are based on shortest paths considering no intermediate
         vertex. */
        for (i = 0; i < V; i++)
            for (j = 0; j < V; j++)
                dist[i][j] = graph[i][j];

        /* Add all vertices one by one to the set of intermediate
         vertices.
         --> Before start of a iteration, we have shortest
             distances between all pairs of vertices such that
             the shortest distances consider only the vertices in
             set {0, 1, 2, .. k-1} as intermediate vertices.
         ----> After the end of a iteration, vertex no. k is added
             to the set of intermediate vertices and the set
             becomes {0, 1, 2, .. k} */
        for (k = 0; k < V; k++)
        {
            // Pick all vertices as source one by one
            for (i = 0; i < V; i++)
            {
                // Pick all vertices as destination for the
                // above picked source
                for (j = 0; j < V; j++)
                {
                    // If vertex k is on the shortest path from
                    // i to j, then update the value of dist[i][j]
                    if (dist[i][k] + dist[k][j] < dist[i][j])
                        dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }

        // Print the shortest distance matrix
        printSolution(dist);
    }

    void printSolution(int dist[])
    {
        System.out.println("Following matrix shows the shortest "+
                           "distances between every pair of vertices");
        for (int i=0; i<V; ++i)
        {

```

```

        for (int j=0; j<V; ++j)
    {
        if (dist[i][j]==INF)
            System.out.print("INF ");
        else
            System.out.print(dist[i][j]+ " ");
    }
    System.out.println();
}

// Driver program to test above function
public static void main (String[] args)
{
    /* Let us create the following weighted graph
     10
     (0)----->(3)
     |       \
     |       |
     5 |       |
     |       |
     \|       |
     (1)----->(2)
     3           *
    int graph[][] = { {0, 5, INF, 10},
                      {INF, 0, 3, INF},
                      {INF, INF, 0, 1},
                      {INF, INF, INF, 0}
                    };
    AllPairShortestPath a = new AllPairShortestPath();

    // Print the solution
    a.floydWarshall(graph);
}
}

// Contributed by Aakash Hasija

```

Python

```

# Python Program for Floyd Warshall Algorithm

# Number of vertices in the graph
V = 4

# Define infinity as the large enough value. This value will be
# used for vertices not connected to each other
INF = 99999

# Solves all pair shortest path via Floyd Warshall Algorithm
def floydWarshall(graph):

    """
    dist[][] will be the output matrix that will finally
    have the shortest distances between every pair of vertices
    """
    """
    initializing the solution matrix same as input graph matrix
    OR we can say that the initial values of shortest distances
    are based on shortest paths considering no
    intermediate vertices
    """
    dist = map(lambda i : map(lambda j : j , i) , graph)

    """
    Add all vertices one by one to the set of intermediate
    vertices.
    --> Before start of a iteration, we have shortest distances
    between all pairs of vertices such that the shortest
    distances consider only the vertices in set
    {0, 1, 2, .. k-1} as intermediate vertices.
    ----> After the end of a iteration, vertex no. k is
    added to the set of intermediate vertices and the
    set becomes {0, 1, 2, .. k}
    """

    for k in range(V):

        # pick all vertices as source one by one
        for i in range(V):

            # Pick all vertices as destination for the
            # above picked source
            for j in range(V):

                # If vertex k is on the shortest path from
                # i to j, then update the value of dist[i][j]
                dist[i][j] = min(dist[i][j],
                                  dist[i][k]+ dist[k][j])
    printSolution(dist)

    # A utility function to print the solution
def printSolution(dist):
    print "Following matrix shows the shortest distances\
between every pair of vertices"
    for i in range(V):
        for j in range(V):
            if(dist[i][j] == INF):
                print "%7s" %("INF"),
            else:
                print "%7d" %(dist[i][j]),
            if j == V-1:
                print ""

# Driver program to test the above program
# Let us create the following weighted graph

```

```

    10
(0)----->(3)
|     | \
5 |     | 1
|     |
(1)----->(2)
3
graph = [[0,5,INF,10],
          [INF,0,3,INF],
          [INF, INF, 0, 1],
          [INF, INF, INF, 0]
        ]
# Print the solution
floydWarshall(graph);
# This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

Following matrix shows the shortest distances between every pair of vertices
  0   5   8   9
0 INF 0   3   4
INF INF 0   1
INF INF INF 0

```

Time Complexity: $O(V^3)$

The above program only prints the shortest distances. We can modify the solution to print the shortest paths also by storing the predecessor information in a separate 2D matrix.

Also, the value of INF can be taken as INT_MAX from limits.h to make sure that we handle maximum possible value. When we take INF as INT_MAX, we need to change the if condition in the above program to avoid arithmetic overflow.

```

#include <limits.h>

#define INF INT_MAX
.....
if ( dist[i][k] != INF &&
    dist[k][j] != INF &&
    dist[i][k] + dist[k][j] < dist[i][j] )
    dist[i][j] = dist[i][k] + dist[k][j];
.....

```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Graph
shortest path

Dynamic Programming | Set 17 (Palindrome Partitioning)

Given a string, a partitioning of the string is a *palindrome partitioning* if every substring of the partition is a palindrome. For example, "aba|b|bbabb|a|b|aba" is a palindrome partitioning of "ababbabbababa". Determine the fewest cuts needed for palindrome partitioning of a given string. For example, minimum 3 cuts are needed for "ababbabbababa". The three cuts are "a|babbbab|bababa". If a string is palindrome, then minimum 0 cuts are needed. If a string of length n containing all different characters, then minimum n-1 cuts are needed.

palindrome-partitioning



Solution

This problem is a variation of Matrix Chain Multiplication problem. If the string is palindrome, then we simply return 0. Else, like the Matrix Chain Multiplication problem, we try making cuts at all possible places, recursively calculate the cost for each cut and return the minimum value.

Let the given string be str and minPalPartion() be the function that returns the fewest cuts needed for palindrome partitioning. following is the optimal substructure property.

```

// i is the starting index and j is the ending index. i must be passed as 0 and j as n-1
minPalPartion(str, i, j) = 0 if i == j. // When string is of length 1.
minPalPartion(str, i, j) = 0 if str[i..j] is palindrome.

// If none of the above conditions is true, then minPalPartion(str, i, j) can be
// calculated recursively using the following formula.
minPalPartion(str, i, j) = Min { minPalPartion(str, i, k) + 1 +
                                minPalPartion(str, k+1, j) }
                                where k varies from i to j-1

```

Following is Dynamic Programming solution. It stores the solutions to subproblems in two arrays P[][] and C[], and reuses the calculated values.

```

// Dynamic Programming Solution for Palindrome Partitioning Problem
#include <stdio.h>
#include <string.h>
#include <limits.h>

// A utility function to get minimum of two integers
int min (int a, int b) { return (a < b)? a : b; }

// Returns the minimum number of cuts needed to partition a string
// such that every part is a palindrome
int minPalPartion(char *str)
{
    // Get the length of the string
    int n = strlen(str);

    /* Create two arrays to build the solution in bottom up manner
       C[i][j] = Minimum number of cuts needed for palindrome partitioning
       of substring str[i..j]
       P[i][j] = true if substring str[i..j] is palindrome, else false
    */

```

```

Note that C[i][j] is 0 if P[i][j] is true */
int C[n][n];
bool P[n][n];

int i, j, k, L; // different looping variables

// Every substring of length 1 is a palindrome
for (i=0; i<n; i++)
{
    P[i][i] = true;
    C[i][i] = 0;
}

/* L is substring length. Build the solution in bottom up manner by
   considering all substrings of length starting from 2 to n.
   The loop structure is same as Matrix Chain Multiplication problem (
   See http://www.geeksforgeeks.org/archives/15553 )*/
for (L=2; L<=n; L++)
{
    // For substring of length L, set different possible starting indexes
    for (i=0; i<n-L+1; i++)
    {
        j = i+L-1; // Set ending index

        // If L is 2, then we just need to compare two characters. Else
        // need to check two corner characters and value of P[i+1][j]-1
        if (L == 2)
            P[i][j] = (str[i] == str[j]);
        else
            P[i][j] = (str[i] == str[j]) && P[i+1][j]-1;

        // If str[i..j] is palindrome, then C[i][j] is 0
        if (P[i][j] == true)
            C[i][j] = 0;
        else
        {
            // Make a cut at every possible location starting from i to j,
            // and get the minimum cost cut.
            C[i][j] = INT_MAX;
            for (k=i; k<=j-1; k++)
                C[i][j] = min (C[i][j], C[i][k] + C[k+1][j]+1);
        }
    }
}

// Return the min cut value for complete string. i.e., str[0..n-1]
return C[0][n-1];
}

// Driver program to test above function
int main()
{
    char str[] = "ababbbaabababa";
    printf("Min cuts needed for Palindrome Partitioning is %d",
           minPalPartition(str));
    return 0;
}

```

Output:

```
Min cuts needed for Palindrome Partitioning is 3
```

Time Complexity: O(n³)

An optimization to above approach

In above approach, we can calculating minimum cut while finding all palindromic substring. If we finding all palindromic substring 1st and then we calculate minimum cut, time complexity will reduce to O(n²). Thanks for Vivek for suggesting this optimization.

```

// Dynamic Programming Solution for Palindrome Partitioning Problem
#include <stdio.h>
#include <string.h>
#include <limits.h>

// A utility function to get minimum of two integers
int min (int a, int b) { return (a < b)? a : b; }

// Returns the minimum number of cuts needed to partition a string
// such that every part is a palindrome
int minPalPartition(char *str)
{
    // Get the length of the string
    int n = strlen(str);

    /* Create two arrays to build the solution in bottom up manner
       C[i] = Minimum number of cuts needed for palindrome partitioning
       of substring str[0..i]
       P[i][j] = true if substring str[i..j] is palindrome, else false
       Note that C[i] is 0 if P[0][i] is true */
    int C[n];
    bool P[n][n];

    int i, j, k, L; // different looping variables

    // Every substring of length 1 is a palindrome
    for (i=0; i<n; i++)
    {
        P[i][i] = true;
    }

    /* L is substring length. Build the solution in bottom up manner by
       considering all substrings of length starting from 2 to n. */
    for (L=2; L<=n; L++)
    {
        // For substring of length L, set different possible starting indexes
        for (i=0; i<n-L+1; i++)
        {

```

```

j = i+L-1; // Set ending index

// If L is 2, then we just need to compare two characters. Else
// need to check two corner characters and value of P[i+1][j]-1
if (L == 2)
    P[i][j] = (str[i] == str[j]);
else
    P[i][j] = (str[i] == str[j]) && P[i+1][j-1];
}

for (i=0; i<n; i++)
{
if (P[0][i] == true)
    C[i] = 0;
else
{
    C[i] = INT_MAX;
    for(j=0;j<i;j++)
    {
        if(P[j+1][i] == true && 1+C[j]<C[i])
            C[i]=1+C[j];
    }
}
}

// Return the min cut value for complete string, i.e., str[0..n-1]
return C[n-1];
}

// Driver program to test above function
int main()
{
char str[] = "ababbabbababa";
printf("Min cuts needed for Palindrome Partitioning is %d",
       minPalPartition(str));
return 0;
}

```

Output:

Min cuts needed for Palindrome Partitioning is 3

Time Complexity: $O(n^2)$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming
palindrome

Dynamic Programming | Set 18 (Partition problem)

Partition problem is to determine whether a given set can be partitioned into two subsets such that the sum of elements in both subsets is same.

Examples

```

arr[] = {1, 5, 11, 5}
Output: true
The array can be partitioned as {1, 5, 5} and {11}

arr[] = {1, 5, 3}
Output: false
The array cannot be partitioned into equal sum sets.

```

We strongly recommend that you click here and practice it, before moving on to the solution.

Following are the two main steps to solve this problem:

- 1) Calculate sum of the array. If sum is odd, there can not be two subsets with equal sum, so return false.
- 2) If sum of array elements is even, calculate sum/2 and find a subset of array with sum equal to sum/2.

The first step is simple. The second step is crucial, it can be solved either using recursion or Dynamic Programming.

Recursive Solution

Following is the recursive property of the second step mentioned above.

```

Let isSubsetSum(arr, n, sum/2) be the function that returns true if
there is a subset of arr[0..n-1] with sum equal to sum/2

The isSubsetSum problem can be divided into two subproblems
a) isSubsetSum() without considering last element
   (reducing n to n-1)
b) isSubsetSum considering the last element
   (reducing sum/2 by arr[n-1] and n to n-1)
If any of the above the above subproblems return true, then return true.
isSubsetSum (arr, n, sum/2) = isSubsetSum (arr, n-1, sum/2) ||
                           isSubsetSum (arr, n-1, sum/2 - arr[n-1])

```

C/C++

```

// A recursive C program for partition problem
#include <stdio.h>

// A utility function that returns true if there is
// a subset of arr[] with sun equal to given sum

```

```

bool isSubsetSum (int arr[], int n, int sum)
{
    // Base Cases
    if (sum == 0)
        return true;
    if (n == 0 && sum != 0)
        return false;

    // If last element is greater than sum, then
    // ignore it
    if (arr[n-1] > sum)
        return isSubsetSum (arr, n-1, sum);

    /* else, check if sum can be obtained by any of
       the following
       (a) including the last element
       (b) excluding the last element
    */
    return isSubsetSum (arr, n-1, sum) ||
           isSubsetSum (arr, n-1, sum-arr[n-1]);
}

// Returns true if arr[] can be partitioned in two
// subsets of equal sum, otherwise false
bool findPartition (int arr[], int n)
{
    // Calculate sum of the elements in array
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum += arr[i];

    // If sum is odd, there cannot be two subsets
    // with equal sum
    if (sum%2 != 0)
        return false;

    // Find if there is subset with sum equal to
    // half of total sum
    return isSubsetSum (arr, n, sum/2);
}

// Driver program to test above function
int main()
{
    int arr[] = {3, 1, 5, 9, 12};
    int n = sizeof(arr)/sizeof(arr[0]);
    if (findPartition(arr, n) == true)
        printf("Can be divided into two subsets "
               "of equal sum");
    else
        printf("Can not be divided into two subsets"
               " of equal sum");
    return 0;
}

```

Java

```

// A recursive Java solution for partition problem
import java.io.*;

class Partition
{
    // A utility function that returns true if there is a
    // subset of arr[] with sun equal to given sum
    static boolean isSubsetSum (int arr[], int n, int sum)
    {
        // Base Cases
        if (sum == 0)
            return true;
        if (n == 0 && sum != 0)
            return false;

        // If last element is greater than sum, then ignore it
        if (arr[n-1] > sum)
            return isSubsetSum (arr, n-1, sum);

        /* else, check if sum can be obtained by any of
           the following
           (a) including the last element
           (b) excluding the last element
        */
        return isSubsetSum (arr, n-1, sum) ||
               isSubsetSum (arr, n-1, sum-arr[n-1]);
    }

    // Returns true if arr[] can be partitioned in two
    // subsets of equal sum, otherwise false
    static boolean findPartition (int arr[], int n)
    {
        // Calculate sum of the elements in array
        int sum = 0;
        for (int i = 0; i < n; i++)
            sum += arr[i];

        // If sum is odd, there cannot be two subsets
        // with equal sum
        if (sum%2 != 0)
            return false;

        // Find if there is subset with sum equal to half
        // of total sum
        return isSubsetSum (arr, n, sum/2);
    }
}

```

```

/*Driver function to check for above function*/
public static void main (String[] args)
{
    int arr[] = {3, 1, 5, 9, 12};
    int n = arr.length;
    if (findPartition(arr, n) == true)
        System.out.println("Can be divided into two "+
                           "subsets of equal sum");
    else
        System.out.println("Can not be divided into "+
                           "two subsets of equal sum");
}
/* This code is contributed by Devesh Agrawal */

```

Output:

Can be divided into two subsets of equal sum

Time Complexity: $O(2^n)$ In worst case, this solution tries two possibilities (whether to include or exclude) for every element.

Dynamic Programming Solution

The problem can be solved using dynamic programming when the sum of the elements is not too big. We can create a 2D array $\text{part}[][]$ of size $(\text{sum}/2) \times (n+1)$. And we can construct the solution in bottom up manner such that every filled entry has following property

$\text{part}[i][j] = \text{true}$ if a subset of $\{\text{arr}[0], \text{arr}[1], \dots, \text{arr}[j-1]\}$ has sum equal to i , otherwise false

C/C++

```

// A Dynamic Programming based C program to partition problem
#include <stdio.h>

// Returns true if arr[] can be partitioned in two subsets of
// equal sum, otherwise false
bool findPartition (int arr[], int n)
{
    int sum = 0;
    int i, j;

    // Calculate sum of all elements
    for (i = 0; i < n; i++)
        sum += arr[i];

    if (sum%2 != 0)
        return false;

    bool part[sum/2+1][n+1];

    // Initialize top row as true
    for (i = 0; i <= n; i++)
        part[0][i] = true;

    // Initialize leftmost column, except part[0][0], as 0
    for (i = 1; i <= sum/2; i++)
        part[i][0] = false;

    // Fill the partition table in bottom up manner
    for (i = 1; i <= sum/2; i++)
    {
        for (j = 1; j <= n; j++)
        {
            part[i][j] = part[i][j-1];
            if (i >= arr[j-1])
                part[i][j] = part[i][j] || part[i - arr[j-1]][j-1];
        }
    }

    /* // uncomment this part to print table
    for (i = 0; i <= sum/2; i++)
    {
        for (j = 0; j <= n; j++)
            printf("%4d", part[i][j]);
        printf("\n");
    } */

    return part[sum/2][n];
}

// Driver program to test above function
int main()
{
    int arr[] = {3, 1, 1, 2, 2, 1};
    int n = sizeof(arr)/sizeof(arr[0]);
    if (findPartition(arr, n) == true)
        printf("Can be divided into two subsets of equal sum");
    else
        printf("Can not be divided into two subsets of equal sum");
    getchar();
    return 0;
}

```

Java

```

// A dynamic programming based Java program for partition problem
import java.io.*;

class Partition {

```

Output:

Can be divided into two subsets of equal sum

Following diagram shows the values in partition table. The diagram is taken from the wiki page of partition problem

Time Complexity: $O(\text{sum}^n)$

Auxiliary Space: $O(\text{sum}^*n)$

Please note that this solution will not be feasible for arrays with big sum.

References:

http://en.wikipedia.org/wiki/Partition_problem

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming | Set 19 (Word Wrap Problem)

Given a sequence of words, and a limit on the number of characters that can be put in one line (line width). Put line breaks in the given sequence such that the lines are printed neatly. Assume that the length of each word is smaller than the line width.

The word processors like MS Word do task of placing line breaks. The idea is to have balanced lines. In other words, not have few lines with lots of extra spaces and some lines with small amount of extra spaces.

The extra spaces includes spaces put at the end of every line except the last one.

The problem is to minimize the following total cost.

Cost of a line = (Number of extra spaces in the line)³

Total Cost = Sum of costs for all lines

For example, consider the following string and line width M = 15

"Geeks for Geeks presents word wrap problem"

Following is the optimized arrangement of words in 3 lines

Geeks for Geeks

presents word

wrap problem

The total extra spaces in line 1, line 2 and line 3 are 0, 2 and 3 respectively.

So optimal value of total cost is $0 + 2^2 + 3^3 = 13$

Please note that the total cost function is not sum of extra spaces, but sum of cubes (or square is also used) of extra spaces. The idea behind this cost function is to balance the spaces among lines. For example, consider the following two arrangement of same set of words:

1) There are 3 lines. One line has 3 extra spaces and all other lines have 0 extra spaces. Total extra spaces = $3 + 0 + 0 = 3$. Total cost = $3^3 + 0^0 + 0^0 = 27$.

2) There are 3 lines. Each of the 3 lines has one extra space. Total extra spaces = $1 + 1 + 1 = 3$. Total cost = $1^1 + 1^1 + 1^1 = 3$.

Total extra spaces are 3 in both scenarios, but second arrangement should be preferred because extra spaces are balanced in all three lines. The cost function with cubic sum serves the purpose because the value of total cost in second scenario is less.

Method 1 (Greedy Solution)

The greedy solution is to place as many words as possible in the first line. Then do the same thing for the second line and so on until all words are placed. This solution gives optimal solution for many cases, but doesn't give optimal solution in all cases. For example, consider the following string "aaa bb cc dddd" and line width as 6. Greedy method will produce following output.

```
aaa bb  
cc  
ddddd
```

Extra spaces in the above 3 lines are 0, 4 and 1 respectively. So total cost is $0 + 64 + 1 = 65$.

But the above solution is not the best solution. Following arrangement has more balanced spaces. Therefore less value of total cost function.

```
aaa  
bb cc  
ddddd
```

Extra spaces in the above 3 lines are 3, 1 and 1 respectively. So total cost is $27 + 1 + 1 = 29$.

Despite being sub-optimal in some cases, the greedy approach is used by many word processors like MS Word and OpenOffice.org Writer.

Method 2 (Dynamic Programming)

The following Dynamic approach strictly follows the algorithm given in solution of Cormen book. First we compute costs of all possible lines in a 2D table $lc[i][j]$. The value $lc[i][j]$ indicates the cost to put words from i to j in a single line where i and j are indexes of words in the input sequences. If a sequence of words from i to j cannot fit in a single line, then $lc[i][j]$ is considered infinite (to avoid it from being a part of the solution). Once we have the $lc[i][j]$ table constructed, we can calculate total cost using following recursive formula. In the following formula, $C[i][j]$ is the optimized total cost for arranging words from 1 to j.



The above recursion has [overlapping subproblem property](#). For example, the solution of subproblem $c(2)$ is used by $c(3)$, $C(4)$ and so on. So Dynamic Programming is used to store the results of subproblems. The array $c[]$ can be computed from left to right, since each value depends only on earlier values.

To print the output, we keep track of what words go on what lines, we can keep a parallel p array that points to where each c value came from. The last line starts at word $p[n]$ and goes through word n . The previous line starts at word $p[n-1]$ and goes through word $p[n-1]$, etc. The function $printSolution()$ uses $p[]$ to print the solution.

In the below program, input is an array $l[]$ that represents lengths of words in a sequence. The value $l[i]$ indicates length of the i th word (i starts from 1) in the input sequence.

```
// A Dynamic programming solution for Word Wrap Problem  
#include <limits.h>  
#include <stdio.h>  
#define INF INT_MAX  
  
// A utility function to print the solution  
int printSolution (int p[], int n);  
  
// l[] represents lengths of different words in input sequence. For example,  
// l[] = {3, 2, 2, 5} is for a sentence like "aaa bb cc dddd". n is size of  
// l[] and M is line width (maximum no. of characters that can fit in a line)  
void solveWordWrap (int l[], int n, int M)  
{  
    // For simplicity, 1 extra space is used in all below arrays  
  
    // extras[i][j] will have number of extra spaces if words from i  
    // to j are put in a single line  
    int extras[n+1][n+1];  
  
    // lc[i][j] will have cost of a line which has words from i  
    // to j  
    int lc[n+1][n+1];  
  
    // c[i] will have total cost of optimal arrangement of words  
    // from 1 to i  
    int c[n+1];  
  
    // p[] is used to print the solution.  
    int p[n+1];  
  
    int i, j;
```

```

// calculate extra spaces in a single line. The value extra[i][j]
// indicates extra spaces if words from word number i to j are
// placed in a single line
for (i = 1; i <= n; i++)
{
    extras[i][0] = M - l[i-1];
    for (j = i+1; j <= n; j++)
        extras[i][j] = extras[i][j-1] - l[j-1] - 1;
}

// Calculate line cost corresponding to the above calculated extra
// spaces. The value lc[i][j] indicates cost of putting words from
// word number i to j in a single line
for (i = 1; i <= n; i++)
{
    for (j = i; j <= n; j++)
    {
        if (extras[i][j] < 0)
            lc[i][j] = INF;
        else if (j == n && extras[i][j] >= 0)
            lc[i][j] = 0;
        else
            lc[i][j] = extras[i][j]*extras[i][j];
    }
}

// Calculate minimum cost and find minimum cost arrangement.
// The value c[i][j] indicates optimized cost to arrange words
// from word number 1 to j.
c[0] = 0;
for (j = 1; j <= n; j++)
{
    c[j] = INF;
    for (i = 1; i <= j; i++)
    {
        if (c[i-1] != INF & & lc[i][j] != INF && (c[i-1] + lc[i][j] < c[j]))
        {
            c[j] = c[i-1] + lc[i][j];
            p[j] = i;
        }
    }
}

printSolution(p, n);
}

int printSolution (int p[], int n)
{
    int k;
    if (p[n] == 1)
        k = 1;
    else
        k = printSolution (p, p[n]-1) + 1;
    printf ("Line number %d: From word no. %d to %d \n", k, p[n], n);
    return k;
}

// Driver program to test above functions
int main()
{
    int l[] = {3, 2, 2, 5};
    int n = sizeof(l)/sizeof(l[0]);
    int M = 6;
    solveWordWrap (l, n, M);
    return 0;
}

```

Output:

```

Line number 1: From word no. 1 to 1
Line number 2: From word no. 2 to 3
Line number 3: From word no. 4 to 4

```

Time Complexity: $O(n^2)$

Auxiliary Space: $O(n^2)$ The auxiliary space used in the above program can be optimized to $O(n)$ (See the reference 2 for details)

References:

http://en.wikipedia.org/wiki/Word_wrap

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming

Dynamic Programming | Set 20 (Maximum Length Chain of Pairs)

You are given n pairs of numbers. In every pair, the first number is always smaller than the second number. A pair (c, d) can follow another pair (a, b) if b Amazon Interview | Set 2

For example, if the given pairs are $\{(5, 24), (39, 60), (15, 28), (27, 40), (50, 90)\}$, then the longest chain that can be formed is of length 3, and the chain is $\{(5, 24), (27, 40), (50, 90)\}$

This problem is a variation of standard Longest Increasing Subsequence problem. Following is a simple two step process.

1) Sort given pairs in increasing order of first (or smaller) element.

2) Now run a modified LIS process where we compare the second element of already finalized LIS with the first element of new LIS being constructed.

The following code is a slight modification of method 2 of [this post](#).

```

#include<stdio.h>
#include<stdlib.h>

// Structure for a pair
struct pair
{

```

```

int a;
int b;
};

// This function assumes that arr[] is sorted in increasing order
// according the first (or smaller) values in pairs.
int maxChainLength( struct pair arr[], int n )
{
    int i, j, max = 0;
    int *mcl = (int*) malloc ( sizeof( int ) * n );

    /* Initialize MCL (max chain length) values for all indexes */
    for ( i = 0; i < n; i++ )
        mcl[i] = 1;

    /* Compute optimized chain length values in bottom up manner */
    for ( i = 1; i < n; i++ )
        for ( j = 0; j < i; j++ )
            if ( arr[i].a > arr[j].b && mcl[i] < mcl[j] + 1 )
                mcl[i] = mcl[j] + 1;

    // mcl[i] now stores the maximum chain length ending with pair i

    /* Pick maximum of all MCL values */
    for ( i = 0; i < n; i++ )
        if ( max < mcl[i] )
            max = mcl[i];

    /* Free memory to avoid memory leak */
    free( mcl );
}

return max;
}

/* Driver program to test above function */
int main()
{
    struct pair arr[] = { { 5, 24}, {15, 25},
                         {27, 40}, {50, 60} };
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Length of maximum size chain is %d\n",
           maxChainLength( arr, n ));
    return 0;
}

```

Output:

Length of maximum size chain is 3

Time Complexity: $O(n^2)$ where n is the number of pairs.

The given problem is also a variation of [Activity Selection problem](#) and can be solved in $(n\log n)$ time. To solve it as a activity selection problem, consider the first element of a pair as start time in activity selection problem, and the second element of pair as end time. Thanks to Palash for suggesting this approach.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming

Dynamic Programming | Set 21 (Variations of LIS)

We have discussed Dynamic Programming solution for Longest Increasing Subsequence problem in [this](#) post and a $O(n\log n)$ solution in [this](#) post. Following are commonly asked variations of the standard LIS problem.

1. Building Bridges: Consider a 2-D map with a horizontal river passing through its center. There are n cities on the southern bank with x -coordinates $a(1) \dots a(n)$ and n cities on the northern bank with x -coordinates $b(1) \dots b(n)$. You want to connect as many north-south pairs of cities as possible with bridges such that no two bridges cross. When connecting cities, you can only connect city i on the northern bank to city i on the southern bank.

```

8   1   4   3   5   2   6   7
-----
-----
1   2   3   4   5   6   7   8

```

Source: [Dynamic Programming Practice Problems](#). The link also has well explained solution for the problem.

2. Maximum Sum Increasing Subsequence: Given an array of n positive integers. Write a program to find the maximum sum subsequence of the given array such that the integers in the subsequence are sorted in increasing order. For example, if input is $\{1, 101, 2, 3, 100, 4, 5\}$, then output should be $\{1, 2, 3, 100\}$. The solution to this problem has been published [here](#).

3. The Longest Chain You are given pairs of numbers. In a pair, the first number is smaller with respect to the second number. Suppose you have two sets (a, b) and (c, d) , the second set can follow the first set if b here.

4. Box Stacking You are given a set of n types of rectangular 3-D boxes, where the i^{th} box has height $h(i)$, width $w(i)$ and depth $d(i)$ (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.

Source: [Dynamic Programming Practice Problems](#). The link also has well explained solution for the problem.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming

Dynamic Programming | Set 22 (Box Stacking Problem)

You are given a set of n types of rectangular 3-D boxes, where the i^{th} box has height $h(i)$, width $w(i)$ and depth $d(i)$ (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.

Source: <http://people.csail.mit.edu/bdean/6.046/dp/>. The link also has video for explanation of solution.



The Box Stacking problem is a variation of LIS problem. We need to build a maximum height stack.

Following are the key points to note in the problem statement:

- 1) A box can be placed on top of another box only if both width and depth of the upper placed box are smaller than width and depth of the lower box respectively.
- 2) We can rotate boxes. For example, if there is a box with dimensions {1x2x3} where 1 is height, 2x3 is base, then there can be three possibilities, {1x2x3}, {2x1x3} and {3x1x2}.
- 3) We can use multiple instances of boxes. What it means is, we can have two different rotations of a box as part of our maximum height stack.

Following is the **solution** based on DP solution of LIS problem.

1) Generate all 3 rotations of all boxes. The size of rotation array becomes 3 times the size of original array. For simplicity, we consider depth as always smaller than or equal to width.

2) Sort the above generated $3n$ boxes in decreasing order of base area.

3) After sorting the boxes, the problem is same as LIS with following optimal substructure property.

$MSH(i)$ = Maximum possible Stack Height with box i at top of stack

$MSH(i) = \max(MSH(j) + \text{height}(i))$ where $j < i$ and $\text{width}(j) > \text{width}(i)$ and $\text{depth}(j) > \text{depth}(i)$.

If there is no such j then $MSH(i) = \text{height}(i)$

4) To get overall maximum height, we return $\max(MSH(i))$ where $0 < i < n$

Following is C++ implementation of the above solution.

```
/* Dynamic Programming implementation of Box Stacking problem */
#include<stdio.h>
#include<stdlib.h>

/* Representation of a box */
struct Box
{
    // h --> height, w --> width, d --> depth
    int h, w, d; // for simplicity of solution, always keep w <= d
};

// A utility function to get minimum of two integers
int min (int x, int y)
{ return (x < y)? x : y; }

// A utility function to get maximum of two integers
int max (int x, int y)
{ return (x > y)? x : y; }

/* Following function is needed for library function qsort(). We
use qsort() to sort boxes in decreasing order of base area.
Refer following link for help of qsort() and compare()
http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ */
int compare (const void *a, const void * b)
{
    return ((*(Box *)b).d * (*(Box *)b).w ) -
        ((*(Box *)a).d * (*(Box *)a).w );
}

/* Returns the height of the tallest stack that can be
formed with give type of boxes */
int maxStackHeight( Box arr[], int n )
{
    /* Create an array of all rotations of given boxes
    For example, for a box {1, 2, 3}, we consider three
    instances{{1, 2, 3}, {2, 1, 3}, {3, 1, 2}} */
    Box rot[3*n];
    int index = 0;
    for (int i = 0; i < n; i++)
    {
        // Copy the original box
        rot[index] = arr[i];
        index++;
    }

    // First rotation of box
    rot[index].h = arr[i].w;
    rot[index].d = max(arr[i].h, arr[i].d);
    rot[index].w = min(arr[i].h, arr[i].d);
    index++;

    // Second rotation of box
    rot[index].h = arr[i].d;
    rot[index].d = max(arr[i].h, arr[i].w);
    rot[index].w = min(arr[i].h, arr[i].w);
    index++;
}

// Now the number of boxes is 3n
n = 3*n;

/* Sort the array 'rot[]' in decreasing order, using library
function for quick sort */
```

```

qsort(rot, n, sizeof(rot[0]), compare);

// Uncomment following two lines to print all rotations
// for (int i = 0; i < n; i++)
//   printf("%d x %d x %d\n", rot[i].h, rot[i].w, rot[i].d);

/* Initialize msh values for all indexes
msh[i] --> Maximum possible Stack Height with box i on top */
int msh[n];
for (int i = 0; i < n; i++)
  msh[i] = rot[i].h;

/* Compute optimized msh values in bottom up manner */
for (int i = 1; i < n; i++)
  for (int j = 0; j < i; j++)
    if ( rot[j].w < rot[i].w &&
        rot[j].d < rot[i].d &&
        msh[i] < msh[j] + rot[i].h)
      {
        msh[i] = msh[j] + rot[i].h;
      }

/* Pick maximum of all msh values */
int max = -1;
for ( int i = 0; i < n; i++)
  if ( max < msh[i] )
    max = msh[i];

return max;
}

/* Driver program to test above function */
int main()
{
  Box arr[] = { {4, 6, 7}, {1, 2, 3}, {4, 5, 6}, {10, 12, 32} };
  int n = sizeof(arr)/sizeof(arr[0]);

  printf("The maximum possible height of stack is %d\n",
         maxStackHeight (arr, n) );

  return 0;
}

```

Output:

The maximum possible height of stack is 60

In the above program, given input boxes are {4, 6, 7}, {1, 2, 3}, {4, 5, 6}, {10, 12, 32}. Following are all rotations of the boxes in decreasing order of base area.

```

10 x 12 x 32
12 x 10 x 32
32 x 10 x 12
4 x 6 x 7
4 x 5 x 6
6 x 4 x 7
5 x 4 x 6
7 x 4 x 6
6 x 4 x 5
1 x 2 x 3
2 x 1 x 3
3 x 1 x 2

```

The height 60 is obtained by boxes { {3, 1, 2}, {1, 2, 3}, {6, 4, 5}, {4, 5, 6}, {4, 6, 7}, {32, 10, 12}, {10, 12, 32} }

Time Complexity: O(n^2)

Auxiliary Space: O(n)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming

Program for Fibonacci numbers

The Fibonacci numbers are the numbers in the following integer sequence.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,

In mathematical terms, the sequence F_n of Fibonacci numbers is defined by the recurrence relation

$$F_n = F_{n-1} + F_{n-2}$$

with seed values

$$F_0 = 0 \text{ and } F_1 = 1.$$

Write a function `int fib(int n)` that returns F_n. For example, if n = 0, then `fib()` should return 0. If n = 1, then it should return 1. For n > 1, it should return F_{n-1} + F_{n-2}

For n = 9
Output:34

Following are different methods to get the nth Fibonacci number.

Method 1 (Use recursion)

A simple method that is a direct recursive implementation mathematical recurrence relation given above.

```
//Fibonacci Series using Recursion
#include<stdio.h>
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}

int main ()
{
    int n = 9;
    printf("%d", fib(n));
    getchar();
    return 0;
}
```

Java

```
//Fibonacci Series using Recursion
class fibonacci
{
    static int fib(int n)
    {
        if (n <= 1)
            return n;
        return fib(n-1) + fib(n-2);
    }

    public static void main (String args[])
    {
        int n = 9;
        System.out.println(fib(n));
    }
}
/* This code is contributed by Rajat Mishra */
```

Python

```
# Function for nth Fibonacci number

def Fibonacci(n):
    if n<0:
        print("Incorrect input")
    # First Fibonacci number is 0
    elif n==1:
        return 0
    # Second Fibonacci number is 1
    elif n==2:
        return 1
    else:
        return Fibonacci(n-1)+Fibonacci(n-2)

# Driver Program

print(Fibonacci(9))

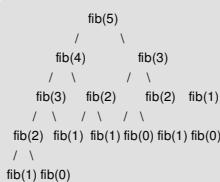
#This code is contributed by Saket Modi
```

Output

34

Time Complexity: $T(n) = T(n-1) + T(n-2)$ which is exponential.

We can observe that this implementation does a lot of repeated work (see the following recursion tree). So this is a bad implementation for nth Fibonacci number.



Extra Space: $O(n)$ if we consider the function call stack size, otherwise $O(1)$.

Method 2 (Use Dynamic Programming)

We can avoid the repeated work done in the method 1 by storing the Fibonacci numbers calculated so far.

C

```
//Fibonacci Series using Dynamic Programming
#include<stdio.h>

int fib(int n)
{
    /* Declare an array to store Fibonacci numbers. */
    int f[n+1];
    int i;

    /* 0th and 1st number of the series are 0 and 1*/
    f[0] = 0;
    f[1] = 1;

    for (i = 2;i <= n;i++)
    {
```

```

/* Add the previous 2 numbers in the series
and store it */
f[i] = f[i-1] + f[i-2];
}

return f[n];
}

int main ()
{
int n = 9;
printf("%d", fib(n));
getchar();
return 0;
}

```

Java

```

// Fibonacci Series using Dynamic Programming
class fibonacci
{
    static int fib(int n)
    {
        /* Declare an array to store Fibonacci numbers.*/
        int f[] = new int[n+1];
        int i;

        /* 0th and 1st number of the series are 0 and 1*/
        f[0] = 0;
        f[1] = 1;

        for (i = 2; i <= n; i++)
        {
            /* Add the previous 2 numbers in the series
            and store it */
            f[i] = f[i-1] + f[i-2];
        }

        return f[n];
    }

    public static void main (String args[])
    {
        int n = 9;
        System.out.println(fib(n));
    }
}
/* This code is contributed by Rajat Mishra */

```

Python

```

# Function for nth fibonacci number - Dynamic Programming
# Taking 1st two fibonacci numbers as 0 and 1

FibArray = [0,1]

def fibonacci(n):
    if n<0:
        print("Incorrect input")
    elif n<=len(FibArray):
        return FibArray[n-1]
    else:
        temp_fib = fibonacci(n-1)+fibonacci(n-2)
        FibArray.append(temp_fib)
        return temp_fib

# Driver Program

print(fibonacci(9))

#This code is contributed by Saket Modi

```

Output:

34

Time Complexity: O(n)
Extra Space: O(n)

Method 3 (Space Optimized Method 2)

We can optimize the space used in method 2 by storing the previous two numbers only because that is all we need to get the next Fibonacci number in series.

C/C++

```

// Fibonacci Series using Space Optimized Method
#include<stdio.h>
int fib(int n)
{
int a = 0, b = 1, c, i;
if( n == 0)
    return a;
for (i = 2; i <= n; i++)
{
    c = a + b;
    a = b;
    b = c;
}
return b;
}

```

```

int main ()
{
    int n = 9;
    printf("%d", fib(n));
    getchar();
    return 0;
}

```

Java

```

// Java program for Fibonacci Series using Space
// Optimized Method
class fibonacci
{
    static int fib(int n)
    {
        int a = 0, b = 1, c;
        if (n == 0)
            return a;
        for (int i = 2; i <= n; i++)
        {
            c = a + b;
            a = b;
            b = c;
        }
        return b;
    }

    public static void main (String args[])
    {
        int n = 9;
        System.out.println(fib(n));
    }
}

// This code is contributed by Mihir Joshi

```

Python

```

# Function for nth fibonacci number - Space Optimisataion
# Taking 1st two fibonacci numbers as 0 and 1

def fibonacci(n):
    a = 0
    b = 1
    if n < 0:
        print("Incorrect input")
    elif n == 0:
        return a
    elif n == 1:
        return b
    else:
        for i in range(2,n):
            c = a + b
            a = b
            b = c
        return b

# Driver Program

print(fibonacci(9))

#This code is contributed by Saket Modi

```

Time Complexity: O(n)

Extra Space: O(1)

Method 4 (Using power of the matrix {{1,1},{1,0}})

This another O(n) which relies on the fact that if we n times multiply the matrix $M = \begin{Bmatrix} 1 & 1 \\ 1 & 0 \end{Bmatrix}$ to itself (in other words calculate $\text{power}(M, n)$), then we get the $(n+1)$ th Fibonacci number as the element at row and column (0, 0) in the resultant matrix.

The matrix representation gives the following closed expression for the Fibonacci numbers:



C

```

#include <stdio.h>

/* Helper function that multiplies 2 matrices F and M of size 2*2, and
   puts the multiplication result back to F[][]
   void multiply(int F[2][2], int M[2][2]);

/* Helper function that calculates F[][] raise to the power n and puts the
   result in F[][]
   Note that this function is designed only for fib() and won't work as general
   power function */
void power(int F[2][2], int n);

int fib(int n)
{
    int F[2][2] = {{1,1},{1,0}};
    if (n == 0)
        return 0;
    power(F, n-1);

    return F[0][0];
}

```

```

void multiply(int F[2][2], int M[2][2])
{
    int x = F[0][0]*M[0][0] + F[0][1]*M[1][0];
    int y = F[0][0]*M[0][1] + F[0][1]*M[1][1];
    int z = F[1][0]*M[0][0] + F[1][1]*M[1][0];
    int w = F[1][0]*M[0][1] + F[1][1]*M[1][1];

    F[0][0] = x;
    F[0][1] = y;
    F[1][0] = z;
    F[1][1] = w;
}

void power(int F[2][2], int n)
{
    int i;
    int M[2][2] = {{1,1},{1,0}};

    // n - 1 times multiply the matrix to {{1,0},{0,1}}
    for (i = 2; i <= n; i++)
        multiply(F, M);
}

/* Driver program to test above function */
int main()
{
    int n = 9;
    printf("%d", fib(n));
    getchar();
    return 0;
}

```

Java

```

class fibonacci
{
    static int fib(int n)
    {
        int F[] = new int[]{1,1,1,0};
        if (n == 0)
            return 0;
        power(F, n-1);

        return F[0];
    }

    /* Helper function that multiplies 2 matrices F and M of size 2*2, and
     * puts the multiplication result back to F[]
     */
    static void multiply(int F[], int M[])
    {
        int x = F[0][0]*M[0][0] + F[0][1]*M[1][0];
        int y = F[0][0]*M[0][1] + F[0][1]*M[1][1];
        int z = F[1][0]*M[0][0] + F[1][1]*M[1][0];
        int w = F[1][0]*M[0][1] + F[1][1]*M[1][1];

        F[0][0] = x;
        F[0][1] = y;
        F[1][0] = z;
        F[1][1] = w;
    }

    /* Helper function that calculates F[] raise to the power n and puts the
     * result in F[]
     * Note that this function is designed only for fib() and won't work as general
     * power function */
    static void power(int F[], int n)
    {
        int i;
        int M[] = new int[]{1,1,1,0};

        // n - 1 times multiply the matrix to {{1,0},{0,1}}
        for (i = 2; i <= n; i++)
            multiply(F, M);
    }

    /* Driver program to test above function */
    public static void main (String args[])
    {
        int n = 9;
        System.out.println(fib(n));
    }
}
/* This code is contributed by Rajat Mishra */

```

Time Complexity: O(n)

Extra Space: O(1)

Method 5 (Optimized Method 4)

The method 4 can be optimized to work in O(Logn) time complexity. We can do recursive multiplication to get power(M, n) in the previous method (Similar to the optimization done in [this post](#))

C

```

#include <stdio.h>

void multiply(int F[2][2], int M[2][2]);

void power(int F[2][2], int n);

```

```

/* function that returns nth Fibonacci number */
int fib(int n)
{
    int F[2][2] = {{1,1},{1,0}};
    if (n == 0)
        return 0;
    power(F, n-1);
    return F[0][0];
}

/* Optimized version of power() in method 4 */
void power(int F[2][2], int n)
{
    if( n == 0 || n == 1)
        return;
    int M[2][2] = {{1,1},{1,0}};

    power(F, n/2);
    multiply(F, M);

    if (n%2 != 0)
        multiply(F, M);
}

void multiply(int F[2][2], int M[2][2])
{
    int x = F[0][0]*M[0][0] + F[0][1]*M[1][0];
    int y = F[0][0]*M[0][1] + F[0][1]*M[1][1];
    int z = F[1][0]*M[0][0] + F[1][1]*M[1][0];
    int w = F[1][0]*M[0][1] + F[1][1]*M[1][1];

    F[0][0] = x;
    F[0][1] = y;
    F[1][0] = z;
    F[1][1] = w;
}

/* Driver program to test above function */
int main()
{
    int n = 9;
    printf("%d", fib(9));
    getchar();
    return 0;
}

```

Java

```

//Fibonacci Series using Optimized Method
class fibonacci
{
    /* function that returns nth Fibonacci number */
    static int fib(int n)
    {
        int F[] = new int[2][2]{{1,1},{1,0}};
        if (n == 0)
            return 0;
        power(F, n-1);

        return F[0][0];
    }

    static void multiply(int F[], int M[])
    {
        int x = F[0][0]*M[0][0] + F[0][1]*M[1][0];
        int y = F[0][0]*M[0][1] + F[0][1]*M[1][1];
        int z = F[1][0]*M[0][0] + F[1][1]*M[1][0];
        int w = F[1][0]*M[0][1] + F[1][1]*M[1][1];

        F[0][0] = x;
        F[0][1] = y;
        F[1][0] = z;
        F[1][1] = w;
    }

    /* Optimized version of power() in method 4 */
    static void power(int F[], int n)
    {
        if( n == 0 || n == 1)
            return;
        int M[] = new int[2][2]{{1,1},{1,0}};

        power(F, n/2);
        multiply(F, M);

        if (n%2 != 0)
            multiply(F, M);
    }

    /* Driver program to test above function */
    public static void main (String args[])
    {
        int n = 9;
        System.out.println(fib(n));
    }
}
/* This code is contributed by Rajat Mishra */

```

Time Complexity: O(Logn)

Extra Space: O(Logn) if we consider the function call stack size, otherwise O(1).

Method 6 (O(Log n) Time)

Below is one more interesting recurrence formula that can be used to find n'th Fibonacci Number in O(Log n) time.

If n is even then k = n/2

$F(n) = [2 \cdot F(k-1) + F(k)] \cdot F(k)$

If n is odd then k = (n + 1)/2

$F(n) = F(k) \cdot F(k) + F(k-1) \cdot F(k-1)$

How does this formula work?

The formula can be derived from above matrix equation.



Taking determinant on both sides, we get

$$(-1)^n = F_{n+1}F_{n-1} - F_n^2$$

Moreover, since $A^n A^m = A^{n+m}$ for any square matrix A, the following identities can be derived (they are obtained from two different coefficients of the matrix product)

$$F_m F_n + F_{m-1} F_{n-1} = F_{m+n-1}$$

By putting n = n+1,

$$F_m F_{n+1} + F_{m-1} F_n = F_{m+n}$$

Putting m = n

$$F_{2n-1} = F_n^2 + F_{n-1}^2$$

$$F_{2n} = (F_{n-1} + F_{n+1})F_n = (2F_{n-1} + F_n)F_n \text{ (Source: Wiki)}$$

To get the formula to be proved, we simply need to do following

If n is even, we can put k = n/2

If n is odd, we can put k = (n+1)/2

Below is C++ implementation of above idea.

```
// C++ Program to find n'th fibonacci Number in
// with O(Log n) arithmetic operations
#include <bits/stdc++.h>
using namespace std;

const int MAX = 1000;

// Create an array for memoization
int f[MAX] = {0};

// Returns n'th fibonacci number using table f[]
int fib(int n)
{
    // Base cases
    if (n == 0)
        return 0;
    if (n == 1 || n == 2)
        return (f[n] = 1);

    // If fib(n) is already computed
    if (f[n])
        return f[n];

    int k = (n & 1)? (n+1)/2 : n/2;

    // Applying above formula [Note value n&1 is 1
    // if n is odd, else 0.
    f[n] = (n & 1)? (fib(k)*fib(k) + fib(k-1)*fib(k-1))
                  : (2*fib(k-1) + fib(k))*fib(k);

    return f[n];
}

/* Driver program to test above function */
int main()
{
    int n = 9;
    printf("%d ", fib(n));
    return 0;
}
```

Output :

34

Time complexity of this solution is O(Log n) as we divide the problem to half in every recursive call.

This method is contributed by Chirag Agarwal.

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

References:

http://en.wikipedia.org/wiki/Fibonacci_number

<http://www.ics.uci.edu/~eppstein/161/960109.html>

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Mathematical
Dynamic Programming
Fibonacci
MathematicalAlgo
series

Minimum number of jumps to reach end

Given an array of integers where each element represents the max number of steps that can be made forward from that element. Write a function to return the minimum number of jumps to reach the end of the array (starting from the first element). If an element is 0, then cannot move through that element.

Example:

```
Input: arr[] = {1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9}
Output: 3 (1->3->8->9)
```

First element is 1, so can only go to 3. Second element is 3, so can make at most 3 steps eg to 5 or 8 or 9.

We strongly recommend that you click here and practice it, before moving on to the solution.

Method 1 (Naive Recursive Approach)

A naive approach is to start from the first element and recursively call for all the elements reachable from first element. The minimum number of jumps to reach end from first can be calculated using minimum number of jumps needed to reach end from the elements reachable from first.

$\text{minJumps}(\text{start}, \text{end}) = \text{Min} (\text{minJumps}(k, \text{end})) \text{ for all } k \text{ reachable from start}$

```
#include <stdio.h>
#include <limits.h>

// Returns minimum number of jumps to reach arr[h] from arr[l]
int minJumps(int arr[], int l, int h)
{
    // Base case: when source and destination are same
    if (h == l)
        return 0;

    // When nothing is reachable from the given source
    if (arr[l] == 0)
        return INT_MAX;

    // Traverse through all the points reachable from arr[l]. Recursively
    // get the minimum number of jumps needed to reach arr[h] from these
    // reachable points.
    int min = INT_MAX;
    for (int i = l+1; i <= h && i <= l + arr[l]; i++)
    {
        int jumps = minJumps(arr, i, h);
        if(jumps != INT_MAX && jumps + 1 < min)
            min = jumps + 1;
    }

    return min;
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 3, 6, 3, 2, 3, 6, 8, 9, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Minimum number of jumps to reach end is %d ", minJumps(arr, 0, n-1));
    return 0;
}
```

If we trace the execution of this method, we can see that there will be overlapping subproblems. For example, $\text{minJumps}(3, 9)$ will be called two times as $\text{arr}[3]$ is reachable from $\text{arr}[1]$ and $\text{arr}[2]$. So this problem has both properties (**optimal substructure** and **overlapping subproblems**) of Dynamic Programming.

Method 2 (Dynamic Programming)

In this method, we build a $\text{jumps}[]$ array from left to right such that $\text{jumps}[i]$ indicates the minimum number of jumps needed to reach $\text{arr}[i]$ from $\text{arr}[0]$. Finally, we return $\text{jumps}[n-1]$.

```
#include <stdio.h>
#include <limits.h>

int min(int x, int y) { return (x < y)? x: y; }

// Returns minimum number of jumps to reach arr[n-1] from arr[0]
int minJumps(int arr[], int n)
{
    int *jumps = new int[n]; // jumps[n-1] will hold the result
    int i, j;

    if (n == 0 || arr[0] == 0)
        return INT_MAX;

    jumps[0] = 0;

    // Find the minimum number of jumps to reach arr[i]
    // from arr[0], and assign this value to jumps[i]
    for (i = 1; i < n; i++)
    {
        jumps[i] = INT_MAX;
        for (j = 0; j < i; j++)
        {
            if (i <= j + arr[j] && jumps[j] != INT_MAX)
            {
                jumps[i] = min(jumps[i], jumps[j] + 1);
                break;
            }
        }
    }
    return jumps[n-1];
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 3, 6, 1, 0, 9};
    int size = sizeof(arr)/sizeof(int);
    printf("Minimum number of jumps to reach end is %d ", minJumps(arr, size));
    return 0;
}
```

Output:

Minimum number of jumps to reach end is 3

Thanks to [paras](#) for suggesting this method.

Time Complexity: $O(n^2)$

Method 3 (Dynamic Programming)

In this method, we build jumps[] array from right to left such that jumps[i] indicates the minimum number of jumps needed to reach arr[n-1] from arr[i]. Finally, we return arr[0].

```
int minJumps(int arr[], int n)
{
    int *jumps = new int[n]; // jumps[0] will hold the result
    int min;

    // Minimum number of jumps needed to reach last element
    // from last elements itself is always 0
    jumps[n-1] = 0;

    int i, j;

    // Start from the second element, move from right to left
    // and construct the jumps[] array where jumps[i] represents
    // minimum number of jumps needed to reach arr[m-1] from arr[i]
    for (i = n-2; i >= 0; i--)
    {
        // If arr[i] is 0 then arr[n-1] can't be reached from here
        if (arr[i] == 0)
            jumps[i] = INT_MAX;

        // If we can directly reach to the end point from here then
        // jumps[i] is 1
        else if (arr[i] >= n - i - 1)
            jumps[i] = 1;

        // Otherwise, to find out the minimum number of jumps needed
        // to reach arr[n-1], check all the points reachable from here
        // and jumps[] value for those points
        else
        {
            min = INT_MAX; // initialize min value

            // following loop checks with all reachable points and
            // takes the minimum
            for (j = i+1; j < n && j <= arr[i] + i; j++)
            {
                if (min > jumps[j])
                    min = jumps[j];
            }

            // Handle overflow
            if (min != INT_MAX)
                jumps[i] = min + 1;
            else
                jumps[i] = min; // or INT_MAX
        }
    }

    return jumps[0];
}
```

Time Complexity: $O(n^2)$ in worst case.

Thanks to [Ashish](#) for suggesting this solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming

Maximum size square sub-matrix with all 1s

Given a binary matrix, find out the maximum size square sub-matrix with all 1s.

For example, consider the below binary matrix.

maximum-size-square-sub-matrix-with-all-1s



We strongly recommend that you click here and practice it, before moving on to the solution.

Algorithm:

Let the given binary matrix be $M[R][C]$. The idea of the algorithm is to construct an auxiliary size matrix $S[R][C]$ in which each entry $S[i][j]$ represents size of the square sub-matrix with all 1s including $M[i][j]$ where $M[i][j]$ is the rightmost and bottommost entry in sub-matrix.

- 1) Construct a sum matrix $S[R][C]$ for the given $M[R][C]$.
- a) Copy first row and first columns as it is from $M[0][0]$ to $S[0][0]$
- b) For other entries, use following expressions to construct $S[i][j]$
 - If $M[i][j]$ is 1 then
 - $S[i][j] = \min(S[i][j-1], S[i-1][j], S[i-1][j-1]) + 1$
 - Else /*If $M[i][j]$ is 0*/

```

S[i][j] = 0
2) Find the maximum entry in S[R][C]
3) Using the value and coordinates of maximum entry in S[i], print
    sub-matrix of M[()][]

```

For the given M[R][C] in above example, constructed S[R][C] would be:

```

0 1 1 0 1
1 1 0 1 0
0 1 1 1 0
1 1 2 2 0
1 2 2 3 1
0 0 0 0 0

```

The value of maximum entry in above matrix is 3 and coordinates of the entry are (4, 3). Using the maximum value and its coordinates, we can find out the required sub-matrix.

```

#include<stdio.h>
#define bool int
#define R 6
#define C 5

void printMaxSubSquare(bool M[R][C])
{
    int i,j;
    int S[R][C];
    int max_of_s, max_i, max_j;

    /* Set first column of S[][]*/
    for(i = 0; i < R; i++)
        S[i][0] = M[i][0];

    /* Set first row of S[][]*/
    for(j = 0; j < C; j++)
        S[0][j] = M[0][j];

    /* Construct other entries of S[][]*/
    for(i = 1; i < R; i++)
    {
        for(j = 1; j < C; j++)
        {
            if(M[i][j] == 1)
                S[i][j] = min(S[i][j-1], S[i-1][j], S[i-1][j-1]) + 1;
            else
                S[i][j] = 0;
        }
    }

    /* Find the maximum entry, and indexes of maximum entry
     * in S[][] */
    max_of_s = S[0][0]; max_i = 0; max_j = 0;
    for(i = 0; i < R; i++)
    {
        for(j = 0; j < C; j++)
        {
            if(max_of_s < S[i][j])
            {
                max_of_s = S[i][j];
                max_i = i;
                max_j = j;
            }
        }
    }

    printf("\n Maximum size sub-matrix is: \n");
    for(i = max_i; i > max_i - max_of_s; i--)
    {
        for(j = max_j; j > max_j - max_of_s; j--)
        {
            printf("%d ", M[i][j]);
        }
        printf("\n");
    }
}

/* UTILITY FUNCTIONS */
/* Function to get minimum of three values */
int min(int a, int b, int c)
{
    int m = a;
    if (m > b)
        m = b;
    if (m > c)
        m = c;
    return m;
}

/* Driver function to test above functions */
int main()
{
    bool M[R][C] = {{0, 1, 1, 0, 1},
                    {1, 1, 0, 1, 0},
                    {0, 1, 1, 1, 0},
                    {1, 1, 2, 2, 0},
                    {1, 2, 2, 3, 1},
                    {0, 0, 0, 0, 0}};

    printMaxSubSquare(M);
    getchar();
}

```

Time Complexity: O(m*n) where m is number of rows and n is number of columns in the given matrix.

Auxiliary Space: O(m*n) where m is number of rows and n is number of columns in the given matrix.

Algorithmic Paradigm: Dynamic Programming

Please write comments if you find any bug in above code/algorithm, or find other ways to solve the same problem

Ugly Numbers

Ugly numbers are numbers whose only prime factors are 2, 3 or 5. The sequence 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, ... shows the first 11 ugly numbers. By convention, 1 is included.

Given a number n, the task is to find n'th Ugly number.

```
Input :n = 7
Output :8
```

```
Input :n = 10
Output :12
```

```
Input :n = 15
Output :24
```

```
Input :n = 150
Output :5832
```

We strongly recommend that you click here and practice it, before moving on to the solution.

Method 1 (Simple)

Loop for all positive integers until ugly number count is smaller than n, if an integer is ugly than increment ugly number count.

To check if a number is ugly, divide the number by greatest divisible powers of 2, 3 and 5, if the number becomes 1 then it is an ugly number otherwise not.

For example, let us see how to check for 300 is ugly or not. Greatest divisible power of 2 is 4, after dividing 300 by 4 we get 75. Greatest divisible power of 3 is 3, after dividing 75 by 3 we get 25. Greatest divisible power of 5 is 25, after dividing 25 by 25 we get 1. Since we get 1 finally, 300 is ugly number.

Implementation:

```
# include<stdio.h>
# include<stdlib.h>

/*This function divides a by greatest divisible
power of b*/
int maxDivide(int a, int b)
{
    while (a%b == 0)
        a = a/b;
    return a;
}

/* Function to check if a number is ugly or not */
int isUgly(int no)
{
    no = maxDivide(no, 2);
    no = maxDivide(no, 3);
    no = maxDivide(no, 5);

    return (no == 1)? 1 : 0;
}

/* Function to get the nth ugly number*/
int getNthUglyNo(int n)
{
    int i = 1;
    int count = 1; /* ugly number count */

    /*Check for all integers until ugly count
    becomes n*/
    while (n > count)
    {
        i++;
        if (isUgly(i))
            count++;
    }
    return i;
}

/* Driver program to test above functions */
int main()
{
    unsigned no = getNthUglyNo(150);
    printf("150th ugly no. is %d", no);
    getchar();
    return 0;
}
```

This method is not time efficient as it checks for all integers until ugly number count becomes n, but space complexity of this method is O(1)

Method 2 (Use Dynamic Programming)

Here is a time efficient solution with O(n) extra space. The ugly-number sequence is 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, ...

because every number can only be divided by 2, 3, 5, one way to look at the sequence is to split the sequence to three groups as below:

- (1) 1×2, 2×2, 3×2, 4×2, 5×2, ...
- (2) 1×3, 2×3, 3×3, 4×3, 5×3, ...
- (3) 1×5, 2×5, 3×5, 4×5, 5×5, ...

We can find that every subsequence is the ugly-sequence itself (1, 2, 3, 4, 5, ...) multiply 2, 3, 5. Then we use similar merge method as merge sort, to get every ugly number from the three subsequence. Every step we choose the smallest one, and move one step after.

```

1 Declare an array for ugly numbers: ugly[n]
2 Initialize first ugly no: ugly[0] = 1
3 Initialize three array index variables i2, i3, i5 to point to
   1st element of the ugly array:
   i2 = i3 = i5 = 0;
4 Initialize 3 choices for the next ugly no:
   next_multiple_of_2 = ugly[i2]*2;
   next_multiple_of_3 = ugly[i3]*3
   next_multiple_of_5 = ugly[i5]*5;
5 Now go in a loop to fill all ugly numbers till 150:
For (i = 1; i <= 150; i++)

```

Example:

Let us see how it works

```

initialize
ugly[] = | 1 |
i2 = i3 = i5 = 0;

First iteration
ugly[1] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5)
          = Min(2, 3, 5)
          = 2
ugly[] = | 1 | 2 |
i2 = 1, i3 = i5 = 0 (i2 got incremented)

Second iteration
ugly[2] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5)
          = Min(4, 3, 5)
          = 3
ugly[] = | 1 | 2 | 3 |
i2 = 1, i3 = 1, i5 = 0 (i3 got incremented)

Third iteration
ugly[3] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5)
          = Min(4, 6, 5)
          = 4
ugly[] = | 1 | 2 | 3 | 4 |
i2 = 2, i3 = 1, i5 = 0 (i2 got incremented)

Fourth iteration
ugly[4] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5)
          = Min(6, 6, 5)
          = 5
ugly[] = | 1 | 2 | 3 | 4 | 5 |
i2 = 2, i3 = 1, i5 = 1 (i5 got incremented)

Fifth iteration
ugly[5] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5)
          = Min(6, 6, 10)
          = 6
ugly[] = | 1 | 2 | 3 | 4 | 5 | 6 |
i2 = 3, i3 = 2, i5 = 1 (i2 and i3 got incremented)

```

Will continue same way till 150.

C/C++

```

// C++ program to find n'th Ugly number
#include<iostream.h>
using namespace std;

/* Function to get the nth ugly number*/
unsigned getNthUglyNo(unsigned n)
{
    unsigned ugly[n]; // To store ugly numbers
    unsigned i2 = 0, i3 = 0, i5 = 0;
    unsigned next_multiple_of_2 = 2;
    unsigned next_multiple_of_3 = 3;
    unsigned next_multiple_of_5 = 5;
    unsigned next_ugly_no = 1;

    ugly[0] = 1;
    for (int i=1; i<n; i++)
    {
        next_ugly_no = min(next_multiple_of_2,
                           min(next_multiple_of_3,
                               next_multiple_of_5));
        ugly[i] = next_ugly_no;
        if (next_ugly_no == next_multiple_of_2)
        {
            i2 = i2+1;
            next_multiple_of_2 = ugly[i2]*2;
        }
        if (next_ugly_no == next_multiple_of_3)
        {
            i3 = i3+1;
            next_multiple_of_3 = ugly[i3]*3;
        }
        if (next_ugly_no == next_multiple_of_5)
        {
            i5 = i5+1;
            next_multiple_of_5 = ugly[i5]*5;
        }
    } /*End of for loop (i=1; i<n; i++) */
    return next_ugly_no;
}

/* Driver program to test above functions */

```

```
int main()
{
    int n = 150;
    cout << getNthUglyNo(n);
    return 0;
}
```

Python

```
# Python program to find n'th Ugly number

# Function to get the nth ugly number
def getNthUglyNo(n):

    ugly = [0] * n # To store ugly numbers

    # 1 is the first ugly number
    ugly[0] = 1

    # i2, i3, i5 will indicate indices for 2,3,5 respectively
    i2 = i3 = i5 = 0

    # set initial multiple value
    next_multiple_of_2 = 2
    next_multiple_of_3 = 3
    next_multiple_of_5 = 5

    # start loop to find value from ugly[1] to ugly[n]
    for i in range(1, n):

        # choose the min value of all available multiples
        ugly[i] = min(next_multiple_of_2, next_multiple_of_3, next_multiple_of_5)

        # increment the value of index accordingly
        if ugly[i] == next_multiple_of_2:
            i2 += 1
            next_multiple_of_2 = ugly[i2] * 2

        if ugly[i] == next_multiple_of_3:
            i3 += 1
            next_multiple_of_3 = ugly[i3] * 3

        if ugly[i] == next_multiple_of_5:
            i5 += 1
            next_multiple_of_5 = ugly[i5] * 5

    # return ugly[n] value
    return ugly[-1]

def main():

    n = 150

    print(getNthUglyNo(n))

if __name__ == '__main__':
    main()

#This code is contributed by Neelam Yadav
```

Output :

```
5832
```

Algorithmic Paradigm: Dynamic Programming

Time Complexity: $O(n)$

Auxiliary Space: $O(n)$

Super Ugly Number (Number whose prime factors are in given set)

Please write comments if you find any bug in the above program or other ways to solve the same problem.

Largest Sum Contiguous Subarray

Write an efficient C program to find the sum of contiguous subarray within a one-dimensional array of numbers which has the largest sum.

kadane-algorithm



Kadane's Algorithm:

Initialize:
max_so_far = 0
max_ending_here = 0

Loop for each element of the array
(a) max_ending_here = max_ending_here + a[i]
(b) if(max_ending_here < 0)
 Set max_ending_here = 0 because max_ending_here

Explanation:

Simple idea of the Kadane's algorithm is to look for all positive contiguous segments of the array (max_ending_here is used for this). And keep track of maximum sum contiguous segment among all positive segments (max_so_far is used for this).

Lets take the example:
{-2, -3, 4, -1, -2, 1, 5, -3}

```
max_so_far = max_ending_here = 0

for i=0, a[0] = -2
    max_ending_here = max_ending_here + (-2)
    Set max_ending_here = 0 because max_ending_here
```

Program:

C++

```
// C++ program to print largest contiguous array sum
#include<iostream>
#include<climits>
using namespace std;

int maxSubArraySum(int a[], int size)
{
    int max_so_far = INT_MIN, max_ending_here = 0;

    for (int i = 0; i < size; i++)
    {
        max_ending_here = max_ending_here + a[i];
        if (max_so_far < max_ending_here)
            max_so_far = max_ending_here;

        if (max_ending_here < 0)
            max_ending_here = 0;
    }
    return max_so_far;
}

/*Driver program to test maxSubArraySum*/
int main()
{
    int a[] = {-2, -3, 4, -1, -2, 1, 5, -3};
    int n = sizeof(a)/sizeof(a[0]);
    int max_sum = maxSubArraySum(a, n);
    cout << "Maximum contiguous sum is " << max_sum;
    return 0;
}
```

Java

```
import java.io.*;
// Java program to print largest contiguous array sum
import java.util.*;

class Kadane
{
    public static void main (String[] args)
    {
        int [] a = {-2, -3, 4, -1, -2, 1, 5, -3};
        System.out.println("Maximum contiguous sum is " +
                           maxSubArraySum(a));
    }

    static int maxSubArraySum(int a[])
    {
        int size = a.length;
        int max_so_far = Integer.MIN_VALUE, max_ending_here = 0;

        for (int i = 0; i < size; i++)
        {
            max_ending_here = max_ending_here + a[i];
            if (max_so_far < max_ending_here)
                max_so_far = max_ending_here;
            if (max_ending_here < 0)
                max_ending_here = 0;
        }
        return max_so_far;
    }
}
```

Python

```
# Python program to find maximum contiguous subarray

# Function to find the maximum contiguous subarray
from sys import maxint
def maxSubArraySum(a,size):

    max_so_far = -maxint - 1
    max_ending_here = 0

    for i in range(0, size):
        max_ending_here = max_ending_here + a[i]
        if (max_so_far < max_ending_here):
            max_so_far = max_ending_here

        if max_ending_here < 0:
            max_ending_here = 0
    return max_so_far

# Driver function to check the above function
a = [-13, -3, -25, -20, -3, -16, -23, -12, -5, -22, -15, -4, -7]
print "Maximum contiguous sum is", maxSubArraySum(a,len(a))

#This code is contributed by _Devesh Agrawal_
```

Output:

```
Maximum contiguous sum is 7
```

Above program can be optimized further, if we compare max_so_far with max_ending_here only if max_ending_here is greater than 0.

C++

```
int maxSubArraySum(int a[], int size)
{
    int max_so_far = 0, max_ending_here = 0;
    for (int i = 0; i < size; i++)
    {
        max_ending_here = max_ending_here + a[i];
        if (max_ending_here < 0)
            max_ending_here = 0;
        /* Do not compare for all elements. Compare only
         * when max_ending_here > 0 */
        else if (max_so_far < max_ending_here)
            max_so_far = max_ending_here;
    }
    return max_so_far;
}
```

Python

```
def maxSubArraySum(a,size):  
  
    max_so_far = 0  
    max_ending_here = 0  
  
    for i in range(0,size):  
        max_ending_here = max_ending_here + a[i]  
        if max_ending_here < 0:  
            max_ending_here = 0  
  
        # Do not compare for all elements. Compare only  
        # when max_ending_here > 0  
        elif (max_so_far < max_ending_here):  
            max_so_far = max_ending_here  
  
    return max_so_far
```

Time Complexity: $O(n)$

Algorithmic Paradigm: Dynamic Programming

Following is another simple implementation suggested by **Mohit Kumar**. The implementation handles the case when all numbers in array are negative.

C++

```
#include<iostream>  
using namespace std;  
  
int maxSubArraySum(int a[], int size)  
{  
    int max_so_far = a[0];  
    int curr_max = a[0];  
  
    for (int i = 1; i < size; i++)  
    {  
        curr_max = max(a[i], curr_max+a[i]);  
        max_so_far = max(max_so_far, curr_max);  
    }  
    return max_so_far;  
}  
  
/* Driver program to test maxSubArraySum */  
int main()  
{  
    int a[] = {-2, -3, 4, -1, -2, 1, 5, -3};  
    int n = sizeof(a)/sizeof(a[0]);  
    int max_sum = maxSubArraySum(a, n);  
    cout << "Maximum contiguous sum is " << max_sum;  
    return 0;  
}
```

Python

```
# Python program to find maximum contiguous subarray  
  
def maxSubArraySum(a,size):  
  
    max_so_far =a[0]  
    curr_max = a[0]  
  
    for i in range(1,size):  
        curr_max = max(a[i], curr_max + a[i])  
        max_so_far = max(max_so_far,curr_max)  
  
    return max_so_far  
  
# Driver function to check the above function  
a = [-2,-3,4,-1,-2,1,5,-3]  
print"Maximum contiguous sum is" , maxSubArraySum(a,len(a))  
  
#This code is contributed by _Devesh Agrawal_
```

Output:

Maximum contiguous sum is 7

To print the subarray with the maximum sum, we maintain indices whenever we get the maximum sum.

```
// C++ program to print largest contiguous array sum
#include<iostream>
#include<climits>
using namespace std;

int maxSubArraySum(int a[], int size)
{
    int max_so_far = INT_MIN, max_ending_here = 0,
        start = 0, end = 0, s = 0;

    for (int i=0; i< size; i++)
    {
        max_ending_here += a[i];

        if (max_so_far < max_ending_here)
        {
            max_so_far = max_ending_here;
            start = s;
            end = i;
        }

        if (max_ending_here < 0)
        {
            max_ending_here = 0;
            s = i+1;
        }
    }
    cout << "Maximum contiguous sum is "
        << max_so_far << endl;
    cout << "Starting index " << start
        << endl << "Ending index " << end << endl;
}

/*Driver program to test maxSubArraySum*/
int main()
{
    int a[] = {-2, -3, 4, -1, -2, 1, 5, -3};
    int n = sizeof(a)/sizeof(a[0]);
    int max_sum = maxSubArraySum(a, n);
    return 0;
}
```

Output:

```
Maximum contiguous sum is 7
Starting index 2
Ending index 6
```

Now try below question

Given an array of integers (possibly some of the elements negative), write a C program to find out the *maximum product* possible by adding 'n' consecutive integers in the array,

References:

http://en.wikipedia.org/wiki/Kadane%27s_Algorithm

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner
Company Wise Coding Practice

Dynamic Programming
Amazon-Question
Dynamic Programming
Visa-Question

Longest Palindromic Substring | Set 1

Given a string, find the longest substring which is palindrome. For example, if the given string is "forgeeksskeegfor", the output should be "geeksskeeg".

We strongly recommend that you click here and practice it, before moving on to the solution.

Method 1 (Brute Force)

The simple approach is to check each substring whether the substring is a palindrome or not. We can run three loops, the outer two loops pick all substrings one by one by fixing the corner characters, the inner loop checks whether the picked substring is palindrome or not.

Time complexity: O (n^3)

Auxiliary complexity: O (1)

Method 2 (Dynamic Programming)

The time complexity can be reduced by storing results of subproblems. The idea is similar to this post. We maintain a boolean table[n][n] that is filled in bottom up manner. The value of table[i][j] is true, if the substring is palindrome, otherwise false. To calculate table[i][j], we first check the value of table[i+1][j-1], if the value is true and str[i] is same as str[j], then we make table[i][j] true. Otherwise, the value of table[i][j] is made false.

```
// A dynamic programming solution for longest palindr.  
// This code is adopted from following link  
// http://www.leetcode.com/2011/11/longest-palindromic-substring-part-i.html
```

```
#include <stdio.h>  
#include <string.h>  
  
// A utility function to print a substring str[low..high]  
void printSubStr( char* str, int low, int high )  
{  
    for( int i = low; i <= high; ++i )  
        printf("%c", str[i]);  
}  
  
// This function prints the longest palindrome substring  
// of str[].  
// It also returns the length of the longest palindrome  
int longestPalSubstr( char *str )  
{  
    int n = strlen( str ); // get length of input string  
  
    // table[i][j] will be false if substring str[i..j]  
    // is not palindrome.  
    // Else table[i][j] will be true  
    bool table[n][n];  
    memset(table, 0, sizeof(table));  
  
    // All substrings of length 1 are palindromes  
    int maxLength = 1;  
    for (int i = 0; i < n; ++i)  
        table[i][i] = true;  
  
    // check for sub-string of length 2.  
    int start = 0;  
    for (int i = 0; i < n-1; ++i)  
    {  
        if (str[i] == str[i+1])  
        {  
            table[i][i+1] = true;  
            start = i;  
            maxLength = 2;  
        }  
    }  
  
    // Check for lengths greater than 2. k is length  
    // of substring  
    for (int k = 3; k <= n; ++k)  
    {  
        // Fix the starting index  
        for (int i = 0; i < n-k+1 ; ++i)  
        {  
            // Get the ending index of substring from  
            // starting index i and length k  
            int j = i + k - 1;  
  
            // checking for sub-string from ith index to  
            // jth index iff str[i+1] to str[j-1] is a  
            // palindrome  
            if (table[i+1][j-1] && str[i] == str[j])  
            {  
                table[i][j] = true;  
  
                if (k > maxLength)  
                {  
                    start = i;  
                    maxLength = k;  
                }  
            }  
        }  
    }  
}
```

```

        }
    }

printf("Longest palindrome substring is: ");
printSubStr( str, start, start + maxLength - 1 );

return maxLength; // return length of LPS
}

// Driver program to test above functions
int main()
{
    char str[] = "forgeeksskeegfor";
    printf("\nLength is: %d\n", longestPalSubstr( str ) );
    return 0;
}

```

Output:

```

Longest palindrome substring is: geeksskeeg
Length is: 10

```

Time complexity: O (n²)

Auxiliary Space: O (n²)

We will soon be adding more optimized methods as separate posts.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Strings
Dynamic Programming

Dynamic Programming | Set 23 (Bellman–Ford Algorithm)

Given a graph and a source vertex *src* in graph, find shortest paths from *src* to all vertices in the given graph. The graph may contain negative weight edges.

We have discussed [Dijkstra's algorithm](#) for this problem. Dijkstra's algorithm is a Greedy algorithm and time complexity is O(VLogV) (with the use of Fibonacci heap). *Dijkstra doesn't work for Graphs with negative weight edges, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.*

Algorithm

Following are the detailed steps.

Input: Graph and a source vertex *src*

Output: Shortest distance to all vertices from *src*. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

1) This step initializes distances from source to all vertices as infinite and distance to source itself as 0. Create an array *dist[]* of size |V| with all values as infinite except *dist[src]* where *src* is source vertex.

2) This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.

.....a) Do following for each edge u-v

.....If *dist[v] > dist[u] + weight of edge uv*, then update *dist[v]*

.....*dist[v] = dist[u] + weight of edge uv*

3) This step reports if there is a negative weight cycle in graph. Do following for each edge u-v

.....If *dist[v] > dist[u] + weight of edge uv*, then "Graph contains negative weight cycle"

The idea of step 3 is, step 2 guarantees shortest distances if graph doesn't contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

How does this work? Like other Dynamic Programming Problems, the algorithm calculate shortest paths in bottom-up manner. It first calculates the shortest distances for the shortest paths which have at-most one edge in the path. Then, it calculates shortest paths with at-most 2 edges, and so on. After the ith iteration of outer loop, the shortest paths with at most i edges are calculated. There can be maximum |V| – 1 edges in any simple path, that is why the outer loop runs |V| – 1 times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges (Proof is simple, you can refer [this](#) or [MIT Video Lecture](#))

Example

Let us understand the algorithm with following example graph. The images are taken from [this](#) source.

Let the given source vertex be 0. Initialize all distances as infinite, except the distance to source itself. Total number of vertices in the graph is 5, so *all edges must be processed 4 times*.

Example Graph



Let all edges are processed in following order: (B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D). We get following distances when all edges are processed first time. The first row in shows initial distances. The second row shows distances when edges (B,E), (D,B), (B,D) and (A,B) are processed. The third row shows distances when (A,C) is processed. The fourth row shows when (D,C), (B,C) and (E,D) are processed.



The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get following distances when all edges are processed second time (The last row shows final values).



The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.

Implementation:

C++

```
// A C / C++ program for Bellman-Ford's single source
// shortest path algorithm.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

// a structure to represent a weighted edge in graph
struct Edge
{
    int src, dest, weight;
};

// a structure to represent a connected, directed and
// weighted graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph =
        (struct Graph*) malloc( sizeof(struct Graph) );
    graph->V = V;
    graph->E = E;

    graph->edge =
        (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );

    return graph;
}

// A utility function used to print the solution
void printArr(int dist[], int n)
{
    printf("Vertex Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t %d\n", i, dist[i]);
}

// The main function that finds shortest distances from src to
// all other vertices using Bellman-Ford algorithm. The function
// also detects negative weight cycle
void BellmanFord(struct Graph* graph, int src)
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];

    // Step 1: Initialize distances from src to all other vertices
    // as INFINITE
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src] = 0;

    // Step 2: Relax all edges |V| - 1 times. A simple shortest
    // path from src to any other vertex can have at-most |V| - 1
    // edges
    for (int i = 1; i <= V-1; i++)
    {
        for (int j = 0; j < E; j++)
        {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }

    // Step 3: check for negative-weight cycles. The above step
    // guarantees shortest distances if graph doesn't contain
    // negative weight cycle. If we get a shorter path, then there
    // is a cycle.
    for (int i = 0; i < E; i++)
    {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
            printf("Graph contains negative weight cycle");
    }
}
```

```

    }

    printArr(dist, V);

    return;
}

// Driver program to test above functions
int main()
{
    /* Let us create the graph given in above example */
    int V = 5; // Number of vertices in graph
    int E = 8; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1 (or A-B in above figure)
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = -1;

    // add edge 0-2 (or A-C in above figure)
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 4;

    // add edge 1-2 (or B-C in above figure)
    graph->edge[2].src = 1;
    graph->edge[2].dest = 2;
    graph->edge[2].weight = 3;

    // add edge 1-3 (or B-D in above figure)
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 2;

    // add edge 1-4 (or A-E in above figure)
    graph->edge[4].src = 1;
    graph->edge[4].dest = 4;
    graph->edge[4].weight = 2;

    // add edge 3-2 (or D-C in above figure)
    graph->edge[5].src = 3;
    graph->edge[5].dest = 2;
    graph->edge[5].weight = 5;

    // add edge 3-1 (or D-B in above figure)
    graph->edge[6].src = 3;
    graph->edge[6].dest = 1;
    graph->edge[6].weight = 1;

    // add edge 4-3 (or E-D in above figure)
    graph->edge[7].src = 4;
    graph->edge[7].dest = 3;
    graph->edge[7].weight = -3;

    BellmanFord(graph, 0);

    return 0;
}

```

Java

```

// A Java program for Bellman-Ford's single source shortest path
// algorithm.
import java.util.*;
import java.lang.*;
import java.io.*;

// A class to represent a connected, directed and weighted graph
class Graph
{
    // A class to represent a weighted edge in graph
    class Edge {
        int src, dest, weight;
        Edge() {
            src = dest = weight = 0;
        }
    };
    int V, E;
    Edge edge[][];
}

// Creates a graph with V vertices and E edges
Graph(int v, int e)
{
    V = v;
    E = e;
    edge = new Edge[e];
    for (int i=0; i<e; ++i)
        edge[i] = new Edge();
}

// The main function that finds shortest distances from src
// to all other vertices using Bellman-Ford algorithm. The
// function also detects negative weight cycle
void BellmanFord(Graph graph,int src)
{
    int V = graph.V, E = graph.E;
    int dist[] = new int[V];

    // Step 1: Initialize distances from src to all other
    // vertices as INFINITE
    for (int i=0; i<V; ++i)
        dist[i] = Integer.MAX_VALUE;

```

```

dist[src] = 0;

// Step 2: Relax all edges |V| - 1 times. A simple
// shortest path from src to any other vertex can
// have at-most |V| - 1 edges
for (int i=1; i<V; ++i)
{
    for (int j=0; j<E; ++j)
    {
        int u = graph.edge[j].src;
        int v = graph.edge[j].dest;
        int weight = graph.edge[j].weight;
        if (dist[u]==Integer.MAX_VALUE &&
            dist[u]+weight<dist[v])
            dist[v]=dist[u]+weight;
    }
}

// Step 3: check for negative-weight cycles. The above
// step guarantees shortest distances if graph doesn't
// contain negative weight cycle. If we get a shorter
// path, then there is a cycle.
for (int j=0; j<E; ++j)
{
    int u = graph.edge[j].src;
    int v = graph.edge[j].dest;
    int weight = graph.edge[j].weight;
    if (dist[u]==Integer.MAX_VALUE &&
        dist[u]+weight<dist[v])
        System.out.println("Graph contains negative weight cycle");
}
printArr(dist, V);
}

// A utility function used to print the solution
void printArr(int dist[], int V)
{
    System.out.println("Vertex Distance from Source");
    for (int i=0; i<V; ++i)
        System.out.println(i+" "+dist[i]);
}

// Driver method to test above function
public static void main(String[] args)
{
    int V = 5; // Number of vertices in graph
    int E = 8; // Number of edges in graph

    Graph graph = new Graph(V, E);

    // add edge 0-1 (or A-B in above figure)
    graph.edge[0].src = 0;
    graph.edge[0].dest = 1;
    graph.edge[0].weight = -1;

    // add edge 0-2 (or A-C in above figure)
    graph.edge[1].src = 0;
    graph.edge[1].dest = 2;
    graph.edge[1].weight = 4;

    // add edge 1-2 (or B-C in above figure)
    graph.edge[2].src = 1;
    graph.edge[2].dest = 2;
    graph.edge[2].weight = 3;

    // add edge 1-3 (or B-D in above figure)
    graph.edge[3].src = 1;
    graph.edge[3].dest = 3;
    graph.edge[3].weight = 2;

    // add edge 1-4 (or B-E in above figure)
    graph.edge[4].src = 1;
    graph.edge[4].dest = 4;
    graph.edge[4].weight = 2;

    // add edge 3-2 (or D-C in above figure)
    graph.edge[5].src = 3;
    graph.edge[5].dest = 2;
    graph.edge[5].weight = 5;

    // add edge 3-1 (or D-B in above figure)
    graph.edge[6].src = 3;
    graph.edge[6].dest = 1;
    graph.edge[6].weight = 1;

    // add edge 4-3 (or E-D in above figure)
    graph.edge[7].src = 4;
    graph.edge[7].dest = 3;
    graph.edge[7].weight = -3;

    graph.BellmanFord(graph, 0);
}
}

// Contributed by Aakash Hasija

```

Python

```

# Python program for Bellman-Ford's single source
# shortest path algorithm.

from collections import defaultdict

#Class to represent a graph
class Graph:

```

```

def __init__(self,vertices):
    self.V= vertices #No. of vertices
    self.graph = {} # default dictionary to store graph

    # function to add an edge to graph
    def addEdge(self,u,v,w):
        self.graph.append([u, v, w])

    # utility function used to print the solution
    def printArr(self,dist):
        print("Vertex Distance from Source")
        for i in range(self.V):
            print("%d \t %d" % (i, dist[i]))

    # The main function that finds shortest distances from src to
    # all other vertices using Bellman-Ford algorithm. The function
    # also detects negative weight cycle
    def BellmanFord(self,src):

        # Step 1: Initialize distances from src to all other vertices
        # as INFINITE
        dist = [float("Inf")] * self.V
        dist[src] = 0

        # Step 2: Relax all edges |V| - 1 times. A simple shortest
        # path from src to any other vertex can have at-most |V| - 1
        # edges
        for i in range(self.V - 1):
            # Update dist value and parent index of the adjacent vertices of
            # the picked vertex. Consider only those vertices which are still in
            # queue
            for u, v, w in self.graph:
                if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w

        # Step 3: check for negative-weight cycles. The above step
        # guarantees shortest distances if graph doesn't contain
        # negative weight cycle. If we get a shorter path, then there
        # is a cycle.

        for u, v, w in self.graph:
            if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                print "Graph contains negative weight cycle"
                return

        # print all distance
        self.printArr(dist)

g = Graph(5)
g.addEdge(0, 1, -1)
g.addEdge(0, 2, 4)
g.addEdge(1, 2, 3)
g.addEdge(1, 3, 2)
g.addEdge(1, 4, 2)
g.addEdge(3, 2, 5)
g.addEdge(3, 1, 1)
g.addEdge(4, 3, -3)

#Print the solution
g.BellmanFord(0)

#This code is contributed by Neelam Yadav

```

Output:

Vertex	Distance from Source
0	0
1	-1
2	2
3	-2
4	1

Notes

- 1) Negative weights are found in various applications of graphs. For example, instead of paying cost for a path, we may get some advantage if we follow the path.
- 2) Bellman-Ford works better (better than Dijksra's) for distributed systems. Unlike Dijksra's where we need to find minimum value of all vertices, in Bellman-Ford, edges are considered one by one.

Exercise

- 1) The standard Bellman-Ford algorithm reports shortest path only if there is no negative weight cycles. Modify it so that it reports minimum distances even if there is a negative weight cycle.
- 2) Can we use Dijksra's algorithm for shortest paths for graphs with negative weights – one idea can be, calculate the minimum weight value, add a positive value (equal to absolute value of minimum weight value) to all weights and run the Dijksra's algorithm for the modified graph. Will this algorithm work?

References:

<http://www.youtube.com/watch?v=Ttezuzs39nk>
http://en.wikipedia.org/wiki/Bellman%20%80%93Ford_algorithm
<http://www.cs.arizona.edu/classes/cs445/spring07/ShortestPath2.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Graph
shortest path

Dynamic Programming | Set 24 (Optimal Binary Search Tree)

Given a sorted array $keys[0..n-1]$ of search keys and an array $freq[0..n-1]$ of frequency counts, where $freq[i]$ is the number of searches to $keys[i]$. Construct a binary search tree of all keys such that the total cost of all the searches is as small as possible.

Let us first define the cost of a BST. The cost of a BST node is level of that node multiplied by its frequency. Level of root is 1.

Example 1

Input: keys[] = {10, 12}, freq[] = {34, 50}

There can be following two possible BSTs



Frequency of searches of 10 and 12 are 34 and 50 respectively.

The cost of tree I is $34 \cdot 1 + 50 \cdot 2 = 134$

The cost of tree II is $50 \cdot 1 + 34 \cdot 2 = 118$

Example 2

Input: keys[] = {10, 12, 20}, freq[] = {34, 8, 50}

There can be following possible BSTs



Among all possible BSTs, cost of the fifth BST is minimum.

Cost of the fifth BST is $1 \cdot 50 + 2 \cdot 34 + 3 \cdot 8 = 142$

1) Optimal Substructure:

The optimal cost for freq[i..j] can be recursively calculated using following formula.



We need to calculate $optCost(i, n-1)$ to find the result.

The idea of above formula is simple, we one by one try all nodes as root (r varies from i to j in second term). When we make r th node as root, we recursively calculate optimal cost from i to $r-1$ and $r+1$ to j. We add sum of frequencies from i to j (see first term in the above formula), this is added because every search will go through root and one comparison will be done for every search.

2) Overlapping Subproblems

Following is recursive implementation that simply follows the recursive structure mentioned above.

```
// A naive recursive implementation of optimal binary search tree problem
#include <stdio.h>
#include <limits.h>

// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j);

// A recursive function to calculate cost of optimal binary search tree
int optCost(int freq[], int i, int j)
{
    // Base cases
    if (j < i)    // If there are no elements in this subarray
        return 0;
    if (j == i)    // If there is one element in this subarray
        return freq[i];

    // Get sum of freq[i], freq[i+1], ... freq[j]
    int fsum = sum(freq, i, j);

    // Initialize minimum value
    int min = INT_MAX;

    // One by one consider all elements as root and recursively find cost
    // of the BST, compare the cost with min and update min if needed
    for (int r = i; r <= j; ++r)
    {
        int cost = optCost(freq, i, r-1) + optCost(freq, r+1, j);
        if (cost < min)
            min = cost;
    }

    // Return minimum value
    return min + fsum;
}

// The main function that calculates minimum cost of a Binary Search Tree.
// It mainly uses optCost() to find the optimal cost.
int optimalSearchTree(int keys[], int freq[], int n)
{
    // Here array keys[] is assumed to be sorted in increasing order.
    // If keys[] is not sorted, then add code to sort keys, and rearrange
    // freq[] accordingly.
    return optCost(freq, 0, n-1);
}

// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j)
{
    int s = 0;
    for (int k = i; k <= j; k++)
        s += freq[k];
    return s;
}

// Driver program to test above functions
int main()
{
    int keys[] = {10, 12, 20};
    int freq[] = {34, 8, 50};
    int n = sizeof(keys)/sizeof(keys[0]);
    printf("Cost of Optimal BST is %d ", optimalSearchTree(keys, freq, n));
    return 0;
}
```

Output:

Cost of Optimal BST is 142

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. We can see many subproblems being repeated in the following recursion tree for freq[1..4].



Since same subproblems are called again, this problem has Overlapping Subproblems property. So optimal BST problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem. Like other typical [Dynamic Programming\(DP\) problems](#), recomputations of same subproblems can be avoided by constructing a temporary array cost[][] in bottom up manner.

Dynamic Programming Solution

Following is C/C++ implementation for optimal BST problem using Dynamic Programming. We use an auxiliary array cost[n][n] to store the solutions of subproblems. cost[0][n-1] will hold the final result. The challenge in implementation is, all diagonal values must be filled first, then the values which lie on the line just above the diagonal. In other words, we must first fill all cost[i][i] values, then all cost[i][i+1] values, then all cost[i][i+2] values. So how to fill the 2D array in such manner? The idea used in the implementation is same as [Matrix Chain Multiplication problem](#), we use a variable 'L' for chain length and increment 'L', one by one. We calculate column number 'j' using the values of 'i' and 'L'.

```
// Dynamic Programming code for Optimal Binary Search Tree Problem
#include <stdio.h>
#include <limits.h>

// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j);

/* A Dynamic Programming based function that calculates minimum cost of
   a Binary Search Tree. */
int optimalSearchTree(int keys[], int freq[], int n)
{
    /* Create an auxiliary 2D matrix to store results of subproblems */
    int cost[n][n];

    /* cost[i][j] = Optimal cost of binary search tree that can be
       formed from keys[i] to keys[j].
       cost[0][n-1] will store the resultant cost */

    /* For a single key, cost is equal to frequency of the key
    for (int i = 0; i < n; i++)
        cost[i][i] = freq[i];

    // Now we need to consider chains of length 2, 3, ...
    // L is chain length.
    for (int L=2; L<=n; L++)
    {
        /* i is row number in cost[][]
           for (int i=0; i<=n-L+1; i++)
        {
            // Get column number j from row number i and chain length L
            int j = i+L-1;
            cost[i][j] = INT_MAX;

            // Try making all keys in interval keys[i..j] as root
            for (int r=i; r<=j; r++)
            {
                // c = cost when keys[r] becomes root of this subtree
                int c = ((r > i)? cost[i][r-1]:0) +
                       ((r < j)? cost[r+1][j]:0) +
                       sum(freq, i, j);
                if (c < cost[i][j])
                    cost[i][j] = c;
            }
        }
    }
    return cost[0][n-1];
}

// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j)
{
    int s = 0;
    for (int k = i; k <= j; k++)
        s += freq[k];
    return s;
}

// Driver program to test above functions
int main()
{
    int keys[] = {10, 12, 20};
    int freq[] = {34, 8, 50};
    int n = sizeof(keys)/sizeof(keys[0]);
    printf("Cost of Optimal BST is %d ", optimalSearchTree(keys, freq, n));
    return 0;
}
```

Output:

```
Cost of Optimal BST is 142
```

Notes

- 1) The time complexity of the above solution is $O(n^4)$. The time complexity can be easily reduced to $O(n^3)$ by pre-calculating sum of frequencies instead of calling sum() again and again.
- 2) In the above solutions, we have computed optimal cost only. The solutions can be easily modified to store the structure of BSTs also. We can create another auxiliary array of size n to store the structure of tree. All we need to do is, store the chosen 'r' in the innermost loop.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Dynamic Programming | Set 26 (Largest Independent Set Problem)

Given a Binary Tree, find size of the Largest Independent Set(LIS) in it. A subset of all tree nodes is an independent set if there is no edge between any two nodes of the subset.

For example, consider the following binary tree. The largest independent set(LIS) is {10, 40, 60, 70, 80} and size of the LIS is 5.



A Dynamic Programming solution solves a given problem using solutions of subproblems in bottom up manner. Can the given problem be solved using solutions to subproblems? If yes, then what are the subproblems? Can we find largest independent set size (LISS) for a node X if we know LISS for all descendants of X? If a node is considered as part of LIS, then its children cannot be part of LIS, but its grandchildren can be. Following is optimal substructure property.

1) Optimal Substructure:

Let LISS(X) indicates size of largest independent set of a tree with root X.

$$\text{LISS}(X) = \max \{ (1 + \text{sum of LISS for all grandchildren of } X), (\text{sum of LISS for all children of } X) \}$$

The idea is simple, there are two possibilities for every node X, either X is a member of the set or not a member. If X is a member, then the value of LISS(X) is 1 plus LISS of all grandchildren. If X is not a member, then the value is sum of LISS of all children.

2) Overlapping Subproblems

Following is recursive implementation that simply follows the recursive structure mentioned above.

```

// A naive recursive implementation of Largest Independent Set problem
#include <stdio.h>
#include <stdlib.h>

// A utility function to find max of two integers
int max(int x, int y) { return (x > y)? x: y; }

/* A binary tree node has data, pointer to left child and a pointer to
right child */
struct node
{
    int data;
    struct node *left, *right;
};

// The function returns size of the largest independent set in a given
// binary tree
int LISS(struct node *root)
{
    if (root == NULL)
        return 0;

    // Calculate size excluding the current node
    int size_excl = LISS(root->left) + LISS(root->right);

    // Calculate size including the current node
    int size_incl = 1;
    if (root->left)
        size_incl += LISS(root->left->left) + LISS(root->left->right);
    if (root->right)
        size_incl += LISS(root->right->left) + LISS(root->right->right);

    // Return the maximum of two sizes
    return max(size_incl, size_excl);
}

// A utility function to create a node
struct node* newNode( int data )
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree given in the above diagram
    struct node *root      = newNode(20);
    root->left           = newNode(8);
    root->left->left     = newNode(4);
    root->left->right    = newNode(12);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(14);
    root->right          = newNode(22);
    root->right->right   = newNode(25);

    printf ("Size of the Largest Independent Set is %d ", LISS(root));

    return 0;
}
  
```

Output:

```
Size of the Largest Independent Set is 5
```

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. For example, LISS of node with value 50 is evaluated for node with values 10 and 20 as 50 is grandchild of 10 and child of 20.
Since same subproblems are called again, this problem has Overlapping Subproblems property. So LISS problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by storing the solutions to subproblems and solving problems in bottom up manner.

Following is C implementation of Dynamic Programming based solution. In the following solution, an additional field 'liss' is added to tree nodes. The initial value of 'liss' is set as 0 for all nodes. The recursive function LISS() calculates 'liss' for a node only if it is not already set.

```
/* Dynamic programming based program for Largest Independent Set problem */
#include <stdio.h>
#include <stdlib.h>

// A utility function to find max of two integers
int max(int x, int y) { return (x > y)? x: y; }

/* A binary tree node has data, pointer to left child and a pointer to
right child */
struct node
{
    int data;
    int liss;
    struct node *left, *right;
};

// A memoization function returns size of the largest independent set in
// a given binary tree
int LISS(struct node *root)
{
    if (root == NULL)
        return 0;

    if (root->liss)
        return root->liss;

    if (root->left == NULL && root->right == NULL)
        return (root->liss = 1);

    // Calculate size excluding the current node
    int liss_excl = LISS(root->left) + LISS(root->right);

    // Calculate size including the current node
    int liss_incl = 1;
    if (root->left)
        liss_incl += LISS(root->left->left) + LISS(root->left->right);
    if (root->right)
        liss_incl += LISS(root->right->left) + LISS(root->right->right);

    // Maximum of two sizes is LISS, store it for future uses.
    root->liss = max(liss_incl, liss_excl);

    return root->liss;
}

// A utility function to create a node
struct node* newNode(int data)
{
    struct node* temp = (struct node*) malloc( sizeof(struct node) );
    temp->data = data;
    temp->left = temp->right = NULL;
    temp->liss = 0;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree given in the above diagram
    struct node *root      = newNode(20);
    root->left           = newNode(8);
    root->left->left    = newNode(4);
    root->left->right   = newNode(12);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(14);
    root->right          = newNode(22);
    root->right->right  = newNode(25);

    printf ("Size of the Largest Independent Set is %d ", LISS(root));

    return 0;
}
```

Output

```
Size of the Largest Independent Set is 5
```

Time Complexity: O(n) where n is the number of nodes in given Binary tree.

Following extensions to above solution can be tried as an exercise.

- 1) Extend the above solution for n-ary tree.
- 2) The above solution modifies the given tree structure by adding an additional field 'liss' to tree nodes. Extend the solution so that it doesn't modify the tree structure.
- 3) The above solution only returns size of LIS, it doesn't print elements of LIS. Extend the solution to print all nodes that are part of LIS.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming | Set 25 (Subset Sum Problem)

Given a set of non-negative integers, and a value sum , determine if there is a subset of the given set with sum equal to given sum .

Examples: $set[] = \{3, 34, 4, 12, 5, 2\}$, $sum = 9$
Output: True //There is a subset (4, 5) with sum 9.

We strongly recommend that you click here and practice it, before moving on to the solution.

Let $\text{isSubSetSum}(\text{int set}[], \text{int } n, \text{int } sum)$ be the function to find whether there is a subset of $set[]$ with sum equal to sum . n is the number of elements in $set[]$.

The isSubSetSum problem can be divided into two subproblems

- ...a) Include the last element, recur for $n = n-1$, $sum = sum - set[n-1]$
- ...b) Exclude the last element, recur for $n = n-1$.

If any of the above the above subproblems return true, then return true.

Following is the recursive formula for $\text{isSubSetSum}()$ problem.

```
isSubSetSum(set, n, sum) = isSubSetSum(set, n-1, sum) ||  
                           isSubSetSum(set, n-1, sum-set[n-1])  
Base Cases:  
isSubSetSum(set, n, sum) = false, if sum > 0 and n == 0  
isSubSetSum(set, n, sum) = true, if sum == 0
```

Following is naive recursive implementation that simply follows the recursive structure mentioned above.

C

```
// A recursive solution for subset sum problem  
#include <stdio.h>  
  
// Returns true if there is a subset of set[] with sun equal to given sum  
bool isSubSetSum(int set[], int n, int sum)  
{  
    // Base Cases  
    if (sum == 0)  
        return true;  
    if (n == 0 && sum != 0)  
        return false;  
  
    // If last element is greater than sum, then ignore it  
    if (set[n-1] > sum)  
        return isSubSetSum(set, n-1, sum);  
  
    /* else, check if sum can be obtained by any of the following  
     * (a) including the last element  
     * (b) excluding the last element */  
    return isSubSetSum(set, n-1, sum) ||  
           isSubSetSum(set, n-1, sum-set[n-1]);  
}  
  
// Driver program to test above function  
int main()  
{  
    int set[] = {3, 34, 4, 12, 5, 2};  
    int sum = 9;  
    int n = sizeof(set)/sizeof(set[0]);  
    if (isSubSetSum(set, n, sum) == true)  
        printf("Found a subset with given sum");  
    else  
        printf("No subset with given sum");  
    return 0;  
}
```

Java

```
// A recursive solution for subset sum problem  
class subset_sum  
{  
    // Returns true if there is a subset of set[] with sum  
    // equal to given sum  
    static boolean isSubSetSum(int set[], int n, int sum)  
{  
        // Base Cases  
        if (sum == 0)  
            return true;  
        if (n == 0 && sum != 0)  
            return false;  
  
        // If last element is greater than sum, then ignore it  
        if (set[n-1] > sum)  
            return isSubSetSum(set, n-1, sum);  
  
        /* else, check if sum can be obtained by any of the following  
         * (a) including the last element  
         * (b) excluding the last element */  
        return isSubSetSum(set, n-1, sum) ||  
               isSubSetSum(set, n-1, sum-set[n-1]);  
    }  
    /* Driver program to test above function */  
    public static void main (String args[])  
    {  
        int set[] = {3, 34, 4, 12, 5, 2};  
        int sum = 9;  
        int n = set.length;  
        if (isSubSetSum(set, n, sum) == true)  
            System.out.println("Found a subset with given sum");  
        else
```

```

        System.out.println("No subset with given sum");
    }
    /* This code is contributed by Rajat Mishra */

```

Output:

```
Found a subset with given sum
```

The above solution may try all subsets of given set in worst case. Therefore time complexity of the above solution is exponential. The problem is in-fact NP-Complete (There is no known polynomial time solution for this problem).

We can solve the problem in Pseudo-polynomial time using Dynamic programming. We create a boolean 2D table `subset[i][j]` and fill it in bottom up manner. The value of `subset[i][j]` will be true if there is a subset of `set[0..j-1]` with sum equal to `i`, otherwise false. Finally, we return `subset[sum][n]`

C

```

// A Dynamic Programming solution for subset sum problem
#include <stdio.h>

// Returns true if there is a subset of set[] with sun equal to given sum
bool isSubsetSum(int set[], int n, int sum)
{
    // The value of subset[i][j] will be true if there is a
    // subset of set[0..j-1] with sum equal to i
    bool subset[sum+1][n+1];

    // If sum is 0, then answer is true
    for (int i = 0; i <= n; i++)
        subset[0][i] = true;

    // If sum is not 0 and set is empty, then answer is false
    for (int i = 1; i <= sum; i++)
        subset[i][0] = false;

    // Fill the subset table in botton up manner
    for (int i = 1; i <= sum; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            subset[i][j] = subset[i][j-1];
            if (i >= set[j-1])
                subset[i][j] = subset[i][j] || subset[i - set[j-1]][j-1];
        }
    }

    /* // uncomment this code to print table
    for (int i = 0; i <= sum; i++)
    {
        for (int j = 0; j <= n; j++)
            printf("%4d", subset[i][j]);
        printf("\n");
    } */

    return subset[sum][n];
}

// Driver program to test above function
int main()
{
    int set[] = {3, 34, 4, 12, 5, 2};
    int sum = 9;
    int n = sizeof(set)/sizeof(set[0]);
    if (isSubsetSum(set, n, sum) == true)
        printf("Found a subset with given sum");
    else
        printf("No subset with given sum");
    return 0;
}

```

Java

```

// A Dynamic Programming solution for subset sum problem
class subset_sum
{
    // Returns true if there is a subset of set[] with sun equal to given sum
    static boolean isSubsetSum(int set[], int n, int sum)
    {
        // The value of subset[i][j] will be true if there
        // is a subset of set[0..j-1] with sum equal to i
        boolean subset[][] = new boolean[sum+1][n+1];

        // If sum is 0, then answer is true
        for (int i = 0; i <= n; i++)
            subset[0][i] = true;

        // If sum is not 0 and set is empty, then answer is false
        for (int i = 1; i <= sum; i++)
            subset[i][0] = false;

        // Fill the subset table in botton up manner
        for (int i = 1; i <= sum; i++)
        {
            for (int j = 1; j <= n; j++)
            {
                subset[i][j] = subset[i][j-1];
                if (i >= set[j-1])
                    subset[i][j] = subset[i][j] || subset[i - set[j-1]][j-1];
            }
        }

        return subset[sum][n];
    }
}

```

```

}

/* // uncomment this code to print table
for (int i = 0; i <= sum; i++)
{
    for (int j = 0; j <= n; j++)
        printf ("%d", subset[i][j]);
    printf ("\n");
} */

return subset[sum][n];
}

/* Driver program to test above function */
public static void main (String args[])
{
    int set[] = {3, 34, 4, 12, 5, 2};
    int sum = 9;
    int n = set.length;
    if (isSubsetSum(set, n, sum) == true)
        System.out.println("Found a subset with given sum");
    else
        System.out.println("No subset with given sum");
}
/* This code is contributed by Rajat Mishra */

```

Output:

Found a subset with given sum

Time complexity of the above solution is $O(\text{sum} \times n)$.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Adobe Question
Dynamic Programming

Dynamic Programming | Set 27 (Maximum sum rectangle in a 2D matrix)

Given a 2D array, find the maximum sum subarray in it. For example, in the following 2D array, the maximum sum subarray is highlighted with blue rectangle and sum of this subarray is 29.



This problem is mainly an extension of [Largest Sum Contiguous Subarray for 1D array](#).

The **naive solution** for this problem is to check every possible rectangle in given 2D array. This solution requires 4 nested loops and time complexity of this solution would be $O(n^4)$.

Kadane's algorithm for 1D array can be used to reduce the time complexity to $O(n^3)$. The idea is to fix the left and right columns one by one and find the maximum sum contiguous rows for every left and right column pair. We basically find top and bottom row numbers (which have maximum sum) for every fixed left and right column pair. To find the top and bottom row numbers, calculate sum of elements in every row from left to right and store these sums in an array say temp[]. So temp[i] indicates sum of elements from left to right in row i. If we apply Kadane's 1D algorithm on temp[], and get the maximum sum subarray of temp, this maximum sum would be the maximum possible sum with left and right as boundary columns. To get the overall maximum sum, we compare this sum with the maximum sum so far.

C

```

// Program to find maximum sum subarray in a given 2D array
#include <stdio.h>
#include <string.h>
#include <limits.h>
#define ROW 4
#define COL 5

// Implementation of Kadane's algorithm for 1D array. The function
// returns the maximum sum and stores starting and ending indexes of the
// maximum sum subarray at addresses pointed by start and finish pointers
// respectively.
int kadane(int* arr, int* start, int* finish, int n)
{
    // initialize sum, maxSum and
    int sum = 0, maxSum = INT_MIN, i;

    // Just some initial value to check for all negative values case
    *finish = -1;

    // local variable
    int local_start = 0;

    for (i = 0; i < n; ++i)
    {
        sum += arr[i];
        if (sum < 0)
        {
            sum = 0;
            local_start = i+1;
        }
        else if (sum > maxSum)
        {

```

```

maxSum = sum;
*start = local_start;
*finish = i;
}

// There is at-least one non-negative number
if (*finish != -1)
    return maxSum;

// Special Case: When all numbers in arr[] are negative
maxSum = arr[0];
*start = *finish = 0;

// Find the maximum element in array
for (i = 1; i < n; i++)
{
    if (arr[i] > maxSum)
    {
        maxSum = arr[i];
        *start = *finish = i;
    }
}
return maxSum;
}

// The main function that finds maximum sum rectangle in M[COL]
void findMaxSum(int M[ROW][COL])
{
    // Variables to store the final output
    int maxSum = INT_MIN, finalLeft, finalRight, finalTop, finalBottom;

    int left, right, i;
    int temp[ROW], sum, start, finish;

    // Set the left column
    for (left = 0; left < COL; ++left)
    {
        // Initialize all elements of temp as 0
        memset(temp, 0, sizeof(temp));

        // Set the right column for the left column set by outer loop
        for (right = left; right < COL; ++right)
        {
            // Calculate sum between current left and right for every row 'i'
            for (i = 0; i < ROW; ++i)
                temp[i] += M[i][right];

            // Find the maximum sum subarray in temp[]. The kadane()
            // function also sets values of start and finish. So 'sum' is
            // sum of rectangle between (start, left) and (finish, right)
            // which is the maximum sum with boundary columns strictly as
            // left and right.
            sum = kadane(temp, &start, &finish, ROW);

            // Compare sum with maximum sum so far. If sum is more, then
            // update maxSum and other output values
            if (sum > maxSum)
            {
                maxSum = sum;
                finalLeft = left;
                finalRight = right;
                finalTop = start;
                finalBottom = finish;
            }
        }
    }

    // Print final values
    printf("(Top, Left) (%d, %d)\n", finalTop, finalLeft);
    printf("(Bottom, Right) (%d, %d)\n", finalBottom, finalRight);
    printf("Max sum is: %d\n", maxSum);
}

// Driver program to test above functions
int main()
{
    int M[ROW][COL] = {{1, 2, -1, -4, -20},
                      {-8, -3, 4, 2, 1},
                      {3, 8, 10, 1, 3},
                      {-4, -1, 1, 7, -6}};
    findMaxSum(M);

    return 0;
}

```

Java

```

import java.util.*;
import java.lang.*;
import java.io.*;

/**
 * Given a 2D array, find the maximum sum subarray in it
 */
class Ideone
{
    public static void main (String[] args) throws java.lang.Exception
    {
        findMaxSubMatrix(new int[][]
        {
            {1, 2, -1, -4, -20},
            {-8, -3, 4, 2, 1},
        });
    }
}


```

```

        {3, 8, 10, 1, 3},
        {-4, -1, 1, 7, -6}
    });
}

/**
 * To find maxSum in 1d array
 *
 * return {maxSum, left, right}
 */
public static int[] kadane(int[] a) {
    //result[0] == maxSum, result[1] == start, result[2] == end;
    int[] result = new int[]{Integer.MIN_VALUE, 0, -1};
    int currentSum = 0;
    int localStart = 0;

    for (int i = 0; i < a.length; i++) {
        currentSum += a[i];
        if (currentSum < 0) {
            currentSum = 0;
            localStart = i + 1;
        } else if (currentSum > result[0]) {
            result[0] = currentSum;
            result[1] = localStart;
            result[2] = i;
        }
    }

    //all numbers in a are negative
    if (result[2] == -1) {
        result[0] = 0;
        for (int i = 0; i < a.length; i++) {
            if (a[i] > result[0]) {
                result[0] = a[i];
                result[1] = i;
                result[2] = i;
            }
        }
    }
}

return result;
}

/**
 * To find and print maxSum, (left, top),(right, bottom)
 */
public static void findMaxSubMatrix(int[][] a) {
    int cols = a[0].length;
    int rows = a.length;
    int[] currentResult;
    int maxSum = Integer.MIN_VALUE;
    int left = 0;
    int top = 0;
    int right = 0;
    int bottom = 0;

    for (int leftCol = 0; leftCol < cols; leftCol++) {
        int[] tmp = new int[rows];

        for (int rightCol = leftCol; rightCol < cols; rightCol++) {

            for (int i = 0; i < rows; i++) {
                tmp[i] += a[i][rightCol];
            }
            currentResult = kadane(tmp);
            if (currentResult[0] > maxSum) {
                maxSum = currentResult[0];
                left = leftCol;
                top = currentResult[1];
                right = rightCol;
                bottom = currentResult[2];
            }
        }
        System.out.println("MaxSum: " + maxSum +
            ", range: [" + left + ", " + top +
            ")([" + right + ", " + bottom + "])");
    }
}
// Thanks to Ilya Savin for contributing this code.

```

Output:

```

(Top, Left) (1, 1)
(Bottom, Right) (3, 3)
Max sum is: 29

```

Time Complexity: $O(n^3)$

This article is compiled by [Aashish Barnwal](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Matrix
Dynamic Programming
Matrix

Count number of binary strings without consecutive 1's

Given a positive integer N, count all possible distinct binary strings of length N such that there are no consecutive 1's.

Examples:

```
Input: N = 2
Output: 3
// The 3 strings are 00, 01, 10
```

```
Input: N = 3
Output: 5
// The 5 strings are 000, 001, 010, 100, 101
```

This problem can be solved using Dynamic Programming. Let $a[i]$ be the number of binary strings of length i which do not contain any two consecutive 1's and which end in 0. Similarly, let $b[i]$ be the number of such strings which end in 1. We can append either 0 or 1 to a string ending in 0, but we can only append 0 to a string ending in 1. This yields the recurrence relation:

```
a[i] = a[i - 1] + b[i - 1]
b[i] = a[i - 1]
```

The base cases of above recurrence are $a[1] = b[1] = 1$. The total number of strings of length i is just $a[i] + b[i]$.

Following is the implementation of above solution. In the following implementation, indexes start from 0. So $a[i]$ represents the number of binary strings for input length $i+1$. Similarly, $b[i]$ represents binary strings for input length $i+1$.

C++

```
// C++ program to count all distinct binary strings
// without two consecutive 1's
#include <iostream>
using namespace std;

int countStrings(int n)
{
    int a[n], b[n];
    a[0] = b[0] = 1;
    for (int i = 1; i < n; i++)
    {
        a[i] = a[i-1] + b[i-1];
        b[i] = a[i-1];
    }
    return a[n-1] + b[n-1];
}

// Driver program to test above functions
int main()
{
    cout << countStrings(3) << endl;
    return 0;
}
```

Java

```
class Subset_sum
{
static int countStrings(int n)
{
    int a[] = new int[n];
    int b[] = new int[n];
    a[0] = b[0] = 1;
    for (int i = 1; i < n; i++)
    {
        a[i] = a[i-1] + b[i-1];
        b[i] = a[i-1];
    }
    return a[n-1] + b[n-1];
}
/* Driver program to test above function */
public static void main (String args[])
{
    System.out.println(countStrings(3));
}
/* This code is contributed by Rajat Mishra */
```

Output:

```
5
```

Source:

courses.csail.mit.edu/6.006/oldquizzes/solutions/q2-f2009-sol.pdf

If we take a closer look at the pattern, we can observe that the count is actually $(n+2)$ 'th Fibonacci number for $n \geq 1$. The Fibonacci Numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 141,

```
n = 1, count = 2 = fib(3)
n = 2, count = 3 = fib(4)
n = 3, count = 5 = fib(5)
n = 4, count = 8 = fib(6)
n = 5, count = 13 = fib(7)
.....
```

Therefore we can count the strings in $O(\log n)$ time also using the method 5 [here](#).

This article is contributed by **Rahul Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
binary-string
Dynamic Programming
Fibonacci

Dynamic Programming | Set 37 (Boolean Parenthesization Problem)

Given a boolean expression with following symbols.

Symbols

'T' ---> true
'F' ---> false

And following operators filled between symbols

Operators

& ---> boolean AND
| ---> boolean OR
^ ---> boolean XOR

Count the number of ways we can parenthesize the expression so that the value of expression evaluates to true.

Let the input be in form of two arrays one contains the symbols (T and F) in order and other contains operators (&, | and ^)

Examples:

Input: symbol[] = {T, F, T}
operator[] = {'^', '&'}
Output: 2

The given expression is "T ^ F & T", it evaluates true
in two ways "(T ^ F) & T" and "(T ^ (F & T))"

Input: symbol[] = {T, F, F}
operator[] = {'^', '|'}
Output: 2

The given expression is "T ^ F | F", it evaluates true
in two ways "(T ^ F) | F" and "(T ^ (F | F))".

Input: symbol[] = {T, T, F, T}
operator[] = {'|', '&', '^'}
Output: 4

The given expression is "T | T & F ^ T", it evaluates true
in 4 ways ((T|T)&(F^T)), (T|(T&(F^T))), (((T|T)&F)^T)
and (T|((T&F)^T)).

Solution:

Let $T(i, j)$ represents the number of ways to parenthesize the symbols between i and j (both inclusive) such that the subexpression between i and j evaluates to true.

trueeq



Let $F(i, j)$ represents the number of ways to parenthesize the symbols between i and j (both inclusive) such that the subexpression between i and j evaluates to false.

falseeq



Base Cases:

$T(i, i) = 1$ if $\text{symbol}[i] = \text{T}$
 $T(i, i) = 0$ if $\text{symbol}[i] = \text{F}$

$F(i, i) = 1$ if $\text{symbol}[i] = \text{F}$
 $F(i, i) = 0$ if $\text{symbol}[i] = \text{T}$

If we draw recursion tree of above recursive solution, we can observe that it many overlapping subproblems. Like other dynamic programming problems, it can be solved by filling a table in bottom up manner. Following is C++ implementation of dynamic programming solution.

```
#include<iostream>
#include<cstring>
using namespace std;

// Returns count of all possible parenthesizations that lead to
// result true for a boolean expression with symbols like true
// and false and operators like &, | and ^ filled between symbols
int countParenth(char symb[], char oper[], int n)
{
    int F[n][n], T[n][n];

    // Fill diagonal entries first
    // All diagonal entries in T[i][i] are 1 if symbol[i]
    // is T (true). Similarly, all F[i][i] entries are 1 if
    // symbol[i] is F (False)
    for (int i = 0; i < n; i++)
    {
        F[i][i] = (symb[i] == 'F')? 1: 0;
        T[i][i] = (symb[i] == 'T')? 1: 0;
    }

    // Now fill T[i][i+1], T[i][i+2], T[i][i+3]... in order
    // And F[i][i+1], F[i][i+2], F[i][i+3]... in order
    for (int gap=1; gap<n; ++gap)
    {
        for (int i=0, j=gap; j<n; ++i, ++j)
        {
            T[i][j] = F[i][j] = 0;
            for (int g=0; g<gap; g++)
            {
                // Find place of parenthesization using current value
                // of gap
                int k = i + g;

                // Store Total[i][k] and Total[k+1][j]
                int tik = T[i][k] + F[i][k];
                int tkj = T[k+1][j] + F[k+1][j];
            }
        }
    }
}
```

```

// Follow the recursive formulas according to the current
// operator
if (oper[k] == '&')
{
    T[i][j] += T[i][k]*T[k+1][j];
    F[i][j] += (tk*tkj - T[i][k]*T[k+1][j]);
}
if (oper[k] == '|')
{
    F[i][j] += F[i][k]*F[k+1][j];
    T[i][j] += (tk*tkj - F[i][k]*F[k+1][j]);
}
if (oper[k] == '^')
{
    T[i][j] += F[i][k]*T[k+1][j] + T[i][k]*F[k+1][j];
    F[i][j] += T[i][k]*T[k+1][j] + F[i][k]*F[k+1][j];
}
}
}
return T[0][n-1];
}

// Driver program to test above function
int main()
{
    char symbols[] = "TTFT";
    char operators[] = "&^|";
    int n = strlen(symbols);

    // There are 4 ways
    // ((T|T)&(F^T)), (T|(T&(F^T))), (((T|T)&F)^T) and (T|((T&F)^T))
    cout << countParenth(symbols, operators, n);
    return 0;
}

```

Output:

4

Time Complexity: $O(n^3)$
Auxiliary Space: $O(n^2)$

References:

http://people.cs.clemson.edu/~bcdean/dp_practice/dp_9.swf

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Dynamic Programming

Count ways to reach the n'th stair

There are n stairs, a person standing at the bottom wants to reach the top. The person can climb either 1 stair or 2 stairs at a time. Count the number of ways, the person can reach the top.



Consider the example shown in diagram. The value of n is 3. There are 3 ways to reach the top. The diagram is taken from [Easier Fibonacci puzzles](#)

More Examples:

```

Input: n = 1
Output: 1
There is only one way to climb 1 stair

Input: n = 2
Output: 2
There are two ways: (1, 1) and (2)

Input: n = 4
Output: 5
(1, 1, 1, 1), (1, 1, 2), (2, 1, 1), (1, 2, 1), (2, 2)

```

We strongly recommend that you click here and practice it, before moving on to the solution.

We can easily find recursive nature in above problem. The person can reach n'th stair from either (n-1)'th stair or from (n-2)'th stair. Let the total number of ways to reach n't stair be 'ways(n)'. The value of 'ways(n)' can be written as following.

```
ways(n) = ways(n-1) + ways(n-2)
```

The above expression is actually the expression for [Fibonacci numbers](#), but there is one thing to notice, the value of ways(n) is equal to fibonacci(n+1).

```
ways(1) = fib(2) = 1  
ways(2) = fib(3) = 2  
ways(3) = fib(4) = 3
```

So we can use function for fibonacci numbers to find the value of ways(n). Following is C++ implementation of the above idea.

C

```
// A C program to count number of ways to reach n'th stair when  
// a person can climb 1, 2, ..m stairs at a time.  
#include<stdio.h>  
  
// A simple recursive program to find n'th fibonacci number  
int fib(int n)  
{  
    if (n <= 1)  
        return n;  
    return fib(n-1) + fib(n-2);  
}  
  
// Returns number of ways to reach s'th stair  
int countWays(int s)  
{  
    return fib(s + 1);  
}  
  
// Driver program to test above functions  
int main ()  
{  
    int s = 4;  
    printf("Number of ways = %d", countWays(s));  
    getchar();  
    return 0;  
}
```

Java

```
class stairs  
{  
    // A simple recursive program to find n'th fibonacci number  
    static int fib(int n)  
{  
        if (n <= 1)  
            return n;  
        return fib(n-1) + fib(n-2);  
    }  
  
    // Returns number of ways to reach s'th stair  
    static int countWays(int s)  
{  
        return fib(s + 1);  
    }  
  
    /* Driver program to test above function */  
    public static void main (String args[])  
    {  
        int s = 4;  
        System.out.println("Number of ways = " + countWays(s));  
    }  
    /* This code is contributed by Rajat Mishra */
```

Output:

```
Number of ways = 5
```

The time complexity of the above implementation is exponential (golden ratio raised to power n). It can be optimized to work in O(Logn) time using the previously [discussed](#) Fibonacci function optimizations.

Generalization of the above problem

How to count number of ways if the person can climb up to m stairs for a given value m? For example if m is 4, the person can climb 1 stair or 2 stairs or 3 stairs or 4 stairs at a time.

We can write the recurrence as following.

```
ways(n, m) = ways(n-1, m) + ways(n-2, m) + ... ways(n-m, m)
```

Following is C++ implementation of above recurrence.

C

```
// A C program to count number of ways to reach n'th stair when  
// a person can climb either 1 or 2 stairs at a time  
#include<stdio.h>  
  
// A recursive function used by countWays  
int countWaysUtil(int n, int m)  
{  
    if (n <= 1)  
        return n;  
    int res = 0;  
    for (int i = 1; i <= m && i <= n; i++)  
        res += countWaysUtil(n-i, m);  
    return res;  
}
```

```

// Returns number of ways to reach s'th stair
int countWays(int s, int m)
{
    return countWaysUtil(s+1, m);
}

// Driver program to test above functions
int main ()
{
    int s = 4, m = 2;
    printf("Nuber of ways = %d", countWays(s, m));
    return 0;
}

```

Java

```

class stairs
{
    // A recursive function used by countWays
    static int countWaysUtil(int n, int m)
    {
        if (n <= 1)
            return n;
        int res = 0;
        for (int i = 1; i<=m && i<=n; i++)
            res += countWaysUtil(n-i, m);
        return res;
    }

    // Returns number of ways to reach s'th stair
    static int countWays(int s, int m)
    {
        return countWaysUtil(s+1, m);
    }

    /* Driver program to test above function */
    public static void main (String args[])
    {
        int s = 4, m = 2;
        System.out.println("Number of ways = "+ countWays(s,m));
    }
}
/* This code is contributed by Rajat Mishra */

```

Output:

```
Number of ways = 5
```

The time complexity of above solution is exponential. It can be optimized to $O(mn)$ by using dynamic programming. Following is dynamic programming based solution. We build a table `res[]` in bottom up manner.

C

```

// A C program to count number of ways to reach n'th stair when
// a person can climb 1, 2, ..m stairs at a time
#include<stdio.h>

// A recursive function used by countWays
int countWaysUtil(int n, int m)
{
    int res[n];
    res[0] = 1; res[1] = 1;
    for (int i=2; i<n; i++)
    {
        res[i] = 0;
        for (int j=1; j<=m && j<=i; j++)
            res[i] += res[i-j];
    }
    return res[n-1];
}

// Returns number of ways to reach s'th stair
int countWays(int s, int m)
{
    return countWaysUtil(s+1, m);
}

// Driver program to test above functions
int main ()
{
    int s = 4, m = 2;
    printf("Nuber of ways = %d", countWays(s, m));
    return 0;
}

```

Java

```

class stairs
{
    // A recursive function used by countWays
    static int countWaysUtil(int n, int m)
    {
        if (n <= 1)
            return n;
        int res = 0;
        for (int i = 1; i<=m && i<=n; i++)
            res += countWaysUtil(n-i, m);
        return res;
    }
}

```

```

    return res;
}
// Returns number of ways to reach s'th stair
static int countWays(int s, int m)
{
    return countWaysUtil(s+1, m);
}

/* Driver program to test above function */
public static void main (String args[])
{
    int s = 4,m = 2;
    System.out.println("Number of ways = "+ countWays(s,m));
}
/* This code is contributed by Rajat Mishra */

```

Output:

Number of ways = 5

This article is contributed by **Abhishek**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Mathematical

Minimum Cost Polygon Triangulation

A triangulation of a convex polygon is formed by drawing diagonals between non-adjacent vertices (corners) such that the diagonals never intersect. The problem is to find the cost of triangulation with the minimum cost. The cost of a triangulation is sum of the weights of its component triangles. Weight of each triangle is its perimeter (sum of lengths of all sides)

See following example taken from [this](#) source.

[PolynomialTriang](#)



Two triangulations of the same convex pentagon. The triangulation on the left has a cost of $8 + 2\sqrt{2} + 2\sqrt{5}$ (approximately 15.30), the one on the right has a cost of $4 + 2\sqrt{2} + 4\sqrt{5}$ (approximately 15.77).

This problem has recursive substructure. The idea is to divide the polygon into three parts: a single triangle, the sub-polygon to the left, and the sub-polygon to the right. We try all possible divisions like this and find the one that minimizes the cost of the triangle plus the cost of the triangulation of the two sub-polygons.

Let Minimum Cost of triangulation of vertices from i to j be minCost(i, j)
If j
Following is C++ implementation of above naive recursive formula.

```

// Recursive implementation for minimum cost convex polygon triangulation
#include <iostream>
#include <cmath>
#define MAX 1000000.0
using namespace std;

// Structure of a point in 2D plane
struct Point
{
    int x, y;
};

// Utility function to find minimum of two double values
double min(double x, double y)
{
    return (x <= y)? x : y;
}

// A utility function to find distance between two points in a plane
double dist(Point p1, Point p2)
{
    return sqrt((p1.x - p2.x)*(p1.x - p2.x) +
               (p1.y - p2.y)*(p1.y - p2.y));
}

// A utility function to find cost of a triangle. The cost is considered
// as perimeter (sum of lengths of all edges) of the triangle
double cost(Point points[], int i, int j, int k)
{
    Point p1 = points[i], p2 = points[j], p3 = points[k];
    return dist(p1, p2) + dist(p2, p3) + dist(p3, p1);
}

// A recursive function to find minimum cost of polygon triangulation
// The polygon is represented by points[..].
double mTC(Point points[], int i, int j)
{
    // There must be at least three points between i and j
    // (including i and j)
    if (j < i+2)
        return 0;

    // Initialize result as infinite
    double res = MAX;

    // Find minimum triangulation by considering all
    for (int k=i+1; k<j; k++)
        res = min(res, (mTC(points, i, k) + mTC(points, k, j)) +
                  cost(points, i, k, j)));
}

return res;

```

```

}

// Driver program to test above functions
int main()
{
    Point points[] = {{0, 0}, {1, 0}, {2, 1}, {1, 2}, {0, 2}};
    int n = sizeof(points)/sizeof(points[0]);
    cout << mTC(points, 0, n-1);
    return 0;
}

```

Output:

15.3006

The above problem is similar to [Matrix Chain Multiplication](#). The following is recursion tree for mTC(points[], 0, 4).

[polyTriang](#)



It can be easily seen in the above recursion tree that the problem has many overlapping subproblems. Since the problem has both properties: [Optimal Substructure](#) and [Overlapping Subproblems](#), it can be efficiently solved using dynamic programming.

Following is C++ implementation of dynamic programming solution.

```

// A Dynamic Programming based program to find minimum cost of convex
// polygon triangulation
#include <iostream>
#include <cmath>
#define MAX 1000000.0
using namespace std;

// Structure of a point in 2D plane
struct Point
{
    int x, y;
};

// Utility function to find minimum of two double values
double min(double x, double y)
{
    return (x <= y)? x : y;
}

// A utility function to find distance between two points in a plane
double dist(Point p1, Point p2)
{
    return sqrt((p1.x - p2.x)*(p1.x - p2.x) +
                (p1.y - p2.y)*(p1.y - p2.y));
}

// A utility function to find cost of a triangle. The cost is considered
// as perimeter (sum of lengths of all edges) of the triangle
double cost(Point points[], int i, int j, int k)
{
    Point p1 = points[i], p2 = points[j], p3 = points[k];
    return dist(p1, p2) + dist(p2, p3) + dist(p3, p1);
}

// A Dynamic programming based function to find minimum cost for convex
// polygon triangulation.
double mTCDP(Point points[], int n)
{
    // There must be at least 3 points to form a triangle
    if (n < 3)
        return 0;

    // table to store results of subproblems. table[i][j] stores cost of
    // triangulation of points from i to j. The entry table[0][n-1] stores
    // the final result.
    double table[n][n];

    // Fill table using above recursive formula. Note that the table
    // is filled in diagonal fashion i.e., from diagonal elements to
    // table[0][n-1] which is the result.
    for (int gap = 0; gap < n; gap++)
    {
        for (int i = 0, j = gap; j < n; i++, j++)
        {
            if (j < i+2)
                table[i][j] = 0.0;
            else
            {
                table[i][j] = MAX;
                for (int k = i+1; k < j; k++)
                {
                    double val = table[i][k] + table[k][j] + cost(points,i,j,k);
                    if (table[i][j] > val)
                        table[i][j] = val;
                }
            }
        }
    }
    return table[0][n-1];
}

```

```

        }
    }
}
return table[0][n-1];
}

// Driver program to test above functions
int main()
{
    Point points[] = {{0, 0}, {1, 0}, {2, 1}, {1, 2}, {0, 2}};
    int n = sizeof(points)/sizeof(points[0]);
    cout << mTCDP(points, n);
    return 0;
}

```

Output:

15.3006

Time complexity of the above dynamic programming solution is $O(n^3)$.

Please note that the above implementations assume that the points of convex polygon are given in order (either clockwise or anticlockwise)

Exercise:

Extend the above solution to print triangulation also. For the above example, the optimal triangulation is 0 3 4, 0 1 3, and 1 2 3.

Sources:

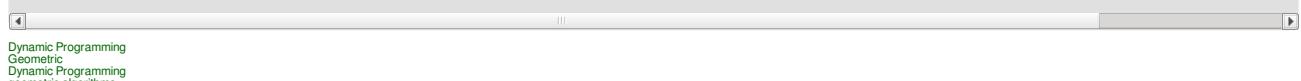
<http://www.cs.utexas.edu/users/djimenez/utsa/cs3343/lecture12.html>

<http://www.cs.utoronto.ca/~heap/Courses/270F02/A4/chains/node2.html>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner

Company Wise Coding Practice



Mobile Numeric Keypad Problem

mobile1



Given the mobile numeric keypad. You can only press buttons that are up, left, right or down to the current button. You are not allowed to press bottom row corner buttons

(i.e. * and #).

Given a number N, find out the number of possible numbers of given length.

Examples:

For N=1, number of possible numbers would be 10 (0, 1, 2, 3,, 9)

For N=2, number of possible numbers would be 36

Possible numbers: 00,08,11,12,14,22,21,23,25 and so on.

If we start with 0, valid numbers will be 00, 08 (count: 2)

If we start with 1, valid numbers will be 11, 12, 14 (count: 3)

If we start with 2, valid numbers will be 22, 21, 23,25 (count: 4)

If we start with 3, valid numbers will be 33, 32, 36 (count: 3)

If we start with 4, valid numbers will be 44,41,45,47 (count: 4)

If we start with 5, valid numbers will be 55,54,52,56,58 (count: 5)

.....

.....

We need to print the count of possible numbers.

We strongly recommend to minimize the browser and try this yourself first.

$N = 1$ is trivial case, number of possible numbers would be 10 (0, 1, 2, 3, ..., 9)

For $N > 1$, we need to start from some button, then move to any of the four direction (up, left, right or down) which takes to a valid button (should not go to *, #). Keep doing this until N length number is obtained (depth first traversal).

Recursive Solution:

Mobile Keypad is a rectangular grid of 4×3 (4 rows and 3 columns)

Lets say $\text{Count}(i, j, N)$ represents the count of N length numbers starting from position (i, j)

```
If N = 1  
    Count(i, j, N) = 10  
Else  
    Count(i, j, N) = Sum of all Count(r, c, N-1) where (r, c) is new  
        position after valid move of length 1 from current  
        position (i, j)
```

Following is C implementation of above recursive formula.

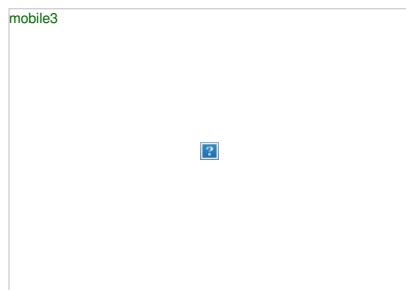
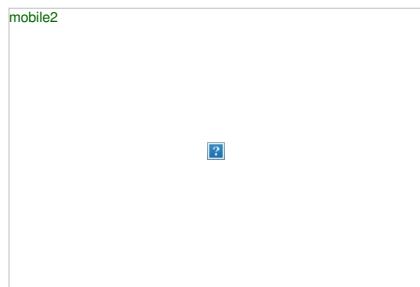
```
// A Naive Recursive C program to count number of possible numbers  
// of given length  
#include <stdio.h>  
  
// left, up, right, down move from current location  
int row[] = {0, 0, -1, 0, 1};  
int col[] = {0, -1, 0, 1, 0};  
  
// Returns count of numbers of length n starting from key position  
// (i, j) in a numeric keyboard.  
int getCountUtil(char keypad[4][3], int i, int j, int n)  
{  
    if (keypad == NULL || n <= 0)  
        return 0;  
  
    // From a given key, only one number is possible of length 1  
    if (n == 1)  
        return 1;  
  
    int k=0, move=0, ro=0, co=0, totalCount = 0;  
  
    // move left, up, right, down from current location and if  
    // new location is valid, then get number count of length  
    // (n-1) from that new position and add in count obtained so far  
    for (move=0; move<5; move++)  
    {  
        ro = i + row[move];  
        co = j + col[move];  
        if (ro >= 0 && ro <= 3 && co >= 0 && co <= 2 &&  
            keypad[ro][co] != '*' && keypad[ro][co] != '#')  
        {  
            totalCount += getCountUtil(keypad, ro, co, n-1);  
        }  
    }  
  
    return totalCount;  
}  
  
// Return count of all possible numbers of length n  
// in a given numeric keyboard  
int getCount(char keypad[4][3], int n)  
{  
    // Base cases  
    if (keypad == NULL || n <= 0)  
        return 0;  
    if (n == 1)  
        return 10;  
  
    int i=0, j=0, totalCount = 0;  
    for (i=0; i<4; i++) // Loop on keypad row  
    {  
        for (j=0; j<3; j++) // Loop on keypad column  
        {  
            // Process for 0 to 9 digits  
            if (keypad[i][j] != '*' && keypad[i][j] != '#')  
            {  
                // Get count when number is starting from key  
                // position (i, j) and add in count obtained so far  
                totalCount += getCountUtil(keypad, i, j, n);  
            }  
        }  
    }  
    return totalCount;  
}  
  
// Driver program to test above function  
int main(int argc, char *argv[]){  
    char keypad[4][3] = {{'1','2','3'},  
                        {'4','5','6'},  
                        {'7','8','9'},  
                        {'*','0','#'}};  
    printf("Count for numbers of length %d: %d\n", 1, getCount(keypad, 1));  
    printf("Count for numbers of length %d: %d\n", 2, getCount(keypad, 2));  
    printf("Count for numbers of length %d: %d\n", 3, getCount(keypad, 3));  
    printf("Count for numbers of length %d: %d\n", 4, getCount(keypad, 4));  
    printf("Count for numbers of length %d: %d\n", 5, getCount(keypad, 5));  
  
    return 0;  
}
```

Output:

```
Count for numbers of length 1: 10  
Count for numbers of length 2: 36  
Count for numbers of length 3: 138  
Count for numbers of length 4: 532  
Count for numbers of length 5: 2062
```

Dynamic Programming

There are many repeated traversal on smaller paths (traversal for smaller N) to find all possible longer paths (traversal for bigger N). See following two diagrams for example. In this traversal, for N = 4 from two starting positions (buttons '4' and '8'), we can see there are few repeated traversals for N = 2 (e.g. 4 -> 1, 6 -> 3, 8 -> 9, 8 -> 7 etc).



Since the problem has both properties: [Optimal Substructure](#) and [Overlapping Subproblems](#), it can be efficiently solved using dynamic programming.

Following is C program for dynamic programming implementation.

```
// A Dynamic Programming based C program to count number of
// possible numbers of given length
#include <stdio.h>

// Return count of all possible numbers of length n
// in a given numeric keyboard
int getCount(char keypad[10][3], int n)
{
    if(keypad == NULL || n <= 0)
        return 0;
    if(n == 1)
        return 10;

    // left, up, right, down move from current location
    int row[] = {0, 0, -1, 0, 1};
    int col[] = {0, -1, 0, 1, 0};

    // taking n+1 for simplicity - count[i][j] will store
    // number count starting with digit i and length j
    int count[10][n+1];
    int i=0, j=0, k=0, move=0, ro=0, co=0, num = 0;
    int nextNum=0, totalCount = 0;

    // count numbers starting with digit i and of lengths 0 and 1
    for (i=0; i<=9; i++)
    {
        count[i][0] = 0;
        count[i][1] = 1;
    }

    // Bottom up - Get number count of length 2, 3, 4, ..., n
    for (k=2; k<=n; k++)
    {
        for (i=0; i<4; i++) // Loop on keypad row
        {
            for (j=0; j<3; j++) // Loop on keypad column
            {
                // Process for 0 to 9 digits
                if (keypad[i][j] != '*' && keypad[i][j] != '#')
                {
                    // Here we are counting the numbers starting with
                    // digit keypad[i][j] and of length k keypad[i][j]
                    // will become 1st digit, and we need to look for
                    // (k-1) more digits
                    num = keypad[i][j] - '0';
                    count[num][k] = 0;

                    // move left, up, right, down from current location
                    // and if new location is valid, then get number
                    // count of length (k-1) from that new digit and
                    // add in count we found so far
                    for (move=0; move<5; move++)
                    {
                        ro = i + row[move];
                        co = j + col[move];
                        if (ro >= 0 && ro <= 3 && co >= 0 && co <= 2 &&
                            keypad[ro][co] != '*' && keypad[ro][co] != '#')
                        {
                            nextNum = keypad[ro][co] - '0';
                            count[num][k] += count[nextNum][k-1];
                        }
                    }
                }
            }
        }
    }
}
```

```

// Get count of all possible numbers of length "n" starting
// with digit 0, 1, 2, ..., 9
totalCount = 0;
for (i=0; i<=9; i++)
    totalCount += count[i][n];
return totalCount;
}

// Driver program to test above function
int main(int argc, char *argv[])
{
    char keypad[4][3] = {{'1','2','3'},
                         {'4','5','6'},
                         {'7','8','9'},
                         {'*','0','#'}};
    printf("Count for numbers of length %d: %d\n", 1, getCount(keypad, 1));
    printf("Count for numbers of length %d: %d\n", 2, getCount(keypad, 2));
    printf("Count for numbers of length %d: %d\n", 3, getCount(keypad, 3));
    printf("Count for numbers of length %d: %d\n", 4, getCount(keypad, 4));
    printf("Count for numbers of length %d: %d\n", 5, getCount(keypad, 5));

    return 0;
}

```

Output:

```

Count for numbers of length 1: 10
Count for numbers of length 2: 36
Count for numbers of length 3: 138
Count for numbers of length 4: 532
Count for numbers of length 5: 2062

```

A Space Optimized Solution:

The above dynamic programming approach also runs in O(n) time and requires O(n) auxiliary space, as only one for loop runs n times, other for loops runs for constant time. We can see that nth iteration needs data from (n-1)th iteration only, so we need not keep the data from older iterations. We can have a space efficient dynamic programming approach with just two arrays of size 10. Thanks to Nik for suggesting this solution.

