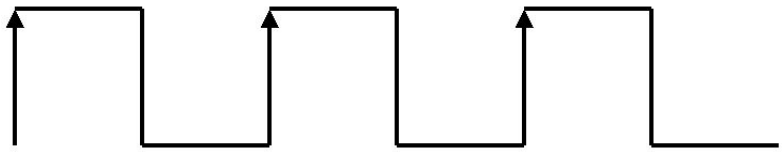
Table of Content

Basic Programming Concepts

Computing Power and Time Limit for Programs:

Clock is a signal used to sync things inside the computer. Take a look at Figure 2, where we show a typical clock signal: it is a square wave changing from “0” to “1” at a fixed rate. On this figure you can see three full clock cycles (“ticks”). The beginning of each cycle is when the clock signal goes from “0” to “1”; we marked this with an arrow. The clock signal is measured in a unit called Hertz (Hz), which is the number of clock cycles per second. A clock of 100 MHz means that in one second there is 100 million clock cycles.



Another way to define it is to say that Clock cycle is the amount of time between two pulses of an oscillator. Generally speaking, the higher number of pulses per second, the faster the computer processor will be able to process information.

CPU’s clock speed, or clock rate, is measured in Hertz — generally in gigahertz, or GHz. A CPU’s clock speed rate is a measure of how many clock cycles a CPU can perform per second. For example, a CPU with a clock rate of 1.8 GHz can perform 1,800,000,000 clock cycles per second.

This seems simple on its face. The more clock cycles a CPU can perform the more things it can get done, right? Well, yes and no.

This may not seem obvious at first, but it’s actually for a very simple reason. It’s because the speed of a computer is also influenced by other factors, such as the efficiency of the processor, the bus architecture, the amount of memory available, and the software that is running on the computer. Some processors can perform more instructions per clock cycle than other processors, making them more efficient than other processors with higher clock speeds. All other things being equal, fewer clock cycles mean the CPU requires less power and produces less heat.

In addition, modern processors also have other improvements that allow them to perform faster. This includes additional CPU cores and larger amounts of CPU cache memory that the CPU can work with. For example, if the cache is not sufficient the computer will wait until the instruction or the data will be retrieved from the RAM and the processor speed will be lower.
The Random Access Memory (RAM) modules inside a computer store temporary data for the Central Processing Unit. Data that cannot be stored inside the RAM must be accessed from the hard drive, slowing down the machine. More RAM means that more programs and larger files can be used simultaneously without any impact on performance. RAM is important because it eliminates the need to "swap" programs in and out.

Time and Space Limits for Programs

Different algorithms have different constants. Some operations are slow - modulo (that slows down the program significantly), if-else conditions, standard minimum/maximum etc., while others, like bitwise operations, are really fast and a good way to optimize a program (they can be used for faster min/max functions, for example).

Then, there are small tricks like adding a condition which for most test data makes the code skip some redundant operations, or noticing that it's hard to make test data for which some assumption doesn't hold. But that's another topic!The point is that sometimes, these tricks (they can be in your code even if you don't know about it) can improve your constant factor, or even complexity, noticeably.

In short, it all depends on experience. When you code more algorithms and note how fast they ran, you'll be able to estimate better what could pass later, even without yet coding it.

- Approximately you can perform about **10^8 operations in one second**. So with that you can estimate whether your algorithm will run in time or not.
- Approximately your program can consume 10^7 bytes of memory (≈ 10MB). From there on it goes on to allocate memory on your virtual memory.

Time Limit Example:
$N = 10^9$
$N*N = 10^{18} \approx 10^9$ Seconds $\approx 10^4$ days ≈ 30years
$NlogN = 10^9 * 30 \approx 30$ seconds

Memory Usage Example:
$10^6 \approx 1$ MB
$10^9 \approx 1$ GB
$10^{10} \approx 10$ GB

32bit vs 64bit architecture

- The key difference: 32-bit processors are perfectly capable of handling a limited amount of RAM, and 64-bit processors are capable of utilizing much more.
- As a general rule, if you have less than 4 GB of RAM in your computer, you don't need a 64-bit CPU, but if you have 4 GB or more, you do.
- More bits means our system can point to or address a larger number of locations in physical memory

One's Complement and Two's Complement

- Two's complement is especially important because it allows us to represent signed numbers in binary, and one's complement is the interim step to finding the two's complement.
- If all bits in a byte are inverted by changing each 1 to 0 and each 0 to 1, we have formed the one's complement of the number.
- Twos Complement is the negative representation of the number
  Example:
  65 -> 01000001
  -65 -> 10111111 [2's complement of 65]
- With a system like two's complement, the circuitry for addition and subtraction can be unified, whereas otherwise they would have to be treated as separate operations.
-

Important Links:

1. https://www.cs.cornell.edu/~tomf/notes/cps104/twoscomp.html
2. http://www.geeksforgeeks.org/memory-layout-of-c-program/
3. http://www.tenouk.com/ModuleZ.html
4. http://electronics.stackexchange.com/questions/123760/how-can-a-cpu-deliver-more-than-one-instruction-per-cycle
5. http://www.c-jump.com/CIS77/CPU/VonNeumann/lecture.html
6. http://www3.ntu.edu.sg/home/ehchua/programming/java/datarepresentation.html
7. https://en.wikipedia.org/wiki/Booting

Data Types, Operators And Bit Operations

Operators

Arithmetic Operators
+, -, *, /, %
post-increment (x++)
pre-increment (++x)
post-decrement (x--)
pre-decrement (--x)
These are used to perform arithmetic/mathematical operations on operands.

Relational Operators
== , != , >, <, >=, <=
Relational operators are used for comparison of two values.

Logical Operators (&&, ||, !)

Bitwise Operators (&, |, ^, ~, >> and <<)

Assignment Operators (=, +=, -=, *=, etc)

Other Operators (conditional, comma, sizeof, address, redirecton)

Bit Manipulations

1. **& (bitwise AND):** Takes two numbers as operand and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.
2. **| (bitwise OR):** Takes two numbers as operand and does OR on every bit of two numbers. The result of OR is 1 any of the two bits is 1.
3. **^ (bitwise XOR):** Takes two numbers as operand and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.
4. **~ (bitwise NOT):** Takes one number and inverts all bits of it

Truth Table

A	b	&	|	^	~a
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

- **<< (left shift):** Takes two numbers, left shifts the bits of first operand, the second operand decides the number of places to shift.
    - "A<< x" implies shifting the bits of A to the left by x positions.
    - The first x bits are lost this way. The last x bits have 0.
    - Example:
        A = 29 (11101) and x = 2,
        So A << 2 means
        0011101 << 2 = 1110100 (116)
    - **A << x is equal to multiplication by 2^x**.

- **>> (right shift):** Takes two numbers, right shifts the bits of first operand, the second operand decides the number of places to shift.
    - "A >> x" implies shifting the bits of A to the right by x positions.
    - The last x bits are lost this way.
    - Example :
        A = 29 (11101) and x = 2,
        So A >> 2 means
        0011101 >> 2 = 0000111 = 7
    - **A >> x is equal to division by 2^x**

Bit Tricks

- **x & (x-1)** will clear the lowest set bit of x
- **x & ~(x-1)** extracts the lowest set bit of x (all others are clear). Pretty patterns when applied to a linear sequence.

Problems

1. Check if a given number is power of 2 or not
   Solution: n & n-1

2. Check if 2 given two signed integers are of the same sign.
   Solution: a XOR b < 0

3. Write a program to print Binary representation of a given number.
   Solution: Extract bit by bit - loop or recursion

4. Find the number of set bits in integer
   Solution: n & n-1 -> unsets the Lesat Set Bit -> repeat untill n becomes 0.

5. Check if a integer is even or odd
   Solution: n&1

Complexity Analysis Of Algorithms

Asymptotic Analysis

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.

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 slower machine. The reason is the order of growth of Binary Search with respect to input size is 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 1000nlogn and 2nlogn time respectively on a machine. Both of these algorithms are asymptotically same (order of growth is nlogn). 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.

Worst, Best And Average Cases

Consider the following code of Linear Search

```
int linearSearch(int arr[], int n, int key) {
    for (int i=0; i<n; i++) {
        if (arr[i] == key)
            return i;
    }
    return -1;
}
```

Worst Case Analysis

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() functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be $\Theta(n)$.

Best Case Analysis

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 $\Theta(1)$.

Average Case Analysis

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).

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.

Analysis Of Loops

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.
   For example swap() function has O(1) time complexity.
   ```
   void swap (int x, int y) {
       int tmp = x;
       x = y;
       y = tmp;
   }
   ```
   Note tha loop/code that runs a constant number of times is also considered as O(1).

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
```

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

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

3. **O(n*c) :** 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.

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

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

4. **O(logn) :** Time Complexity of a loop is considered as O(logn) if the loop variables is divided/multiplied by a constant amount.

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

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

For example Binary Search(refer iterative implementation) has O(logn) time complexity.

Analysis Of Recursive Algorithms: Solving Recurrences

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 $T(n) = 2T(n/2) + n$. There are many other recursive 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 that the guess is correct or incorrect.

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

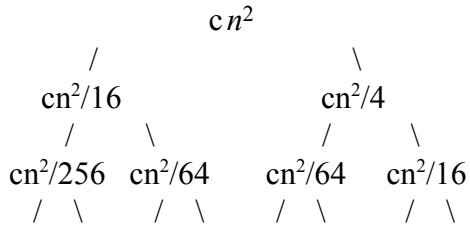
- We guess the solution as T(n) = O(nlogn). Now we use induction to prove our guess.
- We need to prove that T(n) <= (c*n*logn).
- We can assume that it is true for values smaller than n.

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&<= cn*2\log(n/2) + n \\
&= cn\log n - cn\log 2 + n \\
&= cn\log n - cn + n \\
&<= cn\log n
\end{aligned}
$$

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 an arithmetic or a geometric series.

For example consider the recurrence relation

$T(n) = T(n/4) + T(n/2) + cn^2$



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:

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

The above series is geometrical progression with ratio 5/16. To get an upper bound, we can sum the infinite series. [Sum = a/(1-r)]. We get the sum as $(n^2)/(1 - 5/16)$, which is $O(n^2)$

3) Master Theorem:

Master's Theorem is a direct way to get the complexity of a recurrence. It 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 $a >= 1$ and $b > 1$

$t = \log_b a$

There are following three cases:

1. If $f(n) = \Theta(n^c)$ where $c < t$ then $T(n) = \Theta(n^t)$
2. If $f(n) = \Theta(n^c)$ where $c = t$ then $T(n) = \Theta(n^t \log n)$
3. If $f(n) = \Theta(n^c)$ where $c > t$ then $T(n) = \Theta(n^c)$

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: The extra space or temporary space used by an algorithm.

Space Complexity: The 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 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.

Problems

1. 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("hello!!");
}
```

Solution:

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

b. 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 $\Theta(n \log n)$.

c. The expression $\Theta(\log n!) = \Theta(n \log n)$ can be easily derived from following Stirling's approximation (or Stirling's formula).
$\log n! = n \log n - n + O(\log(n))$

2. What is the time complexity of the following code:

```
int ans = 0;
for (int i = 1; i <= n; i++) {
   for (int j = 1; j <= i; j++ ) {
      for (int k = i+1; k <= n; k++) {
         ans++;
      }
   }
}
```

Solution: $T(n) = \sum_{k=1}^{n} k(n-k) = \Theta(n^3)$

Recursion

Recursion

Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem (as opposed to iteration). The approach can be applied to many types of problems, and recursion is one of the central ideas of computer science.

- Most computer programming languages support recursion by allowing a function to call itself within the program text.
- A recursive function definition has one or more base cases, meaning input(s) for which the function produces a result trivially (without recurring).

Examples: Convert json string to object, expression evaluator with brackets

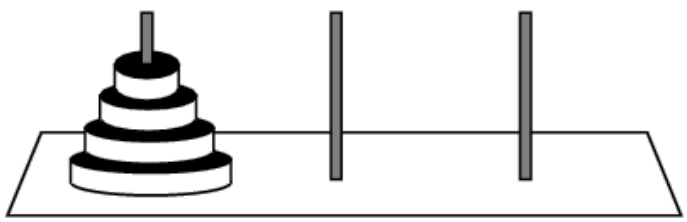
Problems

1. WAF to compute sum of numbers from 1 to n.

2. WAF to print Binary Representation of a given integer.

3. WAF to compute n^{th} Fibonacci number.

4. Tower of Hanoi – [http://mathworld.wolfram.com/TowerofHanoi.html] - Given a stack of n disks arranged from largest on the bottom to smallest on top placed on a rod, together with two empty rods, the towers of Hanoi puzzle asks for the minimum number of moves required to move the stack from one rod to another, where moves are allowed only if they place smaller disks on top of larger disks.



