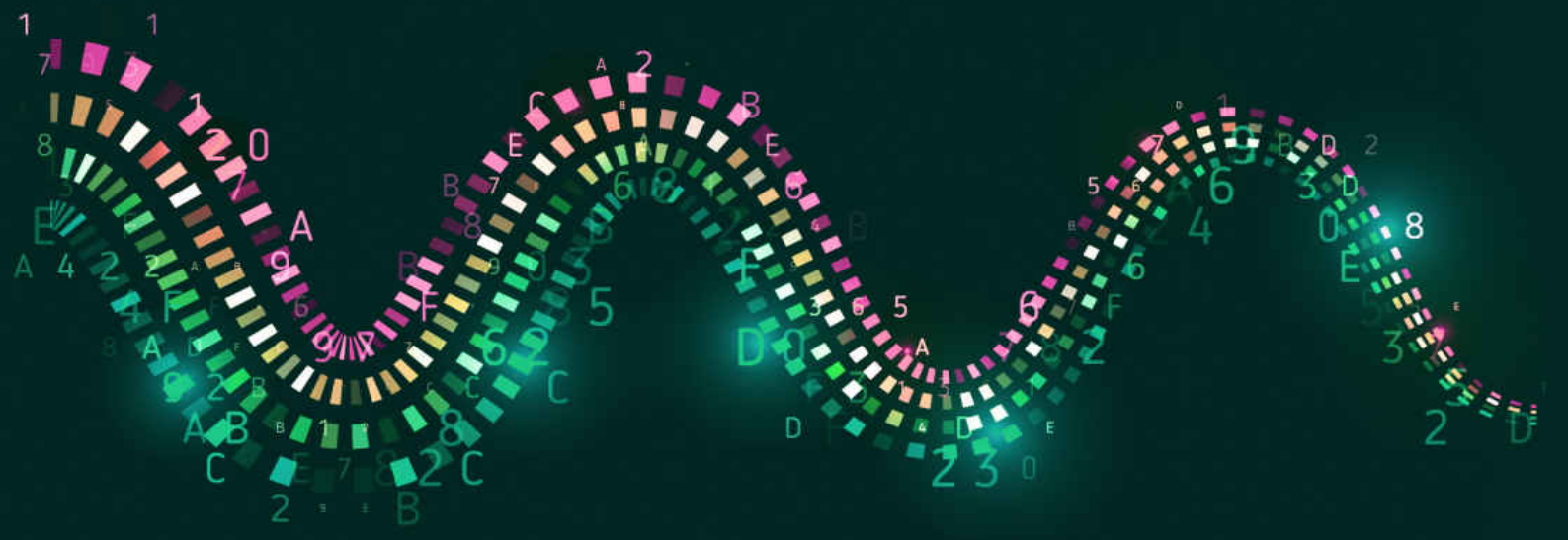
ALGORITHMS



DESIGN ALGORITHMS TO SOLVE COMMON PROBLEMS

ANDY VICKLER

Algorithms

Design Algorithms to Solve Common Problems

Table of Contents

Introduction

Algorithms are sequences of steps used to help solve specific problems that perform some kind of calculation, some data processing, and even automated reasoning. They are used in computer science, data science, and information technology and are one of the most efficient methods that can be expressed in finite space and time.

They are often the best way of efficiently representing a specific problem's solution, and they can be implemented in any programming language – they are independent of any specific language, so, while we have used the C++ language for our examples, the algorithms can be implemented in any language you desire.

Algorithm Design

Algorithm design is important, but the most critical aspect is creating the right algorithm to solve a problem efficiently using minimum space and time. There are a lot of different approaches to solving problems, some efficient in terms of time consumption, others more efficient in terms of memory. It isn't possible to create an algorithm that optimizes memory use and time consumption at the same time. If your algorithm needs to be more time-efficient, it will require more memory while, if an algorithm needs to be more memory efficient, it needs more time to run.

You are undoubtedly familiar with the many algorithms used in the real world today, including graphs, sort, and search algorithms. Perhaps the best-known and the most used are the search and sort algorithms, and these are definitely the best place to start as you begin your journey into the design of algorithms in data structures. An example of one of the most sophisticated search algorithm designs is the Google Search Engine Algorithm. Used by Google, it ranks every web page in its search results based on relevancy.

Over the years, the methods we use to find data have changed significantly. Another example of a popular algorithm,

especially with social media users, is the hashtag algorithm. Hash tagging has one of the most complicated learning curves, but it is fair to say that hashing is incredibly fast at searching through massive lists containing millions of items.

That's just a couple of algorithms where design is an important factor. If you are looking to advance your knowledge or want a career as a software engineer, learning algorithm design in data structures is one of the best starting places.

This guide is designed for those who already have a basic understanding and knowledge of mathematics and programming. You should understand data structures and basic algorithms as this book will dive deeper into design theory and some of the more complex algorithms.