```

// A Space Optimized C program to count number of possible numbers
// of given length
#include <stdio.h>

// Return count of all possible numbers of length n
// in a given numeric keyboard
int getCount(char keypad[][3], int n)
{
    if(keypad == NULL || n <= 0)
        return 0;
    if(n == 1)
        return 10;

    // odd[], even[] arrays represent count of numbers starting
    // with digit i for any length j
    int odd[10], even[10];
    int i = 0, j = 0, useOdd = 0, totalCount = 0;

    for (i=0; i<=9; i++)
        odd[i] = 1; // for j = 1

    for (j=2; j<=n; j++) // Bottom Up calculation from j = 2 to n
    {
        useOdd = 1 - useOdd;

        // Here we are explicitly writing lines for each number 0
        // to 9. But it can always be written as DFS on 4X3 grid
        // using row, column array valid moves
        if(useOdd == 1)
        {
            even[0] = odd[0] + odd[8];
            even[1] = odd[1] + odd[2] + odd[4];
            even[2] = odd[2] + odd[1] + odd[3] + odd[5];
            even[3] = odd[3] + odd[2] + odd[6];
            even[4] = odd[4] + odd[1] + odd[5] + odd[7];
            even[5] = odd[5] + odd[2] + odd[4] + odd[8] + odd[6];
            even[6] = odd[6] + odd[3] + odd[5] + odd[9];
            even[7] = odd[7] + odd[4] + odd[8];
            even[8] = odd[8] + odd[0] + odd[5] + odd[7] + odd[9];
            even[9] = odd[9] + odd[6] + odd[8];
        }
        else
        {
            odd[0] = even[0] + even[8];
            odd[1] = even[1] + even[2] + even[4];
            odd[2] = even[2] + even[1] + even[3] + even[5];
            odd[3] = even[3] + even[2] + even[6];
            odd[4] = even[4] + even[1] + even[5] + even[7];
            odd[5] = even[5] + even[2] + even[4] + even[8] + even[6];
            odd[6] = even[6] + even[3] + even[5] + even[9];
            odd[7] = even[7] + even[4] + even[8];
            odd[8] = even[8] + even[0] + even[5] + even[7] + even[9];
            odd[9] = even[9] + even[6] + even[8];
        }
    }

    // Get count of all possible numbers of length "n" starting
    // with digit 0, 1, 2, ..., 9
    totalCount = 0;
    if(useOdd == 1)
    {
        for (i=0; i<=9; i++)
            totalCount += even[i];
    }
    else
    {
        for (i=0; i<=9; i++)
    }
}
```

```

        totalCount += odd[i];
    }
    return totalCount;
}

// Driver program to test above function
int main()
{
    char keypad[4][3] = {{'1','2','3'},
                         {'4','5','6'},
                         {'7','8','9'},
                         {'*','0','#'}
    };
    printf("Count for numbers of length %d\n", 1, getCount(keypad, 1));
    printf("Count for numbers of length %d\n", 2, getCount(keypad, 2));
    printf("Count for numbers of length %d\n", 3, getCount(keypad, 3));
    printf("Count for numbers of length %d\n", 4, getCount(keypad, 4));
    printf("Count for numbers of length %d\n", 5, getCount(keypad, 5));

    return 0;
}

```

Output:

```

Count for numbers of length 1:10
Count for numbers of length 2:36
Count for numbers of length 3:138
Count for numbers of length 4:532
Count for numbers of length 5:2062

```

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Matrix
Dynamic Programming
Matrix

Count of n digit numbers whose sum of digits equals to given sum

Given two integers 'n' and 'sum', find count of all n digit numbers with sum of digits as 'sum'. Leading 0's are not counted as digits.

1

Example:

```

Input: n = 2, sum = 2
Output: 2
Explanation: Numbers are 11 and 20

Input: n = 2, sum = 5
Output: 5
Explanation: Numbers are 14, 23, 32, 41 and 50

Input: n = 3, sum = 6
Output: 21

```

We strongly recommend that you click here and practice it, before moving on to the solution.

The idea is simple, we subtract all values from 0 to 9 from given sum and recur for sum minus that digit. Below is recursive formula.

```

countRec(n, sum) = ΣfinalCount(n-1, sum-x)
                  where 1 == 0
One important observation is, leading 0's must be
handled explicitly as they are not counted as digits.
So our final count can be written as below.

finalCount(n, sum) = ΣfinalCount(n-1, sum-x)
                  where 0 == 0

```

Below is a simple recursive solution based on above recursive formula.

C++

```

// A recursive program to count numbers with sum
// of digits as given 'sum'
#include<bits/stdc++.h>
using namespace std;

// Recursive function to count 'n' digit numbers
// with sum of digits as 'sum'. This function
// considers leading 0's also as digits, that is
// why not directly called
unsigned long long int countRec(int n, int sum)
{
    // Base case
    if (n == 0)
        return sum == 0;

    // Initialize answer
    unsigned long long int ans = 0;

    // Traverse through every digit and count
    // numbers beginning with it using recursion
    for (int i=0; i<=9; i++)
        if (sum-i >= 0)
            ans += countRec(n-1, sum-i);
}

```

```

    return ans;
}

// This is mainly a wrapper over countRec. It
// explicitly handles leading digit and calls
// countRec() for remaining digits.
unsigned long long int finalCount(int n, int sum)
{
    // Initialize final answer
    unsigned long long int ans = 0;

    // Traverse through every digit from 1 to
    // 9 and count numbers beginning with it
    for (int i = 1; i <= 9; i++)
        if (sum-i >= 0)
            ans += countRec(n-1, sum-i);

    return ans;
}

// Driver program
int main()
{
    int n = 2, sum = 5;
    cout << finalCount(n, sum);
    return 0;
}

```

Java

```

class sum_dig
{
    // Recursive function to count 'n' digit numbers
    // with sum of digits as 'sum'. This function
    // considers leading 0's also as digits, that is
    // why not directly called
    static int countRec(int n, int sum)
    {
        // Base case
        if (n == 0)
            return sum == 0 ? 1 : 0;

        // Initialize answer
        int ans = 0;

        // Traverse through every digit and count
        // numbers beginning with it using recursion
        for (int i=0; i<=9; i++)
            if (sum-i >= 0)
                ans += countRec(n-1, sum-i);

        return ans;
    }

    // This is mainly a wrapper over countRec. It
    // explicitly handles leading digit and calls
    // countRec() for remaining digits.
    static int finalCount(int n, int sum)
    {
        // Initialize final answer
        int ans = 0;

        // Traverse through every digit from 1 to
        // 9 and count numbers beginning with it
        for (int i = 1; i <= 9; i++)
            if (sum-i >= 0)
                ans += countRec(n-1, sum-i);

        return ans;
    }
}

/* Driver program to test above function */
public static void main (String args[])
{
    int n = 2, sum = 5;
    System.out.println(finalCount(n, sum));
}
/* This code is contributed by Rajat Mishra */

```

Output:

5

The time complexity of above solution is exponential. If we draw the complete recursion tree, we can observe that many subproblems are solved again and again. For example, if we start with $n = 3$ and $sum = 10$, we can reach $n = 1$, $sum = 8$, by considering digit sequences 1,1 or 2, 0.

Since same subproblems are called again, this problem has Overlapping Subproblems property. So min square sum problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem.

Below is Memoization based the implementation.

C++

```

// A memoization based recursive program to count
// numbers with sum of n as given 'sum'
#include<bits/stdc++.h>
using namespace std;

// A lookup table used for memoization
unsigned long long int lookup[101][50001];

// Memoization based implementation of recursive

```

```

// function
unsigned long long int countRec(int n, int sum)
{
    // Base case
    if (n == 0)
        return sum == 0;

    // If this subproblem is already evaluated,
    // return the evaluated value
    if (lookup[n][sum] != -1)
        return lookup[n][sum];

    // Initialize answer
    unsigned long long int ans = 0;

    // Traverse through every digit and
    // recursively count numbers beginning
    // with it
    for (int i=0; i<10; i++)
        if (sum-i >= 0)
            ans += countRec(n-1, sum-i);

    return lookup[n][sum] = ans;
}

// This is mainly a wrapper over countRec. It
// explicitly handles leading digit and calls
// countRec() for remaining n.
unsigned long long int finalCount(int n, int sum)
{
    // Initialize all entries of lookup table
    memset(lookup, -1, sizeof(lookup));

    // Initialize final answer
    unsigned long long int ans = 0;

    // Traverse through every digit from 1 to
    // 9 and count numbers beginning with it
    for (int i = 1; i <= 9; i++)
        if (sum-i >= 0)
            ans += countRec(n-1, sum-i);
    return ans;
}

// Driver program
int main()
{
    int n = 3, sum = 5;
    cout << finalCount(n, sum);
    return 0;
}

```

Java

```

class sum_dig
{
    // A lookup table used for memoization
    static int lookup[][] = new int[101][50001];

    // Memoizatiob based implementation of recursive
    // function
    static int countRec(int n, int sum)
    {
        // Base case
        if (n == 0)
            return sum == 0 ? 1 : 0;

        // If this subproblem is already evaluated,
        // return the evaluated value
        if (lookup[n][sum] != -1)
            return lookup[n][sum];

        // Initialize answer
        int ans = 0;

        // Traverse through every digit and
        // recursively count numbers beginning
        // with it
        for (int i=0; i<10; i++)
            if (sum-i >= 0)
                ans += countRec(n-1, sum-i);

        return lookup[n][sum] = ans;
    }

    // This is mainly a wrapper over countRec. It
    // explicitly handles leading digit and calls
    // countRec() for remaining n.
    static int finalCount(int n, int sum)
    {
        // Initialize all entries of lookup table
        for(int i = 0;i<=100;+i){
            for(int j=0;j<=50000;+j){
                lookup[i][j] = -1;
            }
        }
    }

    // Initialize final answer
    int ans = 0;

    // Traverse through every digit from 1 to
    // 9 and count numbers beginning with it
    for (int i = 1; i <= 9; i++)
        if (sum-i >= 0)

```

```

    ans += countRec(n-1, sum-i);
}
/* Driver program to test above function */
public static void main (String args[])
{
    int n = 2, sum = 5;
    System.out.println(finalCount(n, sum));
}
/* This code is contributed by Rajat Mishra */

```

Output:

5

Thanks to [Gaurav Ahirwar](#) for suggesting above solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Mathematical
Dynamic Programming
number-digits

Minimum Initial Points to Reach Destination

Given a grid with each cell consisting of positive, negative or no points i.e., zero points. We can move across a cell only if we have positive points (> 0). Whenever we pass through a cell, points in that cell are added to our overall points. We need to find minimum initial points to reach cell $(m-1, n-1)$ from $(0, 0)$.

Constraints :

- From a cell (i, j) we can move to $(i+1, j)$ or $(i, j+1)$.
- We cannot move from (i, j) if your overall points at (i, j) is
- We have to reach at $(n-1, m-1)$ with minimum positive points i.e., > 0 .

Example:

```

Input: points[m][n] = { {-2, -3, 3},
                      {-5, -10, 1},
                      {10, 30, -5}
                    };

```

Output: 7

Explanation:

7 is the minimum value to reach destination with positive throughout the path. Below is the path.

$(0,0) \rightarrow (0,1) \rightarrow (0,2) \rightarrow (1,2) \rightarrow (2,2)$

We start from $(0, 0)$ with 7, we reach $(0, 1)$ with 5, $(0, 2)$ with 2, $(1, 2)$ with 5, $(2, 2)$ with and finally we have 1 point (we needed greater than 0 points at the end).

We strongly recommend that you click here and practice it, before moving on to the solution.

At the first look, this problem looks similar [Max/Min Cost Path](#), but maximum overall points gained will not guarantee the minimum initial points. Also, it is compulsory in the current problem that the points never drops to zero or below. For instance, Suppose following two paths exists from source to destination cell.

We can solve this problem through bottom-up table filling dynamic programming technique.

- To begin with, we should maintain a 2D array dp of the same size as the grid, where $dp[i][j]$ represents the minimum points that guarantees the continuation of the journey to destination before entering the cell (i, j) . It's but obvious that $dp[0][0]$ is our final solution. Hence, for this problem, we need to fill the table from the bottom right corner to left top.
- Now, let us decide minimum points needed to leave cell (i, j) (remember we are moving from bottom to up). There are only two paths to choose: $(i+1, j)$ and $(i, j+1)$. Of course we will choose the cell that the player can finish the rest of his journey with a smaller initial points. Therefore we have: $\text{min_Points_on_exit} = \min(dp[i+1][j], dp[i][j+1])$

Now we know how to compute $\text{min_Points_on_exit}$, but we need to fill the table $dp[][],$ to get the solution in $dp[0][0]$.

How to compute $dp[i][j]$?

The value of $dp[i][j]$ can be written as below.

$$dp[i][j] = \max(\text{min_Points_on_exit} - \text{points}[i][j], 1)$$

Let us see how above expression covers all cases.

- If $\text{points}[i][j] == 0$, then nothing is gained in this cell; the player can leave the cell with the same points as he enters the room with, i.e. $dp[i][j] = \text{min_Points_on_exit}$.
- If $dp[i][j] > 0$, then the player could enter (i, j) with points as little as $\text{min_Points_on_exit} - \text{points}[i][j]$. since he could gain " $\text{points}[i][j]$ " points in this cell. However, the value of $\text{min_Points_on_exit} - \text{points}[i][j]$ might drop to 0 or below in this situation. When this happens, we must clip the value to 1 in order to make sure $dp[i][j]$ stays positive:

$$dp[i][j] = \max(\text{min_Points_on_exit} - \text{points}[i][j], 1).$$

Finally return $dp[0][0]$ which is our answer.

Below is the implementation of above algorithm.

C++

```

// C++ program to find minimum initial points to reach destination
#include<bits/stdc++.h>
#define R 3
#define C 3
using namespace std;

int minInitialPoints(int points[][C])
{
    // dp[i][j] represents the minimum initial points player

```

```

// should have so that when starts with cell(i, j) successfully
// reaches the destination cell(m-1, n-1)
int dp[R][C];
int m = R, n = C;

// Base case
dp[m-1][n-1] = points[m-1][n-1] > 0? 1:
abs(points[m-1][n-1]) + 1;

// Fill last row and last column as base to fill
// entire table
for (int i = m-2; i >= 0; i--)
    dp[i][n-1] = max(dp[i+1][n-1] - points[i][n-1], 1);
for (int j = n-2; j >= 0; j--)
    dp[m-1][j] = max(dp[m-1][j+1] - points[m-1][j], 1);

// fill the table in bottom-up fashion
for (int i=m-2; i>=0; i--)
{
    for (int j=n-2; j>=0; j--)
    {
        int min_points_on_exit = min(dp[i+1][j], dp[i][j+1]);
        dp[i][j] = max(min_points_on_exit - points[i][j], 1);
    }
}

return dp[0][0];
}

// Driver Program
int main()
{
    int points[R][C] = { {-2,-3,3},
    {-5,-10,1},
    {10,30,-5}
    };
    cout << "Minimum Initial Points Required:"
    << minInitialPoints(points);
    return 0;
}

```

Java

```

class min_steps
{
    static int minInitialPoints(int points[][],int R,int C)
    {
        // dp[i][j] represents the minimum initial points player
        // should have so that when starts with cell(i, j) successfully
        // reaches the destination cell(m-1, n-1)
        int dp[][] = new int[R][C];
        int m = R, n = C;

        // Base case
        dp[m-1][n-1] = points[m-1][n-1] > 0? 1:
        Math.abs(points[m-1][n-1]) + 1;

        // Fill last row and last column as base to fill
        // entire table
        for (int i = m-2; i >= 0; i--)
            dp[i][n-1] = Math.max(dp[i+1][n-1] - points[i][n-1], 1);
        for (int j = n-2; j >= 0; j--)
            dp[m-1][j] = Math.max(dp[m-1][j+1] - points[m-1][j], 1);

        // fill the table in bottom-up fashion
        for (int i=m-2; i>=0; i--)
        {
            for (int j=n-2; j>=0; j--)
            {
                int min_points_on_exit = Math.min(dp[i+1][j], dp[i][j+1]);
                dp[i][j] = Math.max(min_points_on_exit - points[i][j], 1);
            }
        }

        return dp[0][0];
    }

    /* Driver program to test above function */
    public static void main (String args[])
    {
        int points[][] = { {-2,-3,3},
        {-5,-10,1},
        {10,30,-5}
        };
        int R = 3,C = 3;
        System.out.println("Minimum Initial Points Required: "+
        minInitialPoints(points,R,C));
    }
    /* This code is contributed by Rajat Mishra */
}

```

Output:

Minimum Initial Points Required: 7

This article is contributed by [Gaurav Ahirwar](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Total number of non-decreasing numbers with n digits

A number is non-decreasing if every digit (except the first one) is greater than or equal to previous digit. For example, 223, 4455567, 899, are non-decreasing numbers.

So, given the number of digits n , you are required to find the count of total non-decreasing numbers with n digits.

Examples:

Input: $n = 1$

Output: count = 10

Input: $n = 2$

Output: count = 55

Input: $n = 3$

Output: count = 220

We strongly recommend you to minimize your browser and try this yourself first.

One way to look at the problem is, count of numbers is equal to count of n digit number ending with 9 plus count of ending with digit 8 plus count for 7 and so on. How to get count ending with a particular digit? We can recur for $n-1$ length and digits smaller than or equal to the last digit. So below is recursive formula.

Count of n digit numbers = (Count of $(n-1)$ digit numbers Ending with digit 9) +
(Count of $(n-1)$ digit numbers Ending with digit 8) +
.....+
.....+
(Count of $(n-1)$ digit numbers Ending with digit 0)

Let count ending with digit 'd' and length n be $\text{count}(n, d)$

$\text{count}(n, d) = \sum (\text{count}(n-1, i))$ where i varies from 0 to d

Total count = $\sum \text{count}(n-1, d)$ where d varies from 0 to $n-1$

The above recursive solution is going to have many overlapping subproblems. Therefore, we can use Dynamic Programming to build a table in bottom up manner. Below is Dynamic programming based C++ program.

C++

```
// C++ program to count non-decreasing number with n digits
#include<bits/stdc++.h>
using namespace std;

long long int countNonDecreasing(int n)
{
    // dp[i][j] contains total count of non decreasing
    // numbers ending with digit i and of length j
    long long int dp[10][n+1];
    memset(dp, 0, sizeof dp);

    // Fill table for non decreasing numbers of length 1
    // Base cases 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    for (int i = 0; i < 10; i++)
        dp[i][1] = 1;

    // Fill the table in bottom-up manner
    for (int digit = 0; digit <= 9; digit++)
    {
        // Compute total numbers of non decreasing
        // numbers of length 'len'
        for (int len = 2; len <= n; len++)
        {
            // sum of all numbers of length of len-1
            // in which last digit x is <= 'digit'
            for (int x = 0; x <= digit; x++)
                dp[digit][len] += dp[x][len-1];
        }
    }

    long long int count = 0;

    // There total nondecreasing numbers of length n
    // will be dp[0][n] + dp[1][n] ..+ dp[9][n]
    for (int i = 0; i < 10; i++)
        count += dp[i][n];

    return count;
}

// Driver program
int main()
{
    int n = 3;
    cout << countNonDecreasing(n);
    return 0;
}
```

Java

```
class NDN
{
    static int countNonDecreasing(int n)
    {
        // dp[i][j] contains total count of non decreasing
        // numbers ending with digit i and of length j
        int dp[][] = new int[10][n+1];

        // Fill table for non decreasing numbers of length 1
        // Base cases 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
        for (int i = 0; i < 10; i++)
            dp[i][1] = 1;
```

```

dp[0][1] = 1;

// Fill the table in bottom-up manner
for (int digit = 0; digit <= 9; digit++)
{
    // Compute total numbers of non decreasing
    // numbers of length 'len'
    for (int len = 2; len <= n; len++)
    {
        // sum of all numbers of length of len-1
        // in which last digit x is <= 'digit'
        for (int x = 0; x <= digit; x++)
            dp[digit][len] += dp[x][len-1];
    }
}

int count = 0;

// There total nondecreasing numbers of length n
// will be dp[0][n] + dp[1][n] ..+ dp[9][n]
for (int i = 0; i < 10; i++)
    count += dp[i][n];

return count;
}
public static void main(String args[])
{
    int n = 3;
    System.out.println(countNonDecreasing(n));
}
/* This code is contributed by Rajat Mishra */

```

Output:

220

Thanks to [Gaurav Ahirwar](#) for suggesting above method.

Another method is based on below direct formula

```

Count of non-decreasing numbers with n digits =
N*(N+1)/2*(N+2)/3* ....*(N+n-1)/n
Where N = 10

```

Below is a C++ program to compute count using above formula.

```

// C++ program to count non-decreasing number with n digits
#include<bits/stdc++.h>
using namespace std;

long long int countNonDecreasing(int n)
{
    int N = 10;

    // Compute value of N*(N+1)/2*(N+2)/3* ....*(N+n-1)/n
    long long count = 1;
    for (int i=1; i<=n; i++)
    {
        count *= (N+i-1);
        count /= i;
    }

    return count;
}

// Driver program
int main()
{
    int n = 3;
    cout << countNonDecreasing(n);
    return 0;
}

```

Output:

220

Thanks to [Abhishek Somani](#) for suggesting this method.

How does this formula work?

```

N * (N+1)/2 * (N+2)/3 * .... * (N+n-1)/n
Where N = 10

```

Let us try for different values of n.

For n = 1, the value is N from formula.

Which is true as for n = 1, we have all single digit numbers, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

For n = 2, the value is N(N+1)/2 from formula

We can have N numbers beginning with 0, (N-1) numbers beginning with 1, and so on.
So sum is $N + (N-1) + \dots + 1 = N(N+1)/2$

For n = 3, the value is N(N+1)/2(N+2)/3 from formula

We can have $N(N+1)/2$ numbers beginning with 0, $(N-1)N/2$ numbers beginning with 1 (Note that when we begin with 1, we have N-1 digits left to consider for remaining places), $(N-2)(N-1)/2$ beginning with 2, and so on.

Count = $N(N+1)/2 + (N-1)N/2 + (N-2)(N-1)/2 + \dots$

$(N-3)(N-2)/2 \dots 3 + 1$

[Combining first 2 terms, next 2 terms and so on]

$= 1/2[N^2 + (N-2)^2 + \dots 4]$

```
= N*(N+1)*(N+2)/6 [Refer this , putting n=N/2 in the even sum formula]
```

For general n digit case, we can apply Mathematical Induction. The count would be equal to count n-1 digit beginning with 0, i.e., $N*(N+1)/2*(N+2)/3* \dots * (N+n-1-1)/(n-1)$. Plus count of n-1 digit numbers beginning with 1, i.e., $(N-1)*(N)/2*(N+1)/3* \dots * (N-1+n-1-1)/(n-1)$ (Note that N is replaced by N-1) and so on.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Mathematical
Dynamic Programming
number-digits

Find length of the longest consecutive path from a given starting character

Given a matrix of characters. Find length of the longest path from a given character, such that all characters in the path are consecutive to each other, i.e., every character in path is next to previous in alphabetical order. It is allowed to move in all 8 directions from a cell.



Example

```
Input: mat[0][0] = { {a, c, d},  
                     {h, b, e},  
                     {i, g, f} }  
Starting Point = 'e'  
  
Output: 5  
If starting point is 'e', then longest path with consecutive  
characters is "e f g h i".  
  
Input: mat[R][C] = { {b, e, f},  
                     {h, d, a},  
                     {i, c, a} };  
Starting Point = 'b'  
  
Output: 1  
'c' is not present in all adjacent cells of 'b'
```

We strongly recommend you to minimize your browser and try this yourself first.

The idea is to first search given starting character in the given matrix. Do Depth First Search (DFS) from all occurrences to find all consecutive paths. While doing DFS, we may encounter many subproblems again and again. So we use dynamic programming to store results of subproblems.

Below is the implementation of above idea.

C++

```
// C++ program to find the longest consecutive path  
#include<bits/stdc++.h>  
#define R 3  
#define C 3  
using namespace std;  
  
// tool matrices to recur for adjacent cells.  
int x[] = {0, 1, 1, -1, 1, 0, -1, -1};  
int y[] = {1, 0, 1, 1, -1, -1, 0, -1};  
  
// dp[i][j] Stores length of longest consecutive path  
// starting at arr[i][j].  
int dp[R][C];  
  
// check whether mat[i][j] is a valid cell or not.  
bool isValid(int i, int j)  
{  
    if (i < 0 || j < 0 || i >= R || j >= C)  
        return false;  
    return true;  
}  
  
// Check whether current character is adjacent to previous  
// character (character processed in parent call) or not.  
bool isadjacent(char prev, char curr)  
{  
    return ((curr - prev) == 1);  
}  
  
// i, j are the indices of the current cell and prev is the  
// character processed in the parent call.. also mat[i][j]  
// is our current character.  
int getLenUtil(char mat[R][C], int i, int j, char prev)  
{  
    // If this cell is not valid or current character is not  
    // adjacent to previous one (e.g. d is not adjacent to b )
```

```

// or if this cell is already included in the path than return 0.
if (!isValid(i, j) || !isAdjacent(prev, mat[i][j]))
    return 0;

// If this subproblem is already solved , return the answer
if (dp[i][j] != -1)
    return dp[i][j];

int ans = 0; // Initialize answer

// recur for paths with different adjacent cells and store
// the length of longest path.
for (int k=0; k<8; k++)
    ans = max(ans, 1 + getLenUtil(mat, i + x[k],
        j + y[k], mat[i][j]));

// save the answer and return
return dp[i][j] = ans;
}

// Returns length of the longest path with all characters consecutive
// to each other. This function first initializes dp array that
// is used to store results of subproblems, then it calls
// recursive DFS based function getLenUtil() to find max length path
int getLen(char mat[R][C], char s)
{
    memset(dp, -1, sizeof dp);
    int ans = 0;

    for (int i=0; i<R; i++)
    {
        for (int j=0; j<C; j++)
        {
            // check for each possible starting point
            if (mat[i][j] == s) {

                // recur for all eight adjacent cells
                for (int k=0; k<8; k++)
                    ans = max(ans, 1 + getLenUtil(mat,
                        i + x[k], j + y[k], s));
            }
        }
    }
    return ans;
}

// Driver program
int main()
{
    char mat[R][C] = { {'a','c','d'},
                       { 'h','b','a'},
                       { 'i','g','f'}};

    cout << getLen(mat, 'a') << endl;
    cout << getLen(mat, 'e') << endl;
    cout << getLen(mat, 'b') << endl;
    cout << getLen(mat, 'f') << endl;
    return 0;
}

```

Java

```

class path
{
    // tool matrices to recur for adjacent cells.
    static int x[] = {0, 1, 1, -1, 1, 0, -1, -1};
    static int y[] = {1, 0, 1, 1, -1, -1, 0, -1};
    static int R = 3;
    static int C = 3;
    // dp[i][j] Stores length of longest consecutive path
    // starting at arr[i][j].
    static int dp[][] = new int[R][C];

    // check whether mat[i][j] is a valid cell or not.
    static boolean isValid(int i, int j)
    {
        if (i < 0 || j < 0 || i >= R || j >= C)
            return false;
        return true;
    }

    // Check whether current character is adjacent to previous
    // character (character processed in parent call) or not.
    static boolean isAdjacent(char prev, char curr)
    {
        return ((curr - prev) == 1);
    }

    // i, j are the indices of the current cell and prev is the
    // character processed in the parent call.. also mat[i][j]
    // is our current character.
    static int getLenUtil(char mat[][], int i, int j, char prev)
    {
        // If this cell is not valid or current character is not
        // adjacent to previous one (e.g. d is not adjacent to b )
        // or if this cell is already included in the path than return 0.
        if (!isValid(i, j) || !isAdjacent(prev, mat[i][j]))
            return 0;

        // If this subproblem is already solved , return the answer
        if (dp[i][j] != -1)
            return dp[i][j];

        int ans = 0; // Initialize answer

```

```

// recur for paths with different adjacent cells and store
// the length of longest path.
for (int k=0; k<8; k++)
    ans = Math.max(ans, 1 + getLenUtil(mat, i + x[k],
        j + y[k], mat[i][j]));

// save the answer and return
return dp[i][j] = ans;
}

// Returns length of the longest path with all characters consecutive
// to each other. This function first initializes dp array that
// is used to store results of subproblems, then it calls
// recursive DFS based function getLenUtil() to find max length path
static int getLen(char mat[][], char s)
{
    //assigning all dp values to -1
    for(int i = 0;i<R;++i)
        for(int j = 0;j<C;++j)
            dp[i][j] = -1;

    int ans = 0;

    for (int i=0;i<R; i++)
    {
        for (int j=0; j<C; j++)
        {
            // check for each possible starting point
            if (mat[i][j] == s) {

                // recur for all eight adjacent cells
                for (int k=0; k<8; k++)
                    ans = Math.max(ans, 1 + getLenUtil(mat,
                        i + x[k], j + y[k], s));
            }
        }
    }
    return ans;
}
public static void main(String args[])
{
    char mat[][] = {{'a','c','d'},
                   {'h','b','e'},
                   {'i','g','f'}};

    System.out.println(getLen(mat, 'a'));
    System.out.println(getLen(mat, 'e'));
    System.out.println(getLen(mat, 'b'));
    System.out.println(getLen(mat, 'f'));
}
/* This code is contributed by Rajat Mishra */

```

Output:

```

4
0
3
4

```

Thanks to [Gaurav Hirwar](#) for above solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Matrix
Dynamic Programming
Matrix

Tiling Problem

Given a "2 x n" board and tiles of size "2 x 1", count the number of ways to tile the given board using the 2 x 1 tiles. A tile can either be placed horizontally i.e., as a 1 x 2 tile or vertically i.e., as 2 x 1 tile.

Examples:

Input n = 3
Output: 3
Explanation:
We need 3 tiles to tile the board of size 2 x 3.
We can tile the board using following ways
1) Place all 3 tiles vertically.
2) Place first tile vertically and remaining 2 tiles horizontally.
3) Place first 2 tiles horizontally and remaining tiles vertically

Input n = 4
Output: 5
Explanation:
For a 2 x 4 board, there are 5 ways
1) All 4 vertical
2) All 4 horizontal
3) First 2 vertical, remaining 2 horizontal
4) First 2 horizontal, remaining 2 vertical
5) Corner 2 vertical, middle 2 horizontal



We strongly recommend that you click here and practice it, before moving on to the solution.

Let "count(n)" be the count of ways to place tiles on a "2 x n" grid, we have following two ways to place first tile.

- 1) If we place first tile vertically, the problem reduces to "count(n-1)"
- 2) If we place first tile horizontally, we have to place second tile also horizontally. So the problem reduces to "count(n-2)"

Therefore, count(n) can be written as below.

```
count(n) = n if n = 1 or n = 2
count(n) = count(n-1) + count(n-2)
```

The above recurrence is noting but Fibonacci Number expression. We can find n'th Fibonacci number in O(Log n) time, see below for all method to find n'th Fibonacci Number.

Different methods for n'th Fibonacci Number.

This article is contributed by Saurabh Jain. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming
MathematicalAlgo

Minimum number of squares whose sum equals to given number n

A number can always be represented as a sum of squares of other numbers. Note that 1 is a square and we can always break a number as $(1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 + \dots)$. Given a number n, find the minimum number of squares that sum to X.

Examples:

```
Input: n = 100
Output: 1
100 can be written as  $10^2$ . Note that 100 can also be
written as  $5^2 + 5^2 + 5^2 + 5^2$ , but this
representation requires 4 squares.
```

```
Input: n = 6
Output: 3
```

We strongly recommend you to minimize your browser and try this yourself first.

The idea is simple, we start from 1 and go till a number whose square is smaller than or equals to n. For every number x, we recur for n-x. Below is the recursive formula.

```
If n = 1 and x*x
Below is a simple recursive solution based on above recursive formula.
```

C++

```
// A naive recursive C++ program to find minimum
// number of squares whose sum is equal to a given number
#include <bits/stdc++.h>
using namespace std;

// Returns count of minimum squares that sum to n
int getMinSquares(unsigned int n)
{
    // base cases
    if (n <= 3)
        return n;

    // getMinSquares rest of the table using recursive
    // formula
    int res = n; // Maximum squares required is n ( $1 \cdot 1 + 1 \cdot 1 + \dots$ )
    for (int x = 1; x <= n; x++)
    {
        int temp = x*x;
        if (temp > n)
            break;
        else
            res = min(res, 1+getMinSquares(n - temp));
    }
    return res;
}
```

```
// Driver program
int main()
{
    cout << getMinSquares(6);
    return 0;
}
```

Java

```
// A naive recursive JAVA program to find minimum
// number of squares whose sum is equal to a given number
class squares
{
    // Returns count of minimum squares that sum to n
    static int getMinSquares(int n)
    {
        // base cases
        if (n <= 3)
            return n;

        // getMinSquares rest of the table using recursive
        // formula
        int res = n; // Maximum squares required is n (1*1 + 1*1 + ..)

        // Go through all smaller numbers
        // to recursively find minimum
        for (int x = 1; x <= n; x++)
        {
            int temp = x*x;
            if (temp > n)
                break;
            else
                res = Math.min(res, 1+getMinSquares(n - temp));
        }
        return res;
    }
    public static void main(String args[])
    {
        System.out.println(getMinSquares(6));
    }
}/* This code is contributed by Rajat Mishra */
```

Output:

```
3
```

The time complexity of above solution is exponential. If we draw the complete recursion tree, we can observe that many subproblems are solved again and again. For example, when we start from $n = 6$, we can reach 4

Since same subproblems are called again, this problem has Overlapping Subproblems property. So min square sum problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem. Like other typical [DP](#)

C++

```
// A dynamic programming based C++ program to find minimum
// number of squares whose sum is equal to a given number
#include<bits/stdc++.h>
using namespace std;

// Returns count of minimum squares that sum to n
int getMinSquares(int n)
{
    // Create a dynamic programming table
    // to store sq
    int *dp = new int[n+1];

    // getMinSquares table for base case entries
    dp[0] = 0;
    dp[1] = 1;
    dp[2] = 2;
    dp[3] = 3;

    // getMinSquares rest of the table using recursive
    // formula
    for (int i = 4; i <= n; i++)
    {
        // max value is i as i can always be represented
        // as 1*1 + 1*1 + ...
        dp[i] = i;
    }

    // Go through all smaller numbers to
    // to recursively find minimum
    for (int x = 1; x <= i; x++) {
        int temp = x*x;
        if (temp > i)
            break;
        else dp[i] = min(dp[i], 1+dp[i-temp]);
    }
}
```

```

    }

    // Store result and free dp[]
    int res = dp[n];
    delete [] dp;

    return res;
}

// Driver program
int main()
{
    cout << getMinSquares(6);
    return 0;
}

```

Java

```

// A dynamic programming based JAVA program to find minimum
// number of squares whose sum is equal to a given number
class squares
{
    // Returns count of minimum squares that sum to n
    static int getMinSquares(int n)
    {
        // Create a dynamic programming table
        // to store sq
        int dp[] = new int[n+1];

        // getMinSquares table for base case entries
        dp[0] = 0;
        dp[1] = 1;
        dp[2] = 2;
        dp[3] = 3;

        // getMinSquares rest of the table using recursive
        // formula
        for (int i = 4; i <= n; i++)
        {
            // max value is i as i can always be represented
            // as 1*1 + 1*1 + ...
            dp[i] = i;

            // Go through all smaller numbers to
            // to recursively find minimum
            for (int x = 1; x <= i; x++) {
                int temp = x*x;
                if (temp > i)
                    break;
                else dp[i] = Math.min(dp[i], 1+dp[i-temp]);
            }
        }

        // Store result and free dp[]
        int res = dp[n];

        return res;
    }

    public static void main(String args[])
    {
        System.out.println(getMinSquares(6));
    }
}/* This code is contributed by Rajat Mishra */

```

Output:

3

Thanks to Gaurav Ahirwar for suggesting this solution in a comment [here](#).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Company Wise Coding Practice

Dynamic Programming
Mathematical
Dynamic Programming
MathematicalAlgo

Find minimum number of coins that make a given value

Given a value V, if we want to make change for V cents, and we have infinite supply of each of C = {C1, C2, ..., Cm} valued coins, what is the minimum number of coins to make the change?

Examples:

```
Input: coins[] = {25, 10, 5}, V = 30
Output: Minimum 2 coins required
We can use one coin of 25 cents and one of 5 cents
```

```
Input: coins[] = {9, 6, 5, 1}, V = 11
Output: Minimum 2 coins required
We can use one coin of 6 cents and 1 coin of 5 cents
```

We strongly recommend you to minimize your browser and try this yourself first.

This problem is a variation of the problem discussed [Coin Change Problem](#). Here instead of finding total number of possible solutions, we need to find the solution with minimum number of coins.

The minimum number of coins for a value V can be computed using below recursive formula.

```
If V == 0, then 0 coins required.
If V > 0
minCoin(coins[0..m-1], V) = min {1 + minCoins(V-coin[i])}
    where i varies from 0 to m-1
        and coin[i]
```

Below is recursive solution based on above recursive formula.

C++

```
// A Naive recursive C++ program to find minimum of coins
// to make a given change V
#include<bits/stdc++.h>
using namespace std;

// m is size of coins array (number of different coins)
int minCoins(int coins[], int m, int V)
{
    // base case
    if (V == 0) return 0;

    // Initialize result
    int res = INT_MAX;

    // Try every coin that has smaller value than V
    for (int i=0; i<m; i++)
    {
        if (coins[i] <= V)
        {
            int sub_res = minCoins(coins, m, V-coins[i]);

            // Check for INT_MAX to avoid overflow and see if
            // result can minimized
            if (sub_res != INT_MAX && sub_res + 1 < res)
                res = sub_res + 1;
        }
    }
    return res;
}

// Driver program to test above function
int main()
{
    int coins[] = {9, 6, 5, 1};
    int m = sizeof(coins)/sizeof(coins[0]);
    int V = 11;
    cout << "Minimum coins required is "
         << minCoins(coins, m, V);
    return 0;
}
```

Java

```
// A Naive recursive JAVA program to find minimum of coins
// to make a given change V
class coin
{
    // m is size of coins array (number of different coins)
    static int minCoins(int coins[], int m, int V)
```

```

{
    // base case
    if (V == 0) return 0;

    // Initialize result
    int res = Integer.MAX_VALUE;

    // Try every coin that has smaller value than V
    for (int i=0; i<m; i++)
    {
        if (coins[i] <= V)
        {
            int sub_res = minCoins(coins, m, V-coins[i]);

            // Check for INT_MAX to avoid overflow and see if
            // result can minimized
            if (sub_res != Integer.MAX_VALUE && sub_res + 1 < res)
                res = sub_res + 1;
        }
    }
    return res;
}

public static void main(String args[])
{
    int coins[] = {9, 6, 5, 1};
    int m = coins.length;
    int V = 11;
    System.out.println("Minimum coins required is "+ minCoins(coins, m, V));
}
/* This code is contributed by Rajat Mishra */

```

Output:

Minimum coins required is 2

The time complexity of above solution is exponential. If we draw the complete recursion tree, we can observe that many subproblems are solved again and again. For example, when we start from $V = 11$, we can reach $V = 11$ from $V = 11 - 1$ and $V = 11 - 5$.

Since same subproblems are called again, this problem has Overlapping Subproblems property. So the min coins problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem. Like other typical Dynamic Programming problems, we can either use Recursion or Dynamic Programming.

C++

```

// A Dynamic Programming based C++ program to find minimum of coins
// to make a given change V
#include<bits/stdc++.h>
using namespace std;

// m is size of coins array (number of different coins)
int minCoins(int coins[], int m, int V)
{
    // table[i] will be storing the minimum number of coins
    // required for i value. So table[V] will have result
    int table[V+1];

    // Base case (If given value V is 0)
    table[0] = 0;

    // Initialize all table values as Infinite
    for (int i=1; i<=V; i++)
        table[i] = INT_MAX;

    // Compute minimum coins required for all
    // values from 1 to V
    for (int i=1; i<=V; i++)
    {
        // Go through all coins smaller than i
        for (int j=0; j<m; j++)
            if (coins[j] <= i)
            {
                int sub_res = table[i-coins[j]];
                if (sub_res != INT_MAX && sub_res + 1 < table[i])
                    table[i] = sub_res + 1;
            }
    }
    return table[V];
}

// Driver program to test above function
int main()
{
    int coins[] = {9, 6, 5, 1};
    int m = sizeof(coins)/sizeof(coins[0]);
    int V = 11;
    cout << "Minimum coins required is "
         << minCoins(coins, m, V);
    return 0;
}

```

Output:

Minimum coins required is 2

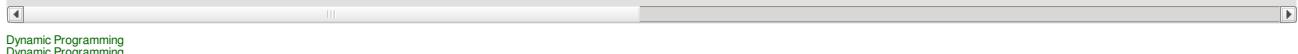
Time complexity of the above solution is O(mV).

Thanks to Goku for suggesting above solution in a comment [here](#) and thanks to Vignesh Mohan for suggesting this problem and initial solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner

Company Wise Coding Practice



Collect maximum points in a grid using two traversals

Given a matrix where every cell represents points. How to collect maximum points using two traversals under following conditions?

Let the dimensions of given grid be R x C.

- 1) The first traversal starts from top left corner, i.e., (0, 0) and should reach left bottom corner, i.e., (R-1, 0). The second traversal starts from top right corner, i.e., (0, C-1) and should reach bottom right corner, i.e., (R-1, C-1)/
- 2) From a point (i, j), we can move to (i+1, j+1) or (i+1, j-1) or (i+1, j)
- 3) A traversal gets all points of a particular cell through which it passes. If one traversal has already collected points of a cell, then the other traversal gets no points if goes through that cell again.

Input :
int arr[R][C] = {{3, 6, 8, 2},
{5, 2, 4, 3},
{1, 1, 20, 10},
{1, 1, 20, 10},
{1, 1, 20, 10},
};

Output: 73

Explanation :
[runninggrid](#)



First traversal collects total points of value $3 + 2 + 20 + 1 + 1 = 27$

Second traversal collects total points of value $2 + 4 + 10 + 20 + 10 = 46$.

Total Points collected = $27 + 46 = 73$.

Source: <http://qa.geeksforgeeks.org/1485/running-through-the-grid-to-get-maximum-nutritional-value>

We strongly recommend you to minimize your browser and try this yourself first.

The idea is to do both traversals concurrently. We start first from (0, 0) and second traversal from (0, C-1) simultaneously. The important thing to note is, at any particular step both traversals will be in same row as in all possible three moves, row number is increased. Let (x1, y1) and (x2, y2) denote current positions of first and second traversals respectively. Thus at any time x1 will be equal to x2 as both of them move forward but variation is possible along y. Since variation in y could occur in 3 ways no change (y), go left (y - 1), go right (y + 1). So in total 9 combinations among y1, y2 are possible. The 9 cases as mentioned below after base cases.

```
Both traversals always move forward along x
Base Cases:
// If destination reached
if (x == R-1 && y1 == 0 && y2 == C-1)
maxPoints(arr, x, y1, y2) = arr[x][y1] + arr[x][y2];

// If any of the two locations is invalid (going out of grid)
if input is not valid
maxPoints(arr, x, y1, y2) = -INF (minus infinite)

// If both traversals are at same cell, then we count the value of cell
// only once.
If y1 and y2 are same
    result = arr[x][y1]
Else
    result = arr[x][y1] + arr[x][y2]

result += max ( // Max of 9 cases
    maxPoints(arr, x+1, y1+1, y2),
    maxPoints(arr, x+1, y1+1, y2+1),
```

```

        maxPoints(arr, x+1, y1+1, y2-1),
        maxPoints(arr, x+1, y1-1, y2),
        maxPoints(arr, x+1, y1-1, y2+1),
        maxPoints(arr, x+1, y1-1, y2-1),
        maxPoints(arr, x+1, y1, y2),
        maxPoints(arr, x+1, y1, y2+1),
        maxPoints(arr, x+1, y1, y2-1)
    }
}

```

The above recursive solution has many subproblems that are solved again and again. Therefore, we can use Dynamic Programming to solve the above problem more efficiently. Below is [memoization](#) (Memoization is alternative to table based iterative solution in Dynamic Programming) based implementation. In below implementation, we use a memoization table 'mem' to keep track of already solved problems.

```

// A Memoization based program to find maximum collection
// using two traversals of a grid
#include<bits/stdc++.h>
using namespace std;
#define R 5
#define C 4

// checks whether a given input is valid or not
bool isValid(int x, int y1, int y2)
{
    return (x >= 0 && x < R && y1 >= 0 &&
            y1 < C && y2 >= 0 && y2 < C);
}

// Driver function to collect max value
int getMaxUtil(int arr[R][C], int mem[R][C][C], int x, int y1, int y2)
{
    /*----- BASE CASES -----*/
    // if P1 or P2 is at an invalid cell
    if (!isValid(x, y1, y2)) return INT_MIN;

    // if both traversals reach their destinations
    if (x == R-1 && y1 == 0 && y2 == C-1)
        return (y1 == y2)? arr[x][y1]: arr[x][y1] + arr[x][y2];

    // If both traversals are at last row but not at their destination
    if (x == R-1) return INT_MIN;

    // If subproblem is already solved
    if (mem[x][y1][y2] != -1) return mem[x][y1][y2];

    // Initialize answer for this subproblem
    int ans = INT_MIN;

    // this variable is used to store gain of current cell(s)
    int temp = (y1 == y2)? arr[x][y1]: arr[x][y1] + arr[x][y2];

    /* Recur for all possible cases, then store and return the
       one with max value */
    ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1, y2-1));
    ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1, y2+1));
    ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1, y2));

    ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1-1, y2));
    ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1-1, y2-1));
    ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1-1, y2+1));

    ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1+1, y2));
    ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1+1, y2-1));
    ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1+1, y2+1));

    return (mem[x][y1][y2] = ans);
}

// This is mainly a wrapper over recursive function getMaxUtil().
// This function creates a table for memoization and calls
// getMaxUtil()
int geMaxCollection(int arr[R][C])
{
    // Create a memoization table and initialize all entries as -1
    int mem[R][C][C];
    memset(mem, -1, sizeof(mem));

    // Calculation maximum value using memoization based function
    // getMaxUtil()
    return getMaxUtil(arr, mem, 0, 0, C-1);
}

// Driver program to test above functions
int main()
{
    int arr[R][C] = {{3, 6, 8, 2},
                     {5, 2, 4, 3},
                     {1, 1, 20, 10},
                     {1, 1, 20, 10},
                     {1, 1, 20, 10},
                     {1, 1, 20, 10}};
    cout << "Maximum collection is " << geMaxCollection(arr);
    return 0;
}

```

Output:

```
Maximum collection is 73
```

Thanks to Gaurav Ahirwar for suggesting above problem and solution [here](#).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Shortest Common Supersequence

Given two strings str1 and str2, find the shortest string that has both str1 and str2 as subsequences.

Examples:

```
Input: str1 = "geek", str2 = "eke"
Output: "geeke"
```

```
Input: str1 = "AGGTAB", str2 = "GXTXAYB"
Output: "AGXGTXAYB"
```

We strongly recommend you to minimize your browser and try this yourself first.

This problem is closely related to [longest common subsequence problem](#). Below are steps.

1) Find Longest Common Subsequence (lcs) of two given strings. For example, lcs of "geek" and "eke" is "ek".

2) Insert non-lcs characters (in their original order in strings) to the lcs found above, and return the result. So "ek" becomes "geeke" which is shortest common supersequence.

Let us consider another example, str1 = "AGGTAB" and str2 = "GXTXAYB". LCS of str1 and str2 is "GTAB". Once we find LCS, we insert characters of both strings in order and we get "AGXGTXAYB"

How does this work?

We need to find a string that has both strings as subsequences and is shortest such string. If both strings have all characters different, then result is sum of lengths of two given strings. If there are common characters, then we don't want them multiple times as the task is to minimize length. Therefore, we first find the longest common subsequence, take one occurrence of this subsequence and add extra characters.

```
Length of the shortest supersequence = (Sum of lengths of given two strings) -
(Length of LCS of two given strings)
```

Below is C implementation of above idea. The below implementation only finds length of the shortest supersequence.

```
/* C program to find length of the shortest supersequence */
#include<stdio.h>
#include<string.h>

/* Utility function to get max of 2 integers */
int max(int a, int b) { return (a > b)? a : b; }

/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n);

// Function to find length of the shortest supersequence
// of X and Y.
int shortestSuperSequence(char *X, char *Y)
{
    int m = strlen(X), n = strlen(Y);

    int l = lcs(X, Y, m, n); // find lcs

    // Result is sum of input string lengths - length of lcs
    return (m + n - l);
}

/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n)
{
    int L[m+1][n+1];
    int i, j;

    /* Following steps build L[m+1][n+1] in bottom up fashion.
     Note that L[i][j] contains length of LCS of X[0..i-1]
     and Y[0..j-1] */
    for (i=0; i<=m; i++)
    {
        for (j=0; j<=n; j++)
        {
            if (i == 0 || j == 0)
                L[i][j] = 0;

            else if (X[i-1] == Y[j-1])
                L[i][j] = L[i-1][j-1] + 1;

            else
                L[i][j] = max(L[i-1][j], L[i][j-1]);
        }
    }

    /* L[m][n] contains length of LCS for X[0..n-1] and
     Y[0..m-1] */
    return L[m][n];
}

/* Driver program to test above function */
int main()
{
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";
    printf("Length of the shortest supersequence is %d\n",
          shortestSuperSequence(X, Y));
    return 0;
}
```

Output:

```
Length of the shortest supersequence is 9
```

Below is [Another Method](#) to solve the above problem.

A simple analysis yields below simple recursive solution.

```
Let X[0..m-1] and Y[0..n-1] be two strings and m and n be respective  
lengths.  
  
if (m == 0) return n;  
if (n == 0) return m;  
  
// If last characters are same, then add 1 to result and  
// recur for X[]  
if (X[m-1] == Y[n-1])  
    return 1 + SCS(X, Y, m-1, n-1);  
  
// Else find shortest of following two  
// a) Remove last character from X and recur  
// b) Remove last character from Y and recur  
else return 1 + min( SCS(X, Y, m-1, n), SCS(X, Y, m, n-1) );
```

Below is simple naive recursive solution based on above recursive formula.

```
/* A Naive recursive C++ program to find length  
of the shortest supersequence */  
#include<bits/stdc++.h>  
using namespace std;  
  
int superSeq(char* X, char* Y, int m, int n)  
{  
    if (l(m)) return n;  
    if (l(n)) return m;  
  
    if (X[m-1] == Y[n-1])  
        return 1 + superSeq(X, Y, m-1, n-1);  
  
    return 1 + min(superSeq(X, Y, m-1, n),  
                   superSeq(X, Y, m, n-1));  
}  
  
// Driver program to test above function  
int main()  
{  
    char X[] = "AGGTAB";  
    char Y[] = "GXTXAYB";  
    cout << "Length of the shortest supersequence is "  
         << superSeq(X, Y, strlen(X), strlen(Y));  
    return 0;  
}
```

Output:

```
Length of the shortest supersequence is 9
```

Time complexity of the above solution exponential $O(2^{\min(m, n)})$. Since there are **overlapping subproblems**, we can efficiently solve this recursive problem using Dynamic Programming. Below is Dynamic Programming based implementation. Time complexity of this solution is $O(mn)$.

```
/* A dynamic programming based C program to find length  
of the shortest supersequence */  
#include<bits/stdc++.h>  
using namespace std;  
  
// Returns length of the shortest supersequence of X and Y  
int superSeq(char* X, char* Y, int m, int n)  
{  
    int dp[m+1][n+1];  
  
    // Fill table in bottom up manner  
    for (int i = 0; i <= m; i++)  
    {  
        for (int j = 0; j <= n; j++)  
        {  
            // Below steps follow above recurrence  
            if (!l())  
                dp[i][j] = j;  
            else if (!j)  
                dp[i][j] = i;  
            else if (X[i-1] == Y[j-1])  
                dp[i][j] = 1 + dp[i-1][j-1];  
            else  
                dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1]);  
        }  
    }  
  
    return dp[m][n];  
}  
  
// Driver program to test above function  
int main()  
{  
    char X[] = "AGGTAB";  
    char Y[] = "GXTXAYB";  
    cout << "Length of the shortest supersequence is "  
         << superSeq(X, Y, strlen(X), strlen(Y));  
    return 0;  
}
```

Output:

```
Length of the shortest supersequence is 9
```

Thanks to [Gaurav Ahirwar](#) for suggesting this solution.

Exercise:

Extend the above program to print shortest supersequence also using [function to print LCS](#).

References:

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming
LCS
subsequence

Compute sum of digits in all numbers from 1 to n

Given a number x, find sum of digits in all numbers from 1 to n.

Examples:

```
Input: n = 5
Output: Sum of digits in numbers from 1 to 5 = 15

Input: n = 12
Output: Sum of digits in numbers from 1 to 12 = 51

Input: n = 328
Output: Sum of digits in numbers from 1 to 328 = 3241
```

Naive Solution:

A naive solution is to go through every number x from 1 to n, and compute sum in x by traversing all digits of x. Below is C++ implementation of this idea.

```
// A Simple C++ program to compute sum of digits in numbers from 1 to n
#include<iostream>
using namespace std;

int sumOfDigits(int);

// Returns sum of all digits in numbers from 1 to n
int sumOfDigitsFrom1ToN(int n)
{
    int result = 0; // initialize result

    // One by one compute sum of digits in every number from
    // 1 to n
    for (int x=1; x<=n; x++)
        result += sumOfDigits(x);

    return result;
}

// A utility function to compute sum of digits in a
// given number x
int sumOfDigits(int x)
{
    int sum = 0;
    while (x != 0)
    {
        sum += x % 10;
        x = x/10;
    }
    return sum;
}

// Driver Program
int main()
{
    int n = 328;
    cout << "Sum of digits in numbers from 1 to " << n << " is "
        << sumOfDigitsFrom1ToN(n);
    return 0;
}
```

Output

```
Sum of digits in numbers from 1 to 328 is 3241
```

Efficient Solution:

Above is a naive solution. We can do it more efficiently by finding a pattern.

Let us take few examples.

```
sum(9) = 1 + 2 + 3 + 4 ..... + 9
       = 9*10/2
       = 45

sum(99) = 45 + (10 + 45) + (20 + 45) + .... (90 + 45)
       = 45*10 + (10 + 20 + 30 ... 90)
       = 45*10 + 10(1 + 2 + ... 9)
       = 45*10 + 45*10
       = sum(9)*10 + 45*10

sum(999) = sum(99)*10 + 45*100
```

In general, we can compute $\text{sum}(10^d - 1)$ using below formula

```
sum(10^d - 1) = sum(10^{d-1} - 1) * 10 + 45*(10^{d-1})
```

In below implementation, the above formula is implemented using **dynamic programming** as there are overlapping subproblems.

The above formula is one core step of the idea. Below is complete algorithm

Algorithm: sum(n)

- 1) Find number of digits minus one in n. Let this value be 'd'.
For 328, d is 2.

- 2) Compute some of digits in numbers from 1 to $10^d - 1$.
 Let this sum be w. For 328, we compute sum of digits from 1 to 99 using above formula.
- 3) Find Most significant digit (msd) in n. For 328, msd is 3.
- 4) Overall sum is sum of following terms
- a) Sum of digits in 1 to "msd * $10^d - 1$ ". For 328, sum of digits in numbers from 1 to 299.
 For 328, we compute $3^2 \cdot \text{sum}(99) + (1+2) \cdot 100$. Note that sum of sum(299) is sum(99) + sum of digits from 100 to 199 + sum of digits from 200 to 299.
 Sum of 100 to 199 is sum(99) + $1 \cdot 100$ and sum of 299 is sum(99) + $2 \cdot 100$.
 In general, this sum can be computed as $w \cdot \text{msd} + (\text{msd} \cdot (\text{msd}-1)/2) \cdot 10^d$
 - b) Sum of digits in msd * 10^d to n. For 328, sum of digits in 300 to 328.
 For 328, this sum is computed as $3^2 \cdot 29 + \text{recursive call "sum}(28)$.
 In general, this sum can be computed as $\text{msd} \cdot (n \% (\text{msd} \cdot 10^d) + 1) + \text{sum}(n \% (10^d))$

Below is C++ implementation of above algorithm.

```
// C++ program to compute sum of digits in numbers from 1 to n
#include<bits/stdc++.h>
using namespace std;

// Function to computer sum of digits in numbers from 1 to n
// Comments use example of 328 to explain the code
int sumOfDigitsFrom1ToN(int n)
{
    // base case: if n<10 return sum of
    // first n natural numbers
    if (n<10)
        return n*(n+1)/2;

    // d = number of digits minus one in n. For 328, d is 2
    int d = log10(n);

    // computing sum of digits from 1 to  $10^d - 1$ ,
    // d=1 a[0]=0;
    // d=2 a[1]=sum of digit from 1 to 9 = 45
    // d=3 a[2]=sum of digit from 1 to 99 = a[1]*10 + 45*10^1 = 900
    // d=4 a[3]=sum of digit from 1 to 999 = a[2]*10 + 45*10^2 = 13500
    int a = new int[d+1];
    a[0] = 0, a[1] = 45;
    for (int i=2; i<=d; i++)
        a[i] = a[i-1]*10 + 45*ceil(pow(10, i-1));

    // computing  $10^d$ 
    int p = ceil(pow(10, d));

    // Most significant digit (msd) of n,
    // For 328, msd is 3 which can be obtained using 328/100
    int msd = n/p;

    // EXPLANATION FOR FIRST and SECOND TERMS IN BELOW LINE OF CODE
    // First two terms compute sum of digits from 1 to 299
    // (sum of digits in range 1-99 stored in a[d]) +
    // (sum of digits in range 100-199, can be calculated as  $1 \cdot 100 + a[d]$ )
    // (sum of digits in range 200-299, can be calculated as  $2 \cdot 100 + a[d]$ )
    // The above sum can be written as  $3 \cdot a[d] + (1+2) \cdot 100$ 

    // EXPLANATION FOR THIRD AND FOURTH TERMS IN BELOW LINE OF CODE
    // The last two terms compute sum of digits in number from 300 to 328
    // The third term adds  $3^2 \cdot 29$  to sum as digit 3 occurs in all numbers
    // from 300 to 328
    // The fourth term recursively calls for 28
    return msd*a[d] + (msd*(msd-1)/2)*p +
        msd*(1+n%p) + sumOfDigitsFrom1ToN(n%p);
}

// Driver Program
int main()
{
    int n = 328;
    cout << "Sum of digits in numbers from 1 to " << n << " is "
        << sumOfDigitsFrom1ToN(n);
    return 0;
}
```

Output

```
Sum of digits in numbers from 1 to 328 is 3241
```

The efficient algorithm has one more advantage that we need to compute the array 'a[]' only once even when we are given multiple inputs.

This article is computed by **Shubham Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
 Mathematical
 Dynamic Programming
 MathematicalAlgo
 number-digits

Count possible ways to construct buildings

Given an input number of sections and each section has 2 plots on either sides of the road. Find all possible ways to construct buildings in the plots such that there is a space between any 2 buildings.

Example:

```

N = 1
Output = 4
Place a building on one side.
Place a building on other side
Do not place any building.
Place a building on both sides.

N = 3
Output = 25
3 sections, which means possible ways for one side are
BSS, BSB, SSS, SBS, SSB where B represents a building
and S represents an empty space
Total possible ways are 25, because a way to place on
one side can correspond to any of 5 ways on other side.

N = 4
Output = 64

```

We strongly recommend to minimize your browser and try this yourself first

We can simplify the problem to first calculate for one side only. If we know the result for one side, we can always do square of the result and get result for two sides.

A new building can be placed on a section if section just before it has space. A space can be placed anywhere (it doesn't matter whether the previous section has a building or not).

```

Let countB(i) be count of possible ways with i sections
    ending with a building.
countS(i) be count of possible ways with i sections
    ending with a space.

// A space can be added after a building or after a space.
countS(N) = countB(N-1) + countS(N-1)

// A building can only be added after a space.
countB[N] = countS(N-1)

// Result for one side is sum of the above two counts.
result1(N) = countS(N) + countB(N)

// Result for two sides is square of result1(N)
result2(N) = result1(N) * result1(N)

```

Below is the implementation of above idea.

C++

```

// C++ program to count all possible way to construct buildings
#include<iostream>
using namespace std;

// Returns count of possible ways for N sections
int countWays(int N)
{
    // Base case
    if (N == 1)
        return 4; // 2 for one side and 4 for two sides

    // countB is count of ways with a building at the end
    // countS is count of ways with a space at the end
    // prev_countB and prev_countS are previous values of
    // countB and countS respectively.

    // Initialize countB and countS for one side
    int countB=1, countS=1, prev_countB, prev_countS;

    // Use the above recursive formula for calculating
    // countB and countS using previous values
    for (int i=2; i<=N; i++)
    {
        prev_countB = countB;
        prev_countS = countS;

        countS = prev_countB + prev_countS;
        countB = prev_countS;
    }

    // Result for one side is sum of ways ending with building
    // and ending with space
    int result = countS + countB;

    // Result for 2 sides is square of result for one side
    return (result*result);
}

// Driver program
int main()
{
    int N = 3;
    cout << "Count of ways for " << N
         << "sections is " << countWays(N);
    return 0;
}

```

Java

```

class Building
{
    // Returns count of possible ways for N sections
    static int countWays(int N)
    {
        // Base case
        if (N == 1)
            return 4; // 2 for one side and 4 for two sides
    }
}

```

```

// countB is count of ways with a building at the end
// countS is count of ways with a space at the end
// prev_countB and prev_countS are previous values of
// countB and countS respectively.

// Initialize countB and countS for one side
int countB=1, countS=1, prev_countB, prev_countS;

// Use the above recursive formula for calculating
// countB and countS using previous values
for (int i=2; i<=N; i++)
{
    prev_countB = countB;
    prev_countS = countS;

    countS = prev_countB + prev_countS;
    countB = prev_countS;
}

// Result for one side is sum of ways ending with building
// and ending with space
int result = countS + countB;

// Result for 2 sides is square of result for one side
return (result*result);
}

public static void main(String args[])
{
    int N = 3;
    System.out.println("Count of ways for "+ N+" sections is "
                       +countWays(N));
}

/* This code is contributed by Rajat Mishra */

```

Output:

25

Time complexity: O(N)

Auxiliary Space: O(1)

Algorithmic Paradigm: Dynamic Programming

Optimized Solution:

Note that the above solution can be further optimized. If we take closer look at the results, for different values, we can notice that the results for two sides are squares of Fibonacci Numbers.

N = 1, result = 4 [result for one side = 2]
N = 2, result = 9 [result for one side = 3]
N = 3, result = 25 [result for one side = 5]
N = 4, result = 64 [result for one side = 8]
N = 5, result = 169 [result for one side = 13]

.....
.....

In general, we can say

$\text{result}(N) = \text{fib}(N+2)^2$

$\text{fib}(N)$ is a function that returns N'th
Fibonacci Number.

Therefore, we can use O(LogN) implementation of Fibonacci Numbers to find number of ways in O(logN) time.

This article is contributed by **GOPINATH**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming

Maximum profit by buying and selling a share at most twice

In a daily share trading, a buyer buys shares in the morning and sells it on same day. If the trader is allowed to make at most 2 transactions in a day, where as second transaction can only start after first one is complete (Sell->buy->sell->buy). Given stock prices throughout day, find out maximum profit that a share trader could have made.

Examples:

```

Input: price[] = {10, 22, 5, 75, 65, 80}
Output: 87
Trader earns 87 as sum of 12 and 75
Buy at price 10, sell at 22, buy at 5 and sell at 80

Input: price[] = {2, 30, 15, 10, 8, 25, 80}
Output: 100
Trader earns 100 as sum of 28 and 72
Buy at price 2, sell at 30, buy at 8 and sell at 80

Input: price[] = {100, 30, 15, 10, 8, 25, 80}
Output: 72
Buy at price 8 and sell at 80.

Input: price[] = {90, 80, 70, 60, 50}
Output: 0
Not possible to earn.

```

We strongly recommend to minimize your browser and try this yourself first.

A Simple Solution is to consider every index 'i' and do following

```
Max profit with at most two transactions =  
    MAX {max profit with one transaction and subarray price[0..i] +  
          max profit with one transaction and subarray price[i+1..n-1] }  
i varies from 0 to n-1.
```

Maximum possible using one transaction can be calculated using following O(n) algorithm

Maximum difference between two elements such that larger element appears after the smaller number

Time complexity of above simple solution is O(n^2).

We can do this O(n) using following Efficient Solution. The idea is to store maximum possible profit of every subarray and solve the problem in following two phases.

1) Create a table profit[0..n-1] and initialize all values in it 0.

2) Traverse price[] from right to left and update profit[i] such that profit[i] stores maximum profit achievable from one transaction in subarray price[i..n-1]

3) Traverse price[] from left to right and update profit[i] such that profit[i] stores maximum profit such that profit[i] contains maximum achievable profit from two transactions in subarray price[0..i].

4) Return profit[n-1]

To do step 1, we need to keep track of maximum price from right to left side and to do step 2, we need to keep track of minimum price from left to right. Why we traverse in reverse directions? The idea is to save space, in second step, we use same array for both purposes, maximum with 1 transaction and maximum with 2 transactions. After an iteration i, the array profit[0..i] contains maximum profit with 2 transactions and profit[i+1..n-1] contains profit with two transactions.

Below are implementations of above idea.

C++

```
// C++ program to find maximum possible profit with at most  
// two transactions  
#include<iostream>  
using namespace std;  
  
// Returns maximum profit with two transactions on a given  
// list of stock prices, price[0..n-1]  
int maxProfit(int price[], int n)  
{  
    // Create profit array and initialize it as 0  
    int *profit = new int[n];  
    for (int i=0; i<n; i++)  
        profit[i] = 0;  
  
    /* Get the maximum profit with only one transaction  
     allowed. After this loop, profit[i] contains maximum  
     profit from price[i..n-1] using at most one trans. */  
    int max_price = price[n-1];  
    for (int i=n-2; i>=0; i--)  
    {  
        // max_price has maximum of price[i..n-1]  
        if (price[i] > max_price)  
            max_price = price[i];  
  
        // we can get profit[i] by taking maximum of:  
        // a) previous maximum, i.e., profit[i+1]  
        // b) profit by buying at price[i] and selling at  
        //    max_price  
        profit[i] = max(profit[i+1], max_price - price[i]);  
    }  
  
    /* Get the maximum profit with two transactions allowed  
     After this loop, profit[n-1] contains the result */  
    int min_price = price[0];  
    for (int i=1; i<n; i++)  
    {  
        // min_price is minimum price in price[0..i]  
        if (price[i] < min_price)  
            min_price = price[i];  
  
        // Maximum profit is maximum of:  
        // a) previous maximum, i.e., profit[i-1]  
        // b) (Buy, Sell) at (min_price, price[i]) and add  
        //    profit of other trans. stored in profit[i]  
        profit[i] = max(profit[i-1], profit[i] +  
                        (price[i]-min_price));  
    }  
    int result = profit[n-1];  
  
    delete [] profit; // To avoid memory leak  
  
    return result;  
}  
  
// Drive program  
int main()  
{  
    int price[] = {2, 30, 15, 10, 8, 25, 80};  
    int n = sizeof(price)/sizeof(price[0]);  
    cout << "Maximum Profit = " << maxProfit(price, n);  
    return 0;  
}
```

Java

```
class Profit  
{  
    // Returns maximum profit with two transactions on a given  
    // list of stock prices, price[0..n-1]  
    static int maxProfit(int price[], int n)  
{  
        // Create profit array and initialize it as 0
```

```

int profit[] = new int[n];
for (int i=0; i<n; i++)
    profit[i] = 0;

/* Get the maximum profit with only one transaction
allowed. After this loop, profit[i] contains maximum
profit from price[i..n-1] using at most one trans. */
int max_price = price[n-1];
for (int i=n-2;i>=0;--i)
{
    // max_price has maximum of price[i..n-1]
    if (price[i] > max_price)
        max_price = price[i];

    // we can get profit[i] by taking maximum of:
    // a) previous maximum, i.e., profit[i+1]
    // b) profit by buying at price[i] and selling at
    // max_price
    profit[i] = Math.max(profit[i+1], max_price - price[i]);
}

/* Get the maximum profit with two transactions allowed
After this loop, profit[n-1] contains the result */
int min_price = price[0];
for (int i=1; i<n; i++)
{
    // min_price is minimum price in price[0..i]
    if (price[i] < min_price)
        min_price = price[i];

    // Maximum profit is maximum of:
    // a) previous maximum, i.e., profit[i-1]
    // b) (Buy, Sell) at (min_price, price[i]) and add
    // profit of other trans. stored in profit[i]
    profit[i] = Math.max(profit[i-1], profit[i] +
        (price[i]-min_price));
}
int result = profit[n-1];
return result;
}

public static void main(String args[])
{
    int price[] = {2, 30, 15, 10, 8, 25, 80};
    int n = price.length;
    System.out.println("Maximum Profit = " + maxProfit(price, n));
}

/* This code is contributed by Rajat Mishra */

```

Python

```

# Returns maximum profit with two transactions on a given
# list of stock prices price[0..n-1]
def maxProfit(price,n):

    # Create profit array and initialize it as 0
    profit = [0]*n

    # Get the maximum profit with only one transaction
    # allowed. After this loop, profit[i] contains maximum
    # profit from price[i..n-1] using at most one trans.
    max_price=price[n-1]

    for i in range( n-2, 0 ,-1):

        if price[i]> max_price:
            max_price = price[i]

        # we can get profit[i] by taking maximum of:
        # a) previous maximum, i.e., profit[i+1]
        # b) profit by buying at price[i] and selling at
        # max_price
        profit[i] = max(profit[i+1], max_price - price[i])

    # Get the maximum profit with two transactions allowed
    # After this loop, profit[n-1] contains the result
    min_price=price[0]

    for i in range(1,n):

        if price[i] < min_price:
            min_price = price[i]

        # Maximum profit is maximum of:
        # a) previous maximum, i.e., profit[i-1]
        # b) (Buy, Sell) at (min_price, A[i]) and add
        # profit of other trans. stored in profit[i]
        profit[i] = max(profit[i-1], profit[i]+(price[i]-min_price))

    result = profit[n-1]

    return result

# Driver function
price = [2, 30, 15, 10, 8, 25, 80]
print("Maximum profit is", maxProfit(price, len(price)))

# This code is contributed by __Devesh Agrawal__

```

Output:

Maximum Profit = 100

Time complexity of the above solution is $O(n)$.

Algorithmic Paradigm: Dynamic Programming

This article is contributed by **Amit Jaiswal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming

How to print maximum number of A's using given four keys

This is a famous interview question asked in [Google](#), [Paytm](#) and many other company interviews.

Below is the problem statement.

Imagine you have a special keyboard with the following keys:

Key 1: Prints 'A' on screen

Key 2: (Ctrl-A): Select screen

Key 3: (Ctrl-C): Copy selection to buffer

Key 4: (Ctrl-V): Print buffer on screen appending it after what has already been printed.

If you can only press the keyboard for N times (with the above four keys), write a program to produce maximum numbers of A's. That is to say, the input parameter is N (No. of keys that you can press), the output is M (No. of As that you can produce).

Examples:

Input: $N = 3$

Output: 3

We can at most get 3 A's on screen by pressing following key sequence.

A, A, A

Input: $N = 7$

Output: 9

We can at most get 9 A's on screen by pressing following key sequence.

A, A, A, Ctrl A, Ctrl C, Ctrl V, Ctrl V

Input: $N = 11$

Output: 27

We can at most get 27 A's on screen by pressing following key sequence.

A, A, A, Ctrl A, Ctrl C, Ctrl V, Ctrl V, Ctrl A,
Ctrl C, Ctrl V, Ctrl V

We strongly recommend that you click here and practice it, before moving on to the solution.

Below are few important points to note.

a) For $N \leq 6$:

The task is to find out the break-point after which we get the above suffix of keystrokes. Definition of a breakpoint is that instance after which we need to only press Ctrl-A, Ctrl-C once and the only Ctrl-V's afterwards to generate the optimal length. If we loop from $N-3$ to 1 and choose each of these values for the break-point, and compute that optimal string they would produce. Once the loop ends, we will have the maximum of the optimal lengths for various breakpoints, thereby giving us the optimal length for N keystrokes.

Below is C implementation based on above idea.

```
/* A recursive C program to print maximum number of A's using
   following four keys */
#include<stdio.h>

// A recursive function that returns the optimal length string
// for N keystrokes
int findoptimal(int N)
{
    // The optimal string length is N when N is smaller than 7
    if (N <= 6)
        return N;

    // Initialize result
    int max = 0;

    // TRY ALL POSSIBLE BREAK-POINTS
    // For any keystroke N, we need to loop from N-3 keystrokes
    // back to 1 keystroke to find a breakpoint 'b' after which we
    // will have Ctrl-A, Ctrl-C and then only Ctrl-V all the way.
    int b;
    for (b=N-3; b>=1; b--)
    {
        // If the breakpoint is at b'th keystroke then
        // the optimal string would have length
        // (n-b-1)*screen[b-1];
        int curr = (N-b-1)*findoptimal(b);
        if (curr > max)
            max = curr;
    }
    return max;
}

// Driver program
int main()
{
    int N;

    // for the rest of the array we will rely on the previous
```

```

// entries to compute new ones
for (N=1; N<=20; N++)
    printf("Maximum Number of A's with %d keystrokes is %d\n",
        N, findoptimal(N));
}

```

Output:

```

Maximum Number of A's with 1 keystrokes is 1
Maximum Number of A's with 2 keystrokes is 2
Maximum Number of A's with 3 keystrokes is 3
Maximum Number of A's with 4 keystrokes is 4
Maximum Number of A's with 5 keystrokes is 5
Maximum Number of A's with 6 keystrokes is 6
Maximum Number of A's with 7 keystrokes is 9
Maximum Number of A's with 8 keystrokes is 12
Maximum Number of A's with 9 keystrokes is 16
Maximum Number of A's with 10 keystrokes is 20
Maximum Number of A's with 11 keystrokes is 27
Maximum Number of A's with 12 keystrokes is 36
Maximum Number of A's with 13 keystrokes is 48
Maximum Number of A's with 14 keystrokes is 64
Maximum Number of A's with 15 keystrokes is 81
Maximum Number of A's with 16 keystrokes is 108
Maximum Number of A's with 17 keystrokes is 144
Maximum Number of A's with 18 keystrokes is 192
Maximum Number of A's with 19 keystrokes is 256
Maximum Number of A's with 20 keystrokes is 324

```

The above function computes the same subproblems again and again. Recomputations of same subproblems can be avoided by storing the solutions to subproblems and solving problems in bottom up manner.

Below is Dynamic Programming based C implementation where an auxiliary array screen[N] is used to store result of subproblems.

```

/* A Dynamic Programming based C program to find maximum number of A's
   that can be printed using four keys */
#include<stdio.h>

// this function returns the optimal length string for N keystrokes
int findoptimal(int N)
{
    // The optimal string length is N when N is smaller than 7
    if (N <= 6)
        return N;

    // An array to store result of subproblems
    int screen[N];

    int b; // To pick a breakpoint

    // Initializing the optimal lengths array for upto 6 input
    // strokes.
    int n;
    for (n=1; n<=6; n++)
        screen[n-1] = n;

    // Solve all subproblems in bottom manner
    for (n=7; n<=N; n++)
    {
        // Initialize length of optimal string for n keystrokes
        screen[n-1] = 0;

        // For any keystroke n, we need to loop from n-3 keystrokes
        // back to 1 keystroke to find a breakpoint 'b' after which we
        // will have ctrl-a, ctrl-c and then only ctrl-v all the way.
        for (b=n-3; b>=1; b--)
        {
            // if the breakpoint is at b'th keystroke then
            // the optimal string would have length
            // (n-b-1)*screen[b-1];
            int curr = (n-b-1)*screen[b-1];
            if (curr > screen[n-1])
                screen[n-1] = curr;
        }
    }

    return screen[N-1];
}

// Driver program
int main()
{
    int N;

    // for the rest of the array we will rely on the previous
    // entries to compute new ones
    for (N=1; N<=20; N++)
        printf("Maximum Number of A's with %d keystrokes is %d\n",
            N, findoptimal(N));
}

```

Output:

```

Maximum Number of A's with 1 keystrokes is 1
Maximum Number of A's with 2 keystrokes is 2
Maximum Number of A's with 3 keystrokes is 3
Maximum Number of A's with 4 keystrokes is 4
Maximum Number of A's with 5 keystrokes is 5
Maximum Number of A's with 6 keystrokes is 6
Maximum Number of A's with 7 keystrokes is 9
Maximum Number of A's with 8 keystrokes is 12
Maximum Number of A's with 9 keystrokes is 16
Maximum Number of A's with 10 keystrokes is 20
Maximum Number of A's with 11 keystrokes is 27
Maximum Number of A's with 12 keystrokes is 36

```

```
Maximum Number of A's with 13 keystrokes is 48
Maximum Number of A's with 14 keystrokes is 64
Maximum Number of A's with 15 keystrokes is 81
Maximum Number of A's with 16 keystrokes is 108
Maximum Number of A's with 17 keystrokes is 144
Maximum Number of A's with 18 keystrokes is 192
Maximum Number of A's with 19 keystrokes is 256
Maximum Number of A's with 20 keystrokes is 324
```

Thanks to **Gaurav Saxena** for providing the above approach to solve this problem.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming

Find the minimum cost to reach destination using a train

There are N stations on route of a train. The train goes from station 0 to N-1. The ticket cost for all pair of stations (i, j) is given where j is greater than i. Find the minimum cost to reach the destination.

Consider the following example:

```
Input:
cost[N][N] = { {0, 15, 80, 90},
               {INF, 0, 40, 50},
               {INF, INF, 0, 70},
               {INF, INF, INF, 0}
             };
There are 4 stations and cost[i][j] indicates cost to reach j
from i. The entries where j
We strongly recommend to minimize your browser and try this yourself first.
```

The minimum cost to reach N-1 from 0 can be recursively written as following:

```
minCost(0, N-1) = MIN ( cost[0][n-1],
                         cost[0][1] + minCost(1, N-1),
                         minCost(0, 2) + minCost(2, N-1),
                         .....
                         minCost(0, N-2) + cost[N-2][n-1] )
```

The following is the implementation of above recursive formula.

C++

```
// A naive recursive solution to find min cost path from station 0
// to station N-1
#include <iostream>
#include <climits>
using namespace std;

// infinite value
#define INF INT_MAX

// Number of stations
#define N 4

// A recursive function to find the shortest path from
// source 's' to destination 'd'.
int minCostRec(int cost[N][N], int s, int d)
{
    // If source is same as destination
    // or destination is next to source
    if (s == d || s+1 == d)
        return cost[s][d];

    // Initialize min cost as direct ticket from
    // source 's' to destination 'd'.
    int min = cost[s][d];

    // Try every intermediate vertex to find minimum
    for (int i = s+1; i<d; i++)
    {
        int c = minCostRec(cost, s, i) +
               minCostRec(cost, i, d);
        if (c < min)
            min = c;
    }
    return min;
}

// This function returns the smallest possible cost to
// reach station N-1 from station 0. This function mainly
// uses minCostRec().
int minCost(int cost[N][N])
{
    return minCostRec(cost, 0, N-1);
}

// Driver program to test above function
int main()
{
    int cost[N][N] = { {0, 15, 80, 90},
```

```

        {INF, 0, 40, 50},
        {INF, INF, 0, 70},
        {INF, INF, INF, 0}
    };
    cout << "The Minimum cost to reach station "
    << N << " is " << minCost(cost);
    return 0;
}

```

Java

```

// A naive recursive solution to find min cost path from station 0
// to station N-1
class shortest_path
{
    static int INF = Integer.MAX_VALUE, N = 4;
    // A recursive function to find the shortest path from
    // source 's' to destination 'd'.
    static int minCostRec(int cost[][], int s, int d)
    {
        // If source is same as destination
        // or destination is next to source
        if (s == d || s+1 == d)
            return cost[s][d];

        // Initialize min cost as direct ticket from
        // source 's' to destination 'd'.
        int min = cost[s][d];

        // Try every intermediate vertex to find minimum
        for (int i = s+1; i<d; i++)
        {
            int c = minCostRec(cost, s, i) +
                minCostRec(cost, i, d);
            if (c < min)
                min = c;
        }
        return min;
    }

    // This function returns the smallest possible cost to
    // reach station N-1 from station 0. This function mainly
    // uses minCostRec().
    static int minCost(int cost[])
    {
        return minCostRec(cost, 0, N-1);
    }

    public static void main(String args[])
    {
        int cost[][] = {{0, 15, 80, 90},
                       {INF, 0, 40, 50},
                       {INF, INF, 0, 70},
                       {INF, INF, INF, 0}};
        System.out.println("The Minimum cost to reach station "+ N+
                           " is "+minCost(cost));
    }
}

/* This code is contributed by Rajat Mishra */

```

Output:

```
The Minimum cost to reach station 4 is 65
```

Time complexity of the above implementation is exponential as it tries every possible path from 0 to N-1. The above solution solves same subproblems multiple times (it can be seen by drawing recursion tree for minCostRec).

Since this problem has both properties of dynamic programming problems ((see [this](#) and [this](#)). Like other typical [Dynamic Programming\(DP\) problems](#), re-computations of same subproblems can be avoided by storing the results of subproblems.

One dynamic programming solution is to create a 2D table and fill the table using above given recursive formula. The extra space required in this solution would be $O(N^2)$ and time complexity would be $O(N^3)$.

We can solve this problem using $O(N)$ extra space and $O(N^2)$ time. The idea is based on the fact that given input matrix is a Directed Acyclic Graph (DAG). The shortest path in DAG can be calculated using the approach discussed in [Shortest Path in DAG](#).

Shortest Path in Directed Acyclic Graph

We need to do less work here compared to above mentioned post as we know [topological sorting](#) of the graph. The topological sorting of vertices here is 0, 1, ..., N-1. Following is the idea once topological sorting is known.

The idea in below code is to first calculate min cost for station 1, then for station 2, and so on. These costs are stored in an array dist[0...N-1].

1) The min cost for station 0 is 0, i.e., dist[0] = 0

2) The min cost for station 1 is cost[0][1], i.e., dist[1] = cost[0][1]

3) The min cost for station 2 is minimum of following two.

a) dist[0] + cost[0][2]

b) $\text{dist}[1] + \text{cost}[1][2]$

3) The min cost for station 3 is minimum of following three.

a) $\text{dist}[0] + \text{cost}[0][3]$

b) $\text{dist}[1] + \text{cost}[1][3]$

c) $\text{dist}[2] + \text{cost}[2][3]$

Similarly, $\text{dist}[4], \text{dist}[5], \dots, \text{dist}[N-1]$ are calculated.

Below is the implementation of above idea.

C++

```
// A Dynamic Programming based solution to find min cost
// to reach station N-1 from station 0.
#include<iostream>
#include<climits>
using namespace std;

#define INF INT_MAX
#define N 4

// This function returns the smallest possible cost to
// reach station N-1 from station 0.
int minCost(int cost[][N])
{
    // dist[] stores minimum cost to reach station i
    // from station 0.
    int dist[N];
    for (int i=0; i<N; i++)
        dist[i] = INF;
    dist[0] = 0;

    // Go through every station and check if using it
    // as an intermediate station gives better path
    for (int i=0; i<N; i++)
        for (int j=i+1; j<N; j++)
            if (dist[j] > dist[i] + cost[i][j])
                dist[j] = dist[i] + cost[i][j];

    return dist[N-1];
}

// Driver program to test above function
int main()
{
    int cost[N][N] = { {0, 15, 80, 90},
                      {INF, 0, 40, 50},
                      {INF, INF, 0, 70},
                      {INF, INF, INF, 0}
                    };
    cout << "The Minimum cost to reach station "
         << N << " is " << minCost(cost);
    return 0;
}
```

Java

```
// A Dynamic Programming based solution to find min cost
// to reach station N-1 from station 0.
class shortest_path
{

    static int INF = Integer.MAX_VALUE,N = 4;
    // A recursive function to find the shortest path from
    // source 's' to destination 'd'.

    // This function returns the smallest possible cost to
    // reach station N-1 from station 0.
    static int minCost(int cost[][])
    {
        // dist[] stores minimum cost to reach station i
        // from station 0.
        int dist[] = new int[N];
        for (int i=0; i<N; i++)
            dist[i] = INF;
        dist[0] = 0;

        // Go through every station and check if using it
        // as an intermediate station gives better path
        for (int i=0; i<N; i++)
            for (int j=i+1; j<N; j++)
                if (dist[j] > dist[i] + cost[i][j])
                    dist[j] = dist[i] + cost[i][j];
    }
}
```

```

        return dist[N-1];
    }

    public static void main(String args[])
    {
        int cost[][] = { { 0, 15, 80, 90 },
                        { INF, 0, 40, 50 },
                        { INF, INF, 0, 70 },
                        { INF, INF, INF, 0 }
                    };
        System.out.println("The Minimum cost to reach station " + N +
                           " is " + minCost(cost));
    }

/* This code is contributed by Rajat Mishra */

```

Output:

The Minimum cost to reach station 4 is 65

This article is contributed by **Udit Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner

Company Wise Coding Practice



Vertex Cover Problem | Set 2 (Dynamic Programming Solution for Tree)

A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either ' u ' or ' v ' is in vertex cover. Although the name is Vertex Cover, the set covers all edges of the given graph.

The problem to find minimum size vertex cover of a graph is NP complete. But it can be solved in polynomial time for trees. In this post a solution for Binary Tree is discussed. The same solution can be extended for n-ary trees.

For example, consider the following binary tree. The smallest vertex cover is {20, 50, 30} and size of the vertex cover is 3.

LargestIndependentSet1



The idea is to consider following two possibilities for root and recursively for all nodes down the root.

1) Root is part of vertex cover: In this case root covers all children edges. We recursively calculate size of vertex covers for left and right subtrees and add 1 to the result (for root).

2) Root is not part of vertex cover: In this case, both children of root must be included in vertex cover to cover all root to children edges. We recursively calculate size of vertex covers of all grandchildren and number of children to the result (for two children of root).

Below is C implementation of above idea.

```

// A naive recursive C implementation for vertex cover problem for a tree
#include <stdio.h>
#include <stdlib.h>

// A utility function to find min of two integers
int min(int x, int y) { return (x < y)? x: y; }

/* A binary tree node has data, pointer to left child and a pointer to
right child */
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

```

```

    struct node *left, *right;
};

// The function returns size of the minimum vertex cover
int vCover(struct node *root)
{
    // The size of minimum vertex cover is zero if tree is empty or there
    // is only one node
    if (root == NULL)
        return 0;
    if (root->left == NULL && root->right == NULL)
        return 0;

    // Calculate size of vertex cover when root is part of it
    int size_incl = 1 + vCover(root->left) + vCover(root->right);

    // Calculate size of vertex cover when root is not part of it
    int size_excl = 0;
    if (root->left)
        size_excl += 1 + vCover(root->left->left) + vCover(root->left->right);
    if (root->right)
        size_excl += 1 + vCover(root->right->left) + vCover(root->right->right);

    // Return the minimum of two sizes
    return min(size_incl, size_excl);
}

// A utility function to create a node
struct node* newNode( int data )
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree given in the above diagram
    struct node *root      = newNode(20);
    root->left       = newNode(8);
    root->left->left   = newNode(4);
    root->left->right  = newNode(12);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(14);
    root->right      = newNode(22);
    root->right->right = newNode(25);

    printf ("Size of the smallest vertex cover is %d ", vCover(root));

    return 0;
}

```

Output:

```
Size of the smallest vertex cover is 3
```

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. For example, vCover of node with value 50 is evaluated twice as 50 is grandchild of 10 and child of 20.

Since same subproblems are called again, this problem has Overlapping Subproblems property. So Vertex Cover problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem. Like other typical [Dynamic Programming\(DP\) problems](#), re-computations of same subproblems can be avoided by storing the solutions to subproblems and solving problems in bottom up manner.

Following is C implementation of Dynamic Programming based solution. In the following solution, an additional field 'vc' is added to tree nodes. The initial value of 'vc' is set as 0 for all nodes. The recursive function vCover() calculates 'vc' for a node only if it is not already set.

```

/* Dynamic programming based program for Vertex Cover problem for
 a Binary Tree */
#include <stdio.h>
#include <stdlib.h>

// A utility function to find min of two integers
int min(int x, int y) { return (x < y)? x: y; }

/* A binary tree node has data, pointer to left child and a pointer to
 right child */
struct node
{
    int data;
    int vc;
    struct node *left, *right;
};

// A memoization based function that returns size of the minimum vertex cover.
int vCover(struct node *root)
{
    // The size of minimum vertex cover is zero if tree is empty or there
    // is only one node
    if (root == NULL)
        return 0;
    if (root->left == NULL && root->right == NULL)
        return 0;

    // If vertex cover for this node is already evaluated, then return it
    // to save recomputation of same subproblem again.
    if (root->vc != 0)
        return root->vc;

    // Calculate size of vertex cover when root is part of it
    int size_incl = 1 + vCover(root->left) + vCover(root->right);

    // Calculate size of vertex cover when root is not part of it
    int size_excl = 0;
    if (root->left)
        size_excl += 1 + vCover(root->left->left) + vCover(root->left->right);

```

```

if (root->right)
    size_excl += 1 + vCover(root->right->left) + vCover(root->right->right);

// Minimum of two values is vertex cover, store it before returning
root->vc = min(size_incl, size_excl);

return root->vc;
}

// A utility function to create a node
struct node* newNode( int data )
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );
    temp->data = data;
    temp->left = temp->right = NULL;
    temp->vc = 0; // Set the vertex cover as 0
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree given in the above diagram
    struct node *root      = newNode(20);
    root->left       = newNode(8);
    root->left->left   = newNode(4);
    root->left->right  = newNode(12);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(14);
    root->right      = newNode(22);
    root->right->right = newNode(25);

    printf ("Size of the smallest vertex cover is %d ", vCover(root));

    return 0;
}

```

Output:

Size of the smallest vertex cover is 3

References:

<http://courses.csail.mit.edu/6.006/spring11/lectures/lec21.pdf>

Exercise:

Extend the above solution for n-ary trees.

This article is contributed by **Udit Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming
NPHard

Count number of ways to reach a given score in a game

Consider a game where a player can score 3 or 5 or 10 points in a move. Given a total score n, find number of ways to reach the given score.

Examples:

```

Input: n = 20
Output: 4
There are following 4 ways to reach 20
(10, 10)
(5, 5, 10)
(5, 5, 5)
(3, 3, 3, 3, 3, 5)

```

```

Input: n = 13
Output: 2
There are following 2 ways to reach 13
(3, 5, 5)
(3, 10)

```

This problem is a variation of [coin change problem](#) and can be solved in O(n) time and O(n) auxiliary space.

The idea is to create a table of size n+1 to store counts of all scores from 0 to n. For every possible move (3, 5 and 10), increment values in table.

```

// A C program to count number of possible ways to a given score
// can be reached in a game where a move can earn 3 or 5 or 10
#include <stdio.h>

// Returns number of ways to reach score n
int count(int n)
{
    // table[i] will store count of solutions for
    // value i.
    int table[n+1], i;

    // Initialize all table values as 0
    memset(table, 0, sizeof(table));

    // Base case (If given value is 0)
    table[0] = 1;

    // One by one consider given 3 moves and update the table[]
    // values after the index greater than or equal to the
    // value of the picked move
    for (i=3; i<=n; i++)
        table[i] += table[i-3];
    for (i=5; i<=n; i++)
        table[i] += table[i-5];
    for (i=10; i<=n; i++)
        table[i] += table[i-10];
}

```

```

        table[i] += table[i-10];

    return table[n];
}

// Driver program
int main(void)
{
    int n = 20;
    printf("Count for %d is %d\n", n, count(n));

    n = 13;
    printf("Count for %d is %d", n, count(n));
    return 0;
}

```

Output:

```

Count for 20 is 4
Count for 13 is 2

```

Exercise: How to count score when (10, 5, 5), (5, 5, 10) and (5, 10, 5) are considered as different sequences of moves. Similarly, (5, 3, 3), (3, 5, 3) and (3, 3, 5) are considered different.

This article is contributed by **Rajeev Arora**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming

Weighted Job Scheduling

Given N jobs where every job is represented by following three elements of it.

1. Start Time
2. Finish Time
3. Profit or Value Associated

Find the maximum profit subset of jobs such that no two jobs in the subset overlap.

Example:

```

Input: Number of Jobs n = 4
Job Details (Start Time, Finish Time, Profit)
Job 1: {1, 2, 50}
Job 2: {3, 5, 20}
Job 3: {6, 19, 100}
Job 4: {2, 100, 200}
Output: The maximum profit is 250.
We can get the maximum profit by scheduling jobs 1 and 4.
Note that there is longer schedules possible Jobs 1, 2 and 3
but the profit with this schedule is 20+50+100 which is less than 250.

```

A simple version of this problem is discussed [here](#) where every job has same profit or value. The [Greedy Strategy for activity selection](#) doesn't work here as a schedule with more jobs may have smaller profit or value.

The above problem can be solved using following recursive solution.

```

1) First sort jobs according to finish time.
2) Now apply following recursive process.
// Here arr[] is array of n jobs
findMaximumProfit(arr[], n)
{
    a) if (n == 1) return arr[0];
    b) Return the maximum of following two profits.
        (i) Maximum profit by excluding current job, i.e.,
            findMaximumProfit(arr, n-1)
        (ii) Maximum profit by including the current job
}

```

How to find the profit including current job?

The idea is to find the latest job before the current job (in sorted array) that doesn't conflict with current job [arr[n-1]]. Once we find such a job, we recur for all jobs till that job and add profit of current job to result.
In the above example, "job 1" is the latest non-conflicting for "job 4" and "job 2" is the latest non-conflicting for "job 3".

The following is C++ implementation of above naive recursive method.

```

// C++ program for weighted job scheduling using Naive Recursive Method
#include <iostream>
#include <algorithm>
using namespace std;

// A job has start time, finish time and profit.
struct Job
{
    int start, finish, profit;
};

// A utility function that is used for sorting events
// according to finish time
bool jobComparatoar(Job s1, Job s2)
{
    return (s1.finish < s2.finish);
}

// Find the latest job (in sorted array) that doesn't
// conflict with the job[i]. If there is no compatible job,

```

```

// then it returns -1.
int latestNonConflict(Job arr[], int i)
{
    for (int j=i-1; j>=0; j--)
    {
        if (arr[j].finish <= arr[i-1].start)
            return j;
    }
    return -1;
}

// A recursive function that returns the maximum possible
// profit from given array of jobs. The array of jobs must
// be sorted according to finish time.
int findMaxProfitRec(Job arr[], int n)
{
    // Base case
    if (n == 1) return arr[n-1].profit;

    // Find profit when current job is included
    int inclProf = arr[n-1].profit;
    int i = latestNonConflict(arr, n);
    if (i != -1)
        inclProf += findMaxProfitRec(arr, i+1);

    // Find profit when current job is excluded
    int exclProf = findMaxProfitRec(arr, n-1);

    return max(inclProf, exclProf);
}

// The main function that returns the maximum possible
// profit from given array of jobs
int findMaxProfit(Job arr[], int n)
{
    // Sort jobs according to finish time
    sort(arr, arr+n, jobComparataor);

    return findMaxProfitRec(arr, n);
}

// Driver program
int main()
{
    Job arr[] = {{3, 10, 20}, {1, 2, 50}, {6, 19, 100}, {2, 100, 200}};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "The optimal profit is " << findMaxProfit(arr, n);
    return 0;
}

```

Output:

```
The optimal profit is 250
```

The above solution may contain many overlapping subproblems. For example if `lastNonConflicting()` always returns previous job, then `findMaxProfitRec(arr, n-1)` is called twice and the time complexity becomes $O(n^2)$. As another example when `lastNonConflicting()` returns previous to previous job, there are two recursive calls, for $n-2$ and $n-1$. In this example case, recursion becomes same as Fibonacci Numbers.

So this problem has both properties of Dynamic Programming, [Optimal Substructure](#) and [Overlapping Subproblems](#).

Like other Dynamic Programming Problems, we can solve this problem by making a table that stores solution of subproblems.

Below is C++ implementation based on Dynamic Programming.

```

// C++ program for weighted job scheduling using Dynamic Programming.
#include <iostream>
#include <algorithm>
using namespace std;

// A job has start time, finish time and profit.
struct Job
{
    int start, finish, profit;
};

// A utility function that is used for sorting events
// according to finish time
bool jobComparataor(Job s1, Job s2)
{
    return (s1.finish < s2.finish);
}

// Find the latest job (in sorted array) that doesn't
// conflict with the job[i]
int latestNonConflict(Job arr[], int i)
{
    for (int j=i-1; j>=0; j--)
    {
        if (arr[j].finish <= arr[i].start)
            return j;
    }
    return -1;
}

// The main function that returns the maximum possible
// profit from given array of jobs
int findMaxProfit(Job arr[], int n)
{
    // Sort jobs according to finish time
    sort(arr, arr+n, jobComparataor);

    // Create an array to store solutions of subproblems. table[i]
    // stores the profit for jobs till arr[i] (including arr[i])
    int *table = new int[n];
    table[0] = arr[0].profit;

    // Fill entries in M[] using recursive property

```

```

for (int i=1; i<n; i++)
{
    // Find profit including the current job
    int inclProf = arr[i].profit;
    int l = latestNonConflict(arr, i);
    if (l != -1)
        inclProf += table[l];

    // Store maximum of including and excluding
    table[i] = max(inclProf, table[i-1]);
}

// Store result and free dynamic memory allocated for table[]
int result = table[n-1];
delete[] table;

return result;
}

// Driver program
int main()
{
    Job arr[] = {{3, 10, 20}, {1, 2, 50}, {6, 19, 100}, {2, 100, 200}};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "The optimal profit is " << findMaxProfit(arr, n);
    return 0;
}

```

Output:

The optimal profit is 250

Time Complexity of the above Dynamic Programming Solution is $O(n^2)$. Note that the above solution can be optimized to $O(n \log n)$ using Binary Search in `latestNonConflict()` instead of linear search. Thanks to Garvit for suggesting this optimization. Please refer below post for details.

[Weighted Job Scheduling in \$O\(n \log n\)\$ time](#)

References:

<http://courses.cs.washington.edu/courses/cse521/13wi/slides/06dp-sched.pdf>

This article is contributed by Shivam. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming

Longest Even Length Substring such that Sum of First and Second Half is same

Given a string 'str' of digits, find length of the longest substring of 'str', such that the length of the substring is $2k$ digits and sum of left k digits is equal to the sum of right k digits.

Examples:

```

Input: str = "123123"
Output: 6
The complete string is of even length and sum of first and second
half digits is same

Input: str = "1538023"
Output: 4
The longest substring with same first and second half sum is "5380"

```

We strongly recommend that you click here and practice it, before moving on to the solution.

Simple Solution [$O(n^3)$]

A Simple Solution is to check every substring of even length. The following is C based implementation of simple approach.

```

// A simple C based program to find length of longest even length
// substring with same sum of digits in left and right
#include<stdio.h>
#include<string.h>

int findLength(char *str)
{
    int n = strlen(str);
    int maxlen = 0; // Initialize result

    // Choose starting point of every substring
    for (int i=0; i<n; i++)
    {
        // Choose ending point of even length substring
        for (int j=i+1; j<n; j += 2)
        {
            int length = j-i+1; //Find length of current substr

            // Calculate left & right sums for current substr
            int leftsum = 0, rightsum = 0;
            for (int k=0; k<length/2; k++)
            {
                leftsum += (str[i+k]-0);
                rightsum += (str[i+k+length/2]-0);
            }

            // Update result if needed
            if (leftsum == rightsum && maxlen < length)
                maxlen = length;
        }
    }
    return maxlen;
}

```

```

}

// Driver program to test above function
int main(void)
{
    char str[] = "1538023";
    printf("Length of the substring is %d", findLength(str));
    return 0;
}

```

Output:

```
Length of the substring is 4
```

Dynamic Programming [O(n²) and O(n²) extra space]

The above solution can be optimized to work in O(n²) using **Dynamic Programming**. The idea is to build a 2D table that stores sums of substrings. The following is C based implementation of Dynamic Programming approach.

```

// A C based program that uses Dynamic Programming to find length of the
// longest even substring with same sum of digits in left and right half
#include <stdio.h>
#include <string.h>

int findLength(char *str)
{
    int n = strlen(str);
    int maxlen = 0; // Initialize result

    // A 2D table where sum[i][j] stores sum of digits
    // from str[i] to str[j]. Only filled entries are
    // the entries where j >= i
    int sum[n][n];

    // Fill the diagonal values for substrings of length 1
    for (int i = 0; i < n; i++)
        sum[i][i] = str[i] - '0';

    // Fill entries for substrings of length 2 to n
    for (int len = 2; len <= n; len++)
    {
        // Pick i and j for current substring
        for (int i = 0; i < n - len + 1; i++)
        {
            int j = i + len - 1;
            int k = len / 2;

            // Calculate value of sum[i][j]
            sum[i][j] = sum[i][j - k] + sum[j - k + 1][j];

            // Update result if 'len' is even, left and right
            // sums are same and len is more than maxlen
            if ((len % 2 == 0) && sum[i][j - k] == sum[j - k + 1][j] && len > maxlen)
                maxlen = len;
        }
    }
    return maxlen;
}

// Driver program to test above function
int main(void)
{
    char str[] = "153803";
    printf("Length of the substring is %d", findLength(str));
    return 0;
}

```

Output:

```
Length of the substring is 4
```

Time complexity of the above solution is O(n²), but it requires O(n²) extra space.

[A O(n²) and O(n) extra space solution]

The idea is to use a single dimensional array to store cumulative sum.

```

// A O(n^2) time and O(n) extra space solution
#include <bits/stdc++.h>
using namespace std;

int findLength(string str, int n)
{
    int sum[n + 1]; // To store cumulative sum from first digit to nth digit
    sum[0] = 0;

    /* Store cumulative sum of digits from first to last digit */
    for (int i = 1; i <= n; i++)
        sum[i] = (sum[i - 1] + str[i] - '0'); /* convert chars to int */

    int ans = 0; // Initialize result

    /* consider all even length substrings one by one */
    for (int len = 2; len <= n; len += 2)
    {
        for (int i = 0; i <= n - len; i++)
        {
            int j = i + len - 1;

            /* Sum of first and second half is same than update ans */
            if (sum[i + len / 2] - sum[i] == sum[i + len] - sum[i + len / 2])
                ans = max(ans, len);
        }
    }
}

```

```

    }
    return ans;
}

// Driver program to test above function
int main()
{
    string str = "123123";
    cout << "Length of the substring is " << findLength(str, str.length());
    return 0;
}

```

Output:

```
Length of the substring is 6
```

Thanks to Gaurav Ahirwar for suggesting this method.

[A O(n²) time and O(1) extra space solution]
The idea is to consider all possible mid points (of even length substrings) and keep expanding on both sides to get and update optimal length as the sum of two sides become equal.

Below is C++ implementation of the above idea.

```

// A O(n^2) time and O(1) extra space solution
#include<bits/stdc++.h>
using namespace std;

int findLength(string str, int n)
{
    int ans = 0; // Initialize result

    // Consider all possible midpoints one by one
    for (int i = 0; i <= n-2; i++)
    {
        /* For current midpoint 'i', keep expanding substring on
        both sides, if sum of both sides becomes equal update
        ans */
        int l = i, r = i + 1;

        /* initialize left and right sum */
        int lsum = 0, rsum = 0;

        /* move on both sides till indexes go out of bounds */
        while (r < n && l >= 0)
        {
            lsum += str[l] - '0';
            rsum += str[r] - '0';
            if (lsum == rsum)
                ans = max(ans, r-l+1);
            l--;
            r++;
        }
    }
    return ans;
}

// Driver program to test above function
int main()
{
    string str = "123123";
    cout << "Length of the substring is " << findLength(str, str.length());
    return 0;
}

```

Output:

```
Length of the substring is 6
```

Thanks to Gaurav Ahirwar for suggesting this method.

This article is contributed by **Ashish Bansal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming

Searching for Patterns | Set 1 (Naive Pattern Searching)

Given a text $txt[0..n-1]$ and a pattern $pat[0..m-1]$, write a function $search(char pat[], char txt[])$ that prints all occurrences of $pat[]$ in $txt[]$. You may assume that $n > m$.

Examples:

```

Input: txt[] = "THIS IS A TEST TEXT"
       pat[] = "TEST"
Output: Pattern found at index 10

Input: txt[] = "AABAACAADAABAABA"
       pat[] = "AABA"
Output: Pattern found at index 0
       Pattern found at index 9
       Pattern found at index 12

```

pattern-searching



Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

We strongly recommend that you click here and practice it, before moving on to the solution.

Naive Pattern Searching:

Slide the pattern over text one by one and check for a match. If a match is found, then slides by 1 again to check for subsequent matches.

C

```
// C program for Naive Pattern Searching algorithm
#include<stdio.h>
#include<string.h>

void search(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    /* A loop to slide pat[] one by one */
    for (int i = 0; i <= N - M; i++)
    {
        int j;

        /* For current index i, check for pattern match */
        for (j = 0; j < M; j++)
            if (txt[i+j] != pat[j])
                break;

        if (j == M) // if pat[0..M-1] = txt[i, i+1, ...i+M-1]
            printf("Pattern found at index %d \n", i);
    }
}

/* Driver program to test above function */
int main()
{
    char txt[] = "AABAACACADAABAAABAA";
    char pat[] = "AABA";
    search(pat, txt);
    return 0;
}
```

Python

```
# Python program for Naive Pattern Searching
def search(pat, txt):
    M = len(pat)
    N = len(txt)

    # A loop to slide pat[] one by one
    for i in range(N-M+1):

        # For current index i, check for pattern match
        for j in range(M):
            if txt[i+j] != pat[j]:
                break

        if j == M-1: # if pat[0..M-1] = txt[i, i+1, ...i+M-1]
            print "Pattern found at index " + str(i)

# Driver program to test the above function
txt = "AABAACACADAABAAABAA"
pat = "AABA"
search (pat, txt)

# This code is contributed by Bhavya Jain
```

Output:

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

What is the best case?

The best case occurs when the first character of the pattern is not present in text at all.

```
txt[] = "AABCCAADDEE"
pat[] = "FAA"
```

The number of comparisons in best case is O(n).

What is the worst case ?

The worst case of Naive Pattern Searching occurs in following scenarios.

- 1) When all characters of the text and pattern are same.

```
txt[] = "AAAAAAAAAAAAAAA"  
pat[] = "AAAAA".
```

2) Worst case also occurs when only the last character is different.

```
txt[] = "AAAAAAAAAAAAAAAB"  
pat[] = "AAAAB"
```

Number of comparisons in worst case is $O(m*(n-m+1))$. Although strings which have repeated characters are not likely to appear in English text, they may well occur in other applications (for example, in binary texts). The KMP matching algorithm improves the worst case to $O(n)$. We will be covering KMP in the next post. Also, we will be writing more posts to cover all pattern searching algorithms and data structures.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Pattern Searching
Pattern Searching

Searching for Patterns | Set 2 (KMP Algorithm)

Given a text $txt[0..n-1]$ and a pattern $pat[0..m-1]$, write a function $search(char pat[], char txt[])$ that prints all occurrences of $pat[]$ in $txt[]$. You may assume that $n > m$.

Examples:

```
Input: txt[] = "THIS IS A TEST TEXT"  
       pat[] = "TEST"
```

Output: Pattern found at index 10

```
Input: txt[] = "AABAACACDABABA"  
       pat[] = "ABA"
```

Output: Pattern found at index 0

Pattern found at index 9

Pattern found at index 12

pattern-searching



We strongly recommend that you click here and practice it, before moving on to the solution.

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

We have discussed Naive pattern searching algorithm in the [previous post](#). The worst case complexity of Naive algorithm is $O(m(n-m+1))$. Time complexity of KMP algorithm is $O(n)$ in worst case.

KMP (Knuth Morris Pratt) Pattern Searching

The [Naive pattern searching algorithm](#) doesn't work well in cases where we see many matching characters followed by a mismatching character. Following are some examples.

```
txt[] = "AAAAAAAAAAAAAAAB"  
pat[] = "AAAAB"
```

```
txt[] = "ABABABCABABABCABABC"  
pat[] = "ABABAC" (not a worst case, but a bad case for Naive)
```

The KMP matching algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst case complexity to $O(n)$. The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of next window. We take advantage of this information to avoid matching the characters that we know will anyway match. Let us consider below example to understand this.

Matching Overview

```
txt = "AAAAABAAABA"  
pat = "AAA"
```

We compare first window of **txt** with **pat**

```
txt = "AAAAABAAABA"  
pat = "AAA" [Initial position]
```

We find a match. This is same as [Naive String Matching](#).

In the next step, we compare next window of **txt** with **pat**.

```
txt = "AAAAAABAAABA"  
pat = "AAA" [Pattern shifted one position]
```

This is where KMP does optimization over Naive. In this second window, we only compare fourth **A** of pattern with fourth character of current window of text to decide whether current window matches or not. Since we know first three characters will anyway match, we skipped matching first three characters.

Need of Preprocessing?

An important question arises from above explanation, how to know how many characters to be skipped. To know this, we pre-process pattern and prepare an integer array **lps[]** that tells us count of characters to be skipped.

Preprocessing Overview:

- KMP algorithm does preprocesses **pat[]** and constructs an auxiliary **lps[]** of size m (same as size of pattern) which is used to skip characters while matching.
- **name lps indicates longest proper prefix which is also suffix.** A proper prefix is prefix with whole string **not allowed**. For example, prefixes of "ABC" are "", "A", "AB" and "ABC". Proper prefixes are "", "A" and "AB". Suffixes of the string are "", "C", "BC" and "ABC".
- For each sub-pattern $pat[0..i]$ where $i = 0$ to $m-1$, $lps[i]$ stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern $pat[0..i]$.

```
lps[i] = the longest proper prefix of pat[0..i]  
which is also a suffix of pat[0..i].
```

Note : `lps[i]` could also be defined as longest prefix which is also proper suffix. We need to use proper at one place to make sure that the whole substring is not considered.

Examples of `lps[]` construction:

For the pattern "AAAAA",

`lps[]` is [0, 1, 2, 3]

For the pattern "ABCDE",

`lps[]` is [0, 0, 0, 0, 0]

For the pattern "AABAACAAABAA",

`lps[]` is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]

For the pattern "AAACAAAAAC",

`lps[]` is [0, 1, 2, 0, 1, 2, 3, 3, 4]

For the pattern "AAABAAA",

`lps[]` is [0, 1, 2, 0, 1, 2, 3]

Searching Algorithm:

Unlike [Naive algorithm](#), where we slide the pattern by one and compare all characters at each shift, we use a value from `lps[]` to decide the next characters to be matched. The idea is to not match character that we know will anyway match.

How to use `lps[]` to decide next positions (or to know number of characters to be skipped)?

- We start comparison of `pat[i]` with $j = 0$ with characters of current window of text.
- We keep matching characters `txt[i]` and `pat[j]` and keep incrementing i and j while `pat[j]` and `txt[i]` keep **matching**.
- When we see a **mismatch**
 - We know that characters `pat[0..j-1]` match with `txt[i-j+1...i-1]` (Note that j starts with 0 and increment it only when there is a match).
 - We also know (from above definition) that `lps[j-1]` is count of characters of `pat[0..j-1]` that are both proper prefix and suffix.
 - From above two points, we can conclude that we do not need to match these `lps[j-1]` characters with `txt[i-j...i-1]` because we know that these characters will anyway match. Let us consider above example to understand this.

```
txt[] = "AAAAAABAAABA"  
pat[] = "AAAA"  
lps[] = {0, 1, 2, 3}
```

```
i = 0, j = 0  
txt[] = "AAAAAABAAABA"  
pat[] = "AAAA"  
txt[] and pat[] match, do i++, j++
```

```
i = 1, j = 1  
txt[] = "AAAAAABAAABA"  
pat[] = "AAAA"  
txt[] and pat[] match, do i++, j++
```

```
i = 2, j = 2  
txt[] = "AAAAAABAAABA"  
pat[] = "AAAA"  
pat[] and pat[j] match, do i++, j++
```

```
i = 3, j = 3  
txt[] = "AAAAABAAABA"  
pat[] = "AAA"  
txt[] and pat[] match, do i++, j++
```

```
i = 4, j = 4  
Since j == M, print pattern found and reset j,  
j = lps[j-1] = lps[3] = 3
```

Here unlike Naive algorithm, we do not match first three characters of this window. Value of `lps[j-1]` (in above step) gave us index of next character to match.

```
i = 4, j = 3  
txt[] = "AAAAABAAABA"  
pat[] = "AAAA"  
txt[] and pat[] match, do i++, j++
```

```
i = 5, j = 4  
Since j == M, print pattern found and reset j,  
j = lps[j-1] = lps[3] = 3
```

Again unlike Naive algorithm, we do not match first three characters of this window. Value of `lps[j-1]` (in above step) gave us index of next character to match.

```
i = 5, j = 3  
txt[] = "AAAAAABAAABA"  
pat[] = "AAAA"  
txt[] and pat[] do NOT match and j > 0, change only j  
j = lps[j-1] = lps[2] = 2
```

```
i = 5, j = 2  
txt[] = "AAAAAABAAABA"  
pat[] = "AAAA"  
txt[] and pat[] do NOT match and j > 0, change only j  
j = lps[j-1] = lps[1] = 1
```

```
i = 5, j = 1  
txt[] = "AAAAAABAAABA"  
pat[] = "AAAA"  
txt[] and pat[] do NOT match and j > 0, change only j  
j = lps[j-1] = lps[0] = 0
```

```
i = 5, j = 0  
txt[] = "AAAAAABAAABA"  
pat[] = "AAAA"  
txt[] and pat[] do NOT match and j is 0, we do i++.
```

```
i = 6, j = 0  
txt[] = "AAAAAABAAABA"  
pat[] = "AAAA"  
txt[] and pat[] match, do i++ and j++
```

```

i = 7, j = 1
txt[] = "AAAAAABAAABA"
pat[] = "AAAA"
txt[i] and pat[j] match, do i++ and j++

```

We continue this way...

C++

```

// C++ program for implementation of KMP pattern searching
// algorithm
#include<bits/stdc++.h>

void computeLPSArray(char *pat, int M, int *lps);

// Prints occurrences of txt[] in pat[]
void KMPSearch(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    // create lps[] that will hold the longest prefix suffix
    // values for pattern
    int lps[M];

    // Preprocess the pattern (calculate lps[] array)
    computeLPSArray(pat, M, lps);

    int i = 0; // index for txt[]
    int j = 0; // index for pat[]
    while (i < N)
    {
        if (pat[j] == txt[i])
        {
            j++;
            i++;
        }

        if (j == M)
        {
            printf("Found pattern at index %d \n", i-j);
            j = lps[j-1];
        }

        // mismatch after j matches
        else if (i < N && pat[j] != txt[i])
        {
            // Do not match lps[0..lps[j-1]] characters,
            // they will match anyway
            if (j != 0)
                j = lps[j-1];
            else
                i = i+1;
        }
    }

    // Fills lps[] for given pattern pat[0..M-1]
    void computeLPSArray(char *pat, int M, int *lps)
    {
        // length of the previous longest prefix suffix
        int len = 0;

        lps[0] = 0; // lps[0] is always 0

        // the loop calculates lps[i] for i = 1 to M-1
        int i = 1;
        while (i < M)
        {
            if (pat[i] == pat[len])
            {
                len++;
                lps[i] = len;
                i++;
            }
            else // (pat[i] != pat[len])
            {
                // This is tricky. Consider the example.
                // AACACAAA and i = 7. The idea is similar
                // to search step.
                if (len != 0)
                {
                    len = lps[len-1];
                }
                // Also, note that we do not increment
                // i here
            }
            else // if (len == 0)
            {
                lps[i] = 0;
                i++;
            }
        }
    }

    // Driver program to test above function
    int main()
    {
        char *txt = "ABABDABACDABABCABAB";
        char *pat = "ABABCABAB";
        KMPSearch(pat, txt);
        return 0;
    }
}

```

Java

```
// JAVA program for implementation of KMP pattern
// searching algorithm

class KMP_String_Matching
{
    void KMPSearch(String pat, String txt)
    {
        int M = pat.length();
        int N = txt.length();

        // create lps[] that will hold the longest
        // prefix suffix values for pattern
        int lps[] = new int[M];
        int j = 0; // index for pat[]

        // Preprocess the pattern (calculate lps[])
        // array)
        computeLPSArray(pat,M,lps);

        int i = 0; // index for txt[]
        while (i < N)
        {
            if (pat.charAt(j) == txt.charAt(i))
            {
                j++;
                i++;
            }
            if (j == M)
            {
                System.out.println("Found pattern "+
                    "at index "+(i-j));
                j = lps[j-1];
            }

            // mismatch after j matches
            else if (i < N && pat.charAt(j) != txt.charAt(i))
            {
                // Do not match lps[0..lps[j]-1] characters,
                // they will match anyway
                if (j != 0)
                    j = lps[j-1];
                else
                    i = i+1;
            }
        }

        void computeLPSArray(String pat, int M, int lps[])
        {
            // length of the previous longest prefix suffix
            int len = 0;
            int i = 1;
            lps[0] = 0; // lps[0] is always 0

            // the loop calculates lps[i] for i = 1 to M-1
            while (i < M)
            {
                if (pat.charAt(i) == pat.charAt(len))
                {
                    len++;
                    lps[i] = len;
                    i++;
                }
                else // (pat[i] != pat[len])
                {
                    // This is tricky. Consider the example.
                    // AACACAAAA and i = 7. The idea is similar
                    // to search step.
                    if (len != 0)
                    {
                        len = lps[len-1];

                        // Also, note that we do not increment
                        // i here
                    }
                    else // if (len == 0)
                    {
                        lps[i] = len;
                        i++;
                    }
                }
            }

            // Driver program to test above function
            public static void main(String args[])
            {
                String txt = "ABABDABACDABABCABAB";
                String pat = "ABABCABAB";
                new KMP_String_Matching().KMPSearch(pat,txt);
            }
        }
    }
}

// This code has been contributed by Amit Khandelwal.
```

Python

```
# Python program for KMP Algorithm
def KMPSearch(pat, txt):
    M = len(pat)
    N = len(txt)
```

```

# create lps[] that will hold the longest prefix suffix
# values for pattern
lps = [0]*M
j = 0 # index for pat[]

# Preprocess the pattern (calculate lps[] array)
computeLPSArray(pat, M, lps)

i = 0 # index for txt[]
while i < N:
    if pat[j] == txt[i]:
        i += 1
        j += 1

    if j == M:
        print "Found pattern at index " + str(i-j)
        j = lps[j-1]

    # mismatch after j matches
    elif i < N and pat[j] != txt[i]:
        # Do not match lps[0..lps[j-1]] characters,
        # they will match anyway
        if j != 0:
            j = lps[j-1]
        else:
            i += 1

def computeLPSArray(pat, M, lps):
    len = 0 # length of the previous longest prefix suffix

    lps[0] # lps[0] is always 0
    i = 1

    # the loop calculates lps[i] for i = 1 to M-1
    while i < M:
        if pat[i]==pat[len]:
            len += 1
            lps[i] = len
            i += 1
        else:
            # This is tricky. Consider the example.
            # AAACAAAA and i = 7. The idea is similar
            # to search step.
            if len != 0:
                len = lps[len-1]

            # Also, note that we do not increment i here
            else:
                lps[i] = 0
                i += 1

txt = "ABABDABACDABABCABAB"
pat = "ABABCABAB"
KMPSearch(pat, txt)

# This code is contributed by Bhavya Jain

```

Output:

Found pattern at index 10

Preprocessing Algorithm:

In the preprocessing part, we calculate values in lps[]. To do that, we keep track of the length of the longest prefix suffix value (we use len variable for this purpose) for the previous index. We initialize lps[0] and len as 0. If pat[len] and pat[i] match, we increment len by 1 and assign the incremented value to lps[i]. If pat[i] and pat[len] do not match and len is not 0, we update len to lps[len-1]. See computeLPSArray () in the below code for details.

Illustration of preprocessing (or construction of lps[])

```

pat[] = "AAACAAAAA"

len = 0, i = 0.
lps[0] is always 0, we move
to i = 1

len = 0, i = 1.
Since pat[len] and pat[i] match, do len++,
store it in lps[i] and do i++.
len = 1, lps[1] = 1, i = 2

len = 1, i = 2.
Since pat[len] and pat[i] match, do len++,
store it in lps[i] and do i++.
len = 2, lps[2] = 2, i = 3

len = 2, i = 3.
Since pat[len] and pat[i] do not match, and len > 0,
set len = lps[len-1] = lps[1] = 1

len = 1, i = 3.
Since pat[len] and pat[i] do not match and len > 0,
len = lps[len-1] = lps[0] = 0

len = 0, i = 3.
Since pat[len] and pat[i] do not match and len = 0,
Set lps[3] = 0 and i = 4.

len = 0, i = 4.
Since pat[len] and pat[i] match, do len++,
store it in lps[i] and do i++.
len = 1, lps[4] = 1, i = 5

len = 1, i = 5.
Since pat[len] and pat[i] match, do len++,
store it in lps[i] and do i++.
len = 2, lps[5] = 2, i = 6

```

```

len = 2, i = 6.
Since pat[len] and pat[i] match, do len++,
store it in lps[i] and do i++.
len = 3, lps[6] = 3, i = 7

len = 3, i = 7.
Since pat[len] and pat[i] do not match and len > 0,
set len = lps[len-1] = lps[2] = 2

len = 2, i = 7.
Since pat[len] and pat[i] match, do len++,
store it in lps[i] and do i++.
len = 3, lps[7] = 3, i = 8

We stop here as we have constructed the whole lps[].

```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Pattern Searching
Pattern Searching

Searching for Patterns | Set 3 (Rabin-Karp Algorithm)

Given a text $txt[0..n-1]$ and a pattern $pat[0..m-1]$, write a function $search(char pat[], char txt[])$ that prints all occurrences of $pat[]$ in $txt[]$. You may assume that $n > m$.

Examples:

```

Input: txt[] = "THIS IS A TEST TEXT"
       pat[] = "TEST"
Output: Pattern found at index 10

```

```

Input: txt[] = "AABAACACAADAABAA"
       pat[] = "AABA"

```

Output: Pattern found at index 0

Pattern found at index 9

Pattern found at index 12

pattern-searching



The Naive String Matching algorithm slides the pattern one by one. After each slide, it one by one checks characters at the current shift and if all characters match then prints the match.

Like the Naive Algorithm, Rabin-Karp algorithm also slides the pattern one by one. But unlike the Naive algorithm, Rabin Karp algorithm matches the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters. So Rabin Karp algorithm needs to calculate hash values for following strings.

1) Pattern itself.

2) All the substrings of text of length m.

Since we need to efficiently calculate hash values for all the substrings of size m of text, we must have a hash function which has following property.

Hash at the next shift must be efficiently computable from the current hash value and next character in text or we can say $hash(txt[s+1 .. s+m])$ must be efficiently computable from $hash(txt[s .. s+m-1])$ and $txt[s+m]$ i.e., $hash(txt[s+1 .. s+m]) = rehash(txt[s+m], hash(txt[s .. s+m-1]))$ and rehash must be O(1) operation.

The hash function suggested by Rabin and Karp calculates an integer value. The integer value for a string is numeric value of a string. For example, if all possible characters are from 1 to 10, the numeric value of "122" will be 122. The number of possible characters is higher than 10 (256 in general) and pattern length can be large. So the numeric values cannot be practically stored as an integer. Therefore, the numeric value is calculated using modular arithmetic to make sure that the hash values can be stored in an integer variable (can fit in memory words). To do rehashing, we need to take off the most significant digit and add the new least significant digit for in hash value. Rehashing is done using the following formula.

$hash(txt[s+1 .. s+m]) = d \cdot (hash(txt[s .. s+m-1]) - txt[s]*h) + txt[s+m] \bmod q$

$hash(txt[s .. s+m-1])$: Hash value at shift s.

$hash(txt[s+1 .. s+m])$: Hash value at next shift (or shift s+1)

d: Number of characters in the alphabet

q: A prime number

h: $d^{m-1} \bmod q$

C/C++

```

/* Following program is a C implementation of Rabin Karp
Algorithm given in the CLRS book */
#include<stdio.h>
#include<string.h>

// d is the number of characters in input alphabet
#define d 256

/* pat -> pattern
   txt -> text
   q -> A prime number
*/
void search(char pat[], char txt[], int q)
{
    int M = strlen(pat);
    int N = strlen(txt);
    int i, j;
    int p = 0; // hash value for pattern
    int t = 0; // hash value for txt
    int h = 1;

    // The value of h would be "pow(d, M-1)%q"
    for (i = 0; i < M-1; i++)
        h = (h * d) % q;
    for (i = 0; i < N; i++)
    {
        t = (d * t + txt[i]) % q;
        p = (d * p + pat[i]) % q;
        if (i < M-1)
            continue;
        if (p == t)
            for (j = 1; j < M; j++)
                if (txt[i+j] != pat[j])
                    break;
            if (j == M)
                printf("Pattern found at index %d\n", i);
    }
}

```

```

h = (h*d)%q;

// Calculate the hash value of pattern and first
// window of text
for (i = 0; i < M; i++)
{
    p = (d*p + pat[i])%q;
    t = (d*t + txt[i])%q;
}

// Slide the pattern over text one by one
for (i = 0; i <= N - M; i++)
{

    // Check the hash values of current window of text
    // and pattern. If the hash values match then only
    // check for characters on by one
    if (p == t)
    {
        /* Check for characters one by one */
        for (j = 0; j < M; j++)
        {
            if (txt[i+j] != pat[j])
                break;
        }

        // If p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
        if (j == M)
            printf("Pattern found at index %d \n", i);
    }

    // Calculate hash value for next window of text: Remove
    // leading digit, add trailing digit
    if (i < N-M )
    {
        t = (d*(t - txt[i]*h) + txt[i+M])%q;

        // We might get negative value of t, converting it
        // to positive
        if (t < 0)
            t = (t + q);
    }
}

/* Driver program to test above function */
int main()
{
    char txt[] = "GEEKS FOR GEEKS";
    char pat[] = "GEEK";
    int q = 101; // A prime number
    search(pat, txt, q);
    return 0;
}

```

Python

```

# Following program is the python implementation of
# Rabin Karp Algorithm given in CLRS book

# d is the number of characters in input alphabet
d = 256

# pat -> pattern
# txt -> text
# q -> A prime number

def search(pat, txt, q):
    M = len(pat)
    N = len(txt)
    i = 0
    j = 0
    p = 0 # hash value for pattern
    t = 0 # hash value for txt
    h = 1

    # The value of h would be "pow(d, M-1)%q"
    for i in xrange(M-1):
        h = (h*d)%q

    # Calculate the hash value of pattern and first window
    # of text
    for i in xrange(M):
        p = (d*p + ord(pat[i]))%q
        t = (d*t + ord(txt[i]))%q

    # Slide the pattern over text one by one
    for i in xrange(N-M+1):
        # Check the hash values of current window of text and
        # pattern if the hash values match then only check
        # for characters on by one
        if p==t:
            # Check for characters one by one
            for j in xrange(M):
                if txt[i+j] != pat[j]:
                    break

            j+=1
            # If p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]
            if j==M:
                print "Pattern found at index " + str(i)

    # Calculate hash value for next window of text: Remove
    # leading digit, add trailing digit

```

```

if i < N-M:
    t = d*(t-ord(txt[i])*h) + ord(txt[i+M]))%q

    # We might get negative values of t, converting it to
    # positive
    if t < 0:
        t = t+q

# Driver program to test the above function
txt = "GEEKS FOR GEEKS"
pat = "GEEK"
q = 101 # A prime number
search(pat,txt,q)

# This code is contributed by Bhavya Jain

```

Output:

```

Pattern found at index 0
Pattern found at index 10

```

The average and best case running time of the Rabin-Karp algorithm is $O(n+m)$, but its worst-case time is $O(nm)$. Worst case of Rabin-Karp algorithm occurs when all characters of pattern and text are same as the hash values of all the substrings of txt[] match with hash value of pat[]. For example pat[] = "AAA" and txt[] = "AAAAAAA".

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:

```

http://net.pku.edu.cn/~course/cs101/2007/resource/Intro2Algorithm/book6/chap34.htm
http://www.cs.princeton.edu/courses/archive/fall04/cos226/lectures/string.4up.pdf
http://en.wikipedia.org/wiki/Rabin-Karp_string_search_algorithm

```

Related Posts:

[Searching for Patterns | Set 1 \(Naive Pattern Searching\)](#)
[Searching for Patterns | Set 2 \(KMP Algorithm\)](#)

GATE CS Corner Company Wise Coding Practice

Pattern Searching
modular-arithmetic
Pattern Searching

Searching for Patterns | Set 4 (A Naive Pattern Searching Question)

Question: We have discussed Naive String matching algorithm [here](#). Consider a situation where all characters of pattern are different. Can we modify the original Naive String Matching algorithm so that it works better for these types of patterns. If we can, then what are the changes to original algorithm?

Solution: In the [original Naive String matching algorithm](#), we always slide the pattern by 1. When all characters of pattern are different, we can slide the pattern by more than 1. Let us see how we can do this. When a mismatch occurs after j matches, we know that the first character of pattern will not match the j matched characters because all characters of pattern are different. So we can always slide the pattern by j without missing any valid shifts. Following is the modified code that is optimized for the special patterns.

C

```

/* C program for A modified Naive Pattern Searching
algorithm that is optimized for the cases when all
characters of pattern are different */
#include<stdio.h>
#include<string.h>

/* A modified Naive Pattern Searching algorithm that is optimized
for the cases when all characters of pattern are different */
void search(char pat[], char txt[])
{
    int M = strlen(pat);
    int N = strlen(txt);
    int i = 0;

    while (i <= N - M)
    {
        int j;

        /* For current index i, check for pattern match */
        for (j = 0; j < M; j++)
            if (txt[i+j] != pat[j])
                break;

        if (j == M) // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
        {
            printf("Pattern found at index %d \n", i);
            i = i + M;
        }
        else if (j == 0)
            i = i + 1;
        else
            i = i + j; // slide the pattern by j
    }
}

/* Driver program to test above function */
int main()
{
    char txt[] = "ABCEABCDEABC";
    char pat[] = "ABCDE";
    search(pat,txt);
    return 0;
}

```

Python

```

# Python program for A modified Naive Pattern Searching
# algorithm that is optimized for the cases when all
# characters of pattern are different
def search(pat, txt):
    M = len(pat)
    N = len(txt)
    i = 0

    while i <= N-M:
        # For current index i, check for pattern match
        for j in xrange(M):
            if txt[i+j] != pat[j]:
                break
            j += 1

        if j==M: # if pat[0..M-1] = txt[i,i+1...i+M-1]
            print "Pattern found at index " + str(i)
            i = i + M
        elif j==0:
            i = i + 1
        else:
            i = i + j # slide the pattern by j

# Driver program to test the above function
txt = "ABCEABCDABCABCABCD"
pat = "ABCD"
search(pat, txt)

# This code is contributed by Bhavya Jain

```

Output:

```

Pattern found at index 4
Pattern found at index 12

```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

[Pattern Searching](#)
[Pattern Searching](#)

Searching for Patterns | Set 5 (Finite Automata)

Given a text $txt[0..n-1]$ and a pattern $pat[0..m-1]$, write a function $search(char pat[], char txt[])$ that prints all occurrences of $pat[]$ in $txt[]$. You may assume that $n > m$.

Examples:

```

Input: txt[] = "THIS IS A TEST TEXT"
       pat[] = "TEST"
Output: Pattern found at index 10

Input: txt[] = "AABAACAAADAAAABAABA"
       pat[] = "AABA"
Output: Pattern found at index 0
       Pattern found at index 9
       Pattern found at index 12
pattern-searching

```



Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

We have discussed the following algorithms in the previous posts:

[Naive Algorithm](#)
[KMP Algorithm](#)
[Rabin Karp Algorithm](#)

In this post, we will discuss Finite Automata (FA) based pattern searching algorithm. In FA based algorithm, we preprocess the pattern and build a 2D array that represents a Finite Automata. Construction of the FA is the main tricky part of this algorithm. Once the FA is built, the searching is simple. In search, we simply need to start from the first state of the automata and the first character of the text. At every step, we consider next character of text, look for the next state in the built FA and move to a new state. If we reach the final state, then the pattern is found in the text. The time complexity of the search process is $O(n)$.

Before we discuss FA construction, let us take a look at the following FA for pattern ACACAGA.





The above diagrams represent graphical and tabular representations of pattern ACACAGA.

Number of states in FA will be $M+1$ where M is length of the pattern. The main thing to construct FA is to get the next state from the current state for every possible character. Given a character x and a state k , we can get the next state by considering the string "pat[0..k-1]x" which is basically concatenation of pattern characters $pat[0], pat[1] \dots pat[k-1]$ and the character x . The idea is to get length of the longest prefix of the given pattern such that the prefix is also suffix of "pat[0..k-1]x". The value of length gives us the next state. For example, let us see how to get the next state from current state 5 and character 'C' in the above diagram. We need to consider the string, "pat[0..5]C" which is "ACACAC". The length of the longest prefix of the pattern such that the prefix is suffix of "ACACAC" is 4 ("ACAC"). So the next state (from state 5) is 4 for character 'C'.

In the following code, `computeTF()` constructs the FA. The time complexity of the `computeTF()` is $O(m^3 \cdot NO_OF_CHARS)$ where m is length of the pattern and NO_OF_CHARS is size of alphabet (total number of possible characters in pattern and text). The implementation tries all possible prefixes starting from the longest possible that can be a suffix of "pat[0..k-1]x". There are better implementations to construct FA in $O(m \cdot NO_OF_CHARS)$ (Hint: we can use something like [lps array construction in KMP algorithm](#)). We have covered the better implementation in our [next post](#) on pattern searching.

```
#include<stdio.h>
#include<string.h>
#define NO_OF_CHARS 256

int getNextState(char *pat, int M, int state, int x)
{
    // If the character c is same as next character in pattern,
    // then simply increment state
    if (state < M && x == pat[state])
        return state+1;

    int ns, i; // ns stores the result which is next state

    // ns finally contains the longest prefix which is also suffix
    // in "pat[0..state-1]c"
    for (ns = state; ns > 0; ns--)
    {
        if (pat[ns-1] == x)
        {
            for(i = 0; i < ns-1; i++)
            {
                if (pat[i] != pat[state-ns+1+i])
                    break;
            }
            if (i == ns-1)
                return ns;
        }
    }

    return 0;
}

/* This function builds the TF table which represents Finite Automata for a
given pattern */
void computeTF(char *pat, int M, int TF[][NO_OF_CHARS])
{
    int state, x;
    for (state = 0; state <= M; ++state)
        for (x = 0; x < NO_OF_CHARS; ++x)
            TF[state][x] = getNextState(pat, M, state, x);
}

/* Prints all occurrences of pat in txt */
void search(char *pat, char *txt)
{
    int M = strlen(pat);
    int N = strlen(txt);

    int TF[M+1][NO_OF_CHARS];

    computeTF(pat, M, TF);

    // Process txt over FA.
    int i, state=0;
    for (i = 0; i < N; i++)
    {
        state = TF[state][txt[i]];
        if (state == M)
        {
            printf ("\n Pattern found at index %d", i-M+1);
        }
    }
}

// Driver program to test above function
int main()
{
    char *txt = "AABAACAADAABAAABAA";
    char *pat = "AABA";
    search(pat, txt);
    return 0;
}
```

Output:

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

References:

Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Pattern Searching
Pattern Searching

Pattern Searching | Set 6 (Efficient Construction of Finite Automata)

In the [previous post](#), we discussed Finite Automata based pattern searching algorithm. The FA (Finite Automata) construction method discussed in previous post takes $O(m^3 * NO_OF_CHARS)$ time. FA can be constructed in $O(m * NO_OF_CHARS)$ time. In this post, we will discuss the $O(m * NO_OF_CHARS)$ algorithm for FA construction. The idea is similar to lps (longest prefix suffix) array construction discussed in the [KMP algorithm](#). We use previously filled rows to fill a new row.



The above diagrams represent graphical and tabular representations of pattern ACACAGA.

Algorithm:

- 1) Fill the first row. All entries in first row are always 0 except the entry for pat[0] character. For pat[0] character, we always need to go to state 1.
- 2) Initialize lps as 0. lps for the first index is always 0.
- 3) Do following for rows at index i = 1 to M. (M is the length of the pattern)
 -a) Copy the entries from the row at index equal to lps.
 -b) Update the entry for pat[i] character to i+1.
 -c) Update lps "lps = TF[lps][pat[i]]" where TF is the 2D array which is being constructed.

Implementation

Following is C implementation for the above algorithm.

```
#include<stdio.h>
#include<string.h>
#define NO_OF_CHARS 256

/* This function builds the TF table which represents Finite Automata for a
   given pattern */
void computeTransFun(char *pat, int M, int TF[][NO_OF_CHARS])
{
    int i, lps = 0, x;

    // Fill entries in first row
    for (x = 0; x < NO_OF_CHARS; x++)
        TF[0][x] = 0;
    TF[0][pat[0]] = 1;

    // Fill entries in other rows
    for (i = 1; i <= M; i++)
    {
        // Copy values from row at index lps
        for (x = 0; x < NO_OF_CHARS; x++)
            TF[i][x] = TF[lps][x];

        // Update the entry corresponding to this character
        TF[i][pat[i]] = i + 1;

        // Update lps for next row to be filled
        if (i < M)
            lps = TF[lps][pat[i]];
    }

    /* Prints all occurrences of pat in txt */
    void search(char *pat, char *txt)
    {
        int M = strlen(pat);
        int N = strlen(txt);

        int TF[M+1][NO_OF_CHARS];

        computeTransFun(pat, M, TF);

        // process text over FA.
        int i, j=0;
        for (i = 0; i < N; i++)
        {
            j = TF[j][txt[i]];
            if (j == M)
            {
                printf ("\n pattern found at index %d", i-M+1);
            }
        }
    }
}
```

```

/* Driver program to test above function */
int main()
{
    char *txt = "GEEKS FOR GEEKS";
    char *pat = "GEEKS";
    search(pat, txt);
    getchar();
    return 0;
}

```

Output:

```

pattern found at index 0
pattern found at index 10

```

Time Complexity for FA construction is $O(M \cdot NO_OF_CHARS)$. The code for search is same as the [previous post](#) and time complexity for it is $O(n)$.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

[Pattern Searching](#)
[Pattern Searching](#)

Pattern Searching | Set 7 (Boyer Moore Algorithm – Bad Character Heuristic)

Given a text $txt[0..n-1]$ and a pattern $pat[0..m-1]$, write a function $search(char pat[], char txt[])$ that prints all occurrences of $pat[]$ in $txt[]$. You may assume that $n > m$.

Examples:

```

Input: txt[] = "THIS IS A TEST TEXT"
       pat[] = "TEST"
Output: Pattern found at index 10

```

```

Input: txt[] = "AABAACCAADAABAAABA"
       pat[] = "AABA"
Output: Pattern found at index 0
       Pattern found at index 9
       Pattern found at index 12

```

pattern-searching



Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

We have discussed the following algorithms in the previous posts:

[Naive Algorithm](#)
[KMP Algorithm](#)
[Rabin Karp Algorithm](#)
[Finite Automata based Algorithm](#)

In this post, we will discuss Boyer Moore pattern searching algorithm. Like [KMP](#) and [Finite Automata](#) algorithms, Boyer Moore algorithm also preprocesses the pattern.

[Boyer Moore is a combination of following two approaches.](#)

- 1) [Bad Character Heuristic](#)
- 2) [Good Suffix Heuristic](#)

Both of the above heuristics can also be used independently to search a pattern in a text. Let us first understand how two independent approaches work together in the Boyer Moore algorithm. If we take a look at the [Naive algorithm](#), it slides the pattern over the text one by one. KMP algorithm does preprocessing over the pattern so that the pattern can be shifted by more than one. The Boyer Moore algorithm does preprocessing for the same reason. It preprocesses the pattern and creates different arrays for both heuristics. At every step, it slides the pattern by max of the slides suggested by the two heuristics. [So it uses best of the two heuristics at every step. Unlike the previous pattern searching algorithms, Boyer Moore algorithm starts matching from the last character of the pattern.](#)

In this post, we will discuss bad character heuristic, and discuss Good Suffix heuristic in the next post.

The idea of bad character heuristic is simple. The character of the text which doesn't match with the current character of pattern is called the Bad Character. Whenever a character doesn't match, we slide the pattern in such a way that aligns the bad character with the last occurrence of it in pattern. We preprocess the pattern and store the last occurrence of every possible character in an array of size equal to alphabet size. If the character is not present at all, then it may result in a shift by m (length of pattern). Therefore, [the bad character heuristic takes \$O\(n/m\)\$ time in the best case.](#)

```

/* Program for Bad Character Heuristic of Boyer Moore String Matching Algorithm */

#include <limits.h>
#include <string.h>
#include <stdio.h>

#define NO_OF_CHARS 256

// A utility function to get maximum of two integers
int max (int a, int b) { return (a > b)? a: b; }

// The preprocessing function for Boyer Moore's bad character heuristic
void badCharHeuristic( char *str, int size, int badchar[NO_OF_CHARS])
{
    int i;

    // Initialize all occurrences as -1
    for (i = 0; i < NO_OF_CHARS; i++)
        badchar[i] = -1;

    // Fill the actual value of last occurrence of a character
    for (i = 0; i < size; i++)
        badchar[(int) str[i]] = i;
}

/* A pattern searching function that uses Bad Character Heuristic of
   Boyer Moore */

```

```

Boyer Moore Algorithm */
void search( char *txt, char *pat)
{
    int m = strlen(pat);
    int n = strlen(txt);

    int badchar[NO_OF_CHARS];

    /* Fill the bad character array by calling the preprocessing
       function badCharHeuristic() for given pattern */
    badCharHeuristic(pat, m, badchar);

    int s = 0; // s is shift of the pattern with respect to text
    while(s <= (n - m))
    {
        int j = m-1;

        /* Keep reducing index j of pattern while characters of
           pattern and text are matching at this shift s */
        while(j >= 0 && pat[j] == txt[s+j])
            j--;

        /* If the pattern is present at current shift, then index j
           will become -1 after the above loop */
        if (j < 0)
        {
            printf("\n pattern occurs at shift = %d", s);

            /* Shift the pattern so that the next character in text
               aligns with the last occurrence of it in pattern.
               The condition s+m < n is necessary for the case when
               pattern occurs at the end of text */
            s += (s+m < n)? m-badchar[txt[s+m]] : 1;
        }
        else
        {
            /* Shift the pattern so that the bad character in text
               aligns with the last occurrence of it in pattern. The
               max function is used to make sure that we get a positive
               shift. We may get a negative shift if the last occurrence
               of bad character in pattern is on the right side of the
               current character. */
            s += max(1, j - badchar[txt[s+j]]);
        }
    }

    /* Driver program to test above function */
int main()
{
    char txt[] = "ABAAABCD";
    char pat[] = "ABC";
    search(txt, pat);
    return 0;
}

```

Output:

```
pattern occurs at shift = 4
```

The Bad Character Heuristic may take O(mn) time in worst case. The worst case occurs when all characters of the text and pattern are same. For example, $txt[] = "AAAAAAAAAAAAA"$ and $pat[] = "AAAAA"$.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Pattern Searching
Pattern Searching

Suffix Array | Set 1 (Introduction)

We strongly recommend to read following post on suffix trees as a pre-requisite for this post.

[Pattern Searching | Set 8 \(Suffix Tree Introduction\)](#)

A **suffix array** is a sorted array of all suffixes of a given string. The definition is similar to **Suffix Tree** which is compressed trie of all suffixes of the given text. Any suffix tree based algorithm can be replaced with an algorithm that uses a suffix array enhanced with additional information and solves the same problem in the same time complexity (Source [Wiki](#)).

A suffix array can be constructed from Suffix tree by doing a DFS traversal of the suffix tree. In fact Suffix array and suffix tree both can be constructed from each other in linear time.

Advantages of suffix arrays over suffix trees include improved space requirements, simpler linear time construction algorithms (e.g., compared to Ukkonen's algorithm) and improved cache locality (Source: [Wiki](#))

Example:

Let the given string be "banana".