Solutions and Complexity

1. WAF to compute sum of numbers from 1 to n.

```
int sum(int n) {
   if(n == 0)    return 0;
   return n + sum(n-1);
}
```

Recurrence: T(n) = T(n-1) + 1
Time Complexity: O(n)

2. WAF to print Binary Representation of a given integer.

```
void binaryRepresentation(int n) {
   if (n==0)    return;
   binaryRepresentation(n/2);
   print n%2;
}
```

Recurrence: T(n) = T(n/2) + 1
Time Complexity: O(logn)

3. WAF to compute n^{th} Fibonacci number.

```
int fibo(int n) {
   if(n<=2)    return 1;
   return fibo(n-1) + fibo(n-2);
}
```

Recurrence: T(n) = T(n-1) + T(n-2) + 1
Time Complexity: $O(2^n)$
[http://cs.stackexchange.com/questions/14733/complexity-of-recursive-fibonacci-algorithm]

4. Tower of Hanoi [Simulation: http://towersofhanoi.info/Animate.aspx]

```
void ToH(int n, char src, char dest, char temp) {
   if(n==0)    return;
   move(n-1, src, temp, dest);
   print 'Move %d from %c to %c',n,src,dest;
   move(n-1, temp, dest, src);
}
```

Recurrence: T(n) = 2T(n-1) + 1
Time Complexity: $O(2^n)$

Sorting

Sorting Algorithms

1. **Bubble Sort**:
   Time Complexity: O (n^2)

2. **Insertion Sort**:
   Time Complexity: O (n^2)

3. **Selection Sort**:
   Time Complexity: O (n^2)

4. **Merge Sort**: Merge sort takes advantage of the ease of merging already sorted lists into a new sorted list. Split the array into 2 halves, sort the left part, sort the right part. It then merges each of the resulting sorted parts into the final sorted list.

```
void merge (int ar[], int begin, int end, int mid) {
   int br[end-begin+1];
   for (int i=begin, j=mid+1, k=0; k<end-begin+1; k++) {
      br[k] = (j==end+1)  ? ar[i++]
            : (i==mid+1)     ? ar[j++]
            : ar[j]<ar[i]       ? ar[j++]
            :                        ar[i++];
   }
   for (int i=begin; i<end+1; i++)
      ar[i] = br[i-begin];
}

void merge_sort (int ar[], int begin, int end) {
   if(begin >= end)      return;
   int mid = (begin + end)/2;
   merge_sort(ar, begin, mid);
   merge_sort(ar, mid+1, end);
   merge(ar, begin, end, mid);
}
```

Time Complexity: O (nlogn) [T(N) = 2T(N/2) + N]

5. **Quick Sort**: Pick an element ("pivot") from the given array and partition the array based on the pivot,ie, move items which are less than pivot to the left and the greater elements to the right. The pivot goes in the middle. Repeat the algorithm on the left and right parts again, until you get a partition of size 1 (which is sorted).

```
int partition(int ar[], int begin, int end) {
    int pivot = ar[end];
    int ret = 0, idx = begin;
    for (int j=begin; j<end; j++)
        if(ar[j]<=pivot) {
            ar[idx] = ar[j]^ar[idx]^(ar[j]=ar[idx]);
            idx++;
        }
    ar[idx] = ar[end]^ar[idx]^(ar[end]=ar[idx]);
    return idx;
}

void quicksort (int ar[], int begin, int end) {
    if(begin >= end)      return;
    int pivot_position = partition(ar, begin, end);
    quicksort(ar, begin, pivot_position-1);
    quicksort(ar, pivot_position+1, end);
}
```

Time Complexity: T(n) = T(k) + T(n-k-1) + n

Worst Case(k=0)	: T(n) = T(n-1) + n	: O (n^2)
Best Case(k=n/2)	: T(n) = 2T(n/2) + n	: O(nlogn)
Average Case(k=n/10)	: T(n) = T(n/10) + T(9n/10) + n	: O(nlogn)

6. **Counting Sort:**

```
void counting_sort(int ar[], int n, int k) {
    int count[k+1];
    fill(count, count+k+1, 0);
    for(int i=0; i<n; i++)
        count[ar[i]]++;
    int idx = 0;
    for(int i=0; i<k+1; i++)
        for(int j=0; j<count[i]; j++)
            ar[idx++] = i;
}
```

Time Complexity: O(n)

Problems

1. Given an integer array of election votes having candidate IDs, WAP to find the winner of the election.
   Solution: Counting Sort

2. Given an array of integers and a number K, WAP to check if there exists i,j such that ar[i] + ar[j] = K, $i \neq j$.
   Solution: Sorting and 2 pointers

3. WAP to count Inversions in an array. Inversion is defined as a pair of indexes (i,j) such that ar[i]>ar[j] and i<j.
   Solution: Merge Sort

4. WAP to merge 2 sorted arrays into 1. Array A is of size n+m, containing n elements and array B is of size m.
   Solution: Merge in array A using 2 pointers, starting from right hand size.

5. Median of two sorted array
   Solution:
   a. http://www.geeksforgeeks.org/median-of-two-sorted-arrays/
   b. http://www.geeksforgeeks.org/median-of-two-sorted-arrays-of-different-sizes/

Searching

Searching Algorithms

1. **Linear Search**:
   Time Complexity: O (N)

2. **Binary Search**:
   a. **Searching for an element in a given sorted array:**

```
bool binary_search(int ar[], int n, int target) {
   int lo = 0, hi = n-1, mid;
   while (lo <= hi) {
      mid = lo + (hi-lo)/2;
      if (ar[mid] == target)
         return true;
      else if (ar[mid] < target)
         lo = mid+1;
      else hi = mid-1;
   }
   return false;
}
```

   Time Complexity: O (log N) [T(n) = T(n/2) + O(1)]

   b. **Given an array and a target value, return the index of the first element in the array, which is greater than or equal to the target value.**
   Let us define a function p(x) which returns true if ar[x] is greater than or equal to the target value.

```
int binary_search(int ar[], int n, int target) {
   int lo = 0, hi = n-1, mid;
   while (lo < hi) {
      mid = lo + (hi-lo)/2;
      if (p(mid) == true)
         hi = mid;
      else
          lo = mid+1;
   }
   if (p(lo) == false)
      return n;              //p(x) is false for all x in the array
   return lo;                // lo is the least x for which p(x) is true
}
```

c. **Given an array and a target value, return the index of the last element in the array, which is less than or equal to the target value.**
Let us define a function p(x) which returns true if ar[x] is less than or equal to the target value.

```
int binary_search(int ar[], int n, int target) {
   int lo = 0, hi = n-1, mid;
   while (lo < hi) {
      mid = lo + (hi-lo+1)/2;
      if (p(mid) == true)
         lo = mid;
      else
         hi = mid-1;
   }
   if (p(lo) == false)
      return -1;            //p(x) is false for all x in the array
   return lo;               // lo is the greatest x for which p(x) is true
}
```

Problems

1. WAF to find floor value of input 'key' in a given array, say A = {-1, 2, 3, 5, 6, 8, 9, 10} and key = 7, we should return 6 as outcome.
   Solution: Use 2c explained above.

2. WAF to count the number of occurrences in a sorted array.
   Solution: Use 2b and 2c explained above.

3. WAF to search for a value in an m x n matrix. Integers in each row are sorted from left to right and the first integer of each row is greater than or equal to the last integer of the previous row.
   Solution: a. Binary Search in each row or column
   b. Use 2c on first column and then use binary search on the row.

Hashing

Introduction

Hashing is the solution that can be used in almost all such situations and performs extremely well compared to data structures like Array, Linked List, Balanced BST in practice. With hashing we get O(1) search time on average (under reasonable assumptions) and O(n) in worst case.

Hash Function:
A hash function is any function that can be used to map data of arbitrary size to data of fixed size. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes. One use is a data structure called a hash table, widely used in computer software for rapid data lookup. Hash functions accelerate table or database lookup by detecting duplicated records in a large file.

A good hash function should have following properties:

1. Efficiently computable.
2. Should uniformly distribute the keys (Each table position equally likely for each key)

For example, for phone numbers a bad hash function is to take first three digits. A better function is consider last three digits. Please note that this may not be the best hash function. There may be better ways.

Hash Table
In computing, a hash table (hash map) is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.

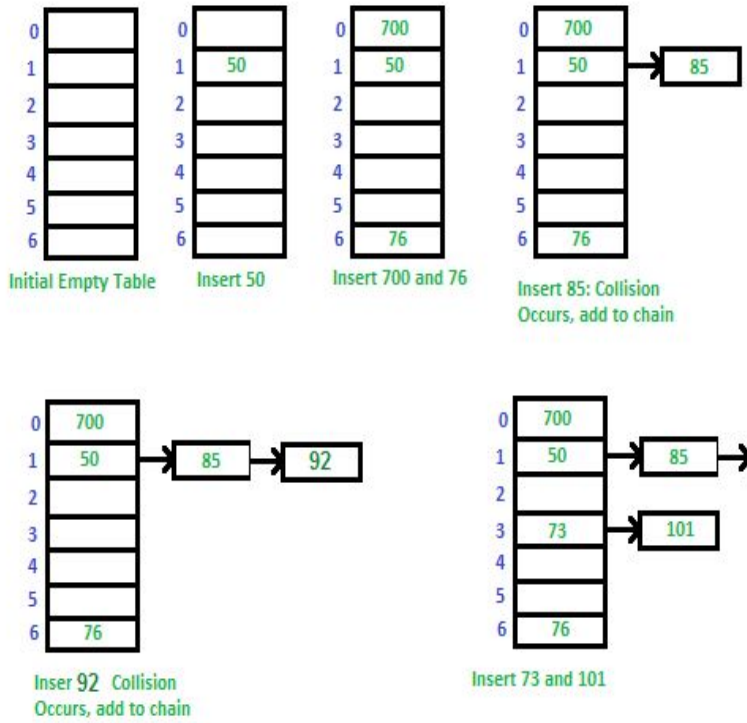
Ideally, the hash function will assign each key to a unique bucket, but it is possible that two keys will generate an identical hash causing both keys to point to the same bucket. Instead, most hash table designs assume that hash collisions—different keys that are assigned by the hash function to the same bucket—will occur and must be accommodated in some way.

In a well-dimensioned hash table, the average cost (number of instructions) for each lookup is independent of the number of elements stored in the table. Many hash table designs also allow arbitrary insertions and deletions of key-value pairs, at constant average cost per operation.

How to handle Collisions?
There are mainly two methods to handle collision:

1. Separate Chaining
2. Open Addressing

Separate Chaining
The idea is to make each cell of hash table point to a linked list of records that have same hash function value.
Let us consider a simple hash function as "key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Open Addressing
Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, size of table must be greater than or equal to total number of keys (Note that we can increase table size by copying old data if needed).

Insert(k): Keep probing until an empty slot is found. Once empty slot is found, insert k.

Search(k): Keep probing until slot's key equal to k or an empty slot is reached.

Delete(k): Delete operation is interesting. If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as "deleted". Insert can insert an item in a deleted slot, but search doesn't stop at a deleted slot.

Open Addressing is done following ways

1. **Linear Probing**
   In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also.

   Example
   Let hash(x) be the slot index computed using hash function and S be the table size. If slot hash(x) % S is full, then we try (hash(x) + 1) % S. If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S. If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S and so on.

   Let us consider a simple hash function as "key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

2. **Quadratic Probing**
   We look for i^2'th slot in i'th iteration.
   Let hash(x) be the slot index computed using hash function. If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S. If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S. If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S and so on.

3. **Double Hashing**
   We use another hash function hash2(x) and look for i*hash2(x) slot in i'th rotation.
   Let hash(x) be the slot index computed using hash function. If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S. If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S. If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S and so on.

Problems

1. Length of the largest sub-array with elements that can be arranged in a continuous sequence (elements can be repeated)
   Solution: Insert elements in a hash table for each sub-array(i...j) and maintain min/max as well. If for a subarray from [i,j] max-min == (j-i+1) and sizeOf(hash table) == j-i+1, this means [i,j] is a valid subarray.

2. Find whether an array A is subset of another array B.
   Solution: Create hash table for B and check if it has all the elements of A.

3. How to check if two given sets are disjoint
   Solution: Create hash table for 1 set and check if it has any element of the 2nd set

4. Find four elements a, b, c and d in an array such that a+b = c+d (all elements are distinct in the array)
   Solution: For all possibe pair sums, create hash table with sum as key and value as pair.

5. Given an array of strings, return all groups of strings that are anagrams.
   Solution: Create hash table with sorted form of string as key and value as list of actual strings.

6. Count distinct elements in every window of size k
   Solution: Create hash table for the 1st window of size k. When you move to the next window, remove the element which is left out and insert the new element in the hash table. At every step, size of the hash table is the count of distinct elements in that window.

References

1. http://courses.csail.mit.edu/6.006/fall11/lectures/lecture10.pdf
   http://courses.csail.mit.edu/6.006/fall09/lecture_notes/lecture05.pdf
2. http://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html
3. https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html
4. http://www.cplusplus.com/reference/unordered_map/unordered_map/
5. http://www.cplusplus.com/reference/unordered_set/unordered_set/

Linked Lists

Types of Linked Lists

Singly Linked List: Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in sequence of nodes. The operations we can perform on singly linked lists are insertion, deletion and traversal.

Doubly Linked List: In a doubly linked list, each node contains two links the first link points to the previous node and the next link points to the next node in the sequence.

Circular Linked List: In the circular linked list the last node of the list contains the address of the first node and forms a circular chain. A simple example is keeping track of whose turn it is in a multi-player board game.