Chapter 1: Designing an Algorithm

Have you used the Google search engine? Stupid question, of course you have! But has it ever occurred to you to question why nearly everyone uses Google when there are many other search engines you can use to find what you want? Is it because it's quicker? Looks nicer? Is it better at searching?

The answer could lie in something that happened a few years ago when search engines competed to be the top one. Google put itself there by separating itself from the others. How? They implemented a special algorithm.

Google's search engine developers, Larry Page and Sergei Brin, came up with a streamlined, efficient way of finding information on the internet. Their algorithm was called PageRank, and they used it to give users accurate and relevant web pages as a result of their searches.

PageRank was akin to being a kind of popularity contest, ranking webpages based on the number of other webpages that kinked to it. It counts how many links to go to a page and considers the quality of those links to get a rough determination of the website's importance. It was assumed that the more links a page got from other pages, the more important the website was.

Let's say that you want to find information on Borzoi dogs, and your search term brings up a thousand pages. PageRank works out which of those pages has the most links to it from other pages, with the idea being that a good website would have loads of links to it.

The idea behind PageRank isn't complicated, but the mathematics can be, delving into eigenvector centrality, row stochastic matrix, and other head-scratching things. The point is mathematics and algorithms are the central points of every piece of software and hardware you use today.

PageRank put Google ahead of all the other search engines, and that is why it is the number one search engine today. It is also what the internet giant can base its current success on.

But what is an algorithm?

One definition states that an algorithm is a " *sequence of steps resulting in a solution when applied to a problem."* Another definition states that it is " *not just a sequence of steps, but one that controls a computation model or abstract machine without requiring human judgment."*

We can illustrate this by creating an algorithm to bake a cake.

We could have an instruction that says, " *if the cake looks cooked and is a little spongy, take it out of the oven."*

The problem with this is that human judgment is required to see if the cake looks done. There is no way a computer could work this out so, perhaps a better instruction would be " *if the cake's internal temperature = 210°F, remove cake from the oven."*

Some of the biggest characteristics in an algorithm are:

- Each step is small and isolated from the others
- Each step occurs in the right sequence
- Each step can be automated – no human judgment is required, and a computer can do the job.

Algorithms are a major component of computer science. The larger the program you create, the more complex the data and processing. Provided you can learn to design algorithms and program them, provided you put the time and work into expanding your knowledge, there won't be any limit to what you can create.

Designing an Algorithm

The easiest way of understanding the components of an algorithm is to go through a problem and design the solution.

The Problem – express a specified number of seconds in hours, minutes, and seconds

Example – a friend told you they took 9331 seconds to drive from one location to another. We need a program that will output 9331 seconds as the number of hours, minutes, and seconds.

These are the steps needed to develop the program:

1. **Top-down design is best**

Top-down design outlines all the steps needed for the input, processing, and output.

Use Top-down Design:

The top-down design will outline the steps involved in terms of input, processing, and output. Our input is 9331 – the number of seconds. Processing requires that to be converted into the number of hours, then minutes, and, lastly, seconds, represented by the number.

2. **The steps should be numbered in the order of execution**

This is important. Not only does it help you to see how everything will work, but it also means the entire operation will work smoothly, as it is meant to.

3. **Create the pseudocode**

Pseudocode is short-form writing combining the English language with your chosen computer programming language. There are two important things to consider with pseudocode:

- It should be readable and understandable by someone who has no experience in computer programming
- It should be translatable into any computer programming language, which means not using terms specific to a particular one.

Our specific problem would have pseudocode like this:

- Obtain the number of seconds
- Calculate hours
- Calculate minutes
- Calculate seconds
- Output hours, minutes, seconds

Now, that's okay, but it doesn't explain how the calculations are being done. This can be refined to something more precise:

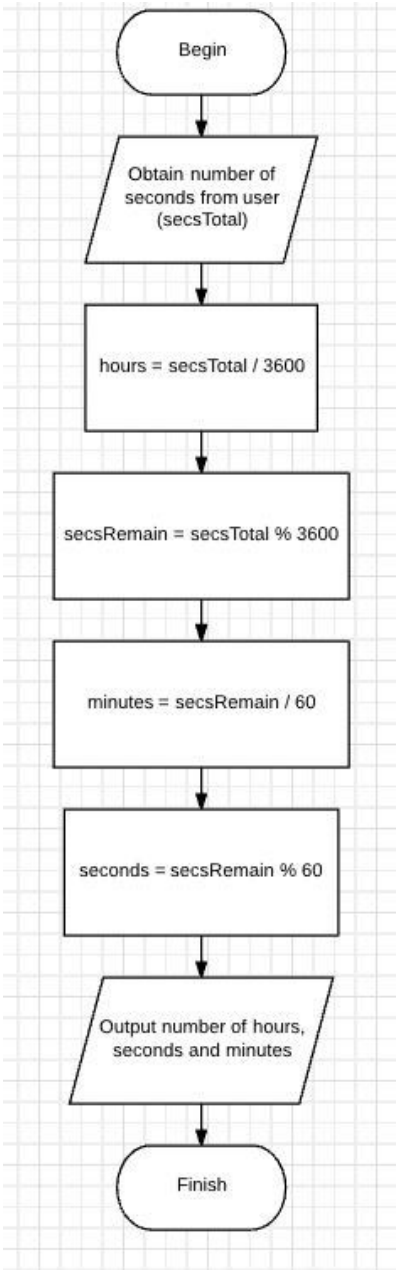
- Obtain the number of seconds

- Calculate hours – total seconds / 3600
- Calculate remaining seconds – total seconds % 3600
- Calculate minutes – remaining seconds / 60
- Calculate seconds – remaining seconds % 60
- Output hours, minutes, seconds

That looks much better and is easily understandable by anyone who reads it.

4. **Create a flowchart**

Once your algorithm has been created, a flowchart is needed, where small code snippets will be used, and it looks something like this:



5. **Write your program**

This part should be easy:

```
int secsTotal, secsRemain, hours, minutes, seconds;

System.out.print ("Please enter the total number of seconds: "); //prompt user for total seconds

hours = secsTotal / 3600;  // determine number of hours

secsRemain – secsTotal % 3600;  // determine remaining seconds
```

```
minutes = secsRemain / 60;  // determine number of minutes

seconds – secsRemain % 60;  // determine number of seconds

System.out.println(“”);

System.out.println(secsTotal + “ seconds is equivalent to “); // output house, minutes, and seconds

System.out.println(hours + “hour(s) – “ + minutes + “minute(s) – “ + seconds + “ second(s).”_;
```

Below you can see a few more computer science problems you need to understand and the algorithms associated with them. Factorials, prime numbers, and the Fibonacci sequence are all important in algorithm design.

Determining Factorials

Factorials are the product of multiplication when every number is multiplied together from 1 to a specified number. It is written as n! Here’s an example showing the factorial of 6:

6! = 6 x 5 x 4 x 3 x 2 x 1
6! = 720

Try these:

1. What is the factorial of 5?

Answer:

5! = 5 x 4 x 3 x 2 x 1
5! = 120

2. What is the factorial of 8?

Answer:

8! = 8 x 7 x 6 x 5 x 4 x 3 x 2 x 1

8! = 40320

3. What is the factorial of 12?

Answer:

12! = 12 x 11 x 10 x 9 x 8 x 7 x 6 x 5 x 4 x 3 x 2 x 1
12! = 479001600

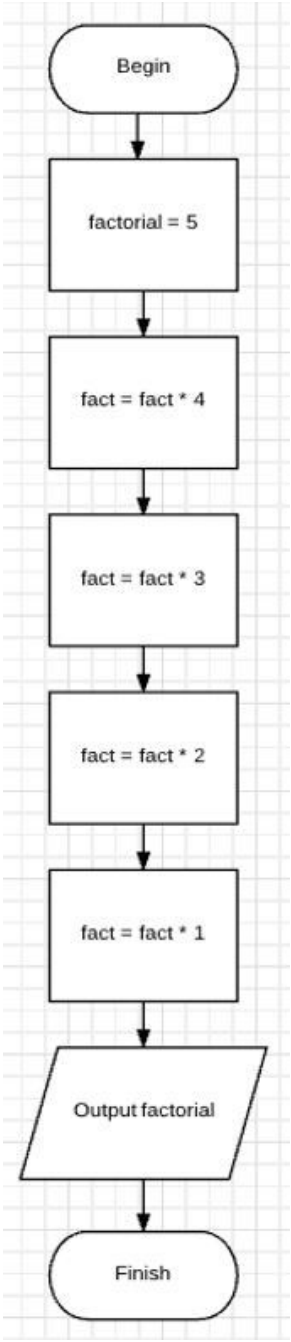
Think of a program you could write where a factorial needs to be calculated. Notice that, while the numbers repeat, they also decrease by a factor of 1 each time. Also, notice that when you start from 1, they increase by 1 each time until the specified number is given. Could you program this efficiently?

Before you think about writing your program, it would probably be best to design your flowchart first, showing how the factorial for one number is calculated. Then you can determine how to generalize it, so it works with any number.