```

0 banana          5 a
1 anana   Sort the Suffixes  3 ana
2 nana   ----->  1 anana
3 ana   alphabetically  0 banana
4 na      4 na
5 a      2 nana

```

So the suffix array for "banana" is {5, 3, 1, 0, 4, 2}

Naive method to build Suffix Array

A simple method to construct suffix array is to make an array of all suffixes and then sort the array. Following is implementation of simple method.

```

// Naive algorithm for building suffix array of a given text
#include <iostream>
#include <cstring>
#include <algorithm>

```

```

using namespace std;

// Structure to store information of a suffix
struct suffix
{
    int index;
    char *suff;
};

// A comparison function used by sort() to compare two suffixes
int cmp(struct suffix a, struct suffix b)
{
    return strcmp(a.suff, b.suff) < 0 ? 1 : 0;
}

// This is the main function that takes a string 'txt' of size n as an
// argument, builds and return the suffix array for the given string
int *buildSuffixArray(char *txt, int n)
{
    // A structure to store suffixes and their indexes
    struct suffix suffixes[n];

    // Store suffixes and their indexes in an array of structures.
    // The structure is needed to sort the suffixes alphabetically
    // and maintain their old indexes while sorting
    for (int i = 0; i < n; i++)
    {
        suffixes[i].index = i;
        suffixes[i].suff = (txt+i);
    }

    // Sort the suffixes using the comparison function
    // defined above.
    sort(suffixes, suffixes+n, cmp);

    // Store indexes of all sorted suffixes in the suffix array
    int *suffixArr = new int[n];
    for (int i = 0; i < n; i++)
        suffixArr[i] = suffixes[i].index;

    // Return the suffix array
    return suffixArr;
}

// A utility function to print an array of given size
void printArr(int arr[], int n)
{
    for(int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    char txt[] = "banana";
    int n = strlen(txt);
    int *suffixArr = buildSuffixArray(txt, n);
    cout << "Following is suffix array for " << txt << endl;
    printArr(suffixArr, n);
    return 0;
}

```

Output:

```

Following is suffix array for banana
5 3 1 0 4 2

```

The time complexity of above method to build suffix array is $O(n^2 \log n)$ if we consider a $O(n \log n)$ algorithm used for sorting. The sorting step itself takes $O(n^2 \log n)$ time as every comparison is a comparison of two strings and the comparison takes $O(n)$ time.

There are many efficient algorithms to build suffix array. We will soon be covering them as separate posts.

Search a pattern using the built Suffix Array

To search a pattern in a text, we preprocess the text and build a suffix array of the text. Since we have a sorted array of all suffixes, [Binary Search](#) can be used to search. Following is the search function. Note that the function doesn't report all occurrences of pattern, it only report one of them.

```

// This code only contains search() and main. To make it a complete running
// above code or see http://code.geeksforgeeks.org/oY7OkD

// A suffix array based search function to search a given pattern
// 'pat' in given text 'txt' using suffix array suffArr[]
void search(char *pat, char *txt, int *suffArr, int n)
{
    int m = strlen(pat); // get length of pattern, needed for strncmp()

    // Do simple binary search for the pat in txt using the
    // built suffix array
    int l = 0, r = n-1; // Initialize left and right indexes
    while (l <= r)
    {
        // See if 'pat' is prefix of middle suffix in suffix array
        int mid = l + (r - l)/2;
        int res = strncmp(pat, txt+suffArr[mid], m);

        // If match found at the middle, print it and return
        if (res == 0)
        {
            cout << "Pattern found at index " << suffArr[mid];
            return;
        }

        // Move to left half if pattern is alphabetically less than
        // the mid suffix
        if (res < 0) r = mid - 1;
    }
}

```

```

    // Otherwise move to right half
    else l = mid + 1;
}

// We reach here if return statement in loop is not executed
cout << "Pattern not found";
}

// Driver program to test above function
int main()
{
    char txt[] = "banana"; // text
    char pat[] = "nan"; // pattern to be searched in text

    // Build suffix array
    int n = strlen(txt);
    int *suffArr = buildSuffixArray(txt, n);

    // search pat in txt using the built suffix array
    search(pat, txt, suffArr, n);

    return 0;
}

```

Output:

Pattern found at index 2

The time complexity of the above search function is $O(m\log n)$. There are more efficient algorithms to search pattern once the suffix array is built. In fact there is a $O(m)$ suffix array based algorithm to search a pattern. We will soon be discussing efficient algorithm for search.

Applications of Suffix Array

Suffix array is an extremely useful data structure, it can be used for a wide range of problems. Following are some famous problems where Suffix array can be used.

- 1) Pattern Searching
- 2) Finding the longest repeated substring
- 3) Finding the longest common substring
- 4) Finding the longest palindrome in a string

See this for more problems where Suffix arrays can be used.

This post is a simple introduction. There is a lot to cover in Suffix arrays. We have discussed a $O(n\log n)$ algorithm for Suffix Array construction here. We will soon be discussing more efficient suffix array algorithms.

References:

<http://www.stanford.edu/class/cs97si/suffix-array.pdf>
http://en.wikipedia.org/wiki/Suffix_array

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
 Pattern Searching
 Advance Data Structures
 Advanced Data Structures
 Pattern Searching
 Suffix-Array

Anagram Substring Search (Or Search for all permutations)

Given a text $txt[0..n-1]$ and a pattern $pat[0..m-1]$, write a function $search(char pat[], char txt[])$ that prints all occurrences of $pat[]$ and its permutations (or anagrams) in $txt[]$. You may assume that $n > m$.
 Expected time complexity is $O(n)$

Examples:

```

1) Input: txt[] = "BACDGABCDA" pat[] = "ABCD"
   Output: Found at Index 0
           Found at Index 5
           Found at Index 6
2) Input: txt[] = "AAABABAA" pat[] = "AABA"
   Output: Found at Index 0
           Found at Index 1
           Found at Index 4

```

We strongly recommend that you click here and practice it, before moving on to the solution.

This problem is slightly different from standard pattern searching problem, here we need to search for anagrams as well. Therefore, we cannot directly apply standard pattern searching algorithms like KMP, Rabin Karp, Boyer Moore, etc.

A simple idea is to modify Rabin Karp Algorithm. For example we can keep the hash value as sum of ASCII values of all characters under modulo of a big prime number. For every character of text, we can add the current character to hash value and subtract the first character of previous window. This solution looks good, but like standard Rabin Karp, the worst case time complexity of this solution is $O(mn)$. The worst case occurs when all hash values match and we one by one match all characters.

We can achieve $O(n)$ time complexity under the assumption that alphabet size is fixed which is typically true as we have maximum 256 possible characters in ASCII. The idea is to use two count arrays:

- 1) The first count array stores frequencies of characters in pattern.
- 2) The second count array stores frequencies of characters in current window of text.

The important thing to note is, time complexity to compare two count arrays is $O(1)$ as the number of elements in them are fixed (independent of pattern and text sizes). Following are steps of this algorithm.

1) Store counts of frequencies of pattern in first count array $countP[]$. Also store counts of frequencies of characters in first window in array $countTW[]$.

- 2) Now run a loop from $i = M$ to $N - 1$. Do following in loop.
 -a) If the two count arrays are identical, we found an occurrence.
 -b) Increment count of current character of text in $countTW[]$
 -c) Decrement count of first character in previous window in $countWT[]$
- 3) The last window is not checked by above loop, so explicitly check it.

Following is C++ implementation of above algorithm.

```

// C++ program to search all anagrams of a pattern in a text
#include<iostream>
#include<cstring>
#define MAX 256
using namespace std;

// This function returns true if contents of arr1[] and arr2[]
// are same, otherwise false.
bool compare(char arr1[], char arr2[])
{
    for (int i=0; i<MAX; i++)
        if (arr1[i] != arr2[i])
            return false;
    return true;
}

// This function search for all permutations of pat[] in txt[]
void search(char *pat, char *txt)
{
    int M = strlen(pat), N = strlen(txt);

    // countP[]: Store count of all characters of pattern
    // countTW[]: Store count of current window of text
    char countP[MAX] = {0}, countTW[MAX] = {0};
    for (int i = 0; i < M; i++)
    {
        (countP[pat[i]])++;
        (countTW[txt[i]])++;
    }

    // Traverse through remaining characters of pattern
    for (int i = M; i < N; i++)
    {
        // Compare counts of current window of text with
        // counts of pattern[]
        if (compare(countP, countTW))
            cout << "Found at Index " << (i - M) << endl;

        // Add current character to current window
        (countTW[txt[i]])++;

        // Remove the first character of previous window
        countTW[txt[i-M]]--;
    }

    // Check for the last window in text
    if (compare(countP, countTW))
        cout << "Found at Index " << (N - M) << endl;
}

/* Driver program to test above function */
int main()
{
    char txt[] = "BACDGABCDA";
    char pat[] = "ABCD";
    search(pat, txt);
    return 0;
}

```

Output:

```

Found at Index 0
Found at Index 5
Found at Index 6

```

This article is contributed by **Piyush Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Pattern Searching
anagram
Pattern Searching
permutation

Pattern Searching using a Trie of all Suffixes

Problem Statement: Given a text $txt[0..n-1]$ and a pattern $pat[0..m-1]$, write a function $search(char pat[], char txt[])$ that prints all occurrences of $pat[]$ in $txt[]$. You may assume that $n > m$.

As discussed in the [previous post](#), we discussed that there are two ways efficiently solve the above problem.

1) Preprocess Pattern: [KMP Algorithm](#), [Rabin Karp Algorithm](#), [Finite Automata](#), [Boyer Moore Algorithm](#).

2) Preprocess Text: [Suffix Tree](#)

The best possible time complexity achieved by first (preprocessing pattern) is $O(n)$ and by second (preprocessing text) is $O(m)$ where m and n are lengths of pattern and text respectively.

Note that the second way does the searching only in $O(m)$ time and it is preferred when text doesn't change very frequently and there are many search queries. We have discussed [Suffix Tree](#) ([A compressed Trie of all suffixes of Text](#)).

Implementation of Suffix Tree may be time consuming for problems to be coded in a technical interview or programming contexts. In this post simple implementation of a [Standard Trie](#) of all Suffixes is discussed. The implementation is close to suffix tree, the only thing is, it's a [simple Trie](#) instead of compressed Trie.

As discussed in [Suffix Tree](#) post, the idea is, every pattern that is present in text (or we can say every substring of text) must be a prefix of one of all possible suffixes. So if we build a Trie of all suffixes, we can find the pattern in $O(m)$ time where m is pattern length.

Building a Trie of Suffixes

- 1) Generate all suffixes of given text.
- 2) Consider all suffixes as individual words and build a trie.

Let us consider an example text "banana\0" where '\0' is string termination character. Following are all suffixes of "banana\0"

```

banana\0
anana\0
nana\0
ana\0

```

```
na0
a0
\0
```

If we consider all of the above suffixes as individual words and build a Trie, we get following.



How to search a pattern in the built Trie?

Following are steps to search a pattern in the built Trie.

1) Starting from the first character of the pattern and root of the Trie, do following for every character.

.....a) For the current character of pattern, if there is an edge from the current node, follow the edge.

.....b) If there is no edge, print "pattern doesn't exist in text" and return.

2) If all characters of pattern have been processed, i.e., there is a path from root for characters of the given pattern, then print all indexes where pattern is present. To store indexes, we use a list with every node that stores indexes of suffixes starting at the node.

Following is C++ implementation of the above idea.

```
// A simple C++ implementation of substring search using trie of suffixes
#include<iostream>
#include<list>
#define MAX_CHAR 256
using namespace std;
```

```
// A Suffix Trie (A Trie of all suffixes) Node
class SuffixTreeNode
{
private:
    SuffixTreeNode *children[MAX_CHAR];
    list<int> *indexes;
public:
    SuffixTreeNode() // Constructor
    {
        // Create an empty linked list for indexes of
        // suffixes starting from this node
        indexes = new list<int>;
    }

    // Initialize all child pointers as NULL
    for (int i = 0; i < MAX_CHAR; i++)
        children[i] = NULL;
```

```
// A recursive function to insert a suffix of the txt
// in subtree rooted with this node
void insertSuffix(string suffix, int index);
```

```
// A function to search a pattern in subtree rooted
// with this node. The function returns pointer to a linked
// list containing all indexes where pattern is present.
// The returned indexes are indexes of last characters
// of matched text.
list<int> *search(string pat);
};
```

```
// A Trie of all suffixes
class SuffixTrie
```

```
{
private:
    SuffixTreeNode root;
public:
    // Constructor (Builds a trie of suffixes of the given text)
    SuffixTrie(string txt)
    {
        // Consider all suffixes of given string and insert
        // them into the Suffix Trie using recursive function
        // insertSuffix() in SuffixTreeNode class
        for (int i = 0; i < txt.length(); i++)
            root.insertSuffix(txt.substr(i), i);
    }
}
```

```
// Function to searches a pattern in this suffix trie.
void search(string pat);
};
```

```
// A recursive function to insert a suffix of the txt in
// subtree rooted with this node
void SuffixTreeNode::insertSuffix(string s, int index)
{
    // Store index in linked list
    indexes->push_front(index);

    // If string has more characters
    if (s.length() > 0)
    {
        // Find the first character
        char cIndex = s.at(0);

        // If there is no edge for this character, add a new edge
        if (children[cIndex] == NULL)
            children[cIndex] = new SuffixTreeNode();

        // Recur for next suffix
        children[cIndex]->insertSuffix(s.substr(1), index);
    }
}
```

```

        children[cIndex]->insertSuffix(s.substr(1), index+1);
    }

    // A recursive function to search a pattern in subtree rooted with
    // this node
    list<int> SuffixTrieNode::search(string s)
    {
        // If all characters of pattern have been processed,
        if (s.length() == 0)
            return indexes;

        // If there is an edge from the current node of suffix trie,
        // follow the edge.
        if (children[s.at(0)] != NULL)
            return (children[s.at(0)])->search(s.substr(1));

        // If there is no edge, pattern doesn't exist in text
        else return NULL;
    }

    /* Prints all occurrences of pat in the Suffix Trie S (built for text)*/
    void SuffixTrie::search(string pat)
    {
        // Let us call recursive search function for root of Trie.
        // We get a list of all indexes (where pat is present in text) in
        // variable 'result'
        list<int> *result = root.search(pat);

        // Check if the list of indexes is empty or not
        if (result == NULL)
            cout << "Pattern not found" << endl;
        else
        {
            list<int>::iterator i;
            int patLen = pat.length();
            for (i = result->begin(); i != result->end(); ++i)
                cout << "Pattern found at position " << *i - patLen << endl;
        }
    }

    // driver program to test above functions
    int main()
    {
        // Let us build a suffix trie for text "geeksforgeeks.org"
        string txt = "geeksforgeeks.org";
        SuffixTrie S(txt);

        cout << "Search for 'ee'" << endl;
        S.search("ee");

        cout << "\nSearch for 'geek'" << endl;
        S.search("geek");

        cout << "\nSearch for 'quiz'" << endl;
        S.search("quiz");

        cout << "\nSearch for 'forgeeks'" << endl;
        S.search("forgeeks");

        return 0;
    }
}

```

Output:

```

Search for 'ee'
Pattern found at position 9
Pattern found at position 1

Search for 'geek'
Pattern found at position 8
Pattern found at position 0

Search for 'quiz'
Pattern not found

Search for 'forgeeks'
Pattern found at position 5

```

Time Complexity of the above search function is $O(m+k)$ where m is length of the pattern and k is the number of occurrences of pattern in text.

This article is contributed by Ashish Anand. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Pattern Searching
Pattern Searching
TRIE

Aho-Corasick Algorithm for Pattern Searching

Given an input text and an array of k words, $arr[]$, find all occurrences of all words in the input text. Let n be the length of text and m be the total number characters in all words, i.e. $m = \text{length}(arr[0]) + \text{length}(arr[1]) + \dots + O(n + \text{length}(arr[k-1]))$. Here k is total numbers of input words.

Example:

```

Input: text = "ahishers"
arr[] = {"he", "she", "hers", "his"}

Output:
Word his appears from 1 to 3
Word he appears from 4 to 5
Word she appears from 3 to 5

```

Word **hers** appears from 4 to 7

If we use a linear time searching algorithm like **KMP**, then we need to one by one search all words in `text[]`. This gives us total time complexity as $O(n + \text{length}(\text{word}[0]) + O(n + \text{length}(\text{word}[1]) + O(n + \text{length}(\text{word}[2]) + \dots + O(n + \text{length}(\text{word}[k-1]))$. This time complexity can be written as $O(n \cdot k + m)$.

Aho-Corasick Algorithm finds all words in $O(n + m + z)$ time where z is total number of occurrences of words in text. The Aho-Corasick string matching algorithm formed the basis of the original Unix command `grep`.

1. **Preprocessing** : Build an automaton of all words in `arr[]` The automaton has mainly three functions:

Go To : This function simply follows edges of Trie of all words in `arr[]`. It is represented as 2D array `g[][]` where we store next state for current state and character.

Failure : This function stores all edges that are followed when current character doesn't have edge in Trie. It is represented as 1D array `f[]` where we store next state for current state.

Output : Stores indexes of all words that end at current state. It is represented as 1D array `o[]` where we store indexes of all matching words as a bitmap for current state.

2. **Matching** : Traverse the given text over built automaton to find all matching words.

Preprocessing:

1. We first Build a **Trie** (or Keyword Tree) of all words.

`aho-corasick`



This part fills entries in goto `g[][]` and output `o[]`.

2. Next we extend Trie into an automaton to support linear time matching.

`aho-corasick2`



This part fills entries in failure `f[]` and output `o[]`.

Go to :

We build **Trie**. And for all characters which don't have an edge at root, we add an edge back to root.

Failure :

For a state s , we find the longest proper suffix which is a proper prefix of some pattern. This is done using Breadth First Traversal of Trie.

Output :

For a state s , indexes of all words ending at s are stored. These indexes are stored as bitwise map (by doing bitwise OR of values). This is also computing using Breadth First Traversal with Failure.

Below is C++ implementation of Aho-Corasick Algorithm

```
// C++ program for implementation of Aho Corasick algorithm
// for string matching
using namespace std;
#include <bits/stdc++.h>

// Max number of states in the matching machine.
// Should be equal to the sum of the length of all keywords.
const int MAXS = 500;

// Maximum number of characters in input alphabet
const int MAXC = 26;

// OUTPUT FUNCTION IS IMPLEMENTED USING out[]
// Bit i in this mask is one if the word with index i
// appears when the machine enters this state.
int out[MAXS];

// FAILURE FUNCTION IS IMPLEMENTED USING f[]

```

```

int f[MAXS];

// GOTO FUNCTION (OR TRIE) IS IMPLEMENTED USING g[]
int g[MAXS][MAXC];

// Builds the string matching machine.
// arr - array of words. The index of each keyword is important:
// "out[state] & (1 << i)" is > 0 if we just found word[i]
// in the text.
// Returns the number of states that the built machine has.
// States are numbered 0 up to the return value - 1, inclusive.
int buildMatchingMachine(string arr[], int k)
{
    // Initialize all values in output function as 0.
    memset(out, 0, sizeof out);

    // Initialize all values in goto function as -1.
    memset(g, -1, sizeof g);

    // Initially, we just have the 0 state
    int states = 1;

    // Construct values for goto function, i.e., fill g[]
    // This is same as building a Trie for arr[]
    for (int i = 0; i < k; ++i)
    {
        const string &word = arr[i];
        int currentState = 0;

        // Insert all characters of current word in arr[]
        for (int j = 0; j < word.size(); ++j)
        {
            int ch = word[j] - 'a';

            // Allocate a new node (create a new state) if a
            // node for ch doesn't exist.
            if (g[currentState][ch] == -1)
                g[currentState][ch] = states++;

            currentState = g[currentState][ch];
        }

        // Add current word in output function
        out[currentState] |= (1 << i);
    }

    // For all characters which don't have an edge from
    // root (or state 0) in Trie, add a goto edge to state
    // 0 itself
    for (int ch = 0; ch < MAXC; ++ch)
        if (g[0][ch] == -1)
            g[0][ch] = 0;
    }

    // Now, let's build the failure function

    // Initialize values in fail function
    memset(f, -1, sizeof f);

    // Failure function is computed in breadth first order
    // using a queue
    queue<int> q;

    // Iterate over every possible input
    for (int ch = 0; ch < MAXC; ++ch)
    {
        // All nodes of depth 1 have failure function value
        // as 0. For example, in above diagram we move to 0
        // from states 1 and 3.
        if (g[0][ch] != 0)
        {
            if (g[0][ch]] == 0;
            q.push(g[0][ch]);
        }
    }

    // Now queue has states 1 and 3
    while (q.size())
    {
        // Remove the front state from queue
        int state = q.front();
        q.pop();

        // For the removed state, find failure function for
        // all those characters for which goto function is
        // not defined.
        for (int ch = 0; ch <= MAXC; ++ch)
        {
            // If goto function is defined for character 'ch'
            // and 'state'
            if (g[state][ch] != -1)
            {
                // Find failure state of removed state
                int failure = f[state];

                // Find the deepest node labeled by proper
                // suffix of string from root to current
                // state.
                while (g[failure][ch] == -1)
                    failure = f[failure];

                failure = g[failure][ch];
                f[g[state][ch]] = failure;
            }

            // Merge output values
            out[g[state][ch]] |= out[failure];
        }

        // Insert the next level node (of Trie) in Queue
    }
}

```

```

        q.push(g[state][ch]);
    }
}

return states;
}

// Returns the next state the machine will transition to using goto
// and failure functions.
// currentState - The current state of the machine. Must be between
// 0 and the number of states - 1, inclusive.
// nextInput - The next character that enters into the machine.
int findNextState(int currentState, char nextInput)
{
    int answer = currentState;
    int ch = nextInput - 'a';

    // If goto is not defined, use failure function
    while (g[answer][ch] == -1)
        answer = f[answer];

    return g[answer][ch];
}

// This function finds all occurrences of all array words
// in text.
void searchWords(string arr[], int k, string text)
{
    // Preprocess patterns.
    // Build machine with goto, failure and output functions
    buildMatchingMachine(arr, k);

    // Initialize current state
    int currentState = 0;

    // Traverse the text through the built machine to find
    // all occurrences of words in arr[]
    for (int i = 0; i < text.size(); ++i)
    {
        currentState = findNextState(currentState, text[i]);

        // If match not found, move to next state
        if (out[currentState] == 0)
            continue;

        // Match found, print all matching words of arr[]
        // using output function.
        for (int j = 0; j < k; ++j)
        {
            if (out[currentState] & (1 << j))
            {
                cout << "Word " << arr[j] << " appears from "
                    << i - arr[j].size() + 1 << " to " << i << endl;
            }
        }
    }
}

// Driver program to test above
int main()
{
    string arr[] = {"he", "she", "hers", "his"};
    string text = "ahishers";
    int k = sizeof(arr)/sizeof(arr[0]);

    searchWords(arr, k, text);

    return 0;
}

```

Output:

```

Word his appears from 1 to 3
Word he appears from 4 to 5
Word she appears from 3 to 5
Word hers appears from 4 to 7

```

Source:

<http://www.cs.uku.fi/~kilpelai/BSA05/lectures/slides04.pdf>

This article is contributed by **Ayush Govil**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Pattern Searching

kasai's Algorithm for Construction of LCP array from Suffix Array

Background

Suffix Array : A suffix array is a sorted array of all suffixes of a given string.
Let the given string be "banana".

0 banana	5 a
1 anana	Sort the Suffixes 3 ana
2 nana	-----> 1 anana
3 ana	alphabetically 0 banana
4 na	4 na
5 a	2 nana

The suffix array for "banana" :

suffix[] = {5, 3, 1, 0, 4, 2}

We have discussed Suffix Array and its O(nLogn) construction .

Once Suffix array is built, we can use it to efficiently search a pattern in a text. For example, we can use Binary Search to find a pattern (Complete code for the same is discussed [here](#))

LCP Array

The Binary Search based solution discussed [here](#) takes $O(m \cdot \log n)$ time where m is length of the pattern to be searched and n is length of the text. With the help of LCP array, we can search a pattern in $O(m + \log n)$ time. For example, if our task is to search "ana" in "banana", $m = 3$, $n = 5$.

LCP Array is an array of size n (like Suffix Array). A value $\text{lcp}[i]$ indicates length of the longest common prefix of the suffixes indexed by $\text{suffix}[i]$ and $\text{suffix}[i+1]$. $\text{suffix}[n-1]$ is not defined as there is no suffix after it.

```
txt[0..n-1] = "banana"
suffix[] = {5, 3, 1, 0, 4, 2}
lcp[] = {1, 3, 0, 0, 2, 0}
```

Suffixes represented by suffix array in order are:
{"a", "ana", "anana", "banana", "na", "nana"}

```
lcp[0] = Longest Common Prefix of "a" and "ana" = 1
lcp[1] = Longest Common Prefix of "ana" and "anana" = 3
lcp[2] = Longest Common Prefix of "anana" and "banana" = 0
lcp[3] = Longest Common Prefix of "banana" and "na" = 0
lcp[4] = Longest Common Prefix of "na" and "nana" = 2
lcp[5] = Longest Common Prefix of "nana" and None = 0
```

How to construct LCP array?

LCP array construction is done two ways:

- 1) Compute the LCP array as a byproduct to the suffix array (Manber & Myers Algorithm)
- 2) Use an already constructed suffix array in order to compute the LCP values. (Kasai Algorithm).

There exist algorithms that can construct Suffix Array in $O(n)$ time and therefore we can always construct LCP array in $O(n)$ time. But in the below implementation, a $O(n \log n)$ algorithm is discussed.

Kasai's Algorithm

In this article Kasai's Algorithm is discussed. The algorithm constructs LCP array from suffix array and input text in $O(n)$ time. The idea is based on below fact:
Let lcp of suffix beginning at $\text{txt}[i]$ be k . If k is greater than 0, then lcp for suffix beginning at $\text{txt}[i+1]$ will be at-least $k-1$. The reason is, relative order of characters remain same. If we delete the first character from both suffixes, we know that at least k characters will match. For example for substring "ana", lcp is 3, so for string "na" lcp will be at-least 2. Refer [this](#) for proof.

Below is C++ implementation of Kasai's algorithm.

```
// C++ program for building LCP array for given text
#include <bits/stdc++.h>
using namespace std;

// Structure to store information of a suffix
struct suffix
{
    int index; // To store original index
    int rank[2]; // To store ranks and next rank pair
};

// A comparison function used by sort() to compare two suffixes
// Compares two pairs, returns 1 if first pair is smaller
int cmp(struct suffix a, struct suffix b)
{
    return (a.rank[0] == b.rank[0]) ? (a.rank[1] < b.rank[1] ? 1 : 0) :
        (a.rank[0] < b.rank[0] ? 1 : 0);
}

// This is the main function that takes a string 'txt' of size n as an
// argument, builds and return the suffix array for the given string
vector<int> buildSuffixArray(string txt, int n)
{
    // A structure to store suffixes and their indexes
    struct suffix suffixes[n];

    // Store suffixes and their indexes in an array of structures.
    // The structure is needed to sort the suffixes alphabetically
    // and maintain their old indexes while sorting
    for (int i = 0; i < n; i++)
    {
        suffixes[i].index = i;
        suffixes[i].rank[0] = txt[i] - 'a';
        suffixes[i].rank[1] = ((i + 1) < n) ? (txt[i + 1] - 'a') : -1;
    }

    // Sort the suffixes using the comparison function
    // defined above.
    sort(suffixes, suffixes+n, cmp);

    // At this point, all suffixes are sorted according to first
    // 2 characters. Let us sort suffixes according to first 4
    // characters, then first 8 and so on
    int ind[n]; // This array is needed to get the index in suffixes[]
    // from original index. This mapping is needed to get
    // next suffix.
    for (int k = 4; k < 2*n; k = k*2)
    {
        // Assigning rank and index values to first suffix
        int rank = 0;
        int prev_rank = suffixes[0].rank[0];
        suffixes[0].rank[0] = rank;
        ind[suffixes[0].index] = 0;

        // Assigning rank to suffixes
        for (int i = 1; i < n; i++)
        {
            // If first rank and next ranks are same as that of previous
            // suffix in array, assign the same new rank to this suffix
            if (suffixes[i].rank[0] == prev_rank &&
                suffixes[i].rank[1] == suffixes[i-1].rank[1])
            {
                prev_rank = suffixes[i].rank[0];
                suffixes[i].rank[0] = rank;
            }
        }
    }
}
```

```

    else // Otherwise increment rank and assign
    {
        prev_rank = suffixes[i].rank[0];
        suffixes[i].rank[0] = ++rank;
    }
    ind[suffixes[i].index] = i;
}

// Assign next rank to every suffix
for (int i = 0; i < n; i++)
{
    int nextIndex = suffixes[i].index + k/2;
    suffixes[i].rank[1] = (nextIndex < n) ? suffixes[ind[nextIndex]].rank[0] : -1;
}

// Sort the suffixes according to first k characters
sort(suffixes, suffixes+n, cmp);
}

// Store indexes of all sorted suffixes in the suffix array
vector<int> suffixArr;
for (int i = 0; i < n; i++)
    suffixArr.push_back(suffixes[i].index);

// Return the suffix array
return suffixArr;
}

/* To construct and return LCP */
vector<int> kasai(string txt, vector<int> suffixArr)
{
    int n = suffixArr.size();

    // To store LCP array
    vector<int> lcp(n, 0);

    // An auxiliary array to store inverse of suffix array
    // elements. For example if suffixArr[0] is 5, the
    // invSuff[5] would store 0. This is used to get next
    // suffix string from suffix array.
    vector<int> invSuff(n, 0);

    // Fill values in invSuff[]
    for (int i=0; i < n; i++)
        invSuff[suffixArr[i]] = i;

    // Initialize length of previous LCP
    int k = 0;

    // Process all suffixes one by one starting from
    // first suffix in txt[]
    for (int i=0; i < n; i++)
    {
        /* If the current suffix is at n-1, then we don't
         have next substring to consider. So lcp is not
         defined for this substring, we put zero. */
        if (invSuff[i] == n-1)
        {
            k = 0;
            continue;
        }

        /* j contains index of the next substring to
         be considered to compare with the present
         substring, i.e., next string in suffix array */
        int j = suffixArr[invSuff[i]+1];

        // Directly start matching from k'th index as
        // at-least k-1 characters will match
        while (i < n && j < n && txt[i+k] == txt[j+k])
            k++;

        lcp[invSuff[i]] = k; // lcp for the present suffix.

        // Deleting the starting character from the string.
        if (k>0)
            k--;
    }

    // return the constructed lcp array
    return lcp;
}

// Utility function to print an array
void printArr(vector<int> arr, int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program
int main()
{
    string str = "banana";

    vector<int> suffixArr = buildSuffixArray(str, str.length());
    int n = suffixArr.size();

    cout << "Suffix Array : \n";
    printArr(suffixArr, n);

    vector<int> lcp = kasai(str, suffixArr);

    cout << "\nLCP Array : \n";
    printArr(lcp, n);
}

```

```
    return 0;
}
```

Output:

```
Suffix Array :  
5 3 1 0 4 2
```

```
LCP Array :  
1 3 0 0 2 0
```

Illustration:

```
txt[] = "banana", suffix[] = {5, 3, 1, 0, 4, 2}
```

```
Suffix array represents  
{"a", "ana", "anana", "banana", "na", "nana"}
```

```
Inverse Suffix Array would be  
invSuff[] = {3, 2, 5, 1, 4, 0}
```

LCP values are evaluated in below order

We first compute LCP of first suffix in text which is **"banana"**. We need next suffix in suffix array to compute LCP (Remember lcp[i] is defined as Longest Common Prefix of suffix[i] and suffix[i+1]). **To find the next suffix in suffixArr[], we use SuffixInv[]**. The next suffix is **"na"**. Since there is no common prefix between **"banana"** and **"na"**, the value of LCP for **"banana"** is 0 and it is at index 3 in suffix array, so we fill **lcp[3]** as 0.

Next we compute LCP of second suffix which **"anana"**. Next suffix of **"anana"** in suffix array is **"banana"**. Since there is no common prefix, the value of LCP for **"anana"** is 0 and it is at index 2 in suffix array, so we fill **lcp[2]** as 0.

Next we compute LCP of third suffix which **"nana"**. Since there is no next suffix, the value of LCP for **"nana"** is not defined. We fill **lcp[5]** as 0.

Next suffix in text is **"ana"**. Next suffix of **"ana"** in suffix array is **"anana"**. Since there is a common prefix of length 3, the value of LCP for **"ana"** is 3. We fill **lcp[1]** as 3.

Now we lcp for next suffix in text which is **"na"**. This is where Kasai's algorithm uses the trick that LCP value must be at least 2 because previous LCP value was 3. Since there is no character after **"na"**, final value of LCP is 2. We fill **lcp[4]** as 2.

Next suffix in text is **"a"**. LCP value must be at least 1 because previous value was 2. Since there is no character after **"a"**, final value of LCP is 1. We fill **lcp[0]** as 1.

We will soon be discussing implementation of search with the help of LCP array and how LCP array helps in reducing time complexity to O(m + Log n).

References:

<http://web.stanford.edu/class/cs97si/suffix-array.pdf>
<http://www.mi.fu-berlin.de/wiki/pub/ABI/RnaSeqP4/suffix-array.pdf>
<http://codeforces.com/blog/entry/12796>

This article is contributed by **Prakhar Agrawal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Advanced Data Structure
Pattern Searching
Technical Scripter

Z algorithm (Linear time pattern searching Algorithm)

This algorithm finds all occurrences of a pattern in a text in linear time. Let length of text be n and of pattern be m, then total time taken is O(m + n) with linear space complexity. Now we can see that both time and space complexity is same as KMP algorithm but this algorithm is Simpler to understand.

In this algorithm, we construct a Z array.

What is Z Array?

For a string str[0..n-1], Z array is of same length as string. An element Z[i] of Z array stores length of the longest substring starting from str[i] which is also a prefix of str[0..n-1]. The first entry of Z array is meaning less as complete string is always prefix of itself.

Example:

Index	0 1 2 3 4 5 6 7 8 9 10 11
Text	a a b c a a b x a a a z
Z values	X 1 0 0 3 1 0 0 2 2 1 0

More Examples:

str = "aaaaaa"

Z[] = {x, 5, 4, 3, 2, 1}

str = "aabaaacd"

Z[] = {x, 1, 0, 2, 1, 0, 0}

str = "abababab"

Z[] = {x, 0, 6, 0, 4, 0, 2, 0}

How is Z array helpful in Searching Pattern in Linear time?

The idea is to concatenate pattern and text, and create a string "P\$T" where P is pattern, \$ is a special character should not be present in pattern and text, and T is text. Build the Z array for concatenated string. In Z array, if Z value at any point is equal to pattern length, then pattern is present at that point.

Example:

Pattern P = "aab", Text T = "baabaa"

The concatenated string is = "aab\$baabaa"

Z array for above concatenated string is {x, 1, 0, 0, 0,

3, 1, 0, 2, 1}.

Since length of pattern is 3, the value 3 in Z array indicates presence of pattern.

How to construct Z array?

A Simple Solution is two run two nested loops, the outer loop goes to every index and the inner loop finds length of the longest prefix that matches substring starting at current index. The time complexity of this solution is O(n²).

We can construct Z array in linear time.

The idea is to maintain an interval $[L, R]$ which is the interval with max R such that $[L, R]$ is prefix substring (substring which is also prefix).

Steps for maintaining this interval are as follows –

- 1) If $i > R$ then there is no prefix substring that starts before i and ends after i , so we reset L and R and compute new $[L, R]$ by comparing $\text{str}[0..]$ to $\text{str}[i..]$ and get $Z[i] (= R - L + 1)$.
- 2) If $i = \min(Z[K], R - i + 1)$ because $\text{str}[i..]$ matches with $\text{str}[K..]$ for atleast $R - i + 1$ characters (they are in $[L, R]$ interval which we know is a prefix substring).
Now two sub cases arise –
 - a) If $Z[K] = R - i + 1$ then it is possible to extend the $[L, R]$ interval thus we will set L as i and start matching from $\text{str}[R]$ onwards and get new R then we will update interval $[L, R]$ and calculate $Z[i] (= R - L + 1)$.

For better understanding of above step by step procedure please check this animation – <http://www.utdallas.edu/~besp/demo/John2010/z-algorithm.htm>

The algorithm runs in linear time because we never compare character less than R and with matching we increase R by one so there are at most T comparisons. In mismatch case, mismatch happen only once for each i (because of which R stops), that's another at most T comparison making overall linear complexity.

Below is C++ implementation of Z algorithm for pattern searching.

```
// A C++ program that implements Z algorithm for pattern searching
#include<iostream>
using namespace std;

void getZarr(string str, int Z[])
{
    // prints all occurrences of pattern in text using Z algo
    void search(string text, string pattern)
    {
        // Create concatenated string "P$T"
        string concat = pattern + "$" + text;
        int l = concat.length();

        // Construct Z array
        int Z[l];
        getZarr(concat, Z);

        // now looping through Z array for matching condition
        for (int i = 0; i < l; ++i)
        {
            // if Z[i] (matched region) is equal to pattern
            // length we got the pattern
            if (Z[i] == pattern.length())
                cout << "Pattern found at index "
                     << i - pattern.length() - 1 << endl;
        }
    }

    // Fills Z array for given string str[]
    void getZarr(string str, int Z[])
    {
        int n = str.length();
        int L, R, K;

        // [L,R] make a window which matches with prefix of s
        L = R = 0;
        for (int i = 1; i < n; ++i)
        {
            // if i>R nothing matches so we will calculate.
            // Z[i] using naive way.
            if (i > R)
            {
                L = R = i;

                // R-L = 0 in starting, so it will start
                // checking from 0'th index. For example,
                // for "ababab" and i = 1, the value of R
                // remains 0 and Z[i] becomes 0. For string
                // "aaaaaa" and i = 1, Z[i] and R become 5
                while (R < n && str[R - L] == str[R])
                    R++;
                Z[i] = R - L;
                R--;
            }
            else
            {
                // k = i-L so k corresponds to number which
                // matches in [L,R] interval.
                K = i - L;

                // if Z[k] is less than remaining interval
                // then Z[i] will be equal to Z[k].
                // For example, str = "ababab", i = 3, R = 5
                // and L = 2
                if (Z[k] < R - i + 1)
                    Z[i] = Z[k];

                // For example str = "aaaaaa" and i = 2, R is 5,
                // L is 0
                else
                {
                    // else start from R and check manually
                    L = i;
                    while (R < n && str[R - L] == str[R])
                        R++;
                    Z[i] = R - L;
                    R--;
                }
            }
        }
    }
}
```

```

// Driver program
int main()
{
    string text = "GEEKS FOR GEEKS";
    string pattern = "GEEK";
    search(text, pattern);
    return 0;
}

```

Output:

```

Pattern found at index 0
Pattern found at index 10

```

This article is contributed by [Utkarsh Trivedi](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

[Pattern Searching](#)
[Pattern Searching](#)

Manacher's Algorithm – Linear Time Longest Palindromic Substring – Part 1

Given a string, find the longest substring which is palindrome.

- if the given string is "forgeeksskeegfor", the output should be "geeksskeeg"
- if the given string is "abaaba", the output should be "abaaba"
- if the given string is "babababa", the output should be "babababa"
- if the given string is "abcbaabcba", the output should be "abcbaabcba"

We have already discussed Naïve [$O(n^3)$] and quadratic [$O(n^2)$] approaches at [Set 1](#) and [Set 2](#).

In this article, we will talk about [Manacher's algorithm](#) which finds Longest Palindromic Substring in linear time.

One way ([Set 2](#)) to find a palindrome is to start from the center of the string and compare characters in both directions one by one. If corresponding characters on both sides (left and right of the center) match, then they will make a palindrome.

Let's consider string "babababa".

Here center of the string is 4th character (with index 3) b. If we match characters in left and right of the center, all characters match and so string "babababa" is a palindrome.



Here center position is not only the actual string character position but it could be the position between two characters also.

Consider string "abaaba" of even length. This string is palindrome around the position between 3rd and 4th characters a and a respectively.



To find Longest Palindromic Substring of a string of length N, one way is take each possible $2N + 1$ centers (the N character positions, N-1 between two character positions and 2 positions at left and right ends), do the character match in both left and right directions at each $2N+1$ centers and keep track of LPS. This approach takes $O(N^2)$ time and that's what we are doing in [Set 2](#).

Let's consider two strings "babababa" and "abaaba" as shown below:



In these two strings, left and right side of the center positions (position 7 in 1st string and position 6 in 2nd string) are symmetric. Why? Because the whole string is palindrome around the center position. If we need to calculate Longest Palindromic Substring at each $2N+1$ positions from left to right, then palindrome's symmetric property could help to avoid some of the unnecessary computations (i.e. character comparison). If there is a palindrome of some length L centered at any position P, then we may not need to compare all characters in left and right side at position P+1. We already calculated LPS at positions before P and they can help to avoid some of the comparisons after position P.

This use of information from previous positions at a later point of time makes the Manacher's algorithm linear. In [Set 2](#), there is no reuse of previous information and so that is quadratic.

Manacher's algorithm is probably considered complex to understand, so here we will discuss it in as detailed way as we can. Some of its portions may require multiple reading to understand it properly.

Let's look at string "babababa". In 3rd figure above, 15 center positions are shown. We need to calculate length of longest palindromic string at each of these positions.

- At position 0, there is no LPS at all (no character on left side to compare), so length of LPS will be 0.
- At position 1, LPS is a, so length of LPS will be 1.
- At position 2, there is no LPS at all (left and right characters a and b don't match), so length of LPS will be 0.
- At position 3, LPS is aba, so length of LPS will be 3.
- At position 4, there is no LPS at all (left and right characters b and a don't match), so length of LPS will be 0.
- At position 5, LPS is ababa, so length of LPS will be 5.

..... and so on

We store all these palindromic lengths in an array, say L. Then string S and LPS Length L look like below:



Similarly, LPS Length L of string "abaaba" will look like:

In LPS Array L:

- LPS length value at odd positions (the actual character positions) will be odd and greater than or equal to 1 (1 will come from the center character itself if nothing else matches in left and right side of it)
- LPS length value at even positions (the positions between two characters, extreme left and right positions) will be even and greater than or equal to 0 (0 will come when there is no match in left and right side)

Position and index for the string are two different things here. For a given string S of length N, indexes will be from 0 to N-1 (total N indexes) and positions will be from 0 to 2*N (total 2*N+1 positions).

LPS length value can be interpreted in two ways, one in terms of index and second in terms of position. LPS value d at position i ($L[i] = d$) tells that:

- Substring from position $i-d$ to $i+d$ is a palindrome of length d (in terms of position)
- Substring from index $(i-d)/2$ to $[(i+d)/2 - 1]$ is a palindrome of length d (in terms of index)

e.g. in string "abaaba", $L[3] = 3$ means substring from position 0 (3-3) to 6 (3+3) is a palindrome which is "aba" of length 3, it also means that substring from index 0 $[(3-3)/2]$ to 2 $[(3+3)/2 - 1]$ is a palindrome which is "aba" of length 3.

Now the main task is to compute LPS array efficiently. Once this array is computed, LPS of string S will be centered at position with maximum LPS length value.

We will see it in [Part 2](#).

This article is contributed by [Anurag Singh](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Pattern Searching
Strings
palindrome

Manacher's Algorithm – Linear Time Longest Palindromic Substring – Part 2

In [Manacher's Algorithm – Part 1](#), we gone through some of the basics and LPS length array.

Here we will see how to calculate LPS length array efficiently.

To calculate LPS array efficiently, we need to understand how LPS length for any position may relate to LPS length value of any previous already calculated position.
For string "abaaba", we see following:

If we look around position 3:

- LPS length value at position 2 and position 4 are same
- LPS length value at position 1 and position 5 are same

We calculate LPS length values from left to right starting from position 0, so we can see if we already know LPS length values at positions 1, 2 and 3 already then we may not need to calculate LPS length at positions 4 and 5 because they are equal to LPS length values at corresponding positions on left side of position 3.

If we look around position 6:

- LPS length value at position 5 and position 7 are same
- LPS length value at position 4 and position 8 are same

..... and so on.

If we already know LPS length values at positions 1, 2, 3, 4, 5 and 6 already then we may not need to calculate LPS length at positions 7, 8, 9, 10 and 11 because they are equal to LPS length values at corresponding positions on left side of position 6.

For string "abababa", we see following:

If we already know LPS length values at positions 1, 2, 3, 4, 5, 6 and 7 already then we may not need to calculate LPS length at positions 8, 9, 10, 11, 12 and 13 because they are equal to LPS length values at corresponding positions on left side of position 7.

Can you see why LPS length values are symmetric around positions 3, 6, 9 in string "abaaba"? That's because there is a palindromic substring around these positions. Same is the case in string "abababa" around position 7.

Is it always true that LPS length values around at palindromic center position are always symmetric (same)?

Answer is NO.

Look at positions 3 and 11 in string "abababa". Both positions have LPS length 3. Immediate left and right positions are symmetric (with value 0), but not the next one. Positions 1 and 5 (around position 3) are not symmetric. Similarly, positions 9 and 13 (around position 11) are not symmetric.

At this point, we can see that if there is a palindrome in a string centered at some position, then LPS length values around the center position may or may not be symmetric depending on some situation. If

we can identify the situation when left and right positions WILL BE SYMMETRIC around the center position, we NEED NOT calculate LPS length of the right position because it will be exactly same as LPS value of corresponding position on the left side which is already known. And this fact where we are avoiding LPS length computation at few positions makes Manacher's Algorithm linear.

In situations when left and right positions WILL NOT BE SYMMETRIC around the center position, we compare characters in left and right side to find palindrome, but here also algorithm tries to avoid certain no of comparisons. We will see all these scenarios soon.

Let's introduce few terms to proceed further:

(click to see it clearly)



- **centerPosition** – This is the position for which LPS length is calculated and let's say LPS length at centerPosition is d (i.e. $L[centerPosition] = d$)
- **centerRightPosition** – This is the position which is right to the centerPosition and d position away from centerPosition (i.e. $centerRightPosition = centerPosition + d$)
- **centerLeftPosition** – This is the position which is left to the centerPosition and d position away from centerPosition (i.e. $centerLeftPosition = centerPosition - d$)
- **currentRightPosition** – This is the position which is right of the centerPosition for which LPS length is not yet known and has to be calculated
- **currentLeftPosition** – This is the position on the left side of centerPosition which corresponds to the currentRightPosition
 $centerPosition - currentLeftPosition = currentRightPosition - centerPosition$
 $currentLeftPosition = 2 * centerPosition - currentRightPosition$
- **i-left palindrome** – The palindrome i positions left of centerPosition, i.e. at currentLeftPosition
- **i-right palindrome** – The palindrome i positions right of centerPosition, i.e. at currentRightPosition
- **center palindrome** – The palindrome at centerPosition

When we are at centerPosition for which LPS length is known, then we also know LPS length of all positions smaller than centerPosition. Let's say LPS length at centerPosition is d , i.e.
 $L[centerPosition] = d$

It means that substring between positions "centerPosition-d" to "centerPosition+d" is a palindrome.

Now we proceed further to calculate LPS length of positions greater than centerPosition.

Let's say we are at currentRightPosition ($>$ centerPosition) where we need to find LPS length.

For this we look at LPS length of currentLeftPosition which is already calculated.

If LPS length of currentLeftPosition is less than "centerRightPosition – currentRightPosition", then LPS length of currentRightPosition will be equal to LPS length of currentLeftPosition. So
 $L[currentRightPosition] = L[currentLeftPosition]$ if $L[currentLeftPosition] < centerRightPosition - currentRightPosition$. This is **Case 1**.

Let's consider below scenario for string "abababa":

(click to see it clearly)



We have calculated LPS length up-to position 7 where $L[7] = 7$, if we consider position 7 as centerPosition, then centerLeftPosition will be 0 and centerRightPosition will be 14.
Now we need to calculate LPS length of other positions on the right of centerPosition.

For $currentRightPosition = 8$, $currentLeftPosition$ is 6 and $L[currentLeftPosition] = 0$

Also $centerRightPosition - currentRightPosition = 14 - 8 = 6$

Case 1 applies here and so $L[currentRightPosition] = L[8] = 0$

Case 1 applies to positions 10 and 12, so,

$L[10] = L[4] = 0$

$L[12] = L[2] = 0$

If we look at position 9, then:

$currentRightPosition = 9$

$currentLeftPosition = 2 * centerPosition - currentRightPosition = 2 * 7 - 9 = 5$

$centerRightPosition - currentRightPosition = 14 - 9 = 5$

Here $L[currentLeftPosition] = centerRightPosition - currentRightPosition$, so Case 1 doesn't apply here. Also note that centerRightPosition is the extreme end position of the string. That means center palindrome is suffix of input string. In that case, $L[currentRightPosition] = L[currentLeftPosition]$. This is **Case 2**.

Case 2 applies to positions 9, 11, 13 and 14, so:

$L[9] = L[5] = 5$

$L[11] = L[3] = 3$

$L[13] = L[1] = 1$

$L[14] = L[0] = 0$

What is really happening in Case 1 and Case 2? This is just utilizing the palindromic symmetric property and without any character match, it is finding LPS length of new positions.

When a bigger length palindrome contains a smaller length palindrome centered at left side of its own center, then based on symmetric property, there will be another same smaller palindrome centered on the right of bigger palindrome center. If left side smaller palindrome is not prefix of bigger palindrome, then **Case 1** applies and if it is a prefix AND bigger palindrome is suffix of the input string itself, then **Case 2** applies.

*The longest palindrome i places to the right of the current center (the i -right palindrome) is as long as the longest palindrome i places to the left of the current center (the i -left palindrome) if the i -left palindrome is completely contained in the longest palindrome around the current center (the center palindrome) and the i -left palindrome is not a prefix of the center palindrome (**Case 1**) or (i.e. when i -left palindrome is a prefix of center palindrome) if the center palindrome is a suffix of the entire string (**Case 2**).*

In Case 1 and Case 2, i -right palindrome can't expand more than corresponding i -left palindrome (can you visualize why it can't expand more?), and so LPS length of i -right palindrome is exactly same as LPS length of i -left palindrome.

Here both i -left and i -right palindromes are completely contained in center palindrome (i.e. $L[currentLeftPosition] \leq centerRightPosition - currentRightPosition$)

Now if i -left palindrome is not a prefix of center palindrome ($L[currentLeftPosition] < centerRightPosition - currentRightPosition$), that means that i -left palindrome was not able to expand up-to position $centerLeftPosition$.

If we look at following with $centerPosition = 11$, then

(click to see it clearly)



$centerLeftPosition$ would be $11 - 9 = 2$, and $centerRightPosition$ would be $11 + 9 = 20$

If we take $currentRightPosition = 15$, it's $currentLeftPosition$ is 7. Case 1 applies here and so $L[15] = 3$. i -left palindrome at position 7 is "bab" which is completely contained in center palindrome at position 11 (which is "dbabcbabd"). We can see that i -right palindrome (at position 15) can't expand more than i -left palindrome (at position 7).

If there was a possibility of expansion, i -left palindrome could have expanded itself more already. But there is no such possibility as i -left palindrome is prefix of center palindrome. So due to symmetry property, i -right palindrome will be exactly same as i -left palindrome and it can't expand more. This makes $L[currentRightPosition] = L[currentLeftPosition]$ in Case 1.

Now if we consider centerPosition = 19, then centerLeftPosition = 12 and centerRightPosition = 26

If we take currentRightPosition = 23, it's currentLeftPosition is 15. Case 2 applies here and so $L[23] = 3$. i-left palindrome at position 15 is "bab" which is completely contained in center palindrome at position 19 (which is "babdbab"). In Case 2, where i-left palindrome is prefix of center palindrome, i-right palindrome can't expand more than length of i-left palindrome because center palindrome is suffix of input string so there are no more character left to compare and expand. This makes $L[currentRightPosition] = L[currentLeftPosition]$ in Case 2.

Case 1: $L[currentRightPosition] = L[currentLeftPosition]$ applies when:

- i-left palindrome is completely contained in center palindrome
- i-left palindrome is NOT a prefix of center palindrome

Above above conditions are satisfied when

$L[currentLeftPosition] < centerRightPosition - currentRightPosition$

Case 2: $L[currentRightPosition] = L[currentLeftPosition]$ applies when:

- i-left palindrome is prefix of center palindrome (means completely contained also)
- center palindrome is suffix of input string

Above conditions are satisfied when

$L[currentLeftPosition] = centerRightPosition - currentRightPosition$ (For 1st condition) AND

$centerRightPosition = 2 * N$ where N is input string length N (For 2nd condition).

Case 3: $L[currentRightPosition] >= L[currentLeftPosition]$ applies when:

- i-left palindrome is prefix of center palindrome (and so i-left palindrome is completely contained in center palindrome)
- center palindrome is NOT suffix of input string

Above conditions are satisfied when

$L[currentLeftPosition] = centerRightPosition - currentRightPosition$ (For 1st condition) AND

$centerRightPosition < 2 * N$ where N is input string length N (For 2nd condition).

In this case, there is a possibility of i-right palindrome expansion and so length of i-right palindrome is at least as long as length of i-left palindrome.

Case 4: $L[currentRightPosition] >= centerRightPosition - currentRightPosition$ applies when:

- i-left palindrome is NOT completely contained in center palindrome

Above condition is satisfied when

$L[currentLeftPosition] > centerRightPosition - currentRightPosition$

In this case, length of i-right palindrome is at least as long ($centerRightPosition - currentRightPosition$) and there is a possibility of i-right palindrome expansion.

In following figure,

(click to see it clearly)



If we take center position 7, then Case 3 applies at currentRightPosition 11 because i-left palindrome at currentLeftPosition 3 is a prefix of center palindrome and i-right palindrome is not suffix of input string, so here $L[11] = 9$, which is greater than i-left palindrome length $L[3] = 3$. In the case, it is guaranteed that $L[11]$ will be at least 3, and so in implementation, we 1st set $L[11] = 3$ and then we try to expand it by comparing characters in left and right side starting from distance 4 (As up-to distance 3, it is already known that characters will match).

If we take center position 11, then Case 4 applies at currentRightPosition 15 because $L[currentLeftPosition] = L[7] = 7 > centerRightPosition - currentRightPosition = 20 - 15 = 5$. In the case, it is guaranteed that $L[15]$ will be at least 5, and so in implementation, we 1st set $L[15] = 5$ and then we try to expand it by comparing characters in left and right side starting from distance 5 (As up-to distance 5, it is already known that characters will match).

Now one point left to discuss is, when we work at one center position and compute LPS lengths for different rightPositions, how to know that what would be next center position. We change centerPosition to currentRightPosition if palindrome centered at currentRightPosition expands beyond centerRightPosition.

Here we have seen four different cases on how LPS length of a position will depend on a previous position's LPS length.

In Part 3, we have discussed code implementation of it and also we have looked at these four cases in a different way and implement that too.

This article is contributed by [Anurag Singh](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Pattern Searching
Strings
palindrome

Manacher's Algorithm – Linear Time Longest Palindromic Substring – Part 3

In Manacher's Algorithm Part 1 and Part 2, we gone through some of the basics, understood LPS length array and how to calculate it efficiently based on four cases. Here we will implement the same.

We have seen that there are no new character comparison needed in case 1 and case 2. In case 3 and case 4, necessary new comparison are needed.

In following figure,

(click to see it clearly)



If at all we need a comparison, we will only compare actual characters, which are at "odd" positions like 1, 3, 5, 7, etc.

Even positions do not represent a character in string, so no comparison will be performed for even positions.

If two characters at different odd positions match, then they will increase LPS length by 2.

There are many ways to implement this depending on how even and odd positions are handled. One way would be to create a new string 1st where we insert some unique character (say #, \$ etc) in all even positions and then run algorithm on that (to avoid different way of even and odd position handling). Other way could be to work on given string itself but here even and odd positions should be handled appropriately.

Here we will start with given string itself. When there is a need of expansion and character comparison required, we will expand in left and right positions one by one. When odd position is found, comparison will be done and LPS Length will be incremented by ONE. When even position is found, no comparison done and LPS Length will be incremented by ONE (So overall, one odd and one even positions on both left and right side will increase LPS Length by TWO).

C/C++

```
// A C program to implement Manacher's Algorithm
#include <stdio.h>
#include <string.h>

char text[100];
```

```

void findLongestPalindromicString()
{
    int N = strlen(text);
    if(N == 0)
        return;
    N = 2*N + 1; //Position count
    int L[N]; //LPS Length Array
    L[0] = 0;
    L[1] = 1;
    int C = 1; //centerPosition
    int R = 2; //centerRightPosition
    int i = 0; //currentRightPosition
    int iMirror; //currentLeftPosition
    int expand = -1;
    int diff = -1;
    int maxLPSLength = 0;
    int maxLPSCenterPosition = 0;
    int start = -1;
    int end = -1;

    //Uncomment it to print LPS Length array
    //printf("%d %d ", L[0], L[1]);
    for (i = 2; i < N; i++)
    {
        //get currentLeftPosition iMirror for currentRightPosition i
        iMirror = 2*C-i;
        //Reset expand - means no expansion required
        expand = 0;
        diff = R - i;
        //If currentRightPosition i is within centerRightPosition R
        if(diff > 0)
        {
            if(L[iMirror] < diff) // Case 1
                L[i] = L[iMirror];
            else if(L[iMirror] == diff && i == N-1) // Case 2
                L[i] = L[iMirror];
            else if(L[iMirror] == diff && i < N-1) // Case 3
            {
                L[i] = L[iMirror];
                expand = 1; //expansion required
            }
            else if(L[iMirror] > diff) // Case 4
            {
                L[i] = diff;
                expand = 1; //expansion required
            }
        }
        else
        {
            L[i] = 0;
            expand = 1; //expansion required
        }

        if(expand == 1)
        {
            //Attempt to expand palindrome centered at currentRightPosition i
            //Here for odd positions, we compare characters and
            //if match then increment LPS Length by ONE
            //If even position, we just increment LPS by ONE without
            //any character comparison
            while ((i + L[i]) < N && (i - L[i]) > 0) &&
                ((i + L[i] + 1) % 2 == 0) ||
                (text[(i + L[i] + 1)/2] == text[(i-L[i]-1)/2]))
            {
                L[i]++;
            }
        }

        if(L[i] > maxLPSLength) // Track maxLPSLength
        {
            maxLPSLength = L[i];
            maxLPSCenterPosition = i;
        }

        // If palindrome centered at currentRightPosition i
        // expand beyond centerRightPosition R,
        // adjust centerPosition C based on expanded palindrome.
        if (i + L[i] > R)
        {
            C = i;
            R = i + L[i];
        }

        //Uncomment it to print LPS Length array
        //printf("%d ", L[i]);
    }

    //printf("\n");
    start = (maxLPSCenterPosition - maxLPSLength)/2;
    end = start + maxLPSLength - 1;
    //printf("start: %d end: %d\n", start, end);
    printf("LPS of string is %s : ", text);
    for(i=start; i<=end; i++)
        printf("%c", text[i]);
    printf("\n");
}

int main(int argc, char *argv[])
{
    strcpy(text, "babcbabcbaccba");
    findLongestPalindromicString();

    strcpy(text, "abaaba");
    findLongestPalindromicString();

    strcpy(text, "abababa");
    findLongestPalindromicString();
}

```

```

strcpy(text, "abcbabcbabcba");
findLongestPalindromicString();

strcpy(text, "forgeeksskeegfor");
findLongestPalindromicString();

strcpy(text, "caba");
findLongestPalindromicString();

strcpy(text, "abacdfgdcaba");
findLongestPalindromicString();

strcpy(text, "abacdfgdcabba");
findLongestPalindromicString();

strcpy(text, "abacdedcaba");
findLongestPalindromicString();

return 0;
}

```

Python

```

# Python program to implement Manacher's Algorithm

def findLongestPalindromicString(text):
    N = len(text)
    if N == 0:
        return
    C = 1 # centerPosition
    R = 2 # centerRightPosition
    i = 0 # currentRightPosition
    iMirror = 0 # currentLeftPosition
    maxLPSLength = 0
    maxLPSCenterPosition = 0
    start = -1
    end = -1
    diff = -1

    # Uncomment it to print LPS Length array
    # printf("%d %d ", L[0], L[1])
    for i in xrange(2,N):

        # get currentLeftPosition iMirror for currentRightPosition i
        iMirror = 2*C-i
        L[i] = 0
        diff = R - i
        # If currentRightPosition i is within centerRightPosition R
        if diff > 0:
            L[i] = min(L[iMirror], diff)

        # Attempt to expand palindrome centered at currentRightPosition i
        # Here for odd positions, we compare characters and
        # if match then increment LPS Length by ONE
        # If even position, we just increment LPS by ONE without
        # any character comparison
        try:
            while ((i+L[i]) < N and (i-L[i]) > 0) and \
                (((i+L[i]+1) % 2 == 0) or \
                (text[(i+L[i]+1)/2] == text[(i-L[i]-1)/2])):
                L[i]+=1
        except Exception as e:
            pass

        if L[i] > maxLPSLength:
            maxLPSLength = L[i]
            maxLPSCenterPosition = i

        # If palindrome centered at currentRightPosition i
        # expand beyond centerRightPosition R,
        # adjust centerPosition C based on expanded palindrome.
        if i + L[i] > R:
            C = i
            R = i + L[i]

    # Uncomment it to print LPS Length array
    # printf("%d ", L[i]);
    start = (maxLPSCenterPosition - maxLPSLength) / 2
    end = start + maxLPSLength - 1
    print "LPS of string is " + text + " :",
    print text[start:end+1],
    print "\n",

# Driver program
text1 = "babcbabcbaccba"
findLongestPalindromicString(text1)

text2 = "abaaba"
findLongestPalindromicString(text2)

text3 = "abababa"
findLongestPalindromicString(text3)

text4 = "abcbabcbabcba"
findLongestPalindromicString(text4)

text5 = "forgeeksskeegfor"
findLongestPalindromicString(text5)

```

```

text6 = "caba"
findLongestPalindromicString(text6)

text7 = "abacdfgdcaba"
findLongestPalindromicString(text7)

text8 = "abacdfgdcabba"
findLongestPalindromicString(text8)

text9 = "abacdedcaba"
findLongestPalindromicString(text9)

# This code is contributed by BHAVYA JAIN

```

Output:

```

LPS of string is babcbabcacba : abcbabca
LPS of string is abaaba : abaaba
LPS of string is abababa : abababa
LPS of string is abcbabcbabca : abcbabcbabca
LPS of string is forgeeksskeegfor : geeksskeeg
LPS of string is caba : aba
LPS of string is abacdgdcab : aba
LPS of string is abacdgdcabba : abba
LPS of string is abacdedcaba : abacdedcaba

```

This is the implementation based on the four cases discussed in Part 2. In Part 4, we have discussed a different way to look at these four cases and few other approaches.

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Pattern Searching
Strings
palindrome

Manacher's Algorithm – Linear Time Longest Palindromic Substring – Part 4

In Manacher's Algorithm [Part 1](#) and [Part 2](#), we gone through some of the basics, understood LPS length array and how to calculate it efficiently based on four cases. In [Part 3](#), we implemented the same.

Here we will review the four cases again and try to see it differently and implement the same.

All four cases depends on LPS length value at currentLeftPosition ($L[iMirror]$) and value of ($centerRightPosition - currentRightPosition$), i.e. $(R - i)$. These two information are know before which helps us to reuse previous available information and avoid unnecessary character comparison.

(click to see it clearly)



If we look at all four cases, we will see that we 1st set minimum of $L[iMirror]$ and $R-i$ to $L[i]$ and then we try to expand the palindrome in whichever case it can expand.

Above observation may look more intuitive, easier to understand and implement, given that one understands LPS length array, position, index, symmetry property etc.

C/C++

```

// A C program to implement Manacher's Algorithm
#include <stdio.h>
#include <string.h>

char text[100];
int min(int a, int b)
{
    int res = a;
    if(b < a)
        res = b;
    return res;
}

void findLongestPalindromicString()
{
    int N = strlen(text);
    if(N == 0)
        return;
    N = 2*N + 1; //Position count
    int L[N]; //LPS Length Array
    L[0] = 0;
    L[1] = 1;
    int C = 1; //centerPosition
    int R = 2; //centerRightPosition
    int i = 0; //currentRightPosition
    int iMirror; //currentLeftPosition
    int maxLPSLength = 0;
    int maxLPSCenterPosition = 0;
    int start = -1;
    int end = -1;
    int diff = -1;

    //Uncomment it to print LPS Length array
    //printf("%d %d ", L[0], L[1]);
    for (i = 2; i < N; i++)
    {
        //get currentLeftPosition iMirror for currentRightPosition i
        iMirror = 2*C-i;
        L[i] = 0;
        diff = R - i;
        //If currentRightPosition i is within centerRightPosition R
        if(diff > 0)
            L[i] = min(L[iMirror], diff);

        //Attempt to expand palindrome centered at currentRightPosition i
        //Here for odd positions, we compare characters and
    }
}

```

```

//if match then increment LPS Length by ONE
//if even position, we just increment LPS by ONE without
//any character comparison
while ( ((i + L[i]) < N && (i - L[i]) > 0) &&
((i + L[i] + 1) % 2 == 0) ||
(text[(i + L[i] + 1)/2] == text[(i - L[i] - 1)/2]))
{
L[i]++;
}

if(L[i] > maxLPSLength) // Track maxLPSLength
{
maxLPSLength = L[i];
maxLPSCenterPosition = i;
}

//If palindrome centered at currentRightPosition i
//expand beyond centerRightPosition R,
//adjust centerPosition C based on expanded palindrome.
if (i + L[i] > R)
{
C = i;
R = i + L[i];
}
//Uncomment it to print LPS Length array
//printf("%d ", L[i]);
}
//printf("\n");
start = (maxLPSCenterPosition - maxLPSLength)/2;
end = start + maxLPSLength - 1;
printf("LPS of string is %s : ", text);
for(i=start; i<=end; i++)
printf("%c", text[i]);
printf("\n");
}

```

```
int main(int argc, char *argv[])
{
```

```

strcpy(text, "babcbabcbaccba");
findLongestPalindromicString();

strcpy(text, "abaaba");
findLongestPalindromicString();

strcpy(text, "abababa");
findLongestPalindromicString();

strcpy(text, "abcbabcbabcb");
findLongestPalindromicString();

strcpy(text, "forgeeksskeegfor");
findLongestPalindromicString();

strcpy(text, "caba");
findLongestPalindromicString();

strcpy(text, "abacdfgdcaba");
findLongestPalindromicString();

strcpy(text, "abacdgedcaba");
findLongestPalindromicString();

return 0;
}
```

Python

```

# Python program to implement Manacher's Algorithm

def findLongestPalindromicString(text):
    N = len(text)
    if N == 0:
        return
    N = 2*N+1 # Position count
    L = [0]*N
    L[0] = 0
    L[1] = 1
    C = 1 # centerPosition
    R = 2 # centerRightPosition
    i = 0 # currentRightPosition
    iMirror = 0 # currentLeftPosition
    maxLPSLength = 0
    maxLPSCenterPosition = 0
    start = -1
    end = -1
    diff = -1

    # Uncomment it to print LPS Length array
    # printf("%d %d ", L[0], L[1]);
    for i in xrange(2,N):

        # get currentLeftPosition iMirror for currentRightPosition i
        iMirror = 2*C-i
        L[i] = 0
        diff = R - i
        # If currentRightPosition i is within centerRightPosition R
        if diff > 0:
            L[i] = min(L[iMirror], diff)

        # Attempt to expand palindrome centered at currentRightPosition i

```

```

# Here for odd positions, we compare characters and
# if match then increment LPS Length by ONE
# If even position, we just increment LPS by ONE without
# any character comparison
try:
    while ((i + L[i]) < N and (i - L[i]) > 0) and \
        (((i + L[i] + 1) % 2 == 0) or \
        (text[(i + L[i] + 1) / 2] == text[(i - L[i] - 1) / 2])):
        L[i] += 1
except Exception as e:
    pass

if L[i] > maxLPSLength:    # Track maxLPSLength
    maxLPSLength = L[i]
    maxLPSCenterPosition = i

# If palindrome centered at currentRightPosition i
# expand beyond centerRightPosition R,
# adjust centerPosition C based on expanded palindrome.
if i + L[i] > R:
    C = i
    R = i + L[i]

# Uncomment it to print LPS Length array
# print("%d ", L[i]);
start = (maxLPSCenterPosition - maxLPSLength) / 2
end = start + maxLPSLength - 1
print "LPS of string is " + text + ":" +
print text[start:end+1],
print "\n",

# Driver program
text1 = "babcbabcbaccba"
findLongestPalindromicString(text1)

text2 = "abaaba"
findLongestPalindromicString(text2)

text3 = "abababa"
findLongestPalindromicString(text3)

text4 = "abcbabcbabcb"
findLongestPalindromicString(text4)

text5 = "forgeeksskeegfor"
findLongestPalindromicString(text5)

text6 = "caba"
findLongestPalindromicString(text6)

text7 = "abacdfgdcaba"
findLongestPalindromicString(text7)

text8 = "abacdflgdcabba"
findLongestPalindromicString(text8)

text9 = "abacdedcaba"
findLongestPalindromicString(text9)

# This code is contributed by BHAVYA JAIN

```

Output:

```

LPS of string is babcbabcbaccba : abcbabcb
LPS of string is abaaba : abaaba
LPS of string is abababa : abababa
LPS of string is abcbabcbabcb : abcbabcbabcb
LPS of string is forgeeksskeegfor : geeksskeeg
LPS of string is caba : caba
LPS of string is abacdflgdcab : abac
LPS of string is abacdflgdcabba : abba
LPS of string is abacdedcaba : abacdedcaba

```

Other Approaches

We have discussed two approaches here. One in [Part 3](#) and other in current article. In both approaches, we worked on given string. Here we had to handle even and odd positions differently while comparing characters for expansion (because even positions do not represent any character in string).

To avoid this different handling of even and odd positions, we need to make even positions also to represent some character (actually all even positions should represent SAME character because they MUST match while character comparison). One way to do this is to set some character at all even positions by modifying given string or create a new copy of given string. For example, if input string is "abcb", new string should be "#a#b#c#b#" if we add # as unique character at even positions.

The two approaches discussed already can be modified a bit to work on modified string where different handling of even and odd positions will not be needed.

We may also add two DIFFERENT characters (not yet used anywhere in string at even and odd positions) at start and end of string as sentinels to avoid bound check. With these changes string "abcb" will look like "#a#b#f#b#\\$" where ^ and \\$ are sentinels.

This implementation may look cleaner with the cost of more memory.

We are not implementing these here as it's a simple change in given implementations.

Implementation of approach discussed in current article on a modified string can be found at [Longest Palindromic Substring Part II](#) and a [Java Translation](#) of the same by Princeton.

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Pattern Searching
Strings
palindrome

Longest Even Length Substring such that Sum of First and Second Half is same

Given a string 'str' of digits, find length of the longest substring of 'str', such that the length of the substring is 2k digits and sum of left k digits is equal to the sum of right k digits.

Examples:

```

Input: str = "123123"
Output: 6
The complete string is of even length and sum of first and second
half digits is same

Input: str = "1538023"
Output: 4
The longest substring with same first and second half sum is "5380"

```

We strongly recommend that you click here and practice it, before moving on to the solution.

Simple Solution [O(n³)]

A Simple Solution is to check every substring of even length. The following is C based implementation of simple approach.

```

// A simple C based program to find length of longest even length
// substring with same sum of digits in left and right
#include<stdio.h>
#include<string.h>

int findLength(char *str)
{
    int n = strlen(str);
    int maxlen = 0; // Initialize result

    // Choose starting point of every substring
    for (int i=0; i<n; i++)
    {
        // Choose ending point of even length substring
        for (int j = i+1; j<n; j += 2)
        {
            int length = j-i+1; //Find length of current substr

            // Calculate left & right sums for current substr
            int leftsum = 0, rightsum = 0;
            for (int k = 0; k<length/2; k++)
            {
                leftsum += (str[i+k]-'0');
                rightsum += (str[i+k+length/2]-'0');
            }

            // Update result if needed
            if (leftsum == rightsum && maxlen < length)
                maxlen = length;
        }
    }
    return maxlen;
}

// Driver program to test above function
int main(void)
{
    char str[] = "1538023";
    printf("Length of the substring is %d", findLength(str));
    return 0;
}

```

Output:

```
Length of the substring is 4
```

Dynamic Programming [O(n²) and O(n²) extra space]

The above solution can be optimized to work in O(n²) using **Dynamic Programming**. The idea is to build a 2D table that stores sums of substrings. The following is C based implementation of Dynamic Programming approach.

```

// A C based program that uses Dynamic Programming to find length of the
// longest even substring with same sum of digits in left and right half
#include <stdio.h>
#include <string.h>

int findLength(char *str)
{
    int n = strlen(str);
    int maxlen = 0; // Initialize result

    // A 2D table where sum[i][j] stores sum of digits
    // from str[i] to str[j]. Only filled entries are
    // the entries where j >= i
    int sum[n][n];

    // Fill the diagonal values for substrings of length 1
    for (int i = 0; i < n; i++)
        sum[i][i] = str[i]-'0';

    // Fill entries for substrings of length 2 to n
    for (int len=2; len<=n; len++)
    {
        // Pick i and j for current substring
        for (int i=0; i<n-len+1; i++)
        {
            int j = i+len-1;
            int k = len/2;

            // Calculate value of sum[i][j]
            sum[i][j] = sum[i][j-k] + sum[j-k+1][j];
        }
    }

    // Update result if 'len' is even, left and right
    // sums are same and len is more than maxlen
    if ((len%2 == 0 && sum[0][k] == sum[(j-k+1)][0]) ||
        && len > maxlen)

```

```

        maxlen = len;
    }
}
return maxlen;
}

// Driver program to test above function
int main(void)
{
    char str[] = "153803";
    printf("Length of the substring is %d", findLength(str));
    return 0;
}

```

Output:

```
Length of the substring is 4
```

Time complexity of the above solution is $O(n^2)$, but it requires $O(n^2)$ extra space.

[A $O(n^2)$ and $O(n)$ extra space solution]

The idea is to use a single dimensional array to store cumulative sum.

```

// A O(n^2) time and O(n) extra space solution
#include<bits/stdc++.h>
using namespace std;

int findLength(string str, int n)
{
    int sum[n+1]; // To store cumulative sum from first digit to nth digit
    sum[0] = 0;

    /* Store cumulative sum of digits from first to last digit */
    for (int i = 1; i <= n; i++)
        sum[i] = (sum[i-1] + str[i-1] - '0'); // convert chars to int

    int ans = 0; // initialize result

    /* consider all even length substrings one by one */
    for (int len = 2; len <= n; len += 2)
    {
        for (int i = 0; i <= n-len; i++)
        {
            int j = i + len - 1;

            /* Sum of first and second half is same than update ans */
            if (sum[i+len/2] - sum[i] == sum[i+len] - sum[i+len/2])
                ans = max(ans, len);
        }
    }
    return ans;
}

// Driver program to test above function
int main()
{
    string str = "123123";
    cout << "Length of the substring is " << findLength(str, str.length());
    return 0;
}

```

Output:

```
Length of the substring is 6
```

Thanks to Gaurav Ahirwar for suggesting this method.

[A $O(n^2)$ time and $O(1)$ extra space solution]

The idea is to consider all possible mid points (of even length substrings) and keep expanding on both sides to get and update optimal length as the sum of two sides become equal.

Below is C++ implementation of the above idea.

```

// A O(n^2) time and O(1) extra space solution
#include<bits/stdc++.h>
using namespace std;

int findLength(string str, int n)
{
    int ans = 0; // Initialize result

    // Consider all possible midpoints one by one
    for (int i = 0; i <= n-2; i++)
    {
        /* For current midpoint 'i', keep expanding substring on
           both sides, if sum of both sides becomes equal update
           ans */
        int l = i, r = i + 1;

        /* initialize left and right sum */
        int lsum = 0, rsum = 0;

        /* move on both sides till indexes go out of bounds */
        while (r < n && l >= 0)
        {
            lsum += str[l] - '0';
            rsum += str[r] - '0';
            if (lsum == rsum)
                ans = max(ans, r-l+1);
            l--;
            r++;
        }
    }
}

```

```

    return ans;
}

// Driver program to test above function
int main()
{
    string str = "123123";
    cout << "Length of the substring is " << findLength(str, str.length());
    return 0;
}

```

Output:

```
Length of the substring is 6
```

Thanks to Gaurav Ahirwar for suggesting this method.

This article is contributed by **Ashish Bansal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Dynamic Programming

Print all possible strings that can be made by placing spaces

Given a string you need to print all possible strings that can be made by placing spaces (zero or one) in between them.

```

Input: str[] = "ABC"
Output: ABC
       AB C
       A BC
       A B C

```

Source: [Amazon Interview Experience | Set 158, Round 1 ,Q 1.](#)

The idea is to use recursion and create a buffer that one by one contains all output strings having spaces. We keep updating buffer in every recursive call. If the length of given string is 'n' our updated string can have maximum length of $n + (n-1)$ i.e. $2n-1$. So we create buffer size of $2n$ (one extra character for string termination).

We leave 1st character as it is, starting from the 2nd character, we can either fill a space or a character. Thus one can write a recursive function like below.

C/C++

```

// C++ program to print permutations of a given string with spaces.
#include <iostream>
#include <cstring>
using namespace std;

/* Function recursively prints the strings having space pattern.
   i and j are indices in 'str[]' and 'buff[]' respectively */
void printPatternUtil(char str[], char buff[], int i, int j, int n)
{
    if (i==n)
    {
        buff[j] = '\0';
        cout << buff << endl;
        return;
    }

    // Either put the character
    buff[j] = str[i];
    printPatternUtil(str, buff, i+1, j+1, n);

    // Or put a space followed by next character
    buff[j] = ' ';
    buff[j+1] = str[i];

    printPatternUtil(str, buff, i+1, j+2, n);
}

// This function creates buff[] to store individual output string and uses
// printPatternUtil() to print all permutations.
void printPattern(char *str)
{
    int n = strlen(str);

    // Buffer to hold the string containing spaces
    char buf[2*n]; // 2n-1 characters and 1 string terminator

    // Copy the first character as it is, since it will be always
    // at first position
    buf[0] = str[0];

    printPatternUtil(str, buf, 1, 1, n);
}

// Driver program to test above functions
int main()
{
    char *str = "ABCD";
    printPattern(str);
    return 0;
}

```

Python

```

# Python program to print permutations of a given string with
# spaces.

# Utility function

```

```

def toString(List):
    s = ""
    for x in List:
        if x == "0":
            break
        s += x
    return s

# Function recursively prints the strings having space pattern.
# i and j are indices in 'str[]' and 'buff[]' respectively
def printPatternUtil(string, buff, i, j, n):
    if i == n:
        buff[j] = '0'
        printString(buff)
        return

    # Either put the character
    buff[j] = string[i]
    printPatternUtil(string, buff, i+1, j+1, n)

    # Or put a space followed by next character
    buff[j] = ' '
    buff[j+1] = string[i]

    printPatternUtil(string, buff, i+1, j+2, n)

# This function creates buff[] to store individual output string
# and uses printPatternUtil() to print all permutations.
def printPattern(string):
    n = len(string)
    buff = [0] * (2*n) # 2n-1 characters and 1 string terminator

    # Copy the first character as it is, since it will be always
    # at first position
    buff[0] = string[0]

    printPatternUtil(string, buff, 1, 1, n)

# Driver program
string = "ABCD"
printPattern(string)

```

This code is contributed by BHAVYA JAIN

Output:

```

ABCD
ABC D
AB CD
AB C D
A BCD
A BC D
A B CD
A B C D

```

Time Complexity: Since number of Gaps are $n-1$, there are total $2^{(n-1)}$ patterns each having length ranging from n to $2n-1$. Thus overall complexity would be $O(n*(2^n))$.

This article is contributed by **Gaurav Sharma**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Strings

Write a program to print all permutations of a given string

A permutation, also called an “arrangement number” or “order,” is a rearrangement of the elements of an ordered list S into a one-to-one correspondence with S itself. A string of length n has $n!$ permutations.
Source: Mathworld(<http://mathworld.wolfram.com/Permutation.html>)

Below are the permutations of string ABC.

ABC ACB BAC BCA CAB

We strongly recommend that you click here and practice it, before moving on to the solution.

Here is a solution that is used as a basis in backtracking.

NewPermutation



C/C++

```

// C program to print all permutations with duplicates allowed
#include <stdio.h>
#include <string.h>

```

```

/* Function to swap values at two pointers */
void swap(char *x, char *y)
{
    char temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

/* Function to print permutations of string
   This function takes three parameters:
   1. String
   2. Starting index of the string
   3. Ending index of the string. */
void permute(char *a, int l, int r)
{
    int i;
    if (l == r)
        printf("%s\n", a);
    else
    {
        for (i = l; i <= r; i++)
        {
            swap((a+l), (a+i));
            permute(a, l+1, r);
            swap((a+l), (a+i)); //backtrack
        }
    }
}

/* Driver program to test above functions */
int main()
{
    char str[] = "ABC";
    int n = strlen(str);
    permute(str, 0, n-1);
    return 0;
}

```

Java

```

// Java program to print all permutations of a
// given string.
public class Permutation
{
    public static void main(String[] args)
    {
        String str = "ABC";
        int n = str.length();
        Permutation permutation = new Permutation();
        permutation.permute(str, 0, n-1);
    }

    /**
     * permutation function
     * @param str string to calculate permutation for
     * @param l starting index
     * @param r end index
     */
    private void permute(String str, int l, int r)
    {
        if (l == r)
            System.out.println(str);
        else
        {
            for (int i = l; i <= r; i++)
            {
                str = swap(str,l,i);
                permute(str, l+1, r);
                str = swap(str,l,i);
            }
        }
    }

    /**
     * Swap Characters at position
     * @param a string value
     * @param i position 1
     * @param j position 2
     * @return swapped string
     */
    public String swap(String a, int i, int j)
    {
        char temp;
        char[] charArray = a.toCharArray();
        temp = charArray[i];
        charArray[i] = charArray[j];
        charArray[j] = temp;
        return String.valueOf(charArray);
    }
}

// This code is contributed by Mihir Joshi

```

Python

```

# Python program to print all permutations with
# duplicates allowed

def toString(List):
    return ''.join(List)

```

```
# Function to print permutations of string
```

```
# This function takes three parameters:
```

```
# 1. String
```

```
# 2. Starting index of the string
```

```
# 3. Ending index of the string.
```

```
def permute(a, l, r):
```

```
    if l==r:
```

```
        print toString(a)
```

```
    else:
```

```
        for i in xrange(l,r+1):
```

```
            a[l], a[i] = a[i], a[l]
```

```
            permute(a, l+1, r)
```

```
            a[l], a[i] = a[i], a[l] # backtrack
```

```
# Driver program to test the above function
```

```
string = "ABC"
```

```
n = len(string)
```

```
a = list(string)
```

```
permute(a, 0, n-1)
```

```
# This code is contributed by Bhavya Jain
```

Output:

```
ABC
```

```
ACB
```

```
BAC
```

```
BCA
```

```
CBA
```

```
CAB
```

Algorithm Paradigm: Backtracking

Time Complexity: $O(n \cdot n!)$ Note that there are $n!$ permutations and it requires $O(n)$ time to print a permutation.

Note : The above solution prints duplicate permutations if there are repeating characters in input string. Please see below link for a solution that prints only distinct permutations even if there are duplicates in input.

[Print all distinct permutations of a given string with duplicates.](#)

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Combinatorial
Strings
Backtracking
MathematicalAlgo
permutation

Backtracking | Set 1 (The Knight's tour problem)

Backtracking is an algorithmic paradigm that tries different solutions until finds a solution that “works”. Problems which are typically solved using backtracking technique have following property in common. These problems can only be solved by trying every possible configuration and each configuration is tried only once. A Naive solution for these problems is to try all configurations and output a configuration that follows given problem constraints. Backtracking works in incremental way and is an optimization over the Naive solution where all possible configurations are generated and tried.

For example, consider the following [Knight's Tour](#) problem.

The knight is placed on the first block of an empty board and, moving according to the rules of chess, must visit each square exactly once.

Path followed by Knight to cover all the cells

knight-tour-problem



Let us first discuss the Naive algorithm for this problem and then the Backtracking algorithm.

Naive Algorithm for Knight's tour

The Naive Algorithm is to generate all tours one by one and check if the generated tour satisfies the constraints.

```
while there are untried tours
{
    generate the next tour
```

```

if this tour covers all squares
{
    print this path;
}

```

Backtracking works in an incremental way to attack problems. Typically, we start from an empty solution vector and one by one add items (Meaning of item varies from problem to problem. In context of Knight's tour problem, an item is a Knight's move). When we add an item, we check if adding the current item violates the problem constraint, if it does then we remove the item and try other alternatives. If none of the alternatives work out then we go to previous stage and remove the item added in the previous stage. If we reach the initial stage back then we say that no solution exists. If adding an item doesn't violate constraints then we recursively add items one by one. If the solution vector becomes complete then we print the solution.

Backtracking Algorithm for Knight's tour

Following is the Backtracking algorithm for Knight's tour problem.

```

If all squares are visited
    print the solution
Else
    a) Add one of the next moves to solution vector and recursively
       check if this move leads to a solution. (A Knight can make maximum
       eight moves. We choose one of the 8 moves in this step).
    b) If the move chosen in the above step doesn't lead to a solution
       then remove this move from the solution vector and try other
       alternative moves.
    c) If none of the alternatives work then return false (Returning false
       will remove the previously added item in recursion and if false is
       returned by the initial call of recursion then "no solution exists")

```

Following are implementations for Knight's tour problem. It prints one of the possible solutions in 2D matrix form. Basically, the output is a 2D 8*8 matrix with numbers from 0 to 63 and these numbers show steps made by Knight.

C

```

// C program for Knight Tour problem
#include<stdio.h>
#define N 8

int solveKTUtil(int x, int y, int movei, int sol[N][N],
                int xMove[], int yMove[]);

/* A utility function to check if i,j are valid indexes
   for N*N chessboard */
bool isSafe(int x, int y, int sol[N][N])
{
    return (x >= 0 && x < N && y >= 0 &&
            y < N && sol[x][y] == -1);
}

/* A utility function to print solution matrix sol[N][N] */
void printSolution(int sol[N][N])
{
    for (int x = 0; x < N; x++)
    {
        for (int y = 0; y < N; y++)
            printf(" %2d ", sol[x][y]);
        printf("\n");
    }
}

/* This function solves the Knight Tour problem using
   Backtracking. This function mainly uses solveKTUtil()
   to solve the problem. It returns false if no complete
   tour is possible, otherwise return true and prints the
   tour.

   Please note that there may be more than one solutions,
   this function prints one of the feasible solutions. */
bool solveKT()
{
    int sol[N][N];

    /* Initialization of solution matrix */
    for (int x = 0; x < N; x++)
        for (int y = 0; y < N; y++)
            sol[x][y] = -1;

    /* xMove[] and yMove[] define next move of Knight.
       xMove[i] is for next value of x coordinate
       yMove[i] is for next value of y coordinate */
    int xMove[8] = { 2, 1, -1, -2, -2, -1, 1, 2 };
    int yMove[8] = { 1, 2, 2, 1, -1, -2, -2, -1 };

    // Since the Knight is initially at the first block
    sol[0][0] = 0;

    /* Start from 0,0 and explore all tours using
       solveKTUtil() */
    if (solveKTUtil(0, 0, 1, sol, xMove, yMove) == false)
    {
        printf("Solution does not exist");
        return false;
    }
    else
        printSolution(sol);

    return true;
}

/* A recursive utility function to solve Knight Tour
   problem */
int solveKTUtil(int x, int y, int movei, int sol[N][N],
                int xMove[N], int yMove[N])
{
    int k, next_x, next_y;
    if (movei == N*N)

```

```

return true;

/* Try all next moves from the current coordinate x, y */
for (k = 0; k < 8; k++) {
{
    next_x = x + xMove[k];
    next_y = y + yMove[k];
    if (isSafe(next_x, next_y, sol)) {
        sol[next_x][next_y] = movei;
        if (solveKTUtil(next_x, next_y, movei+1, sol,
                       xMove, yMove) == true)
            return true;
        else
            sol[next_x][next_y] = -1;//backtracking
    }
}

return false;
}

/* Driver program to test above functions */
int main()
{
    solveKT();
    return 0;
}

```

Java

```

// Java program for Knight Tour problem
class KnightTour {
    static int N = 8;

    /* A utility function to check if i,j are
       valid indexes for N*N chessboard */
    static boolean isSafe(int x, int y, int sol[][]) {
        return (x >= 0 && x < N && y >= 0 &&
               y < N && sol[x][y] == -1);
    }

    /* A utility function to print solution
       matrix sol[N][N] */
    static void printSolution(int sol[][]) {
        for (int x = 0; x < N; x++) {
            for (int y = 0; y < N; y++)
                System.out.print(sol[x][y] + " ");
            System.out.println();
        }
    }

    /* This function solves the Knight Tour problem
       using Backtracking. This function mainly
       uses solveKTUtil() to solve the problem. It
       returns false if no complete tour is possible,
       otherwise return true and prints the tour.
       Please note that there may be more than one
       solutions, this function prints one of the
       feasible solutions. */
    static boolean solveKT() {
        int sol[][] = new int[8][8];

        /* Initialization of solution matrix */
        for (int x = 0; x < N; x++)
            for (int y = 0; y < N; y++)
                sol[x][y] = -1;

        /* xMove[] and yMove[] define next move of Knight.
           xMove[] is for next value of x coordinate
           yMove[] is for next value of y coordinate */
        int xMove[] = {2, 1, -1, -2, -2, -1, 1, 2};
        int yMove[] = {1, 2, 2, 1, -1, -2, -2, -1};

        // Since the Knight is initially at the first block
        sol[0][0] = 0;

        /* Start from 0,0 and explore all tours using
           solveKTUtil() */
        if (!solveKTUtil(0, 0, 1, sol, xMove, yMove)) {
            System.out.println("Solution does not exist");
            return false;
        } else
            printSolution(sol);

        return true;
    }

    /* A recursive utility function to solve Knight
       Tour problem */
    static boolean solveKTUtil(int x, int y, int movei,
                              int sol[][], int xMove[],
                              int yMove[]) {
        int k, next_x, next_y;
        if (movei == N * N)
            return true;

        /* Try all next moves from the current coordinate
           x, y */
        for (k = 0; k < 8; k++) {
            next_x = x + xMove[k];
            next_y = y + yMove[k];
            if (isSafe(next_x, next_y, sol)) {
                sol[next_x][next_y] = movei;
                if (solveKTUtil(next_x, next_y, movei + 1,
                               sol,
                               xMove, yMove) == true)
                    return true;
                else
                    sol[next_x][next_y] = -1;//backtracking
            }
        }

        return false;
    }
}

```

```

        sol, xMove, yMove))
    return true;
else
    sol[next_x][next_y] = -1;// backtracking
}
}

return false;
}

/* Driver program to test above functions */
public static void main(String args[]) {
    solveKT();
}
}

// This code is contributed by Abhishek Shankhadhar

```

Output:

```

0 59 38 33 30 17 8 63
37 34 31 60 9 62 29 16
58 1 36 39 32 27 18 7
35 48 41 26 61 10 15 28
42 57 2 49 40 23 6 19
47 50 45 54 25 20 11 14
56 43 52 3 22 13 24 5
51 46 55 44 53 4 21 12

```

Note that Backtracking is not the best solution for the Knight's tour problem. See [this](#) for other better solutions. The purpose of this post is to explain Backtracking with an example.

References:

<http://see.stanford.edu/materials/icspacs106b/H19-RecBacktrackExamples.pdf>
<http://www.cis.upenn.edu/~matuszek/cit594-2009/Lectures/35-backtracking.ppt>
<http://mathworld.wolfram.com/KnightsTour.html>
http://en.wikipedia.org/wiki/Knight%27s_tour

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Backtracking
Backtracking

Backtracking | Set 2 (Rat in a Maze)

We have discussed Backtracking and Knight's tour problem in [Set 1](#). Let us discuss Rat in a [Maze](#) as another example problem that can be solved using Backtracking.

A Maze is given as N*N binary matrix of blocks where source block is the upper left most block i.e., maze[0][0] and destination block is lower rightmost block i.e., maze[N-1][N-1]. A rat starts from source and has to reach destination. The rat can move only in two directions: forward and down.

In the maze matrix, 0 means the block is dead end and 1 means the block can be used in the path from source to destination. Note that this is a simple version of the typical Maze problem. For example, a more complex version can be that the rat can move in 4 directions and a more complex version can be with limited number of moves.

Following is an example maze.

Gray blocks are dead ends (value = 0).



Following is binary matrix representation of the above maze.

```

{1, 0, 0, 0}
{1, 1, 0, 1}
{0, 1, 0, 0}
{1, 1, 1, 1}

```

Following is maze with highlighted solution path.



Following is the solution matrix (output of program) for the above input matrix.

```

{1, 0, 0, 0}
{1, 1, 0, 0}
{0, 1, 0, 0}
{0, 1, 1, 1}

```

All entries in solution path are marked as 1.

Naive Algorithm

The Naive Algorithm is to generate all paths from source to destination and one by one check if the generated path satisfies the constraints.

```

while there are untried paths
{
    generate the next path
    if this path has all blocks as 1
    {

```

```

    print this path;
}
}

```

Backtracking Algorithm

```

If destination is reached
    print the solution matrix
Else
    a) Mark current cell in solution matrix as 1.
    b) Move forward in horizontal direction and recursively check if this
       move leads to a solution.
    c) If the move chosen in the above step doesn't lead to a solution
       then move down and check if this move leads to a solution.
    d) If none of the above solutions work then unmark this cell as 0
       (BACKTRACK) and return false.

```

Implementation of Backtracking solution

C/C++

```

/* C/C++ program to solve Rat in a Maze problem using
   backtracking */
#include<stdio.h>

// Maze size
#define N 4

bool solveMazeUtil(int maze[N][N], int x, int y, int sol[N][N]);

/* A utility function to print solution matrix sol[N][N] */
void printSolution(int sol[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            printf(" %d ", sol[i][j]);
        printf("\n");
    }
}

/* A utility function to check if x,y is valid index for N*N maze */
bool isSafe(int maze[N][N], int x, int y)
{
    // If (x,y) outside maze) return false
    if(x >= 0 && x < N && y >= 0 && y < N && maze[x][y] == 1)
        return true;

    return false;
}

/* This function solves the Maze problem using Backtracking. It mainly
   uses solveMazeUtil() to solve the problem. It returns false if no
   path is possible, otherwise return true and prints the path in the
   form of 1s. Please note that there may be more than one solutions,
   this function prints one of the feasible solutions.*/
bool solveMaze(int maze[N][N])
{
    int sol[N][N] = { { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 } };

    if(solveMazeUtil(maze, 0, 0, sol) == false)
    {
        printf("Solution doesn't exist");
        return false;
    }

    printSolution(sol);
    return true;
}

/* A recursive utility function to solve Maze problem */
bool solveMazeUtil(int maze[N][N], int x, int y, int sol[N][N])
{
    // If (x,y) is goal) return true
    if(x == N-1 && y == N-1)
    {
        sol[x][y] = 1;
        return true;
    }

    // Check if maze[x][y] is valid
    if(isSafe(maze, x, y) == true)
    {
        // mark x,y as part of solution path
        sol[x][y] = 1;

        /* Move forward in x direction */
        if (solveMazeUtil(maze, x+1, y, sol) == true)
            return true;

        /* If moving in x direction doesn't give solution then
           Move down in y direction */
        if (solveMazeUtil(maze, x, y+1, sol) == true)
            return true;

        /* If none of the above movements work then BACKTRACK:
           unmark x,y as part of solution path */
        sol[x][y] = 0;
        return false;
    }
}

```

```

    return false;
}

// driver program to test above function
int main()
{
    int maze[N][N] = { {1, 0, 0, 0},
                       {1, 1, 0, 1},
                       {0, 1, 0, 0},
                       {1, 1, 1, 1}
                     };

    solveMaze(maze);
    return 0;
}

```

Java

```

/* Java program to solve Rat in a Maze problem using
backtracking */

public class RatMaze
{
    final int N = 4;

    /* A utility function to print solution matrix
     * sol[N][N] */
    void printSolution(int sol[][])
    {
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
                System.out.print(" " + sol[i][j] +
                               " ");
            System.out.println();
        }
    }

    /* A utility function to check if x,y is valid
     * index for N*N maze */
    boolean isSafe(int maze[][], int x, int y)
    {
        // if (x,y outside maze) return false
        return (x >= 0 && x < N && y >= 0 &&
               y < N && maze[x][y] == 1);
    }

    /* This function solves the Maze problem using
     * Backtracking. It mainly uses solveMazeUtil()
     * to solve the problem. It returns false if no
     * path is possible, otherwise return true and
     * prints the path in the form of 1s. Please note
     * that there may be more than one solutions, this
     * function prints one of the feasible solutions.*/
    boolean solveMaze(int maze[][])
    {
        int sol[][] = {{0, 0, 0, 0},
                      {0, 0, 0, 0},
                      {0, 0, 0, 0},
                      {0, 0, 0, 0}
                    };

        if (solveMazeUtil(maze, 0, 0, sol) == false)
        {
            System.out.print("Solution doesn't exist");
            return false;
        }

        printSolution(sol);
        return true;
    }

    /* A recursive utility function to solve Maze
     * problem */
    boolean solveMazeUtil(int maze[][], int x, int y,
                          int sol[][])
    {
        // if (x,y is goal) return true
        if (x == N - 1 && y == N - 1)
        {
            sol[x][y] = 1;
            return true;
        }

        // Check if maze[x][y] is valid
        if (isSafe(maze, x, y) == true)
        {
            // mark x,y as part of solution path
            sol[x][y] = 1;

            /* Move forward in x direction */
            if (solveMazeUtil(maze, x + 1, y, sol))
                return true;

            /* If moving in x direction doesn't give
             * solution then Move down in y direction */
            if (solveMazeUtil(maze, x, y + 1, sol))
                return true;

            /* If none of the above movements work then
             * BACKTRACK: unmark x,y as part of solution
             * path */
            sol[x][y] = 0;
        }
    }
}

```

```

        return false;
    }

    return false;
}

public static void main(String args[])
{
    RatMaze rat = new RatMaze();
    int maze[][], {{1, 0, 0, 0},
    {1, 1, 0, 1},
    {0, 1, 0, 0},
    {1, 1, 1, 1}
    };
    rat.solveMaze(maze);
}
}

// This code is contributed by Abhishek Shankhadhar

```

Output: The 1 values show the path for rat

```

1 0 0 0
1 1 0 0
0 1 0 0
0 1 1 1

```

Below is an extended version of this problem. [Count number of ways to reach destination in a Maze](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Backtracking
Backtracking

Backtracking | Set 3 (N Queen Problem)

We have discussed Knight's tour and Rat in a Maze problems in [Set 1](#) and [Set 2](#) respectively. Let us discuss N Queen as another example problem that can be solved using Backtracking.

The N Queen is the problem of placing N chess queens on an NxN chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.



The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example following is the output matrix for above 4 queen solution.

```

{ 0, 1, 0, 0}
{ 0, 0, 0, 1}
{ 1, 0, 0, 0}
{ 0, 0, 1, 0}

```

Naive Algorithm

Generate all possible configurations of queens on board and print a configuration that satisfies the given constraints.

```

while there are untried configurations
{
    generate the next configuration
    if queens don't attack in this configuration then
    {
        print this configuration;
    }
}

```

Backtracking Algorithm

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes then we backtrack and return false.

- 1) Start in the leftmost column
- 2) If all queens are placed
 - return true
- 3) Try all rows in the current column. Do following for every tried row.
 - a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
 - b) If placing queen in [row, column] leads to a solution then return true.
 - c) If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.
- 3) If all rows have been tried and nothing worked, return false to trigger backtracking.

Implementation of Backtracking solution

C/C++

```

/* C/C++ program to solve N Queen Problem using
backtracking */
#define N 4
#include<stdio.h>

/* A utility function to print solution */
void printSolution(int board[N][N])

```

```

{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            printf("%d ", board[i][j]);
        printf("\n");
    }
}

/* A utility function to check if a queen can
be placed on board[row][col]. Note that this
function is called when "col" queens are
already placed in columns from 0 to col -1.
So we need to check only left side for
attacking queens */
bool isSafe(int board[N][N], int row, int col)
{
    int i, j;

    /* Check this row on left side */
    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;

    /* Check upper diagonal on left side */
    for (i=row, j=col; i>=0 && j>=0; i--, j--)
        if (board[i][j])
            return false;

    /* Check lower diagonal on left side */
    for (i=row, j=col; j>=0 && i<N; i++, j--)
        if (board[i][j])
            return false;

    return true;
}

/* A recursive utility function to solve N
Queen problem */
bool solveNQUtil(int board[N][N], int col)
{
    /* base case: If all queens are placed
    then return true */
    if (col >= N)
        return true;

    /* Consider this column and try placing
    this queen in all rows one by one */
    for (int i = 0; i < N; i++)
    {
        /* Check if queen can be placed on
        board[i][col] */
        if (isSafe(board, i, col) )
        {
            /* Place this queen in board[i][col] */
            board[i][col] = 1;

            /* recur to place rest of the queens */
            if (solveNQUtil(board, col + 1) )
                return true;

            /* If placing queen in board[i][col]
            doesn't lead to a solution, then
            remove queen from board[i][col] */
            board[i][col] = 0; // BACKTRACK
        }
    }

    /* If queen can not be placed in any row in
    this column col then return false */
    return false;
}

/* This function solves the N Queen problem using
Backtracking. It mainly uses solveNQUtil() to
solve the problem. It returns false if queens
cannot be placed, otherwise return true and
prints placement of queens in the form of 1s.
Please note that there may be more than one
solutions, this function prints one of the
feasible solutions.*/
bool solveNQ()
{
    int board[N][N] = { { 0, 0, 0, 0 },
                       { 0, 0, 0, 0 },
                       { 0, 0, 0, 0 },
                       { 0, 0, 0, 0 } };

    if ( solveNQUtil(board, 0) == false )
    {
        printf("Solution does not exist");
        return false;
    }

    printSolution(board);
    return true;
}

// driver program to test above function
int main()
{
    solveNQ();
    return 0;
}

```

Java

```
/* Java program to solve N Queen Problem using
backtracking */
public class NQueenProblem
{
    final int N = 4;

    /* A utility function to print solution */
    void printSolution(int board[][])
    {
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
                System.out.print(" " + board[i][j])
                + " ");
            System.out.println();
        }
    }

    /* A utility function to check if a queen can
    be placed on board[row][col]. Note that this
    function is called when "col" queens are already
    placed in columns from 0 to col -1. So we need
    to check only left side for attacking queens */
    boolean isSafe(int board[][], int row, int col)
    {
        int i, j;

        /* Check this row on left side */
        for (i = 0; i < col; i++)
            if (board[row][i] == 1)
                return false;

        /* Check upper diagonal on left side */
        for (i=row, j=col; i>=0 && j>=0; i--, j--)
            if (board[i][j] == 1)
                return false;

        /* Check lower diagonal on left side */
        for (i=row, j=col; i<N && j<N; i++, j++)
            if (board[i][j] == 1)
                return false;

        return true;
    }

    /* A recursive utility function to solve N
    Queen problem */
    boolean solveNQUtil(int board[][], int col)
    {
        /* base case: If all queens are placed
         then return true */
        if (col >= N)
            return true;

        /* Consider this column and try placing
        this queen in all rows one by one */
        for (int i = 0; i < N; i++)
        {
            /* Check if queen can be placed on
            board[i][col] */
            if (isSafe(board, i, col))
            {
                /* Place this queen in board[i][col] */
                board[i][col] = 1;

                /* recur to place rest of the queens */
                if (solveNQUtil(board, col + 1) == true)
                    return true;

                /* If placing queen in board[i][col]
                doesn't lead to a solution then
                remove queen from board[i][col] */
                board[i][col] = 0; // BACKTRACK
            }
        }

        /* If queen can not be placed in any row in
        this column col, then return false */
        return false;
    }

    /* This function solves the N Queen problem using
    Backtracking. It mainly uses solveNQUtil() to
    solve the problem. It returns false if queens
    cannot be placed, otherwise return true and
    prints placement of queens in the form of 1s.
    Please note that there may be more than one
    solutions, this function prints one of the
    feasible solutions.*/
    boolean solveNQ()
    {
        int board[][] = {{0, 0, 0, 0},
                        {0, 0, 0, 0},
                        {0, 0, 0, 0},
                        {0, 0, 0, 0}};

        if (solveNQUtil(board, 0) == false)
        {
            System.out.print("Solution does not exist");
            return false;
        }
    }
}
```

```

    printSolution(board);
    return true;
}

// driver program to test above function
public static void main(String args[])
{
    NQueenProblem Queen = new NQueenProblem();
    Queen.solveNQ();
}
}

// This code is contributed by Abhishek Shankhadhar

```

Output: The 1 values indicate placements of queens

```

0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

```

Printing all solutions in N-Queen Problem

Sources:

<http://see.stanford.edu/materials/icspacs106b/H19-RecBacktrackExamples.pdf>
http://en.literateprograms.org/Eight_queens_puzzle_%28C%29
http://en.wikipedia.org/wiki/Eight_queens_puzzle

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Backtracking
Backtracking

Backtracking | Set 4 (Subset Sum)

Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K. We are considering the set contains non-negative values. It is assumed that the input set is unique (no duplicates are presented).

We strongly recommend that you click here and practice it, before moving on to the solution.

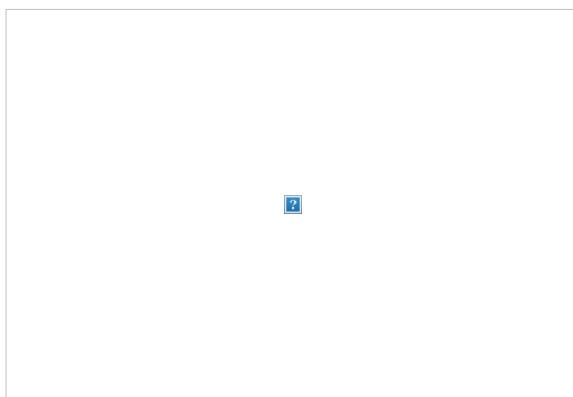
Exhaustive Search Algorithm for Subset Sum

One way to find subsets that sum to K is to consider all possible subsets. A **power set** contains all those subsets generated from a given set. The size of such a power set is 2^N .

Backtracking Algorithm for Subset Sum

Using exhaustive search we consider all subsets irrespective of whether they satisfy given constraints or not. Backtracking can be used to make a systematic consideration of the elements to be selected.

Assume given set of 4 elements, say **w[1] ... w[4]**. Tree diagrams can be used to design backtracking algorithms. The following tree diagram depicts approach of generating variable sized tuple.



In the above tree, a node represents function call and a branch represents candidate element. The root node contains 4 children. In other words, root considers every element of the set as different branch. The next level sub-trees correspond to the subsets that includes the parent node. The branches at each level represent tuple element to be considered. For example, if we are at level 1, `tuple_vector[1]` can take any value of four branches generated. If we are at level 2 of left most node, `tuple_vector[2]` can take any value of three branches generated, and so on...

For example the left most child of root generates all those subsets that include `w[1]`. Similarly the second child of root generates all those subsets that includes `w[2]` and excludes `w[1]`.

As we go down along depth of tree we add elements so far, and if the added sum is satisfying explicit constraints, we will continue to generate child nodes further. Whenever the constraints are not met, we stop further generation of sub-trees of that node, and backtrack to previous node to explore the nodes not yet explored. In many scenarios, it saves considerable amount of processing time.

The tree should trigger a clue to implement the backtracking algorithm (try yourself). It prints all those subsets whose sum add up to given number. We need to explore the nodes along the breadth and depth of the tree. Generating nodes along breadth is controlled by loop and nodes along the depth are generated using recursion (post order traversal). Pseudo code given below.

```

if(subset is satisfying the constraint)
    print the subset
    exclude the current element and consider next element
else
    generate the nodes of present level along breadth of tree and
    recur for next levels

```

Following is C implementation of subset sum using variable size tuple vector. Note that the following program explores all possibilities similar to exhaustive search. It is to demonstrate how backtracking can be used. See next code to verify, how we can optimize the backtracking solution.

```

#include <stdio.h>
#include <stdlib.h>

#define ARRAYSIZE(a) (sizeof(a))/(sizeof(a[0]))

static int total_nodes;
// prints subset found

```

```

void printSubset(int A[], int size)
{
    for(int i = 0; i < size; i++)
    {
        printf("%*d", 5, A[i]);
    }

    printf("\n");
}

// inputs
// s      - set vector
// t      - tuplet vector
// s_size - set size
// t_size - tuplet size so far
// sum   - sum so far
// ite   - nodes count
// target_sum - sum to be found
void subset_sum(int s[], int t[],
                int s_size, int t_size,
                int sum, int ite,
                int const target_sum)
{
    total_nodes++;
    if(target_sum == sum )
    {
        // We found subset
        printSubset(t, t_size);
        // Exclude previously added item and consider next candidate
        subset_sum(s, t, s_size, t_size + 1, sum - s[ite], ite + 1, target_sum);
        return;
    }
    else
    {
        // generate nodes along the breadth
        for( int i = ite; i < s_size; i++ )
        {
            t[t_size] = s[i];
            // consider next level node (along depth)
            subset_sum(s, t, s_size, t_size + 1, sum + s[i], i + 1, target_sum);
        }
    }
}

// Wrapper to print subsets that sum to target_sum
// input is weights vector and target_sum
void generateSubsets(int s[], int size, int target_sum)
{
    int *tuplet_vector = (int *)malloc(size * sizeof(int));

    subset_sum(s, tuplet_vector, size, 0, 0, target_sum);

    free(tuplet_vector);
}

int main()
{
    int weights[] = {10, 7, 5, 18, 12, 20, 15};
    int size = ARRSIZE(weights);

    generateSubsets(weights, size, 35);
    printf("Nodes generated %d\n", total_nodes);
    return 0;
}

```

The power of backtracking appears when we combine explicit and implicit constraints, and we stop generating nodes when these checks fail. We can improve the above algorithm by strengthening the constraint checks and presorting the data. By sorting the initial array, we need not to consider rest of the array, once the sum so far is greater than target number. We can backtrack and check other possibilities.

Similarly, assume the array is presorted and we found one subset. We can generate next node excluding the present node only when inclusion of next node satisfies the constraints. Given below is optimized implementation (it prunes the subtree if it is not satisfying contraints).

```

#include <stdio.h>
#include <stdlib.h>

#define ARRSIZE(a) (sizeof(a))/(sizeof(a[0]))

static int total_nodes;

// prints subset found
void printSubset(int A[], int size)
{
    for(int i = 0; i < size; i++)
    {
        printf("%*d", 5, A[i]);
    }

    printf("\n");
}

// qsort compare function
int comparator(const void *pLhs, const void *pRhs)
{
    int *lhs = (int *)pLhs;
    int *rhs = (int *)pRhs;

    return *lhs > *rhs;
}

// inputs
// s      - set vector
// t      - tuplet vector
// s_size - set size
// t_size - tuplet size so far
// sum   - sum so far
// ite   - nodes count

```

```

// target_sum - sum to be found
void subset_sum(int s[], int t[],
               int s_size, int t_size,
               int sum, int ite,
               int const target_sum)
{
    total_nodes++;

    if( target_sum == sum )
    {
        // We found sum
        printSubset(t, t_size);

        // constraint check
        if( ite + 1 < s_size && sum - s[ite] + s[ite+1] <= target_sum )
        {
            // Exclude previous added item and consider next candidate
            subset_sum(s, t, s_size, t_size-1, sum - s[ite], ite + 1, target_sum);
        }
        return;
    }
    else
    {
        // constraint check
        if( ite < s_size && sum + s[ite] <= target_sum )
        {
            // generate nodes along the breadth
            for( int i = ite; i < s_size; i++ )
            {
                t[t_size] = s[i];

                if( sum + s[i] <= target_sum )
                {
                    // consider next level node (along depth)
                    subset_sum(s, t, s_size + 1, sum + s[i], i + 1, target_sum);
                }
            }
        }
    }
}

// Wrapper that prints subsets that sum to target_sum
void generateSubsets(int s[], int size, int target_sum)
{
    int *tuplet_vector = (int *)malloc(size * sizeof(int));

    int total = 0;

    // sort the set
    qsort(s, size, sizeof(int), &comparator);

    for( int i = 0; i < size; i++ )
    {
        total += s[i];
    }

    if( s[0] <= target_sum && total >= target_sum )
    {
        subset_sum(s, tuplet_vector, size, 0, 0, 0, target_sum);
    }

    free(tuplet_vector);
}

int main()
{
    int weights[] = {15, 22, 14, 26, 32, 9, 16, 8};
    int target = 53;

    int size = ARYSIZE(weights);

    generateSubsets(weights, size, target);

    printf("Nodes generated %d\n", total_nodes);

    return 0;
}

```

As another approach, we can generate the tree in fixed size tuple analogs to binary pattern. We will kill the sub-trees when the constraints are not satisfied.

-- **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

[Backtracking](#)
[Adobe Question](#)
[Bloomberg](#)
About Venki
Software Engineer
View all posts by Venki →

Backtracking | Set 5 (m Coloring Problem)

Given an undirected graph and a number m, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with same color. Here coloring of a graph means assignment of colors to all vertices.

Input:

- 1) A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.
- 2) An integer m which is maximum number of colors that can be used.

Output:

An array color[V] that should have numbers from 1 to m. color[i] should represent the color assigned to the ith vertex. The code should also return false if the graph cannot be colored with m colors.

Following is an example graph (from [Wiki page](#)) that can be colored with 3 colors.



We strongly recommend that you click [here](#) and practice it, before moving on to the solution.

Naive Algorithm

Generate all possible configurations of colors and print a configuration that satisfies the given constraints.

```
while there are untried conflagrations
{
    generate the next configuration
    if no adjacent vertices are colored with same color
    {
        print this configuration;
    }
}
```

There will be V^m configurations of colors.

Backtracking Algorithm

The idea is to assign colors one by one to different vertices, starting from the vertex 0. Before assigning a color, we check for safety by considering already assigned colors to the adjacent vertices. If we find a color assignment which is safe, we mark the color assignment as part of solution. If we do not find color due to clashes then we backtrack and return false.

Implementation of Backtracking solution

C/C++

```
#include<stdio.h>

// Number of vertices in the graph
#define V 4

void printSolution(int color[]);

/* A utility function to check if the current color assignment
   is safe for vertex v */
bool isSafe (int v, bool graph[V][V], int color[], int c)
{
    for (int i = 0; i < V; i++)
        if (graph[v][i] && c == color[i])
            return false;
    return true;
}

/* A recursive utility function to solve m coloring problem */
bool graphColoringUtil(bool graph[V][V], int m, int color[], int v)
{
    /* base case: If all vertices are assigned a color then
       return true */
    if (v == V)
        return true;

    /* Consider this vertex v and try different colors */
    for (int c = 1; c <= m; c++)
    {
        /* Check if assignment of color c to v is fine*/
        if (isSafe(v, graph, color, c))
        {
            color[v] = c;

            /* recur to assign colors to rest of the vertices */
            if (graphColoringUtil (graph, m, color, v+1) == true)
                return true;

            /* If assigning color c doesn't lead to a solution
               then remove it */
            color[v] = 0;
        }
    }

    /* If no color can be assigned to this vertex then return false;
       return false; */
}

/* This function solves the m Coloring problem using Backtracking.
It mainly uses graphColoringUtil() to solve the problem. It returns
false if the m colors cannot be assigned, otherwise return true and
prints assignments of colors to all vertices. Please note that there
may be more than one solutions, this function prints one of the
feasible solutions.*/
bool graphColoring(bool graph[V][V], int m)
{
    // Initialize all color values as 0. This initialization is needed
    // correct functioning of isSafe()
    int *color = new int[V];
    for (int i = 0; i < V; i++)
        color[i] = 0;

    // Call graphColoringUtil() for vertex 0
    if (graphColoringUtil(graph, m, color, 0) == false)
    {
        printf("Solution does not exist");
    }
}
```

```

        return false;
    }

    // Print the solution
    printSolution(color);
    return true;
}

/* A utility function to print solution */
void printSolution(int color[])
{
    printf("Solution Exists:\n");
    " Following are the assigned colors \n";
    for (int i = 0; i < V; i++)
        printf("%d ", color[i]);
    printf("\n");
}

// driver program to test above function
int main()
{
    /* Create following graph and test whether it is 3 colorable
     (3)---(2)
     | / |
     | / |
     | / |
     (0)---(1)
    */
    bool graph[V][V] = {{0, 1, 1, 1},
                        {1, 0, 1, 0},
                        {1, 1, 0, 1},
                        {1, 0, 1, 0},
    };
    int m = 3; // Number of colors
    graphColoring (graph, m);
    return 0;
}

```

Java

```

/* Java program for solution of M Coloring problem
using backtracking */
public class mColoringProblem {
    final int V = 4;
    int color[];

    /* A utility function to check if the current
    color assignment is safe for vertex v */
    boolean isSafe(int v, int graph[][], int color[],
                  int c)
    {
        for (int i = 0; i < V; i++)
            if (graph[v][i] == 1 && c == color[i])
                return false;
        return true;
    }

    /* A recursive utility function to solve m
    coloring problem */
    boolean graphColoringUtil(int graph[][], int m,
                               int color[], int v)
    {
        /* base case: If all vertices are assigned
        a color then return true */
        if (v == V)
            return true;

        /* Consider this vertex v and try different
        colors */
        for (int c = 1; c <= m; c++)
        {
            /* Check if assignment of color c to v
            is fine*/
            if (isSafe(v, graph, color, c))
            {
                color[v] = c;

                /* recur to assign colors to rest
                of the vertices */
                if (graphColoringUtil(graph, m,
                                      color, v + 1))
                    return true;

                /* If assigning color c doesn't lead
                to a solution then remove it */
                color[v] = 0;
            }
        }

        /* If no color can be assigned to this vertex
        then return false */
        return false;
    }

    /* This function solves the m Coloring problem using
    Backtracking. It mainly uses graphColoringUtil()
    to solve the problem. It returns false if the m
    colors cannot be assigned, otherwise return true
    and prints assignments of colors to all vertices.
    Please note that there may be more than one
    solutions, this function prints one of the
    feasible solutions.*/
    boolean graphColoring(int graph[][], int m)
    {

```

```

// Initialize all color values as 0. This
// initialization is needed correct functioning
// of isSafe()
color = new int[V];
for (int i = 0; i < V; i++)
    color[i] = 0;

// Call graphColoringUtil() for vertex 0
if (!graphColoringUtil(graph, m, color, 0))
{
    System.out.println("Solution does not exist");
    return false;
}

// Print the solution
printSolution(color);
return true;
}

/* A utility function to print solution */
void printSolution(int color[])
{
    System.out.println("Solution Exists: Following" +
        " are the assigned colors");
    for (int i = 0; i < V; i++)
        System.out.print(" " + color[i] + " ");
    System.out.println();
}

// driver program to test above function
public static void main(String args[])
{
    mColoringProblem Coloring = new mColoringProblem();
    /* Create following graph and test whether it is
    3 colorable
    (3)---(2)
    | / |
    | / |
    | / |
    (0)---(1)
    */
    int graph[][] = {{0, 1, 1, 1},
                    {1, 0, 1, 0},
                    {1, 1, 0, 1},
                    {1, 0, 1, 0},
                    {1, 0, 1, 0}};
    int m = 3; // Number of colors
    Coloring.graphColoring(graph, m);
}
}
// This code is contributed by Abhishek Shankhadhar

```

Output:

```

Solution Exists: Following are the assigned colors
1 2 3 2

```

References:

http://en.wikipedia.org/wiki/Graph_coloring

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Backtracking
Backtracking

Backtracking | Set 6 (Hamiltonian Cycle)

Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then print the path. Following are the input and output of the required function.

Input:

A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.

Output:

An array path[V] that should contain the Hamiltonian Path. path[i] should represent the ith vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.

For example, a Hamiltonian Cycle in the following graph is {0, 1, 2, 4, 3, 0}. There are more Hamiltonian Cycles in the graph like {0, 3, 4, 2, 1, 0}

```

(0)---(1)---(2)
| / \ |
| / \ |
| / \ |
(3)-----(4)

```

And the following graph doesn't contain any Hamiltonian Cycle.

```

(0)---(1)---(2)
| / \ |
| / \ |
| / \ |
(3)---(4)

```

Naive Algorithm

Generate all possible configurations of vertices and print a configuration that satisfies the given constraints. There will be $n!$ (n factorial) configurations.

```

while there are untried configurations
{

```

```

generate the next configuration
if ( there are edges between two consecutive vertices of this
    configuration and there is an edge from the last vertex to
    the first).
{
    print this configuration;
    break;
}

```

Backtracking Algorithm

Create an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return false.

Implementation of Backtracking solution

Following are implementations of the Backtracking solution.

C/C++

```

/* C/C++ program for solution of Hamiltonian Cycle problem
   using backtracking */
#include<stdio.h>

// Number of vertices in the graph
#define V 5

void printSolution(int path[]);

/* A utility function to check if the vertex v can be added at
   index 'pos' in the Hamiltonian Cycle constructed so far (stored
   in 'path[]') */
bool isSafe(int v, bool graph[V][V], int path[], int pos)
{
    /* Check if this vertex is an adjacent vertex of the previously
       added vertex. */
    if (graph [ path[pos-1] ][ v ] == 0)
        return false;

    /* Check if the vertex has already been included.
       This step can be optimized by creating an array of size V */
    for (int i = 0; i < pos; i++)
        if (path[i] == v)
            return false;

    return true;
}

/* A recursive utility function to solve hamiltonian cycle problem */
bool hamCycleUtil(bool graph[V][V], int path[], int pos)
{
    /* base case: If all vertices are included in Hamiltonian Cycle */
    if (pos == V)
    {
        // And if there is an edge from the last included vertex to the
        // first vertex
        if (graph[ path[pos-1] ][ path[0] ] == 1 )
            return true;
        else
            return false;
    }

    /* Try different vertices as a next candidate in Hamiltonian Cycle.
       We don't try for 0 as we included 0 as starting point in hamCycle()
       for (int v = 1; v < V; v++)
    {
        /* Check if this vertex can be added to Hamiltonian Cycle */
        if (isSafe(v, graph, path, pos))
        {
            path[pos] = v;

            /* recur to construct rest of the path */
            if (hamCycleUtil (graph, path, pos+1) == true)
                return true;

            /* If adding vertex v doesn't lead to a solution,
               then remove it */
            path[pos] = -1;
        }
    }

    /* If no vertex can be added to Hamiltonian Cycle constructed so far,
       then return false */
    return false;
}

/* This function solves the Hamiltonian Cycle problem using Backtracking.
   It mainly uses hamCycleUtil() to solve the problem. It returns false
   if there is no Hamiltonian Cycle possible, otherwise return true and
   prints the path. Please note that there may be more than one solutions,
   this function prints one of the feasible solutions. */
bool hamCycle(bool graph[V][V])
{
    int *path = new int[V];
    for (int i = 0; i < V; i++)
        path[i] = -1;

    /* Let us put vertex 0 as the first vertex in the path. If there is
       a Hamiltonian Cycle, then the path can be started from any point
       of the cycle as the graph is undirected */
    path[0] = 0;
    if ( hamCycleUtil(graph, path, 1) == false )
    {
        printf("\nSolution does not exist");
        return false;
    }
}

```

```

    }

    printSolution(path);
    return true;
}

/* A utility function to print solution */
void printSolution(int path[])
{
    printf("Solution Exists:\n");
    " Following is one Hamiltonian Cycle \n");
    for (int i = 0; i < V; i++)
        printf(" %d ", path[i]);

    // Let us print the first vertex again to show the complete cycle
    printf(" %d ", path[0]);
    printf("\n");
}

// driver program to test above function
int main()
{
    /* Let us create the following graph
     (0)--(1)--(2)
     |   / |
     |  /  |
     | /   |
     (3)-----(4) */

    bool graph1[V][V]={{0, 1, 0, 1, 0},
                       {1, 0, 1, 1, 1},
                       {0, 1, 0, 0, 1},
                       {1, 1, 0, 0, 1},
                       {0, 1, 1, 1, 0},
                       };

    // Print the solution
    hamCycle(graph1);

    /* Let us create the following graph
     (0)--(1)--(2)
     |   / |
     |  /  |
     | /   |
     (3)---(4) */

    bool graph2[V][V]={{0, 1, 0, 1, 0},
                       {1, 0, 1, 1, 1},
                       {0, 1, 0, 0, 1},
                       {1, 1, 0, 0, 0},
                       {0, 1, 1, 0, 0},
                       };

    // Print the solution
    hamCycle(graph2);

    return 0;
}

```

Java

```

/* Java program for solution of Hamiltonian Cycle problem
using backtracking */
class HamiltonianCycle
{
    final int V = 5;
    int path[];

    /* A utility function to check if the vertex v can be
     added at index 'pos' in the Hamiltonian Cycle
     constructed so far (stored in 'path[]') */
    boolean isSafe(int v, int graph[][], int path[], int pos)
    {
        /* Check if this vertex is an adjacent vertex of
         the previously added vertex. */
        if (graph[path[pos - 1]][v] == 0)
            return false;

        /* Check if the vertex has already been included.
         This step can be optimized by creating an array
         of size V */
        for (int i = 0; i < pos; i++)
            if (path[i] == v)
                return false;

        return true;
    }

    /* A recursive utility function to solve hamiltonian
     cycle problem */
    boolean hamCycleUtil(int graph[][], int path[], int pos)
    {
        /* base case: If all vertices are included in
         Hamiltonian Cycle */
        if (pos == V)
        {
            // And if there is an edge from the last included
            // vertex to the first vertex
            if (graph[path[pos - 1]][path[0]] == 1)
                return true;
            else
                return false;
        }

        // Try different vertices as a next candidate in
        // Hamiltonian Cycle. We don't try 0 as we

```

```

// included 0 as starting point in hamCycle()
for (int v = 1; v < V; v++)
{
    /* Check if this vertex can be added to Hamiltonian
     Cycle */
    if (isSafe(v, graph, path, pos))
    {
        path[pos] = v;

        /* recur to construct rest of the path */
        if (hamCycleUtil(graph, path, pos + 1) == true)
            return true;

        /* If adding vertex v doesn't lead to a solution,
         then remove it */
        path[pos] = -1;
    }
}

/* If no vertex can be added to Hamiltonian Cycle
 constructed so far, then return false */
return false;
}

/* This function solves the Hamiltonian Cycle problem using
 Backtracking. It mainly uses hamCycleUtil() to solve the
 problem. It returns false if there is no Hamiltonian Cycle
 possible, otherwise return true and prints the path.
 Please note that there may be more than one solutions,
 this function prints one of the feasible solutions.*/
int hamCycle(int graph[][])
{
    path = new int[V];
    for (int i = 0; i < V; i++)
        path[i] = -1;

    /* Let us put vertex 0 as the first vertex in the path.
     If there is a Hamiltonian Cycle, then the path can be
     started from any point of the cycle as the graph is
     undirected */
    path[0] = 0;
    if (hamCycleUtil(graph, path, 1) == false)
    {
        System.out.println("nSolution does not exist");
        return 0;
    }

    printSolution(path);
    return 1;
}

/* A utility function to print solution */
void printSolution(int path[])
{
    System.out.println("Solution Exists: Following" +
        " is one Hamiltonian Cycle");
    for (int i = 0; i < V; i++)
        System.out.print(" " + path[i] + " ");

    // Let us print the first vertex again to show the
    // complete cycle
    System.out.println(" " + path[0] + " ");
}

// driver program to test above function
public static void main(String args[])
{
    HamiltonianCycle hamiltonian =
        new HamiltonianCycle();
    /* Let us create the following graph
     (0)--(1)--(2)
     | / \ |
     | /   \ |
     |/     \|
     (3)-----(4) */
    int graph1[][] = {{0, 1, 0, 1, 0},
                    {1, 0, 1, 1, 1},
                    {0, 1, 0, 0, 1},
                    {1, 1, 0, 0, 1},
                    {0, 1, 1, 1, 0},
                    };

    // Print the solution
    hamiltonian.hamCycle(graph1);

    /* Let us create the following graph
     (0)--(1)--(2)
     | / \ |
     | /   \ |
     |/     \|
     (3)---(4) */
    int graph2[][] = {{0, 1, 0, 1, 0},
                    {1, 0, 1, 1, 1},
                    {0, 1, 0, 0, 1},
                    {1, 1, 0, 0, 0},
                    {0, 1, 1, 0, 0},
                    };

    // Print the solution
    hamiltonian.hamCycle(graph2);
}
}

// This code is contributed by Abhishek Shankhadhar

```

Output:

Solution Exists: Following is one Hamiltonian Cycle
0 1 2 4 3 0

Solution does not exist

Note that the above code always prints cycle starting from 0. Starting point should not matter as cycle can be started from any point. If you want to change the starting point, you should make two changes to above code.

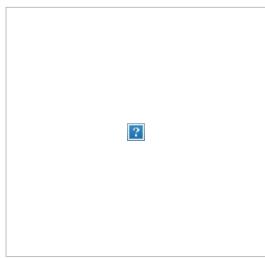
Change "path[0] = 0;" to "path[0] = s;" where s is your new starting point. Also change loop "for (int v = 1; v <

GATE CS Corner Company Wise Coding Practice

Graph
Backtracking
Graph

Backtracking | Set 7 (Sudoku)

Given a partially filled 9×9 2D array 'grid[9][9]', the goal is to assign digits (from 1 to 9) to the empty cells so that every row, column, and subgrid of size 3×3 contains exactly one instance of the digits from 1 to 9.



Naive Algorithm

The Naive Algorithm is to generate all possible configurations of numbers from 1 to 9 to fill the empty cells. Try every configuration one by one until the correct configuration is found.

Backtracking Algorithm

Like all other Backtracking problems, we can solve Sudoku by one by one assigning numbers to empty cells. Before assigning a number, we check whether it is safe to assign. We basically check that the same number is not present in current row, current column and current 3×3 subgrid. After checking for safety, we assign the number, and recursively check whether this assignment leads to a solution or not. If the assignment doesn't lead to a solution, then we try next number for current empty cell. And if none of number (1 to 9) lead to solution, we return false.

```
Find row, col of an unassigned cell
If there is none, return true
For digits from 1 to 9
    a) If there is no conflict for digit at row,col
        assign digit to row,col and recursively try fill in rest of grid
    b) If recursion successful, return true
    c) Else, remove digit and try another
If all digits have been tried and nothing worked, return false
```

Following are C++ and Python implementation for Sudoku problem. It prints the completely filled grid as output.

C/C++

```
// A Backtracking program in C++ to solve Sudoku problem
#include <stdio.h>

// UNASSIGNED is used for empty cells in sudoku grid
#define UNASSIGNED 0

// N is used for size of Sudoku grid. Size will be NxN
#define N 9

// This function finds an entry in grid that is still unassigned
bool FindUnassignedLocation(int grid[N][N], int &row, int &col);

// Checks whether it will be legal to assign num to the given row,col
bool isSafe(int grid[N][N], int row, int col, int num);

/* Takes a partially filled-in grid and attempts to assign values to
   all unassigned locations in such a way to meet the requirements
   for Sudoku solution (non-duplication across rows, columns, and boxes) */
bool SolveSudoku(int grid[N][N])
{
    int row, col;

    // If there is no unassigned location, we are done
    if (!FindUnassignedLocation(grid, row, col))
        return true; // success!

    // consider digits 1 to 9
    for (int num = 1; num <= 9; num++)
    {
        // if looks promising
        if (isSafe(grid, row, col, num))
        {
            // make tentative assignment
            grid[row][col] = num;

            // return, if success, yay!
            if (SolveSudoku(grid))
                return true;

            // failure, unmake & try again
            grid[row][col] = UNASSIGNED;
        }
    }
    return false; // this triggers backtracking
}
```

```

/* Searches the grid to find an entry that is still unassigned. If
   found, the reference parameters row, col will be set the location
   that is unassigned, and true is returned. If no unassigned entries
   remain, false is returned. */
bool FindUnassignedLocation(int grid[N][N], int &row, int &col)
{
    for (row = 0; row < N; row++)
        for (col = 0; col < N; col++)
            if (grid[row][col] == UNASSIGNED)
                return true;
    return false;
}

/* Returns a boolean which indicates whether any assigned entry
   in the specified row matches the given number. */
bool UsedInRow(int grid[N][N], int row, int num)
{
    for (int col = 0; col < N; col++)
        if (grid[row][col] == num)
            return true;
    return false;
}

/* Returns a boolean which indicates whether any assigned entry
   in the specified column matches the given number. */
bool UsedInCol(int grid[N][N], int col, int num)
{
    for (int row = 0; row < N; row++)
        if (grid[row][col] == num)
            return true;
    return false;
}

/* Returns a boolean which indicates whether any assigned entry
   within the specified 3x3 box matches the given number. */
bool UsedInBox(int grid[N][N], int boxStartRow, int boxStartCol, int num)
{
    for (int row = 0; row < 3; row++)
        for (int col = 0; col < 3; col++)
            if (grid[row+boxStartRow][col+boxStartCol] == num)
                return true;
    return false;
}

/* Returns a boolean which indicates whether it will be legal to assign
   num to the given row,col location. */
bool isSafe(int grid[N][N], int row, int col, int num)
{
    /* Check if 'num' is not already placed in current row,
       current column and current 3x3 box */
    return !UsedInRow(grid, row, num) &&
           !UsedInCol(grid, col, num) &&
           !UsedInBox(grid, row - row%3 , col - col%3, num);
}

/* A utility function to print grid */
void printGrid(int grid[N][N])
{
    for (int row = 0; row < N; row++)
    {
        for (int col = 0; col < N; col++)
            printf("%2d", grid[row][col]);
        printf("\n");
    }
}

/* Driver Program to test above functions */
int main()
{
    // 0 means unassigned cells
    int grid[N][N] = {{3, 0, 6, 5, 0, 8, 4, 0, 0},
                      {5, 2, 0, 0, 0, 0, 0, 0, 0},
                      {0, 8, 7, 0, 0, 0, 3, 1},
                      {0, 0, 3, 0, 1, 0, 0, 8, 0},
                      {9, 0, 0, 8, 6, 3, 0, 0, 5},
                      {0, 5, 0, 0, 9, 0, 6, 0, 0},
                      {1, 3, 0, 0, 0, 0, 2, 5, 0},
                      {0, 0, 0, 0, 0, 0, 0, 7, 4},
                      {0, 0, 5, 2, 0, 6, 3, 0, 0}};
    if (SolveSudoku(grid) == true)
        printGrid(grid);
    else
        printf("No solution exists");

    return 0;
}

```

Python

```

# A Backtracking program in Python to solve Sudoku problem

# A Utility Function to print the Grid
def print_grid(arr):
    for i in range(9):
        for j in range(9):
            print arr[i][j],
            print (" ")
    print

# Function to Find the entry in the Grid that is still not used
# Searches the grid to find an entry that is still unassigned. If
# found, the reference parameters row, col will be set the location
# that is unassigned, and true is returned. If no unassigned entries
# found, false is returned
# This function takes a 2D array representing the grid
# It returns True if a solution is found
# It returns False otherwise
# It also prints the solution grid to console
def solve_sudoku(grid):
    # Find an empty cell
    for i in range(9):
        for j in range(9):
            if grid[i][j] == 0:
                # Try numbers from 1 to 9
                for num in range(1, 10):
                    if is_valid(grid, i, j, num):
                        # If valid, place the number
                        grid[i][j] = num
                        # Recur
                        if solve_sudoku(grid):
                            return True
                        # If not valid or doesn't lead to a solution, backtrack
                        grid[i][j] = 0
                return False
    return True

```

```

# remain, false is returned.
# 'l' is a list variable that has been passed from the solve_sudoku function
# to keep track of incrementation of Rows and Columns
def find_empty_location(arr,l):
    for row in range(9):
        for col in range(9):
            if(arr[row][col]==0):
                l[0]=row
                l[1]=col
                return True
    return False

# Returns a boolean which indicates whether any assigned entry
# in the specified row matches the given number.
def used_in_row(arr,row,num):
    for i in range(9):
        if(arr[row][i] == num):
            return True
    return False

# Returns a boolean which indicates whether any assigned entry
# in the specified column matches the given number.
def used_in_col(arr,col,num):
    for i in range(9):
        if(arr[i][col] == num):
            return True
    return False

# Returns a boolean which indicates whether any assigned entry
# within the specified 3x3 box matches the given number
def used_in_box(arr,row,col,num):
    for i in range(3):
        for j in range(3):
            if(arr[i+row][j+col] == num):
                return True
    return False

# Checks whether it will be legal to assign num to the given row,col
# Returns a boolean which indicates whether it will be legal to assign
# num to the given row,col location.
def check_location_is_safe(arr,row,col,num):

    # Check if 'num' is not already placed in current row,
    # current column and current 3x3 box
    return not used_in_row(arr,row,num) and not used_in_col(arr,col,num) and not used_in_box(arr,row - row%3,col - col%3,num)

# Takes a partially filled-in grid and attempts to assign values to
# all unassigned locations in such a way to meet the requirements
# for Sudoku solution (non-duplication across rows, columns, and boxes)
def solve_sudoku(arr):

    # 'l' is a list variable that keeps the record of row and col in find_empty_location Function
    l=[0,0]

    # If there is no unassigned location, we are done
    if(not find_empty_location(arr,l)):
        return True

    # Assigning list values to row and col that we got from the above Function
    row=l[0]
    col=l[1]

    # consider digits 1 to 9
    for num in range(1,10):

        # if looks promising
        if(check_location_is_safe(arr,row,col,num)):

            # make tentative assignment
            arr[row][col]=num

            # return, if success, ya!
            if(solve_sudoku(arr)):
                return True

            # failure, unmake & try again
            arr[row][col]=0

    # this triggers backtracking
    return False

# Driver main function to test above functions
if __name__=="__main__":
    # creating a 2D array for the grid
    grid=[[0 for x in range(9)]for y in range(9)]

    # assigning values to the grid
    grid=[[3,0,6,5,0,8,4,0,0],
          [5,2,0,0,0,0,0,0,0],
          [0,8,7,0,0,0,0,3,1],
          [0,0,3,0,1,0,0,8,0],
          [9,0,0,8,6,3,0,0,5],
          [0,5,0,0,9,0,6,0,0],
          [1,3,0,0,0,0,2,5,0],
          [0,0,0,0,0,0,7,4],
          [0,0,5,2,0,6,3,0,0]]

    # if sucess print the grid
    if(solve_sudoku(grid)):
        print_grid(grid)
    else:
        print "No solution exists"

# The above code has been contributed by Harshit Sidhwa.

```

Output:

```
3 1 6 5 7 8 4 9 2  
5 2 9 1 3 4 7 6 8  
4 8 7 6 2 9 5 3 1  
2 6 3 4 1 5 9 8 7  
9 7 4 8 6 3 1 2 5  
8 5 1 7 9 2 6 4 3  
1 3 8 9 4 7 2 5 6  
6 9 2 3 5 1 8 7 4  
7 4 5 2 8 6 3 1 9
```

References:

<http://see.stanford.edu/materials/icspacs106b/H19-RecBacktrackExamples.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Backtracking
Backtracking

Tug of War

Given a set of n integers, divide the set in two subsets of $n/2$ sizes each such that the difference of the sum of two subsets is as minimum as possible. If n is even, then sizes of two subsets must be strictly $n/2$ and if n is odd, then size of one subset must be $(n-1)/2$ and size of other subset must be $(n+1)/2$.

For example, let given set be {3, 4, 5, -3, 100, 1, 89, 54, 23, 20}, the size of set is 10. Output for this set should be {4, 100, 1, 23, 20} and {3, 5, -3, 89, 54}. Both output subsets are of size 5 and sum of elements in both subsets is same (148 and 148).

Let us consider another example where n is odd. Let given set be {23, 45, -34, 12, 0, 98, -99, 4, 189, -1, 4}. The output subsets should be {45, -34, 12, 98, -1} and {23, 0, -99, 4, 189, 4}. The sums of elements in two subsets are 120 and 121 respectively.

The following solution tries every possible subset of half size. If one subset of half size is formed, the remaining elements form the other subset. We initialize current set as empty and one by one build it. There are two possibilities for every element, either it is part of current set, or it is part of the remaining elements (other subset). We consider both possibilities for every element. When the size of current set becomes $n/2$, we check whether this solution is better than the best solution available so far. If it is, then we update the best solution.

Following is C++ implementation for Tug of War problem. It prints the required arrays.