Code for Doubly Linked List:

```
#include <iostream>
#include<cstdlib>
using namespace std;

typedef struct node {
    int data;
    struct node *next, *prev;
} node;

typedef struct LinkedList {
    node *root, *last;
    int size;
} LinkedList;

node* createNode(int num) {
    node* newNode =  (node*)malloc(sizeof(node*));
    newNode->data = num;
    newNode->next = NULL;
    newNode->prev = NULL;
    return newNode;
}
```

```
// It creates a dummy node as the first node of the linked list
LinkedList* createLinkedList() {
    LinkedList* newLinkedList =  (LinkedList*)malloc(sizeof(LinkedList*));
    node* newNode =  createNode(0);
    newLinkedList->root = newNode;
    newLinkedList->last = newNode;
    newLinkedList->size = 0;
    return newLinkedList;
}

void push_back(LinkedList* linkedList, int num) {
    node* newNode = createNode(num);
    linkedList->last->next = newNode;
    newNode->prev = linkedList->last;
    linkedList->last = newNode;;
    linkedList->size = linkedList->size + 1;
}

void push_front(LinkedList* linkedList, int num) {
    node* newNode = createNode(num);
    newNode->next = linkedList->root->next;
    newNode->prev = linkedList->root;
    linkedList->root->next = newNode;
    if(newNode->next != NULL)
        newNode->next->prev = newNode;
    else
        linkedList->last = newNode;

    linkedList->size = linkedList->size + 1;
}

int pop_back(LinkedList* linkedList) {
    if(linkedList->size == 0)
        return -1;

    node* last = linkedList->last;
    linkedList->last = last->prev;
    linkedList->last->next = NULL;

    int data = last->data;
    free(last);
    linkedList->size = linkedList->size - 1;
    return data;
}
```

```
int pop_front(LinkedList* linkedList) {
    if(linkedList->size == 0)
        return -1;

    node* front = linkedList->root->next;
    linkedList->root->next = front->next;

    if(front->next != NULL)
        front->next->prev = linkedList->root;

    int data = front->data;
    free(front);
    linkedList->size = linkedList->size - 1;
    return data;
}

void deleteNode(LinkedList* linkedList, int num) {
    node* current = linkedList->root;
    while(current->next != NULL) {
        if(current->next->data == num) {
            node* tmp = current->next;
            tmp->next->prev = current;
            current->next = tmp->next;
            free(tmp);
            linkedList->size = linkedList->size - 1;
            break;
        }
        current=current->next;
    }
}

int back(LinkedList* linkedList) {
    if(linkedList->size == 0)
        return -1;
    return linkedList->last->data;
}

int front(LinkedList* linkedList) {
    if(linkedList->size == 0)
        return -1;
    return linkedList->root->next->data;
}

bool isEmpty(LinkedList* linkedList) {
    return linkedList->size == 0;
}
```

```
void printList(LinkedList* linkedList) {
    node* root = linkedList->root->next;
    while(root != NULL) {
        cout<<root->data<<"->";
        root = root->next;
    }
    cout<<"NULL"<<endl;
}

int main() {
    LinkedList* linkedList = createLinkedList();
    cout<<"isEmpty:"<<(isEmpty(linkedList)?"True":"False")<<endl;

    push_front(linkedList, 2);
    push_back(linkedList, 3);

    printList(linkedList);
    pop_front(linkedList);
    printList(linkedList);

    push_back(linkedList, 4);
    push_front(linkedList, 1);
    push_back(linkedList, 5);

    printList(linkedList);
    deleteNode(linkedList, 4);
    printList(linkedList);
    pop_back(linkedList);
    printList(linkedList);
    push_back(linkedList, 5);

    cout<<"isEmpty:"<<(isEmpty(linkedList)?"True":"False")<<endl;
    cout<<"Length:"<<length(linkedList)<<endl;
    printList(linkedList);

    return 0;
}
```

Problems

1. Reverse a linked list – Iterative and Recursive.
   Example: 1->2->3->4->5->NULL, Ans: 5->4->3->2->1->NULL

Iterative	Recursive

node *reverse(node *head) {
  node *prev = NULL, *tmp;
 	
  while(head != NULL) {
  tmp = head->next;
  head->next = prev;
  prev = head;
  head = tmp;
  }
  return prev;
}
	node *reverse(node *head) {
	  if(head == NULL)
	  return head;
	
	  node *ret = reverse(head->next);
	  head->next->next = head;
	  head->next=NULL;
	  return ret;
	}

2. Find the mid-point of a linked-list.
   Example: 1->2->3->4->5->NULL, Ans: 3

   Solution: Take 2 pointers, slow and fast. Move slow pointer by 1 node and fast pointer by 2 nodes. The slow pointer is the middle element of the linked list.node.

3. Sort a singly linked list.
   Solution: Use merge sort
   a. Split the list into 2 halves
   b. Sort the left and right sublist individually
   c. Merge the two sorted lists

4. You are given a Double Link List with one pointer of each node pointing to the next node just like in a single link list. The second pointer however can point to any node in the list and not just the previous node. WAP to clone such a linked list,ie, a linked list with next and random pointer.
   Solution
   a. Create a copy of each node and insert it between the current and the next node.
   b. Update random pointer for the new nodes by:
   original->next->random = original->random->next;
   c. Restore the original and copy linked lists using:
   original->next = original->next->next;
   copy->next = copy->next->next;

5. Implement LRU cache: Given the order of access of page numbers and the cache (or memory) size, LRU caching scheme is to remove the least recently used frame when the cache is full and a new page is referenced which is not there in cache.

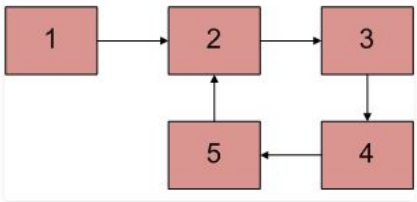
Solution: Use doubly linked list and a map of <PageNumber, Node> to perform operations efficiently. Keep a dummy node to ease things.

a. Keep 2 pointer front and rear – front always points to LRU page and rear always points to the RU node.
b. When a page is accessed:
    i. If it's present in the LL (check using HashTable), move it to the end of the list and update the prev/next pointers. Update rear pointer as well
    ii. If it's not present in the LL(check using HashTable):
        1. If size of LL less than cache size, insert it at the rear of the LL, update rear pointer and insert in the HashTable as well.
        2. Else, remove the LRU (front of the LL) node, both from the LL and the HashTable, and insert new node at the rear of the LL, update rear pointer and insert in the HashTable as well.

6. Add two numbers represented by linked lists into a new Linked List.
Example: 5->6->3->NULL + 8->4->2->NULL, Ans: 1->4->->0->5->NULL

    Solution: Reverse the 2 given linked list, add node by node and maintain a carry pointer into a new list and finally reverse the new list.

7. Detect and Remove Loop in a Linked List.



    Solution: Floyd's cycle-finding algorithm

    a. Slow and fast pointer to get the pointer to a loop node.
    b. Find the length of the loop, say k.
    c. Take 2 pointers pointing at head, move one of them by k.
    d. Move both pointers one at a time, their intersection point is the last/first node of the loop, make node->next = NULL.

8. Implement Stack with getMiddle operation in O(1)
Solution: Use Doubly Linked List and keep a mid pointer, adjust when you perform push and pop operations.

References

1. http://www.geeksforgeeks.org/linked-list-vs-array/
2. http://www.geeksforgeeks.org/xor-linked-list-a-memory-efficient-doubly-linked-list-set-1/
3. http://www.geeksforgeeks.org/xor-linked-list-a-memory-efficient-doubly-linked-list-set-2/

Stacks

Introduction

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

Implementation

There are two ways to implement a stack:

- Using array
- Using linked list

Implementation Using Array

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a stack
struct Stack {
    int top;
    unsigned capacity;
    int* array;
};

// Function to create a stack of given capacity. It initializes size of stack as 0
struct Stack* createStack(unsigned capacity) {
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int*) malloc(stack->capacity * sizeof(int));
    return stack;
}

// Stack is full when top is equal to the last index
int isFull(struct Stack* stack) {
    return stack->top == stack->capacity - 1;
}
```

```
// Stack is empty when top is equal to -1
int isEmpty(struct Stack* stack) {
    return stack->top == -1;
}

// Function to add an item to stack.  It increases top by 1
void push(struct Stack* stack, int item) {
    if (isFull(stack))
        return;
    stack->array[++stack->top] = item;
    printf("%d pushed to stack\n", item);
}

// Function to remove an item from stack.  It decreases top by 1
int pop(struct Stack* stack) {
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top--];
}

// Function to get top item from stack
int peek(struct Stack* stack) {
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top];
}

int main() {
    struct Stack* stack = createStack(100);

    push(stack, 10);
    push(stack, 20);
    push(stack, 30);

    printf("%d popped from stack\n", pop(stack));
    printf("Top item is %d\n", peek(stack));
    return 0;
}
```

Problems

1. Design a data structure for top(), push(), pop() and getMin() operations in O(1) time complexity.
   Solution: Maintain two stacks, one for the actual array elements and other for maintaining minimum element so far.

2. Find the next smallest element on right side for every element a[i] in array a.
   Solution: Start from right side of the array and maintain a stack of elements processed so far. Now, for every element a[i], pop all elements which are less than current element from the stack and set the top of stack as the next greater element for ar[i]. Push a[i] in the s.Repeat for all elements (from right hand-side)

3. Evaluation of Postfix Expression
   Example:
   Input: 2 3 1 * + 9 -
   Output -4 [3*1 + 2 - 9]
   Solution: Keep pushing the operands onto a stack. Whenever you encounter an operator, pop two operands from the stack, perform operation and push the result back into the stack.

4. Check for balanced parentheses in an expression.
   Solution: When you encounter an opening bracket, push it onto the stack. When you encounter a closing bracket, if stack is not empty, pop top element from the stack, else report “Invalid”. If the stack is empty at the end of the process, report “Valid”, else report “Invalid”.

Queues

Introduction

Like Stack, Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Implementation

There are two ways to implement a queue:

- Using array
- Using linked list

Array implementation Of Queue

For implementing queue, we need to keep track of two indices, front and rear. We enqueue an item at the rear and dequeue an item from front. If we simply increment front and rear indices, then there may be problems, front may reach end of the array. The solution to this problem is to increase front and rear in circular manner

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a queue
struct Queue {
    int front, rear, size;
    unsigned capacity;
    int* array;
};

// Function to create a queue of given capacity. It initializes size of queue as 0
struct Queue* createQueue(unsigned capacity) {
    struct Queue* queue = (struct Queue*) malloc(sizeof(struct Queue));
    queue->capacity = capacity;
    queue->front = queue->size = 0;
    queue->rear = capacity - 1;  // This is important, see the enqueue
    queue->array = (int*) malloc(queue->capacity * sizeof(int));
    return queue;
}
```

```
// Queue is full when size becomes equal to the capacity
int isFull(struct Queue* queue) {
    return (queue->size == queue->capacity);
}

// Queue is empty when size is 0
int isEmpty(struct Queue* queue) {
    return (queue->size == 0);
}

// Function to add an item to the queue.  It changes rear and size
void enqueue(struct Queue* queue, int item) {
    if (isFull(queue))
        return;
    queue->rear = (queue->rear + 1)%queue->capacity;
    queue->array[queue->rear] = item;
    queue->size = queue->size + 1;
    printf("%d enqueued to queue\n", item);
}

// Function to remove an item from queue.  It changes front and size
int dequeue(struct Queue* queue) {
    if (isEmpty(queue))
        return INT_MIN;
    int item = queue->array[queue->front];
    queue->front = (queue->front + 1)%queue->capacity;
    queue->size = queue->size - 1;
    return item;
}

// Function to get front of queue
int front(struct Queue* queue) {
    if (isEmpty(queue))
        return INT_MIN;
    return queue->array[queue->front];
}

// Function to get rear of queue
int rear(struct Queue* queue) {
    if (isEmpty(queue))
        return INT_MIN;
    return queue->array[queue->rear];
}
```

```
int main() {
    struct Queue* queue = createQueue(1000);

    enqueue(queue, 10);
    enqueue(queue, 20);
    enqueue(queue, 30);
    enqueue(queue, 40);

    printf("%d dequeued from queue\n", dequeue(queue));
    printf("Front item is %d\n", front(queue));
    printf("Rear item is %d\n", rear(queue));

    return 0;
}
```

Introduction to Dequeue
Dequeue or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends.

Dequeue primarily supports the following basic operations

1. **insetFront():** Adds an item at the front of Dequeue.
2. **insertLast():** Adds an item at the rear of Dequeue.
3. **deleteFront():** Deletes an item from front of Dequeue.
4. **deleteLast():** Deletes an item from rear of Dequeue.

Problems

1. Implement Queue using Stacks
   Solution: Use two stacks one for push and one for pop

2. Design Queue for getMin() in O(1) time complexity.
   Solution: One queue and One dequeue

3. Find maximum number in sliding window of k.
   Example : if array is { 1,5,7,2,1,3,4} and k=3, then
   first window is {1,5,7} so maximum is 7, print 7, then
   next window is {5,7,2} maximum is 7, print 7, then
   next window is {7,2,1} maximum is 7, print 7, and so on.
   Final output is : { 7,7,7,3,4 }

   Solution: Use Dequeue for maintaining the elements of the window, with the front storing the maximum element for the current window. For the new element of the next window, remove elements from the back of the dequeue till the elements are smaller than the current element. Also, for every next window, remove the left out element from the dequeue.

Trees

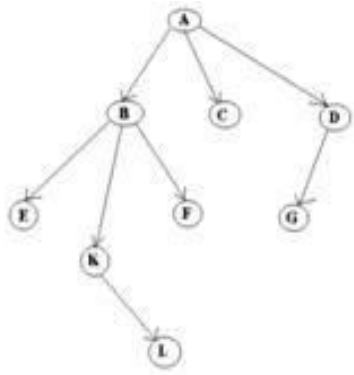
Tree

A tree is a collection of nodes connected by directed (or undirected) edges. A tree is a nonlinear data structure, compared to arrays, linked lists, stacks and queues which are linear data structures. A tree can be empty with no nodes or a tree is a structure consisting of one node called the root and zero or one or more subtrees.

A tree data structure can be defined recursively (locally) as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to the root.

A tree has following general properties:

1. One node is distinguished as a root;
2. Every node (exclude a root) is connected by a directed edge from exactly one other node; A direction is: parent -> children



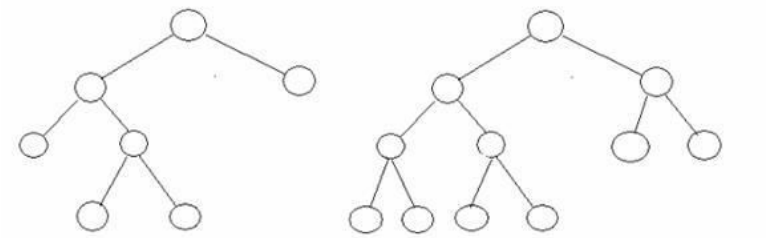
A is a parent of B, C, D. B is called a child of A.
On the other hand, B is a parent of E, F, K.
In the above picture, the root has 3 subtrees.