```
#include <iostream>  
#include <stdlib.h>  
#include <limits.h>  
using namespace std;  
  
// function that tries every possible solution by calling itself recursively  
void TOWUtil(int* arr, int n, bool* curr_elements, int no_of_selected_elements,  
             bool* soln, int* min_diff, int sum, int curr_sum, int curr_position)  
{  
    // checks whether the it is going out of bound  
    if (curr_position == n)  
        return;  
  
    // checks that the numbers of elements left are not less than the  
    // number of elements required to form the solution  
    if ((n/2 - no_of_selected_elements) > (n - curr_position))  
        return;  
  
    // consider the cases when current element is not included in the solution  
    TOWUtil(arr, n, curr_elements, no_of_selected_elements,  
            soln, min_diff, sum, curr_sum, curr_position+1);  
  
    // add the current element to the solution  
    no_of_selected_elements++;  
    curr_sum = curr_sum + arr[curr_position];  
    curr_elements[curr_position] = true;  
  
    // checks if a solution is formed  
    if (no_of_selected_elements == n/2)  
    {  
        // checks if the solution formed is better than the best solution so far  
        if (abs(sum/2 - curr_sum) < *min_diff)  
        {  
            *min_diff = abs(sum/2 - curr_sum);  
            for (int i = 0; i < n; i++)  
                soln[i] = curr_elements[i];  
        }  
    }  
    else  
    {  
        // consider the cases where current element is included in the solution  
        TOWUtil(arr, n, curr_elements, no_of_selected_elements, soln,  
                min_diff, sum, curr_sum, curr_position+1);  
    }  
  
    // removes current element before returning to the caller of this function  
    curr_elements[curr_position] = false;  
}  
  
// main function that generate an arr  
void tugOfWar(int *arr, int n)  
{  
    // the boolean array that contains the inclusion and exclusion of an element  
    // in current set. The number excluded automatically form the other set  
    bool* curr_elements = new bool[n];  
  
    // The inclusion/exclusion array for final solution  
    bool* soln = new bool[n];  
  
    int min_diff = INT_MAX;  
  
    int sum = 0;  
    for (int i=0; i<n; i++)  
    {  
        sum += arr[i];  
    }
```

```

curr_elements[i] = soln[i] = false;
}

// Find the solution using recursive function TOWUtil()
TOWUtil(arr, n, curr_elements, 0, soln, &min_diff, sum, 0, 0);

// Print the solution
cout << "The first subset is:";
for (int i=0; i<n; i++)
{
    if (soln[i] == true)
        cout << arr[i] << " ";
}
cout << "\nThe second subset is:";
for (int i=0; i<n; i++)
{
    if (soln[i] == false)
        cout << arr[i] << " ";
}
}

// Driver program to test above functions
int main()
{
    int arr[] = {23, 45, -34, 12, 0, 98, -99, 4, 189, -1, 4};
    int n = sizeof(arr)/sizeof(arr[0]);
    tugOfWar(arr, n);
    return 0;
}

```

Output:

```

The first subset is: 45 -34 12 98 -1
The second subset is: 23 0 -99 4 189 4

```

This article is compiled by [Ashish Anand](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Backtracking
Backtracking

Backtracking | Set 8 (Solving Cryptarithmic Puzzles)

Newspapers and magazines often have crypt-arithmetic puzzles of the form:

```

SEND
+ MORE
-----
MONEY
-----

```

The goal here is to assign each letter a digit from 0 to 9 so that the arithmetic works out correctly. The rules are that all occurrences of a letter must be assigned the same digit, and no digit can be assigned to more than one letter.

- First, create a list of all the characters that need assigning to pass to Solve
 - If all characters are assigned, return true if puzzle is solved, false otherwise
 - Otherwise, consider the first unassigned character
 - for (every possible choice among the digits not in use)
- make that choice and then recursively try to assign the rest of the characters
- if recursion successful, return true
- if !successful, unmake assignment and try another digit
- If all digits have been tried and nothing worked, return false to trigger backtracking

```

/* ExhaustiveSolve
 * -----
 * This is the "not-very-smart" version of cryptarithmetic solver. It takes
 * the puzzle itself (with the 3 strings for the two addends and sum) and a
 * string of letters as yet unassigned. If no more letters to assign
 * then we've hit a base-case, if the current letter-to-digit mapping solves
 * the puzzle, we're done, otherwise we return false to trigger backtracking
 * If we have letters to assign, we take the first letter from that list, and
 * try assigning it the digits from 0 to 9 and then recursively working
 * through solving puzzle from here. If we manage to make a good assignment
 * that works, we've succeeded, else we need to unassign that choice and try
 * another digit. This version is easy to write, since it uses a simple
 * approach (quite similar to permutations if you think about it) but it is
 * not so smart because it doesn't take into account the structure of the
 * puzzle constraints (for example, once the two digits for the addends have
 * been assigned, there is no reason to try anything other than the correct
 * digit for the sum) yet it tries a lot of useless combos regardless
 */
bool ExhaustiveSolve(puzzleT puzzle, string lettersToAssign)
{
    if (lettersToAssign.empty()) // no more choices to make
        return PuzzleSolved(puzzle); // checks arithmetic to see if works
    for (int digit = 0; digit <= 9; digit++) // try all digits
    {
        if (AssignLetterToDigit(lettersToAssign[0], digit))
        {
            if (ExhaustiveSolve(puzzle, lettersToAssign.substr(1)))
                return true;
            UnassignLetterFromDigit(lettersToAssign[0], digit);
        }
    }
    return false; // nothing worked, need to backtrack
}

```

The algorithm above actually has a lot in common with the permutations algorithm, it pretty much just creates all arrangements of the mapping from characters to digits and tries each until one works or all have been successfully tried. For a large puzzle, this could take a while.

A smarter algorithm could take into account the structure of the puzzle and avoid going down dead-end paths. For example, if we assign the characters starting from the ones place and moving to the left, at each stage, we can verify the correctness of what we have so far before we continue onwards. This definitely complicates the code but leads to a tremendous improvement in efficiency, making it much more feasible to solve large puzzles.

Below pseudocode in this case has more special cases, but the same general design

- Start by examining the rightmost digit of the topmost row, with a carry of 0
- If we are beyond the leftmost digit of the puzzle, return true if no carry, false otherwise
- If we are currently trying to assign a char in one of the addends
 - If char already assigned, just recur on row beneath this one, adding value into sum
 - If not assigned, then
 - for (every possible choice among the digits not in use) make that choice and then on row beneath this one, if successful, return true
 - if !successful, unmake assignment and try another digit
 - return false if no assignment worked to trigger backtracking
- Else if trying to assign a char in the sum
 - If char assigned & matches correct, recur on next column to the left with carry, if success return true,
 - If char assigned & doesn't match, return false
 - If char unassigned & correct digit already used, return false
 - If char unassigned & correct digit unused, assign it and recur on next column to left with carry, if success return true
 - return false to trigger backtracking

Source:

<http://see.stanford.edu/materials/icspacs106b/H19-RecBacktrackExamples.pdf>

GATE CS Corner Company Wise Coding Practice

Backtracking
Backtracking

Divide and Conquer | Set 1 (Introduction)

Like Greedy and Dynamic Programming, Divide and Conquer is an algorithmic paradigm. A typical Divide and Conquer algorithm solves a problem using following three steps.

1. **Divide:** Break the given problem into subproblems of same type.
2. **Conquer:** Recursively solve these subproblems
3. **Combine:** Appropriately combine the answers

Following are some standard algorithms that are Divide and Conquer algorithms.

1) **Binary Search** is a searching algorithm. In each step, the algorithm compares the input element x with the value of the middle element in array. If the values match, return the index of middle. Otherwise, if x is less than the middle element, then the algorithm recurs for left side of middle element, else recurs for right side of middle element.

2) **Quicksort** is a sorting algorithm. The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side. Finally, the algorithm recursively sorts the subarrays on left and right of pivot element.

3) **Merge Sort** is also a sorting algorithm. The algorithm divides the array in two halves, recursively sorts them and finally merges the two sorted halves.

4) **Closest Pair of Points** The problem is to find the closest pair of points in a set of points in $x-y$ plane. The problem can be solved in $O(n^2)$ time by calculating distances of every pair of points and comparing the distances to find the minimum. The Divide and Conquer algorithm solves the problem in $O(n \log n)$ time.

5) **Strassen's Algorithm** is an efficient algorithm to multiply two matrices. A simple method to multiply two matrices need 3 nested loops and is $O(n^3)$. Strassen's algorithm multiplies two matrices in $O(n^{2.8974})$ time.

6) **Cooley-Tukey Fast Fourier Transform (FFT) algorithm** is the most common algorithm for FFT. It is a divide and conquer algorithm which works in $O(n \log n)$ time.

7) **Karatsuba algorithm for fast multiplication** it does multiplication of two n -digit numbers in at most $\boxed{\quad}$ single-digit multiplications in general (and exactly $\boxed{\quad}$ when n is a power of 2). It is therefore faster than the classical algorithm, which requires n^2 single-digit products. If $n = 2^{10} = 1024$, in particular, the exact counts are $3^{10} = 59,049$ and $(2^{10})^2 = 1,048,576$, respectively.

We will publishing above algorithms in separate posts.

Divide and Conquer (D & C) vs Dynamic Programming (DP)

Both paradigms (D & C and DP) divide the given problem into subproblems and solve subproblems. How to choose one of them for a given problem? Divide and Conquer should be used when same subproblems are not evaluated many times. Otherwise Dynamic Programming or Memoization should be used. For example, Binary Search is a Divide and Conquer algorithm, we never evaluate the same subproblems again. On the other hand, for calculating nth Fibonacci number, Dynamic Programming should be preferred (See [this](#) for details).

References

Algorithms by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani

Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

http://en.wikipedia.org/wiki/Karatsuba_algorithm

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Divide and Conquer
Divide and Conquer

Write a program to calculate pow(x,n)

Below solution divides the problem into subproblems of size $y/2$ and call the subproblems recursively.

```
#include<stdio.h>

/* Function to calculate x raised to the power y */
int power(int x, unsigned int y)
{
    if( y == 0)
        return 1;
    else if (y%2 == 0)
        return power(x, y/2)*power(x, y/2);
```

```

    else
        return x*power(x, y/2)*power(x, y/2);

}

/* Program to test function power */
int main()
{
    int x = 2;
    unsigned int y = 3;

    printf("%d", power(x, y));
    getchar();
    return 0;
}

```

Time Complexity: O(n)

Space Complexity: O(1)

Algorithmic Paradigm: Divide and conquer.

Above function can be optimized to O(logn) by calculating power(x, y/2) only once and storing it.

```

/* Function to calculate x raised to the power y in O(logn) */
int power(int x, unsigned int y)
{
    int temp;
    if(y == 0)
        return 1;
    temp = power(x, y/2);
    if(y%2 == 0)
        return temp*temp;
    else
        return x*temp*temp;
}

```

Time Complexity of optimized solution: O(logn)

Let us extend the pow function to work for negative y and float x.

```

/* Extended version of power function that can work
for float x and negative y*/
#include<stdio.h>

float power(float x, int y)
{
    float temp;
    if(y == 0)
        return 1;
    temp = power(x, y/2);
    if(y%2 == 0)
        return temp*temp;
    else
    {
        if(y > 0)
            return x*temp*temp;
        else
            return (temp*temp)/x;
    }
}

/* Program to test function power */
int main()
{
    float x = 2;
    int y = -3;
    printf("%f", power(x, y));
    getchar();
    return 0;
}

```

GATE CS Corner Company Wise Coding Practice

Divide and Conquer
Divide and Conquer

Median of two sorted arrays

Question: There are 2 sorted arrays A and B of size n each. Write an algorithm to find the median of the array obtained after merging the above 2 arrays(i.e. array of length 2n). The complexity should be O(log(n))

We strongly recommend that you click here and practice it, before moving on to the solution.

Median: In probability theory and statistics, a median is described as the number separating the higher half of a sample, a population, or a probability distribution, from the lower half. The median of a finite list of numbers can be found by arranging all the numbers from lowest value to highest value and picking the middle one.

For getting the median of input array { 12, 11, 15, 10, 20 }, first sort the array. We get { 10, 11, 12, 15, 20 } after sorting. Median is the middle element of the sorted array which is 12.

There are different conventions to take median of an array with even number of elements, one can take the mean of the two middle values, or first middle value, or second middle value.

Let us see different methods to get the median of two sorted arrays of size n each. Since size of the set for which we are looking for median is even (2n), we are taking average of middle two numbers in all below solutions.

Method 1 (Simply count while Merging)

Use merge procedure of merge sort. Keep track of count while comparing elements of two arrays. If count becomes n (For 2n elements), we have reached the median. Take the average of the elements at indexes n-1 and n in the merged array. See the below implementation.

Implementation:

```
// A Simple Merge based O(n) solution to find median of
```

```

// two sorted arrays
#include <stdio.h>

/* This function returns median of ar1[] and ar2[].
Assumptions in this function:
Both ar1[] and ar2[] are sorted arrays
Both have n elements */
int getMedian(int ar1[], int ar2[], int n)
{
    int i = 0; /* Current index of i/p array ar1[] */
    int j = 0; /* Current index of i/p array ar2[] */
    int count;
    int m1 = -1, m2 = -1;

    /* Since there are 2n elements, median will be average
    of elements at index n-1 and n in the array obtained after
    merging ar1 and ar2 */
    for (count = 0; count <= n; count++)
    {
        /*Below is to handle case where all elements of ar1[] are
        smaller than smallest(or first) element of ar2[]*/
        if (i == n)
        {
            m1 = m2;
            m2 = ar2[0];
            break;
        }

        /*Below is to handle case where all elements of ar2[] are
        smaller than smallest(or first) element of ar1[]*/
        else if (j == n)
        {
            m1 = m2;
            m2 = ar1[0];
            break;
        }

        if (ar1[i] < ar2[j])
        {
            m1 = m2; /* Store the prev median */
            m2 = ar1[i];
            i++;
        }
        else
        {
            m1 = m2; /* Store the prev median */
            m2 = ar2[j];
            j++;
        }
    }

    return (m1 + m2)/2;
}

/* Driver program to test above function */
int main()
{
    int ar1[] = {1, 12, 15, 26, 38};
    int ar2[] = {2, 13, 17, 30, 45};

    int n1 = sizeof(ar1)/sizeof(ar1[0]);
    int n2 = sizeof(ar2)/sizeof(ar2[0]);
    if (n1 == n2)
        printf("Median is %d", getMedian(ar1, ar2, n1));
    else
        printf("Doesn't work for arrays of unequal size");
    getchar();
    return 0;
}

```

Output

Median is 16

Time Complexity: O(n)

Method 2 (By comparing the medians of two arrays)

This method works by first getting medians of the two sorted arrays and then comparing them.

Let ar1 and ar2 be the input arrays.

Algorithm:

- 1) Calculate the medians m1 and m2 of the input arrays ar1[] and ar2[] respectively.
- 2) If m1 and m2 both are equal then we are done.
return m1 (or m2)
- 3) If m1 is greater than m2, then median is present in one of the below two subarrays.
 - a) From first element of ar1 to m1 (ar1[0..._n/2_...])
 - b) From m2 to last element of ar2 (ar2[_n/2_...n-1])
- 4) If m2 is greater than m1, then median is present in one of the below two subarrays.
 - a) From m1 to last element of ar1 (ar1[_n/2_...n-1])
 - b) From first element of ar2 to m2 (ar2[0..._n/2_...])
- 5) Repeat the above process until size of both the subarrays becomes 2.
- 6) If size of the two arrays is 2 then use below formula to get the median.
Median = (max(ar1[0], ar2[0]) + min(ar1[1], ar2[1]))/2

Example:

```
ar1[] = {1, 12, 15, 26, 38}  
ar2[] = {2, 13, 17, 30, 45}
```

For above two arrays m1 = 15 and m2 = 17

For the above ar1[] and ar2[], m1 is smaller than m2. So median is present in one of the following two subarrays.

```
[15, 26, 38] and [2, 13, 17]
```

Let us repeat the process for above two subarrays:

```
m1 = 26 m2 = 13.
```

m1 is greater than m2. So the subarrays become

```
[15, 26] and [13, 17]  
Now size is 2, so median = (max(ar1[0], ar2[0]) + min(ar1[1], ar2[1]))/2  
= (max(15, 13) + min(26, 17))/2  
= (15 + 17)/2  
= 16
```

Implementation:

```
// A divide and conquer based efficient solution to find median  
// of two sorted arrays of same size.  
#include<bits/stdc++.h>  
using namespace std;  
  
int median(int [], int); /* to get median of a sorted array */  
  
/* This function returns median of ar1[] and ar2[].  
Assumptions in this function:  
Both ar1[] and ar2[] are sorted arrays  
Both have n elements */  
int getMedian(int ar1[], int ar2[], int n)  
{  
    /* return -1 for invalid input */  
    if (n <= 0)  
        return -1;  
    if (n == 1)  
        return (ar1[0] + ar2[0])/2;  
    if (n == 2)  
        return (max(ar1[0], ar2[0]) + min(ar1[1], ar2[1])) / 2;  
  
    int m1 = median(ar1, n); /* get the median of the first array */  
    int m2 = median(ar2, n); /* get the median of the second array */  
  
    /* If medians are equal then return either m1 or m2 */  
    if (m1 == m2)  
        return m1;  
  
    /* if m1 < m2 then median must exist in ar1[m1...] and  
    ar2[...m2] */  
    if (m1 < m2)  
    {  
        if (n % 2 == 0)  
            return getMedian(ar1 + n/2 - 1, ar2, n - n/2 + 1);  
        return getMedian(ar1 + n/2, ar2, n - n/2);  
    }  
  
    /* if m1 > m2 then median must exist in ar1[...m1] and  
    ar2[m2...] */  
    if (n % 2 == 0)  
        return getMedian(ar2 + n/2 - 1, ar1, n - n/2 + 1);  
    return getMedian(ar2 + n/2, ar1, n - n/2);  
}  
  
/* Function to get median of a sorted array */  
int median(int arr[], int n)  
{  
    if (n%2 == 0)  
        return (arr[n/2] + arr[n/2-1])/2;  
    else  
        return arr[n/2];  
}  
  
/* Driver program to test above function */  
int main()  
{  
    int ar1[] = {1, 2, 3, 6};  
    int ar2[] = {4, 6, 8, 10};  
    int n1 = sizeof(ar1)/sizeof(ar1[0]);  
    int n2 = sizeof(ar2)/sizeof(ar2[0]);  
    if (n1 == n2)  
        printf("Median is %d", getMedian(ar1, ar2, n1));  
    else  
        printf("Doesn't work for arrays of unequal size");  
    return 0;  
}
```

Output :

```
Median is 5
```

Time Complexity: O(logn)

Algorithmic Paradigm: Divide and Conquer

Median of two sorted arrays of different sizes

References:

<http://en.wikipedia.org/wiki/Median>

<http://ocw.alfaisal.edu/NR/rdonlyres/Electrical-Engineering-and-Computer-Science/6-046JFall-2005/30C68118-E436-4FE3-8C79-6BAFBB07D935/0/ps9sol.pdf> ds3etph5wn

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Divide and Conquer
Searching
Divide and Conquer

Count Inversions in an array | Set 1 (Using Merge Sort)

Inversion Count for an array indicates – how far (or close) the array is from being sorted. If array is already sorted then inversion count is 0. If array is sorted in reverse order that inversion count is the maximum.

Formally speaking, two elements $a[i]$ and $a[j]$ form an inversion if $a[i] > a[j]$ and $i < j$.

Example:

The sequence 2, 4, 1, 3, 5 has three inversions (2, 1), (4, 1), (4, 3).

METHOD 1 (Simple)

For each element, count number of elements which are on right side of it and are smaller than it.

```
#include <bits/stdc++.h>
int getInvCount(int arr[], int n)
{
    int inv_count = 0;
    for (int i = 0; i < n - 1; i++)
        for (int j = i+1; j < n; j++)
            if (arr[i] > arr[j])
                inv_count++;

    return inv_count;
}

/* Driver program to test above functions */
int main(int argc, char** args)
{
    int arr[] = {1, 20, 6, 4, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Number of inversions are %d \n", getInvCount(arr, n));
    return 0;
}
```

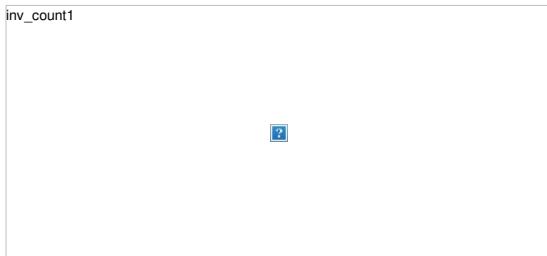
Output:

```
Number of inversions are 5
```

Time Complexity: $O(n^2)$

METHOD 2(Enhance Merge Sort)

Suppose we know the number of inversions in the left half and right half of the array (let be $inv1$ and $inv2$), what kinds of inversions are not accounted for in $inv1 + inv2$? The answer is – the inversions we have to count during the merge step. Therefore, to get number of inversions, we need to add number of inversions in left subarray, right subarray and merge().



How to get number of inversions in merge()?

In merge process, let i is used for indexing left sub-array and j for right sub-array. At any step in merge(), if $a[i]$ is greater than $a[j]$, then there are $(mid - i)$ inversions. because left and right subarrays are sorted, so all the remaining elements in left-subarray ($a[i+1], a[i+2] \dots a[mid]$) will be greater than $a[j]$.



The complete picture:

**Implementation:**

```
#include <bits/stdc++.h>

int _mergeSort(int arr[], int temp[], int left, int right);
int merge(int arr[], int temp[], int left, int mid, int right);

/* This function sorts the input array and returns the
   number of inversions in the array */
int mergeSort(int arr[], int array_size)
{
    int *temp = (int *)malloc(sizeof(int)*array_size);
    return _mergeSort(arr, temp, 0, array_size - 1);
}

/* An auxiliary recursive function that sorts the input array and
   returns the number of inversions in the array. */
int _mergeSort(int arr[], int temp[], int left, int right)
{
    int mid, inv_count = 0;
    if (right > left)
    {
        /* Divide the array into two parts and call _mergeSortAndCountInv()
           for each of the parts */
        mid = (right + left)/2;

        /* Inversion count will be sum of inversions in left-part, right-part
           and number of inversions in merging */
        inv_count = _mergeSort(arr, temp, left, mid);
        inv_count += _mergeSort(arr, temp, mid+1, right);

        /*Merge the two parts*/
        inv_count += merge(arr, temp, left, mid+1, right);
    }
    return inv_count;
}

/* This funt merges two sorted arrays and returns inversion count in
   the arrays.*/
int merge(int arr[], int temp[], int left, int mid, int right)
{
    int i, j, k;
    int inv_count = 0;

    i = left; /* i is index for left subarray*/
    j = mid; /* j is index for right subarray*/
    k = left; /* k is index for resultant merged subarray*/
    while ((i <= mid - 1) && (j <= right))
    {
        if (arr[i] <= arr[j])
        {
            temp[k++] = arr[i++];
        }
        else
        {
            temp[k++] = arr[j++];
            /*this is tricky -- see above explanation/diagram for merge()*/
            inv_count = inv_count + (mid - i);
        }
    }

    /* Copy the remaining elements of left subarray
       (if there are any) to temp*/
    while (i <= mid - 1)
        temp[k++] = arr[i++];

    /* Copy the remaining elements of right subarray
       (if there are any) to temp*/
    while (j <= right)
        temp[k++] = arr[j++];

    /*Copy back the merged elements to original array*/
    for (i=left; i <= right; i++)
        arr[i] = temp[i];

    return inv_count;
}

/* Driver program to test above functions */
int main(int argc, char** args)
{
    int arr[] = {1, 20, 6, 4, 5};
}
```

```

printf(" Number of inversions are %d \n", mergeSort(arr, 5));
getchar();
return 0;
}

```

Output:

```
Number of inversions are 5
```

Note that above code modifies (or sorts) the input array. If we want to count only inversions then we need to create a copy of original array and call mergeSort() on copy.

Time Complexity: O(nlogn)

Algorithmic Paradigm: Divide and Conquer

You may like to see.

[Count inversions in an array | Set 2 \(Using Self-Balancing BST\)](#)

[Counting Inversions using Set in C++ STL](#)

[Count inversions in an array | Set 3 \(Using BIT\)](#)

References:

<http://www.cs.umd.edu/class/fall2009/cmsc451/lectures/Lec08-inversions.pdf>

<http://www.cp.eng.chula.ac.th/~piak/teaching/algo/algo2008/count-inv.htm>

Please write comments if you find any bug in the above program/algorithm or other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Divide and Conquer
Sorting
Divide and Conquer
inversion
Merge Sort

Divide and Conquer | Set 2 (Closest Pair of Points)

We are given an array of n points in the plane, and the problem is to find out the closest pair of points in the array. This problem arises in a number of applications. For example, in air-traffic control, you may want to monitor planes that come too close together, since this may indicate a possible collision. Recall the following formula for distance between two points p and q.



The Brute force solution is $O(n^2)$, compute the distance between each pair and return the smallest. We can calculate the smallest distance in $O(n\log n)$ time using Divide and Conquer strategy. In this post, a $O(n \times (\log n)^2)$ approach is discussed. We will be discussing a $O(n\log n)$ approach in a separate post.

Algorithm

Following are the detailed steps of a $O(n (\log n)^2)$ algorithm.

Input: An array of n points $P[]$

Output: The smallest distance between two points in the given array.

As a pre-processing step, input array is sorted according to x coordinates.

1) Find the middle point in the sorted array, we can take $P[n/2]$ as middle point.

2) Divide the given array in two halves. The first subarray contains points from $P[0]$ to $P[n/2]$. The second subarray contains points from $P[n/2+1]$ to $P[n-1]$.

3) Recursively find the smallest distances in both subarrays. Let the distances be d_l and d_r . Find the minimum of d_l and d_r . Let the minimum be d .



4) From above 3 steps, we have an upper bound d of minimum distance. Now we need to consider the pairs such that one point in pair is from left half and other is from right half. Consider the vertical line passing through $P[n/2]$ and find all points whose x coordinate is closer than d to the middle vertical line. Build an array $strip[]$ of all such points.



5) Sort the array $strip[]$ according to y coordinates. This step is $O(n\log n)$. It can be optimized to $O(n)$ by recursively sorting and merging.

6) Find the smallest distance in $strip[]$. This is tricky. From first look, it seems to be a $O(n^2)$ step, but it is actually $O(n)$. It can be proved geometrically that for every point in $strip$, we only need to check at most 7 points after it (note that $strip$ is sorted according to Y coordinate). See [this](#) for more analysis.

7) Finally return the minimum of d and distance calculated in above step (step 6)

Implementation

Following is C/C++ implementation of the above algorithm.

```

// A divide and conquer program in C/C++ to find the smallest distance from a
// given set of points.

#include <stdio.h>
#include <float.h>
#include <stdlib.h>
#include <math.h>

```

```

// A structure to represent a Point in 2D plane
struct Point
{
    int x, y;
};

/* Following two functions are needed for library function qsort().
   Refer: http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ */

// Needed to sort array of points according to X coordinate
int compareX(const void* a, const void* b)
{
    Point *p1 = (Point *)a, *p2 = (Point *)b;
    return (p1->x - p2->x);
}

// Needed to sort array of points according to Y coordinate
int compareY(const void* a, const void* b)
{
    Point *p1 = (Point *)a, *p2 = (Point *)b;
    return (p1->y - p2->y);
}

// A utility function to find the distance between two points
float dist(Point p1, Point p2)
{
    return sqrt((p1.x - p2.x)*(p1.x - p2.x) +
                (p1.y - p2.y)*(p1.y - p2.y));
}

// A Brute Force method to return the smallest distance between two points
// in P[] of size n
float bruteForce(Point P[], int n)
{
    float min = FLT_MAX;
    for (int i = 0; i < n; ++i)
        for (int j = i+1; j < n; ++j)
            if (dist(P[i], P[j]) < min)
                min = dist(P[i], P[j]);
    return min;
}

// A utility function to find minimum of two float values
float min(float x, float y)
{
    return (x < y)? x : y;
}

// A utility function to find the distance between the closest points of
// strip of given size. All points in strip[] are sorted according to
// y coordinate. They all have an upper bound on minimum distance as d.
// Note that this method seems to be a O(n^2) method, but it's a O(n)
// method as the inner loop runs at most 6 times
float stripClosest(Point strip[], int size, float d)
{
    float min = d; // Initialize the minimum distance as d

    qsort(strip, size, sizeof(Point), compareY);

    // Pick all points one by one and try the next points till the difference
    // between y coordinates is smaller than d.
    // This is a proven fact that this loop runs at most 6 times
    for (int i = 0; i < size; ++i)
        for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)
            if (dist(strip[i], strip[j]) < min)
                min = dist(strip[i], strip[j]);

    return min;
}

// A recursive function to find the smallest distance. The array P contains
// all points sorted according to x coordinate
float closestUtil(Point P[], int n)
{
    // If there are 2 or 3 points, then use brute force
    if (n <= 3)
        return bruteForce(P, n);

    // Find the middle point
    int mid = n/2;
    Point midPoint = P[mid];

    // Consider the vertical line passing through the middle point
    // calculate the smallest distance dl on left of middle point and
    // dr on right side
    float dl = closestUtil(P, mid);
    float dr = closestUtil(P + mid, n-mid);

    // Find the smaller of two distances
    float d = min(dl, dr);

    // Build an array strip[] that contains points close (closer than d)
    // to the line passing through the middle point
    Point strip[n];
    int j = 0;
    for (int i = 0; i < n; i++)
        if (abs(P[i].x - midPoint.x) < d)
            strip[j] = P[i], j++;

    // Find the closest points in strip. Return the minimum of d and closest
    // distance is strip[]
    return min(d, stripClosest(strip, j, d));
}

// The main function that finds the smallest distance
// This method mainly uses closestUtil()

```

```

float closest(Point P[], int n)
{
    qsort(P, n, sizeof(Point), compareX);

    // Use recursive function closestUtil() to find the smallest distance
    return closestUtil(P, n);
}

// Driver program to test above functions
int main()
{
    Point P[] = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}};
    int n = sizeof(P) / sizeof(P[0]);
    printf("The smallest distance is %f ", closest(P, n));
    return 0;
}

```

Output:

The smallest distance is 1.414214

Time Complexity Let Time complexity of above algorithm be $T(n)$. Let us assume that we use a $O(n\log n)$ sorting algorithm. The above algorithm divides all points in two sets and recursively calls for two sets. After dividing, it finds the strip in $O(n)$ time, sorts the strip in $O(n\log n)$ time and finally finds the closest points in strip in $O(n)$ time. So $T(n)$ can be expressed as follows

$$T(n) = 2T(n/2) + O(n) + O(n\log n) + O(n)$$

$$T(n) = 2T(n/2) + O(n\log n)$$

$$T(n) = T(n \times \log n \times \log n)$$

Notes

1) Time complexity can be improved to $O(n\log n)$ by optimizing step 5 of the above algorithm. We will soon be discussing the optimized solution in a separate post.

2) The code finds smallest distance. It can be easily modified to find the points with smallest distance.

3) The code uses quick sort which can be $O(n^2)$ in worst case. To have the upper bound as $O(n(\log n)^2)$, a $O(n\log n)$ sorting algorithm like merge sort or heap sort can be used

References:

<http://www.cs.umd.edu/class/fall2013/cpsc451/Lects/lect10.pdf>
<http://www.youtube.com/watch?v=vS4Zn1a9KUc>
<http://www.youtube.com/watch?v=T3T7T8Ym20M>
http://en.wikipedia.org/wiki/Closest_pair_of_points_problem

GATE CS Corner Company Wise Coding Practice

Divide and Conquer
Geometric
Closest Pair of Points
Divide and Conquer

Divide and Conquer | Set 5 (Strassen's Matrix Multiplication)

Given two square matrices A and B of size $n \times n$ each, find their multiplication matrix.

Naive Method

Following is a simple way to multiply two matrices.

```

void multiply(int A[ ][N], int B[ ][N], int C[ ][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k < N; k++)
            {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}

```

Time Complexity of above method is $O(N^3)$.

Divide and Conquer

Following is simple Divide and Conquer method to multiply two square matrices.

- 1) Divide matrices A and B in 4 sub-matrices of size $N/2 \times N/2$ as shown in the below diagram.
- 2) Calculate following values recursively. $ae + bg$, $af + bh$, $ce + dg$ and $cf + dh$.

strassen_new



In the above method, we do 8 multiplications for matrices of size $N/2 \times N/2$ and 4 additions. Addition of two matrices takes $O(N^2)$ time. So the time complexity can be written as

$$T(N) = 8T(N/2) + O(N^2)$$

From Master's Theorem, time complexity of above method is $O(N^3)$ which is unfortunately same as the above naive method.

Simple Divide and Conquer also leads to $O(N^3)$, can there be a better way?

In the above divide and conquer method, the main component for high time complexity is 8 recursive calls. The idea of **Strassen's method** is to reduce the number of recursive calls to 7. Strassen's method is similar to above simple divide and conquer method in the sense that this method also divides matrices to sub-matrices of size $N/2 \times N/2$ as shown in the above diagram, but in Strassen's method, the four sub-matrices of result are calculated using following formulae.



Time Complexity of Strassen's Method

Addition and Subtraction of two matrices takes $O(N^2)$ time. So time complexity can be written as

$$T(N) = 7T(N/2) + O(N^2)$$

From Master's Theorem, time complexity of above method is $O(N^{\log 7})$ which is approximately $O(N^{2.8074})$

Generally Strassen's Method is not preferred for practical applications for following reasons.

- 1) The constants used in Strassen's method are high and for a typical application Naive method works better.
- 2) For Sparse matrices, there are better methods especially designed for them.
- 3) The submatrices in recursion take extra space.
- 4) Because of the limited precision of computer arithmetic on noninteger values, larger errors accumulate in Strassen's algorithm than in Naive Method (Source: CLRS Book)

Easy way to remember Strassen's Matrix Equation

References:

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

<https://www.youtube.com/watch?v=L0LebQ8nKHA>

<https://www.youtube.com/watch?v=QXY4RskLQcl>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Divide and Conquer
Divide and Conquer

Closest Pair of Points | $O(n\log n)$ Implementation

We are given an array of n points in the plane, and the problem is to find out the closest pair of points in the array. This problem arises in a number of applications. For example, in air-traffic control, you may want to monitor planes that come too close together, since this may indicate a possible collision. Recall the following formula for distance between two points p and q .



We have discussed a [divide and conquer solution](#) for this problem. The time complexity of the implementation provided in the previous post is $O(n (\log n)^2)$. In this post, we discuss an implementation with time complexity as $O(n\log n)$.

Following is a recap of the algorithm discussed in the previous post.

- 1) We sort all points according to x coordinates.
- 2) Divide all points in two halves.
- 3) Recursively find the smallest distances in both subarrays.
- 4) Take the minimum of two smallest distances. Let the minimum be d .
- 5) Create an array $\text{strip}[]$ that stores all points which are at most d distance away from the middle line dividing the two sets.
- 6) Find the smallest distance in $\text{strip}[]$.
- 7) Return the minimum of d and the smallest distance calculated in above step 6.

The great thing about the above approach is, if the array $\text{strip}[]$ is sorted according to y coordinate, then we can find the smallest distance in $\text{strip}[]$ in $O(n)$ time. In the implementation discussed in previous post, $\text{strip}[]$ was explicitly sorted in every recursive call that made the time complexity $O(n (\log n)^2)$, assuming that the sorting step takes $O(n\log n)$ time. In this post, we discuss an implementation where the time complexity is $O(n\log n)$. The idea is to presort all points according to y coordinates. Let the sorted array be $\text{Py}[]$. When we make recursive calls, we need to divide points of $\text{Py}[]$ also according to the vertical line. We can do that by simply processing every point and comparing its x coordinate with x coordinate of middle line.

Following is C++ implementation of $O(n\log n)$ approach.

```
// A divide and conquer program in C++ to find the smallest distance from a
// given set of points.

#include <iostream>
#include <float.h>
#include <stdlib.h>
#include <math.h>
using namespace std;

// A structure to represent a Point in 2D plane
struct Point
{
    int x, y;
};

/* Following two functions are needed for library function qsort().
   Refer: http://www.cplusplus.com/reference/cstdlib/qsort/ */

// Needed to sort array of points according to X coordinate
```

```

int compareX(const void* a, const void* b)
{
    Point *p1 = (Point *)a, *p2 = (Point *)b;
    return (p1->x - p2->x);
}

// Needed to sort array of points according to Y coordinate
int compareY(const void* a, const void* b)
{
    Point *p1 = (Point *)a, *p2 = (Point *)b;
    return (p1->y - p2->y);
}

// A utility function to find the distance between two points
float dist(Point p1, Point p2)
{
    return sqrt( (p1.x - p2.x)*(p1.x - p2.x) +
                 (p1.y - p2.y)*(p1.y - p2.y)
               );
}

// A Brute Force method to return the smallest distance between two points
// in P[] of size n
float bruteForce(Point P[], int n)
{
    float min = FLT_MAX;
    for (int i = 0; i < n; ++i)
        for (int j = i+1; j < n; ++j)
            if (dist(P[i], P[j]) < min)
                min = dist(P[i], P[j]);
    return min;
}

// A utility function to find minimum of two float values
float min(float x, float y)
{
    return (x < y)? x : y;
}

// A utility function to find the distance between the closest points of
// strip of given size. All points in strip[] are sorted according to
// y coordinate. They all have an upper bound on minimum distance as d.
// Note that this method seems to be a O(n^2) method, but it's a O(n)
// method as the inner loop runs at most 6 times
float stripClosest(Point strip[], int size, float d)
{
    float min = d; // Initialize the minimum distance as d

    // Pick all points one by one and try the next points till the difference
    // between y coordinates is smaller than d.
    // This is a proven fact that this loop runs at most 6 times
    for (int i = 0; i < size; ++i)
        for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)
            if (dist(strip[i], strip[j]) < min)
                min = dist(strip[i], strip[j]);

    return min;
}

// A recursive function to find the smallest distance. The array Px contains
// all points sorted according to x coordinates and Py contains all points
// sorted according to y coordinates
float closestUtil(Point Px[], Point Py[], int n)
{
    // If there are 2 or 3 points, then use brute force
    if (n <= 3)
        return bruteForce(Px, n);

    // Find the middle point
    int mid = n/2;
    Point midPoint = Px[mid];

    // Divide points in y sorted array around the vertical line.
    // Assumption: All x coordinates are distinct.
    Point PyL[mid+1]; // y sorted points on left of vertical line
    Point PyR[n-mid-1]; // y sorted points on right of vertical line
    int li = 0, ri = 0; // indexes of left and right subarrays
    for (int i = 0; i < n; i++)
    {
        if (Py[i].x <= midPoint.x)
            PyL[li++] = Py[i];
        else
            PyR[ri++] = Py[i];
    }

    // Consider the vertical line passing through the middle point
    // calculate the smallest distance dl on left of middle point and
    // dr on right side
    float dl = closestUtil(Px, PyL, mid);
    float dr = closestUtil((Px + mid), PyR, n-mid);

    // Find the smaller of two distances
    float d = min(dl, dr);

    // Build an array strip[] that contains points close (closer than d)
    // to the line passing through the middle point
    Point strip[n];
    int j = 0;
    for (int i = 0; i < n; i++)
        if (abs(Py[i].x - midPoint.x) < d)
            strip[j++] = Py[i];
}

// Find the closest points in strip. Return the minimum of d and closest
// distance is strip[]
return min(d, stripClosest(strip, j, d));
}

```

```

// The main function that finds the smallest distance
// This method mainly uses closestUtil()
float closest(Point P[], int n)
{
    Point Px[n];
    Point Py[n];
    for (int i = 0; i < n; i++)
    {
        Px[i] = P[i];
        Py[i] = P[i];
    }

    qsort(Px, n, sizeof(Point), compareX);
    qsort(Py, n, sizeof(Point), compareY);

    // Use recursive function closestUtil() to find the smallest distance
    return closestUtil(Px, Py, n);
}

// Driver program to test above functions
int main()
{
    Point P[] = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}};
    int n = sizeof(P) / sizeof(P[0]);
    cout << "The smallest distance is " << closest(P, n);
    return 0;
}

```

Output:

The smallest distance is 1.41421

Time Complexity: Let Time complexity of above algorithm be $T(n)$. Let us assume that we use a $O(n\log n)$ sorting algorithm. The above algorithm divides all points in two sets and recursively calls for two sets. After dividing, it finds the strip in $O(n)$ time. Also, it takes $O(n)$ time to divide the Py array around the mid vertical line. Finally finds the closest points in strip in $O(n)$ time. So $T(n)$ can be expressed as follows

$$T(n) = 2T(n/2) + O(n) + O(n) + O(n)$$

$$T(n) = 2T(n/2) + O(n)$$

$$T(n) = T(n\log n)$$

References:

<http://www.cs.umd.edu/class/fall2013/cmsc451/Lects/lect10.pdf>
<http://www.youtube.com/watch?v=vS4Zn1a9KUc>
<http://www.youtube.com/watch?v=T3T7T8Ym20M>
http://en.wikipedia.org/wiki/Closest_pair_of_points_problem

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Divide and Conquer
Geometric
Divide and Conquer
geometric algorithms

How to check if two given line segments intersect?

Given two line segments (p_1, q_1) and (p_2, q_2) , find if the given line segments intersect with each other.

Before we discuss solution, let us define notion of **orientation**. Orientation of an ordered triplet of points in the plane can be

- counterclockwise
- clockwise
- colinear

The following diagram shows different possible orientations of (a, b, c)



How is Orientation useful here?

Two segments (p_1, q_1) and (p_2, q_2) intersect if and only if one of the following two conditions is verified

1. General Case:

- (p_1, q_1, p_2) and (p_1, q_1, q_2) have different orientations and
- (p_2, q_2, p_1) and (p_2, q_2, q_1) have different orientations.

Examples:

GeneralCaseExamples



examplesGeneralCase21



2. Special Case

- (p_1, q_1, p_2) , (p_1, q_1, q_2) , (p_2, q_2, p_1) , and (p_2, q_2, q_1) are all collinear and
- the x-projections of (p_1, q_1) and (p_2, q_2) intersect
- the y-projections of (p_1, q_1) and (p_2, q_2) intersect

Examples:

examplesSpecialCase



Following is C++ implementation based on above idea.

```
// A C++ program to check if two given line segments intersect
#include <iostream>
using namespace std;

struct Point
{
    int x;
    int y;
};

// Given three colinear points p, q, r, the function checks if
// point q lies on line segment 'pr'
bool onSegment(Point p, Point q, Point r)
{
    if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&
        q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))
        return true;

    return false;
}

// To find orientation of ordered triplet (p, q, r).
// The function returns following values
// 0 --> p, q and r are colinear
// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(Point p, Point q, Point r)
{
    // See http://www.geeksforgeeks.org/orientation-3-ordered-points/
    // for details of below formula.
    int val = (q.y - p.y) * (r.x - q.x) -
              (q.x - p.x) * (r.y - q.y);

    if (val == 0) return 0; // colinear

    return (val > 0)? 1: 2; // clock or counterclock wise
}

// The main function that returns true if line segment 'p1q1'
// and 'p2q2' intersect.
bool doIntersect(Point p1, Point q1, Point p2, Point q2)
{
    // Find the four orientations needed for general and
    // special cases
    int o1 = orientation(p1, q1, p2);
    int o2 = orientation(p1, q1, q2);
    int o3 = orientation(p2, q2, p1);
    int o4 = orientation(p2, q2, q1);

    // General case
    if (o1 != o2 && o3 != o4)
        return true;

    // Special Cases
    // p1, q1 and p2 are colinear and p2 lies on segment p1q1
    if (o1 == 0 && onSegment(p1, p2, q1))
        return true;

    // p1, q1 and p2 are colinear and q2 lies on segment p1q1
    if (o2 == 0 && onSegment(p1, q2, q1))
        return true;

    // p2, q2 and p1 are colinear and p1 lies on segment p2q2
    if (o3 == 0 && onSegment(p2, p1, q2))
        return true;

    // p2, q2 and q1 are colinear and q1 lies on segment p2q2
    if (o4 == 0 && onSegment(p2, q1, q2))
        return true;

    return false; // Doesn't fall in any of the above cases
}

// Driver program to test above functions
int main()
{
    struct Point p1 = {1, 1}, q1 = {10, 1};
    struct Point p2 = {1, 2}, q2 = {10, 2};

    doIntersect(p1, q1, p2, q2)? cout << "Yes\n": cout << "No\n";
}
```

p1 = {10, 0}, q1 = {0, 10};

```

p2 = {0, 0}, q2 = {10, 10};
doIntersect(p1, q1, p2, q2)? cout << "Yes\n": cout << "No\n";

p1 = {-5, -5}, q1 = {0, 0};
p2 = {1, 1}, q2 = {10, 10};
doIntersect(p1, q1, p2, q2)? cout << "Yes\n": cout << "No\n";

return 0;
}

```

Output:

```

No
Yes
No

```

Sources:

<http://www.dcs.gla.ac.uk/~pat/52233/slides/Geometry1x1.pdf>

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Geometric
geometric algorithms
MathematicalAlgo

How to check if a given point lies inside or outside a polygon?

Given a polygon and a point 'p', find if 'p' lies inside the polygon or not. The points lying on the border are considered inside.



We strongly recommend to see the following post first.

[How to check if two given line segments intersect?](#)

Following is a simple idea to check whether a point is inside or outside.

1) Draw a horizontal line to the right of each point and extend it to infinity

1) Count the number of times the line intersects with polygon edges.

2) A point is inside the polygon if either count of intersections is odd or point lies on an edge of polygon. If none of the conditions is true, then point lies outside.



How to handle point 'g' in the above figure?

Note that we should return true if the point lies on the line or same as one of the vertices of the given polygon. To handle this, after checking if the line from 'p' to extreme intersects, we check whether 'p' is colinear with vertices of current line of polygon. If it is colinear, then we check if the point 'p' lies on current side of polygon, if it lies, we return true, else false.

Following is C++ implementation of the above idea.

```

// A C++ program to check if a given point lies inside a given polygon
// Refer http://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/
// for explanation of functions onSegment(), orientation() and doIntersect()
#include <iostream>
using namespace std;

// Define Infinite (Using INT_MAX caused overflow problems)
#define INF 10000

struct Point
{
    int x;
    int y;
};

// Given three colinear points p, q, r, the function checks if
// point q lies on line segment 'pr'
bool onSegment(Point p, Point q, Point r)
{
    if (q.x <= max(p.x, r.x) && q.x >= min(p.x, r.x) &&
        q.y <= max(p.y, r.y) && q.y >= min(p.y, r.y))
        return true;
    return false;
}

// To find orientation of ordered triplet (p, q, r).
// The function returns following values
// 0 --> p, q and r are colinear
// 1 --> Clockwise
// 2 --> Counter-clockwise
int orientation(Point p, Point q, Point r)
{
    int val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y);
    if (val == 0)
        return 0; // colinear
    return (val > 0) ? 1 : 2; // clock or counter-clockwise
}

// Function to check if line segment 'pr' intersects with segment 'pq'
bool doIntersect(Point p, Point q, Point r, Point s)
{
    // Find the four orientations needed for general and
    // handle if point 'q' lies on line 'pr'
    int o1 = orientation(p, q, r);
    int o2 = orientation(p, q, s);
    int o3 = orientation(r, s, p);
    int o4 = orientation(r, s, q);

    // General case
    if (o1 != o2 && o3 != o4)
        return true;

    // Handle if point 'q' lies on line 'pr'
    if (o1 == 0 && q.x >= min(p.x, r.x) && q.x <= max(p.x, r.x) &&
        q.y >= min(p.y, r.y) && q.y <= max(p.y, r.y))
        return true;

    // Handle if point 'r' lies on line 'pq'
    if (o2 == 0 && r.x >= min(p.x, q.x) && r.x <= max(p.x, q.x) &&
        r.y >= min(p.y, q.y) && r.y <= max(p.y, q.y))
        return true;

    // Handle if point 'p' lies on line 'rs'
    if (o3 == 0 && p.x >= min(r.x, s.x) && p.x <= max(r.x, s.x) &&
        p.y >= min(r.y, s.y) && p.y <= max(r.y, s.y))
        return true;

    // Handle if point 's' lies on line 'pr'
    if (o4 == 0 && s.x >= min(r.x, s.x) && s.x <= max(r.x, s.x) &&
        s.y >= min(r.y, s.y) && s.y <= max(r.y, s.y))
        return true;

    return false;
}

```

```

// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(Point p, Point q, Point r)
{
    int val = (q.y - p.y) * (r.x - q.x) -
              (q.x - p.x) * (r.y - q.y);

    if (val == 0) return 0; // colinear
    return (val > 0)? 1: 2; // clock or counterclock wise
}

// The function that returns true if line segment 'p1q1'
// and 'p2q2' intersect.
bool doIntersect(Point p1, Point q1, Point p2, Point q2)
{
    // Find the four orientations needed for general and
    // special cases
    int o1 = orientation(p1, q1, p2);
    int o2 = orientation(p1, q1, q2);
    int o3 = orientation(p2, q2, p1);
    int o4 = orientation(p2, q2, q1);

    // General case
    if (o1 != o2 && o3 != o4)
        return true;

    // Special Cases
    // p1, q1 and p2 are colinear and p2 lies on segment p1q1
    if (o1 == 0 && onSegment(p1, p2, q1)) return true;

    // p1, q1 and p2 are colinear and q2 lies on segment p1q1
    if (o2 == 0 && onSegment(p1, q2, q1)) return true;

    // p2, q2 and p1 are colinear and p1 lies on segment p2q2
    if (o3 == 0 && onSegment(p2, p1, q2)) return true;

    // p2, q2 and q1 are colinear and q1 lies on segment p2q2
    if (o4 == 0 && onSegment(p2, q1, q2)) return true;

    return false; // Doesn't fall in any of the above cases
}

// Returns true if the point p lies inside the polygon[] with n vertices
bool isInside(Point polygon[], int n, Point p)
{
    // There must be at least 3 vertices in polygon[]
    if (n < 3) return false;

    // Create a point for line segment from p to infinite
    Point extreme = {INF, p.y};

    // Count intersections of the above line with sides of polygon
    int count = 0, i = 0;
    do
    {
        int next = (i+1)%n;

        // Check if the line segment from 'p' to 'extreme' intersects
        // with the line segment from 'polygon[i]' to 'polygon[next]'
        if (doIntersect(polygon[i], polygon[next], p, extreme))
        {
            // If the point 'p' is colinear with line segment 'i-next',
            // then check if it lies on segment. If it lies, return true,
            // otherwise false
            if (orientation(polygon[i], p, polygon[next]) == 0)
                return onSegment(polygon[i], p, polygon[next]);
        }
        count++;
    }
    i = next;
} while (i != 0);

// Return true if count is odd, false otherwise
return count&1; // Same as (count%2 == 1)
}

// Driver program to test above functions
int main()
{
    Point polygon1[] = {{0, 0}, {10, 0}, {10, 10}, {0, 10}};
    int n = sizeof(polygon1)/sizeof(polygon1[0]);
    Point p = {20, 20};
    isInside(polygon1, n, p)? cout << "Yes \n": cout << "No \n";

    p = {5, 5};
    isInside(polygon1, n, p)? cout << "Yes \n": cout << "No \n";

    Point polygon2[] = {{0, 0}, {5, 5}, {5, 0}};
    p = {3, 3};
    n = sizeof(polygon2)/sizeof(polygon2[0]);
    isInside(polygon2, n, p)? cout << "Yes \n": cout << "No \n";

    p = {5, 1};
    isInside(polygon2, n, p)? cout << "Yes \n": cout << "No \n";

    p = {8, 1};
    isInside(polygon2, n, p)? cout << "Yes \n": cout << "No \n";

    Point polygon3[] = {{0, 0}, {10, 0}, {10, 10}, {0, 10}};
    p = {-1,10};
    n = sizeof(polygon3)/sizeof(polygon3[0]);
    isInside(polygon3, n, p)? cout << "Yes \n": cout << "No \n";

    return 0;
}

```

Output:

No
Yes
Yes
Yes
No
No

Time Complexity: O(n) where n is the number of vertices in the given polygon.

Source:

<http://www.dcs.gla.ac.uk/~pat/52233/slides/Geometry1x1.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Geometric
geometric algorithms
MathematicalAlgo

Convex Hull | Set 1 (Jarvis's Algorithm or Wrapping)

Given a set of points in the plane, the convex hull of the set is the smallest convex polygon that contains all the points of it.

convexHull



We strongly recommend to see the following post first.

[How to check if two given line segments intersect?](#)

The idea of Jarvis's Algorithm is simple, we start from the leftmost point (or point with minimum x coordinate value) and we keep wrapping points in counterclockwise direction. The big question is, given a point p as current point, how to find the next point in output? The idea is to use [orientation\(\)](#) here. Next point is selected as the point that beats all other points at counterclockwise orientation, i.e., next point is q if for any other point r, we have "orientation(p, r, q) = counterclockwise". Following is the detailed algorithm.

1) Initialize p as leftmost point.

2) Do following while we don't come back to the first (or leftmost) point.

....a) The next point q is the point such that the triplet (p, q, r) is counterclockwise for any other point r.

....b) next[p] = q (Store q as next of p in the output convex hull).

....c) p = q (Set p as q for next iteration).

jarvis



Below is C++ implementation of above algorithm.

```
// A C++ program to find convex hull of a set of points. Refer
// http://www.geeksforgeeks.org/orientation-3-ordered-points/
// for explanation of orientation()
#include <bits/stdc++.h>
using namespace std;

struct Point
{
    int x, y;
};

// To find orientation of ordered triplet (p, q, r).
// The function returns following values
// 0--> p, q and r are colinear
// 1--> Clockwise
// 2--> Counterclockwise
int orientation(Point p, Point q, Point r)
{
    int val = (q.y - p.y) * (r.x - q.x) -
              (q.x - p.x) * (r.y - q.y);

    if (val == 0) return 0; // colinear
    return (val > 0)? 1: 2; // clock or counterclock wise
}

// Prints convex hull of a set of n points.
void convexHull(Point points[], int n)
{
    // There must be at least 3 points
    if (n < 3) return;

    // Initialize Result
    vector<Point> hull;

    // Find the leftmost point
    int l = 0;
    for (int i = 1; i < n; i++)
        if (points[i].x < points[l].x)
            l = i;

    // Start from leftmost point, keep moving counterclockwise
    // until reach the start point again. This loop runs O(h)
    // times where h is number of points in result or output.
    int p = l, q;
```

```

do
{
    // Add current point to result
    hull.push_back(points[p]);

    // Search for a point 'q' such that orientation(p, x,
    // q) is counterclockwise for all points 'x'. The idea
    // is to keep track of last visited most counterclock-
    // wise point in q. If any point 'i' is more counterclock-
    // wise than q, then update q.
    q = (p+1)%n;
    for (int i = 0; i < n; i++)
    {
        // If i is more counterclockwise than current q, then
        // update q
        if (orientation(points[p], points[i], points[q]) == 2)
            q = i;
    }

    // Now q is the most counterclockwise with respect to p
    // Set p as q for next iteration, so that q is added to
    // result 'hull'
    p = q;
}

} while (p != l); // While we don't come to first point

// Print Result
for (int i = 0; i < hull.size(); i++)
    cout << "(" << hull[i].x << ", "
        << hull[i].y << ")n";
}

// Driver program to test above functions
int main()
{
    Point points[] = {{0, 3}, {2, 2}, {1, 1}, {2, 1},
                      {3, 0}, {0, 0}, {3, 3}};
    int n = sizeof(points)/sizeof(points[0]);
    convexHull(points, n);
    return 0;
}

```

Output: The output is points of the convex hull.

```
(0, 3)
(0, 0)
(3, 0)
(3, 3)
```

Time Complexity: For every point on the hull we examine all the other points to determine the next point. Time complexity is $\Theta(m * n)$ where n is number of input points and m is number of output or hull points ($m \leq n$). In worst case, time complexity is $O(n^2)$. The worst case occurs when all the points are on the hull ($m = n$)

[Set 2- Convex Hull \(Graham Scan\)](#)

Sources:

<http://www.cs.uiuc.edu/~jeffe/teaching/373/notes/x05-convexhull.pdf>
<http://www.dcs.gla.ac.uk/~pat/52233/slides/Hull1x1.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Geometric
geometric algorithms
MathematicalAlgo

Convex Hull | Set 2 (Graham Scan)

Given a set of points in the plane. the convex hull of the set is the smallest convex polygon that contains all the points of it.

[convexHull](#)



We strongly recommend to see the following post first.

[How to check if two given line segments intersect?](#)

We have discussed [Jarvis's Algorithm](#) for Convex Hull. Worst case time complexity of Jarvis's Algorithm is $O(n^2)$. Using Graham's scan algorithm, we can find Convex Hull in $O(n \log n)$ time. Following is Graham's algorithm

Let $\text{points}[0..n-1]$ be the input array.

1) Find the bottom-most point by comparing y coordinate of all points. If there are two points with same y value, then the point with smaller x coordinate value is considered. Let the bottom-most point be P_0 . Put P_0 at first position in output hull.

2) Consider the remaining $n-1$ points and sort them by polar angle in counterclockwise order around $\text{points}[0]$. If polar angle of two points is same, then put the nearest point first.

3) After sorting, check if two or more points have same angle. If two more points have same angle, then remove all same angle points except the point farthest from P_0 . Let the size of new array be m .

4) If m is less than 3, return (Convex Hull not possible)

5) Create an empty stack 'S' and push $\text{points}[0]$, $\text{points}[1]$ and $\text{points}[2]$ to S.

6) Process remaining $m-3$ points one by one. Do following for every point ' $\text{points}[i]$ '

4.1) Keep removing points from stack while [orientation](#) of following 3 points is not counterclockwise (or they don't make a left turn).

- a) Point next to top in stack
- b) Point at the top of stack
- c) $\text{points}[i]$

4.2) Push $\text{points}[i]$ to S

5) Print contents of S

The above algorithm can be divided in two phases.

Phase 1 (Sort points): We first find the bottom-most point. The idea is to pre-process points by sorting them with respect to the bottom-most point. Once the points are sorted, they form a simple closed path (See following diagram).

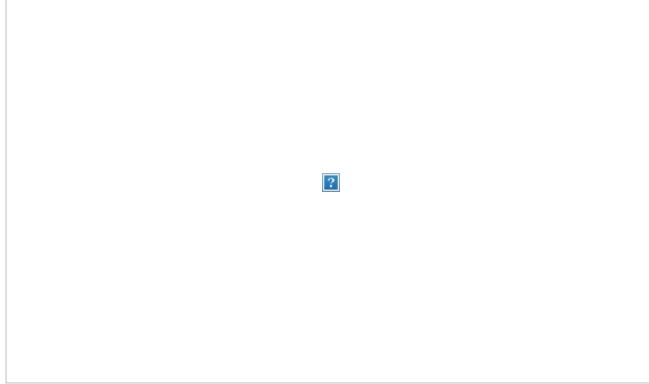
graham



What should be the sorting criteria? computation of actual angles would be inefficient since trigonometric functions are not simple to evaluate. The idea is to use the orientation to compare angles without actually computing them (See the compare() function below)

Phase 2 (Accept or Reject Points): Once we have the closed path, the next step is to traverse the path and remove concave points on this path. How to decide which point to remove and which to keep? Again, orientation helps here. The first two points in sorted array are always part of Convex Hull. For remaining points, we keep track of recent three points, and find the angle formed by them. Let the three points be prev(p), curr(c) and next(n). If orientation of these points (considering them in same order) is not counterclockwise, we discard c, otherwise we keep it. Following diagram shows step by step process of this phase (Source of these diagrams is Ref 2).

Graham1



Following is C++ implementation of the above algorithm.

```
// A C++ program to find convex hull of a set of points. Refer
// http://www.geeksforgeeks.org/orientation-3-ordered-points/
// for explanation of orientation()
#include <iostream>
#include <stack>
#include <stdlib.h>
using namespace std;

struct Point
{
    int x, y;
};

// A global point needed for sorting points with reference
// to the first point Used in compare function of qsort()
Point p0;

// A utility function to find next to top in a stack
Point nextToTop(stack<Point> &S)
{
    Point p = S.top();
    S.pop();
    Point res = S.top();
    S.push(p);
    return res;
}

// A utility function to swap two points
int swap(Point &p1, Point &p2)
{
    Point temp = p1;
    p1 = p2;
    p2 = temp;
}

// A utility function to return square of distance
// between p1 and p2
int distSq(Point p1, Point p2)
{
    return (p1.x - p2.x)*(p1.x - p2.x) +
        (p1.y - p2.y)*(p1.y - p2.y);
}

// To find orientation of ordered triplet (p, q, r).
// The function returns following values
// 0 => p, q and r are colinear
// 1 => Clockwise
// 2 => Counterclockwise
int orientation(Point p, Point q, Point r)
{
    int val = (q.y - p.y) * (r.x - q.x) -
        (q.x - p.x) * (r.y - q.y);

    if (val == 0) return 0; // colinear
    return (val > 0)? 1: 2; // clock or counterclock wise
}

// A function used by library function qsort() to sort an array of
// points with respect to the first point
```

```

int compare(const void *vp1, const void *vp2)
{
    Point *p1 = (Point *)vp1;
    Point *p2 = (Point *)vp2;

    // Find orientation
    int o = orientation(p0, *p1, *p2);
    if (o == 0)
        return (distSq(p0, *p2) >= distSq(p0, *p1)) ? -1 : 1;

    return (o == 2) ? -1 : 1;
}

// Prints convex hull of a set of n points.
void convexHull(Point points[], int n)
{
    // Find the bottommost point
    int ymin = points[0].y, min = 0;
    for (int i = 1; i < n; i++)
    {
        int y = points[i].y;

        // Pick the bottom-most or chose the left
        // most point in case of tie
        if ((y < ymin) || (ymin == y &&
            points[i].x < points[min].x))
            ymin = points[i].y, min = i;
    }

    // Place the bottom-most point at first position
    swap(points[0], points[min]);

    // Sort n-1 points with respect to the first point.
    // A point p1 comes before p2 in sorted output if p2
    // has larger polar angle (in counterclockwise
    // direction) than p1
    p0 = points[0];
    qsort(&points[1], n-1, sizeof(Point), compare);

    // If two or more points make same angle with p0,
    // Remove all but the one that is farthest from p0
    // Remember that, in above sorting, our criteria was
    // to keep the farthest point at the end when more than
    // one points have same angle.
    int m = 1; // Initialize size of modified array
    for (int i=1; i<n; i++)
    {
        // Keep removing i while angle of i and i+1 is same
        // with respect to p0
        while (i < n-1 && orientation(p0, points[i],
            points[i+1]) == 0)
            i++;

        points[m] = points[i];
        m++; // Update size of modified array
    }

    // If modified array of points has less than 3 points,
    // convex hull is not possible
    if (m < 3) return;

    // Create an empty stack and push first three points
    // to it.
    stack<Point> S;
    S.push(points[0]);
    S.push(points[1]);
    S.push(points[2]);

    // Process remaining n-3 points
    for (int i = 3; i < m; i++)
    {
        // Keep removing top while the angle formed by
        // points next-to-top, top, and points[i] makes
        // a non-left turn
        while (orientation(nextToTop(S), S.top(), points[i]) != 2)
            S.pop();
        S.push(points[i]);
    }

    // Now stack has the output points, print contents of stack
    while (!S.empty())
    {
        Point p = S.top();
        cout << "(" << p.x << ", " << p.y << ")" << endl;
        S.pop();
    }
}

// Driver program to test above functions
int main()
{
    Point points[] = {{0, 3}, {1, 1}, {2, 2}, {4, 4},
                      {0, 0}, {1, 2}, {3, 1}, {3, 3}};
    int n = sizeof(points)/sizeof(points[0]);
    convexHull(points, n);
    return 0;
}

```

Output:

```
(0, 3)
(4, 4)
(3, 1)
(0, 0)
```

Time Complexity: Let n be the number of input points. The algorithm takes $O(n\log n)$ time if we use a $O(n\log n)$ sorting algorithm. The first step (finding the bottom-most point) takes $O(n)$ time. The second step (sorting points) takes $O(n\log n)$ time. Third step takes $O(n)$ time. In third step, every element is pushed and popped at most one time. So the sixth step to process points one by one takes $O(n)$ time, assuming that the stack operations take $O(1)$ time. Overall complexity is $O(n) + O(n\log n) + O(n) + O(n)$ which is $O(n\log n)$.

References:

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest
<http://www.dcs.gla.ac.uk/~pat/52233/slides/Hull1x1.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Geometric
geometric algorithms
MathematicalAlgo

Given n line segments, find if any two segments intersect

We have discussed the problem to detect if [two given line segments intersect or not](#). In this post, we extend the problem. Here we are given n line segments and we need to find out if any two line segments intersect or not.

Naive Algorithm A naive solution to solve this problem is to check every pair of lines and check if the pair intersects or not. We can check two line segments in $O(1)$ time. Therefore, this approach takes $O(n^2)$.

Sweep Line Algorithm: We can solve this problem in $O(n\log n)$ time using Sweep Line Algorithm. The algorithm first sorts the end points along the x axis from left to right, then it passes a vertical line through all points from left to right and checks for intersections. Following are detailed steps.

1) Let there be n given lines. There must be $2n$ end points to represent the n lines. Sort all points according to x coordinates. While sorting maintain a flag to indicate whether this point is left point of its line or right point.

2) Start from the leftmost point. Do following for every point

....a) If the current point is a left point of its line segment, check for intersection of its line segment with the segments just above and below it. And add its line to active line segments (line segments for which left end point is seen, but right end point is not seen yet). Note that we consider only those neighbors which are still active.

....b) If the current point is a right point, remove its line segment from active list and check whether its two active neighbors (points just above and below) intersect with each other.

The step 2 is like passing a vertical line from all points starting from the leftmost point to the rightmost point. That is why this algorithm is called Sweep Line Algorithm. The Sweep Line technique is useful in many other geometric algorithms like [calculating the 2D Voronoi diagram](#)

What data structures should be used for efficient implementation?

In step 2, we need to store all active line segments. We need to do following operations efficiently:

- Insert a new line segment
- Delete a line segment
- Find predecessor and successor according to y coordinate values

The obvious choice for above operations is Self-Balancing Binary Search Tree like AVL Tree, Red Black Tree. With a Self-Balancing BST, we can do all of the above operations in $O(\log n)$ time.

Also, in step 1, instead of sorting, we can use min heap data structure. Building a min heap takes $O(n)$ time and every extract min operation takes $O(\log n)$ time (See [this](#)).

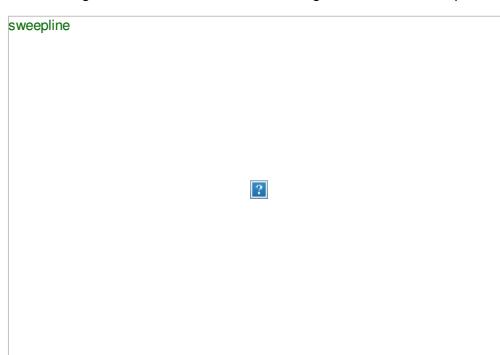
PseudoCode:

The following pseudocode doesn't use heap. It simply sort the array.

```
sweepLineIntersection(Points[0..2n-1]):  
1. Sort Points[] from left to right (according to x coordinate)  
  
2. Create an empty Self-Balancing BST T. It will contain all active line  
Segments ordered by y coordinate.  
  
// Process all 2n points  
3. for i = 0 to 2n-1  
  
    // If this point is left end of its line  
    if (Points[i].isLeft)  
        T.insert(Points[i].line()) // Insert into the tree  
  
    // Check if this points intersects with its predecessor and successor  
    if (doesIntersect(Points[i].line(), T.pred(Points[i].line())) )  
        return true  
    if (doesIntersect(Points[i].line(), T.succ(Points[i].line())) )  
        return true  
  
    else // If it's a right end of its line  
        // Check if its predecessor and successor intersect with each other  
        if (doesIntersect(T.pred(Points[i].line()), T.succ(Points[i].line())) )  
            return true  
        T.delete(Points[i].line()) // Delete from tree  
  
4. return False
```

Example:

Let us consider the following example taken from [here](#). There are 5 line segments [1](#), [2](#), [3](#), [4](#) and [5](#). The dotted green lines show sweep lines.



Following are steps followed by the algorithm. All points from left to right are processed one by one. We maintain a self-balancing binary search tree.

[Left end point of line segment 1 is processed:](#) 1 is inserted into the Tree. The tree contains 1. No intersection.

Left end point of line segment 2 is processed: Intersection of 1 and 2 is checked. 2 is inserted into the Tree. No intersection. The tree contains 1, 2.

Left end point of line segment 3 is processed: Intersection of 3 with 1 is checked. No intersection. 3 is inserted into the Tree. The tree contains 2, 1, 3.

Right end point of line segment 1 is processed: 1 is deleted from the Tree. Intersection of 2 and 3 is checked. Intersection of 2 and 3 is reported. The tree contains 2, 3. Note that the above pseudocode returns at this point. We can continue from here to report all intersection points.

Left end point of line segment 4 is processed: Intersections of line 4 with lines 2 and 3 are checked. No intersection. 4 is inserted into the Tree. The tree contains 2, 4, 3.

Left end point of line segment 5 is processed: Intersection of 5 with 3 is checked. No intersection. 4 is inserted into the Tree. The tree contains 2, 4, 3, 5.

Right end point of line segment 5 is processed: 5 is deleted from the Tree. The tree contains 2, 4, 3.

Right end point of line segment 4 is processed: 4 is deleted from the Tree. The tree contains 2, 4, 3. Intersection of 2 with 3 is checked. Intersection of 2 with 3 is reported. The tree contains 2, 3. Note that the intersection of 2 and 3 is reported again. We can add some logic to check for duplicates.

Right end point of line segment 2 and 3 are processed: Both are deleted from tree and tree becomes empty.

Time Complexity: The first step is sorting which takes $O(n\log n)$ time. The second step process $2n$ points and for processing every point, it takes $O(\log n)$ time. Therefore, overall time complexity is $O(n\log n)$

References:

<http://www.cs.uiuc.edu/~jeffe/teaching/373/notes/x06-sweepline.pdf>
<http://courses.csail.mit.edu/6.006/spring11/lectures/lec24.pdf>
<http://www.youtube.com/watch?v=dePDHvovJIE>
<http://www.eecs.wsu.edu/~cook/aa/lectures/l25/node10.html>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Geometric
geometric algorithms

Check whether a given point lies inside a triangle or not

Given three corner points of a triangle, and one more point P. Write a function to check whether P lies within the triangle or not.

For example, consider the following program, the function should return true for P(10, 15) and false for P'(30, 15)

```
B(10,30)
  \
   / \
  / \
 / P \   P'
 /   \
A(0,0) ----- C(20,0)
```

Source: Microsoft Interview Question

Solution:

Let the coordinates of three corners be (x_1, y_1) , (x_2, y_2) and (x_3, y_3) . And coordinates of the given point P be (x, y)

- 1) Calculate area of the given triangle, i.e., area of the triangle ABC in the above diagram. Area A = $[x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)]/2$
- 2) Calculate area of the triangle PAB. We can use the same formula for this. Let this area be A1.
- 3) Calculate area of the triangle PBC. Let this area be A2.
- 4) Calculate area of the triangle PAC. Let this area be A3.
- 5) If P lies inside the triangle, then $A_1 + A_2 + A_3$ must be equal to A.

```
#include <stdio.h>
#include <stdlib.h>

/* A utility function to calculate area of triangle formed by (x1, y1),
(x2, y2) and (x3, y3) */
float area(int x1, int y1, int x2, int y2, int x3, int y3)
{
    return abs((x1*(y2-y3) + x2*(y3-y1) + x3*(y1-y2))/2.0);
}

/* A function to check whether point P(x, y) lies inside the triangle formed
by A(x1, y1), B(x2, y2) and C(x3, y3) */
bool isInside(int x1, int y1, int x2, int y2, int x3, int y3, int x, int y)
{
    /* Calculate area of triangle ABC */
    float A = area(x1, y1, x2, y2, x3, y3);

    /* Calculate area of triangle PBC */
    float A1 = area(x, y, x2, y2, x3, y3);

    /* Calculate area of triangle PAC */
    float A2 = area(x1, y1, x, y, x3, y3);

    /* Calculate area of triangle PAB */
    float A3 = area(x1, y1, x2, y, x, y);

    /* Check if sum of A1, A2 and A3 is same as A */
    return (A == A1 + A2 + A3);
}

/* Driver program to test above function */
int main()
{
    /* Let us check whether the point P(10, 15) lies inside the triangle
    formed by A(0, 0), B(20, 0) and C(10, 30) */
    if (isInside(0, 0, 20, 0, 10, 30, 10, 15))
        printf ("Inside");
    else
        printf ("Not Inside");

    return 0;
}
```

Output:

Exercise: Given coordinates of four corners of a rectangle, and a point P. Write a function to check whether P lies inside the given rectangle or not.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Geometric
MathematicalAlgo
triangle

How to check if given four points form a square

Given coordinates of four points in a plane, find if the four points form a square or not.

To check for square, we need to check for following.

- a) All four sides formed by points are same.
- b) The angle between any two sides is 90 degree. (This condition is required as Quadrilateral also has same sides.)



We strongly recommend to minimize your browser and try this yourself first.

The idea is to pick any point and calculate its distance from rest of the points. Let the picked point be 'p'. To form a square, distance of two points must be same from 'p', let this distance be d. The distance from one point must be different from that d and must be equal to $\sqrt{2}$ times d. Let this point with different distance be 'q'.

The above condition is not good enough as the point with different distance can be on the other side. We also need to check that q is at same distance from 2 other points and this distance is same as d.

Below is C++ implementation of above idea.

```
// A C++ program to check if four given points form a square or not.
#include<iostream>
using namespace std;

// Structure of a point in 2D space
struct Point
{
    int x, y;
};

// A utility function to find square of distance
// from point 'p' to point 'q'
int distSq(Point p, Point q)
{
    return (p.x - q.x)*(p.x - q.x) +
           (p.y - q.y)*(p.y - q.y);
}

// This function returns true if (p1, p2, p3, p4) form a
// square, otherwise false
bool isSquare(Point p1, Point p2, Point p3, Point p4)
{
    int d2 = distSq(p1, p2); // from p1 to p2
    int d3 = distSq(p1, p3); // from p1 to p3
    int d4 = distSq(p1, p4); // from p1 to p4

    // If lengths of (p1, p2) and (p1, p3) are same, then
    // following conditions must met to form a square.
    // 1) Square of length of (p1, p4) is same as twice
    //    the square of (p1, p2)
    // 2) p4 is at same distance from p2 and p3
    if (d2 == d3 && 2*d2 == d4)
    {
        int d = distSq(p2, p4);
        return (d == distSq(p3, p4) && d == d2);
    }

    // The below two cases are similar to above case
    if (d3 == d4 && 2*d3 == d2)
    {
        int d = distSq(p2, p3);
        return (d == distSq(p2, p4) && d == d3);
    }
    if (d2 == d4 && 2*d2 == d3)
    {
        int d = distSq(p2, p3);
        return (d == distSq(p3, p4) && d == d2);
    }
}

return false;
}

// Driver program to test above function
int main()
{
    Point p1 = {20, 10}, p2 = {10, 20},
          p3 = {20, 20}, p4 = {10, 10};
    isSquare(p1, p2, p3, p4)? cout << "Yes": cout << "No";
    return 0;
}
```

Output:

Yes

Extended Problem:

Check if four segments form a rectangle

This article is contributed by **Anuj**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: [Algorithms](#)

Write an Efficient Method to Check if a Number is Multiple of 3

The very first solution that comes to our mind is the one that we learned in school. If sum of digits in a number is multiple of 3 then number is multiple of 3 e.g., for 612 sum of digits is 9 so it's a multiple of 3. But this solution is not efficient. You have to get all decimal digits one by one, add them and then check if sum is multiple of 3.

There is a pattern in binary representation of the number that can be used to find if number is a multiple of 3. If difference between count of odd set bits (Bits set at odd positions) and even set bits is multiple of 3 then is the number.

Example: 23 (00..10111)

- 1) Get count of all set bits at odd positions (For 23 it's 3).
- 2) Get count of all set bits at even positions (For 23 it's 1).
- 3) If difference of above two counts is a multiple of 3 then number is also a multiple of 3.

(For 23 it's 2 so 23 is not a multiple of 3)

Take some more examples like 21, 15, etc...

```
Algorithm: isMultipleOf3(n)
1) Make n positive if n is negative.
2) If number is 0 then return 1
3) If number is 1 then return 0
4) Initialize: odd_count = 0, even_count = 0
5) Loop while n != 0
   a) If rightmost bit is set then increment odd count.
   b) Right-shift n by 1 bit
   c) If rightmost bit is set then increment even count.
   d) Right-shift n by 1 bit
6) return isMultipleOf3(odd_count - even_count)
```

Proof:

Above can be proved by taking the example of 11 in decimal numbers. (In this context 11 in decimal numbers is same as 3 in binary numbers)
If difference between sum of odd digits and even digits is multiple of 11 then decimal number is multiple of 11. Let's see how.

Let's take the example of 2 digit numbers in decimal

$$AB = 11A - A + B = 11A + (B - A)$$

So if $(B - A)$ is a multiple of 11 then is AB.

Let us take 3 digit numbers.

$$ABC = 99A + A + 11B - B + C = (99A + 11B) + (A + C - B)$$

So if $(A + C - B)$ is a multiple of 11 then is $(A+C-B)$

Let us take 4 digit numbers now.

$$ABCD = 1001A + D + 11C - C + 999B + B - A$$

$$= (1001A - 999B + 11C) + (D + B - A - C)$$

So, if $(B + D - A - C)$ is a multiple of 11 then is ABCD.

This can be continued for all decimal numbers.

Above concept can be proved for 3 in binary numbers in the same way.

Time Complexity: O(logn)

Program:

```
#include<stdio.h>

/* Function to check if n is a multiple of 3*/
int isMultipleOf3(int n)
{
    int odd_count = 0;
    int even_count = 0;

    /* Make no positive if +n is multiple of 3
     * then is -n. We are doing this to avoid
     * stack overflow in recursion*/
    if(n < 0) n = -n;
    if(n == 0) return 1;
    if(n == 1) return 0;

    while(n)
    {
        /* If odd bit is set then
         * increment odd counter */
        if(n & 1)
            odd_count++;
        n = n>>1;

        /* If even bit is set then
         * increment even counter */
        if(n & 1)
            even_count++;
        n = n>>1;
    }

    return isMultipleOf3(abs(odd_count - even_count));
}
```

```

/* Program to test function isMultipleOf3 */
int main()
{
    int num = 23;
    if (isMultipleOf3(num))
        printf("num is multiple of 3");
    else
        printf("num is not a multiple of 3");
    return 0;
}

```

Related Articles:

[Check divisibility in a binary stream](#)
[DFA based division](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Bit Magic
MathematicalAlgo

Efficient way to multiply with 7

We can multiply a number by 7 using bitwise operator. First left shift the number by 3 bits (you will get $8n$) then subtract the original number from the shifted number and return the difference ($8n - n$).

Program:

```

#include<stdio.h>

int multiplyBySeven(unsigned int n)
{
    /* Note the inner bracket here. This is needed
       because precedence of '-' operator is higher
       than '<<' */
    return ((n<<3) - n);
}

/* Driver program to test above function */
int main()
{
    unsigned int n = 4;
    printf("%u", multiplyBySeven(n));

    getchar();
    return 0;
}

```

Time Complexity: O(1)

Space Complexity: O(1)

Note: Works only for positive integers.

Same concept can be used for fast multiplication by 9 or other numbers.

GATE CS Corner Company Wise Coding Practice

Bit Magic
Bit Magic
MathematicalAlgo

Write a program to print all permutations of a given string

A permutation, also called an "arrangement number" or "order," is a rearrangement of the elements of an ordered list S into a one-to-one correspondence with S itself. A string of length n has $n!$ permutations.

Source: Mathworld(<http://mathworld.wolfram.com/Permutation.html>)

Below are the permutations of string ABC.

ABC ACB BAC BCA CBA CAB

We strongly recommend that you click here and practice it, before moving on to the solution.

Here is a solution that is used as a basis in backtracking.

NewPermutation



C/C++

```

// C program to print all permutations with duplicates allowed
#include <stdio.h>
#include <string.h>

/* Function to swap values at two pointers */

```

```

void swap(char *x, char *y)
{
    char temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

/* Function to print permutations of string
This function takes three parameters:
1. String
2. Starting index of the string
3. Ending index of the string. */
void permute(char *a, int l, int r)
{
    int i;
    if (l == r)
        printf("%s\n", a);
    else
    {
        for (i = l; i <= r; i++)
        {
            swap((a+l), (a+i));
            permute(a, l+1, r);
            swap((a+l), (a+i)); //backtrack
        }
    }
}

/* Driver program to test above functions */
int main()
{
    char str[] = "ABC";
    int n = strlen(str);
    permute(str, 0, n-1);
    return 0;
}

```

Java

```

// Java program to print all permutations of a
// given string.
public class Permutation
{
    public static void main(String[] args)
    {
        String str = "ABC";
        int n = str.length();
        Permutation permutation = new Permutation();
        permutation.permute(str, 0, n-1);
    }

    /**
     * permutation function
     * @param str string to calculate permutation for
     * @param l starting index
     * @param r end index
     */
    private void permute(String str, int l, int r)
    {
        if (l == r)
            System.out.println(str);
        else
        {
            for (int i = l; i <= r; i++)
            {
                str = swap(str,l,i);
                permute(str, l+1, r);
                str = swap(str,l,i);
            }
        }
    }

    /**
     * Swap Characters at position
     * @param a a string value
     * @param i position 1
     * @param j position 2
     * @return swapped string
     */
    public String swap(String a, int i, int j)
    {
        char temp;
        char[] charArray = a.toCharArray();
        temp = charArray[i];
        charArray[i] = charArray[j];
        charArray[j] = temp;
        return String.valueOf(charArray);
    }
}

// This code is contributed by Mihir Joshi

```

Python

```

# Python program to print all permutations with
# duplicates allowed

def toString(List):
    return ''.join(List)

```

```

# Function to print permutations of string
# This function takes three parameters:
# 1. String
# 2. Starting index of the string
# 3. Ending index of the string.
def permute(a, l, r):
    if l==r:
        print toString(a)
    else:
        for i in xrange(l,r+1):
            a[l], a[i] = a[i], a[l]
            permute(a, l+1, r)
            a[l], a[i] = a[i], a[l] # backtrack

# Driver program to test the above function
string = "ABC"
n = len(string)
a = list(string)
permute(a, 0, n-1)

# This code is contributed by Bhavya Jain

```

Output:

```

ABC
ACB
BAC
BCA
CBA
CAB

```

Algorithm Paradigm: Backtracking

Time Complexity: $O(n \cdot n!)$ Note that there are $n!$ permutations and it requires $O(n)$ time to print a permutation.

Note : The above solution prints duplicate permutations if there are repeating characters in input string. Please see below link for a solution that prints only distinct permutations even if there are duplicates in input.

Print all distinct permutations of a given string with duplicates.

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

```

Combinatorial
Strings
Backtracking
MathematicalAlgo
permutation

```

Lucky Numbers

Lucky numbers are subset of integers. Rather than going into much theory, let us see the process of arriving at lucky numbers.

Take the set of integers

1,2,3,4,5,6,7,8,9,10,11,12,14,15,16,17,18,19,.....

First, delete every second number, we get following reduced set.

1,3,5,7,9,11,13,15,17,19,.....

Now, delete every third number, we get

1, 3, 7, 9, 13, 15, 19,.....

Continue this process indefinitely.....

Any number that does NOT get deleted due to above process is called "lucky".

Therefore, set of lucky numbers is 1, 3, 7, 13,.....

Now, given an integer 'n', write a function to say whether this number is lucky or not.

```
bool isLucky(int n)
```

Algorithm:

Before every iteration, if we calculate position of the given no, then in a given iteration, we can determine if the no will be deleted. Suppose calculated position for the given no. is P before some iteration, and each Ith no. is going to be removed in this iteration, if P Recursive Way:

```

#include <stdio.h>
#define bool int

/* Returns 1 if n is a lucky no. otherwise returns 0*/
bool isLucky(int n)

```

```

{
static int counter = 2;

/*variable next_position is just for readability of
the program we can remove it and use n only */
int next_position = n;
if(counter > n)
return 1;
if(n%counter == 0)
return 0;

/*calculate next position of input no*/
next_position -= next_position/counter;

counter++;
return isLucky(next_position);
}

/*Driver function to test above function*/
int main()
{
int x = 5;
if( isLucky(x) )
printf("%d is a lucky no.",x);
else
printf("%d is not a lucky no.",x);
getchar();
}

```

Example:

Let's us take an example of 19

1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,15,17,18,19,20,21,.....
 1,3,5,7,9,11,13,15,17,19,.....
 1,3,7,9,13,15,19,.....
 1,3,7,13,15,19,.....
 1,3,7,13,19,.....

In next step every 6th no .in sequence will be deleted. 19 will not be deleted after this step because position of 19 is 5th after this step. Therefore, 19 is lucky. Let's see how above C code finds out:

Current function call	Position after this call	Counter for next call	Next Call
isLucky(19)	10	3	isLucky(10)
isLucky(10)	7	4	isLucky(7)
isLucky(7)	6	5	isLucky(6)
isLucky(6)	5	6	isLucky(5)

When isLucky(6) is called, it returns 1 (because counter > n).

Iterative Way:

Please see [this](#) comment for another simple and elegant implementation of the above algorithm.

Please write comments if you find any bug in the given programs or other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Mathematical
 C Program for Lucky Numbers
 isLucky
 Lucky Number
 Lucky Numbers
 Mathematical
 Print Lucky numbers
 Program for lucky numbers
 Programs for numbers

Write a program to add two numbers in base 14

Asked by Anshya.

Below are the different ways to add base 14 numbers.

Method 1

Thanks to Raj for suggesting this method.

1. Convert both i/p base 14 numbers to base 10.
2. Add numbers.
3. Convert the result back to base 14.

Method 2

Just add the numbers in base 14 in same way we add in base 10. Add numerals of both numbers one by one from right to left. If there is a carry while adding two numerals, consider the carry for adding next numerals.

Let us consider the presentation of base 14 numbers same as hexadecimal numbers

A -> 10
 B -> 11
 C -> 12
 D -> 13

Example:
 num1 = 1 2 A
 num2 = C D 3

1. Add A and 3, we get 13(D). Since 13 is smaller than 14, carry becomes 0 and resultant numeral becomes D
2. Add 2, D and carry(0). we get 15. Since 15 is greater than 13, carry becomes 1 and resultant numeral is $15 - 14 = 1$
3. Add 1, C and carry(1), we get 14. Since 14 is greater than 13, carry becomes 1 and resultant numeral is $14 - 14 = 0$

Finally, there is a carry, so 1 is added as leftmost numeral and the result becomes 101D

Implementation of Method 2

```
# include <stdio.h>
# include <stdlib.h>
# define bool int

int getNumeralValue(char );
char getNumeral(int);

/* Function to add two numbers in base 14 */
char *sumBase14(char *num1, char *num2)
{
    int l1 = strlen(num1);
    int l2 = strlen(num2);
    char *res;
    int i;
    int nml1, nml2, res_nml;
    bool carry = 0;

    if(l1 != l2)
    {
        printf("Function doesn't support numbers of different"
               " lengths. If you want to add such numbers then"
               " prefix smaller number with required no. of zeroes");
        getchar();
        assert(0);
    }

    /* Note the size of the allocated memory is one
     more than i/p lengths for the cases where we
     have carry at the last like adding D1 and A1 */
    res = (char *)malloc(sizeof(char)*(l1 + 1));

    /* Add all numerals from right to left */
    for(i = l1-1; i >= 0; i--)
    {
        /* Get decimal values of the numerals of
         i/p numbers*/
        nml1 = getNumeralValue(num1[i]);
        nml2 = getNumeralValue(num2[i]);

        /* Add decimal values of numerals and carry */
        res_nml = carry + nml1 + nml2;

        /* Check if we have carry for next addition
         of numerals */
        if(res_nml >= 14)
        {
            carry = 1;
            res_nml -= 14;
        }
        else
        {
            carry = 0;
        }
        res[i+1] = getNumeral(res_nml);
    }

    /* if there is no carry after last iteration
     then result should not include 0th character
     of the resultant string */
    if(carry == 0)
        return (res + 1);

    /* if we have carry after last iteration then
     result should include 0th character */
    res[0] = '1';
    return res;
}

/* Function to get value of a numeral
For example it returns 10 for input 'A'
1 for '1', etc */
int getNumeralValue(char num)
{
    if( num >= '0' && num <= '9')
        return (num - '0');
    if( num >= 'A' && num <= 'D')
        return (num - 'A' + 10);

    /* If we reach this line caller is giving
     invalid character so we assert and fail*/
    assert(0);
}

/* Function to get numeral for a value.
For example it returns 'A' for input 10
'1' for 1, etc */
char getNumeral(int val)
{
    if( val >= 0 && val <= 9)
        return (val + '0');
    if( val >= 10 && val <= 14)
        return (val + 'A' - 10);

    /* If we reach this line caller is giving
     invalid no. so we assert and fail*/
    assert(0);
}

/*Driver program to test above functions*/
int main()
{
    char *num1 = "DC2";
    char *num2 = "0A3";
    printf("Result is %s", sumBase14(num1, num2));
}
```

```
getchar();
return 0;
}
```

Notes:

Above approach can be used to add numbers in any base. We don't have to do string operations if base is smaller than 10.

You can try extending the above program for numbers of different lengths.

Please comment if you find any bug in the program or a better approach to do the same.

GATE CS Corner Company Wise Coding Practice

Mathematical
MathematicalAlgo

Babylonian method for square root

Algorithm:

This method can be derived from (but predates) Newton-Raphson method.

- 1 Start with an arbitrary positive start value x (the closer to the root, the better).
- 2 Initialize $y = 1$.
3. Do following until desired approximation is achieved.
 - a) Get the next approximation for root using average of x and y
 - b) Set $y = n/x$

Implementation:

```
/* Returns the square root of n. Note that the function */
float squareRoot(float n)
{
    /*We are using n itself as initial approximation
    This can definitely be improved */
    float x = n;
    float y = 1;
    float e = 0.000001; /* e decides the accuracy level*/
    while(x - y > e)
    {
        x = (x + y)/2;
        y = n/x;
    }
    return x;
}

/* Driver program to test above function*/
int main()
{
    int n = 50;
    printf ("Square root of %d is %f", n, squareRoot(n));
    getchar();
}
```

Example:

```
n = 4 /*n itself is used for initial approximation*/
Initialize x = 4, y = 1
Next Approximation x = (x + y)/2 (= 2.500000),
y = n/x  (=1.600000)
Next Approximation x = 2.050000,
y = 1.951220
Next Approximation x = 2.000610,
y = 1.999390
Next Approximation x = 2.000000,
y = 2.000000
Terminate as (x - y) > e now.
```

If we are sure that n is a perfect square, then we can use following method. The method can go in infinite loop for non-perfect-square numbers. For example, for 3 the below while loop will never terminate.

```
/* Returns the square root of n. Note that the function
will not work for numbers which are not perfect squares*/
unsigned int squareRoot(int n)
{
    int x = n;
    int y = 1;
    while(x > y)
    {
        x = (x + y)/2;
        y = n/x;
    }
    return x;
}

/* Driver program to test above function*/
int main()
{
    int n = 49;
    printf ("root of %d is %d", n, squareRoot(n));
    getchar();
}
```

References:

http://en.wikipedia.org/wiki/Square_root
http://en.wikipedia.org/wiki/Babylonian_method#Babylonian_method

Asked by Snehal

Please write comments if you find any bug in the above program/algorithm, or if you want to share more information about Babylonian method.

Multiply two integers without using multiplication, division and bitwise operators, and no loops

By making use of recursion, we can multiply two integers with the given constraints.

To multiply x and y, recursively add x y times.

```
#include<stdio.h>
/* function to multiply two numbers x and y*/
int multiply(int x, int y)
{
    /* 0 multiplied with anything gives 0 */
    if(y == 0)
        return 0;

    /* Add x one by one */
    if(y > 0)
        return (x + multiply(x, y-1));

    /* the case where y is negative */
    if(y < 0)
        return -multiply(x, -y);
}

int main()
{
    printf("\n %d", multiply(5, -11));
    getchar();
    return 0;
}
```

Time Complexity: O(y) where y is the second argument to function multiply().

Russian Peasant (Multiply two numbers using bitwise operators)

Please write comments if you find any of the above code/algorithm incorrect, or find better ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Print all combinations of points that can compose a given number

You can win three kinds of basketball points, 1 point, 2 points, and 3 points. Given a total score n, print out all the combination to compose n.

Examples:

For n = 1, the program should print following:

1

For n = 2, the program should print following:

1 1

2

For n = 3, the program should print following:

1 1 1

1 2

2 1

3

For n = 4, the program should print following:

1 1 1 1

1 1 2

1 2 1

1 3

2 1 1

2 2

3 1

and so on ...

Algorithm:

At first position we can have three numbers 1 or 2 or 3.

First put 1 at first position and recursively call for n-1.

Then put 2 at first position and recursively call for n-2.

Then put 3 at first position and recursively call for n-3.

If n becomes 0 then we have formed a combination that compose n, so print the current combination.

Below is a generalized implementation. In the below implementation, we can change MAX_POINT if there are higher points (more than 3) in the basketball game.

```
#define MAX_POINT 3
#define ARR_SIZE 100
#include<stdio.h>

/* Utility function to print array arr[] */
void printArray(int arr[], int arr_size);

/* The function prints all combinations of numbers 1, 2,...MAX_POINT
   that sum up to n.
   i is used in recursion keep track of index in arr[] where next
   element is to be added. Initial value of i must be passed as 0 */
void printCompositions(int n, int i)
{
```

```

/* array must be static as we want to keep track
of values stored in arr[] using current calls of
printCompositions() in function call stack*/
static int arr[ARR_SIZE];

if (n == 0)
{
    printArray(arr, i);
}
else if(n > 0)
{
    int k;
    for (k = 1; k <= MAX_POINT; k++)
    {
        arr[i]= k;
        printCompositions(n-k, i+1);
    }
}

/* UTILITY FUNCTIONS */
/* Utility function to print array arr[] */
void printArray(int arr[], int arr_size)
{
    int i;
    for (i = 0; i < arr_size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

/* Driver function to test above functions */
int main()
{
    int n = 5;
    printf("Different compositions formed by 1, 2 and 3 of %d are\n", n);
    printCompositions(n, 0);
    getchar();
    return 0;
}

```

Asked by [Aloe](#)

Please write comments if you find any bug in above code/algorith, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Mathematical
MathematicalAlgo

Write you own Power without using multiplication(*) and division(/) operators

Method 1 (Using Nested Loops)

We can calculate power by using repeated addition.

For example to calculate 5^6 .

- 1) First 5 times add 5, we get 25. (5^2)
- 2) Then 5 times add 25, we get 125. (5^3)
- 3) Then 5 time add 125, we get 625 (5^4)
- 4) Then 5 times add 625, we get 3125 (5^5)
- 5) Then 5 times add 3125, we get 15625 (5^6)

```

/* Works only if a >= 0 and b >= 0 */
int pow(int a, int b)
{
    if (b == 0)
        return 1;
    int answer = a;
    int increment = a;
    int i, j;
    for(i = 1; i < b; i++)
    {
        for(j = 1; j < a; j++)
        {
            answer += increment;
        }
        increment = answer;
    }
    return answer;
}

/* driver program to test above function */
int main()
{
    printf("\n %d", pow(5, 3));
    getchar();
    return 0;
}

```

Method 2 (Using Recursion)

Recursively add a to get the multiplication of two numbers. And recursively multiply to get a raise to the power b .

```

#include<stdio.h>
/* A recursive function to get a^b
Works only if a >= 0 and b >= 0 */
int pow(int a, int b)
{
    if(b)
        return multiply(a, pow(a, b-1));
    else
        return 1;
}

```

```

/* A recursive function to get x*y */
int multiply(int x, int y)
{
    if(y)
        return (x + multiply(x, y-1));
    else
        return 0;
}

/* driver program to test above functions */
int main()
{
    printf("\n %d", pow(5, 3));
    getchar();
    return 0;
}

```

Please write comments if you find any bug in above code/algorith, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Divide and Conquer
Mathematical
MathematicalAlgo

Program for Fibonacci numbers

The Fibonacci numbers are the numbers in the following integer sequence.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,

In mathematical terms, the sequence F_n of Fibonacci numbers is defined by the recurrence relation

$$F_n = F_{n-1} + F_{n-2}$$

with seed values

$$F_0 = 0 \text{ and } F_1 = 1.$$

Write a function `int fib(int n)` that returns F_n . For example, if $n = 0$, then `fib()` should return 0. If $n = 1$, then it should return 1. For $n > 1$, it should return $F_{n-1} + F_{n-2}$

For n = 9
Output:34

Following are different methods to get the nth Fibonacci number.

Method 1 (Use recursion)

A simple method that is a direct recursive implementation mathematical recurrence relation given above.

C

```

//Fibonacci Series using Recursion
#include<stdio.h>
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}

int main ()
{
    int n = 9;
    printf("%d", fib(n));
    getchar();
    return 0;
}

```

Java

```

//Fibonacci Series using Recursion
class fibonacci
{
    static int fib(int n)
    {
        if (n <= 1)

```

```

    return n;
    return fib(n-1) + fib(n-2);
}

public static void main (String args[])
{
int n = 9;
System.out.println(fib(n));
}
}

/* This code is contributed by Rajat Mishra */

```

Python

```

# Function for nth Fibonacci number

def Fibonacci(n):
    if n<0:
        print("Incorrect input")
    # First Fibonacci number is 0
    elif n==1:
        return 0
    # Second Fibonacci number is 1
    elif n==2:
        return 1
    else:
        return Fibonacci(n-1)+Fibonacci(n-2)

# Driver Program

print(Fibonacci(9))

#This code is contributed by Saket Modi

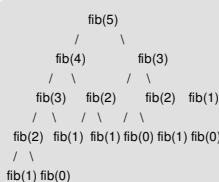
```

Output

34

Time Complexity: $T(n) = T(n-1) + T(n-2)$ which is exponential.

We can observe that this implementation does a lot of repeated work (see the following recursion tree). So this is a bad implementation for nth Fibonacci number.



Extra Space: $O(n)$ if we consider the function call stack size, otherwise $O(1)$.

Method 2 (Use Dynamic Programming)

We can avoid the repeated work done in the method 1 by storing the Fibonacci numbers calculated so far.

C

```

//Fibonacci Series using Dynamic Programming
#include<stdio.h>

int fib(int n)
{
    /* Declare an array to store Fibonacci numbers.*/
    int f[n+1];
    int i;

    /* 0th and 1st number of the series are 0 and 1*/
    f[0] = 0;
    f[1] = 1;

    for (i = 2; i <= n; i++)
    {
        /* Add the previous 2 numbers in the series
        and store it */
        f[i] = f[i-1] + f[i-2];
    }

    return f[n];
}

int main ()
{
    int n = 9;
    printf("%d", fib(n));
    getchar();
    return 0;
}

```

Java

```

// Fibonacci Series using Dynamic Programming
class fibonacci
{
    static int fib(int n)
    {
        /* Declare an array to store Fibonacci numbers.*/
        int f[] = new int[n+1];

```

```

int i;

/* 0th and 1st number of the series are 0 and 1*/
f[0] = 0;
f[1] = 1;

for (i = 2; i <= n; i++)
{
    /* Add the previous 2 numbers in the series
       and store it */
    f[i] = f[i-1] + f[i-2];
}

return f[n];
}

public static void main (String args[])
{
    int n = 9;
    System.out.println(fib(n));
}
*/
/* This code is contributed by Rajat Mishra */

```

Python

```

# Function for nth fibonacci number - Dynamic Programming
# Taking 1st two fibonacci numbers as 0 and 1

FibArray = [0,1]

def fibonacci(n):
    if n<0:
        print("Incorrect input")
    elif n==len(FibArray):
        return FibArray[n-1]
    else:
        temp_fib = fibonacci(n-1)+fibonacci(n-2)
        FibArray.append(temp_fib)
        return temp_fib

# Driver Program

print(fibonacci(9))

#This code is contributed by Saket Modi

```

Output:

34

Time Complexity: O(n)
Extra Space: O(n)

Method 3 (Space Optimized Method 2)

We can optimize the space used in method 2 by storing the previous two numbers only because that is all we need to get the next Fibonacci number in series.

C/C++

```

// Fibonacci Series using Space Optimized Method
#include<stdio.h>
int fib(int n)
{
    int a = 0, b = 1, c, i;
    if( n == 0)
        return a;
    for (i = 2;i <= n;i++)
    {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}

int main ()
{
    int n = 9;
    printf("%d",fib(n));
    getchar();
    return 0;
}

```

Java

```

// Java program for Fibonacci Series using Space
// Optimized Method
class fibonacci
{
    static int fib(int n)
    {
        int a = 0, b = 1, c;
        if (n == 0)
            return a;
        for (int i = 2;i <= n;i++)
        {
            c = a + b;
            a = b;
            b = c;
        }
    }
}

```

```

        }
        return b;
    }

    public static void main (String args[])
    {
        int n = 9;
        System.out.println(fib(n));
    }
}

// This code is contributed by Mihir Joshi

```

Python

```

# Function for nth fibonacci number - Space Optimisataion
# Taking 1st two fibonacci numbers as 0 and 1

def fibonacci(n):
    a = 0
    b = 1
    if n < 0:
        print("Incorrect input")
    elif n == 0:
        return a
    elif n == 1:
        return b
    else:
        for i in range(2,n):
            c = a + b
            a = b
            b = c
        return b

# Driver Program

print(fibonacci(9))

#This code is contributed by Saket Modi

```

Time Complexity: O(n)

Extra Space: O(1)

Method 4 (Using power of the matrix {{1,1},{1,0}})

This another O(n) which relies on the fact that if we n times multiply the matrix $M = \begin{Bmatrix} 1 & 1 \\ 1 & 0 \end{Bmatrix}$ to itself (in other words calculate $\text{power}(M, n)$), then we get the $(n+1)$ th Fibonacci number as the element at row and column (0, 0) in the resultant matrix.

The matrix representation gives the following closed expression for the Fibonacci numbers:



C

```

#include <stdio.h>

/* Helper function that multiplies 2 matrices F and M of size 2*2, and
   puts the multiplication result back to F[][]
   void multiply(int F[2][2], int M[2][2]);

/* Helper function that calculates F[][] raise to the power n and puts the
   result in F[][]
   Note that this function is designed only for fib() and won't work as general
   power function */
void power(int F[2][2], int n);

int fib(int n)
{
    int F[2][2] = {{1,1},{1,0}};
    if (n == 0)
        return 0;
    power(F, n-1);

    return F[0][0];
}

void multiply(int F[2][2], int M[2][2])
{
    int x = F[0][0]*M[0][0] + F[0][1]*M[1][0];
    int y = F[0][0]*M[0][1] + F[0][1]*M[1][1];
    int z = F[1][0]*M[0][0] + F[1][1]*M[1][0];
    int w = F[1][0]*M[0][1] + F[1][1]*M[1][1];

    F[0][0] = x;
    F[0][1] = y;
    F[1][0] = z;
    F[1][1] = w;
}

void power(int F[2][2], int n)
{
    int i;
    int M[2][2] = {{1,1},{1,0}};

    // n - 1 times multiply the matrix to {{1,0},{0,1}}
    for (i = 2; i <= n; i++)
        multiply(F, M);
}

/* Driver program to test above function */
int main()
{

```

```

int n = 9;
printf("%d", fib(n));
getchar();
return 0;
}

```

Java

```

class fibonacci
{
    static int fib(int n)
    {
        int F[] = new int[]{(1,1),(1,0)};
        if (n == 0)
            return 0;
        power(F, n-1);

        return F[0][0];
    }

    /* Helper function that multiplies 2 matrices F and M of size 2*2, and
     * puts the multiplication result back to F */
    static void multiply(int F[], int M[])
    {
        int x = F[0][0]*M[0][0] + F[0][1]*M[1][0];
        int y = F[0][0]*M[0][1] + F[0][1]*M[1][1];
        int z = F[1][0]*M[0][0] + F[1][1]*M[1][0];
        int w = F[1][0]*M[0][1] + F[1][1]*M[1][1];

        F[0][0] = x;
        F[0][1] = y;
        F[1][0] = z;
        F[1][1] = w;
    }

    /* Helper function that calculates F[] raise to the power n and puts the
     * result in F[]
     * Note that this function is designed only for fib() and won't work as general
     * power function */
    static void power(int F[], int n)
    {
        int i;
        int M[] = new int[]{(1,1),(1,0)};

        // n - 1 times multiply the matrix to {(1,0),(0,1)}
        for (i = 2; i <= n; i++)
            multiply(F, M);
    }

    /* Driver program to test above function */
    public static void main (String args[])
    {
        int n = 9;
        System.out.println(fib(n));
    }
}
/* This code is contributed by Rajat Mishra */

```

Time Complexity: O(n)

Extra Space: O(1)

Method 5 (Optimized Method 4)

The method 4 can be optimized to work in O(Logn) time complexity. We can do recursive multiplication to get power(M, n) in the previous method (Similar to the optimization done in [this post](#))

C

```

#include <stdio.h>

void multiply(int F[2][2], int M[2][2]);

void power(int F[2][2], int n);

/* function that returns nth Fibonacci number */
int fib(int n)
{
    int F[2][2] = {{1,1},{1,0}};
    if (n == 0)
        return 0;
    power(F, n-1);
    return F[0][0];
}

/* Optimized version of power() in method 4 */
void power(int F[2][2], int n)
{
    if (n == 0 || n == 1)
        return;
    int M[2][2] = {{1,1},{1,0}};

    power(F, n/2);
    multiply(F, F);

    if (n%2 != 0)
        multiply(F, M);
}

void multiply(int F[2][2], int M[2][2])
{
    int x = F[0][0]*M[0][0] + F[0][1]*M[1][0];
    int y = F[0][0]*M[0][1] + F[0][1]*M[1][1];
    int z = F[1][0]*M[0][0] + F[1][1]*M[1][0];
    int w = F[1][0]*M[0][1] + F[1][1]*M[1][1];

    F[0][0] = x;
    F[0][1] = y;
    F[1][0] = z;
    F[1][1] = w;
}

```

```

int y = F[0][0]*M[0][1] + F[0][1]*M[1][1];
int z = F[1][0]*M[0][0] + F[1][1]*M[1][0];
int w = F[1][0]*M[0][1] + F[1][1]*M[1][1];

F[0][0] = x;
F[0][1] = y;
F[1][0] = z;
F[1][1] = w;
}

/* Driver program to test above function */
int main()
{
    int n = 9;
    printf("%d", fib(9));
    getchar();
    return 0;
}

```

Java

```

//Fibonacci Series using Optimized Method
class fibonacci
{
    /* function that returns nth Fibonacci number */
    static int fib(int n)
    {
        int F[] = new int[]{(1,1),(1,0)};
        if (n == 0)
            return 0;
        power(F, n-1);

        return F[0][0];
    }

    static void multiply(int F[], int M[])
    {
        int x = F[0][0]*M[0][0] + F[0][1]*M[1][0];
        int y = F[0][0]*M[0][1] + F[0][1]*M[1][1];
        int z = F[1][0]*M[0][0] + F[1][1]*M[1][0];
        int w = F[1][0]*M[0][1] + F[1][1]*M[1][1];

        F[0][0] = x;
        F[0][1] = y;
        F[1][0] = z;
        F[1][1] = w;
    }

    /* Optimized version of power() in method 4 */
    static void power(int F[], int n)
    {
        if (n == 0 || n == 1)
            return;
        int M[] = new int[]{(1,1),(1,0)};

        power(F, n/2);
        multiply(F, F);

        if (n%2 != 0)
            multiply(F, M);
    }

    /* Driver program to test above function */
    public static void main (String args[])
    {
        int n = 9;
        System.out.println(fib(n));
    }
}
/* This code is contributed by Rajat Mishra */

```

Time Complexity: O(Logn)

Extra Space: O(Logn) if we consider the function call stack size, otherwise O(1).

Method 6 (O(Log n) Time)

Below is one more interesting recurrence formula that can be used to find n'th Fibonacci Number in O(Log n) time.

```

If n is even then k = n/2:
F(n) = [2*F(k-1) + F(k)]*F(k)

If n is odd then k = (n + 1)/2
F(n) = F(k)*F(k) + F(k-1)*F(k-1)

```

How does this formula work?

The formula can be derived from above matrix equation.



Taking determinant on both sides, we get

$$(-1)^n = F_{n+1}F_{n-1} - F_n^2$$

Moreover, since $A^n A^m = A^{n+m}$ for any square matrix A, the following identities can be derived (they are obtained from two different coefficients of the matrix product)

$$F_m F_n + F_{m-1} F_{n-1} = F_{m+n-1}$$

By putting $n = n+1$,

$$F_m F_{n+1} + F_{m-1} F_n = F_{m+n}$$

Putting $m = n$

$$F_{2n-1} = F_n^2 + F_{n-1}^2$$

$$F_{2n} = (F_{n-1} + F_{n+1})F_n = (2F_{n-1} + F_n)F_n \text{ (Source: Wiki)}$$

To get the formula to be proved, we simply need to do following

If n is even, we can put k = n/2

If n is odd, we can put k = (n+1)/2

Below is C++ implementation of above idea.

```
// C++ Program to find n'th fibonacci Number in
// with O(Log n) arithmetic operations
#include <bits/stdc++.h>
using namespace std;

const int MAX = 1000;

// Create an array for memoization
int f[MAX] = {0};

// Returns n'th fibonaccci number using table f[]
int fib(int n)
{
    // Base cases
    if (n == 0)
        return 0;
    if (n == 1 || n == 2)
        return (f[n] = 1);

    // If fib(n) is already computed
    if (f[n])
        return f[n];

    int k = (n & 1) ? (n+1)/2 : n/2;

    // Applying above formula [Note value n&1 is 1
    // if n is odd, else 0.
    f[n] = (n & 1)? (fib(k)*fib(k) + fib(k-1)*fib(k-1))
                  : (2*fib(k-1) + fib(k))*fib(k);

    return f[n];
}

/* Driver program to test above function */
int main()
{
    int n = 9;
    printf("%d ", fib(n));
    return 0;
}
```

Output :

34

Time complexity of this solution is O(Log n) as we divide the problem to half in every recursive call.

This method is contributed by Chirag Agarwal.

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

References:

http://en.wikipedia.org/wiki/Fibonacci_number
<http://www.ics.uci.edu/~eppstein/161/960109.html>

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Mathematical
Dynamic Programming
Fibonacci
MathematicalAlgo
series

Average of a stream of numbers

Difficulty Level: Rookie

Given a stream of numbers, print average (or mean) of the stream at every point. For example, let us consider the stream as 10, 20, 30, 40, 50, 60, ...

```
Average of 1 numbers is 10.00
Average of 2 numbers is 15.00
Average of 3 numbers is 20.00
Average of 4 numbers is 25.00
Average of 5 numbers is 30.00
Average of 6 numbers is 35.00
.....
```

To print mean of a stream, we need to find out how to find average when a new number is being added to the stream. To do this, all we need is count of numbers seen so far in the stream, previous average and new number. Let n be the count, prev_avg be the previous average and x be the new number being added. The average after including x number can be written as $(prev_avg * n + x) / (n+1)$.

```
#include <stdio.h>

// Returns the new average after including x
float getAvg(float prev_avg, int x, int n)
{
    return (prev_avg * n + x) / (n + 1);
}

// Prints average of a stream of numbers
void streamAvg(float arr[], int n)
{
    float avg = 0;
    for (int i = 0; i < n; i++)
    {
        avg = getAvg(avg, arr[i], i);
    }
}
```

```

    printf("Average of %d numbers is %f\n", i+1, avg);
}
return;
}

// Driver program to test above functions
int main()
{
    float arr[] = {10, 20, 30, 40, 50, 60};
    int n = sizeof(arr)/sizeof(arr[0]);
    streamAvg(arr, n);

    return 0;
}

```

The above function getAvg() can be optimized using following changes. We can avoid the use of prev_avg and number of elements by using static variables (Assuming that only this function is called for average of stream). Following is the optimized version.

```

#include <stdio.h>

// Returns the new average after including x
float getAvg (int x)
{
    static int sum, n;

    sum += x;
    return ((float)sum)/++n;
}

// Prints average of a stream of numbers
void streamAvg(float arr[], int n)
{
    float avg = 0;
    for(int i = 0; i < n; i++)
    {
        avg = getAvg(arr[i]);
        printf("Average of %d numbers is %f\n", i+1, avg);
    }
    return;
}

// Driver program to test above functions
int main()
{
    float arr[] = {10, 20, 30, 40, 50, 60};
    int n = sizeof(arr)/sizeof(arr[0]);
    streamAvg(arr, n);

    return 0;
}

```

Thanks to Abhijeet Deshpande for suggesting this optimized version.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Mathematical
MathematicalAlgo

Count numbers that don't contain 3

Given a number n, write a function that returns count of numbers from 1 to n that don't contain digit 3 in their decimal representation.

Examples:

```

Input: n = 10
Output: 9

Input: n = 45
Output: 31
// Numbers 3, 13, 23, 30, 31, 32, 33, 34,
// 35, 36, 37, 38, 39, 43 contain digit 3.

Input: n = 578
Output: 385

```

We strongly recommend that you click here and practice it, before moving on to the solution.

Solution:

We can solve it recursively. Let count(n) be the function that counts such numbers.

```

'msd'--> the most significant digit in n
'd' --> number of digits in n.

count(n) = n if n < 10 and msd is not 3
count(n) = count(msd * (10^(d-1)) - 1)
           if n > 10 and msd is 3

```

Let us understand the solution with n = 578.
 $\text{count}(578) = 4 * \text{count}(99) + 4 + \text{count}(78)$
The middle term 4 is added to include numbers 100, 200, 400 and 500.

Let us take n = 35 as another example.
 $\text{count}(35) = \text{count}(3 * 10 - 1) = \text{count}(29)$

```

#include <stdio.h>

/* returns count of numbers which are in range from 1 to n and don't contain 3
   as a digit */
int count(int n)
{
    // Base cases (Assuming n is not negative)
    if (n < 3)
        return n;
    if (n >= 3 && n < 10)
        return n-1;

    // Calculate 10^(d-1) (10 raise to the power d-1) where d is
    // number of digits in n. po will be 100 for n = 578
    int po = 1;
    while (n/po > 9)
        po = po*10;

    // find the most significant digit (msd is 5 for 578)
    int msd = n/po;

    if (msd != 3)
        // For 578, total will be 4*count(10^2 - 1) + 4 + count(78)
        return count(msd)*count(po - 1) + count(msd) + count(n%po);
    else
        // For 35, total will be equal to count(29)
        return count(msd*po - 1);
}

// Driver program to test above function
int main()
{
    printf ("%d ", count(578));
    return 0;
}

```

Output:

385

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Mathematical
MathematicalAlgo

Magic Square

A [magic square](#) of order n is an arrangement of n^2 numbers, usually distinct integers, in a square, such that the n numbers in all rows, all columns, and both diagonals sum to the same constant. A magic square contains the integers from 1 to n^2 .

The constant sum in every row, column and diagonal is called the [magic constant or magic sum](#), M . The magic constant of a normal magic square depends only on n and has the following value:
 $M = n(n^2+1)/2$

For normal magic squares of order $n = 3, 4, 5, \dots$, the magic constants are: 15, 34, 65, 111, 175, 260, ...

In this post, we will discuss how programmatically we can generate a magic square of size n . Before we go further, consider the below examples:

Magic Square of size 3

```

2 7 6
9 5 1
4 3 8
Sum in each row & each column = 3*(3^2+1)/2 = 15

```

Magic Square of size 5

```

9 3 22 16 15
2 21 20 14 8
25 19 13 7 1
18 12 6 5 24
11 10 4 23 17
Sum in each row & each column = 5*(5^2+1)/2 = 65

```

Magic Square of size 7

```

20 12 4 45 37 29 28
11 3 44 36 35 27 19
2 43 42 34 26 18 10
49 41 33 25 17 9 1
40 32 24 16 8 7 48
31 23 15 14 6 47 39
22 21 13 5 46 38 30
Sum in each row & each column = 7*(7^2+1)/2 = 175

```

Did you find any pattern in which the numbers are stored?

In any magic square, the first number i.e. 1 is stored at position $(n/2, n-1)$. Let this position be (i,j) . The next number is stored at position $(i-1, j+1)$ where we can consider each row & column as circular array i.e. they wrap around.

Three conditions hold:

1. The position of next number is calculated by decrementing row number of previous number by 1, and incrementing the column number of previous number by 1. At any time, if the calculated row position becomes -1, it will wrap around to $n-1$. Similarly, if the calculated column position becomes n , it will wrap around to 0.
2. If the magic square already contains a number at the calculated position, calculated column position will be decremented by 2, and calculated row position will be incremented by 1.
3. If the calculated row position is -1 & calculated column position is n , the new position would be: $(0, n-2)$.

Example:

Magic Square of size 3

```

-----
2 7 6
9 5 1
4 3 8

Steps:
1. position of number 1 = (3/2, 3-1) = (1, 2)
2. position of number 2 = (1-1, 2+1) = (0, 0)
3. position of number 3 = (0-1, 0+1) = (3-1, 1) = (2, 1)
4. position of number 4 = (2-1, 1+1) = (1, 2)
Since, at this position, 1 is there. So, apply condition 2.
new position=(1+1,2-2)=(2,0)
5. position of number 5=(2-1,0+1)=(1,1)
6. position of number 6=(1-1,1+1)=(0,2)
7. position of number 7 = (0-1, 2+1) = (-1,3) // this is tricky, see condition 3
new position = (0, 3-2) = (0,1)
8. position of number 8=(0-1,1+1)=(-1,2)=(2,2) //wrap around
9. position of number 9=(2-1,2+1)=(1,3)=(1,0) //wrap around

```

Based on the above approach, following is the working code:

```

#include<stdio.h>
#include<string.h>

// A function to generate odd sized magic squares
void generateSquare(int n)
{
    int magicSquare[n][n];

    // set all slots as 0
    memset(magicSquare, 0, sizeof(magicSquare));

    // Initialize position for 1
    int i = n/2;
    int j = n-1;

    // One by one put all values in magic square
    for (int num=1; num <= n*n;)
    {
        if (i== -1 && j==n) //3rd condition
        {
            j = n-2;
            i = 0;
        }
        else
        {
            //1st condition helper if next number goes to out of square's right side
            if (j == n)
                j = 0;
            //1st condition helper if next number is goes to out of square's upper side
            if (i < 0)
                i=n-1;
        }
        if (magicSquare[i][j]) //2nd condition
        {
            j-=2;
            i++;
            continue;
        }
        else
            magicSquare[i][j] = num++; //set number

        j++; i--; //1st condition
    }

    // print magic square
    printf("The Magic Square for n=%d:\nSum of each row or column %d:\n\n",
        n, n*(n*n+1)/2);
    for(i=0; i<n; i++)
    {
        for(j=0; j<n; j++)
            printf("%3d ", magicSquare[i][j]);
        printf("\n");
    }
}

// Driver program to test above function
int main()
{
    int n = 7; // Works only when n is odd
    generateSquare (n);
    return 0;
}

```

Output:

```

The Magic Square for n=7:
Sum of each row or column 175:

20 12  4 45 37 29 28
11  3 44 36 35 27 19
2 43 42 34 26 18 10
49 41 33 25 17 9  1
40 32 24 16 8  7 48
31 23 15 14 6  47 39
22 21 13 5  46 38 30

```

NOTE: This approach works only for odd values of n.

References:

http://en.wikipedia.org/wiki/Magic_square

This article is compiled by **Aashish Barnwal** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Sieve of Eratosthenes

Given a number n , print all primes smaller than or equal to n . It is also given that n is a small number.

For example, if n is 10, the output should be "2, 3, 5, 7". If n is 20, the output should be "2, 3, 5, 7, 11, 13, 17, 19".

The sieve of Eratosthenes is one of the most efficient ways to find all primes smaller than n when n is smaller than 10 million or so (Ref Wiki).

We strongly recommend that you click here and practice it, before moving on to the solution.

Following is the algorithm to find all the prime numbers less than or equal to a given integer n by Eratosthenes' method:

1. Create a list of consecutive integers from 2 to n : (2, 3, 4, ..., n).
2. Initially, let p equal 2, the first prime number.
3. Starting from p , count up in increments of p and mark each of these numbers greater than p itself in the list. These numbers will be $2p, 3p, 4p$, etc.; note that some of them may have already been marked.
4. Find the first number greater than p in the list that is not marked. If there was no such number, stop. Otherwise, let p now equal this number (which is the next prime), and repeat from step 3.

When the algorithm terminates, all the numbers in the list that are not marked are prime.

Explanation with Example:

Let us take an example when $n = 50$. So we need to print all prime numbers smaller than or equal to 50.

We create a list of all numbers from 2 to 50.

Sieve1



According to the algorithm we will mark all the numbers which are divisible by 2.

sieve2



Now we move to our next unmarked number 3 and mark all the numbers which are multiples of 3.

sieve3



We move to our next unmarked number 5 and mark all multiples of 5.

Sieve4



We continue this process and our final table will look like below:

Sieve5



So the prime numbers are the unmarked ones: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47.

Thanks to **Krishan Kumar** for providing above explanation.

Implementation:

Following is C++ implementation of the above algorithm. In the following implementation, a boolean array arr[] of size n is used to mark multiples of prime numbers.

C/C++

```
// C++ program to print all primes smaller than or equal to
// n using Sieve of Eratosthenes
#include <bits/stdc++.h>
using namespace std;

void SieveOfEratosthenes(int n)
{
    // Create a boolean array "prime[0..n]" and initialize
    // all entries it as true. A value in prime[i] will
    // finally be false if i is Not a prime, else true.
    bool prime[n+1];
    memset(prime, true, sizeof(prime));

    for (int p=2; p*p<=n; p++)
    {
        // If prime[p] is not changed, then it is a prime
        if (prime[p] == true)
        {
            // Update all multiples of p
            for (int i=p*p; i<=n; i += p)
                prime[i] = false;
        }
    }

    // Print all prime numbers
    for (int p=2; p<=n; p++)
        if (prime[p])
            cout << p << " ";
}
```

```

}

// Driver Program to test above function
int main()
{
    int n = 30;
    cout << "Following are the prime numbers smaller "
        << "than or equal to " << n << endl;
    SieveOfEratosthenes(n);
    return 0;
}

```

Java

```

// Java program to print all primes smaller than or equal to
// n using Sieve of Eratosthenes

class SieveOfEratosthenes
{
    void sieveOfEratosthenes(int n)
    {
        // Create a boolean array "prime[0..n]" and initialize
        // all entries it as true. A value in prime[i] will
        // finally be false if i is Not a prime, else true.
        boolean prime[] = new boolean[n+1];
        for(int i=0;i<n;i++)
            prime[i] = true;

        for(int p = 2; p*p <=n; p++)
        {
            // If prime[p] is not changed, then it is a prime
            if(prime[p] == true)
            {
                // Update all multiples of p
                for(int i = p*2; i <= n; i += p)
                    prime[i] = false;
            }
        }

        // Print all prime numbers
        for(int i = 2; i <= n; i++)
        {
            if(prime[i] == true)
                System.out.print(i + " ");
        }
    }

    // Driver Program to test above function
    public static void main(String args[])
    {
        int n = 30;
        System.out.print("Following are the prime numbers ");
        System.out.println("smaller than or equal to " + n);
        SieveOfEratosthenes g = new SieveOfEratosthenes();
        g.sieveOfEratosthenes(n);
    }
}

// This code has been contributed by Amit Khandelwal.

```

Python

```

# Python program to print all primes smaller than or equal to
# n using Sieve of Eratosthenes

def SieveOfEratosthenes(n):

    # Create a boolean array "prime[0..n]" and initialize
    # all entries it as true. A value in prime[i] will
    # finally be false if i is Not a prime, else true.
    prime = [True for i in range(n+1)]
    p=2
    while(p * p <= n):

        # If prime[p] is not changed, then it is a prime
        if (prime[p] == True):

            # Update all multiples of p
            for i in range(p * 2, n+1, p):
                prime[i] = False
            p+=1
    lis = []

    # Print all prime numbers
    for p in range(2, n):
        if prime[p]:
            print(p)

    # driver program
if __name__=='__main__':
    n = 30
    print("Following are the prime numbers smaller",
    print("than or equal to", n)
    SieveOfEratosthenes(n)

```

Output:

```

Following are the prime numbers below 30
2 3 5 7 11 13 17 19 23 29

```

You may also like to see :

Segmented Sieve.

Sieve of Eratosthenes in O(n) time complexity

References:

http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

This article is compiled by **Abhinav Priyadarshi** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Mathematical
MathematicalAlgo
prime-number
sieve

Find day of the week for a given date

Write a function that calculates the day of the week for any particular date in the past or future. A typical application is to calculate the day of the week on which someone was born or some other special event occurred.

Following is a simple C function suggested by [Sakamoto, Lachman, Keith and Craver](#) to calculate day. The following function returns 0 for Sunday, 1 for Monday, etc.

```
/* A program to find day of a given date */
#include<stdio.h>

int dayofweek(int d, int m, int y)
{
    static int t[] = { 0, 3, 2, 5, 0, 3, 5, 1, 4, 6, 2, 4 };
    y -= m < 3;
    return ( y + y/4 - y/100 + y/400 + t[m-1] + d ) % 7;
}

/* Driver function to test above function */
int main()
{
    int day = dayofweek(30, 8, 2010);
    printf ("%d", day);

    return 0;
}
```

Output: 1 (Monday)

See this for explanation of the above function.

References:

http://en.wikipedia.org/wiki/Determination_of_the_day_of_the_week

This article is compiled by **Dheeraj Jain** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Mathematical
MathematicalAlgo

DFA based division

Deterministic Finite Automaton (DFA) can be used to check whether a number "num" is divisible by "k" or not. If the number is not divisible, remainder can also be obtained using DFA.

We consider the binary representation of 'num' and build a DFA with k states. The DFA has transition function for both 0 and 1. Once the DFA is built, we process 'num' over the DFA to get remainder.

Let us walk through an example. Suppose we want to check whether a given number 'num' is divisible by 3 or not. Any number can be written in the form: num = 3*a + b where 'a' is the quotient and 'b' is the remainder.

For 3, there can be 3 states in DFA, each corresponding to remainder 0, 1 and 2. And each state can have two transitions corresponding 0 and 1 (considering the binary representation of given 'num').

The transition function F(p, x) = q tells that on reading alphabet x, we move from state p to state q. Let us name the states as 0, 1 and 2. The initial state will always be 0. The final state indicates the remainder. If the final state is 0, the number is divisible.



In the above diagram, double circled state is final state.

1. When we are at state 0 and read 0, we remain at state 0.
2. When we are at state 0 and read 1, we move to state 1, why? The number so formed(1) in decimal gives remainder 1.
3. When we are at state 1 and read 0, we move to state 2, why? The number so formed(10) in decimal gives remainder 2.
4. When we are at state 1 and read 1, we move to state 0, why? The number so formed(11) in decimal gives remainder 0.
5. When we are at state 2 and read 0, we move to state 1, why? The number so formed(100) in decimal gives remainder 1.
6. When we are at state 2 and read 1, we remain at state 2, why? The number so formed(101) in decimal gives remainder 2.

The transition table looks like following:

state	0	1
0	0 1	
1	2 0	
2	1 2	

Let us check whether 6 is divisible by 3?

Binary representation of 6 is 110

state = 0

1. state=0, we read 1, new state=1

2. state=1, we read 1, new state=0

3. state=0, we read 0, new state=0

Since the final state is 0, the number is divisible by 3.

Let us take another example number as 4

- state=0
- 1. state=0, we read 1, new state=1
- 2. state=1, we read 0, new state=2
- 3. state=2, we read 0, new state=1

Since, the final state is not 0, the number is not divisible by 3. The remainder is 1.

Note that the final state gives the remainder.

We can extend the above solution for any value of k. For a value k, the states would be 0, 1, ..., k-1. How to calculate the transition if the decimal equivalent of the binary bits seen so far, crosses the range k? If we are at state p, we have read p (in decimal). Now we read 0, new read number becomes 2^p . If we read 1, new read number becomes $2^p + 1$. The new state can be obtained by subtracting k from these values ($2p$ or $2p+1$) where $0 \leq p < k$.

Based on the above approach, following is the working code:

```
#include <stdio.h>
#include <stdlib.h>

// Function to build DFA for divisor k
void preprocess(int k, int Table[2][2])
{
    int trans0, trans1;

    // The following loop calculates the two transitions for each state,
    // starting from state 0
    for (int state=0; state<k; ++state)
    {
        // Calculate next state for bit 0
        trans0 = state<<1;
        Table[state][0] = (trans0 < k)? trans0: trans0-k;

        // Calculate next state for bit 1
        trans1 = (state<<1) + 1;
        Table[state][1] = (trans1 < k)? trans1: trans1-k;
    }
}

// A recursive utility function that takes a 'num' and DFA (transition
// table) as input and process 'num' bit by bit over DFA
void isDivisibleUtil(int num, int* state, int Table[2][2])
{
    // process "num" bit by bit from MSB to LSB
    if (num != 0)
    {
        isDivisibleUtil(num>>1, state, Table);
        *state = Table[*state][num&1];
    }
}

// The main function that divides 'num' by k and returns the remainder
int isDivisible (int num, int k)
{
    // Allocate memory for transition table. The table will have k*2 entries
    int (*Table)[2] = (int (*)[2])malloc(k*sizeof(*Table));

    // Fill the transition table
    preprocess(k, Table);

    // Process 'num' over DFA and get the remainder
    int state = 0;
    isDivisibleUtil(num, &state, Table);

    // Note that the final value of state is the remainder
    return state;
}

// Driver program to test above functions
int main()
{
    int num = 47; // Number to be divided
    int k = 5; // Divisor

    int remainder = isDivisible (num, k);

    if (remainder == 0)
        printf("Divisible\n");
    else
        printf("Not Divisible: Remainder is %d\n", remainder);

    return 0;
}
```

Output:

```
Not Divisible: Remainder is 2
```

DFA based division can be useful if we have a binary stream as input and we want to check for divisibility of the decimal value of stream at any time.

Related Articles:

- [Check divisibility in a binary stream](#)
- [Check if a stream is Multiple of 3](#)

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Generate integer from 1 to 7 with equal probability

Given a function `foo()` that returns integers from 1 to 5 with equal probability, write a function that returns integers from 1 to 7 with equal probability using `foo()` only. Minimize the number of calls to `foo()` method. Also, use of any other library function is not allowed and no floating point arithmetic allowed.

Solution:

We know `foo()` returns integers from 1 to 5. How we can ensure that integers from 1 to 7 occur with equal probability?

If we somehow generate integers from 1 to a-multiple-of-7 (like 7, 14, 21, ...) with equal probability, we can use modulo division by 7 followed by adding 1 to get the numbers from 1 to 7 with equal probability.

We can generate from 1 to 21 with equal probability using the following expression.

```
5*foo() + foo() - 5
```

Let us see how above expression can be used.

1. For each value of first `foo()`, there can be 5 possible combinations for values of second `foo()`. So, there are total 25 combinations possible.
2. The range of values returned by the above equation is 1 to 25, each integer occurring exactly once.
3. If the value of the equation comes out to be less than 22, return modulo division by 7 followed by adding 1. Else, again call the method recursively. The probability of returning each integer thus becomes 1/7.

The below program shows that the expression returns each integer from 1 to 25 exactly once.

```
#include <stdio.h>

int main()
{
    int first, second;
    for (first=1; first<=5; ++first)
        for (second=1; second<=5; ++second)
            printf ("%d\n", 5*first + second - 5);
    return 0;
}
```

Output:

```
1
2
.
.
24
25
```

The below program depicts how we can use `foo()` to return 1 to 7 with equal probability.

```
#include <stdio.h>

int foo() // given method that returns 1 to 5 with equal probability
{
    // some code here
}

int my_rand() // returns 1 to 7 with equal probability
{
    int i;
    i = 5*foo() + foo() - 5;
    if (i < 22)
        return i%7 + 1;
    return my_rand();
}

int main()
{
    printf ("%d", my_rand());
    return 0;
}
```

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Randomized
MathematicalAlgo

Given a number, find the next smallest palindrome

Given a number, find the next smallest palindrome larger than this number. For example, if the input number is "2 3 5 4 5", the output should be "2 3 6 3 2". And if the input number is "9 9 9", the output should be "1 0 0 1".

The input is assumed to be an array. Every entry in array represents a digit in input number. Let the array be '`num[]`' and size of array be '`n`'

There can be three different types of inputs that need to be handled separately.

- 1) The input number is palindrome and has all 9s. For example "9 9 9". Output should be "1 0 0 1"
- 2) The input number is not palindrome. For example "1 2 3 4". Output should be "1 3 3 1"
- 3) The input number is palindrome and doesn't have all 9s. For example "1 2 2 1". Output should be "1 3 3 1".

We strongly recommend that you click here and practice it, before moving on to the solution.

Solution for input type 1 is easy. The output contains $n + 1$ digits where the corner digits are 1, and all digits between corner digits are 0.

Now let us first talk about input type 2 and 3. How to convert a given number to a greater palindrome? To understand the solution, let us first define the following two terms:

Left Side: The left half of given number. Left side of "1 2 3 4 5 6" is "1 2 3" and left side of "1 2 3 4 5" is "1 2"

Right Side: The right half of given number. Right side of "1 2 3 4 5 6" is "4 5 6" and right side of "1 2 3 4 5" is "4 5"

To convert to palindrome, we can either take the mirror of its left side or take mirror of its right side. However, if we take the mirror of the right side, then the palindrome so formed is not guaranteed to be next larger palindrome. So, we must take the mirror of left side and copy it to right side. But there are some cases that must be handled in different ways. See the following steps.

We will start with two indices i and j. i pointing to the two middle elements (or pointing to two elements around the middle element in case of n being odd). We one by one move i and j away from each other.

Step 1. Initially, ignore the part of left side which is same as the corresponding part of right side. For example, if the number is "8 3 4 2 2 4 6 9", we ignore the middle four elements. i now points to element 3 and j now points to element 6.

Step 2. After step 1, following cases arise:

Case 1: Indices i & j cross the boundary.

This case occurs when the input number is palindrome. In this case, we just add 1 to the middle digit (or digits in case n is even) propagate the carry towards MSB digit of left side and simultaneously copy mirror of the left side to the right side.

For example, if the given number is "1 2 9 2 1", we increment 9 to 10 and propagate the carry. So the number becomes "1 3 0 3 1"

Case 2: There are digits left between left side and right side which are not same. So, we just mirror the left side to the right side & try to minimize the number formed to guarantee the next smallest palindrome.

In this case, there can be **two sub-cases**.

2.1) Copying the left side to the right side is sufficient, we don't need to increment any digits and the result is just mirror of left side. Following are some examples of this sub-case.

Next palindrome for "7 8 3 3 2 2" is "7 8 3 3 8 7"

Next palindrome for "1 2 5 3 2 2" is "1 2 5 5 2 1"

Next palindrome for "1 4 5 8 7 6 7 8 3 2 2" is "1 4 5 8 7 6 7 8 5 4 1"

How do we check for this sub-case? All we need to check is the digit just after the ignored part in step 1. This digit is highlighted in above examples. If this digit is greater than the corresponding digit in right side digit, then copying the left side to the right side is sufficient and we don't need to do anything else.

2.2) Copying the left side to the right side is NOT sufficient. This happens when the above defined digit of left side is smaller. Following are some examples of this case.

Next palindrome for "7 1 3 3 2 2" is "7 1 4 4 1 7"

Next palindrome for "1 2 3 4 6 2 8" is "1 2 3 5 3 2 1"

Next palindrome for "9 4 1 8 7 9 7 8 3 2 2" is "9 4 1 8 8 0 8 8 1 4 9"

We handle this subcase like Case 1. We just add 1 to the middle digit (or digits in case n is even) propagate the carry towards MSB digit of left side and simultaneously copy mirror of the left side to the right side.

```
#include <stdio.h>

// A utility function to print an array
void printArray(int arr[], int n);

// A utility function to check if num has all 9s
int AreAll9s(int num[], int n);

// Returns next palindrome of a given number num[].
// This function is for input type 2 and 3
void generateNextPalindromeUtil (int num[], int n)
{
    // find the index of mid digit
    int mid = n/2;

    // A bool variable to check if copy of left side to right is sufficient or not
    bool leftsmaller = false;

    // end of left side is always 'mid -1'
    int i = mid - 1;

    // Beginning of right side depends if n is odd or even
    int j = (n % 2)? mid + 1 : mid;

    // Initially, ignore the middle same digits
    while (i >= 0 && num[i] == num[j])
        i--;
        j++;

    // Find if the middle digit(s) need to be incremented or not (or copying left
    // side is not sufficient)
    if ( i < 0 || num[i] < num[j])
        leftsmaller = true;

    // Copy the mirror of left to right
    while (i >= 0)
    {
        num[j] = num[i];
        j++;
        i--;
    }

    // Handle the case where middle digit(s) must be incremented.
    // This part of code is for CASE 1 and CASE 2.2
    if (leftsmaller == true)
    {
        int carry = 1;
        i = mid - 1;

        // If there are odd digits, then increment
        // the middle digit and store the carry
        if (n%2 == 1)
        {
            num[mid] += carry;
            carry = num[mid] / 10;
            num[mid] %= 10;
            j = mid + 1;
        }
        else
            j = mid;

        // Add 1 to the rightmost digit of the left side, propagate the carry
        // towards MSB digit and simultaneously copying mirror of the left side
        // to the right side.
        while (i >= 0)
        {
            num[i] += carry;
            carry = num[i] / 10;
            num[i] %= 10;
            num[i+1] = num[i-1]; // copy mirror to right
        }
    }

    // The function that prints next palindrome of a given number num[]
}
```

```

// with n digits.
void generateNextPalindrome( int num[], int n )
{
    int i;

    printf("\nNext palindrome is:\n");

    // Input type 1: All the digits are 9, simply o/p 1
    // followed by n-1 0's followed by 1.
    if( AreAll9s( num, n ) )
    {
        printf(" 1 ");
        for( i = 1; i < n; i++ )
            printf(" 0 ");
        printf(" 1 ");
    }

    // Input type 2 and 3
    else
    {
        generateNextPalindromeUtil ( num, n );

        // print the result
        printArray (num, n);
    }
}

// A utility function to check if num has all 9s
int AreAll9s( int* num, int n )
{
    int i;
    for( i = 0; i < n; ++i )
        if( num[i] != 9 )
            return 0;
    return 1;
}

/* Utility that prints out an array on a line */
void printArray(int arr[], int n)
{
    int i;
    for (i=0;i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver Program to test above function
int main()
{
    int num[] = {9, 4, 1, 8, 7, 9, 7, 8, 3, 2, 2};

    int n = sizeof (num)/ sizeof(num[0]);

    generateNextPalindrome( num, n );

    return 0;
}

```

Output:

```

Next palindrome is:
9 4 1 8 8 0 8 8 1 4 9

```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Mathematical
MathematicalAlgo

Make a fair coin from a biased coin

You are given a function `foo()` that represents a biased coin. When `foo()` is called, it returns 0 with 60% probability, and 1 with 40% probability. Write a new function that returns 0 and 1 with 50% probability each. Your function should use only `foo()`, no other library method.

Solution:

We know `foo()` returns 0 with 60% probability. How can we ensure that 0 and 1 are returned with 50% probability?

The solution is similar to [this post](#). If we can somehow get two cases with equal probability, then we are done. We call `foo()` two times. Both calls will return 0 with 60% probability. So the two pairs (0, 1) and (1, 0) will be generated with equal probability from two calls of `foo()`. Let us see how.

(0, 1): The probability to get 0 followed by 1 from two calls of `foo()` = $0.6 * 0.4 = 0.24$

(1, 0): The probability to get 1 followed by 0 from two calls of `foo()` = $0.4 * 0.6 = 0.24$

So the two cases appear with equal probability. The idea is to return consider only the above two cases, return 0 in one case, return 1 in other case. For other cases [(0, 0) and (1, 1)], recur until you end up in any of the above two cases.

The below program depicts how we can use `foo()` to return 0 and 1 with equal probability.