Each node can have arbitrary number of children. Nodes with no children are called leaves, or external nodes. In the above picture, C, E, F, L, G are leaves. Nodes, which are not leaves, are called internal nodes. Internal nodes have at least one child.

Nodes with the same parent are called siblings. In the picture, B, C, D are called siblings. The depth of a node is the number of edges from the root to the node. The depth of K is 2. The height of a node is the number of edges from the node to the deepest leaf. The height of B is 2. The height of a tree is a height of a root.

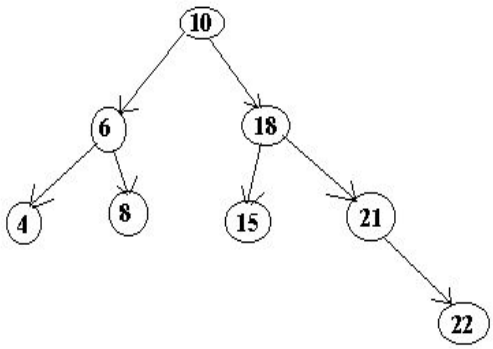
Binary Trees
A tree where each node can have no more than two children is called a Binary Tree.

A binary tree in which each node has exactly zero or two children is called a full binary tree. In a full tree, there are no nodes with exactly one child.
A complete binary tree is a tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right. A complete binary tree of the height h has between 2^h and $2^{(h+1)}$-1 nodes. Here are some examples:



Full Tree **Complete Tree**

Binary Search Trees
Given a binary tree, suppose we visit each node (recursively) as follows. We visit left child, then root and then the right child. For example, visiting the following tree



In the order defined above, this tree will produce the sequence {4, 6, 8, 10, 15, 18, 21, 22} which we call flat(T). A binary search tree (BST) is a tree, where flat(T) is an ordered sequence.
In other words, a binary search tree can be "searched" efficiently using this ordering property. A "balanced" binary search tree can be searched in O(log n) time, where n is the number of nodes in the tree.

A binary tree is a binary search tree (BST) if and only if an inorder traversal of the binary tree results in a sorted sequence. The idea of a binary search tree is that data is stored according to an order, so that it can be retrieved very efficiently.

Traversal methods

Pre-order
1. Display the data part of the root (or current node).
2. Traverse the left subtree by recursively calling the pre-order function.
3. Traverse the right subtree by recursively calling the pre-order function.

In-order
1. Traverse the left subtree by recursively calling the in-order function.
2. Display the data part of the root (or current node).
3. Traverse the right subtree by recursively calling the in-order function.

Post-order

1. Traverse the left subtree by recursively calling the post-order function.
2. Traverse the right subtree by recursively calling the post-order function.
3. Display the data part of the root (or current node).

Code for Binary Search Tree

```
#include <iostream>
using namespace std;

typedef struct node{
   int data;
   struct node *left, *right;
} tree;

tree *createNode(int data) {
   tree *root = (tree *)(malloc)(sizeof(tree *));
   root->data = data;
   root->left = root->right = NULL;
   return root;
}

void inorder(tree *root) {
   if(root == NULL)
      return;
   inorder(root->left);
   cout<<root->data<<" ";
   inorder(root->right);
}

void preorder(tree *root) {
   if(root == NULL)
      return;
   cout<<root->data<<" ";
   preorder(root->left);
   preorder(root->right);
}

void postorder(tree *root) {
   if(root == NULL)
      return;
   postorder(root->left);
   postorder(root->right);
   cout<<root->data<<" ";
}
```

```
int findMax(tree *root) {
   while(root->right!=NULL)
      root = root->right;
   return root->data;
}
tree *insert(int data, tree *root) {
   if(root == NULL)
      return createNode(data);
   if(root->data>data)
      root->left = insert(data, root->left);
   else
      root->right = insert(data, root->right);
   return root;
}

bool search(int data, tree *root) {
   if(root==NULL)
      return false;
   if(root->data == data)
      return true;
   if(root->data > data)
      return search(data, root->left);
   return search(data, root->right);
}

node *deleteNode(int data, tree *root) {
   if(root == NULL)
      return NULL;
   if(root->data > data)
      root->left = deleteNode(data, root->left);
   else if(root->data < data)
      root->right = deleteNode(data, root->right);
   else {
      if(root->left==NULL) {
         node *tmp = root->right;
         free(root);
         return tmp;
      } else if(root->right == NULL) {
         node *tmp = root->left;
         free(root);
         return tmp;
      } else {
         root->data = findMax(root->left);
         root->left = deleteNode(root->data, root->left);
      }
   }
   return root;
}
```

```
int main() {
    tree *root = createNode(4);
    insert(2, root);     insert(1, root);
    insert(3, root);     insert(6, root);
    insert(5, root);     insert(8, root);
    insert(7, root);     insert(10, root);

    inorder(root);
    cout<<endl;
    preorder(root);
    cout<<endl;
    postorder(root); cout<<endl;
    cout<<search(5, root)<<endl;
    cout<<search(4, root)<<endl;
    cout<<search(12, root)<<endl;
    cout<<search(10, root)<<endl;
    cout<<search(-1, root)<<endl;

    root = deleteNode(10, root);
    root = deleteNode(4, root);
    root = deleteNode(5, root);
    root = deleteNode(-1, root);

    inorder(root);     cout<<endl;
    preorder(root);   cout<<endl;
    postorder(root);  cout<<endl;
}
```

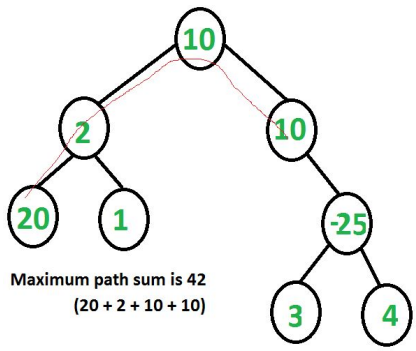
Problems

1. WAF to find the height of a binary tree.
   Solution
   ```
   int height(node *root) {
      if(root == NULL)
         return 0;
      return 1+max(height(root->left), height(root->right));
      }
   ```

2. WAF to convert a binary tree into its mirror tree
   Solution
   ```
   void mirror(node *root) {
      if(root == NULL)
         return ;
      node *tmp = root->left;
      root->left = root->right;
      root->right = tmp;
      mirror(root->left);
            mirror(root->right);
      }
   ```

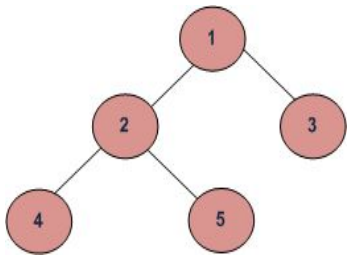
3. Check if a given Binary Tree is a Binary Search Tree or not.
   Solution: The trick is to write a utility helper function isBSTUtil(node* node, int min, int max) that traverses down the tree keeping track of the allowed values of min and max as it goes, looking at each node only once. The initial values for min and max should be INT_MIN and INT_MAX — they narrow from there.

4. Find maximum non empty path sum in a binary tree.
   Solution
   a. For every node, find the maximum sum single path in the left subtree and in the right subtree.
   b. Check the highest sum path which you can form by including the current node.
   c. Update your global answer
   d. Return the maximum possible single path sum from using current node to the parent calling function.




```
int findMaxUtil(Node* root, int &res) {
    if (root == NULL)
        return 0;

    int l = findMaxUtil(root->left, res);
    int r = findMaxUtil(root->right, res);

    // Max path for parent call. This path must include only 1 child of root
    int max_single = max(max(l, r) + root->data, root->data);

    // max_top represents the sum when the current node is the root of the
    maxsum path and no ancestors of root are there in max sum path
    int max_top = max(max_single, l + r + root->data);
    res = max(res, max_top); // Store the Maximum Result.
    return max_single;
}

// Returns maximum path sum in tree with given root
int findMaxSum(Node *root) {
    int res = INT_MIN;
    findMaxUtil(root, res);
    return res;
}
```

5. Level order traversal of a tree.



Level order traversal of this tree is 1 2 3 4 5
Solution: Keep the elements in a queue data structure

```
void levelOrderTraversal(node *root) {
   if(root == NULL)
      return ;
   queue<node*> q;
   q.push(root);

   while(!q.empty()) {
      node *tmp = q.front();
      q.pop();
      cout<<tmp->data<<" ";
      if(tmp->left)
         q.push(tmp->left);
      if(tmp->right)
         q.push(tmp->right);
   }
}
```

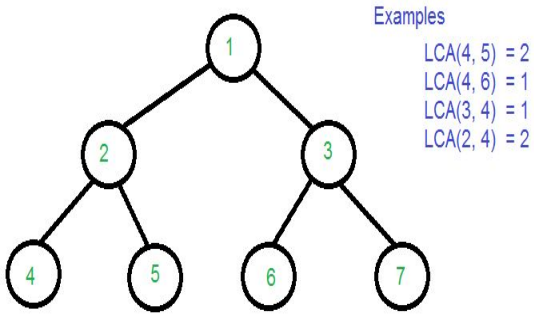
6. Iterative Traversals of Binary Trees.
   a. **In-order**
      i. Create an empty stack S.
      ii. Initialize current node as root
      iii. Push the current node to S and set current = current->left until current is NULL
      iv. If current is NULL and stack is not empty then
      v. Pop the top item from stack.
      vi. Print the popped item, set current = popped_item->right
      vii. Go to step c.
      viii. If current is NULL and stack is empty then we are done.

```
void inOrderIterative(node *root) {
   node *current = root;
   stack< node* > nodeStack;
   while (1) {
      while(current !=  NULL)  {
         nodeStack.push(current);
         current = current->left;
      }

      if (!nodeStack.empty()) {
         current = nodeStack.top();
         nodeStack.pop();
         cout<<(current->data)<<" ";
         current = current->right;
      }
      else break;
   }
}
```

b. **Pre-order**
   i. Create an empty stack nodeStack and push root node to stack.
   ii. Do following while nodeStack is not empty.
      1. Pop an item from stack and print it.
      2. Push right child of popped item to stack
      3. Push left child of popped item to stack

c. **Post-order**
   i. Implement a variation of Pre-order traversal of the form – DRL [not DLR]
   ii. Instead of printing the elements in this order, push the elements to another stack as and when you print.
   iii. After exhausting the entire tree, you can print the second stack and you will get Post-order traversal of the tree – LRD [reverse of DRL].

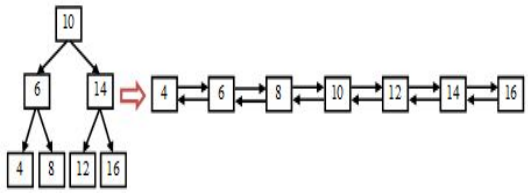
7. Given 2 nodes, WAF to find the Least Common Ancestor(LCA) in a Binary Tree



Solution: The idea is to traverse the tree starting from root. If any of the given keys (n1 and n2) matches with root, then root is LCA (assuming that both keys are present). If root doesn't match with any of the keys, we recur for left and right subtree. The node which has one key present in its left subtree and the other key present in right subtree is the LCA. If both keys lie in left subtree, then left subtree has LCA, otherwise LCA lies in right subtree.

```
node *findLCA(node* root, int n1, int n2) {
   if (root == NULL)
      return NULL;
   if (root->data == n1 || root-> data == n2)
      return root;

   node *left_lca  = findLCA(root->left, n1, n2);
   node *right_lca = findLCA(root->right, n1, n2);
   if (left_lca && right_lca)
      return root;
   return (left_lca != NULL)? left_lca: right_lca;
}
```

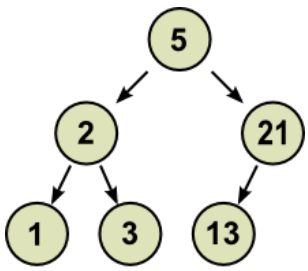
8. Given a Binary Tree (BT), convert it to a Doubly Linked List(DLL). The left and right pointers in nodes are to be used as previous and next pointers respectively in converted DLL. The order of nodes in DLL must be same as Inorder of the given Binary Tree. The first node of Inorder traversal (left most node in BT) must be head node of the DLL.

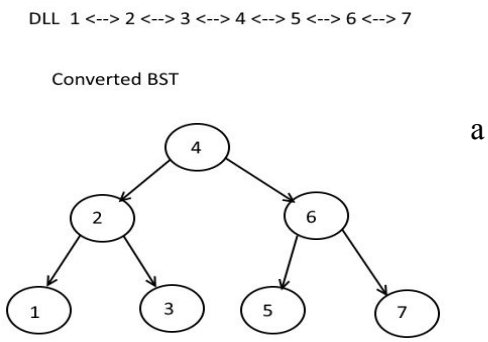


Solution: The idea is to do inorder traversal of the binary tree. While doing inorder traversal, keep track of the previously visited node in a variable say *prev*. For every visited node, make it next of *prev* and previous of this node as *prev*.

```
node *BinaryTree2DoubleLinkedList(node *root) {
   if (root == NULL) return NULL;
   static node* prev = NULL;

   node *head = BinaryTree2DoubleLinkedList(root->left);
   if (prev == NULL)
      head = root;
   else {
      root->left = prev;
      prev->right = root;
   }
   prev = root;
   BinaryTree2DoubleLinkedList(root->right);
   return head;
}
```

9. Given a Doubly Linked List which has data members sorted in ascending order. Construct Balanced Binary Search Tree which has same data members as the given Doubly Linked List. The tree must be constructed in-place.





a

Solution: Let's construct the BST from leaves to root. The idea is to insert nodes in BST in the same order as they appear in Doubly Linked List, so that the tree can be constructed in O(n) time complexity. We first count the number of nodes in the given Linked List. Let the count be n. After counting nodes, we take left n/2 nodes and recursively construct the left subtree. After left subtree is constructed, we assign middle node to root and link the left subtree with root. Finally, we recursively construct the right subtree and link it with root. While constructing the BST, we also keep moving the list head pointer to next so that we have the appropriate pointer in each recursive call.

```
int countNodes(node *head) {
   int count = 0;
   while(head) {
      head = head->right;
      count++;
   }
   return count;
}

node* sortedListToBSTRecur(node **head_ref, int n) {
   if (n <= 0)
      return NULL;

   node *left = sortedListToBSTRecur(head_ref, n/2);
   node *root = *head_ref;
   root->left = left;
   *head_ref = (*head_ref)->right;
   root->right = sortedListToBSTRecur(head_ref, n-n/2-1);

   return root;
}

node* sortedListToBST(node *head) {
   int n = countNodes(head);
   return sortedListToBSTRecur(&head, n);
}
```

Priority Queues And Heaps

Introduction to Priority Queue

In computer science, a **priority queue** is an **abstract data type** , which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority. If two elements have the same priority, they are served according to their order in the queue.

Queues are a standard mechanism for ordering tasks on a first-come, first-served basis. However, some tasks may be more important or timely than others (higher priority). Priority queues store tasks using a partial ordering based on priority and ensure highest priority task ahead of queue. Heaps are the underlying data structure of priority queues.

Priority Queue primarily supports the following three basic operations

1. **getTop( ):** Fetching the top priority element
2. **insert( ):** Insertion of an element
3. **deleteTop():** deleting the top priority element

Implementations of Priority Queue

Binary Heap:

- O(1) getTop
- O($\log_2$n) insert
- O($\log_2$n) deleteTop

Heap is one maximally efficient implementation, and in fact priority queues are often referred to as "**heaps**", regardless of how they may be implemented.
There are two kind of priority queue: **max-priority queue** and **min-priority queue**.
Max-heap is used for max-priority queue and Min-heap is used for min-priority queue.

Note: A *heap* data structure should not be confused with *the heap* which is a common name for the pool of memory from which dynamically allocated memory is allocated. The term was originally used only for the data structure.

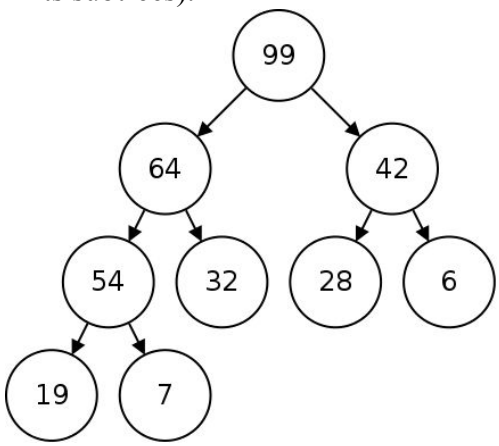
Applications of Priority Queue

- Priority scheduling of processes in OS
- Dijkstra's Algorithm for shortest path
- Prim's Algorithm for minimum spanning tree
- Heapsort

Heaps

A **heap** is a binary tree (in which each node contains a Comparable key value), with two special properties:

A. The **ORDER** property: For every node n, the value in n is **greater than or equal to** the values in its children (and thus is also greater than or equal to all of the values in its subtrees).



This heap is called a *max heap* because the root node is the maximum.

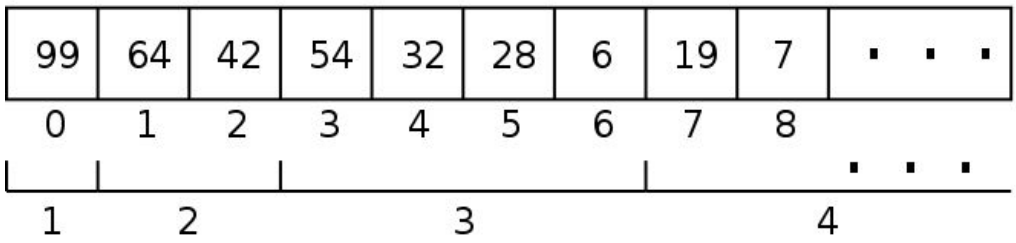
B. The **SHAPE** property:
   1. All leaves are either at depth d or d-1 (for some value d).
   2. All of the leaves at depth d-1 are to the **right** of the leaves at depth d.
   3.
      a. There is at most 1 node with just 1 child.
      b. That child is the **left** child of its parent, and
      c. It is the **rightmost** leaf at depth d.

Implementation of Heaps

Heaps support a number of different operations:

1. create - Create an empty heap.
2. insert - Add an element to the heap.
3. getTop - Return the top element.
4. delete - Remove the top element from the heap.

How these are implemented depends on how the heap is implemented. We can implement trees using a linked structure as we did for binary search trees. However, we can also implement a tree with an array. Doing this with a heap, we will store the root in array location 0, then the two children of the root in the next two locations. The rule that the heap must be balanced and left-aligned implies that there is only one array location where each item can be stored.



a. To find the left child of a node, we can use this formula:
left(node) = node * 2 + 1

b. The right child is directly after the left child, so that formula is:
right(node) = node * 2 + 2

c. We can find the parent of any node fairly easily as well:
parent(node) = (node - 1) / 2

Implementation of Binary Max-Heap:

```
#include<iostream>
#include<cstdio>
#include<cstdlib>
#include<algorithm>

using namespace std;

typedef struct heap {
   int *array;
   int capacity;
   int size;
}maxHeap;

// Heap Creation (Max Heap) : capacity
maxHeap Create(int capacity) {
   maxHeap h = (struct heap*)malloc(sizeof(struct heap));
   if(h == NULL) {
      cout<<"Memory Error"<<endl;
      return NULL;
   }
   h->size = 0;
   h->capacity = capacity;
   h->array = (int *)malloc(sizeof(int)*capacity);
   if(h->array == NULL) {
      cout<<"Memory Error"<<endl;
      return NULL;
   }
   return h;
}
```

```
// Getting Left Child
int GetLeftChild(maxHeap h, int i) {
    int left = 2*i+1;
    if(h->size > left)
        return left;
    else
        return -1;
}

// Getting Right Child
int GetRightChild(maxHeap h, int i) {
    int right = 2*i+2;
    if(h->size > right)
        return right;
    else
        return -1;
}

// Getting Parent
int GetParent(maxHeap h, int i) {
    if(i == 0)
        return -1;
    else
        return h->array[(i-1)/2];
}

// Heapify : Maintaining Heap property
void heapify(maxHeap h, int i) {
    int l = GetLeftChild(h,i);
    int r = GetRightChild(h,i);
    if(l == -1 && r == -1)
        return;

    int max_index, temp;
    max_index = i;
    if(l != -1 && h->array[l] > h->array[i])
        max_index=l;
    if(r != -1 && h->array[r] > h->array[max_index])
        max_index=r;

    if(max_index != i) {
        temp=h->array[i];
        h->array[i]=h->array[max_index];
        h->array[max_index]=temp;
        heapify(h,max_index);
    }
}
```

```
// Getting Maximum Element (Max Heap) : O(1)
int GetMaximum(maxHeap h) {
    if(h->size == 0)
        return -1;
    return h->array[0];
}


// Insertion : O(logn)
void insert(maxHeap h, int val) {
    int i, temp;
    i=h->size;
    h->array[i]=val;
    h->size++;
    while(i != 0 && GetParent(h,i) < h->array[i]) {
        temp=GetParent(h,i);
        h->array[(i-1)/2] = h->array[i];
        h->array[i]=temp;
        i=(i-1)/2;
    }
}


// Deletion (Top Element) : O(logn)
void deletion(maxHeap h) {
    if(h->size==0)
        cout<<"Empty Heap"<<endl;
    else {
        h->array[0]=h->array[h->size-1];
        h->size--;
        heapify(h,0);
    }
}


// Print Heap
void printHeap(maxHeap h) {
    for(int i=0; i<h->size; i++) {
        cout<<h->array[i]<<" ";
    }
    cout<<endl;
}
```

```
int main() {
    int capacity;
    cin>>capacity;
    maxHeap h=Create(capacity);
    insert(h,3);        insert(h,10);
    insert(h,7);        insert(h,9);
    print_heap(h);
    deletion(h);
    print_heap(h);
    insert(h,8);        insert(h,6);
    insert(h,45);       insert(h,45);
    print_heap(h);
    deletion(h);
    print_heap(h);
    return 0;
}
```

Problems

1. Given an array of n integers , devise an algorithm to get median of the subrray **0 to i** for all i **i.e 0 to n**.
   Example:
   Let's assume array is {5,2,3,4}
   M(i) reperesents the median of array from 0 to i
   M(0) = 5, Sorted Array: [5]
   M(1) = (2+5)/2 = 3, Sorted Array: [2, 5]
   M(2) = 3, Sorted Array: [2, 3, 5]
   M(3) = (3+4)/2 = 3, Sorted Array : [2, 3, 4, 5]

   Solution
   a. Observe that in order to find the median of a sub-array [0,i], we only require the center 2 elements of the sorted array [0,i],
   b. In other words, we need the max element from the smaller half elements of the array and the min element from the larger half elements of the array.
   c. Use max-heap to store the smaller half and min-heap to store the larger half.
   d. The 2 heaps should differ in size by at max 1 at any point of time: |size(min-heap) - size(max-heap|<=1|

2. Given an array of n elements, where *each element is at most k away from its actual position in the sorted version of the array*(K-sorted array), devise an algorithm that sorts the given K-sorted array in O(n log k) time.
   Solution
   a. Use Min-heap of size K+1, and insert the 1st K+1 elements of the given array in the heap. This heap is bound to have the smallest element of the array, as the given array is K-sorted

b. For every i (i>k+1), insert ar[i] in the min-heap, and do a deleteMin to get the next element of the sorted version of the array.

3. Given an array of integers, find K smallest elements in the array.
   Solution
   a. Create K sized max Heap using the first K elements of the array. In short, we are assuming that the first K elements are the K smallest elements in the array.
   b. For every i(i>K), do the following:
      if ar[i] < getTop(max-heap: delMax() and insert(ar[i])

4. Given k sorted arrays of size n each, merge them and print the sorted output.
   Solution
   a. Create an output array of size n*k.
   b. Create a min heap of size k and insert 1st element of all the k arrays into the heap
   c. Repeat following step n*k times.
      i. Get minimum element from heap (minimum is always at root) and store it in output array, and track the array number to which this extracted element belongs (say m).
      ii. If array m has no element left, then go back to the above step**,** otherwise insert next element from the array m in the min heap.

5. Given a row and column wise sorted matrix of size n*n, print all elements in sorted order.
   Example:

1	5	10	20
2	8	15	30
12	15	18	35
20	25	30	40

   Solution
   a. Create a min-heap with the first row of the matrix.
   b. Repeat following step n*n times.
      i. Get minimum element from heap (minimum is always at root) and print it, and track the column number to which this extracted element belongs (say m).
      ii. If column m has no element left, then go back to the above step**,** otherwise insert next element from the column m in the min heap.

Trie

Introduction

Trie is an efficient information retrieval data structure. Using trie, search complexities can be brought to optimal limit (key length). If we store keys in binary search tree, a well balanced BST will need time proportional to M * log N, where M is maximum string length and N is number of keys in tree. Using trie, we can search the key in O(M) time.

Let word be a single string and let dictionary be a large set of words. If we have a dictionary, and we need to know if a single word is present in the dictionary, trie is the data structure that can help us. But you may be asking yourself, "Why use tries if hash tables can do the same?" There are two main reasons:

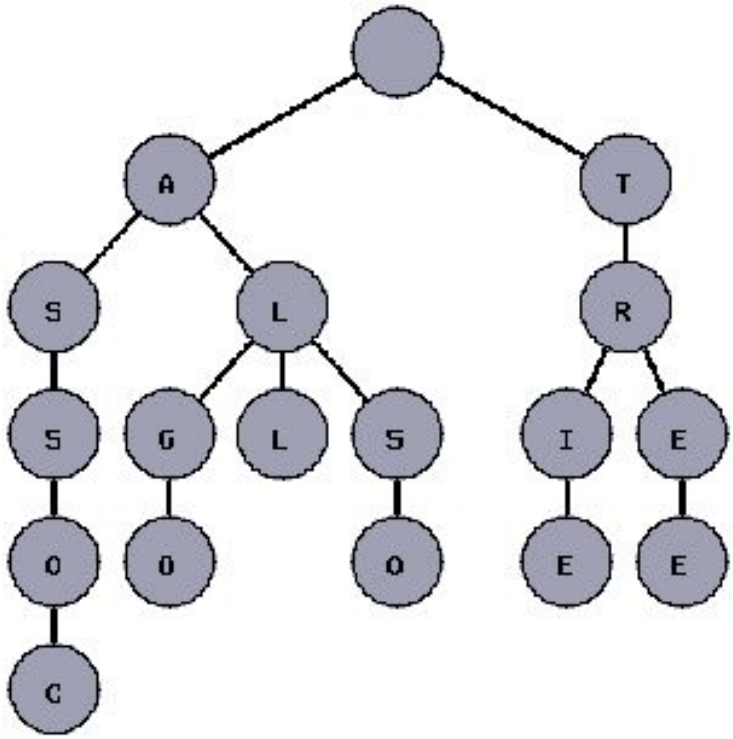
1. The tries can insert and find strings in O(L) time (where L represent the length of a single word). This is a bit faster than a hash table.
2. hash tables can only find in a dictionary words that match exactly with the single word that we are finding; the trie allow us to find words that have a single character different, a prefix in common, a character missing, etc.

What is a Trie?

You may read about how wonderful the tries are, but maybe you don't know yet what the tries are and why the tries have this name. The word trie is an infix of the word "re**trie**val" because the trie can find a single word in a dictionary with only a prefix of the word. The main idea of the trie data structure consists of the following parts:

1. The trie is a tree where each vertex represents a single word or a prefix.
2. The root represents an empty string (""), the vertexes that are direct sons of the root represent prefixes of length 1, the vertexes that are 2 edges of distance from the root represent prefixes of length 2, the vertexes that are 3 edges of distance from the root represent prefixes of length 3 and so on. In other words, a vertex that are *k* edges of distance of the root have an associated prefix of length *k*.
3. Let **v** and **w** be two vertexes of the trie, and assume that **v** is a direct father of **w**, then **v** must have an associated prefix of **w**.