```

#include <stdio.h>

int foo() // given method that returns 0 with 60% probability and 1 with 40%
{
    // some code here
}

// returns both 0 and 1 with 50% probability
int my_fun()
{
    int val1 = foo();
    int val2 = foo();
    if (val1 == 0 && val2 == 1)
        return 0; // Will reach here with 0.24 probability
    if (val1 == 1 && val2 == 0)
        return 1; // Will reach here with 0.24 probability
}

```

```

    return 1; // Will reach here with 0.24 probability
    return my_fun(); // will reach here with (1 - 0.24 - 0.24) probability
}

int main()
{
    printf ("%d ",my_fun());
    return 0;
}

```

References:

http://en.wikipedia.org/wiki/Fair_coin#Fair_results_from_a_biased_coin

This article is compiled by **Shashank Sinha** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

GATE CS Corner Company Wise Coding Practice

Randomized
MathematicalAlgo

Check divisibility by 7

Given a number, check if it is divisible by 7. You are not allowed to use modulo operator, floating point arithmetic is also not allowed.

A simple method is repeated subtraction. Following is another interesting method.

Divisibility by 7 can be checked by a recursive method. A number of the form $10a + b$ is divisible by 7 if and only if $a - 2b$ is divisible by 7. In other words, subtract twice the last digit from the number formed by the remaining digits. Continue to do this until a small number.

Example: the number 371: $37 - (2 \times 1) = 37 - 2 = 35$; $3 - (2 \times 5) = 3 - 10 = -7$; thus, since -7 is divisible by 7, 371 is divisible by 7.

Following is C implementation of the above method

```

// A Program to check whether a number is divisible by 7
#include <stdio.h>

int isDivisibleBy7( int num )
{
    // If number is negative, make it positive
    if( num < 0 )
        return isDivisibleBy7( -num );

    // Base cases
    if( num == 0 || num == 7 )
        return 1;
    if( num < 10 )
        return 0;

    // Recur for ( num / 10 - 2 * num % 10 )
    return isDivisibleBy7( num / 10 - 2 * ( num - num / 10 * 10 ) );
}

// Driver program to test above function
int main()
{
    int num = 616;
    if( isDivisibleBy7(num) )
        printf("Divisible");
    else
        printf("Not Divisible");
    return 0;
}

```

Output:

Divisible

How does this work? Let 'b' be the last digit of a number 'n' and let 'a' be the number we get when we split off 'b'.

The representation of the number may also be multiplied by any number relatively prime to the divisor without changing its divisibility. After observing that 7 divides 21, we can perform the following:

10.a + b

after multiplying by 2, this becomes

20.a + 2.b

and then

21.a - a + 2.b

Eliminating the multiple of 21 gives

-a + 2b

and multiplying by -1 gives

a - 2b

There are other interesting methods to check divisibility by 7 and other numbers. See following Wiki page for details.

References:

http://en.wikipedia.org/wiki/Divisibility_rule

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Find the largest multiple of 3

Given an array of non-negative integers. Find the largest multiple of 3 that can be formed from array elements.

For example, if the input array is {8, 1, 9}, the output should be "9 8 1", and if the input array is {8, 1, 7, 6, 0}, output should be "8 7 6 0".

Method 1 (Brute Force)

The simple & straight forward approach is to generate all the combinations of the elements and keep track of the largest number formed which is divisible by 3.

Time Complexity: $O(n \times 2^n)$. There will be 2^n combinations of array elements. To compare each combination with the largest number so far may take $O(n)$ time.

Auxiliary Space: $O(n)$ // to avoid integer overflow, the largest number is assumed to be stored in the form of array.

Method 2 (Tricky)

This problem can be solved efficiently with the help of $O(n)$ extra space. This method is based on the following facts about numbers which are multiple of 3.

1) A number is multiple of 3 if and only if the sum of digits of number is multiple of 3. For example, let us consider 8760, it is a multiple of 3 because sum of digits is $8 + 7 + 6 + 0 = 21$, which is a multiple of 3.

2) If a number is multiple of 3, then all permutations of it are also multiple of 3. For example, since 6078 is a multiple of 3, the numbers 8760, 7608, 7068, are also multiples of 3.

3) We get the same remainder when we divide the number and sum of digits of the number. For example, if divide number 151 and sum of its digits 7, by 3, we get the same remainder 1.

What is the idea behind above facts?

The value of $10^k \bmod 3$ and $100^k \bmod 3$ is 1. The same is true for all the higher powers of 10, because 3 divides 9, 99, 999, ... etc.

Let us consider a 3 digit number n to prove above facts. Let the first, second and third digits of n be 'a', 'b' and 'c' respectively. n can be written as

$$n = 100.a + 10.b + c$$

Since $(10^k)^m \bmod 3$ is 1 for any k , the above expression gives the same remainder as following expression

$$1.a + 1.b + c$$

So the remainder obtained by sum of digits and ' n ' is same.

Following is a solution based on the above observation.

1. Sort the array in non-decreasing order.

2. Take three queues. One for storing elements which on dividing by 3 gives remainder as 0. The second queue stores digits which on dividing by 3 gives remainder as 1. The third queue stores digits which on dividing by 3 gives remainder as 2. Call them as queue0, queue1 and queue2

3. Find the sum of all the digits.

4. Three cases arise:

.....4.1 The sum of digits is divisible by 3. Dequeue all the digits from the three queues. Sort them in non-increasing order. Output the array.

.....4.2 The sum of digits produces remainder 1 when divided by 3.

Remove one item from queue1. If queue1 is empty, remove two items from queue2. If queue2 contains less than two items, the number is not possible.

.....4.3 The sum of digits produces remainder 2 when divided by 3.

Remove one item from queue2. If queue2 is empty, remove two items from queue1. If queue1 contains less than two items, the number is not possible.

5. Finally empty all the queues into an auxiliary array. Sort the auxiliary array in non-increasing order. Output the auxiliary array.

Based on the above, below is C implementation:

The below code works only if the input arrays has numbers from 0 to 9. It can be easily extended for any positive integer array. We just have to modify the part where we sort the array in decreasing order, at the end of code.

```
/* A program to find the largest multiple of 3 from an array of elements */
#include <stdio.h>
#include <stdlib.h>

// A queue node
typedef struct Queue
{
    int front;
    int rear;
    int capacity;
    int* array;
} Queue;

// A utility function to create a queue with given capacity
Queue* createQueue( int capacity )
{
    Queue* queue = (Queue *) malloc (sizeof(Queue));
    queue->capacity = capacity;
    queue->front = queue->rear = -1;
    queue->array = (int *) malloc (queue->capacity * sizeof(int));
    return queue;
}

// A utility function to check if queue is empty
int isEmpty (Queue* queue)
{
    return queue->front == -1;
}

// A function to add an item to queue
void Enqueue (Queue* queue, int item)
{
    queue->array[ ++queue->rear ] = item;
    if ( isEmpty(queue) )
        ++queue->front;
}

// A function to remove an item from queue
int Dequeue (Queue* queue)
{
    int item = queue->array[ queue->front ];
    if ( queue->front == queue->rear )
        queue->front = queue->rear = -1;
    else

```

```

queue->front++;

return item;
}

// A utility function to print array contents
void printArr( int* arr, int size)
{
    int i;
    for ( i = 0; i < size; ++i)
        printf ("%d ", arr[i]);
}

/* Following two functions are needed for library function qsort().
   Refer following link for help of qsort()
   http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ */
int compareAsc( const void* a, const void* b )
{
    return *(int*)a > *(int*)b;
}
int compareDesc( const void* a, const void* b )
{
    return *(int*)a < *(int*)b;
}

// This function puts all elements of 3 queues in the auxiliary array
void populateAux( int* aux, Queue* queue0, Queue* queue1,
                  Queue* queue2, int* top )
{
    // Put all items of first queue in aux[]
    while ( !isEmpty(queue0) )
        aux[ (*top)++ ] = Dequeue( queue0 );

    // Put all items of second queue in aux[]
    while ( !isEmpty(queue1) )
        aux[ (*top)++ ] = Dequeue( queue1 );

    // Put all items of third queue in aux[]
    while ( !isEmpty(queue2) )
        aux[ (*top)++ ] = Dequeue( queue2 );
}

// The main function that finds the largest possible multiple of
// 3 that can be formed by arr[] elements
int findMaxMultipleOf3( int* arr, int size )
{
    // Step 1: sort the array in non-decreasing order
    qsort( arr, size, sizeof( int ), compareAsc );

    // Create 3 queues to store numbers with remainder 0, 1
    // and 2 respectively
    Queue* queue0 = createQueue( size );
    Queue* queue1 = createQueue( size );
    Queue* queue2 = createQueue( size );

    // Step 2 and 3 get the sum of numbers and place them in
    // corresponding queues
    int i, sum;
    for ( i = 0, sum = 0; i < size; ++i )
    {
        sum += arr[i];
        if ( (arr[i] % 3) == 0 )
            Enqueue( queue0, arr[i] );
        else if ( (arr[i] % 3) == 1 )
            Enqueue( queue1, arr[i] );
        else
            Enqueue( queue2, arr[i] );
    }

    // Step 4.2: The sum produces remainder 1
    if ( (sum % 3) == 1 )
    {
        // either remove one item from queue1
        if ( !isEmpty( queue1 ) )
            Dequeue( queue1 );

        // or remove two items from queue2
        else
        {
            if ( !isEmpty( queue2 ) )
                Dequeue( queue2 );
            else
                return 0;

            if ( !isEmpty( queue2 ) )
                Dequeue( queue2 );
        }
    }

    // Step 4.3: The sum produces remainder 2
    else if ( (sum % 3) == 2 )
    {
        // either remove one item from queue2
        if ( !isEmpty( queue2 ) )
            Dequeue( queue2 );

        // or remove two items from queue1
        else
        {
            if ( !isEmpty( queue1 ) )
                Dequeue( queue1 );
            else
                return 0;

            if ( !isEmpty( queue1 ) )

```

```

        Dequeue( queue1 );
    else
        return 0;
    }

    int aux[size], top = 0;

    // Empty all the queues into an auxiliary array.
    populateAux (aux, queue0, queue1, queue2, &top);

    // sort the array in non-increasing order
    qsort (aux, top, sizeof( int ), compareDesc);

    // print the result
    printArr (aux, top);

    return top;
}

// Driver program to test above functions
int main()
{
    int arr[] = {8, 1, 7, 6, 0};
    int size = sizeof(arr)/sizeof(arr[0]);

    if (findMaxMultipleOf3( arr, size ) == 0)
        printf("Not Possible");

    return 0;
}

```

The above method can be optimized in following ways.

1) We can use Heap Sort or Merge Sort to make the time complexity O(nLogn).

2) We can avoid extra space for queues. We know at most two items will be removed from the input array. So we can keep track of two items in two variables.

3) At the end, instead of sorting the array again in descending order, we can print the ascending sorted array in reverse order. While printing in reverse order, we can skip the two elements to be removed.

Time Complexity: O(nLogn), assuming a O(nLogn) algorithm is used for sorting.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Queue
MathematicalAlgo

Lexicographic rank of a string

Given a string, find its rank among all its permutations sorted lexicographically. For example, rank of "abc" is 1, rank of "acb" is 2, and rank of "cba" is 6.

We strongly recommend that you click here and practice it, before moving on to the solution.

For simplicity, let us assume that the string does not contain any duplicated characters.

One simple solution is to initialize rank as 1, generate all permutations in lexicographic order. After generating a permutation, check if the generated permutation is same as given string, if same, then return rank, if not, then increment the rank by 1. The time complexity of this solution will be exponential in worst case. Following is an efficient solution.

Let the given string be "STRING". In the input string, 'S' is the first character. There are total 6 characters and 4 of them are smaller than 'S'. So there can be $4 * 5!$ smaller strings where first character is smaller than 'S', like following

```
R X X X X X
I X X X X X
N X X X X X
G X X X X X
```

Now let us Fix S' and find the smaller strings starting with 'S'.

Repeat the same process for T, rank is $4*5! + 4*4! + ...$

Now fix T and repeat the same process for R, rank is $4*5! + 4*4! + 3*3! + ...$

Now fix R and repeat the same process for I, rank is $4*5! + 4*4! + 3*3! + 1*2! + ...$

Now fix I and repeat the same process for N, rank is $4*5! + 4*4! + 3*3! + 1*2! + 1*1! + ...$

Now fix N and repeat the same process for G, rank is $4*5! + 4*4! + 3*3! + 1*2! + 1*1! + 0*0!$

Rank = $4*5! + 4*4! + 3*3! + 1*2! + 1*1! + 0*0! = 597$

Since the value of rank starts from 1, the final rank = $1 + 597 = 598$

```

#include <stdio.h>
#include <string.h>

// A utility function to find factorial of n
int fact(int n)
{
    return (n <= 1)? 1 : n * fact(n-1);
}

// A utility function to count smaller characters on right
// of arr[low]
int findSmallerInRight(char* str, int low, int high)
{
    int countRight = 0, i;

    for (i = low+1; i <= high; ++i)
        if (str[i] < str[low])
            ++countRight;
}

```

```

    return countRight;
}

// A function to find rank of a string in all permutations
// of characters
int findRank (char* str)
{
    int len = strlen(str);
    int mul = fact(len);
    int rank = 1;
    int countRight;

    int i;
    for (i = 0; i < len; ++i)
    {
        mul /= len - i;

        // count number of chars smaller than str[i]
        // from str[i+1] to str[len-1]
        countRight = findSmallerInRight(str, i, len-1);

        rank += countRight * mul ;
    }

    return rank;
}

// Driver program to test above function
int main()
{
    char str[] = "string";
    printf ("%d", findRank(str));
    return 0;
}

```

Output

598

The time complexity of the above solution is $O(n^2)$. We can reduce the time complexity to $O(n)$ by creating an auxiliary array of size 256. See following code.

```

// A O(n) solution for finding rank of string
#include <stdio.h>
#include <string.h>
#define MAX_CHAR 256

// A utility function to find factorial of n
int fact(int n)
{
    return (n <= 1)? 1 : n * fact(n-1);
}

// Construct a count array where value at every index
// contains count of smaller characters in whole string
void populateAndIncreaseCount (int* count, char* str)
{
    int i;

    for( i = 0; str[i]; ++i )
        ++count[ str[i] ];

    for( i = 1; i < 256; ++i )
        count[i] += count[i-1];
}

// Removes a character ch from count[] array
// constructed by populateAndIncreaseCount()
void updateCount (int* count, char ch)
{
    int i;
    for( i = ch; i < MAX_CHAR; ++i )
        --count[i];
}

// A function to find rank of a string in all permutations
// of characters
int findRank (char* str)
{
    int len = strlen(str);
    int mul = fact(len);
    int rank = 1, i;
    int count[MAX_CHAR] = {0}; // all elements of count[] are initialized with 0

    // Populate the count array such that count[i] contains count of
    // characters which are present in str and are smaller than i
    populateAndIncreaseCount( count, str );

    for (i = 0; i < len; ++i)
    {
        mul /= len - i;

        // count number of chars smaller than str[i]
        // from str[i+1] to str[len-1]
        rank += count[ str[i] - 1 ] * mul;

        // Reduce count of characters greater than str[i]
        updateCount (count, str[i]);
    }

    return rank;
}

// Driver program to test above function
int main()
{
    char str[] = "string";

```

```

printf ("%d", findRank(str));
return 0;
}

```

The above programs don't work for duplicate characters. To make them work for duplicate characters, find all the characters that are smaller (include equal this time also), do the same as above but, this time divide the rank so formed by p! where p is the count of occurrences of the repeating character.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Strings
MathematicalAlgo

Print all permutations in sorted (lexicographic) order

Given a string, print all permutations of it in sorted order. For example, if the input string is "ABC", then output should be "ABC, ACB, BAC, BCA, CAB, CBA".

We have discussed a program to print all permutations in [this](#) post, but here we must print the permutations in increasing order.

Following are the steps to print the permutations lexicographic-ally

1. Sort the given string in non-decreasing order and print it. The first permutation is always the string sorted in non-decreasing order.

2. Start generating next higher permutation. Do it until next higher permutation is not possible. If we reach a permutation where all characters are sorted in non-increasing order, then that permutation is the last permutation.

Steps to generate the next higher permutation:

1. Take the previously printed permutation and find the rightmost character in it, which is smaller than its next character. Let us call this character as 'first character'.

2. Now find the ceiling of the 'first character'. Ceiling is the smallest character on right of 'first character', which is greater than 'first character'. Let us call the ceil character as 'second character'.

3. Swap the two characters found in above 2 steps.

4. Sort the substring (in non-decreasing order) after the original index of 'first character'.

Let us consider the string "ABCDEF". Let previously printed permutation be "DCFEBA". The next permutation in sorted order should be "DEABCF". Let us understand above steps to find next permutation. The 'first character' will be 'C'. The 'second character' will be 'E'. After swapping these two, we get "DEFCBA". The final step is to sort the substring after the character original index of 'first character'. Finally, we get "DEABCF".

Following is C++ implementation of the algorithm.

```

// Program to print all permutations of a string in sorted order.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Following function is needed for library function qsort(). Refer
 http://www.cplusplus.com/reference/library/cstdlib/qsort/ */
int compare (const void *a, const void * b)
{ return (* (char *)a - *(char *)b);}

// A utility function two swap two characters a and b
void swap (char* a, char* b)
{
    char t = *a;
    *a = *b;
    *b = t;
}

// This function finds the index of the smallest character
// which is greater than 'first' and is present in str[l..h]
int findCeil (char str[], char first, int l, int h)
{
    // initialize index of ceiling element
    int ceilIndex = l;

    // Now iterate through rest of the elements and find
    // the smallest character greater than 'first'
    for (int i = l+1; i <= h; i++)
        if (str[i] > first && str[i] < str[ceilIndex])
            ceilIndex = i;

    return ceilIndex;
}

// Print all permutations of str in sorted order
void sortedPermutations ( char str[])
{
    // Get size of string
    int size = strlen(str);

    // Sort the string in increasing order
    qsort( str, size, sizeof( str[0]), compare );

    // Print permutations one by one
    bool isFinished = false;
    while ( !isFinished )
    {
        // print this permutation
        printf ("%s\n", str);

        // Find the rightmost character which is smaller than its next
        // character. Let us call it 'first char'
        int i;
        for ( i = size - 2; i >= 0; -i )
            if (str[i] < str[i+1])
                break;

        // If there is no such character, all are sorted in decreasing order,
        // means we just printed the last permutation and we are done.
        if ( i == -1 )
            isFinished = true;
        else
    }
}

```

```

// Find the ceil of 'first char' in right of first character.
// Ceiling of a character is the smallest character greater than it
int ceilIndex = findCeil( str, str[i], i + 1, size - 1 );

// Swap first and second characters
swap( &str[i], &str[ceilIndex] );

// Sort the string on right of 'first char'
qsort( str + i + 1, size - i - 1, sizeof(str[0]), compare );
}

}

// Driver program to test above function
int main()
{
    char str[] = "ABCD";
    sortedPermutations( str );
    return 0;
}

```

Output:

```

ABCD
ABDC
...
...
DCAB
DCBA

```

The upper bound on time complexity of the above program is $O(n^2 \times n!)$. We can optimize step 4 of the above algorithm for finding next permutation. Instead of sorting the subarray after the 'first character', we can reverse the subarray, because the subarray we get after swapping is always sorted in non-increasing order. This optimization makes the time complexity as $O(n \times n!)$. See following optimized code.

```

// An optimized version that uses reverse instead of sort for
// finding the next permutation

// A utility function to reverse a string str[l..h]
void reverse(char str[], int l, int h)
{
    while (l < h)
    {
        swap(&str[l], &str[h]);
        l++;
        h--;
    }
}

// Print all permutations of str in sorted order
void sortedPermutations ( char str[] )
{
    // Get size of string
    int size = strlen(str);

    // Sort the string in increasing order
    qsort( str, size, sizeof( str[0] ), compare );

    // Print permutations one by one
    bool isFinished = false;
    while ( !isFinished )
    {
        // print this permutation
        printf ("%s\n", str);

        // Find the rightmost character which is smaller than its next
        // character. Let us call it 'first char'
        int i;
        for ( i = size - 2; i >= 0; -i )
            if (str[i] < str[i+1])
                break;

        // If there is no such character, all are sorted in decreasing order,
        // means we just printed the last permutation and we are done.
        if ( i == -1 )
            isFinished = true;
        else
        {
            // Find the ceil of 'first char' in right of first character.
            // Ceiling of a character is the smallest character greater than it
            int ceilIndex = findCeil( str, str[i], i + 1, size - 1 );

            // Swap first and second characters
            swap( &str[i], &str[ceilIndex] );

            // reverse the string on right of 'first char'
            reverse( str, i + 1, size - 1 );
        }
    }
}

```

The above programs print duplicate permutation when characters are repeated. We can avoid it by keeping track of the previous permutation. While printing, if the current permutation is same as previous permutation, we won't print it.

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Combinatorial
Mathematical
MathematicalAlgo
permutation

Shuffle a given array

Given an array, write a program to generate a random permutation of array elements. This question is also asked as "shuffle a deck of cards" or "randomize a given array".



Let the given array be $arr[]$. A simple solution is to create an auxiliary array $temp[]$ which is initially a copy of $arr[]$. Randomly select an element from $temp[]$, copy the randomly selected element to $arr[0]$ and remove the selected element from $temp[]$. Repeat the same process n times and keep copying elements to $arr[1], arr[2], \dots$. The time complexity of this solution will be $O(n^2)$.

Fisher-Yates shuffle Algorithm works in $O(n)$ time complexity. The assumption here is, we are given a function $rand()$ that generates random number in $O(1)$ time.

The idea is to start from the last element, swap it with a randomly selected element from the whole array (including last). Now consider the array from 0 to $n-2$ (size reduced by 1), and repeat the process till we hit the first element.

Following is the detailed algorithm

```
To shuffle an array a of n elements (indices 0..n-1):
for i from n - 1 downto 1 do
    j = random integer with 0
```

Following is C++ implementation of this algorithm.

```
// C Program to shuffle a given array

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// A utility function to swap two integers
void swap (int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// A utility function to print an array
void printArray (int arr[], int n)
{
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// A function to generate a random permutation of arr[]
void randomize (int arr[], int n)
{
    // Use a different seed value so that we don't get same
    // result each time we run this program
    srand ( time(NULL) );

    // Start from the last element and swap one by one. We don't
    // need to run for the first element that's why i > 0
    for (int i = n-1; i > 0; i--)
    {
        // Pick a random index from 0 to i
        int j = rand() % (i+1);

        // Swap arr[i] with the element at random index
        swap(&arr[i], &arr[j]);
    }
}

// Driver program to test above function.
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};
    int n = sizeof(arr)/sizeof(arr[0]);
    randomize (arr, n);
    printArray(arr, n);

    return 0;
}
```

Output:

```
7 8 4 6 3 1 2 5
```

The above function assumes that $rand()$ generates a random number.

Time Complexity: $O(n)$, assuming that the function $rand()$ takes $O(1)$ time.

How does this work?

The probability that i th element (including the last one) goes to last position is $1/n$, because we randomly pick an element in first iteration.

The probability that i th element goes to second last position can be proved to be $1/n$ by dividing it in two cases.

Case 1: $i = n-1$ (index of last element):

The probability of last element going to second last position is = (probability that last element doesn't stay at its original position) \times (probability that the index picked in previous step is picked again so that the last element

So the probability = $((n-1)/n) \times (1/(n-1)) = 1/n$

Case 2: $0 < i < n-1$:

The probability of i th element going to second position = (probability that i th element is not picked in previous iteration) \times (probability that i th element is picked in this iteration)

So the probability = $((n-1)/n) \times (1/(n-1)) = 1/n$

We can easily generalize above proof for any other position.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner

Company Wise Coding Practice

Randomized
MathematicalAlgo

Space and time efficient Binomial Coefficient

Write a function that takes two parameters n and k and returns the value of Binomial Coefficient $C(n, k)$. For example, your function should return 6 for $n = 4$ and $k = 2$, and it should return 10 for $n = 5$ and $k = 2$.

We have discussed a $O(n*k)$ time and $O(k)$ extra space algorithm in [this post](#). The value of $C(n, k)$ can be calculated in $O(k)$ time and $O(1)$ extra space.

```
C(n, k) = n! / (n-k)! * k!
= [n * (n-1) * ... * 1] / [((n-k) * (n-k-1) * ... * 1) *
                           (k * (k-1) * ... * 1)]
```

After simplifying, we get

```
C(n, k) = [n * (n-1) * ... * (n-k+1)] / [k * (k-1) * ... * 1]
```

```
Also,  $C(n, k) = C(n, n-k)$  // we can change r to n-r if r > n-r
```

Following implementation uses above formula to calculate $C(n, k)$

```
// Program to calculate C(n ,k)
#include <stdio.h>

// Returns value of Binomial Coefficient C(n, k)
int binomialCoeff(int n, int k)
{
    int res = 1;

    // Since C(n, k) = C(n, n-k)
    if ( k > n - k )
        k = n - k;

    // Calculate value of [n * (n-1) * ... * (n-k+1)] / [k * (k-1) * ... * 1]
    for (int i = 0; i < k; ++i)
    {
        res *= (n - i);
        res /= (i + 1);
    }

    return res;
}

/* Driver program to test above function*/
int main()
{
    int n = 8, k = 2;
    printf ("Value of C(%d, %d) is %d ", n, k, binomialCoeff(n, k));
    return 0;
}
```

Value of $C(8, 2)$ is 28

Time Complexity: $O(k)$

Auxiliary Space: $O(1)$

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Mathematical
MathematicalAlgo

Reservoir Sampling

[Reservoir sampling](#) is a family of randomized algorithms for randomly choosing k samples from a list of n items, where n is either a very large or unknown number. Typically n is large enough that the list doesn't fit into main memory. For example, a list of search queries in Google and Facebook.

So we are given a big array (or stream) of numbers (to simplify), and we need to write an efficient function to randomly select k numbers where $1 \leq k \leq n$. Let the input array be $stream[]$.

A **simple solution** is to create an array $reservoir[]$ of maximum size k . One by one randomly select an item from $stream[0..n-1]$. If the selected item is not previously selected, then put it in $reservoir[]$. To check if an item is previously selected or not, we need to search the item in $reservoir[]$. The time complexity of this algorithm will be $O(k^2)$. This can be costly if k is big. Also, this is not efficient if the input is in the form of a stream.

It can be solved in $O(n)$ time. The solution also suits well for input in the form of stream. The idea is similar to [this post](#). Following are the steps.

- 1) Create an array $reservoir[0..k-1]$ and copy first k items of $stream[]$ to it.
- 2) Now one by one consider all items from $(k+1)$ th item to n th item.
 - ..a) Generate a random number from 0 to i where i is index of current item in $stream[]$. Let the generated random number is j .
 - ..b) If j is in range 0 to $k-1$, replace $reservoir[j]$ with $arr[i]$

Following is C implementation of the above algorithm.

```
// An efficient program to randomly select k items from a stream of items
```

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// A utility function to print an array
void printArray(int stream[], int n)
{
    for (int i = 0; i < n; i++)
        printf("%d ", stream[i]);
    printf("\n");
}

// A function to randomly select k items from stream[0..n-1].
void selectKItems(int stream[], int n, int k)
{
    int i; // index for elements in stream[]

    // reservoir[] is the output array. Initialize it with
    // first k elements from stream[]
    int reservoir[k];
    for (i = 0; i < k; i++)
        reservoir[i] = stream[i];

    // Use a different seed value so that we don't get
    // same result each time we run this program
    srand(time(NULL));

    // Iterate from the (k+1)th element to nth element
    for (; i < n; i++)
    {
        // Pick a random index from 0 to i.
        int j = rand() % (i+1);

        // If the randomly picked index is smaller than k, then replace
        // the element present at the index with new element from stream
        if (j < k)
            reservoir[j] = stream[i];
    }

    printf("Following are k randomly selected items \n");
    printArray(reservoir, k);
}

// Driver program to test above function.
int main()
{
    int stream[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
    int n = sizeof(stream)/sizeof(stream[0]);
    int k = 5;
    selectKItems(stream, n, k);
    return 0;
}
```

Output:

```
Following are k randomly selected items  
6 2 11 8 12
```

Time Complexity: O(n)

How does this work?

To prove that this solution works perfectly, we must prove that the probability that any item $stream[i]$ where $0 \leq i < n$ will be in final reservoir is k/n . Let us divide the proof in two cases as first k items are treated differently.

Case 1: For last $n-k$ stream items, i.e., for $stream[i]$ where $i \geq k$

For every such stream item $stream[i]$, we pick a random index from 0 to i and if the picked index is one of the first k indexes, we replace the element at picked index with $stream[i]$.

To simplify the proof, let us first consider the *last item*. The probability that the last item is in final reservoir = The probability that one of the first k indexes is picked for last item = k/n (the probability of picking one of the k items from a list of size n)

Let us now consider the *second last item*. The probability that the second last item is in final reservoir = [Probability that one of the first k indexes is picked in iteration for $stream[n-2]$] X [Probability that the index picked in iteration for $stream[n-1]$ is not same as index picked for $stream[n-2]$] = $[k/(n-1)] * [(n-1)/n] = k/n$.

Similarly, we can consider other items for all stream items from $stream[n-1]$ to $stream[0]$ and generalize the proof.

Case 2: For first k stream items, i.e., for $stream[i]$ where $0 \leq i < k$

The first k items are initially copied to $reservoir[]$ and may be removed later in iterations for $stream[k]$ to $stream[n]$.

The probability that an item from $stream[0..k-1]$ is in final array = Probability that the item is not picked when items $stream[k]$, $stream[k+1]$, ..., $stream[n-1]$ are considered = $[k/(k+1)] * [(k+1)/(k+2)] * [(k+2)/(k+3)] * \dots * [(n-1)/n] = k/n$

References:

http://en.wikipedia.org/wiki/Reservoir_sampling

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Articles
Randomized
MathematicalAlgo

Pascal's Triangle

Pascal's triangle is a triangular array of the binomial coefficients. Write a function that takes an integer value n as input and prints first n lines of the Pascal's triangle. Following are the first 6 rows of Pascal's Triangle.

```
1  
1 1  
1 2 1  
1 3 3 1  
1 4 6 4 1  
1 5 10 10 5 1
```

We strongly recommend that you click here and practice it, before moving on to the solution.

Method 1 (O(n^3) time complexity)

Number of entries in every line is equal to line number. For example, the first line has "1", the second line has "1 1", the third line has "1 2 1", ... and so on. Every entry in a line is value of a Binomial Coefficient. The value of i th entry in line number $line$ is $C(line, i)$. The value can be calculated using following formula.

```
C(line, i) = line! / ((line-i)! * i!)
```

A simple method is to run two loops and calculate the value of Binomial Coefficient in inner loop.

```
// A simple O(n^3) program for Pascal's Triangle  
#include <stdio.h>  
  
// See http://www.geeksforgeeks.org/archives/25621 for details of this function  
int binomialCoeff(int n, int k)  
  
// Function to print first n lines of Pascal's Triangle  
void printPascal(int n)  
{  
    // Iterate through every line and print entries in it  
    for (int line = 0; line < n; line++)  
    {  
        // Every line has number of integers equal to line number  
        for (int i = 0; i <= line; i++)  
            printf("%d ", binomialCoeff(line, i));  
        printf("\n");  
    }  
}  
  
// See http://www.geeksforgeeks.org/archives/25621 for details of this function  
int binomialCoeff(int n, int k)  
{  
    int res = 1;  
    if (k > n - k)  
        k = n - k;  
    for (int i = 0; i < k; ++i)  
    {  
        res *= (n - i);  
        res /= (i + 1);  
    }  
    return res;  
}  
  
// Driver program to test above function  
int main()  
{  
    int n = 7;
```

```

printPascal(n);
return 0;
}

```

Time complexity of this method is $O(n^3)$. Following are optimized methods.

Method 2($O(n^2)$ time and $O(n^2)$ extra space)

If we take a closer at the triangle, we observe that every entry is sum of the two values above it. So we can create a 2D array that stores previously generated values. To generate a value in a line, we can use the previously stored values from array.



```

// A O(n^2) time and O(n^2) extra space method for Pascal's Triangle
void printPascal(int n)
{
    int arr[n][n]; // An auxiliary array to store generated pascal triangle values

    // Iterate through every line and print integer(s) in it
    for (int line = 0; line < n; line++)
    {
        // Every line has number of integers equal to line number
        for (int i = 0; i <= line; i++)
        {
            // First and last values in every row are 1
            if (line == i || i == 0)
                arr[line][i] = 1;
            else // Other values are sum of values just above and left of above
                arr[line][i] = arr[line-1][i-1] + arr[line-1][i];
            printf("%d ", arr[line][i]);
        }
        printf("\n");
    }
}

```

This method can be optimized to use $O(n)$ extra space as we need values only from previous row. So we can create an auxiliary array of size n and overwrite values. Following is another method uses only $O(1)$ extra space.

Method 3 ($O(n^2)$ time and $O(1)$ extra space)

This method is based on method 1. We know that i th entry in a line number $line$ is Binomial Coefficient $C(line, i)$ and all lines start with value 1. The idea is to calculate $C(line, i)$ using $C(line, i-1)$. It can be calculated in $O(1)$ time using the following.

```

C(line, i) = line! / ((line-i)! * i!)
C(line, i-1) = line! / ((line - i + 1)! * (i-1)!)
We can derive following expression from above two expressions.
C(line, i) = C(line, i-1) * (line - i + 1) / i
So C(line, i) can be calculated from C(line, i-1) in O(1) time

```

```

// A O(n^2) time and O(1) extra space function for Pascal's Triangle
void printPascal(int n)
{
    for (int line = 1; line <= n; line++)
    {
        int C = 1; // used to represent C(line, i)
        for (int i = 1; i <= line; i++)
        {
            printf("%d ", C); // The first value in a line is always 1
            C = C * (line - i) / i;
        }
        printf("\n");
    }
}

```

So method 3 is the best method among all, but it may cause integer overflow for large values of n as it multiplies two integers to obtain values.

This article is compiled by **Rahul** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Mathematical
MathematicalAlgo
pattern-printing

Select a random number from stream, with $O(1)$ space

Given a stream of numbers, generate a random number from the stream. You are allowed to use only $O(1)$ space and the input is in the form of stream, so can't store the previously seen numbers.

So how do we generate a random number from the whole stream such that the probability of picking any number is $1/n$. with $O(1)$ extra space? This problem is a variation of [Reservoir Sampling](#). Here the value of k is 1.

- 1) Initialize 'count' as 0, 'count' is used to store count of numbers seen so far in stream.
- 2) For each number 'x' from stream, do following
 -a) Increment 'count' by 1.
 -b) If count is 1, set result as x, and return result.
 -c) Generate a random number from 0 to 'count-1'. Let the generated random number be i.
 -d) If i is equal to 'count - 1', update the result as x.

```

// An efficient program to randomly select a number from stream of numbers.
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

```

```

// A function to randomly select a item from stream[0], stream[1], .. stream[i-1]
int selectRandom(int x)
{
    static int res; // The resultant random number
    static int count = 0; //Count of numbers visited so far in stream

    count++; // increment count of numbers seen so far

    // If this is the first element from stream, return it
    if (count == 1)
        res = x;
    else
    {
        // Generate a random number from 0 to count - 1
        int i = rand() % count;

        // Replace the prev random number with new number with 1/count probability
        if (i == count - 1)
            res = x;
    }
    return res;
}

// Driver program to test above function.
int main()
{
    int stream[] = {1, 2, 3, 4};
    int n = sizeof(stream)/sizeof(stream[0]);

    // Use a different seed value for every run.
    srand(time(NULL));
    for (int i = 0; i < n; ++i)
        printf("Random number from first %d numbers is %d \n",
               i+1, selectRandom(stream[i]));
    return 0;
}

```

Output:

```

Random number from first 1 numbers is 1
Random number from first 2 numbers is 1
Random number from first 3 numbers is 3
Random number from first 4 numbers is 4

```

Auxiliary Space: O(1)

How does this work

We need to prove that every element is picked with $1/n$ probability where n is the number of items seen so far. For every new stream item x , we pick a random number from 0 to 'count -1', if the picked number is 'count-1', we replace the previous result with x .

To simplify proof, let us first consider the last element, the last element replaces the previously stored result with $1/n$ probability. So probability of getting last element as result is $1/n$.

Let us now talk about second last element. When second last element processed first time, the probability that it replaced the previous result is $1/(n-1)$. The probability that previous result stays when nth item is considered is $(n-1)/n$. So probability that the second last element is picked in last iteration is $[1/(n-1)] * [(n-1)/n]$ which is $1/n$.

Similarly, we can prove for third last element and others.

References:

Reservoir Sampling

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Randomized
Mathematical Algo
Random Algorithms

Find the largest multiple of 2, 3 and 5

An array of size n is given. The array contains digits from 0 to 9. Generate the largest number using the digits in the array such that the number is divisible by 2, 3 and 5.

For example, if the arrays is {1, 8, 7, 6, 0}, output must be: 8760. And if the arrays is {7, 7, 7, 6}, output must be: "no number can be formed".

Source: [Amazon Interview | Set 7](#)

This problem is a variation of "Find the largest multiple of 3".

Since the number has to be divisible by 2 and 5, it has to have last digit as 0. So if the given array doesn't contain any zero, then no solution exists.

Once a 0 is available, extract 0 from the given array. Only thing left is, the number should be is divisible by 3 and the largest of all. Which has been discussed [here](#).

Thanks to [shashank](#) for suggesting this solution. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Mathematical
MathematicalAlgo

Efficient program to calculate e^x

The value of Exponential Function e^x can be expressed using following Taylor Series.

$$e^x = 1 + x/1! + x^2/2! + x^3/3! + \dots$$

How to efficiently calculate the sum of above series?

The series can be re-written as

$$e^x = 1 + (x/1) (1 + (x/2) (1 + (x/3) (\dots)))$$

Let the sum needs to be calculated for n terms, we can calculate sum using following loop.

```
for (i = n - 1, sum = 1; i > 0; --i)
    sum = 1 + x * sum / i;
```

Following is implementation of the above idea.

```
// Efficient program to calculate e raise to the power x
#include <stdio.h>

// Returns approximate value of e^x using sum of first n terms of Taylor Series
float exponential(int n, float x)
{
    float sum = 1.0f; // initialize sum of series

    for (int i = n - 1; i > 0; --i)
        sum = 1 + x * sum / i;

    return sum;
}

// Driver program to test above function
int main()
{
    int n = 10;
    float x = 1.0f;
    printf("e^x = %f", exponential(n, x));
    return 0;
}
```

Output:

```
e^x = 2.718282
```

This article is compiled by **Rahul** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Mathematical
MathematicalAlgo

Measure one litre using two vessels and infinite water supply

There are two vessels of capacities 'a' and 'b' respectively. We have infinite water supply. Give an efficient algorithm to make exactly 1 litre of water in one of the vessels. You can throw all the water from any vessel any point of time. Assume that 'a' and 'b' are **Coprimes**.

Following are the steps:

Let V1 be the vessel of capacity 'a' and V2 be the vessel of capacity 'b' and 'a' is smaller than 'b'.

- 1) Do following while the amount of water in V1 is not 1.
 -a) If V1 is empty, then completely fill V1
 -b) Transfer water from V1 to V2. If V2 becomes full, then keep the remaining water in V1 and empty V2
- 2) V1 will have 1 litre after termination of loop in step 1. Return.

Following is C++ implementation of the above algorithm.

```
/* Sample run of the Algo for V1 with capacity 3 and V2 with capacity 7
1. Fill V1:           V1 = 3, V2 = 0
2. Transfer from V1 to V2, and fill V1:  V1 = 3, V2 = 3
2. Transfer from V1 to V2, and fill V1:  V1 = 3, V2 = 6
3. Transfer from V1 to V2, and empty V2: V1 = 2, V2 = 0
4. Transfer from V1 to V2, and fill V1:  V1 = 3, V2 = 2
5. Transfer from V1 to V2, and fill V1:  V1 = 3, V2 = 5
6. Transfer from V1 to V2, and empty V2: V1 = 1, V2 = 0
7. Stop as V1 now contains 1 litre.
```

Note that V2 was made empty in steps 3 and 6 because it became full */

```
#include <iostream>
using namespace std;

// A utility function to get GCD of two numbers
int gcd(int a, int b) { return b ? gcd(b, a % b) : a; }

// Class to represent a Vessel
class Vessel
{
    // A vessel has capacity, and current amount of water in it
    int capacity, current;
public:
    // Constructor: initializes capacity as given, and current as 0
    Vessel(int capacity) { this->capacity = capacity; current = 0; }

    // The main function to fill one litre in this vessel. Capacity of V2
    // must be greater than this vessel and two capacities must be co-prime
    void makeOneLitre(Vessel &V2);

    // Fills vessel with given amount and returns the amount of water
    // transferred to it. If the vessel becomes full, then the vessel
    // is made empty.
    int transfer(int amount);
};

// The main function to fill one litre in this vessel. Capacity
// of V2 must be greater than this vessel and two capacities
// must be coprime
void Vessel::makeOneLitre(Vessel &V2)
{
    // solution exists iff a and b are co-prime
    if (gcd(capacity, V2.capacity) != 1)
        return;

    while (current != 1)
```

```

{
    // fill A (smaller vessel)
    if (current == 0)
        current = capacity;

    cout << "Vessel 1: " << current << " Vessel 2: "
        << V2.current << endl;

    // Transfer water from V1 to V2 and reduce current of V1 by
    // the amount equal to transferred water
    current = current - V2.transfer(current);
}

// Finally, there will be 1 litre in vessel 1
cout << "Vessel 1: " << current << " Vessel 2: "
    << V2.current << endl;
}

// Fills vessel with given amount and returns the amount of water
// transferred to it. If the vessel becomes full, then the vessel
// is made empty
int Vessel::transfer(int amount)
{
    // If the vessel can accommodate the given amount
    if (current + amount < capacity)
    {
        current += amount;
        return amount;
    }

    // If the vessel cannot accommodate the given amount, then
    // store the amount of water transferred
    int transferred = capacity - current;

    // Since the vessel becomes full, make the vessel
    // empty so that it can be filled again
    current = 0;

    return transferred;
}

// Driver program to test above function
int main()
{
    int a = 3, b = 7; // a must be smaller than b

    // Create two vessels of capacities a and b
    Vessel V1(a), V2(b);

    // Get 1 litre in first vessel
    V1.makeOneLitre(V2);

    return 0;
}

```

Output:

```

Vessel 1:3  Vessel 2:0
Vessel 1:3  Vessel 2:3
Vessel 1:3  Vessel 2:6
Vessel 1:2  Vessel 2:0
Vessel 1:3  Vessel 2:2
Vessel 1:3  Vessel 2:5
Vessel 1:1  Vessel 2:0

```

How does this work?

To prove that the algorithm works, we need to proof that after certain number of iterations in the while loop, we will get 1 litre in V1.

Let 'a' be the capacity of vessel V1 and 'b' be the capacity of V2. Since we repeatedly transfer water from V1 to V2 until V2 becomes full, we will have ' $a - b \pmod{a}$ ' water in V1 when V2 becomes full first time. Once V2 becomes full, it is emptied. We will have ' $a - 2b \pmod{a}$ ' water in V1 when V2 is full second time. We repeat the above steps, and get ' $a - nb \pmod{a}$ ' water in V1 after the vessel V2 is filled and emptied 'n' times. We need to prove that the value of ' $a - nb \pmod{a}$ ' will be 1 for a finite integer 'n'. To prove this, let us consider the following property of coprime numbers.

For any two coprime integers 'a' and 'b', the integer 'b' has a multiplicative inverse modulo 'a'. In other words, there exists an integer 'y' such that $b \cdot y \equiv 1 \pmod{a}$. (See 3rd point [here](#)). After ' $(a - 1) \cdot y$ ' iterations, we will have ' $a - [(a - 1) \cdot y \cdot b \pmod{a}]$ ' water in V1, the value of this expression is ' $a - [(a - 1) \cdot 1] \pmod{a}$ ' which is 1. So the algorithm converges and we get 1 litre in V1.

This article is compiled by [Aashish Barnwal](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Mathematical
MathematicalAlgo
modular-arithmetic

Efficient program to print all prime factors of a given number

Given a number n, write an efficient function to print all prime factors of n. For example, if the input number is 12, then output should be "2 2 3". And if the input number is 315, then output should be "3 3 5 7".

Following are the steps to find all prime factors.

- 1) While n is divisible by 2, print 2 and divide n by 2.
- 2) After step 1, n must be odd. Now start a loop from i = 3 to square root of n. While i divides n, print i and divide n by i, increment i by 2 and continue.
- 3) If n is a prime number and is greater than 2, then n will not become 1 by above two steps. So print n if it is greater than 2.

C/C++

```

// Program to print all prime factors
# include <stdio.h>
# include <math.h>

// A function to print all prime factors of a given number n
void primeFactors(int n)
{

```

```

// Print the number of 2s that divide n
while (n%2 == 0)
{
    printf("%d ", 2);
    n = n/2;
}

// n must be odd at this point. So we can skip one element (Note i = i +2)
for (int i = 3; i <= sqrt(n); i += 2)
{
    // While i divides n, print i and divide n
    while (n%i == 0)
    {
        printf("%d ", i);
        n = n/i;
    }
}

// This condition is to handle the case when n is a prime number
// greater than 2
if (n > 2)
    printf ("%d ", n);
}
}

/* Driver program to test above function */
int main()
{
    int n = 315;
    primeFactors(n);
    return 0;
}

```

Java

```

// Program to print all prime factors
import java.io.*;
import java.lang.Math;

class GFG
{
    // A function to print all prime factors
    // of a given number n
    public static void primeFactors(int n)
    {
        // Print the number of 2s that divide n
        while (n%2==0)
        {
            System.out.print(2 + " ");
            n /= 2;
        }

        // n must be odd at this point. So we can
        // skip one element (Note i = i +2)
        for (int i = 3; i <= Math.sqrt(n); i+= 2)
        {
            // While i divides n, print i and divide n
            while (n%i == 0)
            {
                System.out.print(i + " ");
                n /= i;
            }
        }

        // This condition is to handle the case when
        // n is a prime number greater than 2
        if (n > 2)
            System.out.print(n);
    }

    public static void main (String[] args)
    {
        int n = 315;
        primeFactors(n);
    }
}

```

Output:

3 3 5 7

How does this work?

The steps 1 and 2 take care of composite numbers and step 3 takes care of prime numbers. To prove that the complete algorithm works, we need to prove that steps 1 and 2 actually take care of composite numbers. This is clear that step 1 takes care of even numbers. And after step 1, all remaining prime factor must be odd (difference of two prime factors must be at least 2), this explains why i is incremented by 2.

Now the main part is, the loop runs till square root of n not till. To prove that this optimization works, let us consider the following property of composite numbers.

Every composite number has at least one prime factor less than or equal to square root of itself.

This property can be proved using counter statement. Let a and b be two factors of n such that $a \cdot b = n$. If both are greater than \sqrt{n} , then $a \cdot b > \sqrt{n} \cdot \sqrt{n}$, which contradicts the expression " $a \cdot b = n$ ".

In step 2 of the above algorithm, we run a loop and do following in loop

- Find the least prime factor i (must be less than \sqrt{n})
- Remove all occurrences i from n by repeatedly dividing n by i.
- Repeat steps a and b for divided n and i = i + 2. The steps a and b are repeated till n becomes either 1 or a prime number.

Thanks to **Vishwas Garg** for suggesting the above algorithm. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Print all possible combinations of r elements in a given array of size n

Given an array of size n, generate and print all possible combinations of r elements in array. For example, if input array is {1, 2, 3, 4} and r is 2, then output should be {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4} and {3, 4}.

Following are two methods to do this.

Method 1 (Fix Elements and Recur)

We create a temporary array 'data[]' which stores all outputs one by one. The idea is to start from first index (index = 0) in data[], one by one fix elements at this index and recur for remaining indexes. Let the input array be {1, 2, 3, 4, 5} and r be 3. We first fix 1 at index 0 in data[], then recur for remaining indexes, then we fix 2 at index 0 and recur. Finally, we fix 3 and recur for remaining indexes. When number of elements in data[] becomes equal to r (size of a combination), we print data[].

Following diagram shows recursion tree for same input.



Following is C++ implementation of above approach.

C

```
// Program to print all combination of size r in an array of size n
#include <stdio.h>
void combinationUtil(int arr[], int data[], int start, int end,
                     int index, int r);

// The main function that prints all combinations of size r
// in arr[] of size n. This function mainly uses combinationUtil()
void printCombination(int arr[], int n, int r)
{
    // A temporary array to store all combination one by one
    int data[r];

    // Print all combination using temporary array 'data[]'
    combinationUtil(arr, data, 0, n-1, 0, r);
}

/* arr[] --> Input Array
   data[] --> Temporary array to store current combination
   start & end --> Starting and Ending indexes in arr[]
   index --> Current index in data[]
   r --> Size of a combination to be printed */
void combinationUtil(int arr[], int data[], int start, int end,
                     int index, int r)
{
    // Current combination is ready to be printed, print it
    if (index == r)
    {
        for (int j=0; j<r; j++)
            printf("%d ", data[j]);
        printf("\n");
        return;
    }

    // replace index with all possible elements. The condition
    // "end-i+1 >= r-index" makes sure that including one element
    // at index will make a combination with remaining elements
    // at remaining positions
    for (int i=start; i<=end && end-i+1 >= r-index; i++)
    {
        data[index] = arr[i];
        combinationUtil(arr, data, i+1, end, index+1, r);
    }
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    int r = 3;
    int n = sizeof(arr)/sizeof(arr[0]);
    printCombination(arr, n, r);
}
```

Java

```
// Java program to print all combination of size r in an array of size n
import java.io.*;

class Permutation {

    /* arr[] --> Input Array
       data[] --> Temporary array to store current combination
       start & end --> Starting and Ending indexes in arr[]
       index --> Current index in data[]
       r --> Size of a combination to be printed */
    static void combinationUtil(int arr[], int data[], int start,
                               int end, int index, int r)
    {
        // Current combination is ready to be printed, print it
        if (index == r)
        {
            for (int j=0; j<r; j++)
                System.out.print(data[j] + " ");
            System.out.println();
            return;
        }

        // replace index with all possible elements. The condition
        // "end-i+1 >= r-index" makes sure that including one element
        // at index will make a combination with remaining elements
        // at remaining positions
        for (int i=start; i<=end && end-i+1 >= r-index; i++)
        {
            data[index] = arr[i];
            combinationUtil(arr, data, i+1, end, index+1, r);
        }
    }
}
```

```

        System.out.print(data[i]+ " ");
        System.out.println("");
        return;
    }

    // replace index with all possible elements. The condition
    // "end-i+1 >= r-index" makes sure that including one element
    // at index will make a combination with remaining elements
    // at remaining positions
    for (int i=start; i<=end && end-i+1 >= r-index; i++)
    {
        data[index] = arr[i];
        combinationUtil(arr, data, i+1, end, index+1, r);
    }
}

// The main function that prints all combinations of size r
// in arr[] of size n. This function mainly uses combinationUtil()
static void printCombination(int arr[], int n, int r)
{
    // A temporary array to store all combination one by one
    int data[] = new int[r];

    // Print all combination using temporary array 'data[]'
    combinationUtil(arr, data, 0, n-1, 0, r);
}

/*Driver function to check for above function*/
public static void main (String[] args) {
    int arr[] = {1, 2, 3, 4, 5};
    int r = 3;
    int n = arr.length;
    printCombination(arr, n, r);
}
}

/* This code is contributed by Devesh Agrawal */

```

Output:

```

1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5

```

How to handle duplicates?

Note that the above method doesn't handle duplicates. For example, if input array is {1, 2, 1} and r is 2, then the program prints {1, 2} and {2, 1} as two different combinations. We can avoid duplicates by adding following two additional things to above code.

- 1) Add code to sort the array before calling combinationUtil() in printCombination()
- 2) Add following lines at the end of for loop in combinationUtil()

```

    // Since the elements are sorted, all occurrences of an element
    // must be together
    while (arr[i] == arr[i+1])
        i++;

```

See [this](#) for an implementation that handles duplicates.

Method 2 (Include and Exclude every element)

Like the above method, We create a temporary array data[]. The idea here is similar to [Subset Sum Problem](#). We one by one consider every element of input array, and recur for two cases:

- 1) The element is included in current combination (We put the element in data[]) and increment next available index in data[])
- 2) The element is excluded in current combination (We do not put the element and do not change index)

When number of elements in data[] become equal to r (size of a combination), we print it.

This method is mainly based on [Pascal's Identity](#), i.e. $n_{C_r} = n-1_{C_r} + n-1_{C_{r-1}}$

Following is C++ implementation of method 2.

C

```

// Program to print all combination of size r in an array of size n
#include<stdio.h>
void combinationUtil(int arr[], int n, int r, int index, int data[], int i);

// The main function that prints all combinations of size r
// in arr[] of size n. This function mainly uses combinationUtil()
void printCombination(int arr[], int n, int r)
{
    // A temporary array to store all combination one by one
    int data[r];

    // Print all combination using temporary array 'data[]'
    combinationUtil(arr, n, r, 0, data, 0);
}

/* arr[] --> Input Array
n --> Size of input array
r --> Size of a combination to be printed
index --> Current index in data[]
data[] --> Temporary array to store current combination
i --> index of current element in arr[] */
void combinationUtil(int arr[], int n, int r, int index, int data[], int i)
{
    // Current combination is ready, print it

```

```

if (index == r)
{
    for (int j=0; j<r; j++)
        printf("%d ",data[j]);
    printf("\n");
    return;
}

// When no more elements are there to put in data[]
if (i >= n)
    return;

// current is included, put next at next location
data[index] = arr[i];
combinationUtil(arr, n, r, index+1, data, i+1);

// current is excluded, replace it with next (Note that
// i+1 is passed, but index is not changed)
combinationUtil(arr, n, r, index, data, i+1);
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    int r = 3;
    int n = sizeof(arr)/sizeof(arr[0]);
    printCombination(arr, n, r);
    return 0;
}

```

Java

```

// Java program to print all combination of size r in an array of size n
import java.io.*;

class Permutation {

    /* arr[] ----> Input Array
       data[] ----> Temporary array to store current combination
       start & end ----> Starting and Ending indexes in arr[]
       index ----> Current index in data[]
       r ----> Size of a combination to be printed */
    static void combinationUtil(int arr[], int n, int r, int index,
                                int data[], int i)
    {
        // Current combination is ready to be printed, print it
        if (index == r)
        {
            for (int j=0; j<r; j++)
                System.out.print(data[j]+" ");
            System.out.println("");
            return;
        }

        // When no more elements are there to put in data[]
        if (i >= n)
            return;

        // current is included, put next at next location
        data[index] = arr[i];
        combinationUtil(arr, n, r, index+1, data, i+1);

        // current is excluded, replace it with next (Note that
        // i+1 is passed, but index is not changed)
        combinationUtil(arr, n, r, index, data, i+1);
    }

    // The main function that prints all combinations of size r
    // in arr[] of size n. This function mainly uses combinationUtil()
    static void printCombination(int arr[], int n, int r)
    {
        // A temporary array to store all combination one by one
        int data[] = new int[r];

        // Print all combination using temporary array 'data[]'
        combinationUtil(arr, n, r, 0, data, 0);
    }

    /*Driver function to check for above function*/
    public static void main (String[] args) {
        int arr[] = {1, 2, 3, 4, 5};
        int r = 3;
        int n = arr.length;
        printCombination(arr, n, r);
    }
}

/* This code is contributed by Devesh Agrawal */

```

Output:

```

1 2 3
1 2 4
1 2 5
1 3 4
1 3 5
1 4 5
2 3 4
2 3 5
2 4 5
3 4 5

```

How to handle duplicates in method 2?

Like method 1, we can follow two things to handle duplicates.

- 1) Add code to sort the array before calling combinationUtil() in printCombination()
- 2) Add following lines between two recursive calls of combinationUtil() in combinationUtil()

```
// Since the elements are sorted, all occurrences of an element  
// must be together  
while ([arr][i] == [arr][i+1])  
    i++;
```

See [this](#) for an implementation that handles duplicates.

This article is contributed by **Bateesh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Mathematical
MathematicalAlgo
Recursion

Random number generator in arbitrary probability distribution fashion

Given n numbers, each with some frequency of occurrence. Return a random number with probability proportional to its frequency of occurrence.

Example:

```
Let following be the given numbers.  
arr[] = {10, 30, 20, 40}
```

```
Let following be the frequencies of given numbers.  
freq[] = {1, 6, 2, 1}
```

```
The output should be  
10 with probability 1/10  
30 with probability 6/10  
20 with probability 2/10  
40 with probability 1/10
```

It is quite clear that the simple random number generator won't work here as it doesn't keep track of the frequency of occurrence.

We need to somehow transform the problem into a problem whose solution is known to us.

One simple method is to take an auxiliary array (say aux[]) and duplicate the numbers according to their frequency of occurrence. Generate a random number(say r) between 0 to Sum-1(including both), where Sum represents summation of frequency array (freq[] in above example). Return the random number aux[r] (Implementation of this method is left as an exercise to the readers).

The limitation of the above method discussed above is huge memory consumption when frequency of occurrence is high. If the input is 997, 8761 and 1, this method is clearly not efficient.

How can we reduce the memory consumption? Following is detailed algorithm that uses O(n) extra space where n is number of elements in input arrays.

1. Take an auxiliary array (say prefix[]) of size n.
2. Populate it with prefix sum, such that prefix[i] represents sum of numbers from 0 to i.
3. Generate a random number(say r) between 1 to Sum(including both), where Sum represents summation of input frequency array.
4. Find index of Ceil of random number generated in step #3 in the prefix array. Let the index be indexc.
5. Return the random number arr[indexc], where arr[] contains the input n numbers.

Before we go to the implementation part, let us have quick look at the algorithm with an example:

```
arr[]: {10, 20, 30}  
freq[]: {2, 3, 1}  
Prefix[]: {2, 5, 6}
```

Since last entry in prefix is 6, all possible values of r are [1, 2, 3, 4, 5, 6]

- 1: Ceil is 2. Random number generated is 10.
- 2: Ceil is 2. Random number generated is 10.
- 3: Ceil is 5. Random number generated is 20.
- 4: Ceil is 5. Random number generated is 20.
- 5: Ceil is 5. Random number generated is 20.
- 6: Ceil is 6. Random number generated is 30.

In the above example

- 10 is generated with probability 2/6.
- 20 is generated with probability 3/6.
- 30 is generated with probability 1/6.

How does this work?

Any number input[i] is generated as many times as its frequency of occurrence because there exists count of integers in range(prefix[i - 1], prefix[i]) is input[i]. Like in the above example 3 is generated thrice, as there exists 3 integers 3, 4 and 5 whose ceil is 5.

```
/*C program to generate random numbers according to given frequency distribution  
#include <stdio.h>  
#include <stdlib.h>  
  
// Utility function to find ceiling of r in arr[..h]  
int findCeil(int arr[], int r, int l, int h)  
{  
    int mid;  
    while (l < h)  
    {  
        mid = l + ((h - l) >> 1); // Same as mid = (l+h)/2  
        (r > arr[mid]) ? (l = mid + 1) : (h = mid);  
    }  
    return (arr[l] >= r) ? l : -1;  
}  
  
// The main function that returns a random number from arr[] according to  
// distribution array defined by freq[], n is size of arrays.  
int myRand(int arr[], int freq[], int n)  
{  
    // Create and fill prefix array  
    int prefix[n];  
    prefix[0] = freq[0];  
    for (i = 1; i < n; ++i)  
        prefix[i] = prefix[i - 1] + freq[i];
```

```

// prefix[n-1] is sum of all frequencies. Generate a random number
// with value from 1 to this sum
int r = (rand() % prefix[n - 1]) + 1;

// Find index of ceiling of r in prefix array
int indexc = findCeil(prefix, r, 0, n - 1);
return arr[indexc];
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 2, 3, 4};
    int freq[] = {10, 5, 20, 100};
    int i, n = sizeof(arr) / sizeof(arr[0]);

    // Use a different seed value for every run.
    srand(time(NULL));

    // Let us generate 10 random numbers according to
    // given distribution
    for (i = 0; i < 5; i++)
        printf("%d\n", myRand(arr, freq, n));

    return 0;
}

```

Output: May be different for different runs

```

4
3
4
4
4

```

This article is compiled by Aashish Barnwal. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Randomized
MathematicalAlgo

How to check if a given number is Fibonacci number?

Given a number 'n', how to check if n is a Fibonacci number.

We strongly recommend that you click here and practice it, before moving on to the solution.

A simple way is to generate Fibonacci numbers until the generated number is greater than or equal to 'n'. Following is an interesting property about Fibonacci numbers that can also be used to check if a given number is Fibonacci or not.

A number is Fibonacci if and only if one or both of ($5n^2 + 4$) or ($5n^2 - 4$) is a perfect square (Source: [Wiki](#)). Following is a simple program based on this concept.

C++

```

// C++ program to check if x is a perfect square
#include <iostream>
#include <math.h>
using namespace std;

// A utility function that returns true if x is perfect square
bool isPerfectSquare(int x)
{
    int s = sqrt(x);
    return (s*s == x);
}

// Returns true if n is a Fibonacci Number, else false
bool isFibonacci(int n)
{
    // n is Fibinacci if one of 5*n*n + 4 or 5*n*n - 4 or both
    // is a perfect square
    return isPerfectSquare(5*n*n + 4) ||
           isPerfectSquare(5*n*n - 4);
}

// A utility function to test above functions
int main()
{
    for (int i = 1; i <= 10; i++)
        isFibonacci(i)? cout << i << " is a Fibonacci Number \n":
                      cout << i << " is not Fibonacci Number \n";
    return 0;
}

```

Python

```

# python program to check if x is a perfect square
import math

# A utility function that returns true if x is perfect square
def isPerfectSquare(x):
    s = int(math.sqrt(x))
    return s*s == x

# Returns true if n is a Fibonacci Number, else false
def isFibonacci(n):

```

```

# n is Fibinacci if one of 5*n*n + 4 or 5*n*n - 4 or both
# is a perfect square
return isPerfectSquare(5*n*n + 4) or isPerfectSquare(5*n*n - 4)

# A utility function to test above functions
for i in range(1,11):
    if (isFibonacci(i) == True):
        print i,"is a Fibonacci Number"
    else:
        print i,"is a not Fibonacci Number "

```

Output:

```

1 is a Fibonacci Number
2 is a Fibonacci Number
3 is a Fibonacci Number
4 is a not Fibonacci Number
5 is a Fibonacci Number
6 is a not Fibonacci Number
7 is a not Fibonacci Number
8 is a Fibonacci Number
9 is a not Fibonacci Number
10 is a not Fibonacci Number

```

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Mathematical
Fibonacci
MathematicalAlgo

Russian Peasant (Multiply two numbers using bitwise operators)

Given two integers, write a function to multiply them without using multiplication operator.

There are many other ways to multiply two numbers (For example, see [this](#)). One interesting method is the [Russian peasant algorithm](#). The idea is to double the first number and halve the second number repeatedly till the second number doesn't become 1. In the process, whenever the second number becomes odd, we add the first number to result (result is initialized as 0). The following is simple algorithm.

```

Let the two given numbers be 'a' and 'b'
1) Initialize result 'res' as 0.
2) Do following while 'b' is greater than 0
   a) If 'b' is odd, add 'a' to 'res'
   b) Double 'a' and halve 'b'
3) Return 'res'.

```

```

#include <iostream>
using namespace std;

// A method to multiply two numbers using Russian Peasant method
unsigned int russianPeasant(unsigned int a, unsigned int b)
{
    int res = 0; // initialize result

    // While second number doesn't become 1
    while (b > 0)
    {
        // If second number becomes odd, add the first number to result
        if (b & 1)
            res = res + a;

        // Double the first number and halve the second number
        a = a << 1;
        b = b >> 1;
    }
    return res;
}

// Driver program to test above function
int main()
{
    cout << russianPeasant(18, 1) << endl;
    cout << russianPeasant(20, 12) << endl;
    return 0;
}

```

Output:

```

18
240

```

How does this work?

The value of $a \cdot b$ is same as $(a^2) \cdot (b/2)$ if b is even, otherwise the value is same as $((a^2) \cdot (b/2)) + a$. In the while loop, we keep multiplying ' a ' with 2 and keep dividing ' b ' by 2. If ' b ' becomes odd in loop, we add ' a ' to ' res '. When value of ' b ' becomes 1, the value of ' res ' + ' a ', gives us the result.

Note that when ' b ' is a power of 2, the ' res ' would remain 0 and ' a ' would have the multiplication. See the reference for more information.

Reference:

<http://mathforum.org/dr.math/faq/faq.peasant.html>

This article is compiled by **Shalki Agarwal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Bit Magic
Mathematical
MathematicalAlgo

Count all possible groups of size 2 or 3 that have sum as multiple of 3

Given an unsorted integer (positive values only) array of size 'n', we can form a group of two or three, the group should be such that the sum of all elements in that group is a multiple of 3. Count all possible number of groups that can be generated in this way.

```
Input: arr[] = {3, 6, 7, 2, 9}
Output: 8
// Groups are {3,6}, {3,9}, {9,6}, {7,2}, {3,6,9},
// {3,7,2}, {7,2,6}, {7,2,9}
```

```
Input: arr[] = {2, 1, 3, 4}
Output: 4
// Groups are {2,1}, {2,4}, {2,1,3}, {2,4,3}
```

The idea is to see remainder of every element when divided by 3. A set of elements can form a group only if sum of their remainders is multiple of 3. Since the task is to enumerate groups, we count all elements with different remainders.

1. Hash all elements in a count array based on remainder, i.e,
for all elements $a[i]$, do $c[a[i]\%3]++$;
2. Now $c[0]$ contains the number of elements which when divided by 3 leave remainder 0 and similarly $c[1]$ for remainder 1 and $c[2]$ for 2.
3. Now for group of 2, we have 2 possibilities
 - a. 2 elements of remainder 0 group. Such possibilities are $c[0]^*(c[0]-1)/2$
 - b. 1 element of remainder 1 and 1 from remainder 2 group
Such groups are $c[1]*c[2]$.
4. Now for group of 3 we have 4 possibilities
 - a. 3 elements from remainder group 0.
No. of such groups are $c[0]C3$
 - b. 3 elements from remainder group 1.
No. of such groups are $c[1]C3$
 - c. 3 elements from remainder group 2.
No. of such groups are $c[2]C3$
 - d. 1 element from each of 3 groups.
No. of such groups are $c[0]*c[1]*c[2]$.
5. Add all the groups in steps 3 and 4 to obtain the result.

C

```
#include<stdio.h>

// Returns count of all possible groups that can be formed from elements
// of arr.
int findgroups(int arr[], int n)
{
    // Create an array C[3] to store counts of elements with remainder
    // 0, 1 and 2. c[i] would store count of elements with remainder i
    int c[3] = {0, 0, 0};

    int res = 0; // To store the result

    // Count elements with remainder 0, 1 and 2
    for (i=0; i<n; i++)
        c[arr[i]\%3]++;

    // Case 3.a: Count groups of size 2 from 0 remainder elements
    res += ((c[0]*(c[0]-1))>>1);

    // Case 3.b: Count groups of size 2 with one element with 1
    // remainder and other with 2 remainder
    res += c[1] * c[2];

    // Case 4.a: Count groups of size 3 with all 0 remainder elements
    res += (c[0] * (c[0]-1) * (c[0]-2))/6;

    // Case 4.b: Count groups of size 3 with all 1 remainder elements
    res += (c[1] * (c[1]-1) * (c[1]-2))/6;

    // Case 4.c: Count groups of size 3 with all 2 remainder elements
    res += ((c[2]*c[2]-1)*(c[2]-2))/6;

    // Case 4.c: Count groups of size 3 with different remainders
    res += c[0]*c[1]*c[2];

    // Return total count stored in res
    return res;
}

// Driver program to test above functions
int main()
{
    int arr[] = {3, 6, 7, 2, 9};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Required number of groups are %d\n", findgroups(arr,n));
    return 0;
}
```

Java

```
class FindGroups
{
    // Returns count of all possible groups that can be formed from elements
    // of arr.
    int findgroups(int arr[], int n)
    {
        // Create an array C[3] to store counts of elements with remainder
        // 0, 1 and 2. c[i] would store count of elements with remainder i
        int c[] = new int[]{0, 0, 0};
```

```

int i;

int res = 0; // To store the result

// Count elements with remainder 0, 1 and 2
for (i = 0; i < n; i++)
    c[arr[i] % 3]++;
}

// Case 3.a: Count groups of size 2 from 0 remainder elements
res += ((c[0] * (c[0] - 1)) >> 1);

// Case 3.b: Count groups of size 2 with one element with 1
// remainder and other with 2 remainder
res += c[1] * c[2];

// Case 4.a: Count groups of size 3 with all 0 remainder elements
res += (c[0] * (c[0] - 1) * (c[0] - 2)) / 6;

// Case 4.b: Count groups of size 3 with all 1 remainder elements
res += (c[1] * (c[1] - 1) * (c[1] - 2)) / 6;

// Case 4.c: Count groups of size 3 with all 2 remainder elements
res += ((c[2] * (c[2] - 1) * (c[2] - 2)) / 6);

// Case 4.c: Count groups of size 3 with different remainders
res += c[0] * c[1] * c[2];

// Return total count stored in res
return res;
}

public static void main(String[] args)
{
    FindGroups groups = new FindGroups();
    int arr[] = {3, 6, 7, 2, 9};
    int n = arr.length;
    System.out.println("Required number of groups are "
        + groups.findgroups(arr, n));
}
}

```

Output:

```
Required number of groups are 8
```

Time Complexity: O(n)

Auxiliary Space: O(1)

This article is contributed by Amit Jain. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Arrays

C Program for Tower of Hanoi

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

- 1) Only one disk can be moved at a time.
- 2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
- 3) No disk may be placed on top of a smaller disk.

```
#include <stdio.h>

// C recursive function to solve tower of hanoi puzzle
void towerOfHanoi(int n, char fromrod, char torod, char auxrod)
{
    if (n == 1)
    {
        printf("\n Move disk 1 from rod %c to rod %c", fromrod, torod);
        return;
    }
    towerOfHanoi(n-1, fromrod, auxrod, torod);
    printf("\n Move disk %d from rod %c to rod %c", n, fromrod, torod);
    towerOfHanoi(n-1, auxrod, torod, fromrod);
}

int main()
{
    int n = 4; // Number of disks
    towerOfHanoi(n, 'A', 'C', 'B'); // A, B and C are names of rods
    return 0;
}
```

Output:

```
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 3 from rod A to rod B
Move disk 1 from rod C to rod A
Move disk 2 from rod C to rod B
Move disk 1 from rod A to rod B
Move disk 4 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 2 from rod B to rod A
Move disk 1 from rod C to rod A
Move disk 3 from rod B to rod C
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
```

For n disks, total $2^n - 1$ moves are required.

Related Articles

- Recursive Functions
- Iterative solution to TOH puzzle
- Quiz on Recursion

References:

http://en.wikipedia.org/wiki/Tower_of_Hanoi

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Programs

Horner's Method for Polynomial Evaluation

Given a polynomial of the form $c_nx^n + c_{n-1}x^{n-1} + c_{n-2}x^{n-2} + \dots + c_1x + c_0$ and a value of x, find the value of polynomial for a given value of x. Here c_n, c_{n-1}, \dots are integers (may be negative) and n is a positive integer.

Input is in the form of an array say `poly[]` where `poly[0]` represents coefficient for x^0 and `poly[1]` represents coefficient for x^{n-1} and so on.

Examples:

```
// Evaluate value of 2x3 - 6x2 + 2x - 1 for x = 3
Input: poly[] = {2, -6, 2, -1}, x = 3
Output: 5

// Evaluate value of 2x3 + 3x + 1 for x = 2
Input: poly[] = {2, 0, 3, 1}, x = 2
Output: 23
```

A naive way to evaluate a polynomial is to one by one evaluate all terms. First calculate x^n , multiply the value with c_n , repeat the same steps for other terms and return the sum. Time complexity of this approach is $O(n^2)$ if we use a simple loop for evaluation of x^n . Time complexity can be improved to $O(nLogn)$ if we use [O\(Logn\)](#) approach for evaluation of x^n .

Horner's method can be used to evaluate polynomial in $O(n)$ time. To understand the method, let us consider the example of $2x^3 - 6x^2 + 2x - 1$. The polynomial can be evaluated as $((2x - 6)x + 2)x - 1$. The idea is to initialize result as coefficient of x^n which is 2 in this case, repeatedly multiply result with x and add next coefficient to result. Finally return result.

Following is C++ implementation of Horner's Method.

```
#include <iostream>
using namespace std;

// returns value of poly[0]x(n-1) + poly[1]x(n-2) + ... + poly[n-1]
int horner(int poly[], int n, int x)
{
    int result = poly[0]; // Initialize result

    // Evaluate value of polynomial using Horner's method
    for (int i=1; i<n; i++)
        result = result*x + poly[i];

    return result;
}

// Driver program to test above function.
int main()
{
    // Let us evaluate value of 2x3 - 6x2 + 2x - 1 for x = 3
    int poly[] = {2, -6, 2, -1};
    int x = 3;
    int n = sizeof(poly)/sizeof(poly[0]);
    cout << "Value of polynomial is " << horner(poly, n, x);
    return 0;
}
```

Output:

```
Value of polynomial is 5
```

Time Complexity: $O(n)$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Mathematical
MathematicalAlgo

Count trailing zeroes in factorial of a number

Given an integer n, write a function that returns count of trailing zeroes in $n!$.

Examples:

```
Input: n = 5
Output: 1
Factorial of 5 is 20 which has one trailing 0.
```

```
Input: n = 20
Output: 4
Factorial of 20 is 2432902008176640000 which has
4 trailing zeroes.
```

```
Input: n = 100
```

We strongly recommend that you click here and practice it, before moving on to the solution.

A simple method is to first calculate factorial of n, then count trailing 0s in the result (We can count trailing 0s by repeatedly dividing the factorial by 10 till the remainder is 0).

The above method can cause overflow for a slightly bigger numbers as factorial of a number is a big number (See factorial of 20 given in above examples). The idea is to consider prime factors of a factorial n. A trailing zero is always produced by prime factors 2 and 5. If we can count the number of 5s and 2s, our task is done. Consider the following examples.

n = 5: There is one 5 and 3 2s in prime factors of 5! ($2 \cdot 2 \cdot 2 \cdot 3 \cdot 5$). So count of trailing 0s is 1.

n = 11: There are two 5s and three 2s in prime factors of 11! ($2^8 \cdot 3^4 \cdot 5^2 \cdot 7$). So count of trailing 0s is 2.

We can easily observe that the number of 2s in prime factors is always more than or equal to the number of 5s. So if we count 5s in prime factors, we are done. How to count total number of 5s in prime factors of $n!$? A simple way is to calculate $\text{floor}(n/5)$. For example, 7! has one 5, 10! has two 5s. It is done yet, there is one more thing to consider. Numbers like 25, 125, etc have more than one 5. For example if we consider 28!, we get one extra 5 and number of 0s become 6. Handling this is simple, first divide n by 5 and remove all single 5s, then divide by 25 to remove extra 5s and so on. Following is the summarized formula for counting trailing 0s.

```
Trailing 0s in n! = Count of 5s in prime factors of n!
= floor(n/5) + floor(n/25) + floor(n/125) + ...
```

Following is C++ program based on above formula.

```
// C++ program to count trailing 0s in n!
#include <iostream>
using namespace std;

// Function to return trailing 0s in factorial of n
int findTrailingZeros(int n)
{
    // Initialize result
    int count = 0;

    // Keep dividing n by powers of 5 and update count
    for (int i=5; n/i>=1; i *= 5)
        count += n/i;

    return count;
}

// Driver program to test above function
int main()
{
    int n = 100;
    cout << "Count of trailing 0s in " << n
         << "!" << " is " << findTrailingZeros(n);
    return 0;
}
```

Output:

```
Count of trailing 0s in 100! is 24
```

This article is contributed by **Rahul Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Mathematical
factorial
MathematicalAlgo

Program for nth Catalan Number

Catalan numbers are a sequence of natural numbers that occurs in many interesting counting problems like following.

1) Count the number of expressions containing n pairs of parentheses which are correctly matched. For n = 3, possible expressions are ((())), ()(), ()(), (0)0, (00).

2) Count the number of possible Binary Search Trees with n keys (See [this](#))

3) Count the number of full binary trees (A rooted binary tree is full if every vertex has either two children or no children) with n+1 leaves.

See [this](#) for more applications.

The first few Catalan numbers for n = 0, 1, 2, 3, ... are 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, ...

Recursive Solution

Catalan numbers satisfy the following recursive formula.



Following is the implementation of above recursive formula.

C++

```
#include<iostream>
using namespace std;

// A recursive function to find nth catalan number
unsigned long int catalan(unsigned int n)
{
    // Base case
    if (n <= 1) return 1;

    // catalan(n) is sum of catalan(i)*catalan(n-i-1)
    unsigned long int res = 0;
    for (int i=0; i<n; i++)
        res += catalan(i)*catalan(n-i-1);

    return res;
}
```

```

}

// Driver program to test above function
int main()
{
    for (int i=0; i<10; i++)
        cout << catalan(i) << " ";
    return 0;
}

```

Java

```

class CatalnNumber {

    // A recursive function to find nth catalan number

    int catalan(int n) {
        int res = 0;

        // Base case
        if (n <= 1) {
            return 1;
        }
        for (int i = 0; i < n; i++) {
            res += catalan(i) * catalan(n - i - 1);
        }
        return res;
    }

    public static void main(String[] args) {
        CatalnNumber cn = new CatalnNumber();
        for (int i = 0; i < 10; i++) {
            System.out.print(cn.catalan(i) + " ");
        }
    }
}

```

Python

```

# A recursive function to find nth catalan number
def catalan(n):
    # Base Case
    if n <=1 :
        return 1

    # Catalan(n) is the sum of catalan(i)*catalan(n-i-1)
    res = 0
    for i in range(n):
        res += catalan(i) * catalan(n-i-1)

    return res

# Driver Program to test above function
for i in range(10):
    print(catalan(i))

# This code is contributed by Nikhil Kumar Singh (nickzuck_007)

```

Output :

```
1 1 2 5 14 42 132 429 1430 4862
```

Time complexity of above implementation is equivalent to nth catalan number.



The value of nth catalan number is exponential that makes the time complexity exponential.

Dynamic Programming Solution

We can observe that the above recursive implementation does a lot of repeated work (we can see the same by drawing recursion tree). Since there are overlapping subproblems, we can use dynamic programming for this. Following is a Dynamic programming based implementation in C++.

```

#include<iostream>
using namespace std;

// A dynamic programming based function to find nth
// Catalan number
unsigned long int catalanDP(unsigned int n)
{
    // Table to store results of subproblems
    unsigned long int catalan[n+1];

    // Initialize first two values in table
    catalan[0] = catalan[1] = 1;

    // Fill entries in catalan[] using recursive formula
    for (int i=2; i<=n; i++)
    {
        catalan[i] = 0;
        for (int j=0; j<i; j++)
            catalan[i] += catalan[j] * catalan[i-j-1];
    }

    // Return last entry
    return catalan[n];
}

// Driver program to test above function

```

```

int main()
{
    for (int i = 0; i < 10; i++)
        cout << catalanDP(i) << " ";
    return 0;
}

```

Output:

```
1 1 2 5 14 42 132 429 1430 4862
```

Time Complexity: Time complexity of above implementation is $O(n^2)$

Using Binomial Coefficient

We can also use the below formula to find nth catalan number in $O(n)$ time.



We have discussed a $O(n)$ approach to find binomial coefficient nCr .

```

#include<iostream>
using namespace std;

// Returns value of Binomial Coefficient C(n, k)
unsigned long int binomialCoeff(unsigned int n, unsigned int k)
{
    unsigned long int res = 1;

    // Since C(n, k) = C(n, n-k)
    if (k > n - k)
        k = n - k;

    // Calculate value of [n*(n-1)*...*(n-k+1)]/[k*(k-1)*...*1]
    for (int i = 0; i < k; ++i)
    {
        res *= (n - i);
        res /= (i + 1);
    }

    return res;
}

// A Binomial coefficient based function to find nth catalan
// number in O(n) time
unsigned long int catalan(unsigned int n)
{
    // Calculate value of 2nCr
    unsigned long int c = binomialCoeff(2*n, n);

    // return 2nCr/(n+1)
    return c/(n+1);
}

// Driver program to test above functions
int main()
{
    for (int i = 0; i < 10; i++)
        cout << catalan(i) << " ";
    return 0;
}

```

Output:

```
1 1 2 5 14 42 132 429 1430 4862
```

Time Complexity: Time complexity of above implementation is $O(n)$.

We can also use below formula to find nth catalan number in $O(n)$ time.



References:

http://en.wikipedia.org/wiki/Catalan_number

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
Mathematical
catalan
MathematicalAlgo
series

Write a function that generates one of 3 numbers according to given probabilities

You are given a function rand(a, b) which generates equiprobable random numbers between [a, b] inclusive. Generate 3 numbers x, y, z with probability P(x), P(y), P(z) such that P(x) + P(y) + P(z) = 1 using the given rand(a,b) function.

The idea is to utilize the equiprobable feature of the rand(a,b) provided. *Let the given probabilities be in percentage form, for example P(x)=40%, P(y)=25%, P(z)=35%.*

Following are the detailed steps.

- 1) Generate a random number between 1 and 100. Since they are equiprobable, the probability of each number appearing is 1/100.
 - 2) Following are some important points to note about generated random number 'r'.
- a) 'r' is smaller than or equal to P(x) with probability P(x)/100.
 - b) 'r' is greater than P(x) and smaller than or equal P(x) + P(y) with P(y)/100.
 - c) 'r' is greater than P(x) + P(y) and smaller than or equal 100 (or P(x) + P(y) + P(z)) with probability P(z)/100.

```
// This function generates 'x' with probability px/100, 'y' with
// probability py/100 and 'z' with probability pz/100:
// Assumption:px + py + pz = 100 where px, py and pz lie
// between 0 to 100
int random(int x, int y, int z, int px, int py, int pz)
{
    // Generate a number from 1 to 100
    int r = rand(1, 100);

    // r is smaller than px with probability px/100
    if (r <= px)
        return x;

    // r is greater than px and smaller than or equal to px+py
    // with probability py/100
    if (r <= (px+py))
        return y;

    // r is greater than px+py and smaller than or equal to 100
    // with probability pz/100
    else
        return z;
}
```

This function will solve the purpose of generating 3 numbers with given three probabilities.

This article is contributed by **Harsh Agarwal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Mathematical
Randomized
MathematicalAlgo

Find Excel column name from a given column number

MS Excel columns has a pattern like A, B, C, ..., Z, AA, AB, AC, ..., AZ, BA, BB, ..., ZZ, AAA, AAB, etc. In other words, column 1 is named as "A", column 2 as "B", column 27 as "AA".

Given a column number, find its corresponding Excel column name. Following are more examples.

Input	Output
26	Z
51	AY
52	AZ
80	CB
676	YZ
702	ZZ
705	AAC

We strongly recommend that you click here and practice it, before moving on to the solution.

Thanks to Mrigank Dembla for suggesting the below solution in a comment.

Suppose we have a number n, let's say 28. so corresponding to it we need to print the column name. We need to take remainder with 26.

If remainder with 26 comes out to be 0 (meaning 26, 52 and so on) then we put 'Z' in the output string and new n becomes $n/26 - 1$ because here we are considering 26 to be 'Z' while in actual it's 25th with respect to 'A'.

Similarly if the remainder comes out to be non zero. (like 1, 2, 3 and so on) then we need to just insert the char accordingly in the string and do $n = n/26$.

Finally we reverse the string and print.

Example:

$n = 700$

Remainder ($n \% 26$) is 24. So we put 'X' in output string and n becomes $n/26$ which is 26.

Remainder ($26 \% 26$) is 0. So we put 'Z' in output string and n becomes $n/26 - 1$ which is 0.

Following is C++ implementation of above approach.

C++

```
#include<bits/stdc++.h>
#define MAX 50
using namespace std;

// Function to print Excel column name for a given column number
void printString(int n)
{
    char str[MAX]; // To store result (Excel column name)
    int i = 0; // To store current index in str which is result

    while (n>0)
    {
        // Find remainder
        int rem = n%26;

        // If remainder is 0, then a 'Z' must be there in output
        if (rem==0)
        {
            str[i++] = 'Z';
            n = (n/26)-1;
        }
        else // If remainder is non-zero
        {
            str[i++] = (rem-1) + 'A';
            n = n/26;
        }
    }
    str[i] = '\0';

    // Reverse the string and print result
    reverse(str, str + strlen(str));
    cout << str << endl;
}

return;
}

// Driver program to test above function
int main()
{
    printString(26);
    printString(51);
    printString(52);
    printString(80);
    printString(676);
    printString(702);
    printString(705);
    return 0;
}
```

Python

```
# Python program to find Excel column name from a
# given column number

MAX = 50

# Function to print Excel column name for a given column number
def printString(n):

    # To store result (Excel column name)
    string = ["0"]*MAX

    # To store current index in str which is result
    i = 0

    while n > 0:
        # Find remainder
        rem = n%26

        # if remainder is 0, then a 'Z' must be there in output
        if rem == 0:
            string[i] = 'Z'
            i += 1
            n = (n/26)-1
        else:
            string[i] = chr((rem-1) + ord('A'))
            i += 1
            n = n/26
    string[i] = '\0'

    # Reverse the string and print result
    print(string[::-1])
```

```

string = string[::-1]
print "".join(string)

# Driver program to test the above Function
printString(26)
printString(51)
printString(52)
printString(80)
printString(676)
printString(702)
printString(705)

# This code is contributed by BHAVYA JAIN

```

Output

```

Z
AY
AZ
CB
YZ
ZZ
AAC

```

This article is contributed by **Kartik**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Strings

Find next greater number with same set of digits

Given a number n, find the smallest number that has same set of digits as n and is greater than n. If x is the greatest possible number with its set of digits, then print "not possible".

Examples:

For simplicity of implementation, we have considered input number as a string.

```

Input: n = "218765"
Output: "251678"

Input: n = "1234"
Output: "1243"

Input: n = "4321"
Output: "Not Possible"

Input: n = "534976"
Output: "536479"

```

Following are few observations about the next greater number.

- 1) If all digits sorted in descending order, then output is always "Not Possible". For example, 4321.
- 2) If all digits are sorted in ascending order, then we need to swap last two digits. For example, 1234.
- 3) For other cases, we need to process the number from rightmost side (why? because we need to find the smallest of all greater numbers)

You can now try developing an algorithm yourself.

Following is the algorithm for finding the next greater number.

- I) Traverse the given number from rightmost digit, keep traversing till you find a digit which is smaller than the previously traversed digit. For example, if the input number is "534976", we stop at **4** because 4 is smaller than next digit 9. If we do not find such a digit, then output is "Not Possible".
- II) Now search the right side of above found digit 'd' for the smallest digit greater than 'd'. For "534976", the right side of 4 contains "976". The smallest digit greater than 4 is **6**.
- III) Swap the above found two digits, we get **536974** in above example.
- IV) Now sort all digits from position next to 'd' to the end of number. The number that we get after sorting is the output. For above example, we sort digits in **536974**. We get "**536479**" which is the next greater number for input 534976.

Following is C++ implementation of above approach.

```

// C++ program to find the smallest number which greater than a given number
// and has same set of digits as given number
#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;

// Utility function to swap two digits
void swap(char *a, char *b)
{
    char temp = *a;
    *a = *b;
    *b = temp;
}

// Given a number as a char array number[], this function finds the
// next greater number. It modifies the same array to store the result
void findNext(char number[], int n)
{
    int i, j;

    // I) Start from the right most digit and find the first digit that is
    // smaller than the digit next to it.
    for (i = n-1; i > 0; i--)
        if (number[i] > number[i+1])
            break;

    // If no such digit is found, then all digits are in descending order
    // means there cannot be a greater number with same set of digits
    if (i==0)
    {
        cout << "Next number is not possible";
    }
}

```

```

    return;
}

// II) Find the smallest digit on right side of (i-1)'th digit that is
// greater than number[i-1]
int x = number[i-1], smallest = i;
for (j = i+1; j < n; j++)
    if (number[j] > x && number[j] < number[smallest])
        smallest = j;

// III) Swap the above found smallest digit with number[i-1]
swap(&number[smallest], &number[i-1]);

// IV) Sort the digits after (i-1) in ascending order
sort(number + i, number + n);

cout << "Next number with same set of digits is " << number;

return;
}

// Driver program to test above function
int main()
{
    char digits[] = "534976";
    int n = strlen(digits);
    findNext(digits, n);
    return 0;
}

```

Output:

Next number with same set of digits is 536479

The above implementation can be optimized in following ways.

- 1) We can use binary search in step II instead of linear search.
- 2) In step IV, instead of doing simple sort, we can apply some clever technique to do it in linear time. Hint: We know that all digits are linearly sorted in reverse order except one digit which was swapped.

With above optimizations, we can say that the time complexity of this method is O(n).

This article is contributed by **Rahul Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Mathematical
MathematicalAlgo
number-digits

Count Possible Decodings of a given Digit Sequence

Let 1 represent 'A', 2 represents 'B', etc. Given a digit sequence, count the number of possible decodings of the given digit sequence.

Examples:

```

Input: digits[] = "121"
Output: 3
// The possible decodings are "ABA", "AU", "LA"

Input: digits[] = "1234"
Output: 3
// The possible decodings are "ABCD", "LCD", "AWD"

```

An empty digit sequence is considered to have one decoding. It may be assumed that the input contains valid digits from 0 to 9 and there are no leading 0's, no extra trailing 0's and no two or more consecutive 0's.

We strongly recommend to minimize the browser and try this yourself first.

This problem is recursive and can be broken in sub-problems. We start from end of the given digit sequence. We initialize the total count of decodings as 0. We recur for two subproblems.

- 1) If the last digit is non-zero, recur for remaining (n-1) digits and add the result to total count.
- 2) If the last two digits form a valid character (or smaller than 27), recur for remaining (n-2) digits and add the result to total count.

Following is C++ implementation of the above approach.

```

// A naive recursive C++ implementation to count number of decodings
// that can be formed from a given digit sequence
#include <iostream>
#include <cstring>
using namespace std;

// Given a digit sequence of length n, returns count of possible
// decodings by replacing 1 with A, 2 with B, ... 26 with Z
int countDecoding(char *digits, int n)
{
    // base cases
    if (n == 0 || n == 1)
        return 1;

    int count = 0; // Initialize count

    // If the last digit is not 0, then last digit must add to
    // the number of words
    if (digits[n-1] > '0')
        count = countDecoding(digits, n-1);

    // If the last two digits form a number smaller than or equal to 26,
    // then consider last two digits and recur
    if ((digits[n-2] < '2' || (digits[n-2] == '2' && digits[n-1] < '7')) )
        count += countDecoding(digits, n-2);

    return count;
}

// Driver program to test above function

```

```

int main()
{
    char digits[] = "1234";
    int n = strlen(digits);
    cout << "Count is " << countDecoding(digits, n);
    return 0;
}

```

Output:

```
Count is 3
```

The time complexity of above the code is exponential. If we take a closer look at the above program, we can observe that the recursive solution is similar to Fibonacci Numbers. Therefore, we can optimize the above solution to work in $O(n)$ time using [Dynamic Programming](#). Following is C++ implementation for the same.

```

// A Dynamic Programming based C++ implementation to count decodings
#include <iostream>
#include <cstring>
using namespace std;

// A Dynamic Programming based function to count decodings
int countDecodingDP(char *digits, int n)
{
    int count[n+1]; // A table to store results of subproblems
    count[0] = 1;
    count[1] = 1;

    for (int i = 2; i <= n; i++)
    {
        count[i] = 0;

        // If the last digit is not 0, then last digit must add to
        // the number of words
        if (digits[i-1] > '0')
            count[i] = count[i-1];

        // If second last digit is smaller than 2 and last digit is
        // smaller than 7, then last two digits form a valid character
        if (digits[i-2] < '2' || (digits[i-2] == '2' && digits[i-1] < '7'))
            count[i] += count[i-2];
    }
    return count[n];
}

// Driver program to test above function
int main()
{
    char digits[] = "1234";
    int n = strlen(digits);
    cout << "Count is " << countDecodingDP(digits, n);
    return 0;
}

```

Output:

```
Count is 3
```

Time Complexity of the above solution is $O(n)$ and it requires $O(n)$ auxiliary space. We can reduce auxiliary space to $O(1)$ by using space optimized version discussed in the [Fibonacci Number Post](#).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

[Dynamic Programming](#)
[Dynamic Programming](#)
[Fibonacci](#)
[MathematicalAlgo](#)

Calculate the angle between hour hand and minute hand

This problem is known as [Clock angle problem](#) where we need to find angle between hands of an analog clock at a given time.

Examples:

```

Input: h = 12:00, m = 30.00
Output: 165 degree

Input: h = 3:00, m = 30.00
Output: 75 degree

```

The idea is to take 12:00 ($h = 12$, $m = 0$) as a reference. Following are detailed steps.

- 1) Calculate the angle made by hour hand with respect to 12:00 in h hours and m minutes.
- 2) Calculate the angle made by minute hand with respect to 12:00 in h hours and m minutes.
- 3) The difference between two angles is the angle between two hands.

How to calculate the two angles with respect to 12:00?

The minute hand moves 360 degree in 60 minute(or 6 degree in one minute) and hour hand moves 360 degree in 12 hours(or 0.5 degree in 1 minute). In h hours and m minutes, the minute hand would move $(h*60 + m)*6$ and hour hand would move $(h*60 + m)*0.5$.

```

// C program to find angle between hour and minute hands
#include <stdio.h>
#include <stdlib.h>

// Utility function to find minimum of two integers
int min(int x, int y) { return (x < y)? x: y; }

int calcAngle(double h, double m)
{
    // validate the input
    if (h <0 || m < 0 || h >12 || m > 60)
        printf("Wrong input");

```

```

if (h == 12) h = 0;
if (m == 60) m = 0;

// Calculate the angles moved by hour and minute hands
// with reference to 12:00
int hour_angle = 0.5 * (h*60 + m);
int minute_angle = 6*m;

// Find the difference between two angles
int angle = abs(hour_angle - minute_angle);

// Return the smaller angle of two possible angles
angle = min(360-angle, angle);

return angle;
}

// Driver program to test above function
int main()
{
    printf("%d \n", calcAngle(9, 60));
    printf("%d \n", calcAngle(3, 30));
    return 0;
}

```

Output:

```

90
75

```

Exercise: Find all times when hour and minute hands get superimposed.

This article is contributed by **Ashish Bansal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Mathematical
MathematicalAlgo

Count number of binary strings without consecutive 1's

Given a positive integer N, count all possible distinct binary strings of length N such that there are no consecutive 1's.

Examples:

```

Input: N = 2
Output: 3
// The 3 strings are 00, 01, 10

Input: N = 3
Output: 5
// The 5 strings are 000, 001, 010, 100, 101

```

This problem can be solved using Dynamic Programming. Let $a[i]$ be the number of binary strings of length i which do not contain any two consecutive 1's and which end in 0. Similarly, let $b[i]$ be the number of such strings which end in 1. We can append either 0 or 1 to a string ending in 0, but we can only append 0 to a string ending in 1. This yields the recurrence relation:

```

a[i] = a[i - 1] + b[i - 1]
b[i] = a[i - 1]

```

The base cases of above recurrence are $a[1] = b[1] = 1$. The total number of strings of length i is just $a[i] + b[i]$.

Following is the implementation of above solution. In the following implementation, indexes start from 0. So $a[i]$ represents the number of binary strings for input length $i+1$. Similarly, $b[i]$ represents binary strings for input length $i+1$.

C++

```

// C++ program to count all distinct binary strings
// without two consecutive 1's
#include <iostream>
using namespace std;

int countStrings(int n)
{
    int a[n], b[n];
    a[0] = b[0] = 1;
    for (int i = 1; i < n; i++)
    {
        a[i] = a[i-1] + b[i-1];
        b[i] = a[i-1];
    }
    return a[n-1] + b[n-1];
}

// Driver program to test above functions
int main()
{
    cout << countStrings(3) << endl;
    return 0;
}

```

Java

```

class Subset_sum
{
    static int countStrings(int n)
    {

```

```

int a[] = new int[n];
int b[] = new int[n];
a[0] = b[0] = 1;
for (int i = 1; i < n; i++)
{
    a[i] = a[i-1] + b[i-1];
    b[i] = a[i-1];
}
return a[n-1] + b[n-1];
}
/* Driver program to test above function */
public static void main (String args[])
{
    System.out.println(countStrings(3));
}
/* This code is contributed by Rajat Mishra */

```

Output:

5

Source:

courses.csail.mit.edu/6.006/oldquizzes/solutions/q2-f2009-sol.pdf

If we take a closer look at the pattern, we can observe that the count is actually $(n+2)^\text{th}$ Fibonacci number for $n \geq 1$. The Fibonacci Numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 141,

```

n = 1, count = 2 = fib(3)
n = 2, count = 3 = fib(4)
n = 3, count = 5 = fib(5)
n = 4, count = 8 = fib(6)
n = 5, count = 13 = fib(7)
.....

```

Therefore we can count the strings in $O(\log n)$ time also using the method 5 [here](#).

This article is contributed by **Rahul Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Dynamic Programming
binary-string
Dynamic Programming
Fibonacci

Find the smallest number whose digits multiply to a given number n

Given a number 'n', find the smallest number 'p' such that if we multiply all digits of 'p', we get 'n'. The result 'p' should have minimum two digits.

Examples:

```

Input: n = 36
Output: p = 49
// Note that 4*9 = 36 and 49 is the smallest such number

Input: n = 100
Output: p = 455
// Note that 4*5*5 = 100 and 455 is the smallest such number

Input: n = 1
Output: p = 11
// Note that 1*1 = 1

Input: n = 13
Output: Not Possible

```

For a given n, following are the two cases to be considered.

Case 1: n When n is smaller than 10, the output is always n+10. For example for n = 7, output is 17. For n = 9, output is 19.

Case 2: n >= 10 Find all factors of n which are between 2 and 9 (both inclusive). The idea is to start searching from 9 so that the number of digits in result are minimized. For example 9 is preferred over 33 and 8 is preferred over 24.

Store all found factors in an array. The array would contain digits in non-increasing order, so finally print the array in reverse order.

Following is C implementation of above concept.

```

#include<stdio.h>

// Maximum number of digits in output
#define MAX 50

// prints the smallest number whose digits multiply to n
void findSmallest(int n)
{
    int i, j=0;
    int res[MAX]; // To store digits of result in reverse order

    // Case 1: If number is smaller than 10
    if (n < 10)
    {
        printf("%d", n+10);
        return;
    }

    // Case 2: Start with 9 and try every possible digit
    for (i=9; i>1; i--)
    {
        // If current digit divides n, then store all
        // occurrences of current digit in res
        while (n%i == 0)
        {
            n = n/i;
            res[j] = i;
            j++;
        }
    }
}

```

```

    }

}

// If n could not be broken in form of digits (prime factors of n
// are greater than 9)
if (n > 10)
{
    printf("Not possible");
    return;
}

// Print the result array in reverse order
for (i=j-1; i>=0; i--)
    printf("%d", res[i]);
}

// Driver program to test above function
int main()
{
    findSmallest(7);
    printf("\n");

    findSmallest(36);
    printf("\n");

    findSmallest(13);
    printf("\n");

    findSmallest(100);
    return 0;
}

```

Output:

```

17
49
Not possible
455

```

This article is contributed by **Ashish Bansal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Mathematical
MathematicalAlgo
number-digits

Draw a circle without floating point arithmetic

Given a radius of a circle, draw the circle without using floating point arithmetic.

Following program uses a simple concept. Let the radius of the circle be r . Consider a square of size $(2r+1) \times (2r+1)$ around the circle to be drawn. Now walk through every point inside the square. For every point (x, y) , if (x, y) lies inside the circle ($x^2 + y^2 \leq r^2$) { void drawCircle(int r) { // Consider a rectangle of size N*N int N = 2*r+1; int x, y; // Coordinates inside the rectangle // Draw a square of size N*N. for (int i = 0; i < N; i++) { for (int j = 0; j < N; j++) { // Start from the left most corner point x = i-r; y = j-r; // If this point is inside the circle, print it if (x*x + y*y <= r*r) printf("."); else // If outside the circle, print space printf(" "); } } } // Driver Program to test above function int main() { drawCircle(8); return 0; }

Output:



Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Articles

How to check if an instance of 8 puzzle is solvable?

What is 8 puzzle?

Given a 3×3 board with 8 tiles (every tile has one number from 1 to 8) and one empty space. The objective is to place the numbers on tiles in order using the empty space. We can slide four adjacent (left, right, above and below) tiles into the empty space.



How to find if given state is solvable?

Following are two examples, the first example can reach goal state by a series of slides. The second example cannot.

8puzzle



Following is simple rule to check if a 8 puzzle is solvable.

It is not possible to solve an instance of 8 puzzle if number of inversions is odd in the input state. In the examples given in above figure, the first example has 10 inversions, therefore solvable. The second example has 11 inversions, therefore unsolvable.

What is inversion?

A pair of tiles form an inversion if the the values on tiles are in reverse order of their appearance in goal state. For example, the following instance of 8 puzzle has two inversions, (8, 6) and (8, 7).

```
1 2 3  
4 _ 5  
8 6 7
```

Following is a simple C++ program to check whether a given instance of 8 puzzle is solvable or not. The idea is simple, we count inversions in the given 8 puzzle.

```
// C++ program to check if a given instance of 8 puzzle is solvable or not  
#include <iostream>  
using namespace std;  
  
// A utility function to count inversions in given array 'arr[]'  
int getInvCount(int arr[]){  
{  
    int inv_count = 0;  
    for (int i = 0; i < 9 - 1; i++)  
        for (int j = i+1; j < 9; j++)  
            // Value 0 is used for empty space  
            if (arr[j] && arr[i] && arr[i] > arr[j])  
                inv_count++;  
    return inv_count;  
}  
  
// This function returns true if given 8 puzzle is solvable.  
bool isSolvable(int puzzle[3][3])  
{  
    // Count inversions in given 8 puzzle  
    int invCount = getInvCount((int *)puzzle);  
  
    // return true if inversion count is even.  
    return (invCount%2 == 0);  
}  
  
/* Driver progra to test above functions */  
int main(int argc, int* args)  
{  
    int puzzle[3][3] = {{1, 8, 2},  
                       {0, 4, 3}, // Value 0 is used for empty space  
                       {7, 6, 5}};  
    isSolvable(puzzle)? cout << "Solvable":  
                      cout << "Not Solvable";  
    return 0;  
}
```

Output:

```
Solvable
```

Note that the above implementation uses simple algorithm for inversion count. It is done this way for simplicity. The code can be optimized to O(nLogn) using the [merge sort based algorithm for inversion count](#).

How does this work?

The idea is based on the fact of parity of inversions remains same after a set of moves, i.e., if the inversion count is odd in initial stage, then it remain odd after any sequence of moves and if the inversion count is even, then it remains even after any sequence of moves. In the goal state, there are 0 inversions. So we can reach goal state only from a state which has even inversion count.

How parity of inversion count is invariant?

When we slide a tile, we either make a row move (moving a left or right tile into the blank space), or make a column move (moving a up or down tile to the blank space).

a) A row move doesn't change the inversion count. See following example

```
1 2 3 Row Move 1 2 3  
4 _ 5 -----> _ 4 5  
8 6 7         8 6 7  
Inversion count remains 2 after the move
```

```
1 2 3 Row Move 1 2 3  
4 _ 5 -----> 4 5 _  
8 6 7         8 6 7  
Inversion count remains 2 after the move
```

b) A column move does one of the following three.

.....(i) Increases inversion count by 2. See following example.

```
1 2 3 Column Move 1 _ 3  
4 _ 5 -----> 4 2 5  
8 6 7         8 6 7  
Inversion count increases by 2 (changes from 2 to 4)
```

.....(ii) Decreases inversion count by 2

```
1 3 4 Column Move 1 3 4
```

```

5 _ 6 -----> 5 2 6
7 2 8         7 _ 8
Inversion count decreases by 2 (changes from 5 to 3)

```

.....(iii) Keeps the inversion count same.

```

1 2 3 Column Move 1 2 3
4 _ 5 -----> 4 6 5
7 6 8         7 _ 8
Inversion count remains 1 after the move

```

So if a move either increases/decreases inversion count by 2, or keeps the inversion count same, then it is not possible to change parity of a state by any sequence of row/column moves.

Exercise: How to check if a given instance of 15 puzzle is solvable or not. In a 15 puzzle, we have 4×4 board where 15 tiles have a number and one empty space. Note that the above simple rules of inversion count don't directly work for 15 puzzle, the rules need to be modified for 15 puzzle.

Related Article:

[How to check if an instance of 15 puzzle is solvable?](#)

This article is contributed by Ishan. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Mathematical
MathematicalAlgo

Birthday Paradox

How many people must be there in a room to make the probability 100% that two people in the room have same birthday?

Answer: 367 (since there are 366 possible birthdays, including February 29).

The above question was simple. Try the below question yourself.

How many people must be there in a room to make the probability 50% that two people in the room have same birthday?

Answer: 23

The number is surprisingly very low. In fact, we need only 70 people to make the probability 99.9 %.

Let us discuss the generalized formula.

What is the probability that two persons among n have same birthday?

Let the probability that two people in a room with n have same birthday be P(same). P(Same) can be easily evaluated in terms of P(different) where P(different) is the probability that all of them have different birthday.

$$P(\text{same}) = 1 - P(\text{different})$$

P(different) can be written as $1 \times (364/365) \times (363/365) \times (362/365) \times \dots \times (1 - (n-1)/365)$

How did we get the above expression?

Persons from first to last can get birthdays in following order for all birthdays to be distinct:

The first person can have any birthday among 365

The second person should have a birthday which is not same as first person

The third person should have a birthday which is not same as first two persons.

.....

.....

The n'th person should have a birthday which is not same as any of the earlier considered (n-1) persons.

Approximation of above expression

The above expression can be approximated using Taylor's Series.



provides a first-order approximation for ex for x

To apply this approximation to the first expression derived for p(different), set x = -a / 365. Thus,



The above expression derived for p(different) can be written as

$$1 \times (1 - 1/365) \times (1 - 2/365) \times (1 - 3/365) \times \dots \times (1 - (n-1)/365)$$

By putting the value of $1 - a/365$ as $e^{-a/365}$, we get following.

bp4



Therefore,

$$p(\text{same}) = 1 - p(\text{different})$$

An even coarser approximation is given by

$$p(\text{same})$$

By taking Log on both sides, we get the reverse formula.



Using the above approximate formula, we can approximate number of people for a given probability. For example the following C++ function find() returns the smallest n for which the probability is greater than the given p.

C++ Implementation of approximate formula.

The following is C++ program to approximate number of people for a given probability.

```

// C++ program to approximate number of people in Birthday Paradox
// problem
#include <cmath>
#include <iostream>
using namespace std;

// Returns approximate number of people for a given probability
int find(double p)

```

```
{
    return ceil(sqrt(2*365*log(1/(1-p))));
}

int main()
{
    cout << find(0.70);
}
```

Output:

30

Source:

http://en.wikipedia.org/wiki/Birthday_problem

Applications:

- 1) Birthday Paradox is generally discussed with hashing to show importance of collision handling even for a small set of keys.
- 2) Birthday Attack

This article is contributed by **Shubham**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Mathematical
Randomized
MathematicalAlgo

Multiply two polynomials

Given two polynomials represented by two arrays, write a function that multiplies given two polynomials.

Example:

```
Input: A[] = {5, 0, 10, 6}
       B[] = {1, 2, 4}
Output: prod[] = {5, 10, 30, 26, 52, 24}
```

The first input array represents "5 + 0x^1 + 10x^2 + 6x^3"
The second array represents "1 + 2x^1 + 4x^2"
And Output is "5 + 10x^1 + 30x^2 + 26x^3 + 52x^4 + 24x^5"

We strongly recommend that you click here and practice it, before moving on to the solution.

A simple solution is to one by one consider every term of first polynomial and multiply it with every term of second polynomial. Following is algorithm of this simple method.

```
multiply(A[0..m-1], B[0..n-1])
1) Create a product array prod[] of size m+n-1.
2) Initialize all entries in prod[] as 0.
3) Travers array A[] and do following for every element A[i]
...{3.a} Traverse array B[] and do following for every element B[j]
      prod[i+j] = prod[i+j] + A[i] * B[j]
4) Return prod[].
```

The following is C++ implementation of above algorithm.

```
// Simple C++ program to multiply two polynomials
#include <iostream>
using namespace std;

// A[] represents coefficients of first polynomial
// B[] represents coefficients of second polynomial
// m and n are sizes of A[] and B[] respectively
int *multiply(int A[], int B[], int m, int n)
{
    int *prod = new int[m+n-1];

    // Initialize the product polynomial
    for (int i = 0; i<m-n-1; i++)
        prod[i] = 0;

    // Multiply two polynomials term by term

    // Take ever term of first polynomial
    for (int i=0; i<m; i++)
    {
        // Multiply the current term of first polynomial
        // with every term of second polynomial.
        for (int j=0; j<n; j++)
            prod[i+j] += A[i]*B[j];
    }

    return prod;
}

// A utility function to print a polynomial
void printPoly(int poly[], int n)
{
    for (int i=0; i<n; i++)
    {
        cout << poly[i];
        if (i != 0)
            cout << "x^" << i ;
        if (i != n-1)
            cout << "+" ;
    }
}

// Driver program to test above functions
```

```

int main()
{
    // The following array represents polynomial 5 + 10x^2 + 6x^3
    int A[] = {5, 0, 10, 6};

    // The following array represents polynomial 1 + 2x + 4x^2
    int B[] = {1, 2, 4};
    int m = sizeof(A)/sizeof(A[0]);
    int n = sizeof(B)/sizeof(B[0]);

    cout << "First polynomial is \n";
    printPoly(A, m);
    cout << "\nSecond polynomial is \n";
    printPoly(B, n);

    int *prod = multiply(A, B, m, n);

    cout << "\nProduct polynomial is \n";
    printPoly(prod, m+n-1);

    return 0;
}

```

Output

```

First polynomial is
5 + 0x^1 + 10x^2 + 6x^3
Second polynomial is
1 + 2x^1 + 4x^2
Product polynomial is
5 + 10x^1 + 30x^2 + 26x^3 + 52x^4 + 24x^5

```

Time complexity of the above solution is $O(mn)$. If size of two polynomials same, then time complexity is $O(n^2)$.

Can we do better?

There are methods to do multiplication faster than $O(n^2)$ time. These methods are mainly based on divide and conquer. Following is one simple method that divides the given polynomial (of degree n) into two polynomials one containing lower degree terms(lower than $n/2$) and other containing higher degree terms (higher than or equal to $n/2$)

Let the two given polynomials be A and B.
For simplicity, Let us assume that the given two polynomials are of same degree and have degree in powers of 2, i.e., $n = 2^l$

The polynomial 'A' can be written as $A_0 + A_1 \cdot x^{n/2}$
The polynomial 'B' can be written as $B_0 + B_1 \cdot x^{n/2}$

For example $1 + 10x + 6x^2 - 4x^3 + 5x^4$ can be written as $(1 + 10x) + (6 - 4x + 5x^2)x^2$

$$\begin{aligned} A \cdot B &= (A_0 + A_1 \cdot x^{n/2}) \cdot (B_0 + B_1 \cdot x^{n/2}) \\ &= A_0 \cdot B_0 + A_0 \cdot B_1 \cdot x^{n/2} + A_1 \cdot B_0 \cdot x^{n/2} + A_1 \cdot B_1 \cdot x^n \\ &= A_0 \cdot B_0 + (A_0 \cdot B_1 + A_1 \cdot B_0) \cdot x^{n/2} + A_1 \cdot B_1 \cdot x^n \end{aligned}$$

So the above divide and conquer approach requires 4 multiplications and $O(n)$ time to add all 4 results. Therefore the time complexity is $T(n) = 4T(n/2) + O(n)$. The solution of the recurrence is $O(n^2)$ which is same as the above simple solution.

The idea is to reduce number of multiplications to 3 and make the recurrence as $T(n) = 3T(n/2) + O(n)$

How to reduce number of multiplications?

This requires a little trick similar to [Strassen's Matrix Multiplication](#). We do following 3 multiplications.

```

X = (A0 + A1)*(B0 + B1) // First Multiplication
Y = A0B0 // Second
Z = A1B1 // Third

```

The missing middle term in above multiplication equation $A_0 \cdot B_0 + (A_0 \cdot B_1 + A_1 \cdot B_0) \cdot x^{n/2} + A_1 \cdot B_1 \cdot x^n$ can be obtained using below.
 $A_0B_1 + A_1B_0 = X - Y - Z$

So the time taken by this algorithm is $T(n) = 3T(n/2) + O(n)$

The solution of above recurrence is $O(n^{1.5})$ which is better than $O(n^2)$.

We will soon be discussing implementation of above approach.

There is a $O(n\log n)$ algorithm also that uses Fast Fourier Transform to multiply two polynomials (Refer [this](#) and [this](#) for details)

Sources:

<http://www.cse.ust.hk/~dekai/271/notes/L03/L03.pdf>

This article is contributed by Harsh. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Divide and Conquer
Mathematical
Divide and Conquer

Count Distinct Non-Negative Integer Pairs (x, y) that Satisfy the Inequality $x*x + y*y$

Given a positive number n , count all distinct Non-Negative Integer pairs (x, y) that satisfy the inequality $x*x + y*y$

Examples:

```

Input: n = 5
Output: 6
The pairs are (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (0, 2)

```

```

Input: n = 6
Output: 8
The pairs are (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (0, 2),
(1, 2), (2, 1)

```

A **Simple Solution** is to run two loops. The outer loop goes for all possible values of x (from 0 to \sqrt{n}). The inner loops picks all possible values of y for current value of x (picked by outer loop). Following is C++ implementation of simple solution.

```

#include <iostream>
using namespace std;

// This function counts number of pairs (x, y) that satisfy
// the inequality x*x + y*y < n.
int countSolutions(int n)
{
    int res = 0;
    for (int x = 0; x*x < n; x++)
        for (int y = 0; x*x + y*y < n; y++)
            res++;
    return res;
}

// Driver program to test above function
int main()
{
    cout << "Total Number of distinct Non-Negative pairs is "
         << countSolutions(6) << endl;
    return 0;
}

```

Output:

```
Total Number of distinct Non-Negative pairs is 8
```

An upper bound for time complexity of the above solution is $O(n)$. The outer loop runs \sqrt{n} times. The inner loop runs at most \sqrt{n} times.

Using an **Efficient Solution**, we can find the count in $O(\sqrt{n})$ time. The idea is to first find the count of all y values corresponding to the 0 value of x. Let count of distinct y values be yCount. We can find yCount by running a loop and comparing yCount*yCount with n.

After we have initial yCount, we can one by one increase value of x and find the next value of yCount by reducing yCount.

```

// An efficient C program to find different (x, y) pairs that
// satisfy x*x + y*y < n.
#include <iostream>
using namespace std;

// This function counts number of pairs (x, y) that satisfy
// the inequality x*x + y*y < n.
int countSolutions(int n)
{
    int x = 0, yCount, res = 0;

    // Find the count of different y values for x = 0.
    for (yCount = 0; yCount*yCount < n; yCount++);

    // One by one increase value of x, and find yCount for
    // current x. If yCount becomes 0, then we have reached
    // maximum possible value of x.
    while (yCount != 0)
    {
        // Add yCount (count of different possible values of y
        // for current x) to result
        res += yCount;

        // Increment x
        x++;

        // Update yCount for current x. Keep reducing yCount while
        // the inequality is not satisfied.
        while (yCount != 0 && (x*x + (yCount-1)*(yCount-1) >= n))
            yCount--;
    }

    return res;
}

// Driver program to test above function
int main()
{
    cout << "Total Number of distinct Non-Negative pairs is "
         << countSolutions(6) << endl;
    return 0;
}

```

Output:

```
Total Number of distinct Non-Negative pairs is 8
```

Time Complexity of the above solution seems more but if we take a closer look, we can see that it is $O(\sqrt{n})$. In every step inside the inner loop, value of yCount is decremented by 1. The value yCount can decrement at most $O(\sqrt{n})$ times as yCount is count y values for $x = 0$. In the outer loop, the value of x is incremented. The value of x can also increment at most $O(\sqrt{n})$ times as the last x is for yCount equals to 1.

This article is contributed by **Sachin Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Mathematical
MathematicalAlgo

Count ways to reach the n'th stair

There are n stairs, a person standing at the bottom wants to reach the top. The person can climb either 1 stair or 2 stairs at a time. Count the number of ways, the person can reach the top.



Consider the example shown in diagram. The value of n is 3. There are 3 ways to reach the top. The diagram is taken from [Easier Fibonacci puzzles](#)

More Examples:

Input: n = 1
Output: 1
There is only one way to climb 1 stair

Input: n = 2
Output: 2
There are two ways: (1, 1) and (2)

Input: n = 4
Output: 5
(1, 1, 1, 1), (1, 1, 2), (2, 1, 1), (1, 2, 1), (2, 2)

We strongly recommend that you click here and practice it, before moving on to the solution.

We can easily find recursive nature in above problem. The person can reach n'th stair from either (n-1)'th stair or from (n-2)'th stair. Let the total number of ways to reach n't stair be 'ways(n)'. The value of 'ways(n)' can be written as following.

$$\text{ways}(n) = \text{ways}(n-1) + \text{ways}(n-2)$$

The above expression is actually the expression for [Fibonacci numbers](#), but there is one thing to notice, the value of ways(n) is equal to fibonacci(n+1).

ways(1) = fib(2) = 1
ways(2) = fib(3) = 2
ways(3) = fib(4) = 3

So we can use function for fibonacci numbers to find the value of ways(n). Following is C++ implementation of the above idea.

C

```
// A C program to count number of ways to reach n't stair when
// a person can climb 1, 2, ...m stairs at a time.
#include<stdio.h>

// A simple recursive program to find n'th fibonacci number
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}

// Returns number of ways to reach s'th stair
int countWays(int s)
{
    return fib(s + 1);
}

// Driver program to test above functions
int main ()
{
    int s = 4;
    printf("Number of ways = %d", countWays(s));
    getchar();
    return 0;
}
```

Java

```
class stairs
{
    // A simple recursive program to find n'th fibonacci number
    static int fib(int n)
    {
        if (n <= 1)
            return n;
        return fib(n-1) + fib(n-2);
    }

    // Returns number of ways to reach s'th stair
    static int countWays(int s)
    {
        return fib(s + 1);
    }
}
```

```

}

/* Driver program to test above function */
public static void main (String args[])
{
    int s = 4;
    System.out.println("Number of ways = "+ countWays(s));
}
/* This code is contributed by Rajat Mishra */

```

Output:

```
Number of ways = 5
```

The time complexity of the above implementation is exponential (golden ratio raised to power n). It can be optimized to work in O(Logn) time using the previously [discussed](#) Fibonacci function optimizations.

Generalization of the above problem

How to count number of ways if the person can climb up to m stairs for a given value m? For example if m is 4, the person can climb 1 stair or 2 stairs or 3 stairs or 4 stairs at a time.

We can write the recurrence as following.

```
ways(n, m) = ways(n-1, m) + ways(n-2, m) + ... ways(n-m, m)
```

Following is C++ implementation of above recurrence.

C

```

// A C program to count number of ways to reach n'th stair when
// a person can climb either 1 or 2 stairs at a time
#include<stdio.h>

// A recursive function used by countWays
int countWaysUtil(int n, int m)
{
    if (n <= 1)
        return n;
    int res = 0;
    for (int i = 1; i<=m && i<=n; i++)
        res += countWaysUtil(n-i, m);
    return res;
}

// Returns number of ways to reach s'th stair
int countWays(int s, int m)
{
    return countWaysUtil(s+1, m);
}

// Driver program to test above functions
int main ()
{
    int s = 4, m = 2;
    printf("Nuber of ways = %d", countWays(s, m));
    return 0;
}

```

Java

```

class stairs
{
    // A recursive function used by countWays
    static int countWaysUtil(int n, int m)
    {
        if (n <= 1)
            return n;
        int res = 0;
        for (int i = 1; i<=m && i<=n; i++)
            res += countWaysUtil(n-i, m);
        return res;
    }

    // Returns number of ways to reach s'th stair
    static int countWays(int s, int m)
    {
        return countWaysUtil(s+1, m);
    }

    /* Driver program to test above function */
    public static void main (String args[])
    {
        int s = 4, m = 2;
        System.out.println("Number of ways = "+ countWays(s,m));
    }
}
/* This code is contributed by Rajat Mishra */

```

Output:

```
Number of ways = 5
```

The time complexity of above solution is exponential. It can be optimized to O(mn) by using dynamic programming. Following is dynamic programming based solution. We build a table res[] in bottom up manner.

C

```

// A C program to count number of ways to reach n'th stair when
// a person can climb 1, 2, ..m stairs at a time
#include<stdio.h>

// A recursive function used by countWays
int countWaysUtil(int n, int m)
{
    int res[n];
    res[0] = 1; res[1] = 1;
    for (int i=2; i<n; i++)
    {
        res[i] = 0;
        for (int j=1; j<=m && j<=i; j++)
            res[i] += res[i-j];
    }
    return res[n-1];
}

// Returns number of ways to reach s'th stair
int countWays(int s, int m)
{
    return countWaysUtil(s+1, m);
}

// Driver program to test above functions
int main ()
{
    int s = 4, m = 2;
    printf("Number of ways = %d", countWays(s, m));
    return 0;
}

```

Java

```

class stairs
{
    // A recursive function used by countWays
    static int countWaysUtil(int n, int m)
    {
        if (n <= 1)
            return n;
        int res = 0;
        for (int i = 1; i<=m && i<=n; i++)
            res += countWaysUtil(n-i, m);
        return res;
    }
    // Returns number of ways to reach s'th stair
    static int countWays(int s, int m)
    {
        return countWaysUtil(s+1, m);
    }

    /* Driver program to test above function */
    public static void main (String args[])
    {
        int s = 4, m = 2;
        System.out.println("Number of ways = " + countWays(s,m));
    }
    /* This code is contributed by Rajat Mishra */
}

```

Output:

Number of ways = 5

This article is contributed by **Abhishek**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Mathematical

Replace all '0' with '5' in an input Integer

Given a integer as a input and replace all the '0' with '5' in the integer.

Examples:

```

102 - 152
1020 - 1525

```

Use of array to store all digits is not allowed.

Source: [Amazon interview Experience | Set 136 \(For SDE-T\)](#)

We strongly recommend to minimize the browser and try this yourself first.

The idea is simple, we get the last digit using mod operator '%'. If the digit is 0, we replace it with 5, otherwise keep it as it is. Then we recur for remaining digits.

Following is C implementation of the above idea.

```

// C program to replace all '0' with '5' in an input Integer
#include<stdio.h>

// A recursive function to replace all 0s with 5s in an input number
// It doesn't work if input number itself is 0.
int convert0To5Rec(int num)
{
    // Base case for recursion termination
    if (num == 0)
        return 0;

    // Extract the last digit and change it if needed

```

```

int digit = num % 10;
if (digit == 0)
    digit = 5;

// Convert remaining digits and append the last digit
return convert0To5Rec(num/10) * 10 + digit;
}

// It handles 0 and calls convert0To5Rec() for other numbers
int convert0To5(int num)
{
    if (num == 0)
        return 5;
    else return convert0To5Rec(num);
}

// Driver program to test above function
int main()
{
    int num = 10120;
    printf("%d", convert0To5(num));
    return 0;
}

```

Output:

15125

This article is contributed by **Sal Kiran**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Algorithms

Program to add two polynomials

Given two polynomials represented by two arrays, write a function that adds given two polynomials.

Example:

```

Input: A[] = {5, 0, 10, 6}
       B[] = {1, 2, 4}
Output: sum[] = {5, 10, 30, 26, 52, 24}

```

The first input array represents "5 + 0x^1 + 10x^2 + 6x^3"
The second array represents "1 + 2x^1 + 4x^2"
And Output is "6 + 2x^1 + 14x^2 + 6x^3"

We strongly recommend to minimize your browser and try this yourself first.

Addition is simpler than [multiplication of polynomials](#). We initialize result as one of the two polynomials, then we traverse the other polynomial and add all terms to the result.

```

add(A[0..m-1], B[0..n-1])
1) Create a sum array sum[] of size equal to maximum of 'm' and 'n'
2) Copy A[] to sum[].
3) Travers array B[] and do following for every element B[i]
    sum[i] = sum[i] + B[i]
4) Return sum[].

```

The following is C++ implementation of above algorithm.

```

// Simple C++ program to add two polynomials
#include <iostream>
using namespace std;

// A utility function to return maximum of two integers
int max(int m, int n) { return (m > n)? m: n; }

// A[] represents coefficients of first polynomial
// B[] represents coefficients of second polynomial
// m and n are sizes of A[] and B[] respectively
int *add(int A[], int B[], int m, int n)
{
    int size = max(m, n);
    int *sum = new int[size];

    // Initialize the product polynomial
    for (int i = 0; i < m; i++)
        sum[i] = A[i];

    // Take every term of first polynomial
    for (int i=0; i < n; i++)
        sum[i] += B[i];

    return sum;
}

// A utility function to print a polynomial
void printPoly(int poly[], int n)
{
    for (int i=0; i < n; i++)
    {
        cout << poly[i];
        if (i != 0)
            cout << "x^" << i ;
        if (i != n-1)
            cout << " + ";
    }
}

```

```

    }

// Driver program to test above functions
int main()
{
    // The following array represents polynomial 5 + 10x^2 + 6x^3
    int A[] = {5, 0, 10, 6};

    // The following array represents polynomial 1 + 2x + 4x^2
    int B[] = {1, 2, 4};
    int m = sizeof(A)/sizeof(A[0]);
    int n = sizeof(B)/sizeof(B[0]);

    cout << "First polynomial is \n";
    printPoly(A, m);
    cout << "Second polynomial is \n";
    printPoly(B, n);

    int *sum = add(A, B, m, n);
    int size = max(m, n);

    cout << "Sum polynomial is \n";
    printPoly(sum, size);

    return 0;
}

```

Output:

```

First polynomial is
5 + 0x^1 + 10x^2 + 6x^3
Second polynomial is
1 + 2x^1 + 4x^2
Sum polynomial is
6 + 2x^1 + 14x^2 + 6x^3

```

Time complexity of the above algorithm and program is $O(m+n)$ where m and n are orders of two given polynomials.

This article is contributed by Harsh. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See [Placement Course](#) for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: [Algorithms](#)

Print first k digits of 1/n where n is a positive integer

Given a positive integer n , print first k digits after point in value of $1/n$. Your program should avoid overflow and floating point arithmetic.

Examples:

```

Input: n = 3, k = 3
Output: 333

```

```

Input: n = 50, k = 4
Output: 0200

```

We strongly recommend to minimize the browser and try this yourself first.

Let us consider an example $n = 7$, $k = 3$. The first digit of $1/7$ is '1', it can be obtained by doing integer value of $10/7$. Remainder of $10/7$ is 3. Next digit is 4 which can be obtained by taking integer value of $30/7$. Remainder of $30/7$ is 2. Next digits is 2 which can be obtained by taking integer value of $20/7$.

```

#include <iostream>
using namespace std;

// Function to print first k digits after dot in value
// of 1/n. n is assumed to be a positive integer.
void print(int n, int k)
{
    int rem = 1; // Initialize remainder

    // Run a loop k times to print k digits
    for (int i = 0; i < k; i++)
    {
        // The next digit can always be obtained as
        // doing (10*rem)/10
        cout << (10 * rem) / n;

        // Update remainder
        rem = (10 * rem) % n;
    }
}

// Driver program to test above function
int main()
{
    int n = 7, k = 3;
    print(n, k);
    cout << endl;

    n = 21, k = 4;
    print(n, k);

    return 0;
}

```

Output:

Reference:

Algorithms And Programming: Problems And Solutions by Alexander Shen

This article is contributed by **Sachin**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

See Placement Course for placement preparation, [GATE Corner](#) for GATE CS Preparation and [Quiz Corner](#) for all Quizzes on GeeksQuiz.
Category: Algorithms

Given a number as a string, find the number of contiguous subsequences which recursively add up to 9

Given a number as a string, write a function to find the number of substrings (or contiguous subsequences) of the given string which recursively add up to 9.

For example digits of 729 recursively add to 9,

$$7 + 2 + 9 = 18$$

Recur for 18

$$1 + 8 = 9$$

Examples:

```
Input: 4189
Output: 3
There are three substrings which recursively add to 9.
The substrings are 18, 9 and 189.
```

```
Input: 999
Output: 6
There are 6 substrings which recursively add to 9.
9, 99, 999, 9, 99, 9
```

All digits of a number recursively add up to 9, if only if the number is multiple of 9. We basically need to check for $s \% 9$ for all substrings s . One trick used in below program is to do modular arithmetic to avoid overflow for big strings.

Following is a simple implementation based on this approach. The implementation assumes that there are no leading 0's in input number.

```
// C++ program to count substrings with recursive sum equal to 9
#include <iostream>
#include <cstring>
using namespace std;

int count9s(char number[])
{
    int count = 0; // To store result
    int n = strlen(number);

    // Consider every character as beginning of substring
    for (int i = 0; i < n; i++)
    {
        int sum = number[i] - '0'; //sum of digits in current substring

        if (number[i] == '9') count++;

        // One by one choose every character as an ending character
        for (int j = i+1; j < n; j++)
        {
            // Add current digit to sum, if sum becomes multiple of 5
            // then increment count. Let us do modular arithmetic to
            // avoid overflow for big strings
            sum = (sum + number[j]) - '0' % 9;

            if (sum == 0)
                count++;
        }
    }
    return count;
}

// driver program to test above function
int main()
{
    cout << count9s("4189") << endl;
    cout << count9s("1809");
    return 0;
}
```

Output:

```
3
5
```

Time complexity of the above program is $O(n^2)$. Please let me know if there is a better solution.

This article is contributed by **Abhishek**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Notes (According to Official GATE 2017 Syllabus)

GATE CS Corner

Program for Bisection Method

Given a function $f(x)$ on floating number x and two numbers ' a ' and ' b ' such that $f(a) * f(b) < 0$ and $f(x)$ is continuous in $[a, b]$. Here $f(x)$ represents algebraic or transcendental equation. Find root of function in interval $[a, b]$ (Or find a value of x such that $f(x) = 0$).

Input: A **function of x**, for example $x^3 - x^2 + 2$.
And two values: **a** = -200 and **b** = 300 such that
 $f(a) * f(b)$

What are Algebraic and Transcendental functions?

Algebraic function are the one which can be represented in the form of polynomials like $f(x) = a_1x^3 + a_2x^2 + \dots + e$ where a_1, a_2, \dots are constants and x is a variable.

Transcendental function are non algebraic functions, for example $f(x) = \sin(x) * x - 3$ or $f(x) = e^x + x^2$ or $f(x) = \ln(x) + x \dots$

What is Bisection Method?

The method is also called the interval halving method, the binary search method or the dichotomy method. This method is used to find root of an equation in a given interval that is value of ' x ' for which $f(x) = 0$.

The method is based on **The Intermediate Value Theorem** which states that if $f(x)$ is a continuous function and there are two real numbers a and b such that $f(a) * f(b) < 0$. Then $f(x)$ has at least one zero between a and b . If

Assumptions:

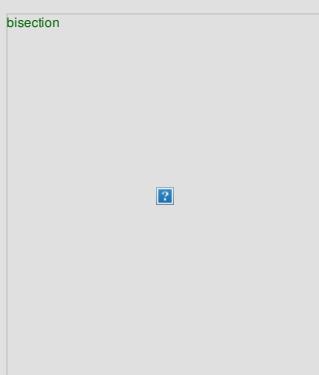
1. $f(x)$ is a continuous function in interval $[a, b]$
2. $f(a) * f(b)$

Steps:

1. Find middle point $c = (a + b)/2$.
2. If $f(c) == 0$, then c is the root of the solution.
3. Else $f(c) != 0$
 1. If value $f(a) * f(c)$
 2. Else If $f(b) * f(c)$
 3. Else given function doesn't follow one of assumptions.

Since root may be a floating point number, we repeat above steps while difference between a and b is less than a value ϵ (A very small value).

Below image is taken from [wiki page](#).



Below is C++ implementation of above steps.

```
// C++ program for implementation of Bisection Method for
// solving equations
#include<bits/stdc++.h>
using namespace std;
#define EPSILON 0.01

// An example function whose solution is determined using
// Bisection Method. The function is x^3 - x^2 + 2
double func(double x)
{
    return x*x*x - x*x + 2;
}

// Prints root of func(x) with error of EPSILON
void bisection(double a, double b)
{
    if (func(a) * func(b) >= 0)
    {
        cout << "You have not assumed right a and b\n";
        return;
    }
```

```

double c = a;
while ((b-a) >= EPSILON)
{
    // Find middle point
    c = (a+b)/2;

    // Check if middle point is root
    if (func(c) == 0.0)
        break;

    // Decide the side to repeat the steps
    else if (func(c)*func(a) < 0)
        b = c;
    else
        a = c;
}
cout << "The value of root is : " << c;
}

// Driver program to test above function
int main()
{
    // Initial values assumed
    double a = -200, b = 300;
    bisection(a, b);
    return 0;
}

```

Output:

The value of root is : -1.0025

Time complexity :- Time complexity of this method depends on the assumed values and the function.

What are pros and cons?

Advantage of the bisection method is that it is guaranteed to be converged. Disadvantage of bisection method is that it cannot detect multiple roots.

In general, Bisection method is used to get an initial rough approximation of solution. Then faster converging methods are used to find the solution.

We will soon be discussing other methods to solve algebraic and transcendental equations

References:

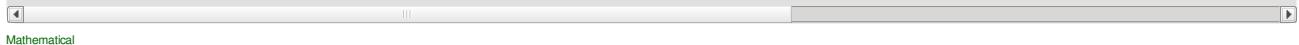
Introductory Methods of Numerical Analysis by S.S. Sastry

https://en.wikipedia.org/wiki/Bisection_method

This article is contributed by **Abhiraj Smit**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner

Company Wise Coding Practice



Program for Method Of False Position

Given a function $f(x)$ on floating number x and two numbers 'a' and 'b' such that $f(a)f(b) < 0$ and $f(x)$ is continuous in $[a, b]$. Here $f(x)$ represents algebraic or transcendental equation. Find root of function in interval $[a, b]$ (Or find a value of x such that $f(x) \approx 0$).

Input: A function of x , for example $x^3 - x^2 + 2$.

And two values: $a = -200$ and $b = 300$ such that

$f(a)f(b) < 0$, i.e., $f(a)$ and $f(b)$ have

opposite signs.

Output: The value of root is : -1.00

OR any other value close to root.

We strongly recommend to refer below post as a prerequisite of this post.

[Solution of Algebraic and Transcendental Equations | Set 1 \(The Bisection Method\)](#)

In this post The Method Of False Position is discussed. This method is also known as Regula Falsi or The Method of Chords.

Similarities with Bisection Method:

1. Same Assumptions: This method also assumes that function is continuous in $[a, b]$ and given two numbers 'a' and 'b' are such that $f(a) * f(b) < 0$
2. Always Converges: like Bisection, it always converges, usually considerably faster than Bisection—but sometimes very much more slowly than Bisection.

Differences with Bisection Method:

It differs in the fact that we make a chord joining the two points $[a, f(a)]$ and $[b, f(b)]$. We consider the point at which the chord touches the x axis and named it as c .

Steps:

1. Write equation of the line connecting the two points.

$$y - f(a) = ((f(b) - f(a)) / (b - a)) * (x - a)$$

Now we have to find the point which touches x axis.
For that we put $y = 0$.

$$\begin{aligned} \text{so } x &= a - ((f(a) / (f(b) - f(a))) * (b - a)) \\ x &= (a * f(b) - b * f(a)) / (f(b) - f(a)) \end{aligned}$$

This will be our c that is $c = x$.

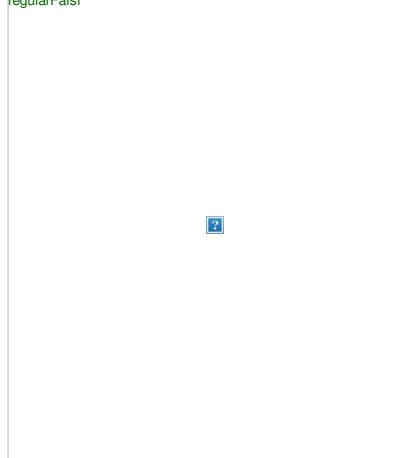
2. If $f(c) == 0$, then c is the root of the solution.

3. Else $f(c) != 0$
 1. If value $f(a)*f(c) < 0$
 2. Else If $f(b)*f(c) < 0$
 3. Else given function doesn't follow one of assumptions.

Since root may be a floating point number and may converge very slow in worst case, we iterate for a very large number of times such that the answer becomes closer to the root.

Below image is taken from [Wiki Page](#)

regularFalsi



Following is C++ implementation.

```
// C++ program for implementation of Bisection Method for
// solving equations
#include<bits/stdc++.h>
using namespace std;
#define MAX_ITER 1000000

// An example function whose solution is determined using
// Bisection Method. The function is x^3 - x^2 + 2
double func(double x)
{
    return x*x*x - x*x + 2;
}

// Prints root of func(x) in interval [a, b]
void regularFalsi(double a, double b)
{
    if (func(a) * func(b) >= 0)
    {
        cout << "You have not assumed right a and b\n";
        return;
    }

    double c = a; // Initialize result

    for (int i=0; i < MAX_ITER; i++)
    {
        // Find the point that touches x axis
        c = (a*func(b) - b*func(a)) / (func(b) - func(a));

        // Check if the above found point is root
        if (func(c)==0)
            break;

        // Decide the side to repeat the steps
        else if (func(c)*func(a) < 0)
            b = c;
        else
            a = c;
    }
    cout << "The value of root is : " << c;
}

// Driver program to test above function
int main()
{
    // Initial values assumed
    double a = -200, b = 300;
    regularFalsi(a, b);
    return 0;
}
```

}

Output:

The value of root is :-1

This method always converges, usually considerably faster than Bisection. But worst case can be very slow.

We will soon be discussing other methods to solve algebraic and transcendental equations.

References:

Introductory Methods of Numerical Analysis by S.S. Sastry

https://en.wikipedia.org/wiki/False_position_method

This article is contributed by **Abhiraj Smit**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Interview Experiences

Program for Newton Raphson Method

Given a function $f(x)$ on floating number x and an initial guess for root, find root of function in interval. Here $f(x)$ represents algebraic or transcendental equation.

For simplicity, we have assumed that derivative of function is also provided as input.

Example:

```
Input: A function of x (for example  $x^3 - x^2 + 2$ ),  
derivative function of x ( $3x^2 - 3x$  for above example)  
and an initial guess  $x_0 = -2.0$   
Output: The value of root is :-1.00  
OR any other value close to root.
```

We have discussed below methods to find root in set 1 and set 2

Set 1: The Bisection Method

Set 2: The Method Of False Position

Comparison with above two methods:

1. In previous methods, we were given an interval. Here we are required an initial guess value of root.
2. The previous two methods are guaranteed to converge, Newton Raphson may not converge in some cases.
3. Newton Raphson method requires derivative. Some functions may be difficult to impossible to differentiate.
4. For many problems, Newton Raphson method converges faster than the above two methods.
5. Also, it can identify repeated roots, since it does not look for changes in the sign of $f(x)$ explicitly

The formula:

Starting from initial guess x_1 , the Newton Raphson method uses below formula to find next value of x , i.e., x_{n+1} from previous value x_n .



Algorithm:

Input: initial x , $func(x)$, $derivFunc(x)$

Output: Root of Func()

1. Compute values of $func(x)$ and $derivFunc(x)$ for given initial x
2. Compute $h = func(x) / derivFunc(x)$
3. While h is greater than allowed error ϵ
 - a. $h = func(x) / derivFunc(x)$
 - b. $x = x - h$

Below is C++ implementation of above algorithm.

```
// C++ program for implementation of Newton Raphson Method for  
// solving equations  
#include<bits/stdc++.h>  
#define EPSILON 0.001  
using namespace std;  
  
// An example function whose solution is determined using  
// Bisection Method. The function is  $x^3 - x^2 + 2$   
double func(double x)  
{  
    return x*x*x - x*x + 2;  
}  
  
// Derivative of the above function which is  $3*x^2 - 2*x$   
double derivFunc(double x)  
{  
    return 3*x*x - 2*x;  
}  
  
// Function to find the root  
void newtonRaphson(double x)  
{  
    double h = func(x) / derivFunc(x);  
    while (abs(h) >= EPSILON)  
    {  
        h = func(x)/derivFunc(x);  
  
        //  $x(i+1) = x(i) - f(x) / f'(x)$   
        x = x - h;  
    }  
  
    cout << "The value of the root is :" << x;  
}
```

```
// Driver program to test above
int main()
{
    double x0 = -20; // Initial values assumed
    newtonRaphson(x0);
    return 0;
}
```

Output:

The value of root is :-1.00

How does this work?

The idea is to draw a line tangent to $f(x)$ at point x_1 . The point where the tangent line crosses the x axis should be a better estimate of the root than x_1 . Call this point x_2 . Calculate $f(x_2)$, and draw a line tangent at x_2 .

`newtonRaphsonMethod`



We know that slope of line from $(x_1, f(x_1))$ to $(x_2, 0)$ is $f'(x_1)$ where f' represents derivative of f .

$$f'(x_1) = (0 - f(x_1)) / (x_2 - x_1)$$

$$f'(x_1) * (x_2 - x_1) = -f(x_1)$$

$$x_2 = x_1 - f(x_1) / f'(x_1)$$

By finding this point x_2 , we move closer towards the root.

We have to keep on repeating the above step till we get really close to the root or we find it.

In general,

$$x_{n+1} = x_n - f(x_n) / f'(x_n)$$

Alternate Explanation using Taylor's Series:

Let x_1 be the initial guess.

We can write x_2 as below:

$$x_{n+1} = x_n + h \quad \dots \quad (1)$$

Here h would be a small value that can be positive or negative.

According to [Taylor's Series](#),

$f(x)$ that is infinitely differentiable can be written as below

$$f(x_{n+1}) = f(x_n + h)$$

$$= f(x_n) + h * f'(x_n) + ((h * h) / 2!) * (f''(x_n)) + \dots$$

Since we are looking for root of function, $f(x_{n+1}) = 0$

$$f(x_n) + h * f'(x_n) + ((h * h) / 2!) * (f''(x_n)) + \dots = 0$$

Now since h is small, $h * h$ would be very small.

So if we ignore higher order terms, we get

$$f(x_n) + h * f'(x_n) = 0$$

Substituting this value of $h = x_{n+1} - x_n$ from equation (1) we get,

$$f(x_n) + (x_{n+1} - x_n) * f'(x_n) = 0$$

$$x_{n+1} = x_n - f(x_n) / f'(x_n)$$

Notes:

1. We generally used this method to improve the result obtained by either bisection method or method of false position.
2. [Babylonian method for square root](#) is derived from the Newton-Raphson method.

References:

[Introductory Methods of Numerical Analysis by S.S. Sastri](#)

https://en.wikipedia.org/wiki/Newton's_method

http://www.cae.tntech.edu/Members/renfro/me2000/lectures/2004-09-07_handouts.pdf at_download/file

This article is contributed by [Abhiraj Smit](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Mathematical

Find the element that appears once

Given an array where every element occurs three times, except one element which occurs only once. Find the element that occurs once. Expected time complexity is $O(n)$ and $O(1)$ extra space.

Examples:

Input: arr[] = {12, 1, 12, 3, 12, 1, 1, 2, 3, 3}

Output: 2

We strongly recommend that you click here and practice it, before moving on to the solution.

We can use sorting to do it in $O(n\log n)$ time. We can also use hashing, but the worst case time complexity of but hashing requires extra space.

The idea is to use bitwise operators for a solution that is $O(n)$ time and uses $O(1)$ extra space. The solution is not easy like other XOR based solutions, because all elements appear odd number of times here. The idea is taken from [here](#).

Run a loop for all elements in array. At the end of every iteration, maintain following two values.

ones: The bits that have appeared 1st time or 4th time or 7th time .. etc.

twos: The bits that have appeared 2nd time or 5th time or 8th time .. etc.

Finally, we return the value of '*ones*'

How to maintain the values of 'ones' and 'twos'

'*ones*' and '*twos*' are initialized as 0. For every new element in array, find out the common set bits in the new element and previous value of '*ones*'. These common set bits are actually the bits that should be added to '*twos*'. So do bitwise OR of the common set bits with '*twos*'. '*twos*' also gets some extra bits that appear third time. These extra bits are removed later.

Update '*ones*' by doing XOR of new element with previous value of '*ones*'. There may be some bits which appear 3rd time. These extra bits are also removed later.

Both '*ones*' and '*twos*' contain those extra bits which appear 3rd time. Remove these extra bits by finding out common set bits in '*ones*' and '*twos*'.

```
#include <stdio.h>

int getSingle(int arr[], int n)
{
    int ones = 0, twos = 0;

    int common_bit_mask;

    // Let us take the example of {3, 3, 2, 3} to understand this
    for( int i=0; i< n; i++ )
    {
        /* The expression "one & arr[i]" gives the bits that are
         * there in both 'ones' and new element from arr[]. We
         * add these bits to 'twos' using bitwise OR

        Value of 'twos' will be set as 0, 3, 3 and 1 after 1st,
        2nd, 3rd and 4th iterations respectively */
        twos = twos | (ones & arr[i]);

        /* XOR the new bits with previous 'ones' to get all bits
         * appearing odd number of times

        Value of 'ones' will be set as 3, 0, 2 and 3 after 1st,
        2nd, 3rd and 4th iterations respectively */
        ones = ones ^ arr[i];

        /* The common bits are those bits which appear third time
         * So these bits should not be there in both 'ones' and 'twos'.
         * common_bit_mask contains all these bits as 0, so that the bits can
         * be removed from 'ones' and 'twos'

        Value of 'common_bit_mask' will be set as 00, 00, 01 and 10
        after 1st, 2nd, 3rd and 4th iterations respectively */
        common_bit_mask = ~(ones & twos);

        /* Remove common bits (the bits that appear third time) from 'ones'

        Value of 'ones' will be set as 3, 0, 0 and 2 after 1st,
        2nd, 3rd and 4th iterations respectively */
        ones &= common_bit_mask;

        /* Remove common bits (the bits that appear third time) from 'twos'

        Value of 'twos' will be set as 0, 3, 1 and 0 after 1st,
        2nd, 3rd and 4th iterations respectively */
        twos &= common_bit_mask;

        // uncomment this code to see intermediate values
        //printf(" %d %d \n", ones, twos);
    }

    return ones;
}

int main()
{
    int arr[] = {3, 3, 2, 3};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("The element with single occurrence is %d ", getSingle(arr, n));
    return 0;
}
```

Output:

2

Time Complexity: $O(n)$

Auxiliary Space: $O(1)$

Following is another $O(n)$ time complexity and $O(1)$ extra space method suggested by [aj](#). We can sum the bits in same positions for all the numbers and take modulo with 3. The bits for which sum is not multiple of 3, are the bits of number with single occurrence.

Let us consider the example array {5, 5, 5, 8}. The 101, 101, 101, 1000

```

Sum of first bits%3 = (1 + 1 + 1 + 0)%3 = 0;
Sum of second bits%3 = (0 + 0 + 0 + 0)%0 = 0;
Sum of third bits%3 = (1 + 1 + 1 + 0)%3 = 0;
Sum of fourth bits%3 = (1)%3 = 1;
Hence number which appears once is 1000

```

```

#include <stdio.h>
#define INT_SIZE 32

int getSingle(int arr[], int n)
{
    // Initialize result
    int result = 0;

    int x, sum;

    // Iterate through every bit
    for (int i = 0; i < INT_SIZE; i++)
    {
        // Find sum of set bits at ith position in all
        // array elements
        sum = 0;
        x = (1 << i);
        for (int j = 0; j < n; j++)
        {
            if (arr[j] & x)
                sum++;
        }

        // The bits with sum not multiple of 3, are the
        // bits of element with single occurrence.
        if (sum % 3)
            result |= x;
    }

    return result;
}

// Driver program to test above function
int main()
{
    int arr[] = {12, 1, 12, 3, 12, 1, 1, 2, 3, 2, 2, 3, 7};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("The element with single occurrence is %d ", getSingle(arr, n));
    return 0;
}

```

7

This article is compiled by **Sumit Jain** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Bit Magic
XOR

Detect if two integers have opposite signs

Given two signed integers, write a function that returns true if the signs of given integers are different, otherwise false. For example, the function should return true -1 and +100, and should return false for -100 and -200. The function should not use any of the arithmetic operators.

Let the given integers be x and y. The sign bit is 1 in negative numbers, and 0 in positive numbers. The XOR of x and y will have the sign bit as 1 iff they have opposite sign. In other words, XOR of x and y will be negative number number iff x and y have opposite signs. The following code use this logic.

```

#include<stdbool.h>
#include<stdio.h>

bool oppositeSigns(int x, int y)
{
    return ((x ^ y) < 0);
}

int main()
{
    int x = 100, y = -100;
    if (oppositeSigns(x, y) == true)
        printf ("Signs are opposite");
    else
        printf ("Signs are not opposite");
    return 0;
}

```

Output:

Signs are opposite

Source: Detect if two integers have opposite signs

We can also solve this by using two comparison operators. See the following code.

```

bool oppositeSigns(int x, int y)
{
    return (x < 0) ? (y >= 0) : (y < 0);
}

```

The first method is more efficient. The first method uses a bitwise XOR and a comparison operator. The second method uses two comparison operators and a bitwise XOR operation is more efficient compared to a comparison operation.

We can also use following method. It doesn't use any comparison operator. The method is suggested by Hongliang and improved by gaurav.

```
bool oppositeSigns(int x, int y)
{
    return ((x ^ y) >> 31);
}
```

The function is written only for compilers where size of an integer is 32 bit. The expression basically checks sign of $(x \wedge y)$ using bitwise operator ' $>>$ '. As mentioned above, the sign bit for negative numbers is always 1. The sign bit is the leftmost bit in binary representation. So we need to check whether the 32nd bit (or leftmost bit) of $x \wedge y$ is 1 or not. We do it by right shifting the value of $x \wedge y$ by 31, so that the sign bit becomes the least significant bit. If sign bit is 1, then the value of $(x \wedge y) >> 31$ will be 1, otherwise 0.

Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Bit Magic
XOR

Count total set bits in all numbers from 1 to n

Given a positive integer n, count the total number of set bits in binary representation of all numbers from 1 to n.

Examples:

```
Input: n = 3
Output: 4
```

```
Input: n = 6
Output: 9
```

```
Input: n = 7
Output: 12
```

```
Input: n = 8
Output: 13
```

We strongly recommend that you click here and practice it, before moving on to the solution.

Source: Amazon Interview Question

Method 1 (Simple)

A simple solution is to run a loop from 1 to n and sum the count of set bits in all numbers from 1 to n.

```
// A simple program to count set bits in all numbers from 1 to n.
#include <stdio.h>

// A utility function to count set bits in a number x
unsigned int countSetBitsUtil(unsigned int x);

// Returns count of set bits present in all numbers from 1 to n
unsigned int countSetBits(unsigned int n)
{
    int bitCount = 0; // initialize the result

    for(int i = 1; i <= n; i++)
        bitCount += countSetBitsUtil(i);

    return bitCount;
}

// A utility function to count set bits in a number x
unsigned int countSetBitsUtil(unsigned int x)
{
    if (x <= 0)
        return 0;
    return (x % 2 == 0 ? 0 : 1) + countSetBitsUtil(x/2);
}

// Driver program to test above functions
int main()
{
    int n = 4;
    printf ("Total set bit count is %d", countSetBits(n));
    return 0;
}
```

Output:

```
Total set bit count is 5
```

Time Complexity: O(nLogn)

Method 2 (Tricky)

If the input number is of the form $2^b - 1$ e.g., 1,3,7,15.. etc, the number of set bits is $b * 2^{(b-1)}$. This is because for all the numbers 0 to $(2^b - 1)$, if you complement and flip the list you end up with the same list (half the bits are on, half off).

If the number does not have all set bits, then some position m is the position of leftmost set bit. The number of set bits in that position is $n - (1|0|0|0|0|0|1|0|0|1|1|1|1|1|0)$

The leftmost set bit is in position 2 (positions are considered starting from 0). If we mask that off what remains is 2 (the "1 0" in the right part of the last row.) So the number of bits in the 2nd position (the lower left box) is 3 (that is, $2 + 1$). The set bits from 0-3 (the upper right box above) is $2^2 * (2-1) = 4$. The box in the lower right is the remaining bits we haven't yet counted, and is the number of set bits for all the numbers up to 2 (the value of the last entry in the lower right box) which can be figured recursively.

```
// A O(Logn) complexity program to count set bits in all numbers from 1 to n
#include <stdio.h>

/* Returns position of leftmost set bit. The rightmost
```

```

position is considered as 0 */
unsigned int getLeftmostBit (int n)
{
    int m = 0;
    while (n > 1)
    {
        n = n >> 1;
        m++;
    }
    return m;
}

/* Given the position of previous leftmost set bit in n (or an upper
bound on leftmost position) returns the new position of leftmost
set bit in n */
unsigned int getNextLeftmostBit (int n, int m)
{
    unsigned int temp = 1 << m;
    while (n < temp)
    {
        temp = temp >> 1;
        m--;
    }
    return m;
}

// The main recursive function used by countSetBits()
unsigned int _countSetBits(unsigned int n, int m);

// Returns count of set bits present in all numbers from 1 to n
unsigned int countSetBits(unsigned int n)
{
    // Get the position of leftmost set bit in n. This will be
    // used as an upper bound for next set bit function
    int m = getLeftmostBit (n);

    // Use the position
    return _countSetBits (n, m);
}

unsigned int _countSetBits(unsigned int n, int m)
{
    // Base Case: if n is 0, then set bit count is 0
    if (n == 0)
        return 0;

    /* get position of next leftmost set bit */
    m = getNextLeftmostBit(n, m);

    // If n is of the form 2^x-1, i.e., if n is like 1, 3, 7, 15, 31... etc,
    // then we are done.
    // Since positions are considered starting from 0, 1 is added to m
    if (n == ((unsigned int)1 << (m+1))-1)
        return (unsigned int)(m+1)*(1<<m);

    // update n for next recursive call
    n = n - (1<<m);
    return (n+1) + countSetBits(n) + m*(1<<(m-1));
}

// Driver program to test above functions
int main()
{
    int n = 17;
    printf ("Total set bit count is %d", countSetBits(n));
    return 0;
}

```

Total set bit count is 35

Time Complexity: O(Logn). From the first look at the implementation, time complexity looks more. But if we take a closer look, statements inside while loop of getNextLeftmostBit() are executed for all 0 bits in n. And the number of times recursion is executed is less than or equal to set bits in n. In other words, if the control goes inside while loop of getNextLeftmostBit(), then it skips those many bits in recursion.

Thanks to [agatsu](#) and [IC](#) for suggesting this solution.

See [this for another solution suggested by Piyush Kapoor](#).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Bit Magic
Amazon

Swap bits in a given number

Given a number x and two positions (from right side) in binary representation of x, write a function that swaps n bits at given two positions and returns the result. It is also given that the two sets of bits do not overlap.

We strongly recommend that you click here and practice it, before moving on to the solution.

Examples:

Let p1 and p2 be the two given positions.

Example 1
Input:
x = 47 (00101111)

```

p1 = 1 (Start from second bit from right side)
p2 = 5 (Start from 6th bit from right side)
n = 3 (No of bits to be swapped)
Output:
227 (11100011)
The 3 bits starting from the second bit (from right side) are
swapped with 3 bits starting from 6th position (from right side)

```

Example 2

```

Input:
x = 28 (11100)
p1 = 0 (Start from first bit from right side)
p2 = 3 (Start from 4th bit from right side)
n = 2 (No of bits to be swapped)
Output:
7 (00111)
The 2 bits starting from 0th position (from right side) are
swapped with 2 bits starting from 4th position (from right side)

```

Solution

We need to swap two sets of bits. XOR can be used in a similar way as it is used to swap 2 numbers. Following is the algorithm.

- 1) Move all bits of first set to rightmost side
 $\text{set1} = (x >> p1) \& ((1U (1U gives a number that contains last n bits set and other bits as 0. We do \& with this expression so that bits other than the last n bits become 0.))$
- 2) Move all bits of second set to rightmost side
 $\text{set2} = (x >> p2) \& ((1U (1U gives a number that contains last n bits set and other bits as 0. We do \& with this expression so that bits other than the last n bits become 0.))$

Implementation:

```

#include<stdio.h>

int swapBits(unsigned int x, unsigned int p1, unsigned int p2, unsigned int n)
{
    /* Move all bits of first set to rightmost side */
    unsigned int set1 = (x >> p1) & ((1U << n) - 1);

    /* Move all bits of second set to rightmost side */
    unsigned int set2 = (x >> p2) & ((1U << n) - 1);

    /* XOR the two sets */
    unsigned int xor = (set1 ^ set2);

    /* Put the xor bits back to their original positions */
    xor = (xor << p1) | (xor << p2);

    /* XOR the 'xor' with the original number so that the
       two sets are swapped */
    unsigned int result = x ^ xor;

    return result;
}

/* Driver program to test above function*/
int main()
{
    int res = swapBits(28, 0, 3, 2);
    printf("\nResult = %d ", res);
    return 0;
}

```

Output:

```
Result = 7
```

Following is a shorter implementation of the same logic

```

int swapBits(unsigned int x, unsigned int p1, unsigned int p2, unsigned int n)
{
    /* xor contains xor of two sets */
    unsigned int xor = ((x >> p1) ^ (x >> p2)) & ((1U << n) - 1);

    /* To swap two sets, we need to again XOR the xor with original sets */
    return x ^ ((xor << p1) | (xor << p2));
}

```

References:

[Swapping individual bits with XOR](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner

Company Wise Coding Practice

Bit Magic
XOR

Add two numbers without using arithmetic operators

Write a function Add() that returns sum of two integers. The function should not use any of the arithmetic operators (+, ++, -, -, .. etc).

Sum of two bits can be obtained by performing XOR (^) of the two bits. Carry bit can be obtained by performing AND (&) of two bits.

Above is simple [Half Adder](#) logic that can be used to add 2 single bits. We can extend this logic for integers. If x and y don't have set bits at same position(s), then bitwise XOR (^) of x and y gives the sum of x and y. To incorporate common set bits also, bitwise AND (&) is used. Bitwise AND of x and y gives all carry bits. We calculate $(x \& y) << 1$ and add it to $x ^ y$ to get the required result.

```
#include<stdio.h>

int Add(int x, int y)
{
    // Iterate till there is no carry
    while (y != 0)
    {
        // carry now contains common set bits of x and y
        int carry = x & y;

        // Sum of bits of x and y where at least one of the bits is not set
        x = x ^ y;

        // Carry is shifted by one so that adding it to x gives the required sum
        y = carry << 1;
    }
    return x;
}

int main()
{
    printf("%d", Add(15, 32));
    return 0;
}
```

Following is recursive implementation for the same approach.

```
int Add(int x, int y)
{
    if (y == 0)
        return x;
    else
        return Add(x ^ y, (x & y) << 1);
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Bit Magic
XOR

Smallest of three integers without comparison operators

Write a C program to find the smallest of three integers, without using any of the comparison operators.

Let 3 input numbers be x, y and z.

Method 1 (Repeated Subtraction)

Take a counter variable c and initialize it with 0. In a loop, repeatedly subtract x, y and z by 1 and increment c. The number which becomes 0 first is the smallest. After the loop terminates, c will hold the minimum of 3.

```
#include<stdio.h>

int smallest(int x, int y, int z)
{
    int c = 0;
    while (x && y && z)
    {
        x--; y--; z--; c++;
    }
    return c;
}

int main()
{
    int x = 12, y = 15, z = 5;
    printf("Minimum of 3 numbers is %d", smallest(x, y, z));
    return 0;
}
```

This method doesn't work for negative numbers. Method 2 works for negative numbers also.

Method 2 (Use Bit Operations)

Use method 2 of [this post to find minimum of two numbers](#) (We can't use Method 1 as Method 1 uses comparison operator). Once we have functionality to find minimum of 2 numbers, we can use this to find minimum of 3 numbers.

```

// See method 2 of http://www.geeksforgeeks.org/archives/2643
#include<stdio.h>
#define CHAR_BIT 8

/* Function to find minimum of x and y*/
int min(int x, int y)
{
    return y + ((x - y) & ((x - y) >>
        (sizeof(int) * CHAR_BIT - 1)));
}

/* Function to find minimum of 3 numbers x, y and z*/
int smallest(int x, int y, int z)
{
    return min(x, min(y, z));
}

int main()
{
    int x = 12, y = 15, z = 5;
    printf("Minimum of 3 numbers is %d", smallest(x, y, z));
    return 0;
}

```

Method 3 (Use Division operator)

We can also use division operator to find minimum of two numbers. If value of (a/b) is zero, then b is greater than a , else a is greater. Thanks to [gopinath](#) and [Vignesh](#) for suggesting this method.

```

#include <stdio.h>

// Using division operator to find minimum of three numbers
int smallest(int x, int y, int z)
{
    if (! (y/x)) // Same as "if (y < x)"
        return (! (y/z)) ? y : z;
    return (! (x/z)) ? x : z;
}

int main()
{
    int x = 78, y = 88, z = 68;
    printf("Minimum of 3 numbers is %d", smallest(x, y, z));
    return 0;
}

```

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Bit Magic

A Boolean Array Puzzle

Input: A array arr[] of two elements having value 0 and 1

Output: Make both elements 0.

Specifications: Following are the specifications to follow.

- 1) It is guaranteed that one element is 0 but we do not know its position.
- 2) We can't say about another element it can be 0 or 1.
- 3) We can only complement array elements, no other operation like and, or, multi, division, etc.
- 4) We can't use if, else and loop constructs.
- 5) Obviously, we can't directly assign 0 to array elements.

There are several ways we can do it as we are sure that always one Zero is there. Thanks to [devendrallit](#) for suggesting following 3 methods.

Method 1

```

void changeToZero(int a[2])
{
    a[ a[1] ] = a[ !a[1] ];
}

int main()
{
    int a[] = {1, 0};
    changeToZero(a);

    printf(" arr[0] = %d \n", a[0]);
    printf(" arr[1] = %d ", a[1]);
    getchar();
    return 0;
}

```

Method 2

```

void changeToZero(int a[2])
{
    a[ !a[0] ] = a[ !a[1] ];
}

```

Method 3

This method doesn't even need complement.

```

void changeToZero(int a[2])
{
    a[ a[1] ] = a[ a[0] ];
}

```

Method 4

Thanks to [purvi](#) for suggesting this method.

```
void changeToZero(int a[2])
{
    a[0] = a[a[0]];
    a[1] = a[0];
}
```

There may be many more methods.

Source: <http://geeksforgeeks.org/forum/topic/google-challenge>

Please write comments if you find the above codes incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Bit Magic

Program to count number of set bits in an (big) array

Given an integer array of length N (an arbitrarily large number). How to count number of set bits in the array?

The simple approach would be, create an efficient method to count set bits in a word (most prominent size, usually equal to bit length of processor), and add bits from individual elements of array.

Various methods of counting set bits of an integer exists, see [this](#) for example. These methods run at best $O(\log N)$ where N is number of bits. Note that on a processor N is fixed, count can be done in $O(1)$ time on 32 bit machine irrespective of total set bits. Overall, the bits in array can be computed in $O(n)$ time, where 'n' is array size.

However, a table look up will be more efficient method when array size is large. Storing table look up that can handle 2^{32} integers will be impractical.

The following code illustrates simple program to count set bits in a randomly generated 64 K integer array. The idea is to generate a look up for first 256 numbers (one byte), and break every element of array at byte boundary. A meta program using C/C++ preprocessor generates the look up table for counting set bits in a byte.

The mathematical derivation behind meta program is evident from the following table (Add the column and row indices to get the number, then look into the table to get set bits in that number. For example, to get set bits in 10, it can be extracted from row named as 8 and column named as 2),

0, 1, 2, 3
0 - 0, 1, 2, 3 ----- GROUP_A(0)
4 - 1, 2, 2, 3 ----- GROUP_A(1)
8 - 1, 2, 2, 3 ----- GROUP_A(1)
12 - 2, 3, 3, 4 ----- GROUP_A(2)
16 - 1, 2, 2, 3 ----- GROUP_A(1)
20 - 2, 3, 3, 4 ----- GROUP_A(2)
24 - 2, 3, 3, 4 ----- GROUP_A(2)
28 - 3, 4, 4, 5 ----- GROUP_A(3) ... so on

From the table, there is a pattern emerging in multiples of 4, both in the table as well as in the group parameter. The sequence can be generalized as shown in the code.

Complexity:

All the operations takes $O(1)$ except iterating over the array. The time complexity is $O(n)$ where 'n' is size of array. Space complexity depends on the meta program that generates look up.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* Size of array 64 K */
#define SIZE (1 << 16)

/* Meta program that generates set bit count
array of first 256 integers */

#define GROUP_A(x) x, x + 1, x + 1, x + 2

/* GROUP_B - When combined with META_LOOK_UP
generates count for 4x4 elements */

#define GROUP_B(x) GROUP_A(x), GROUP_A(x+1), GROUP_A(x+1), GROUP_A(x+2)

/* GROUP_C - When combined with META_LOOK_UP
generates count for 4x4x4 elements */

#define GROUP_C(x) GROUP_B(x), GROUP_B(x+1), GROUP_B(x+1), GROUP_B(x+2)

/* Provide appropriate letter to generate the table */

#define META_LOOK_UP(PARAMETER) \
GROUP_##PARAMETER(0), \
GROUP##PARAMETER(1), \
GROUP##PARAMETER(1), \
GROUP##PARAMETER(2) \

int countSetBits(int array[], size_t array_size)
{
    int count = 0;

    /* META_LOOK_UP(C) - generates a table of 256 integers whose
    sequence will be number of bits in i-th position
    where 0 <= i < 256
    */

    /* A static table will be much faster to access */
    static unsigned char const look_up[] = { META_LOOK_UP(C) };

    /* No shifting funda (for better readability) */
    unsigned char *pData = NULL;

    for(size_t index = 0; index < array_size; index++)
    {
        /* It is fine, bypass the type system */
    }
}
```

```

pData = (unsigned char *)&array[index];

/* Count set bits in individual bytes */
count += look_up[pData[0]];
count += look_up[pData[1]];
count += look_up[pData[2]];
count += look_up[pData[3]];
}

return count;
}

/* Driver program, generates table of random 64 K numbers */
int main()
{
int index;
int random[SIZE];

/* Seed to the random-number generator */
rand((unsigned)time(0));

/* Generate random numbers.*/
for( index = 0; index < SIZE; index++)
{
random[index] = rand();
}

printf("Total number of bits = %d\n", countSetBits(random, SIZE));
return 0;
}

```

Contributed by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Bit Magic
Divide and Conquer
Divide and Conquer
About Venki
Software Engineer
View all posts by Venki →

Next higher number with same number of set bits

Given a number x , find next number with same number of 1 bits in its binary representation.

For example, consider $x = 12$, whose binary representation is 1100 (excluding leading zeros on 32 bit machine). It contains two logic 1 bits. The next higher number with two logic 1 bits is 17 (10001₂).

Algorithm:

When we observe the binary sequence from 0 to $2^n - 1$ (n is # of bits), right most bits (least significant) vary rapidly than left most bits. The idea is to find right most string of 1's in x , and shift the pattern to right extreme, except the left most bit in the pattern. Shift the left most bit in the pattern (omitted bit) to left part of x by one position. An example makes it more clear,

```

x = 156
10
x = 10011100
(2)
10011100
0000011 - right most string of 1's in x
0000001 - right shifted pattern except left most bit ----> [A]
00010000 - isolated left most bit of right most 1's pattern
00100000 - shift left-ed the isolated bit by one position -----> [B]
10000000 - left part of x, excluding right most 1's pattern -----> [C]
10100000 - add B and C (OR operation) -----> [D]
10100011 - add A and D which is required number 163

```

(10)

After practicing with few examples, it's easy to understand. Use the below given program for generating more sets.

Program Design:

We need to note few facts of binary numbers. The expression $x \& -x$ will isolate right most set bit in x (ensuring x will use 2's complement form for negative numbers). If we add the result to x , right most string of 1's in x will be reset, and the immediate '0' left to this pattern of 1's will be set, which is part [B] of above explanation. For example if $x = 156$, $x \& -x$ will result in 00000100, adding this result to x yields 10100000 (see part D). We left with the right shifting part of pattern of 1's (part A of above explanation).

There are different ways to achieve part A. Right shifting is essentially a division operation. What should be our divisor? Clearly, it should be multiple of 2 (avoids 0.5 error in right shifting), and it should shift the right most 1's pattern to right extreme. The expression $(x \& -x)$ will serve the purpose of divisor. An EX-OR operation between the number X and expression which is used to reset right most bits, will isolate the rightmost 1's pattern.

A Correction Factor:

Note that we are adding right most set bit to the bit pattern. The addition operation causes a shift in the bit positions. The weight of binary system is 2, one shift causes an increase by a factor of 2. Since the increased number (*rightOnesPattern* in the code) being used twice, the error propagates twice. The error needs to be corrected. A right shift by 2 positions will correct the result.

The popular name for this program is **same number of one bits**.

```

#include<iostream>

using namespace std;

typedef unsigned int uint_t;

// this function returns next higher number with same number of set bits as x.
uint_t snoob(uint_t x)
{
    uint_t rightOne;
    uint_t nextHigherOneBit;
    uint_t rightOnesPattern;

```

```

uint_t next = 0;

if(x)
{
    // right most set bit
    rightOne = x & -(signed)x;

    // reset the pattern and set next higher bit
    // left part of x will be here
    nextHigherOneBit = x + rightOne;

    // nextHigherOneBit is now part [D] of the above explanation.

    // isolate the pattern
    rightOnesPattern = x ^ nextHigherOneBit;

    // right adjust pattern
    rightOnesPattern = (rightOnesPattern)/rightOne;

    // correction factor
    rightOnesPattern >>= 2;

    // rightOnesPattern is now part [A] of the above explanation.

    // integrate new pattern (Add [D] and [A])
    next = nextHigherOneBit | rightOnesPattern;
}

return next;
}

int main()
{
    int x = 156;
    cout<<"Next higher number with same number of set bits is "<<snoob(x);

    getchar();
    return 0;
}

```

Usage: Finding/Generating subsets.

Variations:

1. Write a program to find a number immediately smaller than given, with same number of logic 1 bits? (Pretty simple)
2. How to count or generate the subsets available in the given set?

References:

1. A nice presentation [here](#).
2. [Hackers Delight](#) by Warren (An excellent and short book on various bit magic algorithms, a must for enthusiasts)
3. C A Reference Manual by Harbison and Steele (A good book on standard C, you can access code part of this post [here](#)).

– **Venki.** Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

[Bit Magic](#)
[About Venki](#)
Software Engineer
View all posts by Venki →

Optimization Techniques | Set 1 (Modulus)

Modulus operator is costly.

The modulus operator (%) in various languages is costly operation. Ultimately every operator/operation must result in processor instructions. Some processors won't have modulus instruction at hardware level, in such case the compilers will insert stubs (predefined functions) to perform modulus. It impacts performance.

There is simple technique to extract remainder when a number is divided by another number (divisor) that is power of 2? A number that is an exact power of 2 will have only one bit set in its binary representation. Consider the following powers of 2 and their binary representations

2 – 10

4 – 100

8 – 1000

16 – 10000

Note those zeros in red color, they contribute to remainder in division operation. We can get mask for those zeros by decrementing the divisor by 1.

Generalizing the above pattern, a number that can be written in 2^n form will have only one bit set followed by n zeros on the right side of 1. When a number (N) divided by (2^n) , the bit positions corresponding to the above mentioned zeros will contribute to the remainder of division operation. An example can make it clear,

N = 87 (1010111 – binary form)

$$N \% 2 = N \& (2-1) = 1010111 \& 1 = 1 = 1$$

$$N \% 4 = N \& (4-1) = 1010111 \& 11 = 11 = 3$$

$$N \% 8 = N \& (8-1) = 1010111 \& 111 = 111 = 7$$

$$N \% 16 = N \& (16-1) = 1010111 \& 1111 = 111 = 7$$

$$N \% 32 = N \& (32-1) = 1010111 \& 11111 = 10111 = 23$$

Modulus operation over exact powers of 2 is simple and faster bitwise ANDing. This is the reason, programmers usually make buffer length as powers of 2.

Note that the technique will work only for divisors that are powers of 2.

An Example:

Implementation of circular queue (ring buffer) using an array. Omitting one position in the circular buffer implementation can make it easy to distinguish between *full* and *empty* conditions. When the buffer reaches SIZE-1, it needs to wrap back to initial position. The wrap back operation can be simple AND operation if the buffer size is power of 2. If we use any other size, we would need to use modulus operation.

Note:

Per experts comments, premature optimization is an evil. The optimization techniques provided are to fine tune your code after finalizing design strategy, algorithm, data structures and implementation. We recommend to avoid them at the start of code development. Code readability is key for maintenance.

Thanks to Venki for writing the above article. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Bit Magic
About Venki
Software Engineer
[View all posts by Venki →](#)

Add 1 to a given number

Write a program to add one to a given number. You are not allowed to use operators like '+', '-', '*', '/', '++', '--' ...etc.

Examples:

Input: 12

Output: 13

Input: 6

Output: 7

Yes, you guessed it right, we can use bitwise operators to achieve this. Following are different methods to achieve same using bitwise operators.

Method 1

To add 1 to a number x (say 0011000111), we need to flip all the bits after the rightmost 0 bit (we get 0011000000). Finally, flip the rightmost 0 bit also (we get 0011001000) and we are done.

```
#include<stdio.h>

int addOne(int x)
{
    int m = 1;

    /* Flip all the set bits until we find a 0 */
    while( x & m )
    {
        x = x ^ m;
        m <= 1;
    }

    /* flip the rightmost 0 bit */
    x = x ^ m;
    return x;
}

/* Driver program to test above functions*/
int main()
{
    printf("%d", addOne(13));
    getchar();
    return 0;
}
```

Method 2

We know that the negative number is represented in 2's complement form on most of the architectures. We have the following lemma hold for 2's complement representation of signed numbers.

Say, x is numerical value of a number, then

$\sim x = -(x+1)$ [\sim is for bitwise complement]

$(x + 1)$ is due to addition of 1 in 2's complement conversion

To get $(x + 1)$ apply negation once again. So, the final expression becomes $(\sim(\sim x))$.

```
int addOne(int x)
{
    return (\sim(\sim x));
}

/* Driver program to test above functions*/
int main()
{
    printf("%d", addOne(13));
    getchar();
    return 0;
}
```

Example, assume the machine word length is one *nibble* for simplicity.

And $x = 2$ (0010),

$\sim x = \sim 2 = 1101$ (13 numerical)

$\sim(\sim x) = \sim 1101 = 0010$

Interpreting bits 1101 in 2's complement form yields numerical value as $-(2^4 - 13) = -3$. Applying ' \sim ' on the result leaves 3. Same analogy holds for decrement. See [this](#) comment for implementation of decrement.

Note that this method works only if the numbers are stored in 2's complement form.

Thanks to [Venki](#) for suggesting this method.

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem

GATE CS Corner Company Wise Coding Practice

Bit Magic

Multiply a given Integer with 3.5

Given a integer x , write a function that multiplies x with 3.5 and returns the integer result. You are not allowed to use `%`, `/`, `*`.

Examples:

Input: 2

Output: 7

Input: 5

Output: 17 (Ignore the digits after decimal point)

Solution:

1. We can get $x \times 3.5$ by adding $2 \times x$, x and $x/2$. To calculate $2 \times x$, left shift x by 1 and to calculate $x/2$, right shift x by 2.

```
#include <stdio.h>

int multiplyWith3Point5(int x)
{
    return (x<<1) + x + (x>>1);
}

/* Driver program to test above functions*/
int main()
{
    int x = 4;
    printf("%d", multiplyWith3Point5(x));
    getchar();
    return 0;
}
```

2. Another way of doing this could be $(8 \times x - x)/2$ (See below code). Thanks to [ajaym](#) for suggesting this.

```
#include <stdio.h>
int multiplyWith3Point5(int x)
{
    return ((x<<3) - x)>>1;
}
```

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem

GATE CS Corner Company Wise Coding Practice

Bit Magic

Turn off the rightmost set bit

Write a C function that unsets the rightmost set bit of an integer.

Examples:

```
Input: 12 (00...01100)
Output: 8 (00...01000)

Input: 7 (00...00111)
Output: 6 (00...00110)
```

Let the input number be n . $n-1$ would have all the bits flipped after the rightmost set bit (including the set bit). So, doing $n \& (n-1)$ would give us the required result.

```
#include<stdio.h>

/* unsets the rightmost set bit of n and returns the result */
int fun(unsigned int n)
{
    return n&(n-1);
}

/* Driver program to test above function */
int main()
{
    int n = 7;
    printf("The number after unsetting the rightmost set bit %d", fun(n));

    getchar();
    return 0;
}
```

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem

GATE CS Corner Company Wise Coding Practice

Bit Magic

Find whether a given number is a power of 4 or not

We strongly recommend that you click here and practice it, before moving on to the solution.

1. A simple method is to take log of the given number on base 4, and if we get an integer then number is power of 4.

2. Another solution is to keep dividing the number by 4, i.e., do $n = n/4$ iteratively. In any iteration, if $n \% 4$ becomes non-zero and n is not 1 then n is not a power of 4, otherwise n is a power of 4.

```
#include<stdio.h>
#define bool int

/* Function to check if x is power of 4*/
bool isPowerOfFour(int n)
{
```

```

if(n == 0)
    return 0;
while(n != 1)
{
    if(n%4 != 0)
        return 0;
    n = n/4;
}
return 1;
}

/*Driver program to test above function*/
int main()
{
    int test_no = 64;
    if(isPowerOfFour(test_no))
        printf("%d is a power of 4", test_no);
    else
        printf("%d is not a power of 4", test_no);
    getchar();
}

```

3. A number n is a power of 4 if following conditions are met.

- a) There is only one bit set in the binary representation of n (or n is a power of 2)
- b) The count of zero bits before the (only) set bit is even.

For example: 16 (10000) is power of 4 because there is only one bit set and count of 0s before the set bit is 4 which is even.

Thanks to [Geek4u](#) for suggesting the approach and providing the code.

```

#include<stdio.h>
#define bool int

bool isPowerOfFour(unsigned int n)
{
    int count = 0;

    /*Check if there is only one bit set in n*/
    if (n && !(n&(n-1)))
    {
        /* count 0 bits before set bit */
        while(n > 1)
        {
            n >>= 1;
            count += 1;
        }

        /*If count is even then return true else false*/
        return (count%2 == 0)? 1 : 0;
    }

    /* If there are more than 1 bit set
     then n is not a power of 4*/
    return 0;
}

/*Driver program to test above function*/
int main()
{
    int test_no = 64;
    if(isPowerOfFour(test_no))
        printf("%d is a power of 4", test_no);
    else
        printf("%d is not a power of 4", test_no);
    getchar();
}

```

Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Bit Magic

Compute the integer absolute value (abs) without branching

We need not to do anything if a number is positive. We want to change only negative numbers. Since negative numbers are stored in [2's complement](#) form, to get the absolute value of a negative number we have to toggle bits of the number and add 1 to the result.

For example -2 in a 8 bit system is stored as follows 1 1 1 1 1 1 1 0 where leftmost bit is the sign bit. To get the absolute value of a negative number, we have to toggle all bits and add 1 to the toggled number i.e, 0 0 0 0 0 0 1 + 1 will give the absolute value of 1 1 1 1 1 1 1 0. Also remember, we need to do these operations only if the number is negative (sign bit is set).

Method 1

- 1) Set the mask as right shift of integer by 31 (assuming integers are stored using 32 bits).

```
mask = n>>31
```

- 2) For negative numbers, above step sets mask as 1 1 1 1 1 1 1 1 and 0 0 0 0 0 0 0 0 for positive numbers. Add the mask to the given number.

```
mask + n
```

- 3) XOR of mask +n and mask gives the absolute value.

```
(mask + n)^mask
```

Implementation:

```

#include <stdio.h>
#define CHAR_BIT 8

/* This function will return absolute value of n*/
unsigned int getAbs(int n)

```

```

{
    int const mask = n >> (sizeof(int) * CHAR_BIT - 1);
    return ((n + mask) ^ mask);
}

/* Driver program to test above function */
int main()
{
    int n = -6;
    printf("Absolute value of %d is %u", n, getAbs(n));

    getchar();
    return 0;
}

```

Method 2:

- 1) Set the mask as right shift of integer by 31 (assuming integers are stored using 32 bits).

```
mask = n>>31
```

- 2) XOR the mask with number

```
mask ^ n
```

- 3) Subtract mask from result of step 2 and return the result.

```
(mask^n) - mask
```

Implementation:

```

/* This function will return absolute value of n*/
unsigned int getAbs(int n)
{
    int const mask = n >> (sizeof(int) * CHAR_BIT - 1);
    return ((n ^ mask) - mask);
}

```

On machines where branching is expensive, the above expression can be faster than the obvious approach, $r = v$ (this for more details about the above two methods).

Please write comments if you find any of the above explanations/algorithms incorrect, or a better ways to solve the same problem.

References:

<http://graphics.stanford.edu/~seander/bithacks.html#IntegerAbs>

GATE CS Corner Company Wise Coding Practice

Bit Magic
Bit Magic

Compute modulus division by a power-of-2-number

Compute n modulo d without division(/) and modulo(%) operators, where d is a power of 2 number.

Let i th bit from right is set in d . For getting n modulus d , we just need to return 0 to $i-1$ (from right) bits of n as they are and other bits as 0.

For example if $n = 6$ (00..110) and $d = 4$ (00..100). Last set bit in d is at position 3 (from right side). So we need to return last two bits of n as they are and other bits as 0, i.e., 00..010.

Now doing it is so easy, guess it....

Yes, you have guessing it right. See the below program.

```

#include<stdio.h>

/* This function will return n % d.
   d must be one of: 1, 2, 4, 8, 16, 32, ... */
unsigned int getModulo(unsigned int n, unsigned int d)
{
    return (n & (d-1));
}

/* Driver program to test above function */
int main()
{
    unsigned int n = 6;
    unsigned int d = 4; /*d must be a power of 2*/
    printf("%u modulo %u is %u", n, d, getModulo(n, d));

    getchar();
    return 0;
}

```

References:

<http://graphics.stanford.edu/~seander/bithacks.html#ModulusDivisionEasy>

Please write comments if you find any bug in the above program/algorithim or other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Bit Magic
Bit Magic

Compute the minimum or maximum of two integers without branching

On some rare machines where branching is expensive, the below obvious approach to find minimum can be slow as it uses branching.

```

/* The obvious approach to find minimum (involves branching) */
int min(int x, int y)
{

```

```

    return (x < y) ? x : y
}

```

Below are the methods to get minimum(or maximum) without using branching. Typically, the obvious approach is best, though.

Method 1(Use XOR and comparison operator)

Minimum of x and y will be

$y \wedge ((x \wedge y) \& \neg(x < y))$

It works because if $x = y$, then $\neg(x \wedge ((x \wedge y) \& \neg(x < y)))$

```

#include<stdio.h>

/*Function to find minimum of x and y*/
int min(int x, int y)
{
    return y \wedge ((x \wedge y) \& \neg(x < y));
}

/*Function to find maximum of x and y*/
int max(int x, int y)
{
    return x \wedge ((x \wedge y) \& \neg(x < y));
}

/* Driver program to test above functions */
int main()
{
    int x = 15;
    int y = 6;
    printf("Minimum of %d and %d is ", x, y);
    printf("%d", min(x, y));
    printf("\nMaximum of %d and %d is ", x, y);
    printf("%d", max(x, y));
    getchar();
}

```

Method 2(Use subtraction and shift)

If we know that

`INT_MIN`

, then we can use the following, which are faster because $(x - y)$ only needs to be evaluated once.

Minimum of x and y will be

$y + ((x - y) \& ((x - y) \gg (\text{sizeof}(int) * \text{CHAR_BIT} - 1)))$

This method shifts the subtraction of x and y by 31 (if size of integer is 32). If $(x - y)$ is smaller than 0, then $(x - y) \gg 31$ will be 1. If $(x - y)$ is greater than or equal to 0, then $(x - y) \gg 31$ will be 0.

So if $x \geq y$, we get minimum as $y + (x - y) \& 0$ which is y.

If $x < y$, we get minimum as $y + ((x - y) \gg (\text{sizeof}(int) * \text{CHAR_BIT} - 1)))$

```

#include<stdio.h>
#define CHAR_BIT 8

/*Function to find minimum of x and y*/
int min(int x, int y)
{
    return y + ((x - y) \& ((x - y) \gg
        (sizeof(int) * CHAR_BIT - 1)));
}

/*Function to find maximum of x and y*/
int max(int x, int y)
{
    return x - ((x - y) \& ((x - y) \gg
        (sizeof(int) * CHAR_BIT - 1)));
}

/* Driver program to test above functions */
int main()
{
    int x = 15;
    int y = 6;
    printf("Minimum of %d and %d is ", x, y);
    printf("%d", min(x, y));
    printf("\nMaximum of %d and %d is ", x, y);
    printf("%d", max(x, y));
    getchar();
}

```

Note that the 1989 ANSI C specification doesn't specify the result of signed right-shift, so above method is not portable. If exceptions are thrown on overflows, then the values of x and y should be unsigned or cast to unsigned.

Source:

<http://graphics.stanford.edu/~seander/bithacks.html#IntegerMinOrMax>

GATE CS Corner

Company Wise Coding Practice

Bit Magic
Bit Magic
XOR

Rotate bits of a number

Bit Rotation: A rotation (or circular shift) is an operation similar to shift except that the bits that fall off at one end are put back to the other end.

In left rotation, the bits that fall off at left end are put back at right end.

In right rotation, the bits that fall off at right end are put back at left end.

Example:

Let n is stored using 8 bits. Left rotation of n = 11100101 by 3 makes n = 00101111 (Left shifted by 3 and first 3 bits are put back in last). If n is stored using 16 bits or 32 bits then left rotation of n (000...11100101) becomes 00..0011100101000.

Right rotation of n = 11100101 by 3 makes n = 10111100 (Right shifted by 3 and last 3 bits are put back in first) if n is stored using 8 bits. If n is stored using 16 bits or 32 bits then right rotation of n (000...11100101) by 3 becomes 101000..0011100.

```
#include<stdio.h>
#define INT_BITS 32

/*Function to left rotate n by d bits*/
int leftRotate(int n, unsigned int d)
{
    /* In n<d, last d bits are 0. To put first 3 bits of n at
       last, do bitwise or of n<d with n >>(INT_BITS - d) */
    return (n << d)|(n >>(INT_BITS - d));
}

/*Function to right rotate n by d bits*/
int rightRotate(int n, unsigned int d)
{
    /* In n>d, first d bits are 0. To put last 3 bits of n at
       first, do bitwise or of n>d with n <<(INT_BITS - d) */
    return (n >> d)|(n <<(INT_BITS - d));
}

/* Driver program to test above functions */
int main()
{
    int n = 16;
    int d = 2;
    printf("Left Rotation of %d by %d is ", n, d);
    printf("%d", leftRotate(n, d));
    printf("\nRight Rotation of %d by %d is ", n, d);
    printf("%d", rightRotate(n, d));
    getchar();
}
```

Please write comments if you find any bug in the above program or other ways to solve the same problem.

GATE CS Corner Company Wise Coding Practice

Bit Magic
Bit Magic
rotation

Find the two non-repeating elements in an array of repeating elements

Asked by SG

Given an array in which all numbers except two are repeated once. (i.e. we have $2n+2$ numbers and n numbers are occurring twice and remaining two have occurred once). Find those two numbers in the most efficient way.

Method 1(Use Sorting)

First sort all the elements. In the sorted array, by comparing adjacent elements we can easily get the non-repeating elements. Time complexity of this method is $O(n\log n)$

Method 2(Use XOR)

Let x and y be the non-repeating elements we are looking for and $\text{arr}[]$ be the input array. First calculate the XOR of all the array elements.

```
xor = arr[0]^arr[1]^arr[2]....arr[n-1]
```

All the bits that are set in xor will be set in one non-repeating element (x or y) and not in other. So if we take any set bit of xor and divide the elements of the array in two sets – one set of elements with same bit set and other set with same bit not set. By doing so, we will get x in one set and y in another set. Now if we do XOR of all the elements in first set, we will get first non-repeating element, and by doing same in other set we will get the second non-repeating element.

Let us see an example.
 $\text{arr}[] = \{2, 4, 7, 9, 2, 4\}$
1) Get the XOR of all the elements.
 $xor = 2^4 \wedge 7 \wedge 9 \wedge 2 \wedge 4 = 14 (1110)$
2) Get a number which has only one set bit of the xor.
Since we can easily get the rightmost set bit, let us use it.
 $\text{set_bit_no} = xor \& \sim(xor-1) = (1110) \& \sim(1101) = 0010$

Now set_bit_no will have only set as rightmost set bit of xor.
 3) Now divide the elements in two sets and do xor of elements in each set, and we get the non-repeating elements 7 and 9. Please see implementation for this step.

Implementation:

```
#include <stdio.h>
#include <stdlib.h>

/* This function sets the values of *x and *y to non-repeating elements in an array arr[] of size n*/
void get2NonRepeatingNos(int arr[], int n, int *x, int *y)
{
    int xor = arr[0]; /* Will hold xor of all elements */
    int set_bit_no; /* Will have only single set bit of xor */
    int i;
    *x = 0;
    *y = 0;

    /* Get the xor of all elements */
    for(i = 1; i < n; i++)
        xor ^= arr[i];

    /* Get the rightmost set bit in set_bit_no */
    set_bit_no = xor & ~(xor-1);

    /* Now divide elements in two sets by comparing rightmost set bit of xor with bit at same position in each element. */
    for(i = 0; i < n; i++)
    {
        if(arr[i] & set_bit_no)
            *x = *x ^ arr[i]; /*XOR of first set*/
        else
            *y = *y ^ arr[i]; /*XOR of second set*/
    }

    /* Driver program to test above function */
    int main()
    {
        int arr[] = {2, 3, 7, 9, 11, 2, 3, 11};
        int *x = (int *)malloc(sizeof(int));
        int *y = (int *)malloc(sizeof(int));
        get2NonRepeatingNos(arr, 8, x, y);
        printf("The non-repeating elements are %d and %d", *x, *y);
        getchar();
    }
}
```

Time Complexity: O(n)

Auxiliary Space: O(1)

GATE CS Corner Company Wise Coding Practice

Bit Magic
Bit Magic
XOR

Find the Number Occurring Odd Number of Times

Given an array of positive integers. All numbers occur even number of times except one number which occurs odd number of times. Find the number in O(n) time & constant space.

Example:

I/P = [1, 2, 3, 2, 3, 1, 3]
 O/P = 3

A **Simple Solution** is to run two nested loops. The outer loop picks all elements one by one and inner loop counts number of occurrences of the element picked by outer loop. Time complexity of this solution is O(n²).

A **Better Solution** is to use Hashing. Use array elements as key and their counts as value. Create an empty hash table. One by one traverse the given array elements and store counts. Time complexity of this solution is O(n). But it requires extra space for hashing.

The **Best Solution** is to do bitwise XOR of all the elements. XOR of all elements gives us odd occurring element. Please note that XOR of two elements is 0 if both elements are same and XOR of a number x with 0 is x.

Below are implementations of this best approach.

Program:

C/C++

```
/*C program to find the element occurring odd number of times

#include <stdio.h>
int getOddOccurrence(int ar[], int ar_size)
{
    int i;
    int res = 0;
    for (i=0; i < ar_size; i++)
        res = res ^ ar[i];

    return res;
}

/* Driver function to test above function */
int main()
{
    int ar[] = {2, 3, 5, 4, 5, 2, 4, 3, 5, 2, 4, 4, 2};
    int n = sizeof(ar)/sizeof(ar[0]);
    printf("%d", getOddOccurrence(ar, n));
}
```

```
    return 0;
}
```

Java

```
//Java program to find the element occurring odd number of times

class OddOccurrence
{
    int getOddOccurrence(int ar[], int ar_size)
    {
        int i;
        int res = 0;
        for (i = 0; i < ar_size; i++)
        {
            res = res ^ ar[i];
        }
        return res;
    }

    public static void main(String[] args)
    {
        OddOccurrence occur = new OddOccurrence();
        int ar[] = new int[]{2, 3, 5, 4, 5, 2, 4, 3, 5, 2, 4, 4, 2};
        int n = ar.length;
        System.out.println(occur.getOddOccurrence(ar, n));
    }
}
// This code has been contributed by Mayank Jaiswal
```

Python

```
# Python program to find the element occurring odd number of times

def getOddOccurrence(arr):
    # Initialize result
    res = 0

    # Traverse the array
    for element in arr:
        # XOR with the result
        res = res ^ element

    return res

# Test array
arr = [2, 3, 5, 4, 5, 2, 4, 3, 5, 2, 4, 4, 2]

print "%d" % getOddOccurrence(arr)
```

Output:

```
5
```

Time Complexity: O(n)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Arrays
Bit Magic
Bit Magic
XOR

Check for Integer Overflow

Write a "C" function, int addOvf(int* result, int a, int b) If there is no overflow, the function places the resultant = sum a+b in "result" and returns 0. Otherwise it returns -1. The solution of casting to long and adding to find detecting the overflow is not allowed.

Method 1

There can be overflow only if signs of two numbers are same, and sign of sum is opposite to the signs of numbers.

- 1) Calculate sum

```

2) If both numbers are positive and sum is negative then return -1
Else
If both numbers are negative and sum is positive then return -1
Else return 0

```

```

#include<stdio.h>
#include<stdlib.h>

/* Takes pointer to result and two numbers as
arguments. If there is no overflow, the function
places the resultant = sum a+b in "result" and
returns 0, otherwise it returns -1 */
int addOvf(int* result, int a, int b)
{
    *result = a + b;
    if(a > 0 && b > 0 && *result < 0)
        return -1;
    if(a < 0 && b < 0 && *result > 0)
        return -1;
    return 0;
}

int main()
{
    int *res = (int *)malloc(sizeof(int));
    int x = 2147483640;
    int y = 10;

    printf("%d", addOvf(res, x, y));

    printf("\n%d", *res);
    getchar();
    return 0;
}

```

Time Complexity : O(1)
Space Complexity: O(1)

Method 2

Thanks to Himanshu Aggarwal for adding this method. This method doesn't modify *result if there is an overflow.

```

#include<stdio.h>
#include<limits.h>
#include<stdlib.h>

int addOvf(int* result, int a, int b)
{
    if( a > INT_MAX - b)
        return -1;
    else
    {
        *result = a + b;
        return 0;
    }
}

int main()
{
    int *res = (int *)malloc(sizeof(int));
    int x = 2147483640;
    int y = 10;

    printf("%d", addOvf(res, x, y));
    printf("\n%d", *res);
    getchar();
    return 0;
}

```

Time Complexity : O(1)
Space Complexity: O(1)

Please write comments if you find any bug in above codes/algorithms, or find other ways to solve the same problem

GATE CS Corner Company Wise Coding Practice

Bit Magic
Bit Magic

Little and Big Endian Mystery

What are these?

Little and big endian are two ways of storing multibyte data-types (int, float, etc). In little endian machines, last byte of binary representation of the multibyte data-type is stored first. On the other hand, in big endian machines, first byte of binary representation of the multibyte data-type is stored first.

Suppose integer is stored as 4 bytes (For those who are using DOS based compilers such as C++ 3.0 , integer is 2 bytes) then a variable x with value 0x01234567 will be stored as following.



Memory representation of integer 0x01234567 inside Big and little endian machines

How to see memory representation of multibyte data types on your machine?

Here is a sample C code that shows the byte representation of int, float and pointer.

```

#include <stdio.h>

/* function to show bytes in memory, from location start to start+n*/
void show_mem_rep(char *start, int n)
{
    int i;
    for (i = 0; i < n; i++)

```

```

    printf("% .2x", start[i]);
    printf("\n");
}

/*Main function to call above function for 0x01234567*/
int main()
{
    int i = 0x01234567;
    show_mem_rep((char*)&i, sizeof(i));
    getchar();
    return 0;
}

```

When above program is run on little endian machine, gives "67 45 23 01" as output , while if it is run on endian machine, gives "01 23 45 67" as output.

Is there a quick way to determine endianness of your machine?

There are n no. of ways for determining endianness of your machine. Here is one quick way of doing the same.

```

#include <stdio.h>
int main()
{
    unsigned int i = 1;
    char *c = (char*)&i;
    if (*c)
        printf("Little endian");
    else
        printf("Big endian");
    getchar();
    return 0;
}

```

In the above program, a character pointer c is pointing to an integer i. Since size of character is 1 byte when the character pointer is de-referenced it will contain only first byte of integer. If machine is little endian then *c will be 1 (because last byte is stored first) and if machine is big endian then *c will be 0.

Does endianness matter for programmers?

Most of the times compiler takes care of endianness, however, endianness becomes an issue in following cases.

It matters in network programming: Suppose you write integers to file on a little endian machine and you transfer this file to a big endian machine. Unless there is little andian to big endian transformation, big endian machine will read the file in reverse order. You can find such a practical example here.

Standard byte order for networks is big endian, also known as network byte order. Before transferring data on network, data is first converted to network byte order (big endian).

Sometimes it matters when you are using type casting, below program is an example.

```

#include <stdio.h>
int main()
{
    unsigned char arr[2] = {0x01, 0x00};
    unsigned short int x = *(unsigned short int *) arr;
    printf("%d", x);
    getchar();
    return 0;
}

```

In the above program, a char array is typecasted to an unsigned short integer type. When I run above program on little endian machine, I get 1 as output, while if I run it on a big endian machine I get 256. To make programs endianness independent, above programming style should be avoided.

What are bi-endians?

Bi-endian processors can run in both modes little and big endian.

What are the examples of little, big endian and bi-endian machines ?

Intel based processors are little endians. ARM processors were little endians. Current generation ARM processors are bi-endian.

Motorola 68K processors are big endians. PowerPC (by Motorola) and SPARK (by Sun) processors were big endian. Current version of these processors are bi-endians.

Does endianness effects file formats?

File formats which have 1 byte as a basic unit are independent of endianness e.g., ASCII files . Other file formats use some fixed endianness format e.g, JPEG files are stored in big endian format.

Which one is better — little endian or big endian

The term little and big endian came from Gulliver's Travels by Jonathan Swift. Two groups could not agree by which end a egg should be opened -a-the little or the big. Just like the egg issue, there is no technological reason to choose one byte ordering convention over the other, hence the arguments degenerate into bickering about sociopolitical issues. As long as one of the conventions is selected and adhered to consistently, the choice is arbitrary.

GATE CS Corner Company Wise Coding Practice

[Bit Magic](#)
[Articles](#)
[Big Endian](#)
[Bit Magic](#)
[Endianness](#)
[Little Endian](#)
[Tutorial](#)

Write an Efficient C Program to Reverse Bits of a Number

Given an unsigned integer, reverse all bits of it and return the number with reversed bits.

```

Input : n = 1
Output : 2147483648
On a machine with size of unsigned
bit as 32. Reverse of 0....001 is
100....0.

```

```

Input : n = 2147483648
Output : 1

```

Method1 – Simple

Loop through all the bits of an integer. If a bit at ith position is set in the i/p no. then set the bit at (NO_OF_BITS – 1) – i in o/p. Where NO_OF_BITS is number of bits present in the given number.

```

/* Function to reverse bits of num */
unsigned int reverseBits(unsigned int num)

```

```

{
    unsigned int NO_OF_BITS = sizeof(num) * 8;
    unsigned int reverse_num = 0, i, temp;

    for (i = 0; i < NO_OF_BITS; i++)
    {
        temp = (num & (1 << i));
        if(temp)
            reverse_num |= (1 << ((NO_OF_BITS - 1) - i));
    }

    return reverse_num;
}

/* Driver function to test above function */
int main()
{
    unsigned int x = 2;
    printf("%u", reverseBits(x));
    getchar();
}

```

Above program can be optimized by removing the use of variable temp. See below the modified code.

```

unsigned int reverseBits(unsigned int num)
{
    unsigned int NO_OF_BITS = sizeof(num) * 8;
    unsigned int reverse_num = 0;
    int i;

    for (i = 0; i < NO_OF_BITS; i++)
    {
        if((num & (1 << i)))
            reverse_num |= 1 << ((NO_OF_BITS - 1) - i);
    }

    return reverse_num;
}

```

Time Complexity: O(log n)

Space Complexity: O(1)

Method 2 – Standard

The idea is to keep putting set bits of the num in reverse_num until num becomes zero. After num becomes zero, shift the remaining bits of reverse_num.

Let num is stored using 8 bits and num be 00000110. After the loop you will get reverse_num as 00000011. Now you need to left shift reverse_num 5 more times and you get the exact reverse 01100000.

```

unsigned int reverseBits(unsigned int num)
{
    unsigned int count = sizeof(num) * 8 - 1;
    unsigned int reverse_num = num;

    num >= 1;
    while(num)
    {
        reverse_num <= 1;
        reverse_num |= num & 1;
        num >= 1;
        count--;
    }
    reverse_num <= count;
    return reverse_num;
}

int main()
{
    unsigned int x = 1;
    printf("%u", reverseBits(x));
    getchar();
}

```

Time Complexity: O(log n)

Space Complexity: O(1)

Method 3 – Lookup Table:

We can reverse the bits of a number in O(1) if we know the size of the number. We can implement it using look up table. Go through the below link for details. You will find some more interesting bit related stuff there.

Source :

<https://graphics.stanford.edu/~seander/bithacks.html>

GATE CS Corner Company Wise Coding Practice

Bit Magic
Bit Magic

Count set bits in an integer

Write an efficient program to count number of 1s in binary representation of an integer.

Examples

```

Input : n = 6
Output : 2
Binary representation of 6 is 110 and has 2 set bits

Input : n = 13
Output : 3
Binary representation of 11 is 1101 and has 3 set bits

```

1. **Simple Method** Loop through all bits in an integer, check if a bit is set and if it is then increment the set bit count. See below program.

```

/* Function to get no of set bits in binary
representation of passed binary no.*/

```

```

unsigned int countSetBits(unsigned int n)
{
    unsigned int count = 0;
    while(n)
    {
        count += n & 1;
        n >= 1;
    }
    return count;
}

/* Program to test function countSetBits */
int main()
{
    int i = 9;
    printf("%d", countSetBits(i));
    getchar();
    return 0;
}

```

Time Complexity: $O(\log n)$ (Theta of logn)

2. Brian Kernighan's Algorithm:

Subtraction of 1 from a number toggles all the bits (from right to left) till the rightmost set bit(including the rightmost set bit). So if we subtract a number by 1 and do bitwise & with itself ($n \& (n-1)$), we unset the rightmost set bit. If we do $n \& (n-1)$ in a loop and count the no of times loop executes we get the set bit count.

Beauty of this solution is number of times it loops is equal to the number of set bits in a given integer.

```

1 Initialize count = 0
2 If integer n is not zero
   (a) Do bitwise & with (n-1) and assign the value back to n
       n = n&(n-1)
   (b) Increment count by 1
   (c) go to step 2
3 Else return count

```

Implementation of Brian Kernighan's Algorithm:

```

#include<stdio.h>

/* Function to get no of set bits in binary
representation of passed binary no. */
unsigned int countSetBits(int n)
{
    unsigned int count = 0;
    while (n)
    {
        n &= (n-1);
        count++;
    }
    return count;
}

/* Program to test function countSetBits */
int main()
{
    int i = 9;
    printf("%d", countSetBits(i));
    getchar();
    return 0;
}

```

Example for Brian Kernighan's Algorithm:

```

n = 9 (1001)
count = 0

Since 9 > 0, subtract by 1 and do bitwise & with (9-1)
n = 9&8 (1001 & 1000)
n = 8
count = 1

Since 8 > 0, subtract by 1 and do bitwise & with (8-1)
n = 8&7 (1000 & 0111)
n = 0
count = 2

Since n = 0, return count which is 2 now.

```

Time Complexity: $O(\log n)$

3. Using Lookup table: We can count bits in $O(1)$ time using lookup table. Please see <http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetTable> for details.

We can find one use of counting set bits at <http://geeksforgeeks.org/?p=1465>

Note: In GCC, we can directly count set bits using `__builtin_popcount()`. So we can avoid a separate function for counting set bits.

```

// C++ program to demonstrate __builtin_popcount()
#include <iostream>
using namespace std;

int main()
{
    cout << __builtin_popcount(4) << endl;
    cout << __builtin_popcount(15);

    return 0;
}

```

Output :

```

1
4

```

References:

<http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetNaive>

GATE CS Corner Company Wise Coding Practice

Bit Magic
Bit Magic
setBitCount

Count number of bits to be flipped to convert A to B

Suggested by Dheeraj

Question: You are given two numbers A and B. Write a program to count number of bits needed to be flipped to convert A to B.

Solution:

1. Calculate XOR of A and B.
`a_xor_b = A ^ B`
2. Count the set bits in the above calculated XOR result.
`countSetBits(a_xor_b)`

XOR of two number will have set bits only at those places where A differs from B.

Example:

```
A = 1001001
B = 0010101
a_xor_b = 1011100
No of bits need to flipped = set bit count in a_xor_b i.e. 4
```

We strongly recommend that you click here and practice it, before moving on to the solution.

To get the set bit count please see another post on this portal <http://geeksforgeeks.org/?p=1176>

GATE CS Corner Company Wise Coding Practice

Bit Magic
Bit Magic
XOR

Smallest power of 2 greater than or equal to n

Write a function that, for a given no n, finds a number p which is greater than or equal to n and is a power of 2.

```
Input : n = 5
Output: 8

Input :n = 17
Output :32

Input :n = 32
Output :32
```

There are plenty of solutions for this. Let us take the example of 17 to explain some of them.

Method 1(Using Log of the number)

1. Calculate Position of set bit in p(next power of 2):
`pos = ceil(log(n) / log(2))`
2. Now calculate p:
`p = pow(2, pos)`

Example

```
Let us try for 17
pos = 5
p = 32
```

Method 2 (By getting the position of only set bit in result)

```

/* If n is a power of 2 then return n */
1 If (n & !(n&(n-1))) then return n
2 Else keep right shifting n until it becomes zero
   and count no of shifts
a. Initialize: count = 0
b. While n != 0
   n = n>>1
   count = count + 1

/* Now count has the position of set bit in result */
3 Return (1

```

Example:

```

Let us try for 17
  count = 5
  p  = 32

```

```

unsigned int nextPowerOf2(unsigned int n)
{
unsigned count = 0;

/* First n in the below condition is for the case where n is 0*/
if (n && !(n&(n-1)))
return n;

while( n != 0)
{
n >= 1;
count += 1;
}

return 1 << count;
}

/* Driver program to test above function */
int main()
{
unsigned int n = 0;
printf("%d", nextPowerOf2(n));
return 0;
}

```

Method 3(Shift result one by one)

Thanks to coderyogi for suggesting this method . This method is a variation of method 2 where instead of getting count, we shift the result one by one in a loop.

```

unsigned int nextPowerOf2(unsigned int n)
{
  unsigned int p = 1;
  if (n && !(n & (n - 1)))
    return n;

  while (p < n)
    p <<= 1;

  return p;
}

/* Driver program to test above function */
int main()
{
  unsigned int n = 5;
  printf("%d", nextPowerOf2(n));
  return 0;
}

```

Time Complexity: O(lgn)

Method 4(Customized and Fast)

```

1. Subtract n by 1
n = n -1

2. Set all bits after the leftmost set bit.

/* Below solution works only if integer is 32 bits */
n = n | (n >> 1);
n = n | (n >> 2);
n = n | (n >> 4);
n = n | (n >> 8);
n = n | (n >> 16);
3. Return n + 1

```

Example:

```

Steps 1 & 3 of above algorithm are to handle cases
of power of 2 numbers e.g., 1, 2, 4, 8, 16,

```

```

Let us try for 17(10001)
step 1
n = n - 1 = 16 (10000)

```

```

step 2
n = n | n >> 1
n = 10000 | 01000
n = 11000
n = n | n >> 2
n = 11000 | 00110
n = 11110
n = n | n >> 4
n = 11110 | 00001
n = 11111
n = n | n >> 8
n = 11111 | 00000
n = 11111
n = n | n >> 16
n = 11110 | 00000
n = 11111

```

step 3: Return n+1
We get n + 1 as 100000 (32)

Program:

```

#include <stdio.h>

/* Finds next power of two for n. If n itself
   is a power of two then returns n*/
unsigned int nextPowerOf2(unsigned int n)
{
    n--;
    n |= n >> 1;
    n |= n >> 2;
    n |= n >> 4;
    n |= n >> 8;
    n |= n >> 16;
    n++;
    return n;
}

/* Driver program to test above function */
int main()
{
    unsigned int n = 5;
    printf("%d", nextPowerOf2(n));
    return 0;
}

```

Time Complexity: $O(\lg n)$

Related Post:

[Highest power of 2 less than or equal to given number](#)

References:

http://en.wikipedia.org/wiki/Power_of_2

GATE CS Corner

Company Wise Coding Practice

Bit Magic

Write an Efficient Method to Check if a Number is Multiple of 3

The very first solution that comes to our mind is the one that we learned in school. If sum of digits in a number is multiple of 3 then number is multiple of 3 e.g., for 612 sum of digits is 9 so it's a multiple of 3. But this solution is not efficient. You have to get all decimal digits one by one, add them and then check if sum is multiple of 3.

There is a pattern in binary representation of the number that can be used to find if number is a multiple of 3. If difference between count of odd set bits (Bits set at odd positions) and even set bits is multiple of 3 then is the number.

Example: 23 (00..10111)

- 1) Get count of all set bits at odd positions (For 23 it's 3).
- 2) Get count of all set bits at even positions (For 23 it's 1).
- 3) If difference of above two counts is a multiple of 3 then number is also a multiple of 3.

(For 23 it's 2 so 23 is not a multiple of 3)

Take some more examples like 21, 15, etc...

```
Algorithm: isMultipleOf3(n)
1) Make n positive if n is negative.
2) If number is 0 then return 1
3) If number is 1 then return 0
4) Initialize: odd_count = 0, even_count = 0
5) Loop while n != 0
   a) If rightmost bit is set then increment odd count.
   b) Right-shift n by 1 bit
   c) If rightmost bit is set then increment even count.
   d) Right-shift n by 1 bit
6) return isMultipleOf3(odd_count - even_count)
```

Proof:

Above can be proved by taking the example of 11 in decimal numbers. (In this context 11 in decimal numbers is same as 3 in binary numbers)
If difference between sum of odd digits and even digits is multiple of 11 then decimal number is multiple of 11. Let's see how.

Let's take the example of 2 digit numbers in decimal

$$AB = 11A \cdot A + B = 11A + (B - A)$$

So if $(B - A)$ is a multiple of 11 then is AB.

Let us take 3 digit numbers.

$$ABC = 99A + A + 11B - B + C = (99A + 11B) + (A + C - B)$$

So if $(A + C - B)$ is a multiple of 11 then is (A+C-B)

Let us take 4 digit numbers now.

$$ABCD = 1001A + D + 11C - C + 999B + B - A$$

$$= (1001A - 999B + 11C) + (D + B - A - C)$$

So, if $(B + D - A - C)$ is a multiple of 11 then is ABCD.

This can be continued for all decimal numbers.

Above concept can be proved for 3 in binary numbers in the same way.

Time Complexity: O(logn)

Program:

```
#include<stdio.h>

/* Function to check if n is a multiple of 3*/
int isMultipleOf3(int n)
{
    int odd_count = 0;
    int even_count = 0;

    /* Make no positive if +n is multiple of 3
     * then is -n. We are doing this to avoid
     * stack overflow in recursion*/
    if(n < 0) n = -n;
    if(n == 0) return 1;
    if(n == 1) return 0;

    while(n)
    {
        /* If odd bit is set then
         * increment odd counter */
        if(n & 1)
            odd_count++;
        n = n>>1;

        /* If even bit is set then
         * increment even counter */
        if(n & 1)
            even_count++;
        n = n>>1;
    }

    return isMultipleOf3(abs(odd_count - even_count));
}

/* Program to test function isMultipleOf3 */
int main()
{
    int num = 23;
    if (isMultipleOf3(num))
        printf("num is multiple of 3");
    else
        printf("num is not a multiple of 3");
    return 0;
}
```

Related Articles:

[Check divisibility in a binary stream](#)

[DFA based division](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Bit Magic
MathematicalAlgo

Write a C program to find the parity of an unsigned integer

Parity: Parity of a number refers to whether it contains an odd or even number of 1-bits. The number has "odd parity", if it contains odd number of 1-bits and is "even parity" if it contains even number of 1-bits.

Main idea of the below solution is – Loop while n is not 0 and in loop unset one of the set bits and invert parity.

```
Algorithm: getParity(n)
```

```

1. Initialize parity = 0
2. Loop while n != 0
   a. Invert parity
      parity = !parity
   b. Unset rightmost set bit
      n = n & (n-1)
3. return parity

```

Example:

Initialize: n = 13 (1101) parity = 0

```

n = 13 & 12 = 12 (1100) parity = 1
n = 12 & 11 = 8 (1000) parity = 0
n = 8 & 7 = 0 (0000) parity = 1

```

Program:

```

#include <stdio.h>
#define bool int

/* Function to get parity of number n. It returns 1
   if n has odd parity, and returns 0 if n has even
   parity */
bool getParity(unsigned int n)
{
    bool parity = 0;
    while (n)
    {
        parity = !parity;
        n = n & (n - 1);
    }
    return parity;
}

/* Driver program to test getParity() */
int main()
{
    unsigned int n = 7;
    printf("Parity of no %d = %s", n,
           (getParity(n)) ? "odd": "even"));

    getchar();
    return 0;
}

```

Above solution can be optimized by using lookup table. Please refer to Bit Twiddle Hacks[1st reference] for details.

Time Complexity: The time taken by above algorithm is proportional to the number of bits set. Worst case complexity is O(Logn).

Uses: Parity is used in error detection and cryptography.

References:

<http://graphics.stanford.edu/~seander/bithacks.html#ParityNaive> – last checked on 30 May 2009.

GATE CS Corner Company Wise Coding Practice

Bit Magic
Bit Magic

Efficient way to multiply with 7

We can multiply a number by 7 using bitwise operator. First left shift the number by 3 bits (you will get 8n) then subtract the original number from the shifted number and return the difference ($8n - n$).

Program:

```

#include<stdio.h>

int multiplyBySeven(unsigned int n)
{
    /* Note the inner bracket here. This is needed
       because precedence of '^' operator is higher
       than '<<' */
    return ((n<<3) - n);
}

/* Driver program to test above function */
int main()
{
    unsigned int n = 4;
    printf("%u", multiplyBySeven(n));

    getchar();
    return 0;
}

```

Time Complexity: O(1)

Space Complexity: O(1)

Note: Works only for positive integers.

Same concept can be used for fast multiplication by 9 or other numbers.

GATE CS Corner Company Wise Coding Practice

Bit Magic
Bit Magic
MathematicalAlgo

Write one line C function to find whether a no is power of two

Given a positive integer, write a function to find if it is a power of two or not.

Examples:

```
Input : n = 4
Output : Yes
22 = 4
```

```
Input : n = 7
Output : No
```

```
Input : n = 32
Output : Yes
25 = 32
```

We strongly recommend that you click here and practice it, before moving on to the solution.

1. A simple method for this is to simply take the log of the number on base 2 and if you get an integer then number is power of 2.

2. Another solution is to keep dividing the number by two, i.e, do $n = n/2$ iteratively. In any iteration, if $n \% 2$ becomes non-zero and n is not 1 then n is not a power of 2. If n becomes 1 then it is a power of 2.

```
#include<stdio.h>
#define bool int

/* Function to check if x is power of 2*/
bool isPowerOfTwo(int n)
{
    if (n == 0)
        return 0;
    while (n != 1)
    {
        if (n%2 != 0)
            return 0;
        n = n/2;
    }
    return 1;
}

/*Driver program to test above function*/
int main()
{
    isPowerOfTwo(31)? printf("Yes\n"): printf("No\n");
    isPowerOfTwo(17)? printf("Yes\n"): printf("No\n");
    isPowerOfTwo(16)? printf("Yes\n"): printf("No\n");
    isPowerOfTwo(2)? printf("Yes\n"): printf("No\n");
    isPowerOfTwo(18)? printf("Yes\n"): printf("No\n");
    isPowerOfTwo(1)? printf("Yes\n"): printf("No\n");
    return 0;
}
```

Output:

```
No
No
Yes
Yes
No
Yes
```

3. All power of two numbers have only one bit set. So count the no. of set bits and if you get 1 then number is a power of 2. Please see <http://geeksforgeeks.org/?p=1176> for counting set bits.

4. If we subtract a power of 2 numbers by 1 then all unset bits after the only set bit become set; and the set bit become unset.

For example for 4 (100) and 16(10000), we get following after subtracting 1

3 -> 011

15 -> 0111

So, if a number n is a power of 2 then bitwise & of n and $n-1$ will be zero. We can say n is a power of 2 or not based on value of $n \& (n-1)$. The expression $n \& (n-1)$ will not work when n is 0. To handle this case also, our expression will become $n \& (n \& (n-1))$ (thanks to [Mohammad](#) for adding this case).

Below is the implementation of this method.

```
#include<stdio.h>
#define bool int

/* Function to check if x is power of 2*/
bool isPowerOfTwo (int x)
{
    /* First x in the below expression is for the case when x is 0 */
    return x && (!x&(x-1));
}

/*Driver program to test above function*/
int main()
{
    isPowerOfTwo(31)? printf("Yes\n"): printf("No\n");
    isPowerOfTwo(17)? printf("Yes\n"): printf("No\n");
    isPowerOfTwo(16)? printf("Yes\n"): printf("No\n");
    isPowerOfTwo(2)? printf("Yes\n"): printf("No\n");
    isPowerOfTwo(18)? printf("Yes\n"): printf("No\n");
    isPowerOfTwo(1)? printf("Yes\n"): printf("No\n");
    return 0;
}
```

Output:

```
No
No
Yes
Yes
No
```

Yes

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Bit Magic
Bit Magic

Position of rightmost set bit

Write a one line C function to return position of first 1 from right to left, in binary representation of an Integer.

I/P 18, Binary Representation 010010
O/P 2
I/P 19, Binary Representation 010011
O/P 1

Let I/P be 12 (1100)

Algorithm: (Example 18(010010))

1. Take two's complement of the given no as all bits are reverted except the first '1' from right to left (10111)
2. Do an bit-wise & with original no, this will return no with the required one only (00010)
3. Take the log2 of the no, you will get position -1 (1)
4. Add 1 (2)

Program:

```
#include<stdio.h>
#include<math.h>

unsigned int getFirstSetBitPos(int n)
{
    return log2(n&-n)+1;
}

int main()
{
    int n = 12;
    printf("%u", getFirstSetBitPos(n));
    getchar();
    return 0;
}
```

GATE CS Corner Company Wise Coding Practice

Bit Magic
Bit Magic

Binary representation of a given number

Write a program to print Binary representation of a given number.

Source: Microsoft Interview Set-3

Method 1: Iterative

For any number, we can check whether its 'i'th bit is 0(OFF) or 1(ON) by bitwise ANDing it with " 2^i " (2 raise to i).

- 1) Let us take number 'NUM' and we want to check whether its 0th bit is ON or OFF
bit = 2^0 (0th bit)
if NUM & bit == 1 means 0th bit is ON else 0th bit is OFF
- 2) Similarly if we want to check whether 5th bit is ON or OFF
bit = 2^5 (5th bit)
if NUM & bit == 1 means its 5th bit is ON else 5th bit is OFF.

Let us take unsigned integer (32 bit), which consist of 0-31 bits. To print binary representation of unsigned integer, start from 31th bit, check whether 31th bit is ON or OFF, if it is ON print "1" else print "0". Now check whether 30th bit is ON or OFF, if it is ON print "1" else print "0", do this for all bits from 31 to 0, finally we will get binary representation of number.

```
void bin(unsigned n)
{
    unsigned i;
    for (i = 1 << 31; i > 0; i = i / 2)
        (n & i)? printf("1"): printf("0");
}

int main(void)
{
    bin(7);
    printf("\n");
    bin(4);
}
```

Method 2: Recursive

Following is recursive method to print binary representation of 'NUM'.

- step 1) if NUM > 1
a) push NUM on stack
b) recursively call function with 'NUM / 2'
step 2)
a) pop NUM from stack, divide it by 2 and print its remainder.

```

void bin(unsigned n)
{
    /* step 1 */
    if (n > 1)
        bin(n/2);

    /* step 2 */
    printf("%d", n % 2);
}

int main(void)
{
    bin(7);
    printf("\n");
    bin(4);
}

```

This article is compiled by **Narendra Kangalkar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Bit Magic

Swap all odd and even bits

Given an unsigned integer, swap all odd bits with even bits. For example, if the given number is 23 (**00010111**), it should be converted to 43 (**00101011**). Every even position bit is swapped with adjacent bit on right side (even position bits are highlighted in binary representation of 23), and every odd position bit is swapped with adjacent on left side.

If we take a closer look at the example, we can observe that we basically need to right shift ($>>$) all even bits (In the above example, even bits of 23 are highlighted) by 1 so that they become odd bits (highlighted in 43), and left shift ($<<$) all odd bits by 1 so that they become even bits. The following solution is based on this observation. The solution assumes that input number is stored using 32 bits.

Let the input number be x

- 1) Get all even bits of x by doing bitwise and of x with **0xAAAAAAA**. The number **0xAAAAAAA** is a 32 bit number with all even bits set as 1 and all odd bits as 0.
- 2) Get all odd bits of x by doing bitwise and of x with **0x55555555**. The number **0x55555555** is a 32 bit number with all odd bits set as 1 and all even bits as 0.
- 3) Right shift all even bits.
- 4) Left shift all odd bits.
- 5) Combine new even and odd bits and return.

C/C++

```

// C program to swap even and odd bits of a given number
#include <stdio.h>

unsigned int swapBits(unsigned int x)
{
    // Get all even bits of x
    unsigned int even_bits = x & 0xAAAAAAA;

    // Get all odd bits of x
    unsigned int odd_bits = x & 0x55555555;

    even_bits >>= 1; // Right shift even bits
    odd_bits <<= 1; // Left shift odd bits

    return (even_bits | odd_bits); // Combine even and odd bits
}

// Driver program to test above function
int main()
{
    unsigned int x = 23; // 00010111

    // Output is 43 (00101011)
    printf("%u ", swapBits(x));

    return 0;
}

```

Output:

43

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Find position of the only set bit

Given a number having only one '1' and all other '0's in its binary representation, find position of the only set bit. Source: Microsoft Interview | 18

We strongly recommend that you click here and practice it, before moving on to the solution.

The idea is to start from rightmost bit and one by one check value of every bit. Following is detailed algorithm.

- 1) If number is power of two then and then only its binary representation contains only one '1'. That's why check whether given number is power of 2 or not. If given number is not power of 2, then print error message and exit.
- 2) Initialize two variables; $i = 1$ (for looping) and $pos = 1$ (to find position of set bit)
- 3) Inside loop, do bitwise AND of i and number ' n '. If value of this operation is true, then "pos" bit is set, so break the loop and return position. Otherwise, increment "pos" by 1 and left shift i by 1 and repeat the procedure.

```
// C program to find position of only set bit in a given number
#include <stdio.h>

// A utility function to check whether n is power of 2 or not. See http://goo.gl/17Arj
int isPowerOfTwo(unsigned n)
{
    return n && !(n & (n-1));
}

// Returns position of the only set bit in 'n'
int findPosition(unsigned n)
{
    if (!isPowerOfTwo(n))
        return -1;

    unsigned i = 1, pos = 1;

    // Iterate through bits of n till we find a set bit
    // i&n will be non-zero only when 'i' and 'n' have a set bit
    // at same position
    while (!(i & n))
    {
        // Unset current bit and set the next bit in 'i'
        i = i << 1;

        // increment position
        ++pos;
    }

    return pos;
}

// Driver program to test above function
int main(void)
{
    int n = 16;
    int pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
        printf("n = %d, Position %d \n", n, pos);

    n = 12;
    pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
        printf("n = %d, Position %d \n", n, pos);

    n = 128;
    pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
        printf("n = %d, Position %d \n", n, pos);

    return 0;
}
```

Output:

```
n = 16, Position 5
n = 12, Invalid number
n = 128, Position 8
```

Following is **another method** for this problem. The idea is to one by one right shift the set bit of given number ' n ' until ' n ' becomes 0. Count how many times we shifted to make ' n ' zero. The final count is position of the set bit.

```
// C program to find position of only set bit in a given number
#include <stdio.h>

// A utility function to check whether n is power of 2 or not
int isPowerOfTwo(unsigned n)
{
    return n && !(n & (n-1));
}

// Returns position of the only set bit in 'n'
int findPosition(unsigned n)
{
    if (!isPowerOfTwo(n))
        return -1;

    unsigned count = 0;

    // One by one move the only set bit to right till it reaches end
    while (n)
    {
        n = n >> 1;

        // increment count of shifts
        ++count;
    }
}
```

```

    return count;
}

// Driver program to test above function
int main(void)
{
    int n = 0;
    int pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
        printf("n = %d, Position %d \n", n, pos);

    n = 12;
    pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
        printf("n = %d, Position %d \n", n, pos);

    n = 128;
    pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
        printf("n = %d, Position %d \n", n, pos);

    return 0;
}

```

Output:

```

n = 0, Invalid number
n = 12, Invalid number
n = 128, Position 8

```

We can also use log base 2 to find the position. Thanks to Arunkumar for suggesting this solution.

```

#include <stdio.h>

unsigned int Log2n(unsigned int n)
{
    return (n > 1)? 1 + Log2n(n/2): 0;
}

int isPowerOfTwo(unsigned n)
{
    return n && (! (n & (n-1)));
}

int findPosition(unsigned n)
{
    if (!isPowerOfTwo(n))
        return -1;
    return Log2n(n) + 1;
}

// Driver program to test above function
int main(void)
{
    int n = 0;
    int pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
        printf("n = %d, Position %d \n", n, pos);

    n = 12;
    pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
        printf("n = %d, Position %d \n", n, pos);

    n = 128;
    pos = findPosition(n);
    (pos == -1)? printf("n = %d, Invalid number\n", n):
        printf("n = %d, Position %d \n", n, pos);

    return 0;
}

```

Output:

```

n = 0, Invalid number
n = 12, Invalid number
n = 128, Position 8

```

This article is compiled by Narendra Kangalkar. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Bit Magic

Divide and Conquer | Set 4 (Karatsuba algorithm for fast multiplication)

Given two binary strings that represent value of two integers, find the product of two strings. For example, if the first bit string is "1100" and second bit string is "1010", output should be 120.

For simplicity, let the length of two strings be same and be n.

A **Naive Approach** is to follow the process we study in school. One by one take all bits of second number and multiply it with all bits of first number. Finally add all multiplications. This algorithm takes $O(n^2)$ time.

product



Using **Divide and Conquer**, we can multiply two integers in less time complexity. We divide the given numbers in two halves. Let the given numbers be X and Y.

For simplicity let us assume that n is even

```
X = XI*2n/2 + Xr [XI and Xr contain leftmost and rightmost n/2 bits of X]  
Y = YI*2n/2 + Yr [YI and Yr contain leftmost and rightmost n/2 bits of Y]
```

The product XY can be written as following.

```
XY = (XI*2n/2 + Xr)(YI*2n/2 + Yr)  
= 2n XIYI + 2n/2(XIYr + XrYl) + XrYr
```

If we take a look at the above formula, there are four multiplications of size n/2, so we basically divided the problem of size n into for sub-problems of size n/2. But that doesn't help because solution of recurrence $T(n) = 4T(n/2) + O(n)$ is $O(n^2)$. The tricky part of this algorithm is to change the middle two terms to some other form so that only one extra multiplication would be sufficient. The following is tricky expression for middle two terms.

```
XIYr + XrYl = (XI + Xr)(YI + Yr) - XIYI - XrYr
```

So the final value of XY becomes

```
XY = 2n XIYI + 2n/2 * [(XI + Xr)(YI + Yr) - XIYI - XrYr] + XrYr
```

With above trick, the recurrence becomes $T(n) = 3T(n/2) + O(n)$ and solution of this recurrence is $O(n^{1.59})$.

What if the lengths of input strings are different and are not even? To handle the different length case, we append 0's in the beginning. To handle odd length, we put $\text{floor}(n/2)$ bits in left half and $\text{ceil}(n/2)$ bits in right half. So the expression for XY changes to following.

```
XY = 22\text{ceil}(n/2) XIYI + 2\text{ceil}(n/2) * [(XI + Xr)(YI + Yr) - XIYI - XrYr] + XrYr
```

The above algorithm is called Karatsuba algorithm and it can be used for any base.

Following is C++ implementation of above algorithm.

```
// C++ implementation of Karatsuba algorithm for bit string multiplication.  
#include<iostream>  
#include<stdio.h>  
  
using namespace std;  
  
// FOLLOWING TWO FUNCTIONS ARE COPIED FROM http://goo.gl/q0OhZ  
// Helper method: given two unequal sized bit strings, converts them to  
// same length by adding leading 0s in the smaller string. Returns the  
// new length  
int makeEqualLength(string &str1, string &str2)  
{  
    int len1 = str1.size();  
    int len2 = str2.size();  
    if (len1 < len2)  
    {  
        for (int i = 0 ; i < len2 - len1 ; i++)  
            str1 = '0' + str1;  
        return len2;  
    }  
    else if (len1 > len2)  
    {  
        for (int i = 0 ; i < len1 - len2 ; i++)  
            str2 = '0' + str2;  
    }  
    return len1; // If len1 >= len2  
}  
  
// The main function that adds two bit sequences and returns the addition  
string addBitStrings( string first, string second )  
{  
    string result; // To store the sum bits  
  
    // make the lengths same before adding  
    int length = makeEqualLength(first, second);  
    int carry = 0; // Initialize carry  
  
    // Add all bits one by one  
    for (int i = length-1 ; i >= 0 ; i--)  
    {  
        int firstBit = first.at(i) - '0';  
        int secondBit = second.at(i) - '0';  
  
        // boolean expression for sum of 3 bits  
        int sum = (firstBit ^ secondBit ^ carry) + '0';  
  
        result = (char)sum + result;  
  
        // boolean expression for 3-bit addition  
        carry = (firstBit&secondBit) | (secondBit&carry) | (firstBit&carry);  
    }  
  
    // if overflow, then add a leading 1  
    if (carry) result = '1' + result;  
  
    return result;  
}  
  
// A utility function to multiply single bits of strings a and b  
int multiplySingleBit(string a, string b)  
{  
    return (a[0] - '0')*(b[0] - '0');  
}  
  
// The main function that multiplies two bit strings X and Y and returns  
// result as long integer  
long int multiply(string X, string Y)  
{  
    // Find the maximum of lengths of x and Y and make length  
    // of smaller string same as that of larger string  
    int n = makeEqualLength(X, Y);  
  
    // Base cases
```

```

if (n == 0) return 0;
if (n == 1) return multiplySingleBit(X, Y);

int fh = n/2; // First half of string, floor(n/2)
int sh = (n-fh); // Second half of string, ceil(n/2)

// Find the first half and second half of first string.
// Refer http://goo.gl/Lmgn for substr method
string Xl = X.substr(0, fh);
string Xr = X.substr(fh, sh);

// Find the first half and second half of second string
string Yl = Y.substr(0, fh);
string Yr = Y.substr(fh, sh);

// Recursively calculate the three products of inputs of size n/2
long int P1 = multiply(Xl, Yl);
long int P2 = multiply(Xr, Yr);
long int P3 = multiply(addBitStrings(Xl, Xr), addBitStrings(Yl, Yr));

// Combine the three products to get the final result.
return P1*(1<<(2*sh)) + (P3 - P1 - P2)*(1<<sh) + P2;
}

// Driver program to test above functions
int main()
{
    printf ("%ld\n", multiply("1100", "1010"));
    printf ("%ld\n", multiply("110", "1010"));
    printf ("%ld\n", multiply("11", "1010"));
    printf ("%ld\n", multiply("1", "1010"));
    printf ("%ld\n", multiply("0", "1010"));
    printf ("%ld\n", multiply("111", "111"));
    printf ("%ld\n", multiply("11", "11"));
}

```

Output:

```

120
60
30
10
0
49
9

```

Time Complexity: Time complexity of the above solution is O($n^{1.59}$).

Time complexity of multiplication can be further improved using another Divide and Conquer algorithm, fast Fourier transform. We will soon be discussing fast Fourier transform as a separate post.

Exercise

The above program returns a long int value and will not work for big strings. Extend the above program to return a string instead of a long int value.

References:

Wikipedia page for Karatsuba algorithm

Algorithms 1st Edition by Sanjoy Dasgupta, Christos Papadimitriou and Umesh Vazirani

<http://courses.csail.mit.edu/6.006/spring11/exams/notes3-karatsuba>

<http://www.cc.gatech.edu/~ninanf/Algos11/lectures/lect0131.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner Company Wise Coding Practice

Divide and Conquer
Divide and Conquer

How to swap two numbers without using a temporary variable?

Given two variables, x and y, swap two variables without using a third variable.

Method 1 (Using Arithmetic Operators)

The idea is to get sum in one of the two given numbers. The numbers can then be swapped using the sum and subtraction from sum.

```

#include <stdio.h>
int main()
{
    int x = 10, y = 5;

    // Code to swap 'x' and 'y'
    x = x + y; // x now becomes 15
    y = x - y; // y becomes 10
    x = x - y; // x becomes 5

    printf("After Swapping: x = %d, y = %d", x, y);

    return 0;
}

```

Output:

```

After Swapping: x = 5, y = 10

```

Multiplication and division can also be used for swapping.

```

#include <stdio.h>
int main()
{
    int x = 10, y = 5;

    // Code to swap 'x' and 'y'
    x = x * y; // x now becomes 50
    y = x / y; // y becomes 10

```

```

x = x / y; // x becomes 5
printf("After Swapping: x = %d, y = %d", x, y);
return 0;
}

```

Output:

After Swapping: x = 5, y = 10

Method 2 (Using Bitwise XOR)

The bitwise XOR operator can be used to swap two variables. The XOR of two numbers x and y returns a number which has all the bits as 1 wherever bits of x and y differ. For example XOR of 10 (In Binary 1010) and 5 (In Binary 0101) is 1111 and XOR of 7 (0111) and 5 (0101) is 0010.

```

#include <stdio.h>
int main()
{
    int x = 10, y = 5;

    // Code to swap 'x' (1010) and 'y' (0101)
    x = x ^ y; // x now becomes 15 (1111)
    y = x ^ y; // y becomes 10 (1010)
    x = x ^ y; // x becomes 5 (0101)

    printf("After Swapping: x = %d, y = %d", x, y);

    return 0;
}

```

Output:

After Swapping: x = 5, y = 10

Problems with above methods

- 1) The multiplication and division based approach doesn't work if one of the numbers is 0 as the product becomes 0 irrespective of the other number.
- 2) Both Arithmetic solutions may cause arithmetic overflow. If x and y are too large, addition and multiplication may go out of integer range.
- 3) When we use pointers to variable and make a function swap, all of the above methods fail when both pointers point to the same variable. Let's take a look what will happen in this case if both are pointing to the same variable.

```

// Bitwise XOR based method
x = x ^ x; // x becomes 0
x = x ^ x; // x remains 0
x = x ^ x; // x remains 0

// Arithmetic based method
x = x + x; // x becomes 2x
x = x - x; // x becomes 0
x = x - x; // x remains 0

```

Let us see the following program.

```

#include <stdio.h>
void swap(int *xp, int *yp)
{
    *xp = *xp ^ *yp;
    *yp = *xp ^ *yp;
    *xp = *xp ^ *yp;
}

int main()
{
    int x = 10;
    swap(&x, &x);
    printf("After swap(&x, &x): x = %d", x);
    return 0;
}

```

Output:

After swap(&x, &x): x = 0

Swapping a variable with itself may be needed in many standard algorithms. For example see [this implementation of QuickSort](#) where we may swap a variable with itself. The above problem can be avoided by putting a condition before the swapping.

```

#include <stdio.h>
void swap(int *xp, int *yp)
{
    if (xp == yp) // Check if the two addresses are same
        return;
    *xp = *xp + *yp;
    *yp = *xp - *yp;
    *xp = *xp - *yp;
}
int main()
{
    int x = 10;
    swap(&x, &x);
    printf("After swap(&x, &x): x = %d", x);
    return 0;
}

```

Output:

After swap(&x, &x): x = 10

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Check if a number is multiple of 9 using bitwise operators

Given a number n, write a function that returns true if n is divisible by 9, else false. The most simple way to check for n's divisibility by 9 is to do $n \% 9$.

Another method is to sum the digits of n. If sum of digits is multiple of 9, then n is multiple of 9.

The above methods are not bitwise operators based methods and require use of % and /.

The **bitwise operators** are generally faster than modulo and division operators. Following is a bitwise operator based method to check divisibility by 9.

```
#include<iostream>
using namespace std;

// Bitwise operator based function to check divisibility by 9
bool isDivBy9(int n)
{
    // Base cases
    if (n == 0 || n == 9)
        return true;
    if (n < 9)
        return false;

    // If n is greater than 9, then recur for [floor(n/9) - n%8]
    return isDivBy9((int)(n>>3) - (int)(n&7));
}

// Driver program to test above function
int main()
{
    // Let us print all multiples of 9 from 0 to 100
    // using above method
    for (int i = 0; i < 100; i++)
        if (isDivBy9(i))
            cout << i << " ";
    return 0;
}
```

Output:

```
0 9 18 27 36 45 54 63 72 81 90 99
```

How does this work?

$n/9$ can be written in terms of $n/8$ using the following simple formula.

$$n/9 = n/8 - n/72$$

Since we need to use bitwise operators, we get the value of $\text{floor}(n/8)$ using $n>>3$ and get value of $n \% 8$ using $n \& 7$. We need to write above expression in terms of $\text{floor}(n/8)$ and $n \% 8$. $n/8$ is equal to " $\text{floor}(n/8) + (n \% 8)/8$ ". Let us write the above expression in terms of $\text{floor}(n/8)$ and $n \% 8$

$$\begin{aligned} n/9 &= \text{floor}(n/8) + (n \% 8)/8 - [\text{floor}(n/8) + (n \% 8)/8]9 \\ n/9 &= \text{floor}(n/8) - [\text{floor}(n/8) - 9(n \% 8)/8 + (n \% 8)/8]9 \\ n/9 &= \text{floor}(n/8) - [\text{floor}(n/8) - n \% 8]9 \end{aligned}$$

From above equation, n is a multiple of 9 only if the expression $\text{floor}(n/8) - [\text{floor}(n/8) - n \% 8]/9$ is an integer. This expression can only be an integer if the sub-expression $[\text{floor}(n/8) - n \% 8]/9$ is an integer. The subexpression can only be an integer if $[\text{floor}(n/8) - n \% 8]$ is a multiple of 9. So the problem reduces to a smaller value which can be written in terms of bitwise operators.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Swap two nibbles in a byte

A **nibble** is a four-bit aggregation, or half an octet. There are two nibbles in a byte.

Given a byte, swap the two nibbles in it. For example 100 is be represented as 01100100 in a byte (or 8 bits). The two nibbles are (0110) and (0100). If we swap the two nibbles, we get 01000110 which is 70 in decimal.

We strongly recommend that you click here and practice it, before moving on to the solution.

To swap the nibbles, we can use bitwise &, bitwise '>' operators. A byte can be represented using a unsigned char in C as size of char is 1 byte in a typical C compiler. Following is C program to swap the two nibbles in a byte.

```
#include <stdio.h>

unsigned char swapNibbles(unsigned char x)
{
    return ( (x & 0x0F)<<4 | (x & 0xF0)>>4 );
}

int main()
{
    unsigned char x = 100;
    printf("%u", swapNibbles(x));
    return 0;
}
```

Output:

```
70
```

Explanation:

100 is 01100100 in binary. The operation can be split mainly in two parts

1) The expression "`x & 0x0F`" gives us last 4 bits of `x`. For `x = 100`, the result is 00000100. Using bitwise '&' operator, we shift the digit to the right 4 times and make the first four bits as 0. The result after shift is 00000110.

At the end we use the bitwise OR '|' operation of the two expressions explained above. The OR operator places first nibble to the end and last nibble to first. For `x = 100`, the value of (01000000) OR (00000110) gives the result 01000110 which is equal to 70 in decimal.

This article is contributed by **Anuj Garg**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Bit Magic

How to turn off a particular bit in a number?

Difficulty Level: Rookie

Given a number `n` and a value `k`, turn off the `k`'th bit in `n`.

Examples:

```
Input: n = 15, k = 1  
Output: 14
```

```
Input: n = 15, k = 2  
Output: 13
```

```
Input: n = 15, k = 3  
Output: 11
```

```
Input: n = 15, k = 4  
Output: 7
```

```
Input: n = 15, k >= 5  
Output: 15
```

The idea is to use bitwise `~(1 ^)`, we get a number which has all bits set, except the `k`'th bit. If we do bitwise `&` of this expression with `n`, we get a number which has all bits same as `n` except the `k`'th bit which is 0.

Following is C++ implementation of this.

```
#include <iostream>  
using namespace std;  
  
// Returns a number that has all bits same as n  
// except the k'th bit which is made 0  
int turnOffK(int n, int k)  
{  
    // k must be greater than 0  
    if (k <= 0) return n;  
  
    // Do & of n with a number with all set bits except  
    // the k'th bit  
    return (n & ~(1 << (k - 1)));  
}  
  
// Driver program to test above function  
int main()  
{  
    int n = 15;  
    int k = 4;  
    cout << turnOffK(n, k);  
    return 0;  
}
```

Output:

```
7
```

Exercise: Write a function `turnOnK()` that turns the `k`'th bit on.

This article is contributed by **Rahul Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner Company Wise Coding Practice

Bit Magic

Check if binary representation of a number is palindrome

Given an integer '`x`', write a C function that returns true if binary representation of `x` is palindrome else return false.

For example a numbers with binary representation as 10..01 is palindrome and number with binary representation as 10..00 is not palindrome.

The idea is similar to [checking a string is palindrome or not](#). We start from leftmost and rightmost bits and compare bits one by one. If we find a mismatch, then return false.

We strongly recommend that you click here and practice it, before moving on to the solution.

Algorithm:

- ```
isPalindrome(x)
1) Find number of bits in x using sizeof() operator.
2) Initialize left and right positions as 1 and n respectively.
3) Do following while left 'l' is smaller than right 'r'.
.....a) If bit at position 'l' is not same as bit at position 'r', then return false.
.....b) Increment 'l' and decrement 'r', i.e., do l++ and r--.
4) If we reach here, it means we didn't find a mismatching bit.
```

To find the bit at a given position, we can use the idea similar to [this post](#). The expression "`x & (1 ^)`" gives us non-zero value if bit at `k`'th position from right is set and gives a zero value if if `k`'th bit is

**not set.**

Following is C++ implementation of the above algorithm.

```
#include<iostream>
using namespace std;

// This function returns true if k'th bit in x is set (or 1).
// For example if x (0010) is 2 and k is 2, then it returns true
bool isKthBitSet(unsigned int x, unsigned int k)
{
 return (x & (1 << (k-1)))? true: false;
}

// This function returns true if binary representation of x is
// palindrome. For example (1000...001) is palindrome
bool isPalindrome(unsigned int x)
{
 int l = 1; // Initialize left position
 int r = sizeof(unsigned int)*8; // initialize right position

 // One by one compare bits
 while (l < r)
 {
 if (isKthBitSet(x, l) != isKthBitSet(x, r))
 return false;
 l++; r--;
 }
 return true;
}

// Driver program to test above function
int main()
{
 unsigned int x = 1<<15 + 1<<16;
 cout << isPalindrome(x) << endl;
 x = 1<<31 + 1;
 cout << isPalindrome(x) << endl;
 return 0;
}
```

Output:

```
1
1
```

This article is contributed by **Saurabh Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner Company Wise Coding Practice

Bit Magic  
binary-representation

## Graph and its representations

Graph is a data structure that consists of following two components:

1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form  $(u, v)$  called as edge. The pair is ordered because  $(u, v)$  is not same as  $(v, u)$  in case of directed graph(di-graph). The pair of form  $(u, v)$  indicates that there is an edge from vertex  $u$  to vertex  $v$ . The edges may contain weight/value/cost.

Graphs are used to represent many real life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedin, facebook. For example, in facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender and locale. See [this](#) for more applications of graph.

Following is an example undirected graph with 5 vertices.



Following two are the most commonly used representations of graph.

1. Adjacency Matrix
2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

### Adjacency Matrix:

Adjacency Matrix is a 2D array of size  $V \times V$  where  $V$  is the number of vertices in a graph. Let the 2D array be  $\text{adj}[][],$  a slot  $\text{adj}[i][j] = 1$  indicates that there is an edge from vertex  $i$  to vertex  $j.$  Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If  $\text{adj}[i][j] = w,$  then there is an edge from vertex  $i$  to vertex  $j$  with weight  $w.$

The adjacency matrix for the above example graph is:



Adjacency Matrix  
Representation of the above graph

**Pros:** Representation is easier to implement and follow. Removing an edge takes  $O(1)$  time. Queries like whether there is an edge from vertex ' $u$ ' to vertex ' $v$ ' are efficient and can be done  $O(1).$

**Cons:** Consumes more space  $O(V^2)$ . Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is  $O(V^2)$  time.

#### Adjacency List:

An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be  $\text{array}[]$ . An entry  $\text{array}[i]$  represents the linked list of vertices adjacent to the  $i$ th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.

#### Adjacency List Representation of Graph



Adjacency List Representation of the above Graph

Below is C code for adjacency list representation of an undirected graph:

```
// A C Program to demonstrate adjacency list representation of graphs

#include <stdio.h>
#include <stdlib.h>

// A structure to represent an adjacency list node
struct AdjListNode
{
 int dest;
 struct AdjListNode* next;
};

// A structure to represent an adjacency list
struct AdjList
{
 struct AdjListNode *head; // pointer to head node of list
};

// A structure to represent a graph. A graph is an array of adjacency lists.
// Size of array will be V (number of vertices in graph)
struct Graph
{
 int V;
 struct AdjList* array;
};

// A utility function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest)
{
 struct AdjListNode* newNode =
 (struct AdjListNode*) malloc(sizeof(struct AdjListNode));
 newNode->dest = dest;
 newNode->next = NULL;
 return newNode;
}

// A utility function that creates a graph of V vertices
struct Graph* createGraph(int V)
{
 struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
 graph->V = V;

 // Create an array of adjacency lists. Size of array will be V
 graph->array = (struct AdjList*) malloc(V * sizeof(struct AdjList));

 // Initialize each adjacency list as empty by making head as NULL
 int i;
 for (i = 0; i < V; ++i)
 graph->array[i].head = NULL;

 return graph;
}

// Adds an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest)
{
 // Add an edge from src to dest. A new node is added to the adjacency
 // list of src. The node is added at the beginning
 struct AdjListNode* newNode = newAdjListNode(dest);
 newNode->next = graph->array[src].head;
 graph->array[src].head = newNode;

 // Since graph is undirected, add an edge from dest to src also
 newNode = newAdjListNode(src);
 newNode->next = graph->array[dest].head;
 graph->array[dest].head = newNode;
}

// A utility function to print the adjacency list representation of graph
void printGraph(struct Graph* graph)
{
 int v;
 for (v = 0; v < graph->V; ++v)
 {
 struct AdjListNode* pCrawl = graph->array[v].head;
 printf("\n Adjacency list of vertex %d\n head ", v);
 while (pCrawl)
 {
 printf("-> %d", pCrawl->dest);
 pCrawl = pCrawl->next;
 }
 printf("\n");
 }
}

// Driver program to test above functions
```

```

int main()
{
 // create the graph given in above figure
 int V = 5;
 struct Graph* graph = createGraph(V);
 addEdge(graph, 0, 1);
 addEdge(graph, 0, 4);
 addEdge(graph, 1, 2);
 addEdge(graph, 1, 3);
 addEdge(graph, 1, 4);
 addEdge(graph, 2, 3);
 addEdge(graph, 3, 4);

 // print the adjacency list representation of the above graph
 printGraph(graph);

 return 0;
}

```

Output:

```
Adjacency list of vertex 0
head -> 4-> 1
```

```
Adjacency list of vertex 1
head -> 4-> 3-> 2-> 0
```

```
Adjacency list of vertex 2
head -> 3-> 1
```

```
Adjacency list of vertex 3
head -> 4-> 2-> 1
```

```
Adjacency list of vertex 4
head -> 3-> 1-> 0
```

**Pros:** Saves space  $O(|V|+|E|)$ . In the worst case, there can be  $C(V, 2)$  number of edges in a graph thus consuming  $O(V^2)$  space. Adding a vertex is easier.

**Cons:** Queries like whether there is an edge from vertex  $u$  to vertex  $v$  are not efficient and can be done  $O(V)$ .

Reference:

[http://en.wikipedia.org/wiki/Graph\\_%28abstract\\_data\\_type%29](http://en.wikipedia.org/wiki/Graph_%28abstract_data_type%29)

Related Post:

[Graph representation using STL for competitive programming | Set 1 \(DFS of Unweighted and Undirected\)](#)

[Graph implementation using STL for competitive programming | Set 2 \(Weighted graph\)](#)

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner Company Wise Coding Practice

Graph  
Graph

## Breadth First Traversal or BFS for a Graph

Breadth First Traversal ([or Search](#)) for a graph is similar to Breadth First Traversal of a tree (See method 2 of [this post](#)). The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.



Following are C++ and Java implementations of simple Breadth First Traversal from a given source.

The C++ implementation uses adjacency list representation of graphs. STL's list container is used to store lists of adjacent nodes and queue of nodes needed for BFS traversal.

### C++

```
// Program to print BFS traversal from a given source vertex. BFS(int s)
```

```

// traverses vertices reachable from s.
#include<iostream>
#include <list>

using namespace std;

// This class represents a directed graph using adjacency list representation
class Graph
{
 int V; // No. of vertices
 list<int> *adj; // Pointer to an array containing adjacency lists
public:
 Graph(int V); // Constructor
 void addEdge(int v, int w); // function to add an edge to graph
 void BFS(int s); // prints BFS traversal from a given source s
};

Graph::Graph(int V)
{
 this->V = V;
 adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
 adj[v].push_back(w); // Add w to v's list.
}

void Graph::BFS(int s)
{
 // Mark all the vertices as not visited
 bool *visited = new bool[V];
 for(int i = 0; i < V; i++)
 visited[i] = false;

 // Create a queue for BFS
 list<int> queue;

 // Mark the current node as visited and enqueue it
 visited[s] = true;
 queue.push_back(s);

 // 'i' will be used to get all adjacent vertices of a vertex
 list<int>::iterator i;

 while(!queue.empty())
 {
 // Dequeue a vertex from queue and print it
 s = queue.front();
 cout << s << " ";
 queue.pop_front();

 // Get all adjacent vertices of the dequeued vertex s
 // If a adjacent has not been visited, then mark it visited
 // and enqueue it
 for(i = adj[s].begin(); i != adj[s].end(); ++i)
 {
 if(!visited[*i])
 {
 visited[*i] = true;
 queue.push_back(*i);
 }
 }
 }
}

// Driver program to test methods of graph class
int main()
{
 // Create a graph given in the above diagram
 Graph g(4);
 g.addEdge(0, 1);
 g.addEdge(0, 2);
 g.addEdge(1, 2);
 g.addEdge(2, 0);
 g.addEdge(2, 3);
 g.addEdge(3, 3);

 cout << "Following is Breadth First Traversal "
 << "(starting from vertex 2) \n";
 g.BFS(2);

 return 0;
}

```

## Java

```

// Java program to print BFS traversal from a given source vertex.
// BFS(int s) traverses vertices reachable from s.
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
 private int V; // No. of vertices
 private LinkedList<Integer> adj[]; //Adjacency Lists

 // Constructor
 Graph(int v)
 {
 V = v;
 adj = new LinkedList[v];

```

```

 for (int i=0; i<v; ++i)
 adj[i] = new LinkedList();
 }

 // Function to add an edge into the graph
 void addEdge(int v,int w)
 {
 adj[v].add(w);
 }

 // prints BFS traversal from a given source s
 void BFS(int s)
 {
 // Mark all the vertices as not visited(By default
 // set as false)
 boolean visited[] = new boolean[V];

 // Create a queue for BFS
 LinkedList<Integer> queue = new LinkedList<Integer>();

 // Mark the current node as visited and enqueue it
 visited[s]=true;
 queue.add(s);

 while (queue.size() != 0)
 {
 // Dequeue a vertex from queue and print it
 s = queue.poll();
 System.out.print(s+" ");

 // Get all adjacent vertices of the dequeued vertex s
 // If a adjacent has not been visited, then mark it
 // visited and enqueue it
 Iterator<Integer> i = adj[s].listIterator();
 while (i.hasNext())
 {
 int n = i.next();
 if (!visited[n])
 {
 visited[n] = true;
 queue.add(n);
 }
 }
 }
 }

 // Driver method to
 public static void main(String args[])
 {
 Graph g = new Graph(4);

 g.addEdge(0, 1);
 g.addEdge(0, 2);
 g.addEdge(1, 2);
 g.addEdge(2, 0);
 g.addEdge(2, 3);
 g.addEdge(3, 3);

 System.out.println("Following is Breadth First Traversal " +
 "(starting from vertex 2)");

 g.BFS(2);
 }
}
// This code is contributed by Aakash Hasija

```

## Python

```

Program to print BFS traversal from a given source
vertex. BFS(int s) traverses vertices reachable
from s.
from collections import defaultdict

This class represents a directed graph using adjacency
list representation
class Graph:

 # Constructor
 def __init__(self):

 # default dictionary to store graph
 self.graph = defaultdict(list)

 # function to add an edge to graph
 def addEdge(self,u,v):
 self.graph[u].append(v)

 # Function to print a BFS of graph
 def BFS(self, s):

 # Mark all the vertices as not visited
 visited = [False]*len(self.graph)

 # Create a queue for BFS
 queue = []

 # Mark the source node as visited and enqueue it
 queue.append(s)
 visited[s] = True

 while queue:

 # Dequeue a vertex from queue and print it
 s = queue.pop(0)

```

```

prints.

Get all adjacent vertices of the dequeued
vertex s. If a adjacent has not been visited,
then mark it visited and enqueue it
for i in self.graph[s]:
 if visited[i] == False:
 queue.append(i)
 visited[i] = True

Driver code
Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print "Following is Breadth First Traversal (starting from vertex 2)"
g.BFS(2)

This code is contributed by Neelam Yadav

```

Output:

```

Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1

```

Note that the above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex (example Disconnected graph). To print all the vertices, we can modify the BFS function to do traversal starting from all nodes one by one (Like the [DFS modified version](#)).

Time Complexity:  $O(V+E)$  where V is number of vertices in the graph and E is number of edges in the graph.

You may like to see below also :

- [Depth First Traversal](#)
- [Applications of Breadth First Traversal](#)
- [Applications of Depth First Search](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner Company Wise Coding Practice

```

Graph
Queue
BFS
Stack-Queue

```

## Depth First Traversal or DFS for a Graph

[Depth First Traversal \(or Search\)](#) for a graph is similar to [Depth First Traversal of a tree](#). The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Depth First Traversal of the following graph is 2, 0, 1, 3.



See this post for all applications of Depth First Traversal.

Following are implementations of simple Depth First Traversal. The C++ implementation uses [adjacency list representation](#) of graphs. STL's list container is used to store lists of adjacent nodes.

### C++

```

// C++ program to print DFS traversal from a given vertex in a given graph
#include<iostream>
#include<list>

using namespace std;

```

```

// Graph class represents a directed graph using adjacency list representation
class Graph
{
 int V; // No. of vertices
 list<int> *adj; // Pointer to an array containing adjacency lists
 void DFSUtil(int v, bool visited[]); // A function used by DFS
public:
 Graph(int V); // Constructor
 void addEdge(int v, int w); // function to add an edge to graph
 void DFS(int v); // DFS traversal of the vertices reachable from v
};

Graph::Graph(int V)
{
 this->V = V;
 adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
 adj[v].push_back(w); // Add w to v's list.
}

void Graph::DFSUtil(int v, bool visited[])
{
 // Mark the current node as visited and print it
 visited[v] = true;
 cout << v << " ";

 // Recur for all the vertices adjacent to this vertex
 list<int>::iterator i;
 for (i = adj[v].begin(); i != adj[v].end(); ++i)
 if (!visited[*i])
 DFSUtil(*i, visited);
}

// DFS traversal of the vertices reachable from v.
// It uses recursive DFSUtil()
void Graph::DFS(int v)
{
 // Mark all the vertices as not visited
 bool *visited = new bool[V];
 for (int i = 0; i < V; i++)
 visited[i] = false;

 // Call the recursive helper function to print DFS traversal
 DFSUtil(v, visited);
}

int main()
{
 // Create a graph given in the above diagram
 Graph g(4);
 g.addEdge(0, 1);
 g.addEdge(0, 2);
 g.addEdge(1, 2);
 g.addEdge(2, 0);
 g.addEdge(2, 3);
 g.addEdge(3, 3);

 cout << "Following is Depth First Traversal (starting from vertex 2) \n";
 g.DFS(2);

 return 0;
}

```

## Java

```

// Java program to print DFS traversal from a given given graph
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
 private int V; // No. of vertices

 // Array of lists for Adjacency List Representation
 private LinkedList<Integer> adj[];

 // Constructor
 Graph(int v)
 {
 V = v;
 adj = new LinkedList[v];
 for (int i=0; i<v; ++i)
 adj[i] = new LinkedList();
 }

 //Function to add an edge into the graph
 void addEdge(int v, int w)
 {
 adj[v].add(w); // Add w to v's list.
 }

 // A function used by DFS
 void DFSUtil(int v,boolean visited[])
 {
 // Mark the current node as visited and print it
 visited[v] = true;
 System.out.print(v+" ");

```

```

// Recur for all the vertices adjacent to this vertex
Iterator<Integer> i = adj[v].listIterator();
while (i.hasNext())
{
 int n = i.next();
 if (!visited[n])
 DFSUtil(n, visited);
}
}

// The function to do DFS traversal. It uses recursive DFSUtil()
void DFS(int v)
{
 // Mark all the vertices as not visited(set as
 // false by default in java)
 boolean visited[] = new boolean[V];

 // Call the recursive helper function to print DFS traversal
 DFSUtil(v, visited);
}

public static void main(String args[])
{
 Graph g = new Graph(4);

 g.addEdge(0, 1);
 g.addEdge(0, 2);
 g.addEdge(1, 2);
 g.addEdge(2, 0);
 g.addEdge(2, 3);
 g.addEdge(3, 3);

 System.out.println("Following is Depth First Traversal "+
 "(starting from vertex 2)");

 g.DFS(2);
}
}

// This code is contributed by Aakash Hasija

```

## Python

```

Python program to print DFS traversal from a
given given graph
from collections import defaultdict

This class represents a directed graph using
adjacency list representation
class Graph:

 # Constructor
 def __init__(self):
 # default dictionary to store graph
 self.graph = defaultdict(list)

 # function to add an edge to graph
 def addEdge(self,u,v):
 self.graph[u].append(v)

 # A function used by DFS
 def DFSUtil(self,v,visited):

 # Mark the current node as visited and print it
 visited[v]= True
 print v,

 # Recur for all the vertices adjacent to this vertex
 for i in self.graph[v]:
 if visited[i]== False:
 self.DFSUtil(i, visited)

 # The function to do DFS traversal. It uses
 # recursive DFSUtil()
 def DFS(self,v):

 # Mark all the vertices as not visited
 visited = [False]*(len(self.graph))

 # Call the recursive helper function to print
 # DFS traversal
 self.DFSUtil(v,visited)

 # Driver code
 # Create a graph given in the above diagram
 g = Graph()
 g.addEdge(0, 1)
 g.addEdge(0, 2)
 g.addEdge(1, 2)
 g.addEdge(2, 0)
 g.addEdge(2, 3)
 g.addEdge(3, 3)

 print "Following is DFS from (starting from vertex 2)"
 g.DFS(2)

This code is contributed by Neelam Yadav

```

Output:

```

Following is Depth First Traversal (starting from vertex 2)
2 0 1 3

```

Note that the above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex (example Disconnected graph). To do complete DFS traversal of such graphs, we must call DFSUtil() for every vertex. Also, before calling DFSUtil(), we should check if it is already printed by some other call of DFSUtil(). Following implementation does the complete graph traversal even if the nodes are unreachable. The differences from the above code are highlighted in the below code.

## C++

```
// C++ program to print DFS traversal for a given given graph
#include<iostream>
#include <list>
using namespace std;

class Graph
{
 int V; // No. of vertices
 list<int> *adj; // Pointer to an array containing adjacency lists
 void DFSUtil(int v, bool visited[]); // A function used by DFS
public:
 Graph(int V); // Constructor
 void addEdge(int v, int w); // function to add an edge to graph
 void DFS(); // prints DFS traversal of the complete graph
};

Graph::Graph(int V)
{
 this->V = V;
 adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
 adj[v].push_back(w); // Add w to v's list.
}

void Graph::DFSUtil(int v, bool visited[])
{
 // Mark the current node as visited and print it
 visited[v] = true;
 cout << v << " ";

 // Recur for all the vertices adjacent to this vertex
 list<int>::iterator i;
 for(i = adj[v].begin(); i != adj[v].end(); ++i)
 if(visited[*i])
 DFSUtil(*i, visited);
}

// The function to do DFS traversal. It uses recursive DFSUtil()
void Graph::DFS()
{
 // Mark all the vertices as not visited
 bool *visited = new bool[V];
 for (int i = 0; i < V; i++)
 visited[i] = false;

 // Call the recursive helper function to print DFS traversal
 // starting from all vertices one by one
 for (int i = 0; i < V; i++)
 if (visited[i] == false)
 DFSUtil(i, visited);
}

int main()
{
 // Create a graph given in the above diagram
 Graph g(4);
 g.addEdge(0, 1);
 g.addEdge(0, 2);
 g.addEdge(1, 2);
 g.addEdge(2, 0);
 g.addEdge(2, 3);
 g.addEdge(3, 3);

 cout << "Following is Depth First Traversal\n";
 g.DFS();

 return 0;
}
```

## Java

```
// Java program to print DFS traversal from a given given graph
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
 private int V; // No. of vertices

 // Array of lists for Adjacency List Representation
 private LinkedList<Integer> adj[];

 // Constructor
 Graph(int v)
 {
 V = v;
 adj = new LinkedList[v];
 for (int i=0; i<v; ++i)
 adj[i] = new LinkedList();
 }

 // Function to print DFS traversal
 void DFSUtil(int v, boolean visited[])
 {
 // Mark the current node as visited and print it
 visited[v] = true;
 System.out.print(v + " ");

 // Recur for all the vertices adjacent to this vertex
 Iterator i = adj[v].iterator();
 while (i.hasNext())
 if (!visited[(int)i.next()])
 DFSUtil((int)i.next(), visited);
 }

 // The function to do DFS traversal. It uses recursive DFSUtil()
 void DFS()
 {
 // Mark all the vertices as not visited
 boolean visited[] = new boolean[V];
 for (int i = 0; i < V; i++)
 visited[i] = false;

 // Call the recursive helper function to print DFS traversal
 // starting from all vertices one by one
 for (int i = 0; i < V; i++)
 if (visited[i] == false)
 DFSUtil(i, visited);
 }
}
```

```

//Function to add an edge into the graph
void addEdge(int v, int w)
{
 adj[v].add(w); // Add w to v's list.
}

// A function used by DFS
void DFSUtil(int v, boolean visited[])
{
 // Mark the current node as visited and print it
 visited[v] = true;
 System.out.print(v+" ");

 // Recur for all the vertices adjacent to this vertex
 Iterator<Integer> i = adj[v].listIterator();
 while (i.hasNext())
 {
 int n = i.next();
 if (!visited[n])
 DFSUtil(n,visited);
 }
}

// The function to do DFS traversal. It uses recursive DFSUtil()
void DFS()
{
 // Mark all the vertices as not visited(set as
 // false by default in java)
 boolean visited[] = new boolean[V];

 // Call the recursive helper function to print DFS traversal
 // starting from all vertices one by one
 for (int i=0; i<V; ++i)
 if (visited[i] == false)
 DFSUtil(i, visited);
}

public static void main(String args[])
{
 Graph g = new Graph(4);

 g.addEdge(0, 1);
 g.addEdge(0, 2);
 g.addEdge(1, 2);
 g.addEdge(2, 0);
 g.addEdge(2, 3);
 g.addEdge(3, 3);

 System.out.println("Following is Depth First Traversal");

 g.DFS();
}
}

// This code is contributed by Aakash Hasija

```

## Python

```

Python program to print DFS traversal for complete graph
from collections import defaultdict

This class represents a directed graph using adjacency
list representation
class Graph:

 # Constructor
 def __init__(self):

 # default dictionary to store graph
 self.graph = defaultdict(list)

 # function to add an edge to graph
 def addEdge(self,u,v):
 self.graph[u].append(v)

 # A function used by DFS
 def DFSUtil(self, v, visited):

 # Mark the current node as visited and print it
 visited[v]= True
 print v,

 # Recur for all the vertices adjacent to
 # this vertex
 for i in self.graph[v]:
 if visited[i]== False:
 self.DFSUtil(i, visited)

 # The function to do DFS traversal. It uses
 # recursive DFSUtil()
 def DFS(self):

 V = len(self.graph) #total vertices

 # Mark all the vertices as not visited
 visited =[False]*(V)

 # Call the recursive helper function to print
 # DFS traversal starting from all vertices one
 # by one
 for i in range(V):
 if visited[i] == False:
 self.DFSUtil(i, visited)

```

```

Driver code
Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print "Following is Depth First Traversal"
g.DFS()

This code is contributed by Neelam Yadav

```

Output:

```

Following is Depth First Traversal
0 1 2 3

```

Time Complexity:  $O(V+E)$  where V is number of vertices in the graph and E is number of edges in the graph.

#### Breadth First Traversal for a Graph

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

### GATE CS Corner Company Wise Coding Practice

[Graph](#)  
[DFS](#)

## Applications of Depth First Search

Depth-first search (DFS) is an algorithm (or technique) for traversing a graph.

Following are the problems that use DFS as a building block.

**1) For an unweighted graph, DFS traversal of the graph produces the minimum spanning tree and all pair shortest path tree.**

#### 2) Detecting cycle in a graph

A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges. (See [this](#) for details)

#### 3) Path Finding

We can specialize the DFS algorithm to find a path between two given vertices u and z.

i) Call DFS(G, u) with u as the start vertex.

ii) Use a stack S to keep track of the path between the start vertex and the current vertex.

iii) As soon as destination vertex z is encountered, return the path as the contents of the stack

See [this](#) for details.

#### 4) Topological Sorting

Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers [2].

#### 5) To test if a graph is bipartite

We can augment either BFS or DFS when we first discover a new vertex, color it opposite its parents, and for each other edge, check it doesn't link two vertices of the same color. The first vertex in any connected component can be red or black! See [this](#) for details.

**6) Finding Strongly Connected Components of a graph** A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. (See [this](#) for DFS based algo for finding Strongly Connected Components)

**7) Solving puzzles with only one solution**, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

Sources:

<http://www8.cs.umu.se/kurser/TDBAfl/VT06/algorithms/LEC/LECTUR16/NODE16.HTM>  
[http://en.wikipedia.org/wiki/Depth-first\\_search](http://en.wikipedia.org/wiki/Depth-first_search)  
<http://www.personal.kent.edu/~muhamma/Algorithms/MyAlgorithms/GraphAlgor/depthSearch.htm>  
<http://ww3.algorithmdesign.net/handouts/DFS.pdf>

## GATE CS Corner Company Wise Coding Practice

Graph  
DFS

### Detect Cycle in a Directed Graph

Given a directed graph, check whether the graph contains a cycle or not. Your function should return true if the given graph contains at least one cycle, else return false. For example, the following graph contains three cycles  $0 \rightarrow 2 \rightarrow 0$ ,  $0 \rightarrow 1 \rightarrow 2 \rightarrow 0$  and  $3 \rightarrow 3$ , so your function must return true.

Depth First Traversal can be used to detect cycle in a Graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back edge present in the graph. A back edge is an edge that is from a node to itself (selfloop) or one of its ancestor in the tree produced by DFS. In the following graph, there are 3 back edges, marked with cross sign. We can observe that these 3 back edges indicate 3 cycles present in the graph.



For a disconnected graph, we get the DFS forest as output. To detect cycle, we can check for cycle in individual trees by checking back edges.

To detect a back edge, we can keep track of vertices currently in recursion stack of function for DFS traversal. If we reach a vertex that is already in the recursion stack, then there is a cycle in the tree. The edge that connects current vertex to the vertex in the recursion stack is back edge. We have used `recStack[]` array to keep track of vertices in the recursion stack.

```
// A C++ Program to detect cycle in a graph
#include<iostream>
#include <list>
#include <limits.h>

using namespace std;

class Graph
{
 int V; // No. of vertices
 list<int*> *adj; // Pointer to an array containing adjacency lists
 bool isCyclicUtil(int v, bool visited[], bool *rs); // used by isCyclic()
public:
 Graph(int V); // Constructor
 void addEdge(int v, int w); // to add an edge to graph
 bool isCyclic(); // returns true if there is a cycle in this graph
};

Graph::Graph(int V)
{
 this->V = V;
 adj = new list<int*>[V];
}

void Graph::addEdge(int v, int w)
{
 adj[v].push_back(w); // Add w to v's list.
}

// This function is a variation of DFSUtil() in http://www.geeksforgeeks.org/archives/18212
bool Graph::isCyclicUtil(int v, bool visited[], bool *recStack)
{
 if(visited[v] == false)
 {
 // Mark the current node as visited and part of recursion stack
 visited[v] = true;
 recStack[v] = true;

 // Recur for all the vertices adjacent to this vertex
 list<int*>::iterator i;
 for(i = adj[v].begin(); i != adj[v].end(); ++i)
 {
 if (!visited[*i] && isCyclicUtil(*i, visited, recStack))
 return true;
 else if (recStack[*i])
 return true;
 }
 }
 return false;
}
```

```

 return true;
 }

}

recStack[v] = false; // remove the vertex from recursion stack
return false;
}

// Returns true if the graph contains a cycle, else false.
// This function is a variation of DFS() in http://www.geeksforgeeks.org/archives/18212
bool Graph::isCyclic()
{
 // Mark all the vertices as not visited and not part of recursion
 // stack
 bool *visited = new bool[V];
 bool *recStack = new bool[V];
 for(int i = 0; i < V; i++)
 {
 visited[i] = false;
 recStack[i] = false;
 }

 // Call the recursive helper function to detect cycle in different
 // DFS trees
 for(int i = 0; i < V; i++)
 if (isCyclicUtil(i, visited, recStack))
 return true;

 return false;
}

int main()
{
 // Create a graph given in the above diagram
 Graph g(4);
 g.addEdge(0, 1);
 g.addEdge(0, 2);
 g.addEdge(1, 2);
 g.addEdge(2, 0);
 g.addEdge(2, 3);
 g.addEdge(3, 3);

 if(g.isCyclic())
 cout << "Graph contains cycle";
 else
 cout << "Graph doesn't contain cycle";
 return 0;
}

```

Output:

Graph contains cycle

Time Complexity of this method is same as time complexity of [DFS traversal](#) which is  $O(V+E)$ .

In the below article, another  $O(V + E)$  method is discussed :

[Detect Cycle in a direct graph using colors](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner Company Wise Coding Practice

Graph  
DFS

## Union-Find Algorithm | Set 1 (Detect Cycle in an Undirected Graph)

A [disjoint-set data structure](#) is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. A [union-find algorithm](#) is an algorithm that performs two useful operations on such a data structure:

**Find:** Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

**Union:** Join two subsets into a single subset.

In this post, we will discuss an application of Disjoint Set Data Structure. The application is to check whether a given graph contains a cycle or not.

[Union-Find Algorithm](#) can be used to check whether an undirected graph contains cycle or not. Note that we have discussed an [algorithm to detect cycle](#). This is another method based on [Union-Find](#). This method assumes that graph doesn't contain any self-loops.

We can keep track of the subsets in a 1D array, let's call it `parent[]`.

Let us consider the following graph:

## cycle-in-graph



For each edge, make subsets using both the vertices of the edge. If both the vertices are in the same subset, a cycle is found.

Initially, all slots of parent array are initialized to -1 (means there is only one item in every subset).

```
0 1 2
-1 -1 -1
```

Now process all edges one by one.

*Edge 0-1:* Find the subsets in which vertices 0 and 1 are. Since they are in different subsets, we take the union of them. For taking the union, either make node 0 as parent of node 1 or vice-versa.

```
0 1 2
```

*Edge 1-2:* 1 is in subset 1 and 2 is in subset 2. So, take union.

```
0 1 2
Edge 0-2: 0 is in subset 2 and 2 is also in subset 2. Hence, including this edge forms a cycle.
```

How subset of 0 is same as 2?

```
0->1->2 // 1 is parent of 0 and 2 is parent of 1
```

Based on the above explanation, below are implementations:

## C/C++

```
// A union-find algorithm to detect cycle in a graph
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// a structure to represent an edge in graph
struct Edge
{
 int src, dest;
};

// a structure to represent a graph
struct Graph
{
 // V-> Number of vertices, E-> Number of edges
 int V, E;

 // graph is represented as an array of edges
 struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
 struct Graph* graph =
 (struct Graph*) malloc(sizeof(struct Graph));
 graph->V = V;
 graph->E = E;

 graph->edge =
 (struct Edge*) malloc(graph->E * sizeof(struct Edge));

 return graph;
}

// A utility function to find the subset of an element i
int find(int parent[], int i)
{
 if (parent[i] == -1)
 return i;
 return find(parent, parent[i]);
}

// A utility function to do union of two subsets
void Union(int parent[], int x, int y)
{
 int xset = find(parent, x);
 int yset = find(parent, y);
 parent[xset] = yset;
}

// The main function to check whether a given graph contains
// cycle or not
int isCycle(struct Graph* graph)
{
 // Allocate memory for creating V subsets
 int *parent = (int*) malloc(graph->V * sizeof(int));

 // Initialize all subsets as single element sets
 memset(parent, -1, sizeof(int) * graph->V);
```

```

// Iterate through all edges of graph, find subset of both
// vertices of every edge, if both subsets are same, then
// there is cycle in graph.
for(int i = 0; i < graph->E; ++i)
{
 int x = find(parent, graph->edge[i].src);
 int y = find(parent, graph->edge[i].dest);

 if (x == y)
 return 1;

 Union(parent, x, y);
}
return 0;
}

// Driver program to test above functions
int main()
{
/* Let us create following graph
 0
 |
 | \
 | \
 1----2 */
int V = 3, E = 3;
struct Graph* graph = createGraph(V, E);

// add edge 0-1
graph->edge[0].src = 0;
graph->edge[0].dest = 1;

// add edge 1-2
graph->edge[1].src = 1;
graph->edge[1].dest = 2;

// add edge 0-2
graph->edge[2].src = 0;
graph->edge[2].dest = 2;

if (isCycle(graph))
 printf("graph contains cycle");
else
 printf("graph doesn't contain cycle");

return 0;
}

```

## Java

```

// Java Program for union-find algorithm to detect cycle in a graph
import java.util.*;
import java.lang.*;
import java.io.*;

class Graph
{
 int V, E; // V-> no. of vertices & E->no.of edges
 Edge edge[]; // collection of all edges

 class Edge
 {
 int src, dest;
 };

 // Creates a graph with V vertices and E edges
 Graph(int v,int e)
 {
 V = v;
 E = e;
 edge = new Edge[E];
 for (int i=0; i<e; ++i)
 edge[i] = new Edge();
 }

 // A utility function to find the subset of an element i
 int find(int parent[], int i)
 {
 if (parent[i] == -1)
 return i;
 return find(parent, parent[i]);
 }

 // A utility function to do union of two subsets
 void Union(int parent[], int x, int y)
 {
 int xset = find(parent, x);
 int yset = find(parent, y);
 parent[xset] = yset;
 }

 // The main function to check whether a given graph
 // contains cycle or not
 int isCycle(Graph graph)
 {
 // Allocate memory for creating V subsets
 int parent[] = new int[graph.V];

 // Initialize all subsets as single element sets

```

```

for (int i=0; i<graph.V; ++i)
 parent[i]=-1;

// Iterate through all edges of graph, find subset of both
// vertices of every edge, if both subsets are same, then
// there is cycle in graph.
for (int i = 0; i < graph.E; ++i)
{
 int x = graph.find(parent, graph.edge[i].src);
 int y = graph.find(parent, graph.edge[i].dest);

 if (x == y)
 return 1;

 graph.Union(parent, x, y);
}
return 0;
}

// Driver Method
public static void main (String[] args)
{
 /* Let us create following graph
 0
 | \
 | \
 1----2 */
 int V = 3, E = 3;
 Graph graph = new Graph(V, E);

 // add edge 0-1
 graph.edge[0].src = 0;
 graph.edge[0].dest = 1;

 // add edge 1-2
 graph.edge[1].src = 1;
 graph.edge[1].dest = 2;

 // add edge 0-2
 graph.edge[2].src = 0;
 graph.edge[2].dest = 2;

 if (graph.isCycle(graph)==1)
 System.out.println("graph contains cycle");
 else
 System.out.println("graph doesn't contain cycle");
}
}

```

## Python

```

Python Program for union-find algorithm to detect cycle in a undirected graph
we have one egde for any two vertex i.e 1-2 is either 1-2 or 2-1 but not both

from collections import defaultdict

#This class represents a undirected graph using adjacency list representation
class Graph:

 def __init__(self,vertices):
 self.V= vertices #No. of vertices
 self.graph = defaultdict(list) # default dictionary to store graph

 # function to add an edge to graph
 def addEdge(self,u,v):
 self.graph[u].append(v)

 # A utility function to find the subset of an element i
 def find_parent(self, parent,i):
 if parent[i] == -1:
 return i
 if parent[i]!= -1:
 return self.find_parent(parent,parent[i])

 # A utility function to do union of two subsets
 def union(self, parent,x,y):
 x_set = self.find_parent(parent, x)
 y_set = self.find_parent(parent, y)
 parent[x_set]= y_set

 # The main function to check whether a given graph
 # contains cycle or not
 def isCyclic(self):

 # Allocate memory for creating V subsets and
 # Initialize all subsets as single element sets
 parent = [-1]*self.V

 # Iterate through all edges of graph, find subset of both
 # vertices of every edge, if both subsets are same, then
 # there is cycle in graph.
 for i in self.graph:
 for j in self.graph[i]:
 x = self.find_parent(parent, i)
 y = self.find_parent(parent, j)
 if x == y:

```

```

 return True
 self.union(parent,x,y)

Create a graph given in the above diagram
g = Graph(3)
g.addEdge(0, 1)
g.addEdge(1, 2)
g.addEdge(2, 0)

if g.isCyclic():
 print "Graph contains cycle"
else :
 print "Graph does not contain cycle"

#This code is contributed by Neelam Yadav

```

Output:

graph contains cycle

Note that the implementation of `union()` and `find()` is naive and takes  $O(n)$  time in worst case. These methods can be improved to  $O(\log n)$  using *Union by Rank* or *Height*. We will soon be discussing *Union by Rank* in

#### Related Articles :

[Union-Find Algorithm | Set 2 \(Union By Rank and Path Compression\)](#)

[Disjoint Set Data Structures \(Java Implementation\)](#)

[Greedy Algorithms | Set 2 \(Kruskal's Minimum Spanning Tree Algorithm\)](#)

[Job Sequencing Problem | Set 2 \(Using Disjoint Set\)](#)

This article is compiled by [Aashish Barnwal](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner

### Company Wise Coding Practice

Graph  
Graph  
union-find

## Detect cycle in an undirected graph

Given an undirected graph, how to check if there is a cycle in the graph? For example, the following graph has a cycle 1-0-2-1.

## cycleGraph



We have discussed [cycle detection for directed graph](#). We have also discussed a [union-find algorithm for cycle detection in undirected graphs](#). The time complexity of the union-find algorithm is  $O(E\log V)$ . Like directed graphs, we can use [DFS](#) to detect cycle in an undirected graph in  $O(V+E)$  time. We do a DFS traversal of the given graph. For every visited vertex 'v', if there is an adjacent 'u' such that u is already visited and u is not parent of v, then there is a cycle in graph. If we don't find such an adjacent for any vertex, we say that there is no cycle. The assumption of this approach is that there are no parallel edges between any two vertices.