The following figure shows a trie with the words “tree”, “trie”, “algo”, “assoc”, “all”, and “also.”



Note that every vertex of the tree does not store entire prefixes or entire words. The idea is that the program should remember the word that represents each vertex while lower in the tree.

Implementation of Trie

The tries can be implemented in many ways, some of them can be used to find a set of words in the dictionary where every word can be a little different than the target word, and other implementations of the tries can provide us with only words that match exactly with the target word. The implementation of the trie that will be exposed here will consist only of finding words that match exactly and counting the words that have some prefix.

Design Data Structure with following operations:

- **addWord( ) :** This function will add a single string word to the dictionary.
- **countPrefixes( ) :** This function will count the number of words in the dictionary that have a string **prefix** as **prefix**.
- **countWords( ) :** This function will count the number of words in the dictionary that match exactly with a given string **word**.

Solution using Trie Data Structure

```
// Asumming that our trie will take input in small letters [a-z]
#include<iostream>
#include<cstdio>
#include<cstring>
#include<cstdlib>
using namespace std;

// Assuming alhphabet is [a-z]
struct trienode {
    int words;//# of words
    int prefixes;//# of words having this prefix
    struct trienode * ref[26];//all possible refrences
};

// Creating Root Node
struct trienode* initialize() {
    struct trienode *p;
    p=(struct trienode*)malloc(sizeof(struct trienode));
    p->words=0;
    p->prefixes=0;
    for(int i=0; i<26; i++)
        p->ref[i]=NULL;
    return p;
}

// Adding Words
void addwords(struct trienode *root,char *word,int k,int wordlen) {
    if(k==wordlen)
        root->words++;
    else {
        int temp=word[k];
        temp-=97;
        if(root->ref[temp]==NULL) {
            root->ref[temp]=initialize();
        }
        root->ref[temp]->prefixes++;
        addwords(root->ref[temp], word, k+1, wordlen);
    }
}
```

```
// Counting Given Word
int countwords(struct trienode *root,char *word,int k,int wordlen) {
    if(k == wordlen)
        return root->words;

    int temp=word[k];
    temp-=97;
    if(root->ref[temp]==NULL)
        return 0;
    else
        return countwords(root->ref[temp], word, k+1, wordlen);
}

// Counting Words Having Same Prefix
int countprefixes(struct trienode *root,char *prefix,int k,int prefixlen) {
    if(k==prefixlen)
        return root->prefixes;

    int temp=prefix[k];
    temp-=97;
    if(root->ref[temp]==NULL)
        return 0;
    else
        return countprefixes(root->ref[temp],prefix,k+1,prefixlen);
}

int main() {
    struct trienode *root;
    root=initialize();

    addwords(root,"tree",0,4);
    addwords(root,"treek",0,5);
    addwords(root,"trie",0,4);
    addwords(root,"assoc",0,5);
    addwords(root,"all",0,3);
    addwords(root,"algo",0,4);
    addwords(root,"also",0,4);

    cout<<countprefixes(root,"tr",0,2)<<endl;
    cout<<countprefixes(root,"al",0,2)<<endl;

    cout<<countwords(root,"tree",0,4)<<endl;
    cout<<countwords(root,"also",0,4)<<endl;

    return 0;
}
```

Problems

1. Given a list of phone numbers, determine if it is consistent in the sense that no number is the prefix of another.

   Example: A:911, B:97625999, C:91125426
   In this case, it's not possible to call C, because the central would direct your call to the A as soon as you had dial the first three digits of C's phone number. So this list would not be consistent.
   Solution: Create a trie with alphabet [0-9] and insert phone numbers in the trie. During insertion check if there is already any word present in the trie which is prefix of current phone number. For identifying nodes which represent end of word, you can maintain a flag (boolean) in the node structure of the trie.

2. Given an array of integers, we have to find two elements whose XOR is maximum.
   Solution:
   a. Create trie using bit representation of all the numbers, starting with the MSB bit. Observer that, the MSB bits contributes more.
   b. Now for any element ar[i], find its correspoding pair which gives maximum xor value for this element using trie.

Greedy Algorithms

Introduction

Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step. For many optimization problems, using dynamic programming to determine the best choices is overkill; there are simpler and more efficient algorithms.
A ***greedy algorithm*** always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. Greedy algorithms do not always yield optimal solutions, but for many problems they do.

Greedy is a strategy that works well on optimization problems with the following characteristics:

1. **Greedy-choice property:** A global optimum can be arrived at by selecting a local optimum.
2. **Optimal substructure:** An optimal solution to the problem contains an optimal solution to subproblems.

An activity-selection problem
You are given n activities with start and finish times. Select the max 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:
Consider the following 6 activities.
start[ ] = {1, 3, 0, 5, 8, 5};
finish[ ] = {2, 4, 6, 7, 9, 9};
The maximum set of activities that can be executed by a single person is 4 – Activity0, Activity1, Activity3, Activity4.

Solution: Sort the activities based on finish time and pick activities in order such that they do not overlap.

Huffman Encoding

1. http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap17.htm
2. http://www.geeksforgeeks.org/greedy-algorithms-set-3-huffman-coding/

Does Greedy always work?
Since the optimal-substructure property is exploited by both greedy and dynamic-programming strategies, one might be tempted to generate a dynamic-programming solution to a problem when a greedy solution suffices, or one might mistakenly think that a greedy solution works when in fact a dynamic-programming solution is required. To illustrate the subtleties between the two techniques, let us investigate two variants of a classical optimization problem.

Knapsack Problem

1. The ***0-1 knapsack problem*** is posed as follows. A thief robbing a store finds n items; the ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. He wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack for some integer W. What items should he take? (This is called the 0-1 knapsack problem because each item must either be taken or left behind; the thief cannot take a fractional amount of an item or take an item more than once.)

2. In the ***fractional knapsack problem***, the setup is the same, but the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item. You can think of an item in the 0-1 knapsack problem as being like a gold ingot, while an item in the fractional knapsack problem is more like gold dust.

Example-1
Item1: \$6/kg, **Item2:** \$5/kg, **Item3:** \$4/kg

Maximum profit for fractional knapsack is **240\$** [10 units of Item1, 20 units of Item2, 20 units of Item3]
Maximum profit for integer knapsack is **220\$** [Item3 and Item2]

Greedy Approach: Pick elements in non-increasing order of value/kg and fill the knapsack.

For above example we can veirfy that for both the knapsack problems, we get the optimal solution using Greedy Aproach. So far, greedy seems to work for both.

Let's take another example on integer knapsack [v denotes value, w denotes weight]
Item1: v = \$28, **w** = 7
Item2: v = \$18, **w** = 6
Item3: v = \$12, **w** = 4
Knapsack size(W) = 10

Item1: \$4/kg, **Item2:** \$3/kg, **Item3:** \$3/kg

Using Greedy Approach: Maximum profit for **integer** knapsack is **\$28** [Item1]
But, the optimal solution for this example is **\$30** [Item2 and Item3]

Conclusion: Greedy works for **Fractional Knapsack** but fails for **Integer Knapsack.** **Integer Knapsack** is solved using dynamic programming, which is discussed in the next section.

Dynamic Programming

Introduction

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:

1. **Overlapping Subproblems**
2. **Optimal Substructure**

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. However, 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 finding nth Fibonacci number
  int fib(int n) {
    if ( n <= 1 )
       return n;
    return fib(n-1) + fib(n-2);
  }
```

There are following two different ways to store the values so that these values can be reused.

1. **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.

```
#include<stdio.h>
#define NIL -1
#define MAX 100

int lookup[MAX];
// Function to initialize NIL values in lookup table
void _initialize() {
   for (int 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\n", fib(n));
   return 0;
}
```

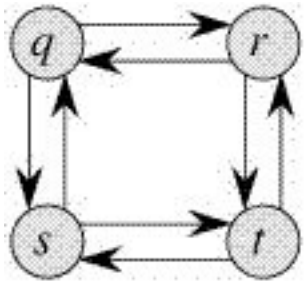
2. **Tabulation (Bottom Up):** The tabulated program for a given problem builds a table in **bottom up** fashion and returns the last entry from table.

```
#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 = 40;
   printf("Fibonacci number is %d\n",fib(n));
   return 0;
}
```

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 problem of finding the shortest path between 2 nodes in a graph 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 algorithm is typical example 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



There are two longest paths from q to t:
q -> r ->t and q ->s->t.

Unlike shortest paths, these longest paths do not have the optimal substructure property. For example, the longest path q->r->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->s->t->r.

Problems

1. **Fibonacci Series and It's Application**
    a. Count ways to reach the n'th stair if one or two steps can be taken at once
    Solution
    DP State: dp[i] = No. of ways to reach i^{th} stair
    dp[1] = 1
    dp[2] = 2
    dp[i] = dp[i-1] + dp[i-2] , 3 <= i <= n
    Ans: dp[n]

    b. Domino problem
        i. No of ways to fill 2xN sized rectangle with dominos of size 1x2[can rotate to 2x1]
        Solution
        DP State: dp[i] = No. of ways to fill 2xi with 2x1 sized dominos
        dp[1] = 1
        dp[2] = 2
        dp[i] = dp[i-1] + dp[i-2], 3 <= i <= n
        Ans: dp[n]

ii. No of ways to fill 5xN sized rectangle with dominos of size 1x5[can rotate to 5x1]

Solution

DP State: dp[i] = No. of ways to fill 5xi with 5x1 sized dominos

```
dp[1] = dp[2] = dp[3] = dp[4] = 1
dp[5] = 2
dp[i] = dp[i-1] + dp[i-5], 6 <= i <= n
```

Ans: dp[n]

2. Compute ${}^{n}C_{r}$ for given values of n and r (No. of ways to choose r items from n items)

Solution:

DP State: dp[i][j] = No. of ways to choose j items from i items

There are two possibilities:

a. **Including i^{th} item :** dp[i-1][j-1]
b. **Excluding i^{th} item :** dp[i-1][j]

So we can write dp[i][j] as following:

```
dp[0][j] = 0, 1 <= j <= r
dp[i][0] = dp[i][i] = 1, 0 <= i <= n
for 2 <= i <= n
    for 1 <= j <= min (r, i-1)
        dp[i][j] = dp[i-1][j-1] + dp[i-1][j]
```

Ans: dp[n][r]

3. **Integer Knapsack:** A thief robbing a store finds n items; the ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. He wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack for some integer W. What items should he take?

Solution

v : Array of values, v[i] : denotes the value of i^{th} item

w: Array of weights, w[i] : denotes the weight of i^{th} item

DP State: dp[i][j] = Maximum profit after processing i items with knapsack size j

```
dp[0][j] = 0, 1 <= j <= W
dp[i][0] = 0, 1 <= i <= n

for 1 <= i <= n        //assume array indexes start from 1
    for 1 <= j <= W
        if (j-w[i] <  0):
            dp[i][j] = dp[i-1][j];
        else
            dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i]] + v[i]);
```

Ans: dp[n][W]

4. **Longest Increasing Sequence:** Given an array of integers, find the length of the longest increasing subsequence.
   Solution
   DP State: dp[i] = Length of longest increasing sub-sequence with i^{th} element as the last element of the sequence

```
for 1 <= i <= n
    dp[i] = 0
    for 1 <= j < i
        if (a[j] < a[i])
            dp[i] = max(dp[i], dp[j])
    dp[i] += 1
```

   Ans: max(dp[i]: for 1 <= i <= n)

5. **Maximum contiguous subarray sum:** Given an array of integers, find the subarray with maximum sum, report the sum.
   Solution
   DP State: dp[i] = maximum sum after processing i elements

```
dp[1] = a[1]
dp[i] = max(dp[i-1] + a[i], a[i]), 2 <= i <= n
```

   Ans: max(dp[i]: for 1 <= i <= n)

6. **Maximum non-contiguous subarray sum:** Given an array of integers, find maximum sum of array elements, such that no 2 chosen elements are at adjacent position.
   Solution
   DP State: dp[i] = maximum sum after processing i elements