### C++

```
// A C++ Program to detect cycle in an undirected graph
#include<iostream>
#include <list>
#include <limits.h>
using namespace std;

// Class for an undirected graph
class Graph
{
 int V; // No. of vertices
 list<int> *adj; // Pointer to an array containing adjacency lists
 bool isCyclicUtil(int v, bool visited[], int parent);
public:
 Graph(int V); // Constructor
 void addEdge(int v, int w); // to add an edge to graph
 bool isCyclic(); // returns true if there is a cycle
};

Graph::Graph(int V)
{
 this->V = V;
 adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
 adj[v].push_back(w); // Add w to v's list.
 adj[w].push_back(v); // Add v to w's list.
}

// A recursive function that uses visited[] and parent to detect
// cycle in subgraph reachable from vertex v.
bool Graph::isCyclicUtil(int v, bool visited[], int parent)
{
 // Mark the current node as visited
 visited[v] = true;

 // Recur for all the vertices adjacent to this vertex
 list<int>::iterator i;
 for (i = adj[v].begin(); i != adj[v].end(); ++i)
 {
 // If an adjacent is not visited, then recur for that adjacent
 if (!visited[*i])
 {
 if (isCyclicUtil(*i, visited, v))
 return true;
 }

 // If an adjacent is visited and not parent of current vertex,
 // then there is a cycle.
 else if (*i != parent)
 return true;
 }
 return false;
}

// Returns true if the graph contains a cycle, else false.
bool Graph::isCyclic()
{
 // Mark all the vertices as not visited and not part of recursion
 // stack
 bool *visited = new bool[V];
 for (int i = 0; i < V; i++)
 visited[i] = false;

 // Call the recursive helper function to detect cycle in different
 // DFS trees
 for (int u = 0; u < V; u++)
 if (!visited[u]) // Don't recur for u if it is already visited
 if (isCyclicUtil(u, visited, -1))
 return true;

 return false;
}

// Driver program to test above functions
int main()
{
 Graph g1(5);
 g1.addEdge(1, 0);
 g1.addEdge(0, 2);
 g1.addEdge(2, 0);
 g1.addEdge(0, 3);
 g1.addEdge(3, 4);
 g1.isCyclic() ? cout << "Graph contains cycle\n":
 cout << "Graph doesn't contain cycle\n";

 Graph g2(3);
}
```

```

g2.addEdge(0, 1);
g2.addEdge(1, 2);
g2.isCyclic() ? cout << "Graph contains cycle\n";
 cout << "Graph doesn't contain cycle\n";

return 0;
}

```

## Java

```

// A Java Program to detect cycle in an undirected graph
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
 private int V; // No. of vertices
 private LinkedList<Integer> adj[]; // Adjacency List Representation

 // Constructor
 Graph(int v) {
 V = v;
 adj = new LinkedList[v];
 for(int i=0; i<v; ++i)
 adj[i] = new LinkedList();
 }

 // Function to add an edge into the graph
 void addEdge(int v,int w) {
 adj[v].add(w);
 adj[w].add(v);
 }

 // A recursive function that uses visited[] and parent to detect
 // cycle in subgraph reachable from vertex v.
 Boolean isCyclicUtil(int v, Boolean visited[], int parent)
 {
 // Mark the current node as visited
 visited[v] = true;
 Integer i;

 // Recur for all the vertices adjacent to this vertex
 Iterator<Integer> it = adj[v].iterator();
 while (it.hasNext())
 {
 i = it.next();

 // If an adjacent is not visited, then recur for that
 // adjacent
 if (!visited[i])
 {
 if (isCyclicUtil(i, visited, v))
 return true;
 }
 }

 // If an adjacent is visited and not parent of current
 // vertex, then there is a cycle.
 else if (i != parent)
 return true;
 }

 return false;
}

// Returns true if the graph contains a cycle, else false.
Boolean isCyclic()
{
 // Mark all the vertices as not visited and not part of
 // recursion stack
 Boolean visited[] = new Boolean[V];
 for (int i = 0; i < V; i++)
 visited[i] = false;

 // Call the recursive helper function to detect cycle in
 // different DFS trees
 for (int u = 0; u < V; u++)
 if ((visited[u]) // Don't recur for u if already visited
 if (isCyclicUtil(u, visited, -1)))
 return true;

 return false;
}

// Driver method to test above methods
public static void main(String args[])
{
 // Create a graph given in the above diagram
 Graph g1 = new Graph(5);
 g1.addEdge(1, 0);
 g1.addEdge(0, 2);
 g1.addEdge(2, 0);
 g1.addEdge(0, 3);
 g1.addEdge(3, 4);
 if (g1.isCyclic())
 System.out.println("Graph contains cycle");
 else
 System.out.println("Graph doesn't contain cycle");

 Graph g2 = new Graph(3);
 g2.addEdge(0, 1);
 g2.addEdge(1, 2);
 if (g2.isCyclic())

```

```

 System.out.println("Graph contains cycle");
 else
 System.out.println("Graph doesn't contain cycle");
 }
}
// This code is contributed by Aakash Hasija

```

## Python

```

Python Program to detect cycle in an undirected graph

from collections import defaultdict

#This class represents a undirected graph using adjacency list representation
class Graph:

 def __init__(self,vertices):
 self.V= vertices #No. of vertices
 self.graph = defaultdict(list) # default dictionary to store graph

 # function to add an edge to graph
 def addEdge(self,v,w):
 self.graph[v].append(w) #Add w to v_s list
 self.graph[w].append(v) #Add v to w_s list

 # A recursive function that uses visited[] and parent to detect
 # cycle in subgraph reachable from vertex v.
 def isCyclicUtil(self,v,visited,parent):

 #Mark the current node as visited
 visited[v]= True

 #Recur for all the vertices adjacent to this vertex
 for i in self.graph[v]:
 # If the node is not visited then recurse on it
 if visited[i]==False :
 if(self.isCyclicUtil(i,visited,v)):
 return True
 # If an adjacent vertex is visited and not parent of current vertex,
 # then there is a cycle
 elif parent!=i:
 return True

 return False

 #Returns true if the graph contains a cycle, else false.
 def isCyclic(self):
 # Mark all the vertices as not visited
 visited =[False]*(self.V)
 # Call the recursive helper function to detect cycle in different
 # DFS trees
 for i in range(self.V):
 if visited[i] ==False: #Don't recur for u if it is already visited
 if(self.isCyclicUtil(i,visited,-1))== True:
 return True
 return False

 # Create a graph given in the above diagram
 g = Graph(5)
 g.addEdge(1, 0)
 g.addEdge(0, 2)
 g.addEdge(2, 0)
 g.addEdge(0, 3)
 g.addEdge(3, 4)

 if g.isCyclic():
 print "Graph contains cycle"
 else :
 print "Graph does not contain cycle "
 g1 = Graph(3)
 g1.addEdge(0,1)
 g1.addEdge(1,2)

 if g1.isCyclic():
 print "Graph contains cycle"
 else :
 print "Graph does not contain cycle "

#This code is contributed by Neelam Yadav

```

Output:

```

Graph contains cycle
Graph doesn't contain cycle

```

**Time Complexity:** The program does a simple DFS Traversal of graph and graph is represented using adjacency list. So the time complexity is  $O(V+E)$

**Exercise:** Can we use BFS to detect cycle in an undirected graph in O(V+E) time? What about directed graphs?

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner Company Wise Coding Practice

Graph  
BFS  
DFS

### Longest Path in a Directed Acyclic Graph

Given a Weighted Directed Acyclic Graph (DAG) and a source vertex s in it, find the longest distances from s to all other vertices in the given graph.

The longest path problem for a general graph is not as easy as the shortest path problem because the longest path problem doesn't have optimal substructure property. In fact, the Longest Path problem is NP-Hard for a general graph. However, the longest path problem has a linear time solution for directed acyclic graphs. The idea is similar to linear time solution for shortest path in a directed acyclic graph., we use Topological Sorting.

We initialize distances to all vertices as minus infinite and distance to source as 0, then we find a topological sorting of the graph. Topological Sorting of a graph represents a linear ordering of the graph (See below, figure (b) is a linear representation of figure (a) ). Once we have topological order (or linear representation), we one by one process all vertices in topological order. For every vertex being processed, we update distances of its adjacent using distance of current vertex.

Following figure shows step by step process of finding longest paths.

LongestPath

Following is complete algorithm for finding longest distances.

1) Initialize dist[] = {NINF, NINF, ...} and dist[s] = 0 where s is the source vertex. Here NINF means negative infinite.

2) Create a topological order of all vertices.

3) Do following for every vertex u in topological order.

.....Do following for every adjacent vertex v of u

.....if (dist[v] < dist[u] + weight(u, v))

.....dist[v] = dist[u] + weight(u, v)

Following is C++ implementation of the above algorithm.

```
// A C++ program to find single source longest distances in a DAG
#include <iostream>
#include <list>
#include <stack>
#include <limits.h>
#define NINF INT_MIN
using namespace std;
#include <assert.h>

// Graph is represented using adjacency list. Every node of adjacency list
// contains vertex number of the vertex to which edge connects. It also
// contains weight of the edge
class AdjListNode
{
 int v;
 int weight;
public:
 AdjListNode(int v, int weight)
 {
 this->v = v;
 this->weight = weight;
 }
};
```

```

public:
 AdjListNode(int _v, int _w) { v = _v; weight = _w;}
 int getV() { return v; }
 int getWeight() { return weight; }
};

// Class to represent a graph using adjacency list representation
class Graph
{
 int V; // No. of vertices

 // Pointer to an array containing adjacency lists
 list<AdjListNode> *adj;

 // A function used by longestPath
 void topologicalSortUtil(int v, bool visited[], stack<int> &Stack);
public:
 Graph(int V); // Constructor

 // function to add an edge to graph
 void addEdge(int u, int v, int weight);

 // Finds longest distances from given source vertex
 void longestPath(int s);
};

Graph::Graph(int V) // Constructor
{
 this->V = V;
 adj = new list<AdjListNode>[V];
}

void Graph::addEdge(int u, int v, int weight)
{
 AdjListNode node(v, weight);
 adj[u].push_back(node); // Add v to u's list
}

// A recursive function used by longestPath. See below link for details
// http://www.geeksforgeeks.org/topological-sorting/
void Graph::topologicalSortUtil(int v, bool visited[], stack<int> &Stack)
{
 // Mark the current node as visited
 visited[v] = true;

 // Recur for all the vertices adjacent to this vertex
 list<AdjListNode>::iterator i;
 for (i = adj[v].begin(); i != adj[v].end(); ++i)
 {
 AdjListNode node = *i;
 if (!visited[node.getV()])
 topologicalSortUtil(node.getV(), visited, Stack);
 }

 // Push current vertex to stack which stores topological sort
 Stack.push(v);
}

// The function to find longest distances from a given vertex. It uses
// recursive topologicalSortUtil() to get topological sorting.
void Graph::longestPath(int s)
{
 stack<int> Stack;
 int dist[V];

 // Mark all the vertices as not visited
 bool *visited = new bool[V];
 for (int i = 0; i < V; i++)
 visited[i] = false;

 // Call the recursive helper function to store Topological Sort
 // starting from all vertices one by one
 for (int i = 0; i < V; i++)
 if (visited[i] == false)
 topologicalSortUtil(i, visited, Stack);

 // Initialize distances to all vertices as infinite and distance
 // to source as 0
 for (int i = 0; i < V; i++)
 dist[i] = NINF;
 dist[s] = 0;

 // Process vertices in topological order
 while (Stack.empty() == false)
 {
 // Get the next vertex from topological order
 int u = Stack.top();
 Stack.pop();

 // Update distances of all adjacent vertices
 list<AdjListNode>::iterator i;
 if (dist[u] != NINF)
 {
 for (i = adj[u].begin(); i != adj[u].end(); ++i)
 if (dist[i->getV()] < dist[u] + i->getWeight())
 dist[i->getV()] = dist[u] + i->getWeight();
 }
 }

 // Print the calculated longest distances
 for (int i = 0; i < V; i++)
 (dist[i] == NINF)? cout << "INF": cout << dist[i] << " ";
}

// Driver program to test above functions
int main()
{

```

```

// Create a graph given in the above diagram. Here vertex numbers are
// 0, 1, 2, 3, 4, 5 with following mappings:
// 0=r, 1=s, 2=t, 3=x, 4=y, 5=z
Graph g(6);
g.addEdge(0, 1, 5);
g.addEdge(0, 2, 3);
g.addEdge(1, 3, 6);
g.addEdge(1, 2, 2);
g.addEdge(2, 4, 4);
g.addEdge(2, 5, 2);
g.addEdge(2, 3, 7);
g.addEdge(3, 5, 1);
g.addEdge(3, 4, -1);
g.addEdge(4, 5, -2);

int s = 1;
cout << "Following are longest distances from source vertex " << s << endl;
g.longestPath(s);

return 0;
}

```

Output:

```

Following are longest distances from source vertex 1
INF 0 2 9 8 10

```

**Time Complexity:** Time complexity of topological sorting is  $O(V+E)$ . After finding topological order, the algorithm processes all vertices and for every vertex, it runs a loop for all adjacent vertices. Total adjacent vertices in a graph is  $O(E)$ . So the inner loop runs  $O(V+E)$  times. Therefore, overall time complexity of this algorithm is  $O(V+E)$ .

**Exercise:** The above solution prints longest distances, extend the code to print paths also.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner Company Wise Coding Practice

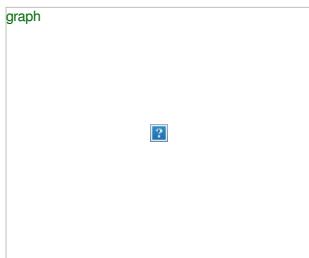
Graph  
shortest path  
Topological Sorting

---

## Topological Sorting

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge  $uv$ , vertex  $u$  comes before  $v$  in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is "5 4 2 3 1 0". There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is "4 5 2 3 1 0". The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).



### Topological Sorting vs Depth First Traversal (DFS):

In **DFS**, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex '5' should be printed before vertex '0', but unlike **DFS**, the vertex '4' should also be printed before vertex '0'. So Topological sorting is different from DFS. For example, a DFS of the above graph is "5 2 3 1 0 4", but it is not a topological sorting.

### Algorithm to find Topological Sorting:

We recommend to first see implementation of DFS [here](#). We can modify **DFS** to find Topological Sorting of a graph. In **DFS**, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don't print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.

Following are C++ and Java implementations of topological sorting. Please see the code for Depth First Traversal for a disconnected Graph and note the differences between the second code given there and the below code.

### C++

```

// A C++ program to print topological sorting of a DAG
#include<iostream>
#include <list>
#include <stack>
using namespace std;

// Class to represent a graph
class Graph
{
 int V; // No. of vertices

 // Pointer to an array containing adjacency listsList
 list<int> *adj;

 // A function used by topologicalSort()
 void topologicalSortUtil(int v, bool visited[], stack<int> &Stack);
public:
 Graph(int V); // Constructor

 // function to add an edge to graph
 void addEdge(int v, int w);

 // prints a Topological Sort of the complete graph
 void topologicalSort();
};


```

```

};

Graph::Graph(int V)
{
 this->V = V;
 adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
 adj[v].push_back(w); // Add w to v's list.
}

// A recursive function used by topologicalSort
void Graph::topologicalSortUtil(int v, bool visited[],
 stack<int> &Stack)
{
 // Mark the current node as visited.
 visited[v] = true;

 // Recur for all the vertices adjacent to this vertex
 list<int>::iterator i;
 for (i = adj[v].begin(); i != adj[v].end(); ++i)
 if (!visited[*i])
 topologicalSortUtil(*i, visited, Stack);

 // Push current vertex to stack which stores result
 Stack.push(v);
}

// The function to do Topological Sort. It uses recursive
// topologicalSortUtil()
void Graph::topologicalSort()
{
 stack<int> Stack;

 // Mark all the vertices as not visited
 bool *visited = new bool[V];
 for (int i = 0; i < V; i++)
 visited[i] = false;

 // Call the recursive helper function to store Topological
 // Sort starting from all vertices one by one
 for (int i = 0; i < V; i++)
 if (visited[i] == false)
 topologicalSortUtil(i, visited, Stack);

 // Print contents of stack
 while (Stack.empty() == false)
 {
 cout << Stack.top() << " ";
 Stack.pop();
 }
}

// Driver program to test above functions
int main()
{
 // Create a graph given in the above diagram
 Graph g(6);
 g.addEdge(5, 2);
 g.addEdge(5, 0);
 g.addEdge(4, 0);
 g.addEdge(4, 1);
 g.addEdge(2, 1);
 g.addEdge(3, 1);

 cout << "Following is a Topological Sort of the given graph \n";
 g.topologicalSort();

 return 0;
}

```

## Java

```

// A Java program to print topological sorting of a DAG
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
 private int V; // No. of vertices
 private LinkedList<Integer> adj[]; // Adjacency List

 // Constructor
 Graph(int v)
 {
 V = v;
 adj = new LinkedList[v];
 for (int i=0; i<v; ++i)
 adj[i] = new LinkedList();
 }

 // Function to add an edge into the graph
 void addEdge(int v,int w) { adj[v].add(w);}

 // A recursive function used by topologicalSort
 void topologicalSortUtil(int v, boolean visited[],
 Stack stack)
 {
 // Mark the current node as visited.
 visited[v] = true;

```

```

Integer i;

// Recur for all the vertices adjacent to this
// vertex
Iterator<Integer> it = adj[v].iterator();
while (it.hasNext())
{
 i = it.next();
 if (!visited[i])
 topologicalSortUtil(i, visited, stack);
}

// Push current vertex to stack which stores result
stack.push(new Integer(v));
}

// The function to do Topological Sort. It uses
// recursive topologicalSortUtil()
void topologicalSort()
{
 Stack stack = new Stack();

 // Mark all the vertices as not visited
 boolean visited[] = new boolean[V];
 for (int i = 0; i < V; i++)
 visited[i] = false;

 // Call the recursive helper function to store
 // Topological Sort starting from all vertices
 // one by one
 for (int i = 0; i < V; i++)
 if (visited[i] == false)
 topologicalSortUtil(i, visited, stack);

 // Print contents of stack
 while (stack.empty() == false)
 System.out.print(stack.pop() + " ");
}

// Driver method
public static void main(String args[])
{
 // Create a graph given in the above diagram
 Graph g = new Graph(6);
 g.addEdge(5, 2);
 g.addEdge(5, 0);
 g.addEdge(4, 0);
 g.addEdge(4, 1);
 g.addEdge(2, 3);
 g.addEdge(3, 1);

 System.out.println("Following is a Topological " +
 "sort of the given graph");
 g.topologicalSort();
}
}

// This code is contributed by Aakash Hasija

```

## Python

```

#Python program to print topological sorting of a DAG
from collections import defaultdict

#Class to represent a graph
class Graph:
 def __init__(self,vertices):
 self.graph = defaultdict(list) #dictionary containing adjacency List
 self.V = vertices #No. of vertices

 # function to add an edge to graph
 def addEdge(self,u,v):
 self.graph[u].append(v)

 # A recursive function used by topologicalSort
 def topologicalSortUtil(self,v,visited,stack):

 # Mark the current node as visited.
 visited[v] = True

 # Recur for all the vertices adjacent to this vertex
 for i in self.graph[v]:
 if visited[i] == False:
 self.topologicalSortUtil(i,visited,stack)

 # Push current vertex to stack which stores result
 stack.insert(0,v)

 # The function to do Topological Sort. It uses recursive
 # topologicalSortUtil()
 def topologicalSort(self):
 # Mark all the vertices as not visited
 visited = [False]*self.V
 stack =[]

 # Call the recursive helper function to store Topological
 # Sort starting from all vertices one by one
 for i in range(self.V):
 if visited[i] == False:
 self.topologicalSortUtil(i,visited,stack)

 # Print contents of stack
 print stack

g= Graph(6)

```

```

g.addEdge(5, 2);
g.addEdge(5, 0);
g.addEdge(4, 0);
g.addEdge(4, 1);
g.addEdge(2, 3);
g.addEdge(3, 1);

print "Following is a Topological Sort of the given graph"
g.topologicalSort()
#This code is contributed by Neelam Yadav

```

Output:

```

Following is a Topological Sort of the given graph
5 4 2 3 1 0

```

**Time Complexity:** The above algorithm is simply DFS with an extra stack. So time complexity is same as DFS which is  $O(V+E)$ .

#### Applications:

Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers [2].

#### Related Articles:

[Kahn's algorithm for Topological Sorting](#) : Another  $O(V + E)$  algorithm.

[All Topological Sorts of a Directed Acyclic Graph](#)

#### References:

<http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/topoSort.htm>  
[http://en.wikipedia.org/wiki/Topological\\_sorting](http://en.wikipedia.org/wiki/Topological_sorting)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner Company Wise Coding Practice

Graph  
DFS  
Topological Sorting

### Check whether a given graph is Bipartite or not

A [Bipartite Graph](#) is a graph whose vertices can be divided into two independent sets,  $U$  and  $V$  such that every edge  $(u, v)$  either connects a vertex from  $U$  to  $V$  or a vertex from  $V$  to  $U$ . In other words, for every edge  $(u, v)$ , either  $u$  belongs to  $U$  and  $v$  to  $V$ , or  $u$  belongs to  $V$  and  $v$  to  $U$ . We can also say that there is no edge that connects vertices of same set.

Bipartite1



A bipartite graph is possible if the graph coloring is possible using two colors such that vertices in a set are colored with the same color. Note that it is possible to color a cycle graph with even cycle using two colors. For example, see the following graph.

Bipartite2



It is not possible to color a cycle graph with odd cycle using two colors.

Bipartite3



#### Algorithm to check if a graph is Bipartite:

One approach is to check whether the graph is 2-colorable or not using [backtracking algorithm](#) in coloring problem.

Following is a simple algorithm to find out whether a given graph is Bipartite or not using Breadth First Search (BFS).

1. Assign RED color to the source vertex (putting into set  $U$ ).

2. Color all the neighbors with BLUE color (putting into set V).
3. Color all neighbor's neighbor with RED color (putting into set U).
4. This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where m = 2.
5. While assigning colors, if we find a neighbor which is colored with same color as current vertex, then the graph cannot be colored with 2 vertices (or graph is not Bipartite)

## C++

```
// C++ program to find out whether a given graph is Bipartite or not
#include <iostream>
#include <queue>
#define V 4
using namespace std;

// This function returns true if graph G[V][V] is Bipartite, else false
bool isBipartite(int G[][V], int src)
{
 // Create a color array to store colors assigned to all vertices. Vertex
 // number is used as index in this array. The value '-1' of colorArr[i]
 // is used to indicate that no color is assigned to vertex 'i'. The value
 // 1 is used to indicate first color is assigned and value 0 indicates
 // second color is assigned.
 int colorArr[V];
 for (int i = 0; i < V; ++i)
 colorArr[i] = -1;

 // Assign first color to source
 colorArr[src] = 1;

 // Create a queue (FIFO) of vertex numbers and enqueue source vertex
 // for BFS traversal
 queue<int> q;
 q.push(src);

 // Run while there are vertices in queue (Similar to BFS)
 while (!q.empty())
 {
 // Dequeue a vertex from queue (Refer http://goo.gl/35oz8)
 int u = q.front();
 q.pop();

 // Find all non-colored adjacent vertices
 for (int v = 0; v < V; ++v)
 {
 // An edge from u to v exists and destination v is not colored
 if (G[u][v] && colorArr[v] == -1)
 {
 // Assign alternate color to this adjacent v of u
 colorArr[v] = 1 - colorArr[u];
 q.push(v);
 }
 }

 // An edge from u to v exists and destination v is colored with
 // same color as u
 else if (G[u][v] && colorArr[v] == colorArr[u])
 return false;
 }

 // If we reach here, then all adjacent vertices can be colored with
 // alternate color
 return true;
}

// Driver program to test above function
int main()
{
 int G[][V] = {{0, 1, 0, 1},
 {1, 0, 1, 0},
 {0, 1, 0, 1},
 {1, 0, 1, 0}};
};

isBipartite(G, 0) ? cout << "Yes" : cout << "No";
return 0;
}
```

## Java

```
// Java program to find out whether a given graph is Bipartite or not
import java.util.*;
import java.lang.*;
import java.io.*;

class Bipartite
{
 final static int V = 4;// No. of Vertices

 // This function returns true if graph G[V][V] is Bipartite, else false
 boolean isBipartite(int G[][],int src)
 {
 // Create a color array to store colors assigned to all vertices.
 // Vertex number is used as index in this array. The value '-1'
 // of colorArr[i] is used to indicate that no color is assigned
 // to vertex 'i'. The value 1 is used to indicate first color
 // is assigned and value 0 indicates second color is assigned.
 int colorArr[] = new int[V];
 for (int i=0; i<V; ++i)
 colorArr[i] = -1;

 // Assign first color to source
 colorArr[src] = 1;

 // Create a queue (FIFO) of vertex numbers and enqueue source vertex
 // for BFS traversal
 Queue<Integer> q = new LinkedList<Integer>();
 q.add(src);

 // Run while there are vertices in queue (Similar to BFS)
 while (!q.isEmpty())
 {
 // Dequeue a vertex from queue (Refer http://goo.gl/35oz8)
 int u = q.remove();

 // Find all non-colored adjacent vertices
 for (int v = 0; v < V; ++v)
 {
 // An edge from u to v exists and destination v is not colored
 if (G[u][v] && colorArr[v] == -1)
 {
 // Assign alternate color to this adjacent v of u
 colorArr[v] = 1 - colorArr[u];
 q.add(v);
 }
 }

 // An edge from u to v exists and destination v is colored with
 // same color as u
 else if (G[u][v] && colorArr[v] == colorArr[u])
 return false;
 }

 // If we reach here, then all adjacent vertices can be colored with
 // alternate color
 return true;
 }
}
```

```

// Create a queue (FIFO) of vertex numbers and enqueue
// source vertex for BFS traversal
LinkedList<Integer>q = new LinkedList<Integer>();
q.add(src);

// Run while there are vertices in queue (Similar to BFS)
while (q.size() != 0)
{
 // Dequeue a vertex from queue
 int u = q.poll();

 // Find all non-colored adjacent vertices
 for (int v=0;v<V; ++v)
 {
 // An edge from u to v exists and destination v is
 // not colored
 if (G[u][v]==1 && colorArr[v]==-1)
 {
 // Assign alternate color to this adjacent v of u
 colorArr[v] = 1-colorArr[u];
 q.add(v);
 }

 // An edge from u to v exists and destination v is
 // colored with same color as u
 else if (G[u][v]==1 && colorArr[v]==colorArr[u])
 return false;
 }
}

// If we reach here, then all adjacent vertices can
// be colored with alternate color
return true;
}

// Driver program to test above function
public static void main (String[] args)
{
 int G[][] = {{0, 1, 0, 1},
 {1, 0, 1, 0},
 {0, 1, 0, 1},
 {1, 0, 1, 0}};
}

Bipartite b = new Bipartite();
if (b.isBipartite(G, 0))
 System.out.println("Yes");
else
 System.out.println("No");
}
}

// Contributed by Aakash Hasija

```

Output:

Yes

The above algorithm works only if the graph is strongly connected. In above code, we always start with source 0 and assume that vertices are visited from it. One important observation is a graph with no edges is also Bipartite. Note that the Bipartite condition says all edges should be from one set to another.

We can extend the above code to handle cases when a graph is not connected. The idea is repeatedly call above method for all not yet visited vertices.

```

// C++ program to find out whether a given graph is Bipartite or not.
// It works for disconnected graph also.
#include <bits/stdc++.h>
using namespace std;

const int V = 4;

// This function returns true if graph G[V][V] is Bipartite,
// else false
bool isBipartiteUtil(int G[][V], int src, int colorArr[])
{
 colorArr[src] = 1;

 // Create a queue (FIFO) of vertex numbers and enqueue
 // source vertex for BFS traversal
 queue <int> q;
 q.push(src);

 // Run while there are vertices in queue (Similar to BFS)
 while (!q.empty())
 {
 // Dequeue a vertex from queue (Refer http://goo.gl/35oz8)
 int u = q.front();
 q.pop();

 // Find all non-colored adjacent vertices
 for (int v = 0; v < V; ++v)
 {
 // An edge from u to v exists and
 // destination v is not colored
 if (G[u][v] && colorArr[v] == -1)
 {
 // Assign alternate color to this
 // adjacent v of u
 colorArr[v] = 1 - colorArr[u];
 q.push(v);
 }

 // An edge from u to v exists and destination
 // v is colored with same color as u
 else if (G[u][v] && colorArr[v] == colorArr[u])
 return false;
 }
 }
}

// Contributed by Aakash Hasija

```

```

}

// If we reach here, then all adjacent vertices can
// be colored with alternate color
return true;
}

// Returns true if G[][], is Bipartite, else false
bool isBipartite(int G[][V])
{
 // Create a color array to store colors assigned to all
 // vertices. Vertex/ number is used as index in this
 // array. The value '-1' of colorArr[i] is used to
 // indicate that no color is assigned to vertex 'i'.
 // The value 1 is used to indicate first color is
 // assigned and value 0 indicates second color is
 // assigned.
 int colorArr[V];
 for (int i = 0; i < V; ++i)
 colorArr[i] = -1;

 // This code is to handle disconnected graph
 for (int i = 0; i < V; ++i)
 if (colorArr[i] == -1)
 if (isBipartiteUtil(G, i, colorArr) == false)
 return false;

 return true;
}

// Driver program to test above function
int main()
{
 int G[][V] = {{0, 1, 0, 1},
 {1, 0, 1, 0},
 {0, 1, 0, 1},
 {1, 0, 1, 0}};
};

isBipartite(G) ? cout << "Yes" : cout << "No";
return 0;
}

```

Output:

Yes

Time Complexity of the above approach is same as that Breadth First Search. In above implementation is  $O(V^2)$  where  $V$  is number of vertices. If graph is represented using adjacency list, then the complexity becomes  $O(V+E)$ .

#### Exercise:

1. Can DFS algorithm be used to check the bipartite-ness of a graph? If yes, how?

#### References:

[http://en.wikipedia.org/wiki/Graph\\_coloring](http://en.wikipedia.org/wiki/Graph_coloring)  
[http://en.wikipedia.org/wiki/Bipartite\\_graph](http://en.wikipedia.org/wiki/Bipartite_graph)

This article is compiled by Aashish Barnwal. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner Company Wise Coding Practice

Graph  
BFS

## Snake and Ladder Problem

Given a snake and ladder board, find the minimum number of dice throws required to reach the destination or last cell from source or 1st cell. Basically, the player has total control over outcome of dice throw and wants to find out minimum number of throws required to reach last cell.

If the player reaches a cell which is base of a ladder, the player has to climb up that ladder and if reaches a cell is mouth of the snake, has to go down to the tail of snake without a dice throw.



For example consider the board shown on right side (taken from [here](#)), the minimum number of dice throws required to reach cell 30 from cell 1 is 3. Following are steps.

- a) First throw two on dice to reach cell number 3 and then ladder to reach 22
- b) Then throw 6 to reach 28.
- c) Finally throw 2 to reach 30.

There can be other solutions as well like (2, 2, 6), (2, 4, 4), (2, 3, 5).. etc.

#### We strongly recommend to minimize the browser and try this yourself first.

The idea is to consider the given snake and ladder board as a directed graph with number of vertices equal to the number of cells in the board. The problem reduces to finding the shortest path in a graph. Every vertex of the graph has an edge to next six vertices if next 6 vertices do not have a snake or ladder. If any of the next six vertices has a snake or ladder, then the edge from current vertex goes to the top of the ladder or tail of the snake. Since all edges are of equal weight, we can efficiently find shortest path using Breadth First Search of the graph.

Following is C++ implementation of the above idea. The input is represented by two things, first is 'N' which is number of cells in the given board, second is an array 'move[0...N-1]' of size N. An entry move[i] is -1 if there is no snake and no ladder from i, otherwise move[i] contains index of destination cell for the snake or the ladder at i.

```

// C++ program to find minimum number of dice throws required to
// reach last cell from first cell of a given snake and ladder
// board
#include<iostream>

```

```

#include <queue>
using namespace std;

// An entry in queue used in BFS
struct queueEntry
{
 int v; // Vertex number
 int dist; // Distance of this vertex from source
};

// This function returns minimum number of dice throws required to
// Reach last cell from 0'th cell in a snake and ladder game.
// move[] is an array of size N where N is no. of cells on board
// If there is no snake or ladder from cell i, then move[i] is -1
// Otherwise move[i] contains cell to which snake or ladder at i
// takes to.
int getMinDiceThrows(int move[], int N)
{
 // The graph has N vertices. Mark all the vertices as
 // not visited
 bool *visited = new bool[N];
 for (int i = 0; i < N; i++)
 visited[i] = false;

 // Create a queue for BFS
 queue<queueEntry> q;

 // Mark the node 0 as visited and enqueue it.
 visited[0] = true;
 queueEntry s = {0, 0}; // distance of 0'th vertex is also 0
 q.push(s); // Enqueue 0'th vertex

 // Do a BFS starting from vertex at index 0
 queueEntryqe; // A queue entry (qe)
 while (!q.empty())
 {
 qe = q.front();
 int v = qe.v; // vertex no. of queue entry

 // If front vertex is the destination vertex,
 // we are done
 if (v == N-1)
 break;

 // Otherwise dequeue the front vertex and enqueue
 // its adjacent vertices (or cell numbers reachable
 // through a dice throw)
 q.pop();
 for (int j=v+1; j<=(v+6) && j<N; ++j)
 {
 // If this cell is already visited, then ignore
 if (!visited[j])
 {
 // Otherwise calculate its distance and mark it
 // as visited
 queueEntry a;
 a.dist = (qe.dist + 1);
 visited[j] = true;

 // Check if there a snake or ladder at 'j'
 // then tail of snake or top of ladder
 // become the adjacent of 'i'
 if (move[j] != -1)
 a.v = move[j];
 else
 a.v = j;
 q.push(a);
 }
 }
 }

 // We reach here when 'qe' has last vertex
 // return the distance of vertex in 'qe'
 return qe.dist;
}

// Driver program to test methods of graph class
int main()
{
 // Let us construct the board given in above diagram
 int N = 30;
 int moves[N];
 for (int i = 0; i < N; i++)
 moves[i] = -1;

 // Ladders
 moves[2] = 21;
 moves[4] = 7;
 moves[10] = 25;
 moves[19] = 28;

 // Snakes
 moves[26] = 0;
 moves[20] = 8;
 moves[16] = 3;
 moves[18] = 6;

 cout << "Min Dice throws required is " << getMinDiceThrows(moves, N);
 return 0;
}

```

Output:

Min Dice throws required is 3

Time complexity of the above solution is O(N) as every cell is added and removed only once from queue. And a typical enqueue or dequeue operation takes O(1) time.

## GATE CS Corner Company Wise Coding Practice

Graph  
BFS  
shortest path

### Biconnected Components

A **biconnected component** is a maximal **biconnected** subgraph.

Biconnected Graph is already discussed [here](#). In this article, we will see how to find biconnected component in a graph using algorithm by John Hopcroft and Robert Tarjan.

#### Biconnected Components



In above graph, following are the biconnected components:

- 4–2 3–4 3–1 2–3 1–2
- 8–9
- 8–5 7–8 5–7
- 6–0 5–6 1–5 0–1
- 10–11

Algorithm is based on Disc and Low Values discussed in [Strongly Connected Components Article](#).

Idea is to store visited edges in a stack while DFS on a graph and keep looking for **Articulation Points** (highlighted in above figure). As soon as an **Articulation Point** u is found, all edges visited while DFS from node u onwards will form one **biconnected component**. When DFS completes for one **connected component**, all edges present in stack will form a biconnected component.

If there is no **Articulation Point** in graph, then graph is biconnected and so there will be one biconnected component which is the graph itself.

#### C++

```
// A C++ program to find biconnected components in a given undirected graph
#include<iostream>
#include <list>
#include <stack>
#define NIL -1
using namespace std;
int count = 0;
class Edge
{
public:
 int u;
 int v;
 Edge(int u, int v);
};
Edge::Edge(int u, int v)
{
 this->u = u;
 this->v = v;
}

// A class that represents an directed graph
class Graph
{
 int V; // No. of vertices
 int E; // No. of edges
 list<int> *adj; // A dynamic array of adjacency lists

 // A Recursive DFS based function used by BCC()
 void BCCUtil(int u, int disc[], int low[],
 list<Edge> *st, int parent[]);
public:
 Graph(int V); // Constructor
 void addEdge(int v, int w); // function to add an edge to graph
 void BCC(); // prints strongly connected components
};

Graph::Graph(int V)
{
 this->V = V;
 this->E = 0;
 adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
 adj[v].push_back(w);
 E++;
}

// A recursive function that finds and prints strongly connected
// components using DFS traversal
// u -> The vertex to be visited next
// disc[] -> Stores discovery times of visited vertices
// low[] --> earliest visited vertex (the vertex with minimum
// discovery time) that can be reached from subtree
// rooted with current vertex
// *st --> To store visited edges
void Graph::BCCUtil(int u, int disc[], int low[], list<Edge> *st,
 int parent[])
{
 // A static variable is used for simplicity, we can avoid use
```

```

// of static variable by passing a pointer.
static int time = 0;

// Initialize discovery time and low value
disc[u] = low[u] = ++time;
int children = 0;

// Go through all vertices adjacent to this
list<int>::iterator i;
for (i = adj[u].begin(); i != adj[u].end(); ++i)
{
 int v = *i; // v is current adjacent of u'

 // If v is not visited yet, then recur for it
 if (disc[v] == -1)
 {
 children++;
 parent[v] = u;
 //Store the edge in stack
 st->push_back(Edge(u,v));
 BCCUtil(v, disc, low, st, parent);

 // Check if the subtree rooted with 'v' has a
 // connection to one of the ancestors of 'u'
 // Case 1 -- per Strongly Connected Components Article
 low[u] = min(low[u], low[v]);
 }

 // If u is an articulation point,
 // pop all edges from stack till u -- v
 if ((disc[u] == 1 && children > 1) ||
 (disc[u] > 1 && low[v] >= disc[u]))
 {
 while(st->back().u != u || st->back().v != v)
 {
 cout << st->back().u << "--" << st->back().v << " ";
 st->pop_back();
 }
 cout << st->back().u << "--" << st->back().v;
 st->pop_back();
 cout << endl;
 }
 count++;
}
}

// Update low value of 'u' only if 'v' is still in stack
// (i.e. it's a back edge, not cross edge).
// Case 2 -- per Strongly Connected Components Article
else if(v != parent[u] && disc[v] < low[u])
{
 low[u] = min(low[u], disc[v]);
 st->push_back(Edge(u,v));
}
}

// The function to do DFS traversal. It uses BCCUtil()
void Graph::BCC()
{
 int *disc = new int[V];
 int *low = new int[V];
 int *parent = new int[V];
 list<Edge> *st = new list<Edge>[E];

 // Initialize disc and low, and parent arrays
 for (int i = 0; i < V; i++)
 {
 disc[i] = NIL;
 low[i] = NIL;
 parent[i] = NIL;
 }

 for (int i = 0; i < V; i++)
 {
 if (disc[i] == NIL)
 BCCUtil(i, disc, low, st, parent);

 int j = 0;
 //If stack is not empty, pop all edges from stack
 while(st->size() > 0)
 {
 j = 1;
 cout << st->back().u << "--" << st->back().v << " ";
 st->pop_back();
 }
 if(j == 1)
 {
 cout << endl;
 count++;
 }
 }
}

// Driver program to test above function
int main()
{
 Graph g(12);
 g.addEdge(0,1).g.addEdge(1,0);
 g.addEdge(1,2).g.addEdge(2,1);
 g.addEdge(1,3).g.addEdge(3,1);
 g.addEdge(2,3).g.addEdge(3,2);
 g.addEdge(2,4).g.addEdge(4,2);
 g.addEdge(3,4).g.addEdge(4,3);
 g.addEdge(1,5).g.addEdge(5,1);
 g.addEdge(0,6).g.addEdge(6,0);
 g.addEdge(5,6).g.addEdge(6,5);
 g.addEdge(5,7).g.addEdge(7,5);
 g.addEdge(5,8).g.addEdge(8,5);
}

```

```

g.addEdge(7,8);g.addEdge(8,7);
g.addEdge(8,9);g.addEdge(9,8);
g.addEdge(10,11);g.addEdge(11,10);
g.BCC();
cout << "Above are " << count << " biconnected components in graph";
return 0;
}

```

## Java

```

// A Java program to find biconnected components in a given
// undirected graph
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
 private int V, E; // No. of vertices & Edges respectively
 private LinkedList<Integer> adj[]; // Adjacency List

 // Count is number of biconnected components. time is
 // used to find discovery times
 static int count = 0, time = 0;

 class Edge
 {
 int u;
 int v;
 Edge(int u, int v)
 {
 this.u = u;
 this.v = v;
 }
 };

 //Constructor
 Graph(int v)
 {
 V = v;
 E = 0;
 adj = new LinkedList[v];
 for (int i=0; i<v; ++i)
 adj[i] = new LinkedList();
 }

 //Function to add an edge into the graph
 void addEdge(int v,int w)
 {
 adj[v].add(w);
 E++;
 }

 // A recursive function that finds and prints strongly connected
 // components using DFS traversal
 // u -> The vertex to be visited next
 // disc[] --> Stores discovery times of visited vertices
 // low[] --> earliest visited vertex (the vertex with minimum
 // discovery time) that can be reached from subtree
 // rooted with current vertex
 // *st --> To store visited edges
 void BCCUtil(int u, int disc[], int low[], LinkedList<Edge>st,
 int parent[])
 {

 // Initialize discovery time and low value
 disc[u] = low[u] = ++time;
 int children = 0;

 // Go through all vertices adjacent to this
 Iterator<Integer> it = adj[u].iterator();
 while (it.hasNext())
 {
 int v = it.next(); // v is current adjacent of 'u'

 // If v is not visited yet, then recur for it
 if (disc[v] == -1)
 {
 children++;
 parent[v] = u;

 // store the edge in stack
 st.add(new Edge(u,v));
 BCCUtil(v, disc, low, st, parent);

 // Check if the subtree rooted with 'v' has a
 // connection to one of the ancestors of 'u'
 // Case 1 -- per Strongly Connected Components Article
 if ((low[u] > low[v])
 low[u] = low[v];
 }

 // If u is an articulation point,
 // pop all edges from stack till u -- v
 if ((disc[u] == 1 && children > 1) ||
 (disc[u] > 1 && low[v] >= disc[u]))
 {
 while (st.getLast().u != u || st.getLast().v != v)
 {
 System.out.print(st.getLast().u + "--"
 st.getLast().v + " ");
 st.removeLast();
 }
 System.out.println(st.getLast().u + "--" +

```

```

 st.getLast().v + " ");
 st.removeLast();

 count++;
}

}

// Update low value of 'u' only if 'v' is still in stack
// (i.e. it's a back edge, not cross edge).
// Case 2 -- per Strongly Connected Components Article
else if (v != parent[u] && disc[v] < low[u])
{
 if (low[u] > disc[v])
 low[u] = disc[v];
 st.add(new Edge(u,v));
}
}

// The function to do DFS traversal. It uses BCCUtil()
void BCC()
{
 int disc[] = new int[V];
 int low[] = new int[V];
 int parent[] = new int[V];
 LinkedList<Edge> st = new LinkedList<Edge>();

 // Initialize disc and low, and parent arrays
 for (int i = 0; i < V; i++)
 {
 disc[i] = -1;
 low[i] = -1;
 parent[i] = -1;
 }

 for (int i = 0; i < V; i++)
 {
 if (disc[i] == -1)
 BCCUtil(i, disc, low, st, parent);

 int j = 0;

 // If stack is not empty, pop all edges from stack
 while (st.size() > 0)
 {
 j = 1;
 System.out.print(st.getLast().u + "--"
 + st.getLast().v + " ");
 st.removeLast();
 }
 if (j == 1)
 {
 System.out.println();
 count++;
 }
 }
}

public static void main(String args[])
{
 Graph g = new Graph(12);
 g.addEdge(0,1);
 g.addEdge(1,0);
 g.addEdge(1,2);
 g.addEdge(2,1);
 g.addEdge(1,3);
 g.addEdge(3,1);
 g.addEdge(2,3);
 g.addEdge(3,2);
 g.addEdge(2,4);
 g.addEdge(4,2);
 g.addEdge(3,4);
 g.addEdge(4,3);
 g.addEdge(1,5);
 g.addEdge(5,1);
 g.addEdge(0,6);
 g.addEdge(6,0);
 g.addEdge(5,6);
 g.addEdge(6,5);
 g.addEdge(5,7);
 g.addEdge(7,5);
 g.addEdge(5,8);
 g.addEdge(8,5);
 g.addEdge(7,8);
 g.addEdge(8,7);
 g.addEdge(8,9);
 g.addEdge(9,8);
 g.addEdge(10,11);
 g.addEdge(11,10);

 g.BCC();

 System.out.println("Above are " + g.count +
 " biconnected components in graph");
}
}

// This code is contributed by Aakash Hasija

```

## Python

```

Python program to find biconnected components in a given
undirected graph
#Complexity : O(V+E)

```

```

from collections import defaultdict

#This class represents an directed graph
using adjacency list representation
class Graph:

 def __init__(self,vertices):
 #No. of vertices
 self.V= vertices

 # default dictionary to store graph
 self.graph = defaultdict(list)

 # time is used to find discovery times
 self.Time = 0

 # Count is number of biconnected components
 self.count = 0

 # function to add an edge to graph
 def addEdge(self,u,v):
 self.graph[u].append(v)
 self.graph[v].append(u)

 """A recursive function that finds and prints strongly connected
 components using DFS traversal
 u --> The vertex to be visited next
 disc[] --> Stores discovery times of visited vertices
 low[] --> earliest visited vertex (the vertex with minimum
 discovery time) that can be reached from subtree
 rooted with current vertex
 st -->> To store visited edges"""
 def BCCUtil(self,u, parent, low, disc, st):

 #Count of children in current node
 children =0

 # Initialize discovery time and low value
 disc[u] = self.Time
 low[u] = self.Time
 self.Time += 1

 #Recur for all the vertices adjacent to this vertex
 for v in self.graph[u]:
 # If v is not visited yet, then make it a child of u
 # in DFS tree and recur for it
 if disc[v] == -1 :
 parent[v] = u
 children += 1
 st.append((u, v)) #store the edge in stack
 self.BCCUtil(v, parent, low, disc, st)

 # Check if the subtree rooted with v has a connection to
 # one of the ancestors of u
 # Case 1 -- per Strongly Connected Components Article
 low[u] = min(low[u], low[v])

 # If u is an articulation point,pop
 # all edges from stack till (u, v)
 if parent[u] == -1 and children > 1 or parent[u] != -1 and low[v] >= disc[u]:
 self.count +=1 # increment count
 w = -1
 while w != (u,v):
 w = st.pop()
 print w,
 print ""

 elif v != parent[u] and low[u] > disc[v]:
 """Update low value of 'u' only if 'v' is still in stack
 (i.e. it's a back edge, not cross edge).
 Case 2
 -- per Strongly Connected Components Article"""

 low[u] = min(low[u], disc[v])
 st.append((u,v))

 #The function to do DFS traversal.
 # It uses recursive BCCUtil()
 def BCC(self):

 # Initialize disc and low, and parent arrays
 disc = [-1] * (self.V)
 low = [-1] * (self.V)
 parent = [-1] * (self.V)
 st = []

 # Call the recursive helper function to
 # find articulation points
 # in DFS tree rooted with vertex 'i'
 for i in range(self.V):
 if disc[i] == -1:
 self.BCCUtil(i, parent, low, disc, st)

 #If stack is not empty, pop all edges from stack
 if st:
 self.count = self.count + 1

 while st:
 w = st.pop()
 print w,
 print ""

 # Create a graph given in the above diagram

```

```

g = Graph(12)
g.addEdge(0,1)
g.addEdge(1,2)
g.addEdge(1,3)
g.addEdge(2,3)
g.addEdge(2,4)
g.addEdge(3,4)
g.addEdge(1,5)
g.addEdge(0,6)
g.addEdge(5,6)
g.addEdge(5,7)
g.addEdge(5,8)
g.addEdge(7,8)
g.addEdge(8,9)
g.addEdge(10,11)

g.BCC();
print ("Above are %d biconnected components in graph" %(g.count));

#This code is contributed by Neelam Yadav

```

Output:

```

4--2 3--4 3--1 2--3 1--2
8--9
8--5 7--8 5--7
6--0 5--6 1--5 0--1
10--11
Above are 5 biconnected components in graph

```

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner Company Wise Coding Practice

Graph  
graph-connectivity

### Check if a given graph is tree or not

Write a function that returns true if a given undirected graph is tree and false otherwise. For example, the following graph is a tree.

`cycleGraph`



But the following graph is not a tree.

`cycleGraph`



An undirected graph is tree if it has following properties.

- 1) There is no cycle.
- 2) The graph is connected.

For an undirected graph we can either use [BFS](#) or [DFS](#) to detect above two properties.

#### How to detect cycle in an undirected graph?

We can either use BFS or DFS. For every visited vertex 'v', if there is an adjacent 'u' such that u is already visited and u is not parent of v, then there is a cycle in graph. If we don't find such an adjacent for any vertex, we say that there is no cycle (See [Detect cycle in an undirected graph](#) for more details).

#### How to check for connectivity?

Since the graph is undirected, we can start BFS or DFS from any vertex and check if all vertices are reachable or not. If all vertices are reachable, then graph is connected, otherwise not.

### C++

```

// A C++ Program to check whether a graph is tree or not
#include<iostream>
#include <list>
#include <limits.h>
using namespace std;

// Class for an undirected graph
class Graph
{
 int V; // No. of vertices
 list<int> *adj; // Pointer to an array for adjacency lists
 bool isCyclicUtil(int v, bool visited[], int parent);

public:
 Graph(int V); // Constructor
 void addEdge(int v, int w); // to add an edge to graph
 bool isTree(); // returns true if graph is tree
};

Graph::Graph(int V)
{
 this->V = V;
 adj = new list<int>[V];
}

bool Graph::isCyclicUtil(int v, bool visited[], int parent)
{
 if (visited[v] == true)
 return true;
 visited[v] = true;
 list<int> ::iterator i;
 for (i = adj[v].begin(); i != adj[v].end(); i++)
 {
 if (*i == parent)
 continue;
 if (isCyclicUtil(*i, visited, v))
 return true;
 }
 return false;
}

bool Graph::isTree()
{
 bool visited[V];
 for (int i = 0; i < V; i++)
 visited[i] = false;
 if (isCyclicUtil(0, visited, -1))
 return false;
 for (int i = 0; i < V; i++)
 if (!visited[i])
 return false;
 return true;
}

```

```

void Graph::addEdge(int v, int w)
{
 adj[v].push_back(w); // Add w to v's list.
 adj[w].push_back(v); // Add v to w's list.
}

// A recursive function that uses visited[] and parent to
// detect cycle in subgraph reachable from vertex v.
bool Graph::isCyclicUtil(int v, bool visited[], int parent)
{
 // Mark the current node as visited
 visited[v] = true;

 // Recur for all the vertices adjacent to this vertex
 list<int>::iterator i;
 for (i = adj[v].begin(); i != adj[v].end(); ++i)
 {
 // If an adjacent is not visited, then recur for
 // that adjacent
 if (!visited[*i])
 {
 if (isCyclicUtil(*i, visited, v))
 return true;
 }
 }

 // If an adjacent is visited and not parent of current
 // vertex, then there is a cycle.
 else if (*i != parent)
 return true;
 }

 return false;
}

// Returns true if the graph is a tree, else false.
bool Graph::isTree()
{
 // Mark all the vertices as not visited and not part of
 // recursion stack
 bool *visited = new bool[V];
 for (int i = 0; i < V; i++)
 visited[i] = false;

 // The call to isCyclicUtil serves multiple purposes.
 // It returns true if graph reachable from vertex 0
 // is cyclic. It also marks all vertices reachable
 // from 0.
 if (isCyclicUtil(0, visited, -1))
 return false;

 // If we find a vertex which is not reachable from 0
 // (not marked by isCyclicUtil()), then we return false
 for (int u = 0; u < V; u++)
 if (!visited[u])
 return false;

 return true;
}

// Driver program to test above functions
int main()
{
 Graph g1(5);
 g1.addEdge(1, 0);
 g1.addEdge(0, 2);
 g1.addEdge(0, 3);
 g1.addEdge(3, 4);
 g1.isTree() ? cout << "Graph is Tree\n":
 cout << "Graph is not Tree\n";

 Graph g2(5);
 g2.addEdge(1, 0);
 g2.addEdge(0, 2);
 g2.addEdge(2, 1);
 g2.addEdge(0, 3);
 g2.addEdge(3, 4);
 g2.isTree() ? cout << "Graph is Tree\n":
 cout << "Graph is not Tree\n";

 return 0;
}

```

## Java

```

// A Java Program to check whether a graph is tree or not
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
 private int V; // No. of vertices
 private LinkedList<Integer> adj[]; //Adjacency List

 // Constructor
 Graph(int v)
 {
 V = v;
 adj = new LinkedList[v];
 for (int i=0; i<v; ++i)
 adj[i] = new LinkedList();
 }

 // Function to add an edge into the graph

```

```

void addEdge(int v,int w)
{
 adj[v].add(w);
 adj[w].add(v);
}

// A recursive function that uses visited[] and parent
// to detect cycle in subgraph reachable from vertex v.
Boolean isCyclicUtil(int v, Boolean visited[], int parent)
{
 // Mark the current node as visited
 visited[v] = true;
 Integer i;

 // Recur for all the vertices adjacent to this vertex
 Iterator<Integer> it = adj[v].iterator();
 while (it.hasNext())
 {
 i = it.next();

 // If an adjacent is not visited, then recur for
 // that adjacent
 if (!visited[i])
 {
 if (isCyclicUtil(i, visited, v))
 return true;
 }

 // If an adjacent is visited and not parent of
 // current vertex, then there is a cycle.
 else if (i != parent)
 return true;
 }
 return false;
}

// Returns true if the graph is a tree, else false.
Boolean isTree()
{
 // Mark all the vertices as not visited and not part
 // of recursion stack
 Boolean visited[] = new Boolean[V];
 for (int i = 0; i < V; i++)
 visited[i] = false;

 // The call to isCyclicUtil serves multiple purposes
 // It returns true if graph reachable from vertex 0
 // is cyclic. It also marks all vertices reachable
 // from 0.
 if (isCyclicUtil(0, visited, -1))
 return false;

 // If we find a vertex which is not reachable from 0
 // (not marked by isCyclicUtil()), then we return false
 for (int u = 0; u < V; u++)
 if (!visited[u])
 return false;

 return true;
}

// Driver method
public static void main(String args[])
{
 // Create a graph given in the above diagram
 Graph g1 = new Graph(5);
 g1.addEdge(1, 0);
 g1.addEdge(0, 2);
 g1.addEdge(0, 3);
 g1.addEdge(3, 4);
 if (g1.isTree())
 System.out.println("Graph is Tree");
 else
 System.out.println("Graph is not Tree");

 Graph g2 = new Graph(5);
 g2.addEdge(1, 0);
 g2.addEdge(0, 2);
 g2.addEdge(2, 1);
 g2.addEdge(0, 3);
 g2.addEdge(3, 4);

 if (g2.isTree())
 System.out.println("Graph is Tree");
 else
 System.out.println("Graph is not Tree");
}
}

// This code is contributed by Aakash Hasija

```

Output:

```

Graph is Tree
Graph is not Tree

```

Thanks to **Vinit Verma** for suggesting this problem and initial solution. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**GATE CS Notes (According to Official GATE 2017 Syllabus)**

**GATE CS Corner**

## Greedy Algorithms | Set 5 (Prim's Minimum Spanning Tree (MST))

We have discussed [Kruskal's algorithm for Minimum Spanning Tree](#). Like Kruskal's algorithm, Prim's algorithm is also a [Greedy algorithm](#). It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

A group of edges that connects two set of vertices in a graph is called [cut in graph theory](#). So, at every step of Prim's algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the vertices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).

**How does Prim's Algorithm Work?** The idea behind Prim's algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a *Spanning Tree*. And they must be connected with the minimum weight edge to make it a *Minimum Spanning Tree*.

### Algorithm

- 1) Create a set *mstSet* that keeps track of vertices already included in MST.
- 2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
- 3) While *mstSet* doesn't include all vertices
  - ....a) Pick a vertex *u* which is not there in *mstSet* and has minimum key value.
  - ....b) Include *u* to *mstSet*.
  - ....c) Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u-v* is less than the previous key value of *v*, update the key value as weight of *u-v*

The idea of using key values is to pick the minimum weight edge from [cut](#). The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

Let us understand with the following example:



The set *mstSet* is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum key value. The vertex 0 is picked, include it in *mstSet*. So *mstSet* becomes {0}. After including to *mstSet*, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.



Pick the vertex with minimum key value and not already included in MST (not in *mstSET*). The vertex 1 is picked and added to *mstSet*. So *mstSet* now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.



Pick the vertex with minimum key value and not already included in MST (not in *mstSET*). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So *mstSet* now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (7 and 1 respectively).



Pick the vertex with minimum key value and not already included in MST (not in *mstSET*). Vertex 6 is picked. So *mstSet* now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.



We repeat the above steps until *mstSet* includes all vertices of given graph. Finally, we get the following graph.



### How to implement the above algorithm?

We use a boolean array *mstSet[]* to represent the set of vertices included in MST. If a value *mstSet[v]* is true, then vertex *v* is included in MST, otherwise not. Array *key[]* is used to store key values of all vertices. Another array *parent[]* to store indexes of parent nodes in MST. The parent array is the output array which is used to show the constructed MST.

### C/C++

```
// A C / C++ program for Prim's Minimum Spanning Tree (MST) algorithm.
// The program is for adjacency matrix representation of the graph
```

```

#include <stdio.h>
#include <limits.h>

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with minimum key value, from
// the set of vertices not yet included in MST
int minKey(int key[], bool mstSet[])
{
 // Initialize min value
 int min = INT_MAX, min_index;

 for (int v = 0; v < V; v++)
 if (mstSet[v] == false && key[v] < min)
 min = key[v], min_index = v;

 return min_index;
}

// A utility function to print the constructed MST stored in parent[]
int printMST(int parent[], int n, int graph[V][V])
{
 printf("Edge Weight\n");
 for (int i = 1; i < V; i++)
 printf("%d - %d %d \n", parent[i], i, graph[i][parent[i]]);

}

// Function to construct and print MST for a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
 int parent[V]; // Array to store constructed MST
 int key[V]; // Key values used to pick minimum weight edge in cut
 bool mstSet[V]; // To represent set of vertices not yet included in MST

 // Initialize all keys as INFINITE
 for (int i = 0; i < V; i++)
 key[i] = INT_MAX, mstSet[i] = false;

 // Always include first vertex in MST.
 key[0] = 0; // Make key 0 so that this vertex is picked as first vertex
 parent[0] = -1; // First node is always root of MST

 // The MST will have V vertices
 for (int count = 0; count < V-1; count++)
 {
 // Pick the minimum key vertex from the set of vertices
 // not yet included in MST
 int u = minKey(key, mstSet);

 // Add the picked vertex to the MST Set
 mstSet[u] = true;

 // Update key value and parent index of the adjacent vertices of
 // the picked vertex. Consider only those vertices which are not yet
 // included in MST
 for (int v = 0; v < V; v++)
 {
 // graph[u][v] is non zero only for adjacent vertices of m
 // mstSet[v] is false for vertices not yet included in MST
 // Update the key only if graph[u][v] is smaller than key[v]
 if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
 parent[v] = u, key[v] = graph[u][v];
 }
 }

 // print the constructed MST
 printMST(parent, V, graph);
}

// driver program to test above function
int main()
{
 /* Let us create the following graph
 * 2 3
 * (0)--(1)--(2)
 * | / |
 * 6| 8/ 5|7
 * | / \|
 * (3)-----(4)
 * 9 */
 int graph[V][V] = {{0, 2, 0, 6, 0},
 {2, 0, 3, 8, 5},
 {0, 3, 0, 0, 7},
 {6, 8, 0, 0, 9},
 {0, 5, 7, 9, 0},
 };

 // Print the solution
 primMST(graph);

 return 0;
}

```

## Java

```

// A Java program for Prim's Minimum Spanning Tree (MST) algorithm.
// The program is for adjacency matrix representation of the graph

import java.util.*;
import java.lang.*;
import java.io.*;

```

```

class MST
{
 // Number of vertices in the graph
 private static final int V=5;

 // A utility function to find the vertex with minimum key
 // value, from the set of vertices not yet included in MST
 int minKey(int key[], Boolean mstSet[])
 {
 // Initialize min value
 int min = Integer.MAX_VALUE, min_index=-1;

 for (int v = 0; v < V; v++)
 if (mstSet[v] == false && key[v] < min)
 {
 min = key[v];
 min_index = v;
 }

 return min_index;
 }

 // A utility function to print the constructed MST stored in
 // parent[]
 void printMST(int parent[], int n, int graph[][])
 {
 System.out.println("Edge Weight");
 for (int i = 1; i < V; i++)
 System.out.println(parent[i]+ " - "+ i+ " "+
 graph[i][parent[i]]);
 }

 // Function to construct and print MST for a graph represented
 // using adjacency matrix representation
 void primMST(int graph[][])
 {
 // Array to store constructed MST
 int parent[] = new int[V];

 // Key values used to pick minimum weight edge in cut
 int key[] = new int[V];

 // To represent set of vertices not yet included in MST
 Boolean mstSet[] = new Boolean[V];

 // Initialize all keys as INFINITE
 for (int i = 0; i < V; i++)
 {
 key[i] = Integer.MAX_VALUE;
 mstSet[i] = false;
 }

 // Always include first 1st vertex in MST.
 key[0] = 0; // Make key 0 so that this vertex is
 // picked as first vertex
 parent[0] = -1; // First node is always root of MST

 // The MST will have V vertices
 for (int count = 0; count < V-1; count++)
 {
 // Pick thd minimum key vertex from the set of vertices
 // not yet included in MST
 int u = minKey(key, mstSet);

 // Add the picked vertex to the MST Set
 mstSet[u] = true;

 // Update key value and parent index of the adjacent
 // vertices of the picked vertex. Consider only those
 // vertices which are not yet included in MST
 for (int v = 0; v < V; v++)

 // graph[u][v] is non zero only for adjacent vertices of m
 // mstSet[v] is false for vertices not yet included in MST
 // Update the key only if graph[u][v] is smaller than key[v]
 if (graph[u][v]>0 && mstSet[v] == false &&
 graph[u][v] < key[v])
 {
 parent[v] = u;
 key[v] = graph[u][v];
 }
 }

 // print the constructed MST
 printMST(parent, V, graph);
 }

 public static void main (String[] args)
 {
 /* Let us create the following graph
 2 3
 (0)-(1)-(2)
 | / |
 6 8 / 5 \ 7
 | / \ \
 (3)-----(4)
 9 */
 MST t = new MST();
 int graph[][] = new int[][] {{0, 2, 0, 6, 0},
 {2, 0, 3, 8, 5},
 {0, 3, 0, 0, 7},
 {6, 8, 0, 0, 9},
 {0, 5, 7, 9, 0},
 };
 }

 // Print the solution
}

```

```
t.primMST(graph);
}
// This code is contributed by Aakash Hasija
```

Output:

| Edge  | Weight |
|-------|--------|
| 0 - 1 | 2      |
| 1 - 2 | 3      |
| 0 - 3 | 6      |
| 1 - 4 | 5      |

Time Complexity of the above program is  $O(V^2)$ . If the input graph is represented using adjacency list, then the time complexity of Prim's algorithm can be reduced to  $O(E \log V)$  with the help of binary heap. Please see [Prim's MST for Adjacency List Representation](#) for more details.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner Company Wise Coding Practice

Graph  
Greedy  
Greedy Algorithm  
Minimum Spanning Tree  
Prim's Algorithm.MST

## Applications of Minimum Spanning Tree Problem

Minimum Spanning Tree (MST) problem: Given connected graph G with positive edge weights, find a min weight set of edges that connects all of the vertices.

MST is fundamental problem with diverse applications.



### Network design.

- telephone, electrical, hydraulic, TV cable, computer, road

The standard application is to a problem like phone network design. You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. It should be a spanning tree, since if a network isn't a tree you can always remove some edges and save money.

### Approximation algorithms for NP-hard problems.

- traveling salesperson problem, Steiner tree

A less obvious application is that the minimum spanning tree can be used to approximately solve the traveling salesman problem. A convenient formal way of defining this problem is to find the shortest path that visits each point at least once.

Note that if you have a path visiting all points exactly once, it's a special kind of tree. For instance in the example above, twelve of sixteen spanning trees are actually paths. If you have a path visiting some vertices more than once, you can always drop some edges to get a tree. So in general the MST weight is less than the TSP weight, because it's a minimization over a strictly larger set.

On the other hand, if you draw a path tracing around the minimum spanning tree, you trace each edge twice and visit all points, so the TSP weight is less than twice the MST weight. Therefore this tour is within a factor of two of optimal.

### Indirect applications.

- max bottleneck paths
- LDPC codes for error correction
- image registration with Renyi entropy
- learning salient features for real-time face verification
- reducing data storage in sequencing amino acids in a protein
- model locality of particle interactions in turbulent fluid flows
- autoconfig protocol for Ethernet bridging to avoid cycles in a network

### Cluster analysis

k clustering problem can be viewed as finding an MST and deleting the k-1 most expensive edges.

**Sources:**

<http://www.cs.princeton.edu/courses/archive/spr07/cos226/lectures/mst.pdf>  
<http://www.ics.uci.edu/~eppstein/161/960206.html>

**GATE CS Corner Company Wise Coding Practice**

Graph  
Greedy  
Trees  
Graph  
MST

## Greedy Algorithms | Set 6 (Prim's MST for Adjacency List Representation)

We recommend to read following two posts as a prerequisite of this post.

1. [Greedy Algorithms | Set 5 \(Prim's Minimum Spanning Tree \(MST\)\)](#)
2. [Graph and its representations](#)

We have discussed [Prim's algorithm](#) and its implementation for adjacency matrix representation of graphs. The time complexity for the matrix representation is  $O(V^2)$ . In this post,  $O(E \log V)$  algorithm for adjacency list representation is discussed.

As discussed in the previous post, in Prim's algorithm, two sets are maintained, one set contains list of vertices already included in MST, other set contains vertices not yet included. With adjacency list representation, all vertices of a graph can be traversed in  $O(V+E)$  time using [BFS](#). The idea is to traverse all vertices of graph using [BFS](#) and use a Min Heap to store the vertices not yet included in MST. Min Heap is used as a priority queue to get the minimum weight edge from the [cut](#). Min Heap is used as time complexity of operations like extracting minimum element and decreasing key value is  $O(\log V)$  in Min Heap.

Following are the detailed steps.

- 1) Create a Min Heap of size  $V$  where  $V$  is the number of vertices in the given graph. Every node of min heap contains vertex number and key value of the vertex.
- 2) Initialize Min Heap with first vertex as root (the key value assigned to first vertex is 0). The key value assigned to all other vertices is INF (infinite).
- 3) While Min Heap is not empty, do following
  - .....a) Extract the min value node from Min Heap. Let the extracted vertex be  $u$ .
  - .....b) For every adjacent vertex  $v$  of  $u$ , check if  $v$  is in Min Heap (not yet included in MST). If  $v$  is in Min Heap and its key value is more than weight of  $u-v$ , then update the key value of  $v$  as weight of  $u-v$ .

Let us understand the above algorithm with the following example:



Initially, key value of first vertex is 0 and INF (infinite) for all other vertices. So vertex 0 is extracted from Min Heap and key values of vertices adjacent to 0 (1 and 7) are updated. Min Heap contains all vertices except vertex 0.

The vertices in green color are the vertices included in MST.



Since key value of vertex 1 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 1 are updated (Key is updated if the a vertex is not in Min Heap and previous key value is greater than the weight of edge from 1 to the adjacent). Min Heap contains all vertices except vertex 0 and 1.



Since key value of vertex 7 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 7 are updated (Key is updated if the a vertex is not in Min Heap and previous key value is greater than the weight of edge from 7 to the adjacent). Min Heap contains all vertices except vertex 0, 1 and 7.



Since key value of vertex 6 is minimum among all nodes in Min Heap, it is extracted from Min Heap and key values of vertices adjacent to 6 are updated (Key is updated if the a vertex is not in Min Heap and previous key value is greater than the weight of edge from 6 to the adjacent). Min Heap contains all vertices except vertex 0, 1, 7 and 6.



The above steps are repeated for rest of the nodes in Min Heap till Min Heap becomes empty



```
// C / C++ program for Prim's MST for adjacency list representation of graph
```

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a node in adjacency list
struct AdjListNode
{
 int dest;
 int weight;
 struct AdjListNode* next;
};

// A structure to represent an adjacency list
struct AdjList
{
 struct AdjListNode *head; // pointer to head node of list
};

// A structure to represent a graph. A graph is an array of adjacency lists.
// Size of array will be V (number of vertices in graph)
struct Graph
{
 int V;
 struct AdjList* array;
};

// A utility function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest, int weight)
{
 struct AdjListNode* newNode =
 (struct AdjListNode*) malloc(sizeof(struct AdjListNode));
 newNode->dest = dest;
 newNode->weight = weight;
 newNode->next = NULL;
 return newNode;
}

// A utility function that creates a graph of V vertices
struct Graph* createGraph(int V)
{
 struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
 graph->V = V;

 // Create an array of adjacency lists. Size of array will be V
 graph->array = (struct AdjList*) malloc(V * sizeof(struct AdjList));

 // Initialize each adjacency list as empty by making head as NULL
 for (int i = 0; i < V; ++i)
 graph->array[i].head = NULL;

 return graph;
}

// Adds an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest, int weight)
{
 // Add an edge from src to dest. A new node is added to the adjacency
 // list of src. The node is added at the beginning
 struct AdjListNode* newNode = newAdjListNode(dest, weight);
 newNode->next = graph->array[src].head;
 graph->array[src].head = newNode;

 // Since graph is undirected, add an edge from dest to src also
 newNode = newAdjListNode(src, weight);
 newNode->next = graph->array[dest].head;
 graph->array[dest].head = newNode;
}

// Structure to represent a min heap node
struct MinHeapNode
{
 int v;
 int key;
};

// Structure to represent a min heap
struct MinHeap
{
 int size; // Number of heap nodes present currently
 int capacity; // Capacity of min heap
 int *pos; // This is needed for decreaseKey()
 struct MinHeapNode **array;
};

// A utility function to create a new Min Heap Node
struct MinHeapNode* newMinHeapNode(int v, int key)
```

```

{
 struct MinHeapNode* minHeapNode =
 (struct MinHeapNode*) malloc(sizeof(struct MinHeapNode));
 minHeapNode->v = v;
 minHeapNode->key = key;
 return minHeapNode;
}

// A utilit function to create a Min Heap
struct MinHeap* createMinHeap(int capacity)
{
 struct MinHeap* minHeap =
 (struct MinHeap*) malloc(sizeof(struct MinHeap));
 minHeap->pos = (int *)malloc(capacity * sizeof(int));
 minHeap->size = 0;
 minHeap->capacity = capacity;
 minHeap->array =
 (struct MinHeapNode**) malloc(capacity * sizeof(struct MinHeapNode*));
 return minHeap;
}

// A utility function to swap two nodes of min heap. Needed for min heapify
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b)
{
 struct MinHeapNode* t = *a;
 *a = *b;
 *b = t;
}

// A standard function to heapify at given idx
// This function also updates position of nodes when they are swapped.
// Position is needed for decreaseKey()
void minHeapify(struct MinHeap* minHeap, int idx)
{
 int smallest, left, right;
 smallest = idx;
 left = 2 * idx + 1;
 right = 2 * idx + 2;

 if (left < minHeap->size &&
 minHeap->array[left]->key < minHeap->array[smallest]->key)
 smallest = left;

 if (right < minHeap->size &&
 minHeap->array[right]->key < minHeap->array[smallest]->key)
 smallest = right;

 if (smallest != idx)
 {
 // The nodes to be swapped in min heap
 MinHeapNode *smallestNode = minHeap->array[smallest];
 MinHeapNode *idxNode = minHeap->array[idx];

 // Swap positions
 minHeap->pos[smallestNode->v] = idx;
 minHeap->pos[idxNode->v] = smallest;

 // Swap nodes
 swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

 minHeapify(minHeap, smallest);
 }
}

// A utility function to check if the given minHeap is ampty or not
int isEmpty(struct MinHeap* minHeap)
{
 return minHeap->size == 0;
}

// Standard function to extract minimum node from heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap)
{
 if (isEmpty(minHeap))
 return NULL;

 // Store the root node
 struct MinHeapNode* root = minHeap->array[0];

 // Replace root node with last node
 struct MinHeapNode* lastNode = minHeap->array[minHeap->size - 1];
 minHeap->array[0] = lastNode;

 // Update position of last node
 minHeap->pos[root->v] = minHeap->size-1;
 minHeap->pos[lastNode->v] = 0;

 // Reduce heap size and heapify root
 --minHeap->size;
 minHeapify(minHeap, 0);

 return root;
}

// Function to decreasy key value of a given vertex v. This function
// uses pos[] of min heap to get the current index of node in min heap
void decreaseKey(struct MinHeap* minHeap, int v, int key)
{
 // Get the index of v in heap array
 int i = minHeap->pos[v];

 // Get the node and update its key value
 minHeap->array[i]->key = key;

 // Travel up while the complete tree is not heepified.
 // This is a O(Logn) loop
 while (i && minHeap->array[i]->key < minHeap->array[(i - 1) / 2]->key)
}

```

```

{
 // Swap this node with its parent
 minHeap->pos[minHeap->array[i]->v] = (i-1)/2;
 minHeap->pos[minHeap->array[(i-1)/2]->v] = i;
 swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

 // move to parent index
 i = (i - 1) / 2;
}
}

// A utility function to check if a given vertex
// 'v' is in min heap or not
bool isInMinHeap(struct MinHeap *minHeap, int v)
{
 if (minHeap->pos[v] < minHeap->size)
 return true;
 return false;
}

// A utility function used to print the constructed MST
void printArr(int arr[], int n)
{
 for (int i = 1; i < n; ++i)
 printf("%d - %d\n", arr[i], i);
}

// The main function that constructs Minimum Spanning Tree (MST)
// using Prim's algorithm
void PrimMST(struct Graph* graph)
{
 int V = graph->V;// Get the number of vertices in graph
 int parent[V]; // Array to store constructed MST
 int key[V]; // Key values used to pick minimum weight edge in cut

 // minHeap represents set E
 struct MinHeap* minHeap = createMinHeap(V);

 // Initialize min heap with all vertices. Key value of
 // all vertices (except 0th vertex) is initially infinite
 for (int v = 1; v < V; ++v)
 {
 parent[v] = -1;
 key[v] = INT_MAX;
 minHeap->array[v] = newMinHeapNode(v, key[v]);
 minHeap->pos[v] = v;
 }

 // Make key value of 0th vertex as 0 so that it
 // is extracted first
 key[0] = 0;
 minHeap->array[0] = newMinHeapNode(0, key[0]);
 minHeap->pos[0] = 0;

 // Initially size of min heap is equal to V
 minHeap->size = V;

 // In the followin loop, min heap contains all nodes
 // not yet added to MST.
 while (!isEmpty(minHeap))
 {
 // Extract the vertex with minimum key value
 struct MinHeapNode* minHeapNode = extractMin(minHeap);
 int u = minHeapNode->v; // Store the extracted vertex number

 // Traverse through all adjacent vertices of u (the extracted
 // vertex) and update their key values
 struct AdjListNode* pCrawl = graph->array[u].head;
 while (pCrawl != NULL)
 {
 int v = pCrawl->dest;

 // If v is not yet included in MST and weight of u-v is
 // less than key value of v, then update key value and
 // parent of v
 if (isInMinHeap(minHeap, v) && pCrawl->weight < key[v])
 {
 key[v] = pCrawl->weight;
 parent[v] = u;
 decreaseKey(minHeap, v, key[v]);
 }
 pCrawl = pCrawl->next;
 }
 }

 // print edges of MST
 printArr(parent, V);
}

// Driver program to test above functions
int main()
{
 // Let us create the graph given in above fugure
 int V = 9;
 struct Graph* graph = createGraph(V);
 addEdge(graph, 0, 1, 4);
 addEdge(graph, 0, 7, 8);
 addEdge(graph, 1, 2, 8);
 addEdge(graph, 1, 7, 11);
 addEdge(graph, 2, 3, 7);
 addEdge(graph, 2, 8, 2);
 addEdge(graph, 2, 5, 4);
 addEdge(graph, 3, 4, 9);
 addEdge(graph, 3, 5, 14);
 addEdge(graph, 4, 5, 10);
 addEdge(graph, 5, 6, 2);
 addEdge(graph, 6, 7, 1);
}

```

```

 addEdge(graph, 6, 8, 6);
 addEdge(graph, 7, 8, 7);

 PrimMST(graph);

 return 0;
}

```

Output:

```

0 - 1
5 - 2
2 - 3
3 - 4
6 - 5
7 - 6
0 - 7
2 - 8

```

**Time Complexity:** The time complexity of the above code/algorithm looks  $O(V^2)$  as there are two nested while loops. If we take a closer look, we can observe that the statements in inner loop are executed  $O(V+E)$  times (similar to BFS). The inner loop has `decreaseKey()` operation which takes  $O(\log V)$  time. So overall time complexity is  $O(E+V) \cdot O(\log V)$  which is  $O((E+V) \cdot \log V) = O(E \log V)$  (For a connected graph,  $V = O(E)$ )

#### References:

Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

[http://en.wikipedia.org/wiki/Prim's\\_algorithm](http://en.wikipedia.org/wiki/Prim's_algorithm)

This article is compiled by Aashish Barnwal and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner Company Wise Coding Practice

Graph  
Greedy  
Graph  
Greedy Algorithm  
MST

## Greedy Algorithms | Set 2 (Kruskal's Minimum Spanning Tree Algorithm)

*What is Minimum Spanning Tree?*

Given a connected and undirected graph, a *spanning tree* of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A *minimum spanning tree (MST)* or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

*How many edges does a minimum spanning tree has?*

A minimum spanning tree has  $(V - 1)$  edges where  $V$  is the number of vertices in the given graph.

*What are the applications of Minimum Spanning Tree?*

See this for applications of MST.

Below are the steps for finding MST using Kruskal's algorithm

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are  $(V-1)$  edges in the spanning tree.

The step#2 uses [Union-Find algorithm](#) to detect cycle. So we recommend to read following post as a prerequisite.

[Union-Find Algorithm | Set 1 \(Detect Cycle in a Graph\)](#)

[Union-Find Algorithm | Set 2 \(Union By Rank and Path Compression\)](#)

The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example: Consider the below input graph.



The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having  $(9 - 1) = 8$  edges.

After sorting:  
Weight Src Dest

```

1 7 6
2 8 2
2 6 5
4 0 1
4 2 5
6 8 6
7 2 3
7 7 8
8 0 7
8 1 2
9 3 4
10 5 4
11 1 7
14 3 5

```

Now pick all edges one by one from sorted list of edges

1. Pick edge 7-6: No cycle is formed, include it.



2. Pick edge 8-2: No cycle is formed, include it.



3. Pick edge 6-5: No cycle is formed, include it.



4. Pick edge 0-1: No cycle is formed, include it.



5. Pick edge 2-5: No cycle is formed, include it.



6. Pick edge 8-6: Since including this edge results in cycle, discard it.

7. Pick edge 2-3: No cycle is formed, include it.



8. Pick edge 7-8: Since including this edge results in cycle, discard it.

9. Pick edge 0-7: No cycle is formed, include it.



10. Pick edge 1-2: Since including this edge results in cycle, discard it.

11. Pick edge 3-4: No cycle is formed, include it.



Since the number of edges included equals ( $V - 1$ ), the algorithm stops here.

## C/C++

```

// C++ program for Kruskal's algorithm to find Minimum Spanning Tree
// of a given connected, undirected and weighted graph
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// a structure to represent a weighted edge in graph
struct Edge
{
 int src, dest, weight;
};

```

```

// a structure to represent a connected, undirected and weighted graph
struct Graph
{
 // V-> Number of vertices, E-> Number of edges
 int V, E;

 // graph is represented as an array of edges. Since the graph is
 // undirected, the edge from src to dest is also edge from dest
 // to src. Both are counted as 1 edge here.
 struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
 struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
 graph->V = V;
 graph->E = E;

 graph->edge = (struct Edge*) malloc(graph->E * sizeof(struct Edge));

 return graph;
}

// A structure to represent a subset for union-find
struct subset
{
 int parent;
 int rank;
};

// A utility function to find set of an element i
// (uses path compression technique)
int find(struct subset subsets[], int i)
{
 // find root and make root as parent of i (path compression)
 if (subsets[i].parent != i)
 subsets[i].parent = find(subsets, subsets[i].parent);

 return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
{
 int xroot = find(subsets, x);
 int yroot = find(subsets, y);

 // Attach smaller rank tree under root of high rank tree
 // (Union by Rank)
 if (subsets[xroot].rank < subsets[yroot].rank)
 subsets[xroot].parent = yroot;
 else if (subsets[xroot].rank > subsets[yroot].rank)
 subsets[yroot].parent = xroot;

 // If ranks are same, then make one as root and increment
 // its rank by one
 else
 {
 subsets[yroot].parent = xroot;
 subsets[xroot].rank++;
 }
}

// Compare two edges according to their weights.
// Used in qsort() for sorting an array of edges
int myComp(const void* a, const void* b)
{
 struct Edge* a1 = (struct Edge*)a;
 struct Edge* b1 = (struct Edge*)b;
 return a1->weight > b1->weight;
}

// The main function to construct MST using Kruskal's algorithm
void KruskalMST(struct Graph* graph)
{
 int V = graph->V;
 struct Edge result[V]; // This will store the resultant MST
 int e = 0; // An index variable, used for result[]
 int i = 0; // An index variable, used for sorted edges

 // Step 1: Sort all the edges in non-decreasing order of their weight
 // If we are not allowed to change the given graph, we can create a copy of
 // array of edges
 qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

 // Allocate memory for creating V ssubsets
 struct subset* subsets =
 (struct subset*) malloc(V * sizeof(struct subset));

 // Create V subsets with single elements
 for (int v = 0; v < V; ++v)
 {
 subsets[v].parent = v;
 subsets[v].rank = 0;
 }

 // Number of edges to be taken is equal to V-1
 while (e < V - 1)
 {
 // Step 2: Pick the smallest edge. And increment the index
 // for next iteration
 struct Edge next_edge = graph->edge[i++];

 int x = find(subsets, next_edge.src);
 int y = find(subsets, next_edge.dest);

 if (x != y)
 {
 result[e] = next_edge;
 e++;

 if (subsets[x].rank < subsets[y].rank)
 subsets[x].parent = y;
 else if (subsets[x].rank > subsets[y].rank)
 subsets[y].parent = x;
 else
 {
 subsets[y].parent = x;
 subsets[x].rank++;
 }
 }
 }
}

```

```

// If including this edge doesn't cause cycle, include it
// in result and increment the index of result for next edge
if (x != y)
{
 result[e++] = next_edge;
 Union(subsets, x, y);
}
// Else discard the next_edge
}

// print the contents of result[] to display the built MST
print("Following are the edges in the constructed MST\n");
for (i = 0; i < e; ++i)
 printf("%d - %d == %d\n", result[i].src, result[i].dest,
 result[i].weight);
return;
}

// Driver program to test above functions
int main()
{
 /* Let us create following weighted graph
 10
 0-----1
 | \ |
 6| 5\ |15
 | \ |
 2-----3
 4 */
 int V = 4; // Number of vertices in graph
 int E = 5; // Number of edges in graph
 struct Graph* graph = createGraph(V, E);

 // add edge 0-1
 graph->edge[0].src = 0;
 graph->edge[0].dest = 1;
 graph->edge[0].weight = 10;

 // add edge 0-2
 graph->edge[1].src = 0;
 graph->edge[1].dest = 2;
 graph->edge[1].weight = 6;

 // add edge 0-3
 graph->edge[2].src = 0;
 graph->edge[2].dest = 3;
 graph->edge[2].weight = 5;

 // add edge 1-3
 graph->edge[3].src = 1;
 graph->edge[3].dest = 3;
 graph->edge[3].weight = 15;

 // add edge 2-3
 graph->edge[4].src = 2;
 graph->edge[4].dest = 3;
 graph->edge[4].weight = 4;

 KruskalMST(graph);

 return 0;
}

```

## Java

```

// Java program for Kruskal's algorithm to find Minimum Spanning Tree
// of a given connected, undirected and weighted graph
import java.util.*;
import java.lang.*;
import java.io.*;

class Graph
{
 // A class to represent a graph edge
 class Edge implements Comparable<Edge>
 {
 int src, dest, weight;

 // Comparator function used for sorting edges based on
 // their weight
 public int compareTo(Edge compareEdge)
 {
 return this.weight - compareEdge.weight;
 }
 };

 // A class to represent a subset for union-find
 class subset
 {
 int parent, rank;
 };

 int V, E; // V-> no. of vertices & E->no.of edges
 Edge edge[]; // collection of all edges

 // Creates a graph with V vertices and E edges
 Graph(int v, int e)
 {
 V = v;
 E = e;
 edge = new Edge[E];
 for (int i=0; i<e; ++i)

```

```

 edge[i] = new Edge();
 }

 // A utility function to find set of an element i
 // (uses path compression technique)
 int find(subset subsets[], int i)
 {
 // find root and make root as parent of i (path compression)
 if (subsets[i].parent != i)
 subsets[i].parent = find(subsets, subsets[i].parent);

 return subsets[i].parent;
 }

 // A function that does union of two sets of x and y
 // (uses union by rank)
 void Union(subset subsets[], int x, int y)
 {
 int xroot = find(subsets, x);
 int yroot = find(subsets, y);

 // Attach smaller rank tree under root of high rank tree
 // (Union by Rank)
 if (subsets[xroot].rank < subsets[yroot].rank)
 subsets[xroot].parent = yroot;
 else if (subsets[xroot].rank > subsets[yroot].rank)
 subsets[yroot].parent = xroot;

 // If ranks are same, then make one as root and increment
 // its rank by one
 else
 {
 subsets[yroot].parent = xroot;
 subsets[xroot].rank++;
 }
 }

 // The main function to construct MST using Kruskal's algorithm
 void KruskalMST()
 {
 Edge result[] = new Edge[V]; // This will store the resultant MST
 int e = 0; // An index variable, used for result[]
 int i = 0; // An index variable, used for sorted edges
 for (i=0; i<V; ++i)
 result[i] = new Edge();

 // Step 1: Sort all the edges in non-decreasing order of their
 // weight. If we are not allowed to change the given graph, we
 // can create a copy of array of edges
 Arrays.sort(edge);

 // Allocate memory for creating V subsets
 subset subsets[] = new subset[V];
 for(i=0; i<V; ++i)
 subsets[i]=new subset();

 // Create V subsets with single elements
 for (int v = 0; v < V; ++v)
 {
 subsets[v].parent = v;
 subsets[v].rank = 0;
 }

 i = 0; // Index used to pick next edge

 // Number of edges to be taken is equal to V-1
 while (e < V - 1)
 {
 // Step 2: Pick the smallest edge. And increment the index
 // for next iteration
 Edge next_edge = new Edge();
 next_edge = edge[i++];

 int x = find(subsets, next_edge.src);
 int y = find(subsets, next_edge.dest);

 // If including this edge doesn't cause cycle, include it
 // in result and increment the index of result for next edge
 if (x != y)
 {
 result[e++] = next_edge;
 Union(subsets, x, y);
 }
 // Else discard the next_edge
 }

 // print the contents of result[] to display the built MST
 System.out.println("Following are the edges in the constructed MST");
 for (i = 0; i < e; ++i)
 System.out.println(result[i].src+" -- "+result[i].dest+" == "+
 result[i].weight);
 }

 // Driver Program
 public static void main (String[] args)
 {

 /* Let us create following weighted graph
 10
 0-----1
 | \ |
 6| 5\ |15
 | \ |
 2-----3
 4 */
 int V = 4; // Number of vertices in graph
 int E = 5; // Number of edges in graph
 }

```

```

Graph graph = new Graph(V, E);

// add edge 0-1
graph.edge[0].src = 0;
graph.edge[0].dest = 1;
graph.edge[0].weight = 10;

// add edge 0-2
graph.edge[1].src = 0;
graph.edge[1].dest = 2;
graph.edge[1].weight = 6;

// add edge 0-3
graph.edge[2].src = 0;
graph.edge[2].dest = 3;
graph.edge[2].weight = 5;

// add edge 1-3
graph.edge[3].src = 1;
graph.edge[3].dest = 3;
graph.edge[3].weight = 15;

// add edge 2-3
graph.edge[4].src = 2;
graph.edge[4].dest = 3;
graph.edge[4].weight = 4;

graph.KruskalMST();
}

}

//This code is contributed by Aakash Hasija

```

## Python

```

Python program for Kruskal's algorithm to find Minimum Spanning Tree
of a given connected, undirected and weighted graph

from collections import defaultdict

#Class to represent a graph
class Graph:

 def __init__(self,vertices):
 self.V= vertices #No. of vertices
 self.graph = {} # default dictionary to store graph

 # function to add an edge to graph
 def addEdge(self,u,v,w):
 self.graph[(u,v,w)] = 1

 # A utility function to find set of an element i
 # (uses path compression technique)
 def find(self, parent, i):
 if parent[i] == i:
 return i
 return self.find(parent, parent[i])

 # A function that does union of two sets of x and y
 # (uses union by rank)
 def union(self, parent, rank, x, y):
 xroot = self.find(parent, x)
 yroot = self.find(parent, y)

 # Attach smaller rank tree under root of high rank tree
 # (Union by Rank)
 if rank[xroot] < rank[yroot]:
 parent[xroot] = yroot
 elif rank[xroot] > rank[yroot]:
 parent[yroot] = xroot
 #If ranks are same, then make one as root and increment
 # its rank by one
 else :
 parent[yroot] = xroot
 rank[xroot] += 1

 # The main function to construct MST using Kruskal's algorithm
 def KruskalMST(self):

 result =[] #This will store the resultant MST

 i = 0 # An index variable, used for sorted edges
 e = 0 # An index variable, used for result[]

 #Step 1: Sort all the edges in non-decreasing order of their
 # weight. If we are not allowed to change the given graph, we
 # can create a copy of graph
 self.graph = sorted(self.graph,key=lambda item: item[2])
 #print self.graph

 parent = [] ; rank = []

 # Create V subsets with single elements
 for node in range(self.V):
 parent.append(node)
 rank.append(0)

 # Number of edges to be taken is equal to V-1
 while e < self.V - 1 :

 # Step 2: Pick the smallest edge and increment the index
 # for next iteration
 u,v,w = self.graph[i]
 i = i + 1

```

```

x = self.find(parent, u)
y = self.find(parent,v)

If including this edge doesn't cause cycle, include it
in result and increment the index of result for next edge
if x != y:
 e = e + 1
 result.append([u,v,w])
 self.union(parent, rank, x, y)
Else discard the edge

print the contents of result[] to display the built MST
print "Following are the edges in the constructed MST"
for u,v,weight in result:
 #print str(u) + " -- " + str(v) + " == " + str(weight)
 print ("%d -- %d == %d" % (u,v,weight))

g = Graph(4)
g.addEdge(0, 1, 10)
g.addEdge(0, 2, 6)
g.addEdge(0, 3, 5)
g.addEdge(1, 3, 15)
g.addEdge(2, 3, 4)

g.KruskalMST()

#This code is contributed by Neelam Yadav

```

Following are the edges in the constructed MST  
 2 -- 3 == 4  
 0 -- 3 == 5  
 0 -- 1 == 10

**Time Complexity:**  $O(E\log E)$  or  $O(E\log V)$ . Sorting of edges takes  $O(E\log E)$  time. After sorting, we iterate through all edges and apply find-union algorithm. The find and union operations can take atmost  $O(\log V)$  time. So overall complexity is  $O(E\log E + E\log V)$  time. The value of E can be atmost  $O(V^2)$ , so  $O(\log V)$  are  $O(\log E)$  same. Therefore, overall time complexity is  $O(E\log E)$  or  $O(E\log V)$

#### References:

<http://www.ics.uci.edu/~eppstein/161/960206.html>  
[http://en.wikipedia.org/wiki/Minimum\\_spanning\\_tree](http://en.wikipedia.org/wiki/Minimum_spanning_tree)

This article is compiled by Aashish Barnwal and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner Company Wise Coding Practice

Graph  
 Greedy  
 Graph  
 Greedy Algorithm  
 Kruskal  
 Kruskal'sAlgorithm  
 MST

## Greedy Algorithms | Set 9 (Boruvka's algorithm)

We have discussed following topics on Minimum Spanning Tree.

[Applications of Minimum Spanning Tree Problem](#)  
[Kruskal's Minimum Spanning Tree Algorithm](#)  
[Prim's Minimum Spanning Tree Algorithm](#)

In this post, Boruvka's algorithm is discussed. Like Prim's and Kruskal's, Boruvka's algorithm is also a Greedy algorithm. Below is complete algorithm.

- 1) Input is a connected, weighted and directed graph.
- 2) Initialize all vertices as individual components (or sets).
- 3) Initialize MST as empty.
- 4) While there are more than one components, do following for each component.
  - a) Find the closest weight edge that connects this component to any other component.
  - b) Add this closest edge to MST if not already added.
- 5) Return MST.

Below is the idea behind above algorithm (The idea is same as [Prim's MST algorithm](#)).

A spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a Spanning Tree. And they must be connected with the minimum weight edge to make it a Minimum Spanning Tree.

Let us understand the algorithm with below example.



Initially MST is empty. Every vertex is single component as highlighted in blue color in below diagram.



For every component, find the cheapest edge that connects it to some other component.

| Component | Cheapest Edge that connects it to some other component |
|-----------|--------------------------------------------------------|
|-----------|--------------------------------------------------------|

```

{0} 0-1
{1} 0-1
{2} 2-8
{3} 2-3
{4} 3-4
{5} 5-6
{6} 6-7
{7} 6-7
{8} 2-8

```

The cheapest edges are highlighted with green color. Now MST becomes {0-1, 2-8, 2-3, 3-4, 5-6, 6-7}.



After above step, components are {{0,1}, {2,3,4,8}, {5,6,7}}. The components are encircled with blue color.



We again repeat the step, i.e., for every component, find the cheapest edge that connects it to some other component.

| Component | Cheapest Edge that connects<br>it to some other component |
|-----------|-----------------------------------------------------------|
| {0,1}     | 1-2 (or 0-7)                                              |
| {2,3,4,8} | 2-5                                                       |
| {5,6,7}   | 2-5                                                       |

The cheapest edges are highlighted with green color. Now MST becomes {0-1, 2-8, 2-3, 3-4, 5-6, 6-7, 1-2, 2-5}.



At this stage, there is only one component {0, 1, 2, 3, 4, 5, 6, 7, 8} which has all edges. Since there is only one component left, we stop and return MST.

#### Implementation:

Below is implementation of above algorithm. The input graph is represented as a collection of edges and [union-find data structure](#) is used to keep track of components.

#### C/C++

```

// Boruvka's algorithm to find Minimum Spanning
// Tree of a given connected, undirected and
// weighted graph
#include <stdio.h>

// a structure to represent a weighted edge in graph
struct Edge
{
 int src, dest, weight;
};

// a structure to represent a connected, undirected
// and weighted graph as a collection of edges.
struct Graph
{
 // V-> Number of vertices, E-> Number of edges
 int V, E;

 // graph is represented as an array of edges.
 // Since the graph is undirected, the edge
 // from src to dest is also edge from dest
 // to src. Both are counted as 1 edge here.
 Edge* edge;
};

// A structure to represent a subset for union-find
struct subset
{
 int parent;
 int rank;
};

// Function prototypes for union-find (These functions are defined
// after boruvkaMST())
int find(struct subset subsets[], int i);
void Union(struct subset subsets[], int x, int y);

// The main function for MST using Boruvka's algorithm
void boruvkaMST(struct Graph* graph)
{
 // Get data of given graph
 int V = graph->V, E = graph->E;
 Edge *edge = graph->edge;

 // Allocate memory for creating V subsets.
 struct subset *subsets = new subset[V];

 // An array to store index of the cheapest edge of
 // subset. The stored index for indexing array 'edge[]'

```

```

int *cheapest = new int[V];

// Create V subsets with single elements
for (int v = 0; v < V; ++v)
{
 subsets[v].parent = v;
 subsets[v].rank = 0;
 cheapest[v] = -1;
}

// Initially there are V different trees.
// Finally there will be one tree that will be MST
int numTrees = V;
int MSTweight = 0;

// Keep combining components (or sets) until all
// components are not combined into single MST.
while (numTrees > 1)
{
 // Traverse through all edges and update
 // cheapest of every component
 for (int i=0; i<E; i++)
 {
 // Find components (or sets) of two corners
 // of current edge
 int set1 = find(subsets, edge[i].src);
 int set2 = find(subsets, edge[i].dest);

 // If two corners of current edge belong to
 // same set, ignore current edge
 if (set1 == set2)
 continue;

 // Else check if current edge is closer to previous
 // cheapest edges of set1 and set2
 else
 {
 if (cheapest[set1] == -1 ||
 edge[cheapest[set1]].weight > edge[i].weight)
 cheapest[set1] = i;

 if (cheapest[set1] == -1 ||
 edge[cheapest[set2]].weight > edge[i].weight)
 cheapest[set2] = i;
 }
 }

 // Consider the above picked cheapest edges and add them
 // to MST
 for (int i=0; i<V; i++)
 {
 // Check if cheapest for current set exists
 if (cheapest[i] != -1)
 {
 int set1 = find(subsets, edge[cheapest[i]].src);
 int set2 = find(subsets, edge[cheapest[i]].dest);

 if (set1 == set2)
 continue;
 MSTweight += edge[cheapest[i]].weight;
 print("Edge %d-%d included in MST\n",
 edge[cheapest[i]].src, edge[cheapest[i]].dest,
 edge[cheapest[i]].weight);

 // Do a union of set1 and set2 and decrease number
 // of trees
 Union(subsets, set1, set2);
 numTrees--;
 }
 }
}

printf("Weight of MST is %d\n", MSTweight);
return;
}

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
 Graph* graph = new Graph;
 graph->V = V;
 graph->E = E;
 graph->edge = new Edge[E];
 return graph;
}

// A utility function to find set of an element i
// (uses path compression technique)
int find(struct subset subsets[], int i)
{
 // find root and make root as parent of i
 // (path compression)
 if (subsets[i].parent != i)
 subsets[i].parent =
 find(subsets, subsets[i].parent);

 return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
{
 int xroot = find(subsets, x);
 int yroot = find(subsets, y);

 // Attach smaller rank tree under root of high

```

```

// rank tree (Union by Rank)
if (subsets[xroot].rank < subsets[yroot].rank)
 subsets[xroot].parent = yroot;
else if (subsets[xroot].rank > subsets[yroot].rank)
 subsets[yroot].parent = xroot;

// If ranks are same, then make one as root and
// increment its rank by one
else
{
 subsets[yroot].parent = xroot;
 subsets[xroot].rank++;
}

// Driver program to test above functions
int main()
{
 /* Let us create following weighted graph
 10
 0-----1
 | \ |
 6| 5\ |15
 | \| |
 2-----3
 4 */
 int V = 4; // Number of vertices in graph
 int E = 5; // Number of edges in graph
 struct Graph* graph = createGraph(V, E);

 // add edge 0-1
 graph->edge[0].src = 0;
 graph->edge[0].dest = 1;
 graph->edge[0].weight = 10;

 // add edge 0-2
 graph->edge[1].src = 0;
 graph->edge[1].dest = 2;
 graph->edge[1].weight = 6;

 // add edge 0-3
 graph->edge[2].src = 0;
 graph->edge[2].dest = 3;
 graph->edge[2].weight = 5;

 // add edge 1-3
 graph->edge[3].src = 1;
 graph->edge[3].dest = 3;
 graph->edge[3].weight = 15;

 // add edge 2-3
 graph->edge[4].src = 2;
 graph->edge[4].dest = 3;
 graph->edge[4].weight = 4;

 boruvkaMST(graph);

 return 0;
}

```

## Python

```

Boruvka's algorithm to find Minimum Spanning
Tree of a given connected, undirected and weighted graph

from collections import defaultdict

#Class to represent a graph
class Graph:

 def __init__(self,vertices):
 self.v= vertices #No. of vertices
 self.graph = [] # default dictionary to store graph

 # function to add an edge to graph
 def addEdge(self,u,v,w):
 self.graph.append([u,v,w])

 # A utility function to find set of an element i
 # (uses path compression technique)
 def find(self, parent, i):
 if parent[i] == i:
 return i
 return self.find(parent, parent[i])

 # A function that does union of two sets of x and y
 # (uses union by rank)
 def union(self, parent, rank, x, y):
 xroot = self.find(parent, x)
 yroot = self.find(parent, y)

 # Attach smaller rank tree under root of high rank tree
 # (Union by Rank)
 if rank[xroot] < rank[yroot]:
 parent[xroot] = yroot
 elif rank[xroot] > rank[yroot]:
 parent[yroot] = xroot
 #If ranks are same, then make one as root and increment
 # its rank by one
 else :
 parent[yroot] = xroot
 rank[xroot] += 1

```

```

The main function to construct MST using Kruskal's algorithm
def boruvkaMST(self):
 parent = []
 rank = []

 # An array to store index of the cheapest edge of
 # subset. It store [u,v,w] for each component
 cheapest = []

 # Initially there are V different trees.
 # Finally there will be one tree that will be MST
 numTrees = self.V
 MSTweight = 0

 # Create V subsets with single elements
 for node in range(self.V):
 parent.append(node)
 rank.append(0)
 cheapest = [-1] * self.V

 # Keep combining components (or sets) until all
 # components are not combined into single MST

 while numTrees > 1:

 # Traverse through all edges and update
 # cheapest of every component
 for i in range(len(self.graph)):

 # Find components (or sets) of two corners
 # of current edge
 u, v, w = self.graph[i]
 set1 = self.find(parent, u)
 set2 = self.find(parent, v)

 # If two corners of current edge belong to
 # same set, ignore current edge. Else check if
 # current edge is closer to previous
 # cheapest edges of set1 and set2
 if set1 != set2:

 if cheapest[set1] == -1 or cheapest[set1][2] > w:
 cheapest[set1] = [u, v, w]

 if cheapest[set2] == -1 or cheapest[set2][2] > w:
 cheapest[set2] = [u, v, w]

 # Consider the above picked cheapest edges and add them
 # to MST
 for node in range(self.V):

 #Check if cheapest for current set exists
 if cheapest[node] != -1:
 u, v, w = cheapest[node]
 set1 = self.find(parent, u)
 set2 = self.find(parent, v)

 if set1 != set2:
 MSTweight += w
 self.union(parent, rank, set1, set2)
 print("Edge %d-%d with weight %d included in MST" % (u, v, w))
 numTrees = numTrees - 1

 #reset cheapest array
 cheapest = [-1] * self.V

 print("Weight of MST is %d" % MSTweight)

g = Graph(4)
g.addEdge(0, 1, 10)
g.addEdge(0, 2, 6)
g.addEdge(0, 3, 5)
g.addEdge(1, 3, 15)
g.addEdge(2, 3, 4)

g.boruvkaMST()

#This code is contributed by Neelam Yadav

```

Output:

```

Edge 0-3 included in MST
Edge 0-1 included in MST
Edge 2-3 included in MST
Weight of MST is 19

```

#### Interesting Facts about Boruvka's algorithm:

1) Time Complexity of Boruvka's algorithm is  $O(E \log V)$  which is same as Kruskal's and Prim's algorithms.

2) Boruvka's algorithm is used as a step in a **faster randomized algorithm that works in linear time  $O(E)$** .

3) Boruvka's algorithm is the oldest minimum spanning tree algorithm was discovered by Boruvka in 1926, long before computers even existed. The algorithm was published as a method of constructing an efficient electricity network.

#### Exercise:

The above code assumes that input graph is connected and it fails if a disconnected graph is given. Extend the above algorithm so that it works for a disconnected graph also and produces a forest.

#### References:

[http://en.wikipedia.org/wiki/Bor%C5%AFvka%27s\\_algorithm](http://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Greedy Algorithms | Set 7 (Dijkstra's shortest path algorithm)

Given a graph and a source vertex in graph, find shortest paths from source to all vertices in the given graph.

Dijkstra's algorithm is very similar to [Prim's algorithm for minimum spanning tree](#). Like Prim's MST, we generate a *SPT* (*shortest path tree*) with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has minimum distance from source.

Below are the detailed steps used in Dijkstra's algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Algorithm

1) Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as **INFINITE**. Assign distance value as 0 for the source vertex so that it is picked first.

3) While *sptSet* doesn't include all vertices

....a) Pick a vertex *u* which is not there in *sptSet* and has minimum distance value.

....b) Include *u* to *sptSet*.

....c) Update distance value of all adjacent vertices of *u*. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex *v*, if sum of distance value of *u* (from source) and weight of edge *u-v*, is less than the distance value of *v*, then update the distance value of *v*.

Let us understand with the following example:



The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green color.



Pick the vertex with minimum distance value and not already included in SPT (not in *sptSET*). The vertex 1 is picked and added to *sptSet*. So *sptSet* now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



Pick the vertex with minimum distance value and not already included in SPT (not in *sptSET*). Vertex 7 is picked. So *sptSet* now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance value and not already included in SPT (not in *sptSET*). Vertex 6 is picked. So *sptSet* now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until *sptSet* doesn't include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).



**How to implement the above algorithm?**

We use a boolean array *sptSet[]* to represent the set of vertices included in SPT. If a value *sptSet[v]* is true, then vertex *v* is included in SPT, otherwise not. Array *dist[]* is used to store shortest distance values of all vertices.

**C/C++**

```
// A C / C++ program for Dijkstra's single source shortest path algorithm.
// The program is for adjacency matrix representation of the graph
```

```

#include <stdio.h>
#include <limits.h>

// Number of vertices in the graph
#define V 9

// A utility function to find the vertex with minimum distance value, from
// the set of vertices not yet included in shortest path tree
int minDistance(int dist[], bool sptSet[])
{
 // Initialize min value
 int min = INT_MAX, min_index;

 for (int v = 0; v < V; v++)
 if (sptSet[v] == false && dist[v] <= min)
 min = dist[v], min_index = v;

 return min_index;
}

// A utility function to print the constructed distance array
int printSolution(int dist[], int n)
{
 printf("Vertex Distance from Source\n");
 for (int i = 0; i < V; i++)
 printf("%d \t %d\n", i, dist[i]);
}

// Function that implements Dijkstra's single source shortest path algorithm
// for a graph represented using adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
 int dist[V]; // The output array. dist[i] will hold the shortest
 // distance from src to i

 bool sptSet[V]; // sptSet[i] will true if vertex i is included in shortest
 // path tree or shortest distance from src to i is finalized

 // Initialize all distances as INFINITE and sptSet[] as false
 for (int i = 0; i < V; i++)
 dist[i] = INT_MAX, sptSet[i] = false;

 // Distance of source vertex from itself is always 0
 dist[src] = 0;

 // Find shortest path for all vertices
 for (int count = 0; count < V - 1; count++)
 {
 // Pick the minimum distance vertex from the set of vertices not
 // yet processed. u is always equal to src in first iteration.
 int u = minDistance(dist, sptSet);

 // Mark the picked vertex as processed
 sptSet[u] = true;

 // Update dist value of the adjacent vertices of the picked vertex.
 for (int v = 0; v < V; v++)

 // Update dist[v] only if it is not in sptSet, there is an edge from
 // u to v, and total weight of path from src to v through u is
 // smaller than current value of dist[v]
 if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
 && dist[u] + graph[u][v] < dist[v])
 dist[v] = dist[u] + graph[u][v];
 }

 // print the constructed distance array
 printSolution(dist, V);
}

// driver program to test above function
int main()
{
 /* Let us create the example graph discussed above */
 int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},
 {4, 0, 8, 0, 0, 0, 11, 0},
 {0, 8, 0, 7, 0, 4, 0, 0, 2},
 {0, 0, 7, 0, 9, 14, 0, 0, 0},
 {0, 0, 0, 9, 0, 10, 0, 0, 0},
 {0, 0, 4, 14, 10, 0, 2, 0, 0},
 {0, 0, 0, 0, 2, 0, 1, 6},
 {8, 11, 0, 0, 0, 0, 1, 0, 7},
 {0, 0, 2, 0, 0, 0, 6, 7, 0}
 };

 dijkstra(graph, 0);

 return 0;
}

```

## Java

```

// A Java program for Dijkstra's single source shortest path algorithm.
// The program is for adjacency matrix representation of the graph
import java.util.*;
import java.lang.*;
import java.io.*;

class ShortestPath
{
 // A utility function to find the vertex with minimum distance value,
 // from the set of vertices not yet included in shortest path tree
 static final int V=9;
 int minDistance(int dist[], Boolean sptSet[])

```

```

{
 // Initialize min value
 int min = Integer.MAX_VALUE, min_index=-1;

 for (int v = 0; v < V; v++)
 if (sptSet[v] == false && dist[v] <= min)
 {
 min = dist[v];
 min_index = v;
 }

 return min_index;
}

// A utility function to print the constructed distance array
void printSolution(int dist[], int n)
{
 System.out.println("Vertex Distance from Source");
 for (int i = 0; i < V; i++)
 System.out.println(i+" "+dist[i]);
}

// Function that implements Dijkstra's single source shortest path
// algorithm for a graph represented using adjacency matrix
// representation
void dijkstra(int graph[][], int src)
{
 int dist[] = new int[V]; // The output array. dist[i] will hold
 // the shortest distance from src to i

 // sptSet[i] will true if vertex i is included in shortest
 // path tree or shortest distance from src to i is finalized
 Boolean sptSet[] = new Boolean[V];

 // Initialize all distances as INFINITE and sptSet[] as false
 for (int i = 0; i < V; i++)
 {
 dist[i] = Integer.MAX_VALUE;
 sptSet[i] = false;
 }

 // Distance of source vertex from itself is always 0
 dist[src] = 0;

 // Find shortest path for all vertices
 for (int count = 0; count < V-1; count++)
 {
 // Pick the minimum distance vertex from the set of vertices
 // not yet processed. u is always equal to src in first
 // iteration.
 int u = minDistance(dist, sptSet);

 // Mark the picked vertex as processed
 sptSet[u] = true;

 // Update dist value of the adjacent vertices of the
 // picked vertex.
 for (int v = 0; v < V; v++)

 // Update dist[v] only if it is not in sptSet, there is an
 // edge from u to v, and total weight of path from src to
 // v through u is smaller than current value of dist[v]
 if (!sptSet[v] && graph[u][v]!=0 &&
 dist[u] != Integer.MAX_VALUE &&
 dist[u]+graph[u][v] < dist[v])
 dist[v] = dist[u] + graph[u][v];
 }

 // print the constructed distance array
 printSolution(dist, V);
}

// Driver method
public static void main (String[] args)
{
 /* Let us create the example graph discussed above */
 int graph[][] = new int[][]{{0, 4, 0, 0, 0, 0, 0, 8, 0},
 {4, 0, 8, 0, 0, 0, 0, 11, 0},
 {0, 8, 0, 7, 0, 4, 0, 0, 2},
 {0, 0, 7, 0, 9, 14, 0, 0, 0},
 {0, 0, 0, 9, 0, 10, 0, 0, 0},
 {0, 0, 4, 14, 10, 0, 2, 0, 0},
 {0, 0, 0, 0, 2, 0, 1, 6},
 {8, 11, 0, 0, 0, 0, 1, 0, 7},
 {0, 0, 2, 0, 0, 0, 6, 7, 0}
 };
 ShortestPath t = new ShortestPath();
 t.dijkstra(graph, 0);
}
}

//This code is contributed by Aakash Hasija

```

Output:

| Vertex | Distance from Source |
|--------|----------------------|
| 0      | 0                    |
| 1      | 4                    |
| 2      | 12                   |
| 3      | 19                   |
| 4      | 21                   |
| 5      | 11                   |
| 6      | 9                    |
| 7      | 8                    |
| 8      | 14                   |

Notes:

- 1) The code calculates shortest distance, but doesn't calculate the path information. We can create a parent array, update the parent array when distance is updated (like prim's implementation) and use it to show the shortest path from source to different vertices.
  - 2) The code is for undirected graph, same dijkstra function can be used for directed graphs also.
  - 3) The code finds shortest distances from source to all vertices. If we are interested only in shortest distance from source to a single target, we can break the for loop when the picked minimum distance vertex is equal to target (Step 3.a of algorithm).
  - 4) Time Complexity of the implementation is  $O(V^2)$ . If the input graph is represented using adjacency list, it can be reduced to  $O(E \log V)$  with the help of binary heap. Please see [Dijkstra's Algorithm for Adjacency List Representation](#) for more details.
  - 5) Dijkstra's algorithm doesn't work for graphs with negative weight edges. For graphs with negative weight edges, [Bellman–Ford algorithm](#) can be used, we will soon be discussing it as a separate post.
- [Dijkstra's Algorithm for Adjacency List Representation](#)
- [Printing Paths in Dijkstra's Shortest Path Algorithm](#)
- [Dijkstra's shortest path algorithm using set in STL](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner Company Wise Coding Practice

```
Graph
Greedy
Dijkstra
Graph
Greedy Algorithm
shortest path
```

---

## Greedy Algorithms | Set 8 (Dijkstra's Algorithm for Adjacency List Representation)

We recommend to read following two posts as a prerequisite of this post.

1. [Greedy Algorithms | Set 7 \(Dijkstra's shortest path algorithm\)](#)
2. [Graph and its representations](#)

We have discussed Dijkstra's algorithm and its implementation for adjacency matrix representation of graphs. The time complexity for the matrix representation is  $O(V^2)$ . In this post,  $O(E \log V)$  algorithm for adjacency list representation is discussed.

As discussed in the previous post, in Dijkstra's algorithm, two sets are maintained, one set contains list of vertices already included in SPT (Shortest Path Tree), other set contains vertices not yet included. With adjacency list representation, all vertices of a graph can be traversed in  $O(V+E)$  time using [BFS](#). The idea is to traverse all vertices of graph using [BFS](#) and use a Min Heap to store the vertices not yet included in SPT (or the vertices for which shortest distance is not finalized yet). Min Heap is used as a priority queue to get the minimum distance vertex from set of not yet included vertices. Time complexity of operations like extract-min and decrease-key value is  $O(\log V)$  for Min Heap.

Following are the detailed steps.

- 1) Create a Min Heap of size  $V$  where  $V$  is the number of vertices in the given graph. Every node of min heap contains vertex number and distance value of the vertex.
- 2) Initialize Min Heap with source vertex as root (the distance value assigned to source vertex is 0). The distance value assigned to all other vertices is INF (infinite).
- 3) While Min Heap is not empty, do following
  - Extract the vertex with minimum distance value node from Min Heap. Let the extracted vertex be  $u$ .
  - For every adjacent vertex  $v$  of  $u$ , check if  $v$  is in Min Heap. If  $v$  is in Min Heap and distance value is more than weight of  $u-v$  plus distance value of  $u$ , then update the distance value of  $v$ .

Let us understand with the following example. Let the given source vertex be 0



Initially, distance value of source vertex is 0 and INF (infinite) for all other vertices. So source vertex is extracted from Min Heap and distance values of vertices adjacent to 0 (1 and 7) are updated. Min Heap contains all vertices except vertex 0.

The vertices in green color are the vertices for which minimum distances are finalized and are not in Min Heap



Since distance value of vertex 1 is minimum among all nodes in Min Heap, it is extracted from Min Heap and distance values of vertices adjacent to 1 are updated (distance is updated if the a vertex is not in Min Heap and distance through 1 is shorter than the previous distance). Min Heap contains all vertices except vertex 0 and 1.



Pick the vertex with minimum distance value from min heap. Vertex 7 is picked. So min heap now contains all vertices except 0, 1 and 7. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance from min heap. Vertex 6 is picked. So min heap now contains all vertices except 0, 1, 7 and 6. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



Above steps are repeated till min heap doesn't become empty. Finally, we get the following shortest path tree.



```
// C / C++ program for Dijkstra's shortest path algorithm for adjacency
// list representation of graph

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a node in adjacency list
struct AdjListNode
{
 int dest;
 int weight;
 struct AdjListNode* next;
};

// A structure to represent an adjacency list
struct AdjList
{
 struct AdjListNode *head; // pointer to head node of list
};

// A structure to represent a graph. A graph is an array of adjacency lists.
// Size of array will be V (number of vertices in graph)
struct Graph
{
 int V;
 struct AdjList* array;
};

// A utility function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest, int weight)
{
 struct AdjListNode* newNode =
 (struct AdjListNode*) malloc(sizeof(struct AdjListNode));
 newNode->dest = dest;
 newNode->weight = weight;
 newNode->next = NULL;
 return newNode;
}

// A utility function that creates a graph of V vertices
struct Graph* createGraph(int V)
{
 struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
 graph->V = V;

 // Create an array of adjacency lists. Size of array will be V
 graph->array = (struct AdjList*) malloc(V * sizeof(struct AdjList));

 // Initialize each adjacency list as empty by making head as NULL
 for (int i = 0; i < V; ++i)
 graph->array[i].head = NULL;
}

// Adds an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest, int weight)
{
 // Add an edge from src to dest. A new node is added to the adjacency
 // list of src. The node is added at the beginning
 struct AdjListNode* newNode = newAdjListNode(dest, weight);
 newNode->next = graph->array[src].head;
 graph->array[src].head = newNode;
}
```

```

// Since graph is undirected, add an edge from dest to src also
newNode = newAdjListNode(src, weight);
newNode->next = graph->array[dest].head;
graph->array[dest].head = newNode;
}

// Structure to represent a min heap node
struct MinHeapNode
{
 int v;
 int dist;
};

// Structure to represent a min heap
struct MinHeap
{
 int size; // Number of heap nodes present currently
 int capacity; // Capacity of min heap
 int *pos; // This is needed for decreaseKey()
 struct MinHeapNode **array;
};

// A utility function to create a new Min Heap Node
struct MinHeapNode* newMinHeapNode(int v, int dist)
{
 struct MinHeapNode* minHeapNode =
 (struct MinHeapNode*) malloc(sizeof(struct MinHeapNode));
 minHeapNode->v = v;
 minHeapNode->dist = dist;
 return minHeapNode;
}

// A utility function to create a Min Heap
struct MinHeap* createMinHeap(int capacity)
{
 struct MinHeap* minHeap =
 (struct MinHeap*) malloc(sizeof(struct MinHeap));
 minHeap->pos = (int *)malloc(capacity * sizeof(int));
 minHeap->size = 0;
 minHeap->capacity = capacity;
 minHeap->array =
 (struct MinHeapNode**) malloc(capacity * sizeof(struct MinHeapNode*));
 return minHeap;
}

// A utility function to swap two nodes of min heap. Needed for min heapify
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b)
{
 struct MinHeapNode* t = *a;
 *a = *b;
 *b = t;
}

// A standard function to heapify at given idx
// This function also updates position of nodes when they are swapped.
// Position is needed for decreaseKey()
void minHeapify(struct MinHeap* minHeap, int idx)
{
 int smallest, left, right;
 smallest = idx;
 left = 2 * idx + 1;
 right = 2 * idx + 2;

 if (left < minHeap->size &&
 minHeap->array[left]->dist < minHeap->array[smallest]->dist)
 smallest = left;

 if (right < minHeap->size &&
 minHeap->array[right]->dist < minHeap->array[smallest]->dist)
 smallest = right;

 if (smallest != idx)
 {
 // The nodes to be swapped in min heap
 MinHeapNode *smallestNode = minHeap->array[smallest];
 MinHeapNode *idxNode = minHeap->array[idx];

 // Swap positions
 minHeap->pos[smallestNode->v] = idx;
 minHeap->pos[idxNode->v] = smallest;

 // Swap nodes
 swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

 minHeapify(minHeap, smallest);
 }
}

// A utility function to check if the given minHeap is ampty or not
int isEmpty(struct MinHeap* minHeap)
{
 return minHeap->size == 0;
}

// Standard function to extract minimum node from heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap)
{
 if (isEmpty(minHeap))
 return NULL;

 // Store the root node
 struct MinHeapNode* root = minHeap->array[0];

 // Replace root node with last node
 struct MinHeapNode* lastNode = minHeap->array[minHeap->size - 1];
 minHeap->array[0] = lastNode;
}

```

```

// Update position of last node
minHeap->pos[root->v] = minHeap->size-1;
minHeap->pos[astNode->v] = 0;

// Reduce heap size and heapify root
--minHeap->size;
minHeapify(minHeap, 0);

return root;
}

// Function to decrease dist value of a given vertex v. This function
// uses pos[] of min heap to get the current index of node in min heap
void decreaseKey(struct MinHeap* minHeap, int v, int dist)
{
 // Get the index of v in heap array
 int i = minHeap->pos[v];

 // Get the node and update its dist value
 minHeap->array[i]->dist = dist;

 // Travel up while the complete tree is not heified.
 // This is a O(Logn) loop
 while (i && minHeap->array[i]->dist < minHeap->array[(i - 1) / 2]->dist)
 {
 // Swap this node with its parent
 minHeap->pos[minHeap->array[i]->v] = (i-1)/2;
 minHeap->pos[minHeap->array[(i-1)/2]->v] = i;
 swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

 // move to parent index
 i = (i - 1) / 2;
 }
}

// A utility function to check if a given vertex
// V is in min heap or not
bool isInMinHeap(struct MinHeap *minHeap, int v)
{
 if (minHeap->pos[v] < minHeap->size)
 return true;
 return false;
}

// A utility function used to print the solution
void printArr(int dist[], int n)
{
 printf("Vertex Distance from Source\n");
 for (int i = 0; i < n; ++i)
 printf("%d \t %d\n", i, dist[i]);
}

// The main function that calculates distances of shortest paths from src to all
// vertices. It is a O(ELogV) function
void dijkstra(struct Graph* graph, int src)
{
 int V = graph->V;// Get the number of vertices in graph
 int dist[V]; // dist values used to pick minimum weight edge in cut

 // minHeap represents set E
 struct MinHeap* minHeap = createMinHeap(V);

 // Initialize min heap with all vertices. dist value of all vertices
 for (int v = 0; v < V; ++v)
 {
 dist[v] = INT_MAX;
 minHeap->array[v] = newMinHeapNode(v, dist[v]);
 minHeap->pos[v] = v;
 }

 // Make dist value of src vertex as 0 so that it is extracted first
 minHeap->array[src] = newMinHeapNode(src, dist[src]);
 minHeap->pos[src] = src;
 dist[src] = 0;
 decreaseKey(minHeap, src, dist[src]);

 // Initially size of min heap is equal to V
 minHeap->size = V;

 // In the followin loop, min heap contains all nodes
 // whose shortest distance is not yet finalized.
 while (!isEmpty(minHeap))
 {
 // Extract the vertex with minimum distance value
 struct MinHeapNode* minHeapNode = extractMin(minHeap);
 int u = minHeapNode->v;// Store the extracted vertex number

 // Traverse through all adjacent vertices of u (the extracted
 // vertex) and update their distance values
 struct AdjListNode* pCrawl = graph->array[u].head;
 while (pCrawl != NULL)
 {
 int v = pCrawl->dest;

 // If shortest distance to v is not finalized yet, and distance to v
 // through u is less than its previously calculated distance
 if (isInMinHeap(minHeap, v) && dist[u] != INT_MAX &&
 pCrawl->weight + dist[u] < dist[v])
 {
 dist[v] = dist[u] + pCrawl->weight;

 // update distance value in min heap also
 decreaseKey(minHeap, v, dist[v]);
 }
 pCrawl = pCrawl->next;
 }
 }
}

```

```

 // print the calculated shortest distances
 printArr(dist, V);
}

// Driver program to test above functions
int main()
{
 // create the graph given in above figure
 int V = 9;
 struct Graph* graph = createGraph(V);
 addEdge(graph, 0, 1, 4);
 addEdge(graph, 0, 7, 8);
 addEdge(graph, 1, 2, 8);
 addEdge(graph, 1, 7, 11);
 addEdge(graph, 2, 3, 7);
 addEdge(graph, 2, 8, 2);
 addEdge(graph, 2, 5, 4);
 addEdge(graph, 3, 4, 9);
 addEdge(graph, 3, 5, 14);
 addEdge(graph, 4, 5, 10);
 addEdge(graph, 5, 6, 2);
 addEdge(graph, 6, 7, 1);
 addEdge(graph, 6, 8, 6);
 addEdge(graph, 7, 8, 7);

 dijkstra(graph, 0);

 return 0;
}

```

Output:

| Vertex | Distance from Source |
|--------|----------------------|
| 0      | 0                    |
| 1      | 4                    |
| 2      | 12                   |
| 3      | 19                   |
| 4      | 21                   |
| 5      | 11                   |
| 6      | 9                    |
| 7      | 8                    |
| 8      | 14                   |

**Time Complexity:** The time complexity of the above code/algorithm looks  $O(V^2)$  as there are two nested while loops. If we take a closer look, we can observe that the statements in inner loop are executed  $O(V+E)$  times (similar to BFS). The inner loop has `decreaseKey()` operation which takes  $O(\log V)$  time. So overall time complexity is  $O(E+V) \cdot O(\log V)$  which is  $O((E+V) \cdot \log V) = O(E \log V)$ . Note that the above code uses Binary Heap for Priority Queue implementation. Time complexity can be reduced to  $O(E + V \log V)$  using Fibonacci Heap. The reason is, Fibonacci Heap takes  $O(1)$  time for decrease-key operation while Binary Heap takes  $O(\log n)$  time.

#### Notes:

- 1) The code calculates shortest distance, but doesn't calculate the path information. We can create a parent array, update the parent array when distance is updated (like prim's implementation) and use it to show the shortest path from source to different vertices.
- 2) The code is for undirected graph, same dijkstra function can be used for directed graphs also.
- 3) The code finds shortest distances from source to all vertices. If we are interested only in shortest distance from source to a single target, we can break the for loop when the picked minimum distance vertex is equal to target (Step 3.a of algorithm).
- 4) Dijkstra's algorithm doesn't work for graphs with negative weight edges. For graphs with negative weight edges, Bellman–Ford algorithm can be used, we will soon be discussing it as a separate post.

[Printing Paths in Dijkstra's Shortest Path Algorithm](#)

[Dijkstra's shortest path algorithm using set in STL](#)

#### References:

[Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](#)  
[Algorithms by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner Company Wise Coding Practice

Graph  
Greedy  
Dijkstra  
shortest path

## Dynamic Programming | Set 23 (Bellman–Ford Algorithm)

Given a graph and a source vertex `src` in graph, find shortest paths from `src` to all vertices in the given graph. The graph may contain negative weight edges.

We have discussed [Dijkstra's algorithm](#) for this problem. Dijkstra's algorithm is a Greedy algorithm and time complexity is  $O(V \log V)$  (with the use of Fibonacci heap). [Dijkstra doesn't work for Graphs with negative weight edges](#), [Bellman–Ford works for such graphs](#). [Bellman–Ford is also simpler than Dijkstra and suites well for distributed systems](#). But time complexity of Bellman–Ford is  $O(VE)$ , which is more than Dijkstra.

#### Algorithm

Following are the detailed steps.

**Input:** Graph and a source vertex `src`

**Output:** Shortest distance to all vertices from `src`. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

- 1) This step initializes distances from source to all vertices as infinite and distance to source itself as 0. Create an array `dist[]` of size  $|V|$  with all values as infinite except `dist[src]` where `src` is source vertex.
- 2) This step calculates shortest distances. Do following  $|V|-1$  times where  $|V|$  is the number of vertices in given graph.
  - a) Do following for each edge  $u-v$ 
    - .....If  $dist[v] > dist[u] + \text{weight of edge } uv$ , then update  $dist[v]$
    - ..... $dist[v] = dist[u] + \text{weight of edge } uv$
- 3) This step reports if there is a negative weight cycle in graph. Do following for each edge  $u-v$ 
  - .....If  $dist[v] > dist[u] + \text{weight of edge } uv$ , then "Graph contains negative weight cycle"

The idea of step 3 is, step 2 guarantees shortest distances if graph doesn't contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

**How does this work?** Like other Dynamic Programming Problems, the algorithm calculate shortest paths in bottom-up manner. It first calculates the shortest distances for the shortest paths which have at-

most one edge in the path. Then, it calculates shortest paths with at-most 2 edges, and so on. After the  $i$ th iteration of outer loop, the shortest paths with at most  $i$  edges are calculated. There can be maximum  $|V| - 1$  edges in any simple path, that is why the outer loop runs  $|V| - 1$  times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most  $i$  edges, then an iteration over all edges guarantees to give shortest path with at-most  $(i+1)$  edges (Proof is simple, you can refer [this](#) or [MIT Video Lecture](#))

#### Example

Let us understand the algorithm with following example graph. The images are taken from [this](#) source.

Let the given source vertex be 0. Initialize all distances as infinite, except the distance to source itself. Total number of vertices in the graph is 5, so *all edges must be processed 4 times*.

#### Example Graph



Let all edges are processed in following order: (B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D). We get following distances when all edges are processed first time. The first row in shows initial distances. The second row shows distances when edges (B,E), (D,B), (B,D) and (A,B) are processed. The third row shows distances when (A,C) is processed. The fourth row shows when (D,C), (B,C) and (E,D) are processed.



The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get following distances when all edges are processed second time (The last row shows final values).



The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.

#### Implementation:

#### C++

```
// A C / C++ program for Bellman-Ford's single source
// shortest path algorithm.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

// a structure to represent a weighted edge in graph
struct Edge
{
 int src, dest, weight;
};

// a structure to represent a connected, directed and
// weighted graph
struct Graph
{
 // V-> Number of vertices, E-> Number of edges
 int V, E;

 // graph is represented as an array of edges.
 struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
 struct Graph* graph =
 (struct Graph*) malloc(sizeof(struct Graph));
 graph->V = V;
 graph->E = E;

 graph->edge =
 (struct Edge*) malloc(graph->E * sizeof(struct Edge));

 return graph;
}

// A utility function used to print the solution
void printArr(int dist[], int n)
{
 printf("Vertex Distance from Source\n");
 for (int i = 0; i < n; ++i)
 printf("%d \t %d\n", i, dist[i]);
}

// The main function that finds shortest distances from src to
// all other vertices using Bellman-Ford algorithm. The function
// also detects negative weight cycle
void BellmanFord(struct Graph* graph, int src)
{
 int V = graph->V;
 int E = graph->E;
 int dist[V];
}
```

```

// Step 1: Initialize distances from src to all other vertices
// as INFINITE
for (int i = 0; i < V; i++)
 dist[i] = INT_MAX;
dist[src] = 0;

// Step 2: Relax all edges |V| - 1 times. A simple shortest
// path from src to any other vertex can have at-most |V| - 1
// edges
for (int i = 1; i <= V-1; i++)
{
 for (int j = 0; j < E; j++)
 {
 int u = graph->edge[j].src;
 int v = graph->edge[j].dest;
 int weight = graph->edge[j].weight;
 if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
 dist[v] = dist[u] + weight;
 }
}

// Step 3: check for negative-weight cycles. The above step
// guarantees shortest distances if graph doesn't contain
// negative weight cycle. If we get a shorter path, then there
// is a cycle.
for (int i = 0; i < E; i++)
{
 int u = graph->edge[i].src;
 int v = graph->edge[i].dest;
 int weight = graph->edge[i].weight;
 if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
 printf("Graph contains negative weight cycle");
}

printArr(dist, V);

return;
}

// Driver program to test above functions
int main()
{
 /* Let us create the graph given in above example */
 int V = 5; // Number of vertices in graph
 int E = 8; // Number of edges in graph
 struct Graph* graph = createGraph(V, E);

 // add edge 0-1 (or A-B in above figure)
 graph->edge[0].src = 0;
 graph->edge[0].dest = 1;
 graph->edge[0].weight = -1;

 // add edge 0-2 (or A-C in above figure)
 graph->edge[1].src = 0;
 graph->edge[1].dest = 2;
 graph->edge[1].weight = 4;

 // add edge 1-2 (or B-C in above figure)
 graph->edge[2].src = 1;
 graph->edge[2].dest = 2;
 graph->edge[2].weight = 3;

 // add edge 1-3 (or B-D in above figure)
 graph->edge[3].src = 1;
 graph->edge[3].dest = 3;
 graph->edge[3].weight = 2;

 // add edge 1-4 (or A-E in above figure)
 graph->edge[4].src = 1;
 graph->edge[4].dest = 4;
 graph->edge[4].weight = 2;

 // add edge 3-2 (or D-C in above figure)
 graph->edge[5].src = 3;
 graph->edge[5].dest = 2;
 graph->edge[5].weight = 5;

 // add edge 3-1 (or D-B in above figure)
 graph->edge[6].src = 3;
 graph->edge[6].dest = 1;
 graph->edge[6].weight = 1;

 // add edge 4-3 (or E-D in above figure)
 graph->edge[7].src = 4;
 graph->edge[7].dest = 3;
 graph->edge[7].weight = -3;

 BellmanFord(graph, 0);

 return 0;
}

```

## Java

```

// A Java program for Bellman-Ford's single source shortest path
// algorithm.
import java.util.*;
import java.lang.*;
import java.io.*;

// A class to represent a connected, directed and weighted graph
class Graph
{
 // A class to represent a weighted edge in graph

```

```

class Edge {
 int src, dest, weight;
 Edge() {
 src = dest = weight = 0;
 }
};

int V, E;
Edge edge[]; // Creates a graph with V vertices and E edges

Graph(int v, int e)
{
 V = v;
 E = e;
 edge = new Edge[e];
 for (int i=0; i<e; ++i)
 edge[i] = new Edge();
}

// The main function that finds shortest distances from src
// to all other vertices using Bellman-Ford algorithm. The
// function also detects negative weight cycle
void BellmanFord(Graph graph,int src)
{
 int V = graph.V, E = graph.E;
 int dist[] = new int[V];

 // Step 1: Initialize distances from src to all other
 // vertices as INFINITE
 for (int i=0; i<V; ++i)
 dist[i] = Integer.MAX_VALUE;
 dist[src] = 0;

 // Step 2: Relax all edges |V| - 1 times. A simple
 // shortest path from src to any other vertex can
 // have at-most |V| - 1 edges
 for (int i=1; i<V; ++i)
 {
 for (int j=0; j<E; ++j)
 {
 int u = graph.edge[j].src;
 int v = graph.edge[j].dest;
 int weight = graph.edge[j].weight;
 if (dist[u]==Integer.MAX_VALUE &&
 dist[u]+weight<dist[v])
 dist[v]=dist[u]+weight;
 }
 }

 // Step 3: check for negative-weight cycles. The above
 // step guarantees shortest distances if graph doesn't
 // contain negative weight cycle. If we get a shorter
 // path, then there is a cycle.
 for (int j=0; j<E; ++j)
 {
 int u = graph.edge[j].src;
 int v = graph.edge[j].dest;
 int weight = graph.edge[j].weight;
 if (dist[u]==Integer.MAX_VALUE &&
 dist[u]+weight<dist[v])
 System.out.println("Graph contains negative weight cycle");
 }
 printArr(dist, V);
}

// A utility function used to print the solution
void printArr(int dist[], int V)
{
 System.out.println("Vertex Distance from Source");
 for (int i=0; i<V; ++i)
 System.out.println(i+" "+dist[i]);
}

// Driver method to test above function
public static void main(String[] args)
{
 int V = 5; // Number of vertices in graph
 int E = 8; // Number of edges in graph

 Graph graph = new Graph(V, E);

 // add edge 0-1 (or A-B in above figure)
 graph.edge[0].src = 0;
 graph.edge[0].dest = 1;
 graph.edge[0].weight = -1;

 // add edge 0-2 (or A-C in above figure)
 graph.edge[1].src = 0;
 graph.edge[1].dest = 2;
 graph.edge[1].weight = 4;

 // add edge 1-2 (or B-C in above figure)
 graph.edge[2].src = 1;
 graph.edge[2].dest = 2;
 graph.edge[2].weight = 3;

 // add edge 1-3 (or B-D in above figure)
 graph.edge[3].src = 1;
 graph.edge[3].dest = 3;
 graph.edge[3].weight = 2;

 // add edge 1-4 (or A-E in above figure)
 graph.edge[4].src = 1;
 graph.edge[4].dest = 4;
 graph.edge[4].weight = 2;
}

```

```

// add edge 3-2 (or D-C in above figure)
graph.edge[5].src = 3;
graph.edge[5].dest = 2;
graph.edge[5].weight = 5;

// add edge 3-1 (or D-B in above figure)
graph.edge[6].src = 3;
graph.edge[6].dest = 1;
graph.edge[6].weight = 1;

// add edge 4-3 (or E-D in above figure)
graph.edge[7].src = 4;
graph.edge[7].dest = 3;
graph.edge[7].weight = -3;

graph.BellmanFord(graph, 0);
}
}

// Contributed by Aakash Hasija

```

## Python

```

Python program for Bellman-Ford's single source
shortest path algorithm.

from collections import defaultdict

#Class to represent a graph
class Graph:

 def __init__(self,vertices):
 self.V= vertices #No. of vertices
 self.graph = {} # default dictionary to store graph

 # function to add an edge to graph
 def addEdge(self,u,v,w):
 self.graph[u] = self.graph.get(u,[])
 self.graph[u].append([v,w])

 # utility function used to print the solution
 def printArr(self, dist):
 print("Vertex Distance from Source")
 for i in range(self.V):
 print("%d \t %d" % (i, dist[i]))

 # The main function that finds shortest distances from src to
 # all other vertices using Bellman-Ford algorithm. The function
 # also detects negative weight cycle
 def BellmanFord(self, src):

 # Step 1: Initialize distances from src to all other vertices
 # as INFINITE
 dist = [float("Inf")] * self.V
 dist[src] = 0

 # Step 2: Relax all edges |V| - 1 times. A simple shortest
 # path from src to any other vertex can have at-most |V| - 1
 # edges
 for i in range(self.V - 1):
 # Update dist value and parent index of the adjacent vertices
 # of the picked vertex. Consider only those vertices which are still in
 # queue
 for u, v, w in self.graph:
 if dist[u] != float("Inf") and dist[u] + w < dist[v]:
 dist[v] = dist[u] + w

 # Step 3: check for negative-weight cycles. The above step
 # guarantees shortest distances if graph doesn't contain
 # negative weight cycle. If we get a shorter path, then there
 # is a cycle.

 for u, v, w in self.graph:
 if dist[u] != float("Inf") and dist[u] + w < dist[v]:
 print("Graph contains negative weight cycle")
 return

 # print all distance
 self.printArr(dist)

g = Graph(5)
g.addEdge(0, 1, -1)
g.addEdge(0, 2, 4)
g.addEdge(1, 2, 3)
g.addEdge(1, 3, 2)
g.addEdge(1, 4, 2)
g.addEdge(3, 2, 5)
g.addEdge(3, 1, 5)
g.addEdge(4, 3, -3)

#Print the solution
g.BellmanFord(0)

#This code is contributed by Neelam Yadav

```

Output:

| Vertex | Distance from Source |
|--------|----------------------|
| 0      | 0                    |
| 1      | -1                   |
| 2      | 2                    |
| 3      | -2                   |
| 4      | 1                    |

#### Notes

- 1) Negative weights are found in various applications of graphs. For example, instead of paying cost for a path, we may get some advantage if we follow the path.
- 2) Bellman-Ford works better (better than Dijkstra's) for distributed systems. Unlike Dijkstra's where we need to find minimum value of all vertices, in Bellman-Ford, edges are considered one by one.

#### Exercise

- 1) The standard Bellman-Ford algorithm reports shortest path only if there is no negative weight cycles. Modify it so that it reports minimum distances even if there is a negative weight cycle.
- 2) Can we use Dijkstra's algorithm for shortest paths for graphs with negative weights – one idea can be, calculate the minimum weight value, add a positive value (equal to absolute value of minimum weight value) to all weights and run the Dijkstra's algorithm for the modified graph. Will this algorithm work?

#### References:

<http://www.youtube.com/watch?v=Ttezus39nk>  
[http://en.wikipedia.org/wiki/Bellman%E2%80%93Ford\\_algorithm](http://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm)  
<http://www.cs.arizona.edu/classes/cs445/spring07/ShortestPath2.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner Company Wise Coding Practice

Dynamic Programming  
Graph  
shortest path

## Dynamic Programming | Set 16 (Floyd Warshall Algorithm)

The [Floyd Warshall Algorithm](#) is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

Example:

**Input:**  
graph[] = { {0, 5, INF, 10},  
 {INF, 0, 3, INF},  
 {INF, INF, 0, 1},  
 {INF, INF, INF, 0} }  
which represents the following graph



Note that the value of graph[i][j] is 0 if i is equal to j.  
And graph[i][j] is INF (infinite) if there is no edge from vertex i to j.

**Output:**  
Shortest distance matrix  
0 5 8 9  
INF 0 3 4  
INF INF 0 1  
INF INF INF 0

#### Floyd Warshall Algorithm

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of source and destination vertices respectively, there are two possible cases.

- 1) k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.
- 2) k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j].

The following figure is taken from the Cormen book. It shows the above optimal substructure property in the all-pairs shortest path problem.



Following is implementations of the Floyd Warshall algorithm.

#### C/C++

```
// C Program for Floyd Warshall Algorithm
#include<stdio.h>

// Number of vertices in the graph
#define V 4

/* Define Infinite as a large enough value. This value will be used
 for vertices not connected to each other */
#define INF 99999

/* A function to print the solution matrix */
void printSolution(int dist[][V]);

/* Solves the all-pairs shortest path problem using Floyd Warshall algorithm */
void floydWarshall (int graph[][V])
{
 /* dist[][] will be the output matrix that will finally have the shortest
 distances between every pair of vertices */
 int dist[V][V], i, j, k;

 /* Initialize the solution matrix same as input graph matrix. Or
 we can say the initial values of shortest distances are based
 on shortest paths considering no intermediate vertex. */
 for (i = 0; i < V; i++)
 for (j = 0; j < V; j++)
 dist[i][j] = graph[i][j];

 /* Add all vertices one by one to the set of intermediate vertices.
 Note that at this stage, dist[i][j] contains shortest distance
 from src i to dest j using vertices in set {0, 1, ..., k-1} as
 intermediate vertices. */
 for (k = 0; k < V; k++)
 {
 /* Update dist[][] so that dist[i][j] now contains the shortest
 distance between vertex i and j and also considers the
 intermediate vertex k */
 for (i = 0; i < V; i++)
 for (j = 0; j < V; j++)
 if (dist[i][j] > dist[i][k] + dist[k][j])
 dist[i][j] = dist[i][k] + dist[k][j];
 }

 /* Print the shortest distance matrix */
 printSolution(dist);
}
```

```

/* Add all vertices one by one to the set of intermediate vertices.
--> Before start of a iteration, we have shortest distances between all
pairs of vertices such that the shortest distances consider only the
vertices in set {0, 1, 2, .. k-1} as intermediate vertices.
----> After the end of a iteration, vertex no. k is added to the set of
intermediate vertices and the set becomes {0, 1, 2, .. k}
for (k = 0; k < V; k++)
{
 // Pick all vertices as source one by one
 for (i = 0; i < V; i++)
 {
 // Pick all vertices as destination for the
 // above picked source
 for (j = 0; j < V; j++)
 {
 // If vertex k is on the shortest path from
 // i to j, then update the value of dist[i][j]
 if (dist[i][k] + dist[k][j] < dist[i][j])
 dist[i][j] = dist[i][k] + dist[k][j];
 }
 }
}

// Print the shortest distance matrix
printSolution(dist);
}

/* A utility function to print solution */
void printSolution(int dist[][V])
{
 printf ("Following matrix shows the shortest distances"
 "between every pair of vertices \n");
 for (int i = 0; i < V; i++)
 {
 for (int j = 0; j < V; j++)
 {
 if (dist[i][j] == INF)
 printf("%s", "INF");
 else
 printf ("%7d", dist[i][j]);
 }
 printf("\n");
 }
}

// driver program to test above function
int main()
{
 /* Let us create the following weighted graph
 10
 (0)----->(3)
 | /\
 | |
 | [1]
 | |
 (1)----->(2)
 3 */
 int graph[V][V] = { {0, 5, INF, 10},
 {INF, 0, 3, INF},
 {INF, INF, 0, 1},
 {INF, INF, INF, 0}
 };

 // Print the solution
 floydWarshall(graph);
 return 0;
}

```

## Java

```

// A Java program for Floyd Warshall All Pairs Shortest
// Path algorithm.
import java.util.*;
import java.lang.*;
import java.io.*;

class AllPairShortestPath
{
 final static int INF = 99999, V = 4;

 void floydWarshall(int graph[][])
 {
 int dist[][] = new int[V][V];
 int i, j, k;

 /* Initialize the solution matrix same as input graph matrix.
 Or we can say the initial values of shortest distances
 are based on shortest paths considering no intermediate
 vertex. */
 for (i = 0; i < V; i++)
 for (j = 0; j < V; j++)
 dist[i][j] = graph[i][j];

 /* Add all vertices one by one to the set of intermediate
 vertices.
 --> Before start of a iteration, we have shortest
 distances between all pairs of vertices such that
 the shortest distances consider only the vertices in
 set {0, 1, 2, .. k-1} as intermediate vertices.
 ----> After the end of a iteration, vertex no. k is added
 to the set of intermediate vertices and the set
 becomes {0, 1, 2, .. k} */
 for (k = 0; k < V; k++)

```

```

{
 // Pick all vertices as source one by one
 for (i = 0; i < V; i++)
 {
 // Pick all vertices as destination for the
 // above picked source
 for (j = 0; j < V; j++)
 {
 // If vertex k is on the shortest path from
 // i to j, then update the value of dist[i][j]
 if (dist[i][k] + dist[k][j] < dist[i][j])
 dist[i][j] = dist[i][k] + dist[k][j];
 }
 }

 // Print the shortest distance matrix
 printSolution(dist);
}

void printSolution(int dist[][])
{
 System.out.println("Following matrix shows the shortest "+
 "distances between every pair of vertices");
 for (int i=0; i<V; i++)
 {
 for (int j=0; j<V; j++)
 {
 if (dist[i][j]==INF)
 System.out.print("INF ");
 else
 System.out.print(dist[i][j]+ " ");
 }
 System.out.println();
 }
}

// Driver program to test above function
public static void main (String[] args)
{
 /* Let us create the following weighted graph
 10
 (0)----->(3)
 | \
 | |
 5 | 1
 \V |
 (1)----->(2)
 3 '
 int graph[][] = {{0, 5, INF, 10},
 {INF, 0, 3, INF},
 {INF, INF, 0, 1},
 {INF, INF, INF, 0}
 };
 AllPairShortestPath a = new AllPairShortestPath();

 // Print the solution
 a.floydWarshall(graph);
}
}

// Contributed by Aakash Hasija

```

## Python

```

Python Program for Floyd Warshall Algorithm

Number of vertices in the graph
V = 4

Define infinity as the large enough value. This value will be
used for vertices not connected to each other
INF = 99999

Solves all pair shortest path via Floyd Warshall Algorithm
def floydWarshall(graph):

 """ dist[][] will be the output matrix that will finally
 have the shortest distances between every pair of vertices """
 """ initializing the solution matrix same as input graph matrix
 OR we can say that the initial values of shortest distances
 are based on shortest paths considering no
 intermediate vertices """
 dist = map(lambda i : map(lambda j : j , i) , graph)

 """ Add all vertices one by one to the set of intermediate
 vertices.
 --> Before start of a iteration, we have shortest distances
 between all pairs of vertices such that the shortest
 distances consider only the vertices in set
 {0, 1, 2, .. k-1} as intermediate vertices.
 ----> After the end of a iteration, vertex no. k is
 added to the set of intermediate vertices and the
 set becomes {0, 1, 2, .. k}
 """

 for k in range(V):

 # pick all vertices as source one by one
 for i in range(V):

 # Pick all vertices as destination for the
 # above picked source
 for j in range(V):

```

```

If vertex k is on the shortest path from
i to j, then update the value of dist[i][j]
dist[i][j] = min(dist[i][j],
 dist[i][k]+ dist[k][j])
)
printSolution(dist)

A utility function to print the solution
def printSolution(dist):
 print "Following matrix shows the shortest distances\
between every pair of vertices"
 for i in range(V):
 for j in range(V):
 if(dist[i][j] == INF):
 print "%7s" %(INF),
 else:
 print "%7d" %(dist[i][j]),
 if j == V-1:
 print ""
 print ""

Driver program to test the above program
Let us create the following weighted graph
"""
 10
 (0)----->(3)
 | /\
 | |
 5 | |
 | | 1
 \| |
 (1)----->(2)
 3 """
graph = [[0.5,INF,10],
 [INF,0.3,INF],
 [INF,INF,0, 1],
 [INF,INF,INF, 0]
]
Print the solution
floydWarshall(graph);
This code is contributed by Nikhil Kumar Singh(nickzuck_007)

```

Output:

```

Following matrix shows the shortest distances between every pair of vertices
 0 5 8 9
INF 0 3 4
INF INF 0 1
INF INF INF 0

```

Time Complexity:  $O(V^3)$

The above program only prints the shortest distances. We can modify the solution to print the shortest paths also by storing the predecessor information in a separate 2D matrix. Also, the value of INF can be taken as INT\_MAX from limits.h to make sure that we handle maximum possible value. When we take INF as INT\_MAX, we need to change the if condition in the above program to avoid arithmetic overflow.

```

#include <limits.h>

#define INF INT_MAX
.....
if (dist[i][k] != INF &&
 dist[k][j] != INF &&
 dist[i][k] + dist[k][j] < dist[i][j])
)
dist[i][j] = dist[i][k] + dist[k][j];
.....

```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner Company Wise Coding Practice

Dynamic Programming  
Graph  
shortest path

## Johnson's algorithm for All-pairs shortest paths

The problem is to find shortest paths between every pair of vertices in a given weighted directed Graph and weights may be negative. We have discussed [Floyd Warshall Algorithm](#) for this problem. Time complexity of Floyd Warshall Algorithm is  $\Theta(V^3)$ . Using Johnson's algorithm, we can find all pair shortest paths in  $O(V^2 \log V + VE)$  time. Johnson's algorithm uses both Dijkstra and Bellman-Ford as subroutines.

If we apply Dijkstra's Single Source shortest path algorithm for every vertex, considering every vertex as source, we can find all pair shortest paths in  $O(V^2V\log V)$  time. So using Dijkstra's single source shortest path seems to be a better option than Floyd Warshall, but the problem with Dijkstra's algorithm is, it doesn't work for negative weight edge.

*The idea of Johnson's algorithm is to re-weight all edges and make them all positive, then apply Dijkstra's algorithm for every vertex.*

### How to transform a given graph to a graph with all non-negative weight edges?

One may think of a simple approach of finding the minimum weight edge and adding this weight to all edges. Unfortunately, this doesn't work as there may be different number of edges in different paths (See [this](#) for an example). If there are multiple paths from a vertex u to v, then all paths must be increased by same amount, so that the shortest path remains the shortest in the transformed graph.

The idea of Johnson's algorithm is to assign a weight to every vertex. Let the weight assigned to vertex u be  $h[u]$ . We reweight edges using vertex weights. For example, for an edge  $(u, v)$  of weight  $w(u, v)$ , the new weight becomes  $w(u, v) + h[u] - h[v]$ . The great thing about this reweighting is, all set of paths between any two vertices are increased by same amount and all negative weights become non-negative. Consider any path between two vertices s and t, weight of every path is increased by  $h[s] - h[t]$ , all  $h[]$  values of vertices on path from s to t cancel each other.

How do we calculate  $h[]$  values? [Bellman-Ford algorithm](#) is used for this purpose. Following is the complete algorithm. A new vertex is added to the graph and connected to all existing vertices. The shortest distance values from new vertex to all existing vertices are  $h[]$  values.

### Algorithm:

- 1) Let the given graph be G. Add a new vertex s to the graph, add edges from new vertex to all vertices of G. Let the modified graph be  $G'$ .
- 2) Run [Bellman-Ford algorithm](#) on  $G'$  with s as source. Let the distances calculated by Bellman-Ford be  $h[0], h[1], \dots h[V-1]$ . If we find a negative weight cycle, then return. Note that the negative weight cycle

cannot be created by new vertex s as there is no edge to s. All edges are from s.

3) Reweight the edges of original graph. For each edge  $(u, v)$ , assign the new weight as "original weight +  $h[u] - h[v]$ ".

4) Remove the added vertex s and run Dijkstra's algorithm for every vertex.

#### How does the transformation ensure nonnegative weight edges?

The following property is always true about  $h[]$  values as they are shortest distances.

$h[v]$

The property simply means, shortest distance from s to v must be smaller than or equal to shortest distance from s to u plus weight of edge  $(u, v)$ . The new weights are  $w(u, v) + h[u] - h[v]$ . The value of the new weights must be non-negative.

Let us consider the following graph.



We add a source s and add edges from s to all vertices of the original graph. In the following diagram s is 4.



We calculate the shortest distances from 4 to all other vertices using Bellman-Ford algorithm. The shortest distances from 4 to 0, 1, 2 and 3 are 0, -5, -1 and 0 respectively, i.e.,  $h[] = \{0, -5, -1, 0\}$ . Once we get these distances,



Since all weights are positive now, we can run Dijkstra's shortest path algorithm for every vertex as source.

**Time Complexity:** The main steps in algorithm are Bellman Ford Algorithm called once and Dijkstra called V times. Time complexity of Bellman Ford is  $O(VE)$  and time complexity of Dijkstra is  $O(VLogV)$ . So overall time

The time complexity of Johnson's algorithm becomes same as [Floyd Warshall](#) when the graphs is complete (For a complete graph  $E = O(V^2)$ ). But for sparse graphs, the algorithm performs much better than [Floyd Warshall](#).

#### References:

[Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](#)

<http://www.youtube.com/watch?v=b6LOHvCzmkI>

<http://www.youtube.com/watch?v=TV2Z6nbo1ic>

[http://en.wikipedia.org/wiki/Johnson%27s\\_algorithm](http://en.wikipedia.org/wiki/Johnson%27s_algorithm)

<http://www.youtube.com/watch?v=Syyq1e0xWnM>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner

### Company Wise Coding Practice

## Shortest Path in Directed Acyclic Graph

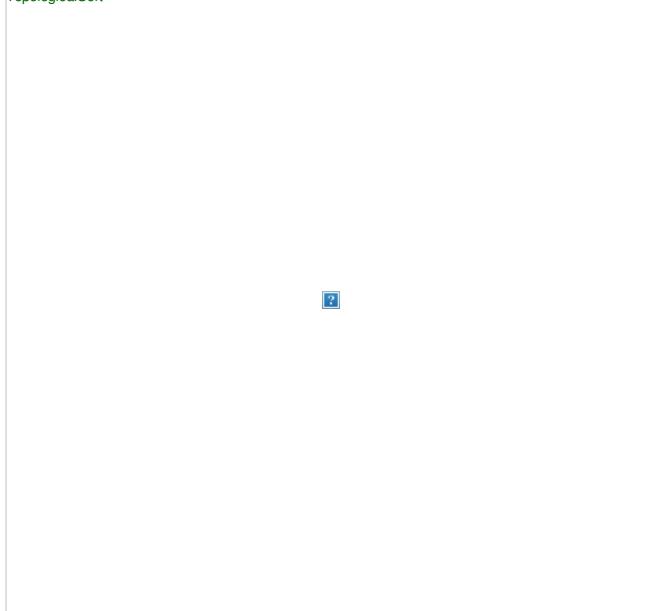
Given a Weighted Directed Acyclic Graph and a source vertex in the graph, find the shortest paths from given source to all other vertices.

For a general weighted graph, we can calculate single source shortest distances in  $O(VE)$  time using [Bellman-Ford Algorithm](#). For a graph with no negative weights, we can do better and calculate single source shortest distances in  $O(E + V\log V)$  time using [Dijkstra's algorithm](#). Can we do even better for Directed Acyclic Graph (DAG)? We can calculate single source shortest distances in  $O(V+E)$  time for DAGs. The idea is to use [Topological Sorting](#).

We initialize distances to all vertices as infinite and distance to source as 0, then we find a topological sorting of the graph. [Topological Sorting](#) of a graph represents a linear ordering of the graph (See below, figure (b) is a linear representation of figure (a)). Once we have topological order (or linear representation), we one by one process all vertices in topological order. For every vertex being processed, we update distances of its adjacent using distance of current vertex.

Following figure is taken from [this](#) source. It shows step by step process of finding shortest paths.

TopologicalSort



Following is complete algorithm for finding shortest distances.

- 1) Initialize  $\text{dist}[] = \{\text{INF}, \text{INF}, \dots\}$  and  $\text{dist}[s] = 0$  where  $s$  is the source vertex.
- 2) Create a topological order of all vertices.
- 3) Do following for every vertex  $u$  in topological order.  
.....Do following for every adjacent vertex  $v$  of  $u$   
.....if ( $\text{dist}[v] > \text{dist}[u] + \text{weight}(u, v)$ )  
..... $\text{dist}[v] = \text{dist}[u] + \text{weight}(u, v)$

### C++

```
// C++ program to find single source shortest paths for Directed Acyclic Graphs
#include<iostream>
#include <list>
#include <stack>
#include <limits.h>
#define INF INT_MAX
using namespace std;

// Graph is represented using adjacency list. Every node of adjacency list
// contains vertex number of the vertex to which edge connects. It also
// contains weight of the edge
class AdjListNode
{
 int v;
 int weight;
public:
 AdjListNode(int _v, int _w) { v = _v; weight = _w; }
 int getV() { return v; }
 int getWeight() { return weight; }
};

// Class to represent a graph using adjacency list representation
class Graph
{
 int V; // No. of vertices

 // Pointer to an array containing adjacency lists
 list<AdjListNode> *adj;

 // A function used by shortestPath
 void topologicalSortUtil(int v, bool visited[], stack<int> &Stack);
public:
 Graph(int V); // Constructor

 // function to add an edge to graph
 void addEdge(int u, int v, int weight);

 // Finds shortest paths from given source vertex
 void shortestPath(int s);
};

Graph::Graph(int V)
{
```

```

this->V = V;
adj = new list<AdjListNode>[V];
}

void Graph::addEdge(int u, int v, int weight)
{
 AdjListNode node(v, weight);
 adj[u].push_back(node); // Add v to u's list
}

// A recursive function used by shortestPath. See below link for details
// http://www.geeksforgeeks.org/topological-sorting/
void Graph::topologicalSortUtil(int v, bool visited[], stack<int> &Stack)
{
 // Mark the current node as visited
 visited[v] = true;

 // Recur for all the vertices adjacent to this vertex
 list<AdjListNode>::iterator i;
 for (i = adj[v].begin(); i != adj[v].end(); ++i)
 {
 AdjListNode node = *i;
 if (!visited[node.getV()])
 topologicalSortUtil(node.getV(), visited, Stack);
 }

 // Push current vertex to stack which stores topological sort
 Stack.push(v);
}

// The function to find shortest paths from given vertex. It uses recursive
// topologicalSortUtil() to get topological sorting of given graph.
void Graph::shortestPath(int s)
{
 stack<int> Stack;
 int dist[V];

 // Mark all the vertices as not visited
 bool *visited = new bool[V];
 for (int i = 0; i < V; i++)
 visited[i] = false;

 // Call the recursive helper function to store Topological Sort
 // starting from all vertices one by one
 for (int i = 0; i < V; i++)
 if (visited[i] == false)
 topologicalSortUtil(i, visited, Stack);

 // Initialize distances to all vertices as infinite and distance
 // to source as 0
 for (int i = 0; i < V; i++)
 dist[i] = INF;
 dist[s] = 0;

 // Process vertices in topological order
 while (Stack.empty() == false)
 {
 // Get the next vertex from topological order
 int u = Stack.top();
 Stack.pop();

 // Update distances of all adjacent vertices
 list<AdjListNode>::iterator i;
 if (dist[u] != INF)
 {
 for (i = adj[u].begin(); i != adj[u].end(); ++i)
 if (dist[i->getV()] > dist[u] + i->getWeight())
 dist[i->getV()] = dist[u] + i->getWeight();
 }
 }

 // Print the calculated shortest distances
 for (int i = 0; i < V; i++)
 (dist[i] == INF) ? cout << "INF" : cout << dist[i] << " ";
}

// Driver program to test above functions
int main()
{
 // Create a graph given in the above diagram. Here vertex numbers are
 // 0, 1, 2, 3, 4, 5 with following mappings:
 // 0=r, 1=s, 2=t, 3=x, 4=y, 5=z
 Graph g(6);
 g.addEdge(0, 1, 5);
 g.addEdge(0, 2, 3);
 g.addEdge(1, 3, 6);
 g.addEdge(1, 2, 2);
 g.addEdge(2, 4, 4);
 g.addEdge(2, 5, 2);
 g.addEdge(2, 3, 7);
 g.addEdge(3, 4, -1);
 g.addEdge(4, 5, -2);

 int s = 1;
 cout << "Following are shortest distances from source " << s << "\n";
 g.shortestPath(s);

 return 0;
}

```

## Java

```

// Java program to find single source shortest paths in Directed Acyclic Graphs
import java.io.*;

```

```

import java.util.*;

class ShortestPath
{
 static final int INF=Integer.MAX_VALUE;
 class AdjListNode
 {
 private int v;
 private int weight;
 AdjListNode(int _v, int _w) { v = _v; weight = _w; }
 int getV() { return v; }
 int getWeight() { return weight; }
 }

 // Class to represent graph as an adjacency list of
 // nodes of type AdjListNode
 class Graph
 {
 private int V;
 private LinkedList<AdjListNode>adj[];
 Graph(int v)
 {
 V=v;
 adj = new LinkedList[V];
 for (int i=0;i<V; ++i)
 adj[i] = new LinkedList<AdjListNode>();
 }
 void addEdge(int u, int v, int weight)
 {
 AdjListNode node = new AdjListNode(v,weight);
 adj[u].add(node); // Add v to u's list
 }
 }

 // A recursive function used by shortestPath.
 // See below link for details
 void topologicalSortUtil(int v, Boolean visited[], Stack stack)
 {
 // Mark the current node as visited.
 visited[v] = true;
 Integer i;

 // Recur for all the vertices adjacent to this vertex
 Iterator<AdjListNode> it = adj[v].iterator();
 while (it.hasNext())
 {
 AdjListNode node = it.next();
 if (!visited[node.getV()])
 topologicalSortUtil(node.getV(), visited, stack);
 }
 // Push current vertex to stack which stores result
 stack.push(new Integer(v));
 }

 // The function to find shortest paths from given vertex. It
 // uses recursive topologicalSortUtil() to get topological
 // sorting of given graph.
 void shortestPath(int s)
 {
 Stack stack = new Stack();
 int dist[] = new int[V];

 // Mark all the vertices as not visited
 Boolean visited[] = new Boolean[V];
 for (int i = 0; i < V; i++)
 visited[i] = false;

 // Call the recursive helper function to store Topological
 // Sort starting from all vertices one by one
 for (int i = 0; i < V; i++)
 if (visited[i] == false)
 topologicalSortUtil(i, visited, stack);

 // Initialize distances to all vertices as infinite and
 // distance to source as 0
 for (int i = 0; i < V; i++)
 dist[i] = INF;
 dist[s] = 0;

 // Process vertices in topological order
 while (stack.empty() == false)
 {
 // Get the next vertex from topological order
 int u = (int)stack.pop();

 // Update distances of all adjacent vertices
 Iterator<AdjListNode> it;
 if (dist[u] != INF)
 {
 it = adj[u].iterator();
 while (it.hasNext())
 {
 AdjListNode i=it.next();
 if (dist[i.getV()] > dist[u] + i.getWeight())
 dist[i.getV()] = dist[u] + i.getWeight();
 }
 }
 }

 // Print the calculated shortest distances
 for (int i = 0; i < V; i++)
 {
 if (dist[i] == INF)
 System.out.print("INF ");
 else
 System.out.print(dist[i] + " ");
 }
 }
}

```

```

}

// Method to create a new graph instance through an object
// of ShortestPath class.
Graph newGraph(int number)
{
 return new Graph(number);
}

public static void main(String args[])
{
 // Create a graph given in the above diagram. Here vertex
 // numbers are 0, 1, 2, 3, 4, 5 with following mappings:
 // 0=r, 1=s, 2=t, 3=x, 4=y, 5=z
 ShortestPath t = new ShortestPath();
 Graph g = t.newGraph(6);
 g.addEdge(0, 1, 5);
 g.addEdge(0, 2, 3);
 g.addEdge(1, 3, 6);
 g.addEdge(1, 2, 2);
 g.addEdge(2, 4, 4);
 g.addEdge(2, 5, 2);
 g.addEdge(2, 3, 7);
 g.addEdge(3, 4, -1);
 g.addEdge(4, 5, -2);

 int s = 1;
 System.out.println("Following are shortest distances "+
 "from source " + s);
 g.shortestPath(s);
}
}

//This code is contributed by Aakash Hasija

```

## Python

```

Python program to find single source shortest paths
for Directed Acyclic Graphs Complexity :O(V+E)
from collections import defaultdict

Graph is represented using adjacency list. Every
node of adjacency list contains vertex number of
the vertex to which edge connects. It also contains
weight of the edge
class Graph:
 def __init__(self,vertices):
 self.V = vertices # No. of vertices

 # dictionary containing adjacency List
 self.graph = defaultdict(list)

 # function to add an edge to graph
 def addEdge(self,u,v,w):
 self.graph[u].append((v,w))

 # A recursive function used by shortestPath
 def topologicalSortUtil(self,v,visited,stack):

 # Mark the current node as visited.
 visited[v] = True

 # Recur for all the vertices adjacent to this vertex
 if v in self.graph.keys():
 for node,weight in self.graph[v]:
 if visited[node] == False:
 self.topologicalSortUtil(node,visited,stack)

 # Push current vertex to stack which stores topological sort
 stack.append(v)

 """ The function to find shortest paths from given vertex.
 It uses recursive topologicalSortUtil() to get topological
 sorting of given graph."""
 def shortestPath(self, s):

 # Mark all the vertices as not visited
 visited = [False]*self.V
 stack = []

 # Call the recursive helper function to store Topological
 # Sort starting from source vertex
 for i in range(self.V):
 if visited[i] == False:
 self.topologicalSortUtil(s,visited,stack)

 # Initialize distances to all vertices as infinite and
 # distance to source as 0
 dist = [float("Inf")] * (self.V)
 dist[s] = 0

 # Process vertices in topological order
 while stack:

 # Get the next vertex from topological order
 i = stack.pop()

 # Update distances of all adjacent vertices
 for node,weight in self.graph[i]:
 if dist[node] > dist[i] + weight:
 dist[node] = dist[i] + weight

```

```

Print the calculated shortest distances
for i in range(self.V):
 print ("%d"%dist[i]) if dist[i] != float("Inf") else "Inf",

g = Graph(6)
g.addEdge(0, 1, 5)
g.addEdge(0, 2, 3)
g.addEdge(1, 3, 6)
g.addEdge(1, 2, 2)
g.addEdge(2, 4, 4)
g.addEdge(2, 5, 2)
g.addEdge(2, 3, 7)
g.addEdge(3, 4, -1)
g.addEdge(4, 5, -2)

source = 1
s = 1

print ("Following are shortest distances from source %d "% s)
g.shortestPath(s)

This code is contributed by Neelam Yadav

```

Output:

```

Following are shortest distances from source 1
INF 0 2 6 5 3

```

**Time Complexity:** Time complexity of topological sorting is  $O(V+E)$ . After finding topological order, the algorithm processes all vertices and for every vertex, it runs a loop for all adjacent vertices. Total adjacent vertices in a graph is  $O(E)$ . So the inner loop runs  $O(V+E)$  times. Therefore, overall time complexity of this algorithm is  $O(V+E)$ .

References:

<http://www.utdallas.edu/~sizheng/CS4349.d/I-notes.d/L17.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner Company Wise Coding Practice

Graph  
shortest path  
Topological Sorting

## Some interesting shortest path questions | Set 1

**Question 1:** Given a directed weighted graph. You are also given the shortest path from a source vertex 's' to a destination vertex 't'. If weight of every edge is increased by 10 units, does the shortest path remain same in the modified graph?

The shortest path may change. The reason is, there may be different number of edges in different paths from s to t. For example, let shortest path be of weight 15 and has 5 edges. Let there be another path with 2 edges and total weight 25. The weight of the shortest path is increased by  $5 \times 10$  and becomes  $15 + 50$ . Weight of the other path is increased by  $2 \times 10$  and becomes  $25 + 20$ . So the shortest path changes to the other path with weight as 45.

**Question 2:** This is similar to above question. Does the shortest path change when weights of all edges are multiplied by 10?

If we multiply all edge weights by 10, the shortest path doesn't change. The reason is simple, weights of all paths from s to t get multiplied by same amount. The number of edges on a path doesn't matter. It is like changing unit of weights.

**Question 3:** Given a directed graph where every edge has weight as either 1 or 2, find the shortest path from a given source vertex 's' to a given destination vertex 't'. Expected time complexity is  $O(V+E)$ .

If we apply Dijkstra's shortest path algorithm, we can get a shortest path in  $O(E + V\log V)$  time. How to do it in  $O(V+E)$  time? The idea is to use BFS. One important observation about BFS is, the path used in BFS always has least number of edges between any two vertices. So if all edges are of same weight, we can use BFS to find the shortest path. For this problem, we can modify the graph and split all edges of weight 2 into two edges of weight 1 each. In the modified graph, we can use BFS to find the shortest path. How is this approach  $O(V+E)$ ? In worst case, all edges are of weight 2 and we need to do  $O(E)$  operations to split all edges, so the time complexity becomes  $O(E) + O(V+E)$  which is  $O(V+E)$ .

**Question 4:** Given a directed acyclic weighted graph, how to find the shortest path from a source s to a destination t in  $O(V+E)$  time?

See: Shortest Path in Directed Acyclic Graph

**More Questions** See following links for more questions.

<http://algs4.cs.princeton.edu/44sp/>

<http://geeksquiz.com/graph-shortest-paths/>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner Company Wise Coding Practice

Graph  
Graph  
shortest path

## Shortest path with exactly k edges in a directed and weighted graph

Given a directed and two vertices 'u' and 'v' in it, find shortest path from 'u' to 'v' with exactly k edges on the path.

The graph is given as **adjacency matrix representation** where value of  $\text{graph}[i][j]$  indicates the weight of an edge from vertex  $i$  to vertex  $j$  and a value INF(infinite) indicates no edge from  $i$  to  $j$ .

For example consider the following graph. Let source 'u' be vertex 0, destination 'v' be 3 and  $k$  be 2. There are two walks of length 2, the walks are {0, 2, 3} and {0, 1, 3}. The shortest among the two is {0, 2, 3} and weight of path is  $3+6 = 9$ .



The idea is to browse through all paths of length  $k$  from  $u$  to  $v$  using the approach discussed in the previous post and return weight of the shortest path. A **simple solution** is to start from  $u$ , go to all adjacent vertices and recur for adjacent vertices with  $k$  as  $k-1$ , source as adjacent vertex and destination as  $v$ . Following are C++ and Java implementations of this simple solution.

## C++

```
// C++ program to find shortest path with exactly k edges
#include <iostream>
#include <climits>
using namespace std;

// Define number of vertices in the graph and infinite value
#define V 4
#define INF INT_MAX

// A naive recursive function to count walks from u to v with k edges
int shortestPath(int graph[][V], int u, int v, int k)
{
 // Base cases
 if (k == 0 && u == v) return 0;
 if (k == 1 && graph[u][v] != INF) return graph[u][v];
 if (k <= 0) return INF;

 // Initialize result
 int res = INF;

 // Go to all adjacents of u and recur
 for (int i = 0; i < V; i++)
 {
 if (graph[u][i] != INF && u != i && v != i)
 {
 int rec_res = shortestPath(graph, i, v, k-1);
 if (rec_res != INF)
 res = min(res, graph[u][i] + rec_res);
 }
 }
 return res;
}

// driver program to test above function
int main()
{
 /* Let us create the graph shown in above diagram*/
 int graph[V][V] = { { 0, 10, 3, 2 },
 { INF, 0, INF, 7 },
 { INF, INF, 0, 6 },
 { INF, INF, INF, 0 } };
 int u = 0, v = 3, k = 2;
 cout << "Weight of the shortest path is " <<
 shortestPath(graph, u, v, k);
 return 0;
}
```

## Java

```
// Dynamic Programming based Java program to find shortest path
// with exactly k edges
import java.util.*;
import java.lang.*;
import java.io.*;

class ShortestPath
{
 // Define number of vertices in the graph and infinite value
 static final int V = 4;
 static final int INF = Integer.MAX_VALUE;

 // A naive recursive function to count walks from u to v
 // with k edges
 int shortestPath(int graph[][], int u, int v, int k)
 {
 // Base cases
 if (k == 0 && u == v) return 0;
 if (k == 1 && graph[u][v] != INF) return graph[u][v];
 if (k <= 0) return INF;

 // Initialize result
 int res = INF;

 // Go to all adjacents of u and recur
 for (int i = 0; i < V; i++)
 {
 if (graph[u][i] != INF && u != i && v != i)
 {
 int rec_res = shortestPath(graph, i, v, k-1);
 if (rec_res != INF)
 res = Math.min(res, graph[u][i] + rec_res);
 }
 }
 return res;
 }

 public static void main (String[] args)
 {
 /* Let us create the graph shown in above diagram*/
 int graph[][] = new int[][]{ { 0, 10, 3, 2 },
 { INF, 0, INF, 7 },
 { INF, INF, 0, 6 },
 { INF, INF, INF, 0 } };
 ShortestPath t = new ShortestPath();
 int u = 0, v = 3, k = 2;
 System.out.println("Weight of the shortest path is "+
 t.shortestPath(graph, u, v, k));
 }
}
```

Output:

```
Weight of the shortest path is 9
```

The worst case time complexity of the above function is  $O(V^k)$  where  $V$  is the number of vertices in the given graph. We can simply analyze the time complexity by drawing recursion tree. The worst occurs for a complete graph. In worst case, every internal node of recursion tree would have exactly  $V$  children.

We can optimize the above solution using **Dynamic Programming**. The idea is to build a 3D table where first dimension is source, second dimension is destination, third dimension is number of edges from source to destination, and the value is count of walks. Like other Dynamic Programming problems, we fill the 3D table in bottom up manner.

## C++

```
// Dynamic Programming based C++ program to find shortest path with
// exactly k edges
#include <iostream>
#include <climits>
using namespace std;

// Define number of vertices in the graph and infinite value
#define V 4
#define INF INT_MAX

// A Dynamic programming based function to find the shortest path from
// u to v with exactly k edges.
int shortestPath(int graph[][V], int u, int v, int k)
{
 // Table to be filled up using DP. The value sp[i][j][e] will store
 // weight of the shortest path from i to j with exactly k edges
 int sp[V][V][k+1];

 // Loop for number of edges from 0 to k
 for (int e = 0; e <= k; e++)
 {
 for (int i = 0; i < V; i++) // for source
 {
 for (int j = 0; j < V; j++) // for destination
 {
 // initialize value
 sp[i][j][e] = INF;

 // from base cases
 if (e == 0 && i == j)
 sp[i][j][e] = 0;
 if (e == 1 && graph[i][j] != INF)
 sp[i][j][e] = graph[i][j];

 // go to adjacent only when number of edges is more than 1
 if (e > 1)
 {
 for (int a = 0; a < V; a++)
 {
 // There should be an edge from i to a and a
 // should not be same as either i or j
 if (graph[i][a] != INF && i != a &&
 j != a && sp[a][j][e-1] != INF)
 sp[i][j][e] = min(sp[i][j][e], graph[i][a] +
 sp[a][j][e-1]);
 }
 }
 }
 }
 }
 return sp[u][v][k];
}

// driver program to test above function
int main()
{
 /* Let us create the graph shown in above diagram*/
 int graph[V][V] = { { 0, 10, 3, 2 },
 { INF, 0, INF, 7 },
 { INF, INF, 0, 6 },
 { INF, INF, INF, 0 } };
 int u = 0, v = 3, k = 2;
 cout << shortestPath(graph, u, v, k);
 return 0;
}
```

## Java

```
// Dynamic Programming based Java program to find shortest path with
// exactly k edges
import java.util.*;
import java.lang.*;
import java.io.*;

class ShortestPath
{
 // Define number of vertices in the graph and infinite value
 static final int V = 4;
 static final int INF = Integer.MAX_VALUE;

 // A Dynamic programming based function to find the shortest path
 // from u to v with exactly k edges.
 int shortestPath(int graph[][], int u, int v, int k)
 {
 // Table to be filled up using DP. The value sp[i][j][e] will
 // store weight of the shortest path from i to j with exactly
 // k edges
 int sp[][][] = new int[V][V][k+1];
```

```

// Loop for number of edges from 0 to k
for (int e = 0; e <= k; e++)
{
 for (int i = 0; i < V; i++) // for source
 {
 for (int j = 0; j < V; j++) // for destination
 {
 // initialize value
 sp[i][j][e] = INF;

 // from base cases
 if (e == 0 && i == j)
 sp[i][j][e] = 0;
 if (e == 1 && graph[i][j] != INF)
 sp[i][j][e] = graph[i][j];

 // go to adjacent only when number of edges is
 // more than 1
 if (e > 1)
 {
 for (int a = 0; a < V; a++)
 {
 // There should be an edge from i to a and
 // a should not be same as either i or j
 if (graph[i][a] != INF && i != a &&
 j != a && sp[a][j][e-1] != INF)
 sp[i][j][e] = Math.min(sp[i][j][e],
 graph[i][a] + sp[a][j][e-1]);
 }
 }
 }
 }
}
return sp[u][v][k];
}

public static void main (String[] args)
{
 /* Let us create the graph shown in above diagram*/
 int graph[][] = new int[][] { {0, 10, 3, 2},
 {INF, 0, INF, 7},
 {INF, INF, 0, 6},
 {INF, INF, INF, 0}
 };
 ShortestPath t = new ShortestPath();
 int u = 0, v = 3, k = 2;
 System.out.println("Weight of the shortest path is "+
 t.shortestPath(graph, u, v, k));
}
}

//This code is contributed by Aakash Hasija

```

Output:

Weight of the shortest path is 9

Time complexity of the above DP based solution is  $O(V^3K)$  which is much better than the naive solution.

This article is contributed by **Abhishek**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner Company Wise Coding Practice

Dynamic Programming  
Graph  
Dynamic Programming  
shortest path

## Find if there is a path between two vertices in a directed graph

Given a Directed Graph and two vertices in it, check whether there is a path from the first given vertex to second. For example, in the following graph, there is a path from vertex 1 to 3. As another example, there is no path from 3 to 0.



We can either use **Breadth First Search (BFS)** or **Depth First Search (DFS)** to find path between two vertices. Take the first vertex as source in BFS (or DFS), follow the standard BFS (or DFS). If we see the second vertex in our traversal, then return true. Else return false.

Following are C++, Java and Python codes that use BFS for finding reachability of second vertex from first vertex.

### C++

```

// C++ program to check if there is exist a path between two vertices
// of a graph.
#include<iostream>
#include <list>
using namespace std;

// This class represents a directed graph using adjacency list
// representation
class Graph
{
 int V; // No. of vertices

```

```

list<int> *adj; // Pointer to an array containing adjacency lists
public:
Graph(int V); // Constructor
void addEdge(int v, int w); // function to add an edge to graph
bool isReachable(int s, int d);
};

Graph::Graph(int V)
{
 this->V = V;
 adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
 adj[v].push_back(w); // Add w to v's list.
}

// A BFS based function to check whether d is reachable from s.
bool Graph::isReachable(int s, int d)
{
 // Base case
 if (s == d)
 return true;

 // Mark all the vertices as not visited
 bool *visited = new bool[V];
 for (int i = 0; i < V; i++)
 visited[i] = false;

 // Create a queue for BFS
 list<int> queue;

 // Mark the current node as visited and enqueue it
 visited[s] = true;
 queue.push_back(s);

 // it will be used to get all adjacent vertices of a vertex
 list<int>::iterator i;

 while (!queue.empty())
 {
 // Dequeue a vertex from queue and print it
 s = queue.front();
 queue.pop_front();

 // Get all adjacent vertices of the dequeued vertex s
 // If a adjacent has not been visited, then mark it visited
 // and enqueue it
 for (i = adj[s].begin(); i != adj[s].end(); ++i)
 {
 // If this adjacent node is the destination node, then
 // return true
 if (*i == d)
 return true;

 // Else, continue to do BFS
 if (!visited[*i])
 {
 visited[*i] = true;
 queue.push_back(*i);
 }
 }
 }

 // If BFS is complete without visiting d
 return false;
}

// Driver program to test methods of graph class
int main()
{
 // Create a graph given in the above diagram
 Graph g(4);
 g.addEdge(0, 1);
 g.addEdge(0, 2);
 g.addEdge(1, 2);
 g.addEdge(2, 0);
 g.addEdge(2, 3);
 g.addEdge(3, 3);

 int u = 1, v = 3;
 if(g.isReachable(u, v))
 cout<< "\n There is a path from " << u << " to " << v;
 else
 cout<< "\n There is no path from " << u << " to " << v;

 u = 3, v = 1;
 if(g.isReachable(u, v))
 cout<< "\n There is a path from " << u << " to " << v;
 else
 cout<< "\n There is no path from " << u << " to " << v;

 return 0;
}

```

## Java

```

// Java program to check if there is exist a path between two vertices
// of a graph.
import java.io.*;
import java.util.*;
import java.util.LinkedList;

```

```

// This class represents a directed graph using adjacency list
// representation
class Graph
{
 private int V; // No. of vertices
 private LinkedList<Integer> adj[]; //Adjacency List

 //Constructor
 Graph(int v)
 {
 V = v;
 adj = new LinkedList[v];
 for (int i=0; i<v; ++i)
 adj[i] = new LinkedList();
 }

 //Function to add an edge into the graph
 void addEdge(int v,int w) { adj[v].add(w); }

 //prints BFS traversal from a given source s
 Boolean isReachable(int s, int d)
 {
 LinkedList<Integer>temp;

 // Mark all the vertices as not visited(By default set
 // as false)
 boolean visited[] = new boolean[V];

 // Create a queue for BFS
 LinkedList<Integer> queue = new LinkedList<Integer>();

 // Mark the current node as visited and enqueue it
 visited[s]=true;
 queue.add(s);

 // 'i' will be used to get all adjacent vertices of a vertex
 Iterator<Integer> i;
 while (queue.size()!=0)
 {
 // Dequeue a vertex from queue and print it
 s = queue.poll();

 int n;
 i = adj[s].listIterator();

 // Get all adjacent vertices of the dequeued vertex s
 // If a adjacent has not been visited, then mark it
 // visited and enqueue it
 while (i.hasNext())
 {
 n = i.next();

 // If this adjacent node is the destination node,
 // then return true
 if (n==d)
 return true;

 // Else, continue to do BFS
 if (!visited[n])
 {
 visited[n] = true;
 queue.add(n);
 }
 }
 }

 // If BFS is complete without visited d
 return false;
 }

 // Driver method
 public static void main(String args[])
 {
 // Create a graph given in the above diagram
 Graph g = new Graph(4);
 g.addEdge(0, 1);
 g.addEdge(0, 2);
 g.addEdge(1, 2);
 g.addEdge(2, 0);
 g.addEdge(2, 3);
 g.addEdge(3, 3);

 int u = 1;
 int v = 3;
 if (g.isReachable(u, v))
 System.out.println("There is a path from " + u + " to " + v);
 else
 System.out.println("There is no path from " + u + " to " + v);

 u = 3;
 v = 1;
 if (g.isReachable(u, v))
 System.out.println("There is a path from " + u + " to " + v);
 else
 System.out.println("There is no path from " + u + " to " + v);
 }
}
// This code is contributed by Aakash Hasija

```

## Python

```

program to check if there is exist a path between two vertices
of a graph

```

```

from collections import defaultdict

#This class represents a directed graph using adjacency list representation
class Graph:

 def __init__(self,vertices):
 self.V= vertices #No. of vertices
 self.graph = defaultdict(list) # default dictionary to store graph

 # function to add an edge to graph
 def addEdge(self,u,v):
 self.graph[u].append(v)

 # Use BFS to check path between s and d:
 def isReachable(self, s, d):
 # Mark all the vertices as not visited
 visited =[False]*(self.V)

 # Create a queue for BFS
 queue=[]

 # Mark the source node as visited and enqueue it
 queue.append(s)
 visited[s] = True

 while queue:

 #Dequeue a vertex from queue
 n = queue.pop(0)

 # If this adjacent node is the destination node,
 # then return true
 if n == d:
 return True

 # Else, continue to do BFS
 for i in self.graph[n]:
 if visited[i] == False:
 queue.append(i)
 visited[i] = True
 # If BFS is complete without visited d
 return False

 # Create a graph given in the above diagram
 g = Graph(4)
 g.addEdge(0, 1)
 g.addEdge(0, 2)
 g.addEdge(1, 2)
 g.addEdge(2, 0)
 g.addEdge(2, 3)
 g.addEdge(3, 3)

 u =1; v = 3

 if g.isReachable(u, v):
 print("There is a path from %d to %d" % (u,v))
 else :
 print("There is no path from %d to %d" % (u,v))

 u = 3; v = 1
 if g.isReachable(u, v) :
 print("There is a path from %d to %d" % (u,v))
 else :
 print("There is no path from %d to %d" % (u,v))

#This code is contributed by Neelam Yadav

```

Output:

```

There is a path from 1 to 3
There is no path from 3 to 1

```

As an exercise, try an extended version of the problem where the complete path between two vertices is also needed.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner Company Wise Coding Practice

Graph  
BFS

### Check if a graph is strongly connected | Set 1 (Kosaraju using DFS)

Given a directed graph, find out whether the graph is strongly connected or not. A directed graph is strongly connected if there is a path between any two pair of vertices. For example, following is a strongly connected graph.

connectivity3



**It is easy for undirected graph**, we can just do a BFS and DFS starting from any vertex. If BFS or DFS visits all vertices, then the given undirected graph is connected. This approach won't work for a directed graph. For example, consider the following graph which is not strongly connected. If we start DFS (or BFS) from vertex 0, we can reach all vertices, but if we start from any other vertex, we cannot reach all vertices.

**How to do for directed graph?**

A simple idea is to use a all pair shortest path algorithm like [Floyd Warshall](#) or [find Transitive Closure](#) of graph. Time complexity of this method would be  $O(v^3)$ .

We can also do **DFS V times** starting from every vertex. If any DFS, doesn't visit all vertices, then graph is not strongly connected. This algorithm takes  $O(V*(V+E))$  time which can be same as transitive closure for a dense graph.

A better idea can be **Strongly Connected Components (SCC) algorithm**. We can find all SCCs in  $O(V+E)$  time. If number of SCCs is one, then graph is strongly connected. The algorithm for SCC does extra work as it finds all SCCs.

Following is **Kosaraju's DFS based simple algorithm that does two DFS traversals** of graph:

1) Initialize all vertices as not visited.

2) Do a DFS traversal of graph starting from any arbitrary vertex v. If DFS traversal doesn't visit all vertices, then return false.

3) Reverse all arcs (or find transpose or reverse of graph)

4) Mark all vertices as not-visited in reversed graph.

5) Do a DFS traversal of reversed graph starting from same vertex v (Same as step 2). If DFS traversal doesn't visit all vertices, then return false. Otherwise return true.

The idea is, if every node can be reached from a vertex v, and every node can reach v, then the graph is strongly connected. In step 2, we check if all vertices are reachable from v. In step 4, we check if all vertices can reach v (In reversed graph, if all vertices are reachable from v, then all vertices can reach v in original graph).

Following is the implementation of above algorithm.

**C++**

```
// C++ program to check if a given directed graph is strongly
// connected or not
#include <iostream>
#include <list>
#include <stack>
using namespace std;

class Graph
{
 int V; // No. of vertices
 list<int> *adj; // An array of adjacency lists

 // A recursive function to print DFS starting from v
 void DFSUtil(int v, bool visited[]);

public:
 // Constructor and Destructor
 Graph(int V) { this->V = V; adj = new list<int>[V]; }
 ~Graph() { delete [] adj; }

 // Method to add an edge
 void addEdge(int v, int w);

 // The main function that returns true if the graph is strongly
 // connected, otherwise false
 bool isSC();

 // Function that returns reverse (or transpose) of this graph
 Graph getTranspose();
};

// A recursive function to print DFS starting from v
void Graph::DFSUtil(int v, bool visited[])
{
 // Mark the current node as visited and print it
 visited[v] = true;

 // Recur for all the vertices adjacent to this vertex
 list<int>::iterator i;
 for(i = adj[v].begin(); i != adj[v].end(); ++i)
 if (!visited[*i])
 DFSUtil(*i, visited);
}

// Function that returns reverse (or transpose) of this graph
Graph Graph::getTranspose()
{
 Graph g(V);
 for (int v = 0; v < V; v++)
 {
 // Recur for all the vertices adjacent to this vertex
 list<int>::iterator i;
 for(i = adj[v].begin(); i != adj[v].end(); ++i)
 {
 g.adj[*i].push_back(v);
 }
 }
 return g;
}

void Graph::addEdge(int v, int w)
{
 adj[v].push_back(w); // Add w to v's list.
}

// The main function that returns true if graph is strongly connected
bool Graph::isSC()
{
 // Step 1: Mark all the vertices as not visited (For first DFS)
 bool visited[V];
 for (int i = 0; i < V; i++)
 visited[i] = false;

 // Step 2: Do DFS traversal starting from first vertex.
}
```

```

DFSUtil(0, visited);

// If DFS traversal doesn't visit all vertices, then return false.
for (int i = 0; i < V; i++)
 if (visited[i] == false)
 return false;

// Step 3: Create a reversed graph
Graph gr = getTranspose();

// Step 4: Mark all the vertices as not visited (For second DFS)
for (int i = 0; i < V; i++)
 visited[i] = false;

// Step 5: Do DFS for reversed graph starting from first vertex.
// Starting Vertex must be same starting point of first DFS
gr.DFSUtil(0, visited);

// If all vertices are not visited in second DFS, then
// return false
for (int i = 0; i < V; i++)
 if (visited[i] == false)
 return false;

return true;
}

// Driver program to test above functions
int main()
{
 // Create graphs given in the above diagrams
 Graph g1(5);
 g1.addEdge(0, 1);
 g1.addEdge(1, 2);
 g1.addEdge(2, 3);
 g1.addEdge(3, 0);
 g1.addEdge(2, 4);
 g1.addEdge(4, 2);
 g1.isSC() ? cout << "Yes\n" : cout << "No\n";

 Graph g2(4);
 g2.addEdge(0, 1);
 g2.addEdge(1, 2);
 g2.addEdge(2, 3);
 g2.isSC() ? cout << "Yes\n" : cout << "No\n";

 return 0;
}

```

## Java

```

// Java program to check if a given directed graph is strongly
// connected or not
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
 private int V; // No. of vertices
 private LinkedList<Integer> adj[]; //Adjacency List

 //Constructor
 Graph(int v)
 {
 V = v;
 adj = new LinkedList[v];
 for (int i=0; i<v; ++i)
 adj[i] = new LinkedList();
 }

 //Function to add an edge into the graph
 void addEdge(int v,int w) { adj[v].add(w);}

 // A recursive function to print DFS starting from v
 void DFSUtil(int v,Boolean visited[])
 {
 // Mark the current node as visited and print it
 visited[v] = true;

 int n;

 // Recur for all the vertices adjacent to this vertex
 Iterator<Integer> i = adj[v].iterator();
 while (i.hasNext())
 {
 n = i.next();
 if (!visited[n])
 DFSUtil(n,visited);
 }
 }

 // Function that returns transpose of this graph
 Graph getTranspose()
 {
 Graph g = new Graph(V);
 for (int v = 0; v < V; v++)
 {
 // Recur for all the vertices adjacent to this vertex
 Iterator<Integer> i = adj[v].listIterator();
 while (i.hasNext())
 g.adj[i.next()].add(v);
 }
 return g;
 }
}

```

```

 }
 return g;
}

// The main function that returns true if graph is strongly
// connected
Boolean isSC()
{
 // Step 1: Mark all the vertices as not visited
 // (For first DFS)
 Boolean visited[] = new Boolean[V];
 for (int i = 0; i < V; i++)
 visited[i] = false;

 // Step 2: Do DFS traversal starting from first vertex.
 DFSUtil(0, visited);

 // If DFS traversal doesn't visit all vertices, then
 // return false.
 for (int i = 0; i < V; i++)
 if (visited[i] == false)
 return false;

 // Step 3: Create a reversed graph
 Graph gr = getTranspose();

 // Step 4: Mark all the vertices as not visited (For
 // second DFS)
 for (int i = 0; i < V; i++)
 visited[i] = false;

 // Step 5: Do DFS for reversed graph starting from
 // first vertex. Staring Vertex must be same starting
 // point of first DFS
 gr.DFSUtil(0, visited);

 // If all vertices are not visited in second DFS, then
 // return false
 for (int i = 0; i < V; i++)
 if (visited[i] == false)
 return false;

 return true;
}

public static void main(String args[])
{
 // Create graphs given in the above diagrams
 Graph g1 = new Graph(5);
 g1.addEdge(0, 1);
 g1.addEdge(1, 2);
 g1.addEdge(2, 3);
 g1.addEdge(3, 0);
 g1.addEdge(2, 4);
 g1.addEdge(4, 2);
 if (g1.isSC())
 System.out.println("Yes");
 else
 System.out.println("No");

 Graph g2 = new Graph(4);
 g2.addEdge(0, 1);
 g2.addEdge(1, 2);
 g2.addEdge(2, 3);
 if (g2.isSC())
 System.out.println("Yes");
 else
 System.out.println("No");
}
}

// This code is contributed by Aakash Hasija

```

## Python

```

Python program to check if a given directed graph is strongly
connected or not

from collections import defaultdict

#This class represents a directed graph using adjacency list representation
class Graph:

 def __init__(self,vertices):
 self.V= vertices #No. of vertices
 self.graph = defaultdict(list) # default dictionary to store graph

 # function to add an edge to graph
 def addEdge(self,u,v):
 self.graph[u].append(v)

 #A function used by isSC() to perform DFS
 def DFSUtil(self,v,visited):

 # Mark the current node as visited
 visited[v]= True

 #Recur for all the vertices adjacent to this vertex
 for i in self.graph[v]:
 if visited[i]==False:
 self.DFSUtil(i,visited)

 # Function that returns reverse (or transpose) of this graph

```

```

def getTranspose(self):
 g = Graph(self.V)

 # Recur for all the vertices adjacent to this vertex
 for i in self.graph:
 for j in self.graph[i]:
 g.addEdge(j,i)

 return g

The main function that returns true if graph is strongly connected
def isSC(self):

 # Step 1: Mark all the vertices as not visited (For first DFS)
 visited =[False]*(self.V)

 # Step 2: Do DFS traversal starting from first vertex.
 self.DFSUtil(0,visited)

 # If DFS traversal doesn't visit all vertices, then return false
 if any(i == False for i in visited):
 return False

 # Step 3: Create a reversed graph
 gr = self.getTranspose()

 # Step 4: Mark all the vertices as not visited (For second DFS)
 visited =[False]*(self.V)

 # Step 5: Do DFS for reversed graph starting from first vertex.
 # Starting Vertex must be same starting point of first DFS
 gr.DFSUtil(0,visited)

 # If all vertices are not visited in second DFS, then
 # return false
 if any(i == False for i in visited):
 return False

 return True

Create a graph given in the above diagram
g1 = Graph(5)
g1.addEdge(0, 1)
g1.addEdge(1, 2)
g1.addEdge(2, 3)
g1.addEdge(3, 0)
g1.addEdge(2, 4)
g1.addEdge(4, 2)
print "Yes" if g1.isSC() else "No"

g2 = Graph(4)
g2.addEdge(0, 1)
g2.addEdge(1, 2)
g2.addEdge(2, 3)
print "Yes" if g2.isSC() else "No"

```

#This code is contributed by Neelam Yadav

Output:

Yes  
No

**Time Complexity:** Time complexity of above implementation is same as [Depth First Search](#) which is  $O(V+E)$  if the graph is represented using adjacency list representation.

**Can we improve further?**

The above approach requires two traversals of graph. We can find whether a graph is strongly connected or not in one traversal using [Tarjan's Algorithm to find Strongly Connected Components](#).

**Exercise:**

Can we use BFS instead of DFS in above algorithm? See [this](#).

**References:**

<http://www.ieor.berkeley.edu/~hochbaum/files/ieor266-2012.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner Company Wise Coding Practice

Graph  
BFS  
DFS  
graph-connectivity

## Articulation Points (or Cut Vertices) in a Graph

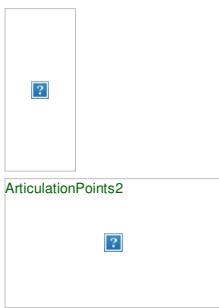
A vertex in an undirected connected graph is an articulation point (or cut vertex) iff removing it (and edges through it) disconnects the graph. Articulation points represent vulnerabilities in a connected network – single points whose failure would split the network into 2 or more disconnected components. They are useful for designing reliable networks.

For a disconnected undirected graph, an articulation point is a vertex removing which increases number of connected components.

Following are some example graphs with articulation points encircled with red color.

ArticulationPoints





ArticulationPoints2



#### How to find all articulation points in a given graph?

A simple approach is to one by one remove all vertices and see if removal of a vertex causes disconnected graph. Following are steps of simple approach for connected graph.

1) For every vertex v, do following

.....a) Remove v from graph

.....b) See if the graph remains connected (We can either use BFS or DFS)

.....c) Add v back to the graph

Time complexity of above method is  $O(V*(V+E))$  for a graph represented using adjacency list. Can we do better?

#### A $O(V+E)$ algorithm to find all Articulation Points (APs)

The idea is to use DFS (Depth First Search). In DFS, we follow vertices in tree form called DFS tree. In DFS tree, a vertex u is parent of another vertex v, if v is discovered by u (obviously v is an adjacent of u in graph). In DFS tree, a vertex u is articulation point if one of the following two conditions is true.

1) u is root of DFS tree and it has at least two children.

2) u is not root of DFS tree and it has a child v such that no vertex in subtree rooted with v has a back edge to one of the ancestors (in DFS tree) of u.

Following figure shows same points as above with one additional point that a leaf in DFS Tree can never be an articulation point. (Source Ref 2)

ArticulationPoints4



We do DFS traversal of given graph with additional code to find out Articulation Points (APs). In DFS traversal, we maintain a parent[] array where parent[u] stores parent of vertex u. Among the above mentioned two cases, the first case is simple to detect. For every vertex, count children. If currently visited vertex u is root (parent[u] is NIL) and has more than two children, print it.

How to handle second case? The second case is trickier. We maintain an array disc[] to store discovery time of vertices. For every node u, we need to find out the earliest visited vertex (the vertex with minimum discovery time) that can be reached from subtree rooted with u. So we maintain an additional array low[] which is defined as follows.

```
low[u] = min(disc[u], disc[w])
where w is an ancestor of u and there is a back edge from
some descendant of u to w.
```

Following are C++, Java and Python implementation of Tarjan's algorithm for finding articulation points.

#### C++

```
// A C++ program to find articulation points in an undirected graph
#include<iostream>
#include <list>
#define NIL -1
using namespace std;

// A class that represents an undirected graph
class Graph
{
 int V; // No. of vertices
 list<int> *adj; // A dynamic array of adjacency lists
 void APUtil(int v, bool visited[], int disc[], int low[],
 int parent[], bool ap[]);
public:
 Graph(int V); // Constructor
 void addEdge(int v, int w); // function to add an edge to graph
 void AP(); // prints articulation points
};

Graph::Graph(int V)
{
 this->V = V;
 adj = new list<int*>[V];
}

void Graph::addEdge(int v, int w)
{
 adj[v].push_back(w);
 adj[w].push_back(v); // Note: the graph is undirected
}

// A recursive function that finds articulation points using DFS traversal
// u --> The vertex to be visited next
// visited[] --> keeps tract of visited vertices
// disc[] --> Stores discovery times of visited vertices
// parent[] --> Stores parent vertices in DFS tree
// ap[] --> Stores articulation points
void Graph::APUtil(int u, bool visited[], int disc[],
 int low[], int parent[], bool ap[])
{
 // Initialize current discovery time
 disc[u] = low[u] = ++time;
 int children = 0;
 for (int v : adj[u]) {
 if (disc[v] == NIL) { // If v is not yet visited
 parent[v] = u;
 if (u == NIL) children++;
 else children++;
 APUtil(v, visited, disc, low, parent, ap);
 low[u] = min(low[u], low[v]);
 if (parent[u] == NIL && children > 2) // Root has more than 2 children
 ap[u] = true;
 if (parent[u] != NIL && low[v] >= disc[u]) // This is an articulation point
 ap[u] = true;
 } else if (v != parent[u]) { // If v is visited, and it is not parent of u
 low[u] = min(low[u], disc[v]);
 }
 }
}
```

```
{
 // A static variable is used for simplicity, we can avoid use of static
 // variable by passing a pointer.
 static int time = 0;

 // Count of children in DFS Tree
 int children = 0;

 // Mark the current node as visited
 visited[u] = true;

 // Initialize discovery time and low value
 disc[u] = low[u] = ++time;

 // Go through all vertices adjacent to this
 list<int>::iterator i;
 for (i = adj[u].begin(); i != adj[u].end(); ++i)
 {
 int v = *i; // v is current adjacent of u

 // If v is not visited yet, then make it a child of u
 // in DFS tree and recur for it
 if (!visited[v])
 {
 children++;
 parent[v] = u;
 APUtil(v, visited, disc, low, parent, ap);

 // Check if the subtree rooted with v has a connection to
 // one of the ancestors of u
 low[u] = min(low[u], low[v]);
 }
 }

 // u is an articulation point in following cases

 // (1) u is root of DFS tree and has two or more children.
 if (parent[u] == NIL && children > 1)
 ap[u] = true;

 // (2) If u is not root and low value of one of its children is more
 // than discovery value of u.
 if (parent[u] != NIL && low[v] >= disc[u])
 ap[u] = true;
 }

 // Update low value of u for parent function calls.
 else if (v != parent[u])
 low[u] = min(low[u], disc[v]);
}
}

// The function to do DFS traversal. It uses recursive function APUtil()
void Graph::AP()
{
 // Mark all the vertices as not visited
 bool *visited = new bool[V];
 int *disc = new int[V];
 int *low = new int[V];
 int *parent = new int[V];
 bool *ap = new bool[V]; // To store articulation points

 // Initialize parent and visited, and ap(articulation point) arrays
 for (int i = 0; i < V; i++)
 {
 parent[i] = NIL;
 visited[i] = false;
 ap[i] = false;
 }

 // Call the recursive helper function to find articulation points
 // in DFS tree rooted with vertex 'i'
 for (int i = 0; i < V; i++)
 if (visited[i] == false)
 APUtil(i, visited, disc, low, parent, ap);

 // Now ap[] contains articulation points, print them
 for (int i = 0; i < V; i++)
 if (ap[i] == true)
 cout << i << " ";
}

// Driver program to test above function
int main()
{
 // Create graphs given in above diagrams
 cout << "\nArticulation points in first graph \n";
 Graph g1(5);
 g1.addEdge(1, 0);
 g1.addEdge(0, 2);
 g1.addEdge(2, 1);
 g1.addEdge(0, 3);
 g1.addEdge(3, 4);
 g1.AP();

 cout << "\nArticulation points in second graph \n";
 Graph g2(4);
 g2.addEdge(0, 1);
 g2.addEdge(1, 2);
 g2.addEdge(2, 3);
 g2.AP();

 cout << "\nArticulation points in third graph \n";
 Graph g3(7);
 g3.addEdge(0, 1);
 g3.addEdge(1, 2);
 g3.addEdge(2, 0);
 g3.addEdge(1, 3);
 g3.addEdge(1, 4);
}
```

```
g3.addEdge(1, 6);
g3.addEdge(3, 5);
g3.addEdge(4, 5);
g3.AP();
```

```
return 0;
}
```

## Java

```
// A Java program to find articulation points in an undirected graph
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents an undirected graph using adjacency list
// representation
class Graph
{
 private int V; // No. of vertices

 // Array of lists for Adjacency List Representation
 private LinkedList<Integer> adj[][];
 int time = 0;
 static final int NIL = -1;

 // Constructor
 Graph(int v)
 {
 V = v;
 adj = new LinkedList[v];
 for (int i=0; i<v; ++i)
 adj[i] = new LinkedList();
 }

 //Function to add an edge into the graph
 void addEdge(int v, int w)
 {
 adj[v].add(w); // Add w to v's list.
 adj[w].add(v); //Add v to w's list
 }

 // A recursive function that find articulation points using DFS
 // u -> The vertex to be visited next
 // visited[] -> keeps tract of visited vertices
 // disc[] -> Stores discovery times of visited vertices
 // parent[] -> Stores parent vertices in DFS tree
 // ap[] -> Store articulation points
 void APUtil(int u, boolean visited[], int disc[],
 int low[], int parent[], boolean ap[])
 {

 // Count of children in DFS Tree
 int children = 0;

 // Mark the current node as visited
 visited[u] = true;

 // Initialize discovery time and low value
 disc[u] = low[u] = ++time;

 // Go through all vertices adjacent to this
 Iterator<Integer> i = adj[u].iterator();
 while (i.hasNext())
 {
 int v = i.next(); // v is current adjacent of u

 // If v is not visited yet, then make it a child of u
 // in DFS tree and recur for it
 if (!visited[v])
 {
 children++;
 parent[v] = u;
 APUtil(v, visited, disc, low, parent, ap);

 // Check if the subtree rooted with v has a connection to
 // one of the ancestors of u
 low[u] = Math.min(low[u], low[v]);
 }

 // u is an articulation point in following cases

 // (1) u is root of DFS tree and has two or more children.
 if (parent[u] == NIL && children > 1)
 ap[u] = true;

 // (2) If u is not root and low value of one of its children
 // is more than discovery value of u.
 if (parent[u] != NIL && low[v] >= disc[u])
 ap[u] = true;
 }

 // Update low value of u for parent function calls.
 else if (v != parent[u])
 low[u] = Math.min(low[u], disc[v]);
 }
}

// The function to do DFS traversal. It uses recursive function APUtil()
void AP()
{
 // Mark all the vertices as not visited
 boolean visited[] = new boolean[V];
 int disc[] = new int[V];
 int low[] = new int[V];
```

```

int parent[] = new int[V];
boolean ap[] = new boolean[V]; // To store articulation points

// Initialize parent and visited, and ap(articulation point)
// arrays
for (int i = 0; i < V; i++)
{
 parent[i] = NIL;
 visited[i] = false;
 ap[i] = false;
}

// Call the recursive helper function to find articulation
// points in DFS tree rooted with vertex 'i'
for (int i = 0; i < V; i++)
 if (visited[i] == false)
 APUtil(i, visited, disc, low, parent, ap);

// Now ap[] contains articulation points, print them
for (int i = 0; i < V; i++)
 if (ap[i] == true)
 System.out.print(i + " ");
}

// Driver method
public static void main(String args[])
{
 // Create graphs given in above diagrams
 System.out.println("Articulation points in first graph");
 Graph g1 = new Graph(5);
 g1.addEdge(1, 0);
 g1.addEdge(0, 2);
 g1.addEdge(2, 1);
 g1.addEdge(0, 3);
 g1.addEdge(3, 4);
 g1.AP();
 System.out.println();

 System.out.println("Articulation points in Second graph");
 Graph g2 = new Graph(4);
 g2.addEdge(0, 1);
 g2.addEdge(1, 2);
 g2.addEdge(2, 3);
 g2.AP();
 System.out.println();

 System.out.println("Articulation points in Third graph ");
 Graph g3 = new Graph(7);
 g3.addEdge(0, 1);
 g3.addEdge(1, 2);
 g3.addEdge(2, 0);
 g3.addEdge(1, 3);
 g3.addEdge(1, 4);
 g3.addEdge(1, 6);
 g3.addEdge(3, 5);
 g3.addEdge(4, 5);
 g3.AP();
}
}

// This code is contributed by Aakash Hasija

```

## Python

```

Python program to find articulation points in an undirected graph

from collections import defaultdict

#This class represents an undirected graph
#using adjacency list representation
class Graph:

 def __init__(self,vertices):
 self.v= vertices #No. of vertices
 self.graph = defaultdict(list) # default dictionary to store graph
 self.Time = 0

 # function to add an edge to graph
 def addEdge(self,u,v):
 self.graph[u].append(v)
 self.graph[v].append(u)

 """A recursive function that finds articulation points
 using DFS traversal
 u --> The vertex to be visited next
 visited[] --> keeps tract of visited vertices
 disc[] --> Stores discovery times of visited vertices
 parent[] --> Stores parent vertices in DFS tree
 ap[] --> Store articulation points"""
 def APUtil(self,u, visited, ap, parent, low, disc):

 #Count of children in current node
 children =0

 # Mark the current node as visited and print it
 visited[u]= True

 # Initialize discovery time and low value
 disc[u] = self.Time
 low[u] = self.Time
 self.Time += 1

 # Recur for all the vertices adjacent to this vertex
 for v in self.graph[u]:
 # If v is not visited yet, then make it a child of u
 if visited[v]== False:

```

```

in DFS tree and recur for it
if visited[v] == False :
 parent[v] = u
 children += 1
 self.APUtil(v, visited, ap, parent, low, disc)

Check if the subtree rooted with v has a connection to
one of the ancestors of u
low[u] = min(low[u], low[v])

u is an articulation point in following cases
(1) u is root of DFS tree and has two or more children.
if parent[u] == -1 and children > 1:
 ap[u] = True

#(2) If u is not root and low value of one of its children is more
than discovery value of u.
if parent[u] != -1 and low[v] >= disc[u]:
 ap[u] = True

Update low value of u for parent function calls
elif v != parent[u]:
 low[u] = min(low[u], disc[v])

#The function to do DFS traversal. It uses recursive APUtil()
def AP(self):

 # Mark all the vertices as not visited
 # and Initialize parent and visited,
 # and ap(articulation point) arrays
 visited = [False] * (self.V)
 disc = [float("Inf")] * (self.V)
 low = [float("Inf")] * (self.V)
 parent = [-1] * (self.V)
 ap = [False] * (self.V) #To store articulation points

 # Call the recursive helper function
 # to find articulation points
 # in DFS tree rooted with vertex 'i'
 for i in range(self.V):
 if visited[i] == False:
 self.APUtil(i, visited, ap, parent, low, disc)

 for index, value in enumerate (ap):
 if value == True: print index,

 # Create a graph given in the above diagram
 g1 = Graph(5)
 g1.addEdge(1, 0)
 g1.addEdge(0, 2)
 g1.addEdge(2, 1)
 g1.addEdge(0, 3)
 g1.addEdge(3, 4)

 print "\nArticulation points in first graph "
 g1.AP()

 g2 = Graph(4)
 g2.addEdge(0, 1)
 g2.addEdge(1, 2)
 g2.addEdge(2, 3)
 print "\nArticulation points in second graph "
 g2.AP()

 g3 = Graph (7)
 g3.addEdge(0, 1)
 g3.addEdge(1, 2)
 g3.addEdge(2, 0)
 g3.addEdge(1, 3)
 g3.addEdge(1, 4)
 g3.addEdge(1, 6)
 g3.addEdge(3, 5)
 g3.addEdge(4, 5)
 print "\nArticulation points in third graph "
 g3.AP()

#This code is contributed by Neelam Yadav

```

Output:

```

Articulation points in first graph
0 3
Articulation points in second graph
1 2
Articulation points in third graph
1

```

**Time Complexity:** The above function is simple DFS with additional arrays. So time complexity is same as DFS which is  $O(V+E)$  for adjacency list representation of graph.

**References:**

<https://www.cs.washington.edu/education/courses/421/04su/slides/artic.pdf>  
<http://www.slideshare.net/TraianRebedea/algoritm-design-and-complexity-course-8>  
[http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250\\_Weiss/L25-Connectivity.htm](http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250_Weiss/L25-Connectivity.htm)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner Company Wise Coding Practice

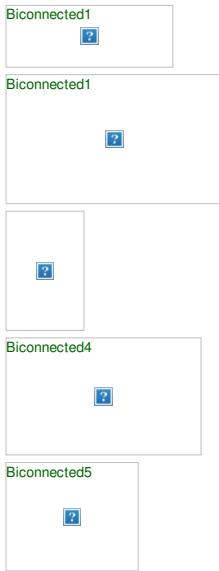
Graph  
BFS  
DFS  
graph-connectivity

## Biconnected graph

An undirected graph is called Biconnected if there are two vertex-disjoint paths between any two vertices. In a Biconnected Graph, there is a simple cycle through any two vertices.

By convention, two nodes connected by an edge form a biconnected graph, but this does not verify the above properties. For a graph with more than two vertices, the above properties must be there for it to be Biconnected.

Following are some examples.



See [this](#) for more examples.

### How to find if a given graph is Biconnected or not?

A connected graph is Biconnected if it is connected and doesn't have any [Articulation Point](#). We mainly need to check two things in a graph.

- 1) The graph is connected.
- 2) There is not articulation point in graph.

We start from any vertex and do DFS traversal. In DFS traversal, we check if there is any articulation point. If we don't find any articulation point, then the graph is Biconnected. Finally, we need to check whether all vertices were reachable in DFS or not. If all vertices were not reachable, then the graph is not even connected.

Following is C++ implementation of above approach.

### C++

```
// A C++ program to find if a given undirected graph is
// biconnected
#include <iostream>
#include <list>
#define NIL -1
using namespace std;

// A class that represents an undirected graph
class Graph
{
 int V; // No. of vertices
 list<int*> *adj; // A dynamic array of adjacency lists
 bool isBCUtil(int v, bool visited[], int disc[], int low[],
 int parent[]);
public:
 Graph(int V); // Constructor
 void addEdge(int v, int w); // to add an edge to graph
 bool isBC(); // returns true if graph is Biconnected
};

Graph::Graph(int V)
{
 this->V = V;
 adj = new list<int*>[V];
}

void Graph::addEdge(int v, int w)
{
 adj[v].push_back(w);
 adj[w].push_back(v); // Note: the graph is undirected
}

// A recursive function that returns true if there is an articulation
// point in given graph, otherwise returns false.
// This function is almost same as isAPUtil() here (http://goo.gl/Mc9Fw)
// u --> The vertex to be visited next
// visited[] --> keeps tract of visited vertices
// disc[] --> Stores discovery times of visited vertices
// parent[] --> Stores parent vertices in DFS tree
bool Graph::isBCUtil(int u, bool visited[], int disc[], int low[], int parent[])
{
 // A static variable is used for simplicity, we can avoid use of static
 // variable by passing a pointer.
 static int time = 0;

 // Count of children in DFS Tree
 int children = 0;
```

```

// Mark the current node as visited
visited[u] = true;

// Initialize discovery time and low value
disc[u] = low[u] = ++time;

// Go through all vertices adjacent to this
list<int>::iterator i;
for (i = adj[u].begin(); i != adj[u].end(); ++i)
{
 int v = *i; // v is current adjacent of u

 // If v is not visited yet, then make it a child of u
 // in DFS tree and recur for it
 if (!visited[v])
 {
 children++;
 parent[v] = u;

 // check if subgraph rooted with v has an articulation point
 if (isBCUtil(v, visited, disc, low, parent))
 return true;

 // Check if the subtree rooted with v has a connection to
 // one of the ancestors of u
 low[u] = min(low[u], low[v]);
 }
}

// u is an articulation point in following cases

// (1) u is root of DFS tree and has two or more children.
if (parent[u] == NIL && children > 1)
 return true;

// (2) If u is not root and low value of one of its children is
// more than discovery value of u.
if (parent[u] != NIL && low[v] >= disc[u])
 return true;
}

// Update low value of u for parent function calls.
else if (v != parent[u])
 low[u] = min(low[u], disc[v]);
}
return false;
}

// The main function that returns true if graph is Biconnected,
// otherwise false. It uses recursive function isBCUtil()
bool Graph::isBC()
{
 // Mark all the vertices as not visited
 bool *visited = new bool[V];
 int *disc = new int[V];
 int *low = new int[V];
 int *parent = new int[V];

 // Initialize parent and visited, and ap(articulation point)
 // arrays
 for (int i = 0; i < V; i++)
 {
 parent[i] = NIL;
 visited[i] = false;
 }

 // Call the recursive helper function to find if there is an articulation
 // point in given graph. We do DFS traversal starting from vertex 0
 if (isBCUtil(0, visited, disc, low, parent) == true)
 return false;

 // Now check whether the given graph is connected or not. An undirected
 // graph is connected if all vertices are reachable from any starting
 // point (we have taken 0 as starting point)
 for (int i = 0; i < V; i++)
 if (visited[i] == false)
 return false;

 return true;
}

// Driver program to test above function
int main()
{
 // Create graphs given in above diagrams
 Graph g1(2);
 g1.addEdge(0, 1);
 g1.isBC() ? cout << "Yes\n" : cout << "No\n";

 Graph g2(5);
 g2.addEdge(1, 0);
 g2.addEdge(0, 2);
 g2.addEdge(2, 1);
 g2.addEdge(0, 3);
 g2.addEdge(3, 4);
 g2.addEdge(2, 4);
 g2.isBC() ? cout << "Yes\n" : cout << "No\n";

 Graph g3(3);
 g3.addEdge(0, 1);
 g3.addEdge(1, 2);
 g3.isBC() ? cout << "Yes\n" : cout << "No\n";

 Graph g4(5);
 g4.addEdge(1, 0);
 g4.addEdge(0, 2);
 g4.addEdge(2, 1);
 g4.addEdge(0, 3);
}

```

```

g4.addEdge(3, 4);
g4.isBC()? cout << "Yes\n" : cout << "No\n";

Graph g5(3);
g5.addEdge(0, 1);
g5.addEdge(1, 2);
g5.addEdge(2, 0);
g5.isBC()? cout << "Yes\n" : cout << "No\n";

return 0;
}

```

## Java

```

// A Java program to find if a given undirected graph is
// biconnected
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents a directed graph using adjacency
// list representation
class Graph
{
 private int V; // No. of vertices

 // Array of lists for Adjacency List Representation
 private LinkedList<Integer> adj[];

 int time = 0;
 static final int NIL = -1;

 // Constructor
 Graph(int v)
 {
 V = v;
 adj = new LinkedList[v];
 for (int i=0; i<v; ++i)
 adj[i] = new LinkedList();
 }

 //Function to add an edge into the graph
 void addEdge(int v, int w)
 {
 adj[v].add(w); //Note that the graph is undirected.
 adj[w].add(v);
 }

 // A recursive function that returns true if there is an articulation
 // point in given graph, otherwise returns false.
 // This function is almost same as isAPUtil() @ http://goo.gl/Me9Fw
 // u --> The vertex to be visited next
 // visited[] --> keeps tract of visited vertices
 // disc[] --> Stores discovery times of visited vertices
 // parent[] --> Stores parent vertices in DFS tree
 boolean isBCUtil(int u, boolean visited[], int disc[], int low[],
 int parent[])
 {

 // Count of children in DFS Tree
 int children = 0;

 // Mark the current node as visited
 visited[u] = true;

 // Initialize discovery time and low value
 disc[u] = low[u] = ++time;

 // Go through all vertices adjacent to this
 Iterator<Integer> i = adj[u].iterator();
 while (i.hasNext())
 {
 int v = i.next(); // v is current adjacent of u

 // If v is not visited yet, then make it a child of u
 // in DFS tree and recur for it
 if (!visited[v])
 {
 children++;
 parent[v] = u;

 // check if subgraph rooted with v has an articulation point
 if (isBCUtil(v, visited, disc, low, parent))
 return true;

 // Check if the subtree rooted with v has a connection to
 // one of the ancestors of u
 low[u] = Math.min(low[u], low[v]);
 }
 }

 // u is an articulation point in following cases

 // (1) u is root of DFS tree and has two or more children.
 if (parent[u] == NIL && children > 1)
 return true;

 // (2) If u is not root and low value of one of its
 // child is more than discovery value of u.
 if (parent[u] != NIL && low[v] >= disc[u])
 return true;
 }

 // Update low value of u for parent function calls.
 else if (v != parent[u])
 low[u] = Math.min(low[u], disc[v]);
}

```

```

 }
 return false;
}

// The main function that returns true if graph is Biconnected,
// otherwise false. It uses recursive function isBCUtil()
boolean isBC()
{
 // Mark all the vertices as not visited
 boolean visited[] = new boolean[V];
 int disc[] = new int[V];
 int low[] = new int[V];
 int parent[] = new int[V];

 // Initialize parent and visited, and ap(articulation point)
 // arrays
 for (int i = 0; i < V; i++)
 {
 parent[i] = NIL;
 visited[i] = false;
 }

 // Call the recursive helper function to find if there is an
 // articulation/ point in given graph. We do DFS traversal
 // starting from vertex 0
 if (isBCUtil(0, visited, disc, low, parent) == true)
 return false;

 // Now check whether the given graph is connected or not.
 // An undirected graph is connected if all vertices are
 // reachable from any starting point (we have taken 0 as
 // starting point)
 for (int i = 0; i < V; i++)
 if (visited[i] == false)
 return false;

 return true;
}

// Driver method
public static void main(String args[])
{
 // Create graphs given in above diagrams
 Graph g1 = new Graph(2);
 g1.addEdge(0, 1);
 if (g1.isBC())
 System.out.println("Yes");
 else
 System.out.println("No");

 Graph g2 = new Graph(5);
 g2.addEdge(1, 0);
 g2.addEdge(0, 2);
 g2.addEdge(2, 1);
 g2.addEdge(0, 3);
 g2.addEdge(3, 4);
 g2.addEdge(2, 4);
 if (g2.isBC())
 System.out.println("Yes");
 else
 System.out.println("No");

 Graph g3 = new Graph(3);
 g3.addEdge(0, 1);
 g3.addEdge(1, 2);
 if (g3.isBC())
 System.out.println("Yes");
 else
 System.out.println("No");

 Graph g4 = new Graph(5);
 g4.addEdge(1, 0);
 g4.addEdge(0, 2);
 g4.addEdge(2, 1);
 g4.addEdge(0, 3);
 g4.addEdge(3, 4);
 if (g4.isBC())
 System.out.println("Yes");
 else
 System.out.println("No");

 Graph g5 = new Graph(3);
 g5.addEdge(0, 1);
 g5.addEdge(1, 2);
 g5.addEdge(2, 0);
 if (g5.isBC())
 System.out.println("Yes");
 else
 System.out.println("No");
}
}

// This code is contributed by Aakash Hasija

```

## Python

```

Python program to find if a given undirected graph is
biconnected

from collections import defaultdict

#This class represents an undirected graph using adjacency list representation
class Graph:

 def __init__(self,vertices):

```

```

self.V= vertices #No. of vertices
self.graph = defaultdict(list) # default dictionary to store graph
self.Time = 0

function to add an edge to graph
def addEdge(self,u,v):
 self.graph[u].append(v)
 self.graph[v].append(u)

"""A recursive function that returns true if there is an articulation
point in given graph, otherwise returns false.
This function is almost same as isAPUtil()
u --> The vertex to be visited next
visited[] --> keeps tract of visited vertices
disc[] --> Stores discovery times of visited vertices
parent[] --> Stores parent vertices in DFS tree"""
def isBCUtil(self,u, visited, parent, low, disc):

 #Count of children in current node
 children =0

 # Mark the current node as visited and print it
 visited[u]= True

 # Initialize discovery time and low value
 disc[u] = self.Time
 low[u] = self.Time
 self.Time += 1

 #Recur for all the vertices adjacent to this vertex
 for v in self.graph[u]:
 if v not in visited:
 # If v is not visited yet, then make it a child of u
 # in DFS tree and recur for it
 if visited[v] == False :
 parent[v] = u
 children += 1
 if self.isBCUtil(v, visited, parent, low, disc):
 return True

 # Check if the subtree rooted with v has a connection to
 # one of the ancestors of u
 low[u] = min(low[u], low[v])

 # u is an articulation point in following cases
 # (1) u is root of DFS tree and has two or more children.
 if parent[u] == -1 and children > 1:
 return True

 #(2) If u is not root and low value of one of its child is more
 # than discovery value of u.
 if parent[u] != -1 and low[v] >= disc[u]:
 return True

 elif v != parent[u]: # Update low value of u for parent function calls.
 low[u] = min(low[u], disc[v])

 return False

The main function that returns true if graph is Biconnected,
otherwise false. It uses recursive function isBCUtil()
def isBC(self):

 # Mark all the vertices as not visited and Initialize parent and visited,
 # and ap(articulation point) arrays
 visited = [False] * (self.V)
 disc = [float("Inf")] * (self.V)
 low = [float("Inf")] * (self.V)
 parent = [-1] * (self.V)

 # Call the recursive helper function to find if there is an
 # articulation points in given graph. We do DFS traversal starting
 # from vertex 0
 if self.isBCUtil(0, visited, parent, low, disc):
 return False

 """Now check whether the given graph is connected or not.
 An undirected graph is connected if all vertices are
 reachable from any starting point (we have taken 0 as
 starting point)"""
 if any(i == False for i in visited):
 return False

 return True

Create a graph given in the above diagram
g1 = Graph(2)
g1.addEdge(0, 1)
print "Yes" if g1.isBC() else "No"

g2 = Graph(5)
g2.addEdge(1, 0)
g2.addEdge(0, 2)
g2.addEdge(2, 1)
g2.addEdge(0, 3)
g2.addEdge(3, 4)
g2.addEdge(2, 4)
print "Yes" if g2.isBC() else "No"

g3 = Graph(3)
g3.addEdge(0, 1)
g3.addEdge(1, 2)
print "Yes" if g3.isBC() else "No"

g4 = Graph (5)

```

```

g4.addEdge(1, 0)
g4.addEdge(0, 2)
g4.addEdge(2, 1)
g4.addEdge(0, 3)
g4.addEdge(3, 4)
print "Yes" if g4.isBC() else "No"

g5 = Graph(3)
g5.addEdge(0, 1)
g5.addEdge(1, 2)
g5.addEdge(2, 0)
print "Yes" if g5.isBC() else "No"

#This code is contributed by Neelam Yadav

```

Output:

```

Yes
Yes
No
No
Yes

```

**Time Complexity:** The above function is a simple DFS with additional arrays. So time complexity is same as DFS which is  $O(V+E)$  for adjacency list representation of graph.

**References:**

<http://www.cs.purdue.edu/homes/ayg/CS251/slides/chap9d.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner Company Wise Coding Practice

Graph  
graph-connectivity

### Bridges in a graph

An edge in an undirected connected graph is a bridge iff removing it disconnects the graph. For a disconnected undirected graph, definition is similar, a bridge is an edge removing which increases number of connected components.

Like [Articulation Points](#), bridges represent vulnerabilities in a connected network and are useful for designing reliable networks. For example, in a wired computer network, an articulation point indicates the critical computers and a bridge indicates the critical wires or connections.

Following are some example graphs with bridges highlighted with red color.



#### How to find all bridges in a given graph?

A simple approach is to one by one remove all edges and see if removal of a edge causes disconnected graph. Following are steps of simple approach for connected graph.

- 1) For every edge  $(u, v)$ , do following
  - a) Remove  $(u, v)$  from graph
  - b) See if the graph remains connected (We can either use BFS or DFS)
  - c) Add  $(u, v)$  back to the graph.

Time complexity of above method is  $O(E^2(V+E))$  for a graph represented using adjacency list. Can we do better?

#### A $O(V+E)$ algorithm to find all Bridges

The idea is similar to [O\(V+E\) algorithm for Articulation Points](#). We do DFS traversal of the given graph. In DFS tree an edge  $(u, v)$  ( $u$  is parent of  $v$  in DFS tree) is bridge if there does not exist any other alternative to reach  $u$  or an ancestor of  $u$  from subtree rooted with  $v$ . As discussed in the [previous post](#), the value  $low[v]$  indicates earliest visited vertex reachable from subtree rooted with  $v$ . *The condition for an edge  $(u, v)$  to be a bridge is,  $low[v] > disc[u]$ .*

Following are C++ and Java implementations of above approach.

#### C++

```

// A C++ program to find bridges in a given undirected graph
#include<iostream>
#include <list>
#define NIL -1
using namespace std;

// A class that represents an undirected graph
class Graph {
 int V; // No. of vertices
 list<int> *adj; // A dynamic array of adjacency lists
 void bridgeUtil(int v, bool visited[], int disc[], int low[],
 int parent[]);
public:
 Graph(int V); // Constructor
 void addEdge(int v, int w); // to add an edge to graph
 void bridge(); // prints all bridges
};

Graph::Graph(int V)
{
 this->V = V;
 adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
 adj[v].push_back(w);
}

```

```

adj[w].push_back(v); // Note: the graph is undirected
}

// A recursive function that finds and prints bridges using
// DFS traversal
// u --> The vertex to be visited next
// visited[] --> keeps tract of visited vertices
// disc[] --> Stores discovery times of visited vertices
// parent[] --> Stores parent vertices in DFS tree
void Graph::bridgeUtil(int u, bool visited[], int disc[],
 int low[], int parent[])
{
 // A static variable is used for simplicity, we can
 // avoid use of static variable by passing a pointer.
 static int time = 0;

 // Mark the current node as visited
 visited[u] = true;

 // Initialize discovery time and low value
 disc[u] = low[u] = ++time;

 // Go through all vertices adjacent to this
 list<int>::iterator i;
 for (i = adj[u].begin(); i != adj[u].end(); ++i)
 {
 int v = *i; // v is current adjacent of u

 // If v is not visited yet, then recur for it
 if (!visited[v])
 {
 parent[v] = u;
 bridgeUtil(v, visited, disc, low, parent);

 // Check if the subtree rooted with v has a
 // connection to one of the ancestors of u
 low[u] = min(low[u], low[v]);
 }

 // If the lowest vertex reachable from subtree
 // under v is below u in DFS tree, then u-v
 // is a bridge
 if (low[v] > disc[u])
 cout << u << " -> " << v << endl;
 }

 // Update low value of u for parent function calls.
 else if (v != parent[u])
 low[u] = min(low[u], disc[v]);
}
}

// DFS based function to find all bridges. It uses recursive
// function bridgeUtil()
void Graph::bridge()
{
 // Mark all the vertices as not visited
 bool *visited = new bool[V];
 int *disc = new int[V];
 int *low = new int[V];
 int *parent = new int[V];

 // Initialize parent and visited arrays
 for (int i = 0; i < V; i++)
 {
 parent[i] = NIL;
 visited[i] = false;
 }

 // Call the recursive helper function to find Bridges
 // in DFS tree rooted with vertex 'i'
 for (int i = 0; i < V; i++)
 if (visited[i] == false)
 bridgeUtil(i, visited, disc, low, parent);
}

// Driver program to test above function
int main()
{
 // Create graphs given in above diagrams
 cout << "\nBridges in first graph \n";
 Graph g1(5);
 g1.addEdge(1, 0);
 g1.addEdge(0, 2);
 g1.addEdge(2, 1);
 g1.addEdge(0, 3);
 g1.addEdge(3, 4);
 g1.bridge();

 cout << "\nBridges in second graph \n";
 Graph g2(4);
 g2.addEdge(0, 1);
 g2.addEdge(1, 2);
 g2.addEdge(2, 3);
 g2.bridge();

 cout << "\nBridges in third graph \n";
 Graph g3(7);
 g3.addEdge(0, 1);
 g3.addEdge(1, 2);
 g3.addEdge(2, 0);
 g3.addEdge(1, 3);
 g3.addEdge(1, 4);
 g3.addEdge(1, 6);
 g3.addEdge(3, 5);
 g3.addEdge(4, 5);
 g3.bridge();
}

```

```
 return 0;
}
```

## Java

```
// A Java program to find bridges in a given undirected graph
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents a undirected graph using adjacency list
// representation
class Graph
{
 private int V; // No. of vertices

 // Array of lists for Adjacency List Representation
 private LinkedList<Integer> adj[][];
 int time = 0;
 static final int NIL = -1;

 // Constructor
 Graph(int v)
 {
 V = v;
 adj = new LinkedList[v];
 for (int i=0; i<v; ++i)
 adj[i] = new LinkedList();
 }

 // Function to add an edge into the graph
 void addEdge(int v, int w)
 {
 adj[v].add(w); // Add w to v's list.
 adj[w].add(v); // Add v to w's list
 }

 // A recursive function that finds and prints bridges
 // using DFS traversal
 // u -> The vertex to be visited next
 // visited[] --> keeps tract of visited vertices
 // disc[] --> Stores discovery times of visited vertices
 // parent[] --> Stores parent vertices in DFS tree
 void bridgeUtil(int u, boolean visited[], int disc[],
 int low[], int parent[])
 {

 // Count of children in DFS Tree
 int children = 0;

 // Mark the current node as visited
 visited[u] = true;

 // Initialize discovery time and low value
 disc[u] = low[u] = ++time;

 // Go through all vertices aadjacent to this
 Iterator<Integer> i = adj[u].iterator();
 while (i.hasNext())
 {
 int v = i.next(); // v is current adjacent of u

 // If v is not visited yet, then make it a child
 // of u in DFS tree and recur for it.
 // If v is not visited yet, then recur for it
 if (!visited[v])
 {
 parent[v] = u;
 bridgeUtil(v, visited, disc, low, parent);

 // Check if the subtree rooted with v has a
 // connection to one of the ancestors of u
 low[u] = Math.min(low[u], low[v]);
 }

 // If the lowest vertex reachable from subtree
 // under v is below u in DFS tree, then u-v is
 // a bridge
 if (low[v] > disc[u])
 System.out.println(u+" "+v);
 }

 // Update low value of u for parent function calls.
 else if (v != parent[u])
 low[u] = Math.min(low[u], disc[v]);
 }
}

// DFS based function to find all bridges. It uses recursive
// function bridgeUtil()
void bridge()
{
 // Mark all the vertices as not visited
 boolean visited[] = new boolean[V];
 int disc[] = new int[V];
 int low[] = new int[V];
 int parent[] = new int[V];

 // Initialize parent and visited, and ap(articulation point)
 // arrays
 for (int i = 0; i < V; i++)
 {
 parent[i] = NIL;
```

```

 visited[i] = false;
}

// Call the recursive helper function to find Bridges
// in DFS tree rooted with vertex i
for (int i = 0; i < V; i++)
 if (visited[i] == false)
 bridgeUtil(i, visited, disc, low, parent);
}

public static void main(String args[])
{
 // Create graphs given in above diagrams
 System.out.println("Bridges in first graph ");
 Graph g1 = new Graph(5);
 g1.addEdge(1, 0);
 g1.addEdge(0, 2);
 g1.addEdge(2, 1);
 g1.addEdge(0, 3);
 g1.addEdge(3, 4);
 g1.bridge();
 System.out.println();

 System.out.println("Bridges in Second graph");
 Graph g2 = new Graph(4);
 g2.addEdge(0, 1);
 g2.addEdge(1, 2);
 g2.addEdge(2, 3);
 g2.bridge();
 System.out.println();

 System.out.println("Bridges in Third graph");
 Graph g3 = new Graph(7);
 g3.addEdge(0, 1);
 g3.addEdge(1, 2);
 g3.addEdge(2, 0);
 g3.addEdge(1, 3);
 g3.addEdge(1, 4);
 g3.addEdge(1, 6);
 g3.addEdge(3, 5);
 g3.addEdge(4, 5);
 g3.bridge();
}
}

// This code is contributed by Aakash Hasija

```

## Python

```

Python program to find bridges in a given undirected graph
#Complexity : O(V+E)

from collections import defaultdict

#This class represents an undirected graph using adjacency list representation
class Graph:

 def __init__(self,vertices):
 self.V= vertices #No. of vertices
 self.graph = defaultdict(list) # default dictionary to store graph
 self.Time = 0

 # function to add an edge to graph
 def addEdge(self,u,v):
 self.graph[u].append(v)
 self.graph[v].append(u)

 """A recursive function that finds and prints bridges
 using DFS traversal
 u --> The vertex to be visited next
 visited[] --> keeps tract of visited vertices
 disc[] --> Stores discovery times of visited vertices
 parent[] --> Stores parent vertices in DFS tree"""
 def bridgeUtil(self,u, visited, parent, low, disc):

 #Count of children in current node
 children =0

 #Mark the current node as visited and print it
 visited[u]= True

 # Initialize discovery time and low value
 disc[u] = self.Time
 low[u] = self.Time
 self.Time += 1

 #Recur for all the vertices adjacent to this vertex
 for v in self.graph[u]:
 # If v is not visited yet, then make it a child of u
 # in DFS tree and recur for it
 if visited[v] == False :
 parent[v] = u
 children += 1
 self.bridgeUtil(v, visited, parent, low, disc)

 # Check if the subtree rooted with v has a connection to
 # one of the ancestors of u
 low[u] = min(low[u], low[v])

 """ If the lowest vertex reachable from subtree
 under v is below u in DFS tree, then u-v is
 a bridge"""
 if low[v]> disc[u]:
 print ("%d %d" %(u,v))

 }

}

This code is contributed by Aakash Hasija

```

```
 elif v != parent[u]: # Update low value of u for parent function calls.
 low[u] = min(low[u], disc[v])
```

```
DFS based function to find all bridges. It uses recursive
function bridgeUtil()
def bridge(self):

 # Mark all the vertices as not visited and Initialize parent and visited,
 # and ap(articulation point) arrays
 visited = [False] * (self.V)
 disc = [float("Inf")] * (self.V)
 low = [float("Inf")] * (self.V)
 parent = [-1] * (self.V)

 # Call the recursive helper function to find bridges
 # in DFS tree rooted with vertex i'
 for i in range(self.V):
 if visited[i] == False:
 self.bridgeUtil(i, visited, parent, low, disc)
```

```
Create a graph given in the above diagram
```

```
g1 = Graph(5)
g1.addEdge(1, 0)
g1.addEdge(0, 2)
g1.addEdge(2, 1)
g1.addEdge(0, 3)
g1.addEdge(3, 4)
```

```
print "Bridges in first graph "
g1.bridge()
```

```
g2 = Graph(4)
g2.addEdge(0, 1)
g2.addEdge(1, 2)
g2.addEdge(2, 3)
print "\nBridges in second graph "
g2.bridge()
```

```
g3 = Graph (7)
g3.addEdge(0, 1)
g3.addEdge(1, 2)
g3.addEdge(2, 0)
g3.addEdge(1, 3)
g3.addEdge(1, 4)
g3.addEdge(1, 6)
g3.addEdge(3, 5)
g3.addEdge(4, 5)
print "\nBridges in third graph "
g3.bridge()
```

```
#This code is contributed by Neelam Yadav
```

Output:

```
Bridges in first graph
3 4
0 3
```

```
Bridges in second graph
2 3
1 2
0 1
```

```
Bridges in third graph
1 6
```

**Time Complexity:** The above function is simple DFS with additional arrays. So time complexity is same as DFS which is  $O(V+E)$  for adjacency list representation of graph.

**References:**

<https://www.cs.washington.edu/education/courses/421/04su/slides/artic.pdf>  
<http://www.slideshare.net/TraianRebedea/algoritm-design-and-complexity-course-8>  
[http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250\\_Weiss/L25-Connectivity.htm](http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250_Weiss/L25-Connectivity.htm)  
<http://www.youtube.com/watch?v=bmnyxNyZKzI>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner Company Wise Coding Practice

Graph  
graph-connectivity

## Eulerian path and circuit for undirected graph

Eulerian Path is a path in graph that visits every edge exactly once. Eulerian Circuit is an Eulerian Path which starts and ends on the same vertex.

Euler1



## Euler2



## Euler3



### How to find whether a given graph is Eulerian or not?

The problem is same as following question. "Is it possible to draw a given graph without lifting pencil from the paper and without tracing any of the edges more than once".

A graph is called Eulerian if it has an Eulerian Cycle and called Semi-Eulerian if it has an Eulerian Path. The problem seems similar to [Hamiltonian Path](#) which is NP complete problem for a general graph. Fortunately, we can find whether a given graph has a Eulerian Path or not in polynomial time. In fact, we can find it in  $O(V+E)$  time.

Following are some interesting properties of undirected graphs with an Eulerian path and cycle. We can use these properties to find whether a graph is Eulerian or not.

#### Eulerian Cycle

An undirected graph has Eulerian cycle if following two conditions are true.

....a) All vertices with non-zero degree are connected. We don't care about vertices with zero degree because they don't belong to Eulerian Cycle or Path (we only consider all edges).

....b) All vertices have even degree.

#### Eulerian Path

An undirected graph has Eulerian Path if following two conditions are true.

....a) Same as condition (a) for Eulerian Cycle

....b) If zero or two vertices have odd degree and all other vertices have even degree. Note that only one vertex with odd degree is not possible in an undirected graph (sum of all degrees is always even in an undirected graph)

Note that a graph with no edges is considered Eulerian because there are no edges to traverse.

#### How does this work?

In Eulerian path, each time we visit a vertex  $v$ , we walk through two unvisited edges with one end point as  $v$ . Therefore, all middle vertices in Eulerian Path must have even degree. For Eulerian Cycle, any vertex can be middle vertex, therefore all vertices must have even degree.

## C++

```
// A C++ program to check if a given graph is Eulerian or not
#include<iostream>
#include <list>
using namespace std;

// A class that represents an undirected graph
class Graph
{
 int V; // No. of vertices
 list<int> *adj; // A dynamic array of adjacency lists
public:
 // Constructor and destructor
 Graph(int V) {this->V = V; adj = new list<int>[V];}
 ~Graph() { delete [] adj; } // To avoid memory leak

 // function to add an edge to graph
 void addEdge(int v, int w);

 // Method to check if this graph is Eulerian or not
 int isEulerian();

 // Method to check if all non-zero degree vertices are connected
 bool isConnected();

 // Function to do DFS starting from v. Used in isConnected();
 void DFSUtil(int v, bool visited[]);

};

void Graph::addEdge(int v, int w)
{
 adj[v].push_back(w);
 adj[w].push_back(v); // Note: the graph is undirected
}

void Graph::DFSUtil(int v, bool visited[])
{
 // Mark the current node as visited and print it
 visited[v] = true;

 // Recur for all the vertices adjacent to this vertex
 list<int>::iterator i;
 for (i = adj[v].begin(); i != adj[v].end(); ++i)
 if (!visited[*i])
 DFSUtil(*i, visited);
}

// Method to check if all non-zero degree vertices are connected.
// It mainly does DFS traversal starting from
bool Graph::isConnected()
{
 // Mark all the vertices as not visited
 bool visited[V];
 int i;
 for (i = 0; i < V; i++)
 visited[i] = false;

 // Find a vertex with non-zero degree
```

```

for (i = 0; i < V; i++)
 if (adj[i].size() != 0)
 break;

// If there are no edges in the graph, return true
if (i == V)
 return true;

// Start DFS traversal from a vertex with non-zero degree
DFSUtil(i, visited);

// Check if all non-zero degree vertices are visited
for (i = 0; i < V; i++)
 if (visited[i] == false && adj[i].size() > 0)
 return false;

return true;
}

/* The function returns one of the following values
0 -> If graph is not Eulerian
1 -> If graph has an Euler path (Semi-Eulerian)
2 -> If graph has an Euler Circuit (Eulerian) */
int Graph::isEulerian()
{
 // Check if all non-zero degree vertices are connected
 if (isConnected() == false)
 return 0;

 // Count vertices with odd degree
 int odd = 0;
 for (int i = 0; i < V; i++)
 if (adj[i].size() & 1)
 odd++;

 // If count is more than 2, then graph is not Eulerian
 if (odd > 2)
 return 0;

 // If odd count is 2, then semi-eulerian.
 // If odd count is 0, then eulerian
 // Note that odd count can never be 1 for undirected graph
 return (odd)? 1 : 2;
}

// Function to run test cases
void test(Graph &g)
{
 int res = g.isEulerian();
 if (res == 0)
 cout << "graph is not Eulerian\n";
 else if (res == 1)
 cout << "graph has a Euler path\n";
 else
 cout << "graph has a Euler cycle\n";
}

// Driver program to test above function
int main()
{
 // Let us create and test graphs shown in above figures
 Graph g1(5);
 g1.addEdge(1, 0);
 g1.addEdge(0, 2);
 g1.addEdge(2, 1);
 g1.addEdge(0, 3);
 g1.addEdge(3, 4);
 test(g1);

 Graph g2(5);
 g2.addEdge(1, 0);
 g2.addEdge(0, 2);
 g2.addEdge(2, 1);
 g2.addEdge(0, 3);
 g2.addEdge(3, 4);
 g2.addEdge(4, 0);
 test(g2);

 Graph g3(5);
 g3.addEdge(1, 0);
 g3.addEdge(0, 2);
 g3.addEdge(2, 1);
 g3.addEdge(0, 3);
 g3.addEdge(3, 4);
 g3.addEdge(1, 3);
 test(g3);

 // Let us create a graph with 3 vertices
 // connected in the form of cycle
 Graph g4(3);
 g4.addEdge(0, 1);
 g4.addEdge(1, 2);
 g4.addEdge(2, 0);
 test(g4);

 // Let us create a graph with all vertices
 // with zero degree
 Graph g5(3);
 test(g5);

 return 0;
}

```

```

// A Java program to check if a given graph is Eulerian or not
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents an undirected graph using adjacency list
// representation
class Graph
{
 private int V; // No. of vertices

 // Array of lists for Adjacency List Representation
 private LinkedList<Integer> adj[];

 // Constructor
 Graph(int v)
 {
 V = v;
 adj = new LinkedList[v];
 for (int i=0; i<v; ++i)
 adj[i] = new LinkedList();
 }

 //Function to add an edge into the graph
 void addEdge(int v, int w)
 {
 adj[v].add(w); // Add w to v's list.
 adj[w].add(v); //The graph is undirected
 }

 // A function used by DFS
 void DFSUtil(int v,boolean visited[])
 {
 // Mark the current node as visited
 visited[v] = true;

 // Recur for all the vertices adjacent to this vertex
 Iterator<Integer> i = adj[v].listIterator();
 while (i.hasNext())
 {
 int n = i.next();
 if (!visited[n])
 DFSUtil(n, visited);
 }
 }

 // Method to check if all non-zero degree vertices are
 // connected. It mainly does DFS traversal starting from
 boolean isConnected()
 {
 // Mark all the vertices as not visited
 boolean visited[] = new boolean[V];
 int i;
 for (i = 0; i < V; i++)
 visited[i] = false;

 // Find a vertex with non-zero degree
 for (i = 0; i < V; i++)
 if (adj[i].size() != 0)
 break;

 // If there are no edges in the graph, return true
 if (i == V)
 return true;

 // Start DFS traversal from a vertex with non-zero degree
 DFSUtil(i, visited);

 // Check if all non-zero degree vertices are visited
 for (i = 0; i < V; i++)
 if (visited[i] == false && adj[i].size() > 0)
 return false;

 return true;
 }

 /* The function returns one of the following values
 * 0 -> If graph is not Eulerian
 * 1 -> If graph has an Euler path (Semi-Eulerian)
 * 2 -> If graph has an Euler Circuit (Eulerian) */
 int isEulerian()
 {
 // Check if all non-zero degree vertices are connected
 if (isConnected() == false)
 return 0;

 // Count vertices with odd degree
 int odd = 0;
 for (int i = 0; i < V; i++)
 if (adj[i].size()%2!=0)
 odd++;

 // If count is more than 2, then graph is not Eulerian
 if (odd > 2)
 return 0;

 // If odd count is 2, then semi-eulerian.
 // If odd count is 0, then eulerian
 // Note that odd count can never be 1 for undirected graph
 return (odd==2)? 1 : 2;
 }

 // Function to run test cases
 void test()
 {
 int res = isEulerian();
 if (res == 0)

```

```

 System.out.println("graph is not Eulerian");
 else if (res == 1)
 System.out.println("graph has a Euler path");
 else
 System.out.println("graph has a Euler cycle");
 }

 // Driver method
 public static void main(String args[])
 {
 // Let us create and test graphs shown in above figures
 Graph g1 = new Graph(5);
 g1.addEdge(1, 0);
 g1.addEdge(0, 2);
 g1.addEdge(2, 1);
 g1.addEdge(0, 3);
 g1.addEdge(3, 4);
 g1.test();

 Graph g2 = new Graph(5);
 g2.addEdge(1, 0);
 g2.addEdge(0, 2);
 g2.addEdge(2, 1);
 g2.addEdge(0, 3);
 g2.addEdge(3, 4);
 g2.addEdge(4, 0);
 g2.test();

 Graph g3 = new Graph(5);
 g3.addEdge(1, 0);
 g3.addEdge(0, 2);
 g3.addEdge(2, 1);
 g3.addEdge(0, 3);
 g3.addEdge(3, 4);
 g3.addEdge(1, 3);
 g3.test();

 // Let us create a graph with 3 vertices
 // connected in the form of cycle
 Graph g4 = new Graph(3);
 g4.addEdge(0, 1);
 g4.addEdge(1, 2);
 g4.addEdge(2, 0);
 g4.test();

 // Let us create a graph with all vertices
 // with zero degree
 Graph g5 = new Graph(3);
 g5.test();
 }
}

// This code is contributed by Aakash Hasija

```

## Python

```

Python program to check if a given graph is Eulerian or not
#Complexity : O(V+E)

from collections import defaultdict

#This class represents a undirected graph using adjacency list representation
class Graph:

 def __init__(self,vertices):
 self.V= vertices #No. of vertices
 self.graph = defaultdict(list) # default dictionary to store graph

 # function to add an edge to graph
 def addEdge(self,u,v):
 self.graph[u].append(v)
 self.graph[v].append(u)

 #A function used by isConnected
 def DFSUtil(self,v,visited):
 # Mark the current node as visited
 visited[v]= True

 #Recur for all the vertices adjacent to this vertex
 for i in self.graph[v]:
 if visited[i]==False:
 self.DFSUtil(i,visited)

 """Method to check if all non-zero degree vertices are
 connected. It mainly does DFS traversal starting from
 node with non-zero degree"""
 def isConnected(self):

 # Mark all the vertices as not visited
 visited =[False]*self.V

 # Find a vertex with non-zero degree
 for i in range(self.V):
 if len(self.graph[i]) > 1:
 break

 # If there are no edges in the graph, return true
 if i == self.V-1:
 return True

 # Start DFS traversal from a vertex with non-zero degree
 self.DFSUtil(i,visited)

 # Check if all non-zero degree vertices are visited

```

```

for i in range(self.V):
 if visited[i]==False and len(self.graph[i]) > 0:
 return False

return True

"""The function returns one of the following values
 0 -> If graph is not Eulerian
 1 -> If graph has an Euler path (Semi-Eulerian)
 2 -> If graph has an Euler Circuit (Eulerian) """
def isEulerian(self):
 # Check if all non-zero degree vertices are connected
 if self.isConnected() == False:
 return 0
 else:
 #Count vertices with odd degree
 odd = 0
 for i in range(self.V):
 if len(self.graph[i]) % 2 !=0:
 odd +=1

 """If odd count is 2, then semi-eulerian.
 If odd count is 0, then eulerian
 If count is more than 2, then graph is not Eulerian
 Note that odd count can never be 1 for undirected graph"""
 if odd == 0:
 return 2
 elif odd == 2:
 return 1
 elif odd > 2:
 return 0

Function to run test cases
def test(self):
 res = self.isEulerian()
 if res == 0:
 print "graph is not Eulerian"
 elif res ==1 :
 print "graph has a Euler path"
 else:
 print "graph has a Euler cycle"

#Let us create and test graphs shown in above figures
g1 = Graph(5);
g1.addEdge(1, 0)
g1.addEdge(0, 2)
g1.addEdge(2, 1)
g1.addEdge(0, 3)
g1.addEdge(3, 4)
g1.test()

g2 = Graph(5)
g2.addEdge(1, 0)
g2.addEdge(0, 2)
g2.addEdge(2, 1)
g2.addEdge(0, 3)
g2.addEdge(3, 4)
g2.addEdge(4, 0)
g2.test();

g3 = Graph(5)
g3.addEdge(1, 0)
g3.addEdge(0, 2)
g3.addEdge(2, 1)
g3.addEdge(0, 3)
g3.addEdge(3, 4)
g3.addEdge(1, 3)
g3.test()

#Let us create a graph with 3 vertices
connected in the form of cycle
g4 = Graph(3)
g4.addEdge(0, 1)
g4.addEdge(1, 2)
g4.addEdge(2, 0)
g4.test()

Let us create a graph with all veritces
with zero degree
g5 = Graph(3)
g5.test()

#This code is contributed by Neelam Yadav

```

Output:

```

graph has a Euler path
graph has a Euler cycle
graph is not Eulerian
graph has a Euler cycle
graph has a Euler cycle

```

Time Complexity: O(V+E)

#### Next Articles:

[Eulerian Path and Circuit for a Directed Graphs.](#)

[Fleury's Algorithm to print a Eulerian Path or Circuit?](#)

#### References:

[http://en.wikipedia.org/wiki/Eulerian\\_path](http://en.wikipedia.org/wiki/Eulerian_path)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Fleury's Algorithm for printing Eulerian Path or Circuit

Eulerian Path is a path in graph that visits every edge exactly once. Eulerian Circuit is an Eulerian Path which starts and ends on the same vertex.

We strongly recommend to first read the following post on Euler Path and Circuit.

<http://www.geeksforgeeks.org/eulerian-path-and-circuit/>

In the above mentioned post, we discussed the problem of finding out whether a given graph is Eulerian or not. In this post, an algorithm to print Eulerian trail or circuit is discussed.

Following is Fleury's Algorithm for printing Eulerian trail or cycle (Source [Ref1](#)).

1. Make sure the graph has either 0 or 2 odd vertices.
2. If there are 0 odd vertices, start anywhere. If there are 2 odd vertices, start at one of them.
3. Follow edges one at a time. If you have a choice between a bridge and a non-bridge, *always choose the non-bridge*.
4. Stop when you run out of edges.

The idea is, “**don't burn bridges**” so that we can come back to a vertex and traverse remaining edges. For example let us consider the following graph.



There are two vertices with odd degree, '2' and '3', we can start path from any of them. Let us start tour from vertex '2'.



There are three edges going out from vertex '2', which one to pick? We don't pick the edge '2-3' because that is a bridge (we won't be able to come back to '3'). We can pick any of the remaining two edge. Let us say we pick '2-0'. We remove this edge and move to vertex '0'.



There is only one edge from vertex '0', so we pick it, remove it and move to vertex '1'. Euler tour becomes '2-0 0-1'.



There is only one edge from vertex '1', so we pick it, remove it and move to vertex '2'. Euler tour becomes '2-0 0-1 1-2'.



Again there is only one edge from vertex 2, so we pick it, remove it and move to vertex 3. Euler tour becomes '2-0 0-1 1-2 2-3'.



There are no more edges left, so we stop here. Final tour is '2-0 0-1 1-2 2-3'.

See [this](#) for and [this](#) fore more examples.

Following is C++ implementation of above algorithm. In the following code, it is assumed that the given graph has an Eulerian trail or Circuit. The main focus is to print an Eulerian trail or circuit. We can use `isEulerian()` to first check whether there is an Eulerian Trail or Circuit in the given graph.

We first find the starting point which must be an odd vertex (if there are odd vertices) and store it in variable 'u'. If there are zero odd vertices, we start from vertex '0'. We call `printEulerUtil()` to print Euler tour starting with u. We traverse all adjacent vertices of u, if there is only one adjacent vertex, we immediately consider it. If there are more than one adjacent vertices, we consider an adjacent v only if edge u-v is not a bridge. How to find if a given is edge is bridge? We count number of vertices reachable from u. We remove edge u-v and again count number of reachable vertices from u. If number of reachable vertices are reduced, then edge u-v is a bridge. To count reachable vertices, we can either use BFS or DFS, we have used DFS in the above code. The function `DFSCount(u)` returns number of vertices reachable from u.

Once an edge is processed (included in Euler tour), we remove it from the graph. To remove the edge, we replace the vertex entry with -1 in adjacency list. Note that simply deleting the node may not work as the code is recursive and a parent call may be in middle of adjacency list.

### C/C++

```
// A C++ program print Eulerian Trail in a given Eulerian or Semi-Eulerian Graph
#include <iostream>
#include <string.h>
#include <algorithm>
#include <list>
```

```

using namespace std;

// A class that represents an undirected graph
class Graph
{
 int V; // No. of vertices
 list<int> *adj; // A dynamic array of adjacency lists
public:
 // Constructor and destructor
 Graph(int V) { this->V = V; adj = new list<int>[V]; }
 ~Graph() { delete [] adj; }

 // functions to add and remove edge
 void addEdge(int u, int v) { adj[u].push_back(v); adj[v].push_back(u); }
 void rmvEdge(int u, int v);

 // Methods to print Eulerian tour
 void printEulerTour();
 void printEulerUtil(int s);

 // This function returns count of vertices reachable from v. It does DFS
 int DFSCount(int v, bool visited[]);

 // Utility function to check if edge u-v is a valid next edge in
 // Eulerian trail or circuit
 bool isValidNextEdge(int u, int v);
};

/* The main function that print Eulerian Trail. It first finds an odd
degree vertex (if there is any) and then calls printEulerUtil()
to print the path */
void Graph::printEulerTour()
{
 // Find a vertex with odd degree
 int u = 0;
 for (int i = 0; i < V; i++)
 if (adj[i].size() & 1)
 { u = i; break; }

 // Print tour starting from oddv
 printEulerUtil(u);
 cout << endl;
}

// Print Euler tour starting from vertex u
void Graph::printEulerUtil(int u)
{
 // Recur for all the vertices adjacent to this vertex
 list<int>::iterator i;
 for (i = adj[u].begin(); i != adj[u].end(); ++i)
 {
 int v = *i;

 // If edge u-v is not removed and it's a valid next edge
 if (v != -1 && isValidNextEdge(u, v))
 {
 cout << u << " " << v << " ";
 rmvEdge(u, v);
 printEulerUtil(v);
 }
 }
}

// The function to check if edge u-v can be considered as next edge in
// Euler Tour
bool Graph::isValidNextEdge(int u, int v)
{
 // The edge u-v is valid in one of the following two cases:

 // 1) If v is the only adjacent vertex of u
 int count = 0; // To store count of adjacent vertices
 list<int>::iterator i;
 for (i = adj[u].begin(); i != adj[u].end(); ++i)
 if (*i != -1)
 count++;
 if (count == 1)
 return true;

 // 2) If there are multiple adjacents, then u-v is not a bridge
 // Do following steps to check if u-v is a bridge

 // 2.a) count of vertices reachable from u
 bool visited[V];
 memset(visited, false, V);
 int count1 = DFSCount(u, visited);

 // 2.b) Remove edge (u, v) and after removing the edge, count
 // vertices reachable from u
 rmvEdge(u, v);
 memset(visited, false, V);
 int count2 = DFSCount(u, visited);

 // 2.c) Add the edge back to the graph
 addEdge(u, v);

 // 2.d) If count1 is greater, then edge (u, v) is a bridge
 return (count1 > count2)? false: true;
}

// This function removes edge u-v from graph. It removes the edge by
// replacing adjacent vertex value with -1.
void Graph::rmvEdge(int u, int v)
{
 // Find v in adjacency list of u and replace it with -1
 list<int>::iterator iv = find(adj[u].begin(), adj[u].end(), v);
 *iv = -1;
}

```

```

// Find u in adjacency list of v and replace it with -1
list<int>::iterator iu = find(adj[v].begin(), adj[v].end(), u);
*iu = -1;
}

// A DFS based function to count reachable vertices from v
int Graph::DFSCount(int v, bool visited[])
{
 // Mark the current node as visited
 visited[v] = true;
 int count = 1;

 // Recur for all vertices adjacent to this vertex
 list<int>::iterator i;
 for (i = adj[v].begin(); i != adj[v].end(); ++i)
 if (*i != -1 && !visited[*i])
 count += DFSCount(*i, visited);

 return count;
}

// Driver program to test above function
int main()
{
 // Let us first create and test graphs shown in above figure
 Graph g1(4);
 g1.addEdge(0, 1);
 g1.addEdge(0, 2);
 g1.addEdge(1, 2);
 g1.addEdge(2, 3);
 g1.printEulerTour();

 Graph g2(3);
 g2.addEdge(0, 1);
 g2.addEdge(1, 2);
 g2.addEdge(2, 0);
 g2.printEulerTour();

 Graph g3(5);
 g3.addEdge(1, 0);
 g3.addEdge(0, 2);
 g3.addEdge(2, 1);
 g3.addEdge(0, 3);
 g3.addEdge(3, 4);
 g3.addEdge(3, 2);
 g3.addEdge(3, 1);
 g3.addEdge(2, 4);
 g3.printEulerTour();

 return 0;
}

```

## Python

```

Python program print Eulerian Trail in a given Eulerian or Semi-Eulerian Graph

from collections import defaultdict

#This class represents an undirected graph using adjacency list representation
class Graph:

 def __init__(self,vertices):
 self.V= vertices #No. of vertices
 self.graph = defaultdict(list) # default dictionary to store graph
 self.Time = 0

 # function to add an edge to graph
 def addEdge(self,u,v):
 self.graph[u].append(v)
 self.graph[v].append(u)

 # This function removes edge u-v from graph
 def rmvEdge(self, u, v):
 for index, key in enumerate(self.graph[u]):
 if key == v:
 self.graph[u].pop(index)
 for index, key in enumerate(self.graph[v]):
 if key == u:
 self.graph[v].pop(index)

 # A DFS based function to count reachable vertices from v
 def DFSCount(self, v, visited):
 count = 1
 visited[v] = True
 for i in self.graph[v]:
 if visited[i] == False:
 count = count + self.DFSCount(i, visited)
 return count

 # The function to check if edge u-v can be considered as next edge in
 # Euler Tour
 def isValidNextEdge(self, u, v):
 # The edge u-v is valid in one of the following two cases:

 # 1) If v is the only adjacent vertex of u
 if len(self.graph[u]) == 1:
 return True
 else:
 ""

 # 2) If there are multiple adjacents, then u-v is not a bridge
 # Do following steps to check if u-v is a bridge

 2.a) count of vertices reachable from u"

```

```

visited =[False]*(self.V)
count1 = self.DFSCount(u, visited)

'''2.b) Remove edge (u, v) and after removing the edge, count
vertices reachable from u'''
self.rmvEdge(u, v)
visited =[False]*(self.V)
count2 = self.DFSCount(u, visited)

#2.c) Add the edge back to the graph
self.addEdge(u,v)

2.d) If count1 is greater, then edge (u, v) is a bridge
return False if count1 > count2 else True

Print Euler tour starting from vertex u
def printEulerUtil(self, u):
 #Recur for all the vertices adjacent to this vertex
 for v in self.graph[u]:
 #If edge u-v is not removed and it's a valid next edge
 if self.isValidNextEdge(u, v):
 print("%d-%d " %(u,v)),
 self.rmvEdge(u, v)
 self.printEulerUtil(v)

'''The main function that print Eulerian Trail. It first finds an odd
degree vertex (if there is any) and then calls printEulerUtil()
to print the path '''
def printEulerTour(self):
 #Find a vertex with odd degree
 u = 0
 for i in range(self.V):
 if len(self.graph[i]) %2 != 0 :
 u = i
 break
 # Print tour starting from odd vertex
 print ("n")
 self.printEulerUtil(u)

Create a graph given in the above diagram

g1 = Graph(4)
g1.addEdge(0, 1)
g1.addEdge(0, 2)
g1.addEdge(1, 2)
g1.addEdge(2, 3)
g1.printEulerTour()

g2 = Graph(3)
g2.addEdge(0, 1)
g2.addEdge(1, 2)
g2.addEdge(2, 0)
g2.printEulerTour()

g3 = Graph (5)
g3.addEdge(1, 0)
g3.addEdge(0, 2)
g3.addEdge(2, 1)
g3.addEdge(0, 3)
g3.addEdge(3, 4)
g3.addEdge(3, 2)
g3.addEdge(3, 1)
g3.addEdge(2, 4)
g3.printEulerTour()

#This code is contributed by Neelam Yadav

```

Output:

```

2-0 0-1 1-2 2-3
0-1 1-2 2-0
0-1 1-2 2-0 0-3 3-4 4-2 2-3 3-1

```

Note that the above code modifies given graph, we can create a copy of graph if we don't want the given graph to be modified.

**Time Complexity:** Time complexity of the above implementation is  $O((V+E)^2)$ . The function `printEulerUtil()` is like DFS and it calls `isValidNextEdge()` which also does DFS two times. Time complexity of DFS for adjacency list representation is  $O(V+E)$ . Therefore overall time complexity is  $O((V+E)*(V+E))$  which can be written as  $O(E^2)$  for a connected graph.

There are better algorithms to print Euler tour, [Hierholzer's Algorithm](#) finds in  $O(V+E)$  time.

#### References:

<http://www.math.ku.edu/~jmartin/courses/math105-F11/Lectures/chapter5-part2.pdf>  
[http://en.wikipedia.org/wiki/Eulerian\\_path#Fleury.27s\\_algorithm](http://en.wikipedia.org/wiki/Eulerian_path#Fleury.27s_algorithm)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner Company Wise Coding Practice

Graph  
Euler-Circuit  
Graph

## Strongly Connected Components

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (**SCC**) of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph.

## SCC



We can find all strongly connected components in  $O(V+E)$  time using [Kosaraju's algorithm](#). Following is detailed Kosaraju's algorithm.

- 1) Create an empty stack 'S' and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack. In the above graph, if we start DFS from vertex 0, we get vertices in stack as 1, 2, 4, 3, 0.
- 2) Reverse directions of all arcs to obtain the transpose graph.
- 3) One by one pop a vertex from S while S is not empty. Let the popped vertex be 'v'. Take v as source and do DFS (call [DFSUtil\(v\)](#)). The DFS starting from v prints strongly connected component of v. In the above example, we process vertices in order 0, 3, 4, 2, 1 (One by one popped from stack).

### How does this work?

The above algorithm is DFS based. It does DFS two times. DFS of a graph produces a single tree if all vertices are reachable from the DFS starting point. Otherwise DFS produces a forest. So DFS of a graph with only one SCC always produces a tree. The important point to note is DFS may produce a tree or a forest when there are more than one SCCs depending upon the chosen starting point. For example, in the above diagram, if we start DFS from vertices 0 or 1 or 2, we get a tree as output. And if we start from 3 or 4, we get a forest. To find and print all SCCs, we would want to start DFS from vertex 4 (which is a sink vertex), then move to 3 which is sink in the remaining set (set excluding 4) and finally any of the remaining vertices (0, 1, 2). So how do we find this sequence of picking vertices as starting points of DFS? Unfortunately, there is no direct way for getting this sequence. However, if we do a DFS of graph and store vertices according to their finish times, we make sure that the finish time of a vertex that connects to other SCCs (other than its own SCC), will always be greater than finish time of vertices in the other SCC (See [this](#) for proof). For example, in DFS of above example graph, finish time of 0 is always greater than 3 and 4 (irrespective of the sequence of vertices considered for DFS). And finish time of 3 is always greater than 4. DFS doesn't guarantee about other vertices, for example finish times of 1 and 2 may be smaller or greater than 3 and 4 depending upon the sequence of vertices considered for DFS. So to use this property, we do DFS traversal of complete graph and push every finished vertex to a stack. In stack, 3 always appears after 4, and 0 appear after both 3 and 4.

In the next step, we reverse the graph. Consider the graph of SCCs. In the reversed graph, the edges that connect two components are reversed. So the SCC {0, 1, 2} becomes sink and the SCC {4} becomes source. As discussed above, in stack, we always have 0 before 3 and 4. So if we do a DFS of the reversed graph using sequence of vertices in stack, we process vertices from sink to source (in reversed graph). That is what we wanted to achieve and that is all needed to print SCCs one by one.

### SCCGraph2



Following is C++ implementation of Kosaraju's algorithm.

### C++

```
// C++ Implementation of Kosaraju's algorithm to print all SCCs
#include <iostream>
#include <list>
#include <stack>
using namespace std;

class Graph
{
 int V; // No. of vertices
 list<int> *adj; // An array of adjacency lists

 // Fills Stack with vertices (in increasing order of finishing
 // times). The top element of stack has the maximum finishing
 // time
 void fillOrder(int v, bool visited[], stack<int> &Stack);

 // A recursive function to print DFS starting from v
 void DFSUtil(int v, bool visited[]);

public:
 Graph(int V);
 void addEdge(int v, int w);

 // The main function that finds and prints strongly connected
 // components
 void printSCCs();

 // Function that returns reverse (or transpose) of this graph
 Graph getTranspose();
};

Graph::Graph(int V)
{
 this->V = V;
 adj = new list<int>[V];
}

// A recursive function to print DFS starting from v
void Graph::DFSUtil(int v, bool visited[])
{
 // Mark the current node as visited and print it
 visited[v] = true;
 cout << v << " ";

 // Recur for all the vertices adjacent to this vertex
 list<int>::iterator i;
 for (i = adj[v].begin(); i != adj[v].end(); ++i)
 if (!visited[*i])
 DFSUtil(*i, visited);
}

Graph::Graph()
{
 V = 0;
}
```

```

Graph Graph::getTranspose()
{
 Graph g(V);
 for (int v = 0; v < V; v++)
 {
 // Recur for all the vertices adjacent to this vertex
 list<int>::iterator i;
 for(i = adj[v].begin(); i != adj[v].end(); ++i)
 {
 g.adj[*i].push_back(v);
 }
 }
 return g;
}

void Graph::addEdge(int v, int w)
{
 adj[v].push_back(w); // Add w to v's list.
}

void Graph::fillOrder(int v, bool visited[], stack<int> &Stack)
{
 // Mark the current node as visited and print it
 visited[v] = true;

 // Recur for all the vertices adjacent to this vertex
 list<int>::iterator i;
 for(i = adj[v].begin(); i != adj[v].end(); ++i)
 if(visited[*i])
 fillOrder(*i, visited, Stack);

 // All vertices reachable from v are processed by now, push v
 Stack.push(v);
}

// The main function that finds and prints all strongly connected
// components
void Graph::printSCCs()
{
 stack<int> Stack;

 // Mark all the vertices as not visited (For first DFS)
 bool *visited = new bool[V];
 for(int i = 0; i < V; i++)
 visited[i] = false;

 // Fill vertices in stack according to their finishing times
 for(int i = 0; i < V; i++)
 if(visited[i] == false)
 fillOrder(i, visited, Stack);

 // Create a reversed graph
 Graph gr = getTranspose();

 // Mark all the vertices as not visited (For second DFS)
 for(int i = 0; i < V; i++)
 visited[i] = false;

 // Now process all vertices in order defined by Stack
 while (Stack.empty() == false)
 {
 // Pop a vertex from stack
 int v = Stack.top();
 Stack.pop();

 // Print Strongly connected component of the popped vertex
 if (visited[v] == false)
 {
 gr.DFSUtil(v, visited);
 cout << endl;
 }
 }
}

// Driver program to test above functions
int main()
{
 // Create a graph given in the above diagram
 Graph g(5);
 g.addEdge(1, 0);
 g.addEdge(0, 2);
 g.addEdge(2, 1);
 g.addEdge(0, 3);
 g.addEdge(3, 4);

 cout << "Following are strongly connected components in "
 "given graph \n";
 g.printSCCs();

 return 0;
}

```

## Java

```

// Java implementation of Kosaraju's algorithm to print all SCCs
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents a directed graph using adjacency list
// representation
class Graph
{

```

```

private int V; // No. of vertices
private LinkedList<Integer> adj[]; //Adjacency List

//Constructor
Graph(int v)
{
 V = v;
 adj = new LinkedList[v];
 for (int i=0;i<v; ++i)
 adj[i] = new LinkedList();
}

//Function to add an edge into the graph
void addEdge(int v, int w) { adj[v].add(w);}

// A recursive function to print DFS starting from v
void DFSUtil(int v,boolean visited[])
{
 // Mark the current node as visited and print it
 visited[v] = true;
 System.out.print(v + " ");

 int n;

 // Recur for all the vertices adjacent to this vertex
 Iterator<Integer> i =adj[v].iterator();
 while (i.hasNext())
 {
 n = i.next();
 if (!visited[n])
 DFSUtil(n,visited);
 }
}

// Function that returns reverse (or transpose) of this graph
Graph getTranspose()
{
 Graph g = new Graph(V);
 for (int v = 0; v < V; v++)
 {
 // Recur for all the vertices adjacent to this vertex
 Iterator<Integer> i =adj[v].listIterator();
 while(i.hasNext())
 g.adj[i.next()].add(v);
 }
 return g;
}

void fillOrder(int v, boolean visited[], Stack stack)
{
 // Mark the current node as visited and print it
 visited[v] = true;

 // Recur for all the vertices adjacent to this vertex
 Iterator<Integer> i = adj[v].iterator();
 while (i.hasNext())
 {
 int n = i.next();
 if(!visited[n])
 fillOrder(n, visited, stack);
 }

 // All vertices reachable from v are processed by now,
 // push v to Stack
 stack.push(new Integer(v));
}

// The main function that finds and prints all strongly
// connected components
void printSCCs()
{
 Stack stack = new Stack();

 // Mark all the vertices as not visited (For first DFS)
 boolean visited[] = new boolean[V];
 for(int i = 0;i < V; i++)
 visited[i] = false;

 // Fill vertices in stack according to their finishing
 // times
 for (int i = 0; i < V; i++)
 if (visited[i] == false)
 fillOrder(i, visited, stack);

 // Create a reversed graph
 Graph gr = getTranspose();

 // Mark all the vertices as not visited (For second DFS)
 for (int i = 0; i < V; i++)
 visited[i] = false;

 // Now process all vertices in order defined by Stack
 while (stack.empty() == false)
 {
 // Pop a vertex from stack
 int v = (int)stack.pop();

 // Print Strongly connected component of the popped vertex
 if (visited[v] == false)
 {
 gr.DFSUtil(v, visited);
 System.out.println();
 }
 }
}

// Driver method

```

```

public static void main(String args[])
{
 // Create a graph given in the above diagram
 Graph g = new Graph(5);
 g.addEdge(1, 0);
 g.addEdge(0, 2);
 g.addEdge(2, 1);
 g.addEdge(0, 3);
 g.addEdge(3, 4);

 System.out.println("Following are strongly connected components " +
 "in given graph ");
 g.printSCCs();
}
}

// This code is contributed by Aakash Hasija

```

## Python

```

Python implementation of Kosaraju's algorithm to print all SCCs

from collections import defaultdict

#This class represents a directed graph using adjacency list representation
class Graph:

 def __init__(self,vertices):
 self.V= vertices #No. of vertices
 self.graph = defaultdict(list) # default dictionary to store graph

 # function to add an edge to graph
 def addEdge(self,u,v):
 self.graph[u].append(v)

 # A function used by DFS
 def DFSUtil(self,v,visited):
 # Mark the current node as visited and print it
 visited[v]= True
 print v,
 #Recur for all the vertices adjacent to this vertex
 for i in self.graph[v]:
 if visited[i]==False:
 self.DFSUtil(i,visited)

 def fillOrder(self,v,visited, stack):
 # Mark the current node as visited
 visited[v]= True
 #Recur for all the vertices adjacent to this vertex
 for i in self.graph[v]:
 if visited[i]==False:
 self.fillOrder(i,visited, stack)
 stack = stack.append(v)

 # Function that returns reverse (or transpose) of this graph
 def getTranspose(self):
 g = Graph(self.V)

 # Recur for all the vertices adjacent to this vertex
 for i in self.graph:
 for j in self.graph[i]:
 g.addEdge(j,i)
 return g

 # The main function that finds and prints all strongly
 # connected components
 def printSCCs(self):

 stack = []
 # Mark all the vertices as not visited (For first DFS)
 visited =[False]*(self.V)
 # Fill vertices in stack according to their finishing
 # times
 for i in range(self.V):
 if visited[i]==False:
 self.fillOrder(i, visited, stack)

 # Create a reversed graph
 gr = self.getTranspose()

 # Mark all the vertices as not visited (For second DFS)
 visited =[False]*(self.V)

 # Now process all vertices in order defined by Stack
 while stack:
 i = stack.pop()
 if visited[i]==False:
 gr.DFSUtil(i, visited)
 print""

 # Create a graph given in the above diagram
 g = Graph(5)
 g.addEdge(1, 0)
 g.addEdge(0, 2)
 g.addEdge(2, 1)
 g.addEdge(0, 3)
 g.addEdge(3, 4)

 print ("Following are strongly connected components " +
 "in given graph")

```

```
g.printSCCs()
#This code is contributed by Neelam Yadav
```

Output:

```
Following are strongly connected components in given graph
0 1 2
3
4
```

**Time Complexity:** The above algorithm calls DFS, finds reverse of the graph and again calls DFS. DFS takes  $O(V+E)$  for a graph represented using adjacency list. Reversing a graph also takes  $O(V+E)$  time. For reversing the graph, we simply traverse all adjacency lists.

The above algorithm is asymptotically best algorithm, but there are other algorithms like [Tarjan's algorithm](#) and [path-based](#) which have same time complexity but find SCCs using single DFS. The Tarjan's algorithm is discussed in the following post.

[Tarjan's Algorithm to find Strongly Connected Components](#)

**Applications:**

SCC algorithms can be used as a first step in many graph algorithms that work only on strongly connected graph.

In social networks, a group of people are generally strongly connected (For example, students of a class or any other common place). Many people in these groups generally like some common pages or play common games. The SCC algorithms can be used to find such groups and suggest the commonly liked pages or games to the people in the group who have not yet liked commonly liked a page or played a game.

**References:**

[http://en.wikipedia.org/wiki/Kosaraju%27s\\_algorithm](http://en.wikipedia.org/wiki/Kosaraju%27s_algorithm)  
<https://www.youtube.com/watch?v=PZQ0Pdk15RA>

You may also like to see [Tarjan's Algorithm to find Strongly Connected Components](#).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner Company Wise Coding Practice

Graph  
DFS  
graph-connectivity

### Transitive closure of a graph

Given a directed graph, find out if a vertex  $j$  is reachable from another vertex  $i$  for all vertex pairs  $(i, j)$  in the given graph. Here reachable mean that there is a path from vertex  $i$  to  $j$ . The reachability matrix is called transitive closure of a graph.

```
For example, consider below graph
transitiveclosure
```



Transitive closure of above graphs is

```
1 1 1 1
1 1 1 1
1 1 1 1
0 0 0 1
```

The graph is given in the form of adjacency matrix say 'graph[V][V]' where  $graph[i][j]$  is 1 if there is an edge from vertex  $i$  to vertex  $j$  or  $i$  is equal to  $j$ , otherwise  $graph[i][j]$  is 0.

[Floyd Warshall Algorithm](#) can be used, we can calculate the distance matrix  $dist[V][V]$  using [Floyd Warshall](#), if  $dist[i][j]$  is infinite, then  $j$  is not reachable from  $i$ , otherwise  $j$  is reachable and value of  $dist[i][j]$  will be less than  $V$ .

Instead of directly using Floyd Warshall, we can optimize it in terms of space and time, for this particular problem. Following are the optimizations:

- 1) Instead of integer resultant matrix ([dist\[V\]\[V\] in floyd warshall](#)), we can create a boolean reachability matrix  $reach[V][V]$  (we save space). The value  $reach[i][j]$  will be 1 if  $j$  is reachable from  $i$ , otherwise 0.
- 2) Instead of using arithmetic operations, we can use logical operations. For arithmetic operation '+', logical and '&&' is used, and for min, logical or '|' is used. (We save time by a constant factor. Time complexity is same though)

### C++

```
// Program for transitive closure using Floyd Warshall Algorithm
#include<stdio.h>

// Number of vertices in the graph
#define V 4

// A function to print the solution matrix
void printSolution(int reach[V][V]);

// Prints transitive closure of graph[V][V] using Floyd Warshall algorithm
void transitiveClosure(int graph[V][V])
{
 /* reach[V][V] will be the output matrix that will finally have the
 shortest distances between every pair of vertices */
 int reach[V][V], i, j, k;

 /* Initialize the solution matrix same as input graph matrix. Or
 we can say the initial values of shortest distances are based
 on shortest paths considering no intermediate vertex. */
 for (i = 0; i < V; i++)
 for (j = 0; j < V; j++)
 reach[i][j] = graph[i][j];

 /* Add all vertices one by one to the set of intermediate vertices.
 ---> Before start of a iteration, we have reachability values for
 all pairs of vertices such that the reachability values
 consider only the vertices in set {0, 1, 2, .. k-1} as
```

```

intermediate vertices.
----> After the end of a iteration, vertex no. k is added to the
 set of intermediate vertices and the set becomes {0, 1, .. k} */
for (k = 0; k < V; k++)
{
 // Pick all vertices as source one by one
 for (i = 0; i < V; i++)
 {
 // Pick all vertices as destination for the
 // above picked source
 for (j = 0; j < V; j++)
 {
 // If vertex k is on a path from i to j,
 // then make sure that the value of reach[i][j] is 1
 // then make sure that the value of reach[i][j] is 1
 reach[i][j] = reach[i][j] || (reach[i][k] && reach[k][j]);
 }
 }
}

// Print the shortest distance matrix
printSolution(reach);
}

/* A utility function to print solution */
void printSolution(int reach[V][V])
{
 printf ("Following matrix is transitive closure of the given graph\n");
 for (int i = 0; i < V; i++)
 {
 for (int j = 0; j < V; j++)
 printf ("%d ", reach[i][j]);
 printf("\n");
 }
}

// driver program to test above function
int main()
{
 /* Let us create the following weighted graph
 10
 (0)----->(3)
 | / \
 5 | |
 | | 1
 \| |
 (1)----->(2)
 3 */
 int graph[V][V] = {{1, 1, 0, 1},
 {0, 1, 1, 0},
 {0, 0, 1, 1},
 {0, 0, 0, 1}};
}

// Print the solution
transitiveClosure(graph);
return 0;
}

```

## Java

```

// Program for transitive closure using Floyd Warshall Algorithm
import java.util.*;
import java.lang.*;
import java.io.*;

class GraphClosure
{
 final static int V = 4; //Number of vertices in a graph

 // Prints transitive closure of graph[][] using Floyd
 // Warshall algorithm
 void transitiveClosure(int graph[][])
 {
 /* reach[][] will be the output matrix that will finally
 have the shortest distances between every pair of
 vertices */
 int reach[][] = new int[V][V];
 int i,j,k;

 /* Initialize the solution matrix same as input graph
 matrix. Or we can say the initial values of shortest
 distances are based on shortest paths considering
 no intermediate vertex. */
 for (i = 0; i < V; i++)
 for (j = 0; j < V; j++)
 reach[i][j] = graph[i][j];

 /* Add all vertices one by one to the set of intermediate
 vertices.
 ----> Before start of a iteration, we have reachability
 values for all pairs of vertices such that the
 reachability values consider only the vertices in
 set {0, 1, 2, .. k-1} as intermediate vertices.
 ----> After the end of a iteration, vertex no. k is
 added to the set of intermediate vertices and the
 set becomes {0, 1, 2, .. k} */
 for (k = 0; k < V; k++)
 {
 // Pick all vertices as source one by one
 for (i = 0; i < V; i++)
 {
 // Pick all vertices as destination for the
 // above picked source
 for (j = 0; j < V; j++)

```

```

 {
 // If vertex k is on a path from i to j,
 // then make sure that the value of reach[i][j] is 1
 reach[i][j] = (reach[i][j]==0) ||
 ((reach[i][k]==0) && (reach[k][j]==0))?1:0;
 }
}

// Print the shortest distance matrix
printSolution(reach);
}

/* A utility function to print solution */
void printSolution(int reach[][])
{
 System.out.println("Following matrix is transitive closure"+
 " of the given graph");
 for (int i = 0; i < V; i++)
 {
 for (int j = 0; j < V; j++)
 System.out.print(reach[i][j]+" ");
 System.out.println();
 }
}

// Driver program to test above function
public static void main (String[] args)
{
 /* Let us create the following weighted graph
 10
 (0)----->(3)
 | /\
 | |
 5 | |
 | | 1
 \| |
 (1)----->(2)
 3 */
}

/* Let us create the following weighted graph

 10
 (0)----->(3)
 | /\
 | |
 5 | |
 | | 1
 \| |
 (1)----->(2)
 3 */

int graph[][] = new int[][]{{1, 1, 0, 1},
 {0, 1, 1, 0},
 {0, 0, 1, 1},
 {0, 0, 0, 1}};
}

// Print the solution
GraphClosure g = new GraphClosure();
g.transitiveClosure(graph);
}
}

// This code is contributed by Aakash Hasija

```

## Python

```

Python program for transitive closure using Floyd Warshall Algorithm
#Complexity : O(V^3)

from collections import defaultdict

#Class to represent a graph
class Graph:

 def __init__(self, vertices):
 self.V = vertices

 # A utility function to print the solution
 def printSolution(self, reach):
 print ("Following matrix transitive closure of the given graph ")
 for i in range(self.V):
 for j in range(self.V):
 print "%d" %(reach[i][j]),
 print ""

 # Prints transitive closure of graph[V] using Floyd Warshall algorithm
 def transitiveClosure(self,graph):
 "reach[] will be the output matrix that will finally
 have reachability values.
 Initialize the solution matrix same as input graph matrix"
 reach =[[] for i in graph]
 "Add all vertices one by one to the set of intermediate
 vertices.
 --> Before start of a iteration, we have reachability value
 for all pairs of vertices such that the reachability values
 consider only the vertices in set
 {0, 1, 2, .. k-1} as intermediate vertices.
 ----> After the end of an iteration, vertex no. k is
 added to the set of intermediate vertices and the
 set becomes {0, 1, 2, .. k}"
 for k in range(self.V):

 # Pick all vertices as source one by one
 for i in range(self.V):

```

```

Pick all vertices as destination for the
above picked source
for j in range(self.V):

 # If vertex k is on a path from i to j,
 # then make sure that the value of reach[i][j] is 1
 reach[i][j] = reach[i][k] or (reach[i][k] and reach[k][j])

self.printSolution(reach)

g= Graph(4)

graph = [[1, 1, 0, 1],
 [0, 1, 1, 0],
 [0, 0, 1, 1],
 [0, 0, 0, 1]]

#print the solution
g.transitiveClosure(graph)

#This code is contributed by Neelam Yadav

```

Output:

```

Following matrix is transitive closure of the given graph
1 1 1 1
0 1 1 1
0 0 1 1
0 0 0 1

```

Time Complexity:  $O(V^3)$  where  $V$  is number of vertices in the given graph.

See below post for a  $O(V^2)$  solution.

[Transitive Closure of a Graph using DFS](#)

**References:**

[Introduction to Algorithms](#) by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner Company Wise Coding Practice

Graph  
Graph