```
dp[1] = a[1]
dp[2] = max(a[1], a[2])
dp[i] = max(dp[i-1], dp[i-2] + a[i], a[i]), 3 <= i <= n
```

   Ans: max(dp[i]: for 1 <= i <= n)

7. Given a 2D maze where each cell has A_{ij} number of apples, find the maximum number of apples you can collect on your path to reach (n,n), if you start from (0,0), and you can move only right or down, ie, from (i,j) you can go to (i+1,j) or (i,j+1).
   Solution
   DP State: dp[i][j] = #maximum number of apples you can collect on path to (i,j) starting from (0,0)

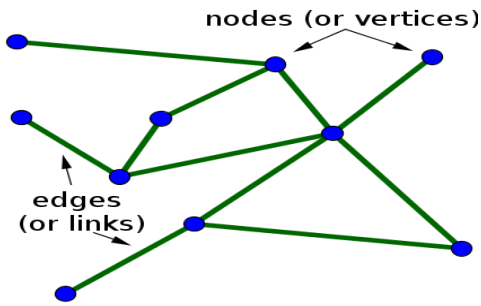
```
dp[0][0] = A[0][0]
dp[0][j] = dp[0][j-1] + A[0][j], 1<=j<=n
dp[i][0] = dp[i-1][0] + A[i][0], 1<=i<=n
for 1 <= i <= n
    for 1 <= j <= n
        dp[i][j] = max(dp[i-1][j], dp[i][j-1]) + A[i][j]
```

   Ans: dp[n][n]

Graph Theory

Definition of Graph

Informally, a graph is a diagram consisting of points, called vertices, joined together by lines, called edges; each edge joins exactly two vertices.



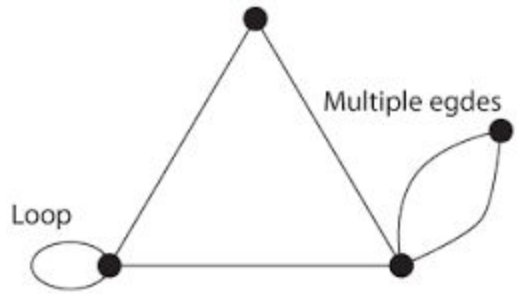
A graph G = (V, E) consists of a (finite) set denoted by V, or by V(G) if one wishes to make clear which graph is under consideration, and a collection E, or E(G), of unordered pairs {u, v} of distinct elements from V. Each element of V is called a vertex or a point or a node, and each element of E is called an edge or a line or a link.

Formally, a graph G is an ordered pair of disjoint sets (V, E), where $E \subseteq V \times V$. Set V is called the vertex or node set, while set E is the edge set of graph G. Typically, it is assumed that self-loops (i.e. edges of the form (u, u), for some $u \in V$) are not contained in a graph.

Directed and Undirected Graph

A graph G = (V, E) is directed if the edge set is composed of ordered vertex (node) pairs. A graph is undirected if the edge set is composed of unordered vertex pair.

Loop and Multiple Edges



A loop is an edge whose endpoints are equal i.e., an edge joining a vertex to itself is called a loop. We say that the graph has multiple edges if in the graph $\exists$ two or more edges joining the same pair of vertices.

Simple Graph

A graph with no loops or multiple edges is called a simple graph. We specify a simple graph by its set of vertices and set of edges, treating the edge set as a set of unordered pairs of vertices and write e = uv (or e = vu) for an edge e with endpoints u and v.
When u and v are endpoints of an edge, they are adjacent and are neighbors.

Connected Graph

A graph that is in one piece is said to be connected, whereas one which splits into several pieces is disconnected.

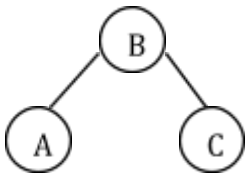
A graph G is connected if there is a path in G between any given pair of vertices, otherwise it is disconnected. Every disconnected graph can be split up into a number of connected subgraphs, called components. A **forest** is a graph with each connected component a tree.

Cyclic and Acyclic Graphs

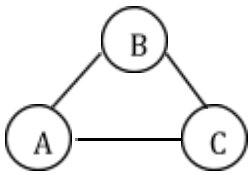
Cyclic, as the name suggests contains cycles, and the opposite for an acyclic graph.

Directed and Undirected Graphs

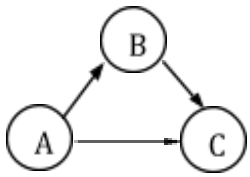
In a directed graph, edges are unidirectional. If 2 nodes (say A and B) are connected by a directed edge, then one can traverse from A to B only and not from B to A using this edge. In an undirected graph, all edges are bidirectional.



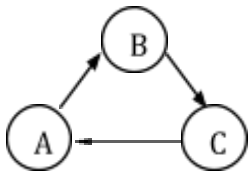
Undirected Acyclic



Undirected Cyclic



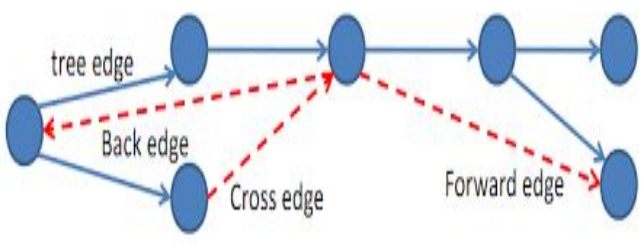
Directed Acyclic Graph (DAG)



Directed Cyclic

DFS Edge Classification

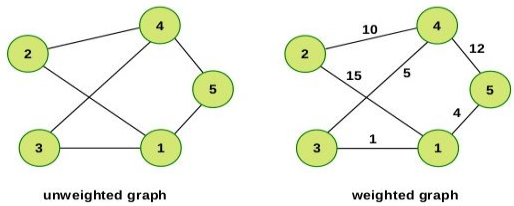
The edges we traverse as we execute a depth-first search can be classified into four edge types. During a DFS execution, the classification of edge (u, v), the edge from vertex u to vertex v, depends on whether we have visited v before in the DFS and if so, the relationship between u and v.



1. If v is visited for the first time as we traverse the edge (u, v), then the edge is a **tree edge**.
2. Else, v has already been visited:
    a. If v is an ancestor of u, then edge (u, v) is a **back edge**.
    b. Else, if v is a descendant of u, then edge (u, v) is a **forward edge**.
    c. Else, if v is neither an ancestor or descendant of u, then edge (u, v) is a **cross edge**.

After executing DFS on graph G, every edge in G can be classified as one of these four edge types. We can use edge type information to learn some things about G. For example, tree edges form trees containing each vertex DFS visited in G. ***Also, G has a cycle if and only if DFS finds at least one back edge***.

Weighted and Unweighted Graphs



Weighted graph: edges have a weight

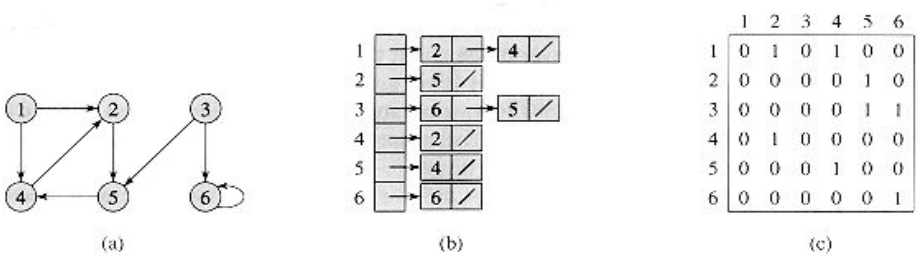
Unweighted graph: edges have no weight

Representing Graphs

To represent a graph, all we need is the set of vertices, and for each vertex the neighbors of the vertex (vertices directly connected to it by an edge) and if its a weighted graph, the weight associated with each edge. There are different ways to optimally represent a graph, depending on the density of its edges. A Sparse graph has relatively less number of edges m = O(n), where as a dense graph has significantly many edges m = $O(n^2)$, where n is the number of vertices in the graph.

1. **Adjacency List:** In this representation, for each node in the graph we maintain the list of its neighbors. We have an array of vertices, indexed by the vertex number and for each vertex *v*, the corresponding array element points to a singly-linked list of neighbors of *v*.

2. **Adjacency Matrix:** In this representation, we construct a *nxn* matrix A. If there is an edge from a vertex *i* to vertex *j*, then the corresponding element of A, $a_{i,j} = 1$, otherwise $a_{i,j} = 0$. Note, even if the graph on 100 vertices contains only 1 edge, we still have to create a 100x100 matrix with lots of zeroes. For a weighted graph, instead of 1s and 0s, we can store the weight of the edge. In that case, to indicate that there is no edge we can put a safely large value (we can call it INF (infinity)).



Graph Traversals

So far we have learned how to represent our graph in memory, but now we need to start doing something with this information. There are two methods for searching graphs that are extremely prevalent, and will form the foundations for more advanced algorithms later on. These two methods are the **Depth First Search** and the **Breadth First Search**.

Depth First Search

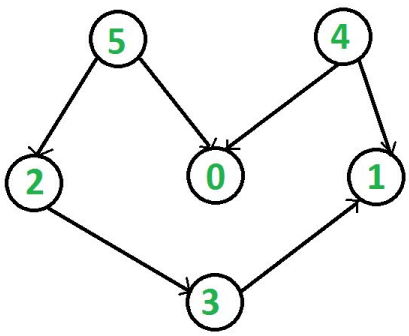
The Depth First search is well geared towards problems where we want to find any solution to the problem (not necessarily the shortest path), or to visit all of the nodes in the graph. The basic concept is to visit a node, then push all of the nodes to be visited onto the **stack**. To find the next node to visit we simply pop a node of the stack, and then push all the nodes connected to that one onto the stack as well and we continue doing this until all nodes are visited. It is a key property of the Depth First search that we not visit the same node more than once, otherwise it is quite possible that we will recurse infinitely. We do this by marking the node as we visit it.

Breadth First Search

The Breadth First search is an extremely useful searching technique. It differs from the depth-first search in that it uses a **queue** to perform the search, so the order in which the nodes are visited is quite different. It has the extremely useful property that if all of the edges in a graph are unweighted (or the same weight) then the first time a node is visited is the shortest path to that node from the source node. You can verify this by thinking about what using a queue means to the search order. When we visit a node and add all the neighbors into the queue, then pop the next thing off of the queue, we will get the neighbors of the first node as the first elements in the queue. This comes about naturally from the **FIFO** property of the queue and ends up being an extremely useful property. One thing that we have to be careful about in a Breadth First search is that we do not want to visit the same node twice, or we will lose the property that when a node is visited it is the quickest path to that node from the source. We do this by marking the node as we visit it.

Topological Sort

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 in-coming edges).

The usual algorithms for topological sorting have running time linear in the number of nodes plus the number of edges, asymptotically, O(|V| + |E|)

1. **Topological Sort using Kahn's Algorithm:** One of these algorithms works by choosing vertices in the same order as the eventual topological sort. First, find a list of "start nodes" which have no incoming edges(ie, indegree=0) and insert them into a set S; at least one such node must exist in a non-empty acyclic graph. Then:

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edges
while S is non-empty do
   remove a node n from S
   add n to tail of L
   for each node m with an edge e from n to m do
      remove edge e from the graph
      if m has no other incoming edges, ie, indegree(m)=0 then
         insert m into S

if graph has edges then
   return error (graph has at least one cycle)  //used to detect cycle in a directed graph
else
   return L (a topologically sorted order)
```

2. **Topological Sort using Depth First Search:** An alternative algorithm for topological sorting is based on depth-first search. The algorithm loops through each node of the graph, in an arbitrary order, initiating a depth-first search that terminates when it hits any node that has already been visited since the beginning of the topological sort or the node has no outgoing edges (i.e. a leaf node):

```
L ← Empty list that will contain the sorted nodes
while there are unmarked nodes do
   select an unmarked node n
   visit(n)

function visit(node n)
   if n has a temporary mark then stop (not a DAG)  //detect cycle in a directed graph
      if n is not marked (i.e. has not been visited yet) then
         mark n temporary
         for each node m with an edge from n to m do
            visit(m)

         mark n permanently
         unmark n temporarily
         add n to head of L
```

Each node *n* gets prepended to the output list L only after considering all other nodes which depend on *n* (all descendants of *n* in the graph). Specifically, when the algorithm adds node *n*, we are guaranteed that all nodes which depend on *n* are already in the output list L: they were added to L either by the recursive call to visit() which ended before the call to visit *n*, or by a call to visit() which started even before the call to visit *n*. Since each edge and node is visited once, the algorithm runs in linear time.

Shortest Path Algorithms

An extremely common problem on graphs is to find the path from one position to another. In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.

The problem of finding the shortest path between two intersections on a road map (the graph's vertices correspond to intersections and the edges correspond to road segments, each weighted by the length of its road segment) may be modeled by a special case of the shortest path problem in graphs.

There are a few different ways for going about this, each of which has different uses. We will discuss few of them here.

Single-source shortest paths
Given a graph and a source vertex **S**, find the shortest path to all other vertices.

1. **Unweighted Graph -** Simply use **Breadth First Search**.
   Algorithm:
   a. Store **S** in a queue and set the distance to **s** to be 0 in a distance array, d.
   b. Initialize the distances to all other vertices as "not computed", say -1.
   c. While there are vertices in the queue:
      Read a vertex **v** from the queue
      For all edges vertices **v->w**:
         If distance to **w** is -1 (not computed) do:
            Make distance to **w** equal to d[**v**] + 1, ie, d[w] = d[v] + 1
            Make path to **w** equal to **v**, ie, path[w] = v //Store path from **S** to w **
            Append **w** to the queue

   Complexity: O(|E| + |V|)

   ** To store the path from node A to node B, we only need to store the previous node and we can print the path by traversing on the path array from B to A (*path*[*u*] holds the predecessor of *u* in the shortest path from *s* to *u*)

2. **Weighted Graph – Dijkstra's algorithm** is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks.

   Algorithm: Use priority queue based on the distance of current node from S.
   Let the node at which we are starting be called the **source node(S)**. Let d[w] be the distance from the **S** to w. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.
   a. Store **S** in a priority queue with distance as 0. Set d[S] = 0.
   b. Initialize the distances to all other vertices as "not computed", sentinel value infinity.

c. While there are vertices in the queue:
   DeleteMin a vertex v from the queue
   For all adjacent vertices w do:
      if d[v] + weight(v,w) < d[w]:
         d[w] = d[v] + weight(v,w); // update distance to w
         insert w in the priority queue with distance d[w]
         Make path to w equal to v, ie, path[w] = v //Store path from S to w **

   Complexity: O(|E|log|V| + |V|log|V|) = O((|E| + |V|)log(|V|))

   Note: Dijkstra's Algorithm can be used for both weighted Directed and Undirected Graphs. However, for a weighted DAG, topological ordering can also be used to quickly compute shortest paths.
   Ref:
   https://en.wikipedia.org/wiki/Topological_sorting#Application_to_shortest_path_finding
   http://www.geeksforgeeks.org/shortest-path-for-directed-acyclic-graphs/

3. **All-pairs shortest path:** Find the shortest paths between every pair of vertices u,v in the graph. **Floyd–Warshall** is an algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles). A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between all pairs of vertices.

   Consider a graph G with vertices V numbered 1 through N. Further consider a function shortestPath(i, j, k) that returns the shortest possible path from i to j using vertices only from the set {1,2,...,k} as intermediate points along the way. Now, given this function, our goal is to find the shortest path from each i to each j using only vertices 1 to k + 1.
   For each of these pairs of vertices, the true shortest path could be either:
   a. A path that only uses vertices in the set {1, ..., k}, or
   b. A path that goes from i to k + 1 and then from k + 1 to j.

   We know that the best path from i to j that only uses vertices 1 through k is defined by shortestPath(i, j, k), and it is clear that if there were a better path from i to k + 1 to j, then the length of this path would be the concatenation of the shortest path from i to k + 1 (using vertices in {1, ..., k}) and the shortest path from {k + 1} to j (also using vertices in {1, ..., k}).

   If w(i, j) is the weight of the edge between vertices i and j, we can define shortestPath(i, j, k + 1) in terms of the following recursive formula:
   shortestPath(i,j,k+1) = min(shortestPath(i,j,k), shortestPath(i,k+1,k) + shortestPath(k+1,j,k))
   Base case: shortestPath(i, j, 0) = w(i, j)
   This formula is the heart of the Floyd–Warshall algorithm. The algorithm works by first computing shortestPath(i, j, k) for all (i, j) pairs for k = 1, then k = 2, etc. This process continues until k = N, and we have found the shortest path for all (i, j) pairs using any intermediate vertices.

Pseudo code

a. let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
b. for each vertex v:
    dist[v][v] ← 0
c. for each edge (u,v):
    dist[u][v] ← w(u,v) // the weight of the edge (u,v)
d. for k from 1 to |V|:
    for i from 1 to |V|:
        for j from 1 to |V|:
            if dist[i][j] > dist[i][k] + dist[k][j]:
                dist[i][j] ← dist[i][k] + dist[k][j]

Complexity: $O(n^3)$

Codes:

1. **Graph Representation using Adjacency matrix.**

```
int n,m;    //vertices and edges
cin>>n>>m;

int graph[n+1][n+1];      //vertices are generally numbered starting with 1
int u,v,w;    // w is the weight of the edge
for(int i=0; i<m; i++) {
   cin>>u>>v>>w;
   ar[u][v] = w;          // assign with 1 if the graph is unweighted
   ar[v][u] = w;          //if the graph is bidirectional
}
```

2. **Graph Representation using Adjacency list.**

```
int n,m;    //vertices and edges
cin>>n>>m;

// can use vector<vector< int> > graph(n+1) if the graph is unweighted
vector<vector<pair<int,int> > > graph(n+1);
int u,v,w;    // w is the weight of the edge
for(int i=0; i<m; i++) {
   cin>>u>>v>>w;
   graph[u].push_back(make_pair(v,w));    // graph[u].push_back(v) - unweighted
   graph[v].push_back(make_pair(u,w));   // if the graph is bidirectional
}
```

3. **Depth First Search traversal of a graph.**

```
void dfs(int u, bool vis[], vector<vector<int> > graph) {
   if(vis[u])   return;
   cout<<u<<" ";
   vis[u] = true;
   for(int i=0; i<graph[u].size(); i++)
      dfs(graph[u][i], vis, graph);
}
```

Main:

```
   bool vis[n+1];
   fill(vis, vis+n+1, false);    // #include<algorithm>
   for(int i=1; i<=n; i++)
      if(!vis[i]) {
         dfs(i, vis, graph);
         cout<<endl;
   }
```

4. **Breadth First Search traversal of a graph**

```
void bfs(int u, bool vis[], vector<vector<int> > graph) {
   queue<int> q;
   q.push(u);
   vis[u] = true;

   while(!q.empty()) {
      u = q.front();
      q.pop();
      cout<<u<<" ";
      for(int i=0; i<graph[u].size(); i++)
         if(!vis[graph[u][i]]) {
            q.push(graph[u][i]);
            vis[graph[u][i]] = true;
         }
   }
}
```

Main:

```
   bool vis[n+1];
   fill(vis, vis+n+1, false);     // #include<algorithm>
   for(int i=1; i<=n; i++)
      if(!vis[i]) {
         bfs(i, vis, graph);
         cout<<endl;
      }
```

5. Topological Sorting using Depth First Search Traversal

```
bool topoSortUsingDFS(int u, int mark[], vector<vector<int> > graph, stack<int>
&topologicalSorting) {
   if(mark[u] == 1)              //there is a cycle in the graph
      return false;
   if(mark[u] == 2)
      return true;

   mark[u] = 1;
   bool retValue = true;
   for(int i=0; i<graph[u].size() && retValue; i++)
      retValue &= topoSortUsingDFS(graph[u][i], mark, graph, topologicalSorting);

   topologicalSorting.push(u);
   mark[u] = 2;
   return retValue;
}
```

Main:

```
   int mark[n+1];
   fill(mark, mark+n+1, 0);
   stack<int> topologicalSorting;
   bool noCycle = true;

   for(int i=1; i<=n && noCycle; i++)
      if(mark[i] == 0)
         noCycle = topoSortUsingDFS(i, mark, graph, topologicalSorting);

   if(noCycle) {
      while(!topologicalSorting.empty()) {
         cout<<topologicalSorting.top()<<" ";
         topologicalSorting.pop();
      }
      cout<<endl;
   } else
      cout<<"Given graph is not a DAG, topological sorting failed!!"<<endl;
```

Here,
mark[u] = 0: unvisited
mark[u] = 1: in current DFS cycle and not completely processed
mark[u] = 2: processed

6. Dijkstra's Algorithm – Shortest path between 2 nodes in a weighted graph

```
int Dijkstra(int source, int destination, vector<vector<pair<int, int> > > graph) {
    int n = graph.size();
    int distance[n];
    fill(distance, distance + n, INT_MAX);
    distance[source] = 0;

    //Declaring a min heap of pair – (distance, nodeNumber)
    //Heap property will be maintained using the 1st element of the pair - Distance
    priority_queue< pair<int, int>, vector<pair<int, int> >, greater<pair<int, int> > >
pq;
    pq.push(make_pair(0, source));

    pair<int, int> p;
    int u, d, v, w;

    while(!pq.empty()) {
        p = pq.top();
        pq.pop();
        if(p.second == destination)
            return p.first;

        u = p.second;
        d = p.first;

        if(distance[u] == d) {
            for(int i=0; i<graph[u].size(); i++) {
                v = graph[u][i].first;
                w = graph[u][i].second;
                if(d + w < distance[v]) {
                    distance[v] = d+w;
                    pq.push(make_pair(distance[v], v));
                }
            }
        }
    }

    return distance[destination];
}
```

```
int main() {
    int n,m;       //vertices and edges
    cin>>n>>m;

    vector<vector<pair<int, int> > > graph(n+1);
    int u,v,w;
    for(int i=0; i<m; i++) {
        cin>>u>>v>>w;
        graph[u].push_back(make_pair(v,w));
        graph[v].push_back(make_pair(u,w));
    }

    cout<<Dijkstra(1, 7, graph)<<endl;
    cout<<Dijkstra(3, 9, graph)<<endl;
    return 0;
}
```

7. Floyd Warshall – All pair shortest path

```
void FloydWarshall(vector<vector<int> > &graph) {
    int n = graph.size() - 1;
    for(int k=1; k<=n; k++)
        for(int i=1; i<=n; i++)
            for(int j=1; j<=n; j++)
                graph[i][j] = min(graph[i][j], graph[i][k] + graph[k][j]);
}

int main() {
    int n,m;       //vertices and edges
    cin>>n>>m;

    vector<vector<int> > graph(n+1, vector<int> (n+1, 1e6));
    int u,v,w;
    for(int i=0; i<m; i++) {
        cin>>u>>v>>w;
        graph[u][v] = w;
        graph[v][u] = w;
    }

    FloydWarshall(graph);
    cout<<graph[1][7]<<endl;
    cout<<graph[3][6]<<endl;
    return 0;

}
```

Problems

1. Check if there exists a path between 2 given nodes A and B of a graph.
   Solution: Simply run BFS/DFS from A, if you reach B -> path exists

2. Detect Cycle in a Directed Graph.
   Solution
   a. Using Topo Sort – if Topo Sort fails, cycle exists
   b. Using Back Edge – if Back Edge exists, cycle exists
      i. Using Start/Finish Time
      ii. Using mark array. = {0, 1, 2} – code explained above

3. Detect Cycle in an Undirected Graph.
   Observation: You cannot use the methods explained for a directed graph above, because edges in an undirected graph are bidirectional and any edge will be marked as a back edge if we use Back Edge detection mechanism.

   Solution
   a. Using Kahn's Topological Sorting Algorithm
   b. Use edge marking + DFS Back Edge Detection – Basically, if you encounter the same node again in your current DFS traversal though an unmarked edge, that means this is a back edge and cycle exists

4. Given an undirected graph, check if it's a tree.
   Solution: A tree can be defined as a **connected acyclic** graph. For the graph to be **acyclic**, given n nodes, there should not be more than n-1 edges in the graph. After the first verification step, for the graph to be a tree, you can simply run a BFS/DFS to check if the graph is **connected**.
   Related problem: http://www.spoj.com/problems/PT07Y/

5. Given a matrix of 0s and 1s, find the number of islands considering N4 neighborhood. A group of connected 1s forms an island. For example, the below matrix contains 5 islands:
   {1, 1, 0, 0, 0},
   {0, 1, 0, 0, 1},
   {1, 0, 0, 1, 1},
   {0, 0, 0, 0, 0},
   {1, 0, 1, 0, 1}
   Solution: Run BFS/DFS [considering N4 neighborhood] for every unvisited cell of the matrix which contains a 1.

6. Given an unweighted, undirected tree, find the length of the longest path in that tree. The length of a path is the number of edges in that path.
   Solution: Use 2 BFS. At the end of the 1st BFS, you will reach an edge node of the longest path, say S. Start another BFS from S, the length of the longest path will be the number of BFS iterations as you started from an edge node of the longest path.
   Related problems: http://www.spoj.com/problems/GCPC11J/, http://www.spoj.com/problems/PT07Z/

7. There are N C programming library files. The compiling of a library has the constraint that a library must be compiled after any library it depends on. Given the list of dependencies, find the valid compilation order of the library files, if possible.
   Solution: Represent the dependencies as a directed graph, and apply Topological Sorting. If successful, compile the files in reverse order of topological sorting, else print Impossible.
   Related Problem:
   http://www.geeksforgeeks.org/given-sorted-dictionary-find-precedence-characters/

8. Given 2 nodes A and B in an unweighted graph, find the length of the shortest path between the given nodes.
   Solution: Use **BFS**: Start BFS from A, the number of BFS iterations to reach B is the length of the shortest path between A and B.

9. Given a dictionary, and multiple queries of the form:
   'start' word and 'target' word(both of same length), find the length of the smallest chain from 'start' to 'target', such that adjacent words in the chain only differ by one character and each word in the chain is a valid word i.e., it exists in the dictionary. It may be assumed that both 'start' and 'target' words exists in dictionary and length of all dictionary words is same.

   Solution: Convert the dictionary of words into an undirected graph, where an edge between 2 words can be created iff the words only differ by one character. Now, for each query, your problem has reduced to finding the shortest path between 'start' and 'target' words – Use BFS
   Related Problem: http://www.spoj.com/problems/PPATH/

10. Given a matrix each cell can have values 0, 1 or 2 which has the following meaning: 0: Empty cell, 1: Cell has fresh orange, 2: Cell has rotten orange. You have to determine the minimum time required so that all the oranges become rotten. A rotten orange at index [i,j] can rot other fresh orange at indexes [i-1,j], [i+1,j], [i,j-1], [i,j+1] (N4 neighborhood - up, down, left and right) in 1 unit of time. If it is impossible to rot every orange then simply return -1.

    Solution: Start a BFS with start nodes as all the cells having a rotten orange. Using N4 neighborhood, continue the BFS traversal and the units of time needed to rot all the oranges is the number of BFS iterations. If after completion of BFS, there is still a fresh orange left in the matrix, return -1.

11. Shortest Path Problems:
    http://www.spoj.com/problems/MICEMAZE/
    http://www.spoj.com/problems/SHPATH/
    http://www.spoj.com/problems/HIGHWAYS/
    https://www.codechef.com/problems/HOMDEL

C++ Quick Reference

Data Structures

- vector - http://www.cplusplus.com/reference/vector/vector/
- stack - http://www.cplusplus.com/reference/stack/stack/
- queue - http://www.cplusplus.com/reference/queue/queue
- priority queue - http://www.cplusplus.com/reference/queue/priority_queue/
- deque - http://www.cplusplus.com/reference/deque/deque/
- set - http://www.cplusplus.com/reference/set/set/
- map - http://www.cplusplus.com/reference/map/map/
- pair - http://www.cplusplus.com/reference/utility/pair/

Algorithms

- fill - http://www.cplusplus.com/reference/algorithm/fill/
- sort - http://www.cplusplus.com/reference/algorithm/sort/
- Binary search -
    - binary_search - http://www.cplusplus.com/reference/algorithm/binary_search/
    - lower_bound - http://www.cplusplus.com/reference/algorithm/lower_bound/
    - upper_bound - http://www.cplusplus.com/reference/algorithm/upper_bound/
- Others -
    - http://www.cplusplus.com/reference/algorithm/
    - https://www.topcoder.com/community/data-science/data-science-tutorials/power-up-c-with-the-standard-template-library-part-1/
    - https://www.topcoder.com/community/data-science/data-science-tutorials/power-up-c-with-the-standard-template-library-part-2/

Online Resourses

Tutorials

1. https://www.topcoder.com/community/data-science/data-science-tutorials/
2. https://www.quora.com/Which-sites-are-best-for-learning-algorithms-and-data-structures-over-video

Online Judges

1. http://www.geeksforgeeks.org
2. https://www.codechef.com/contests
3. https://www.topcoder.com/community/events/
4. http://www.spoj.com/problems/classical/
5. http://codeforces.com/contests/
6. https://www.hackerearth.com/challenges/