Let's go back to our first factorial problem, 5! One way you could solve it is this:

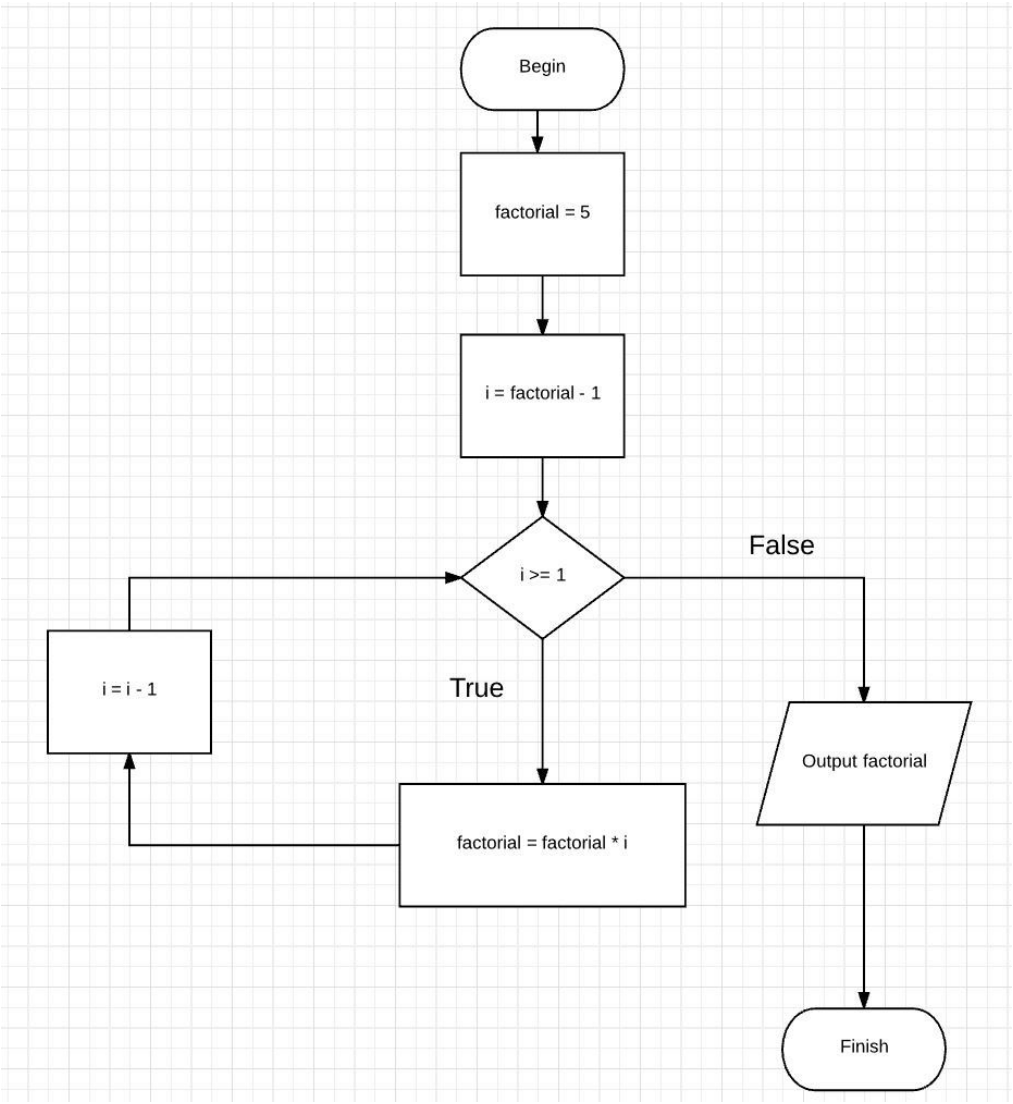
1. As you want to find 5!, that's the number to remember for the first calculation
2. 5 x 4 = 20. 20 is the number to remember for the next calculation
3. 20 x 3 = 60. 60 is the number to remember for the next calculation
4. 60 x 2 = 120. 120 is the number to remember for the next calculation
5. 120 x 1 = 120. That's it; you're finished!

The flowchart could look something like this:



You might have spotted no small amount of repetition in this flowchart, especially in the equations. The stored values are multiplied by one less every time. This is important because, in computer programming, when you repeat a step multiple times, decreasing by 1 each time, it's known as a for loop.

Rewriting that flowchart using a for loop would look like this:



Prime Numbers

As you should know from your high school Math classes, prime numbers can only be divided by themselves and by 1. A number is only divisible by another number when it can be divided with no remainders.

For example, 12 is divisible by 6 as 12 % 6 = 0. However, 23 is not divisible by 6, as 23 % 6 = 5.

Prime numbers are considered important in computer science because they encrypt data and can generate random numbers.

Let's say you needed to determine if 13 is a prime number. Although you know it is, your brain might do this:

Is 13 divisible by 2? No.
Is 13 divisible by 3? No.

Is 13 divisible by 4? No.
Is 13 divisible by 5? No.
Is 13 divisible by 6? No.
Is 13 divisible by 7? No.
Is 13 divisible by 8? No.
Is 13 divisible by 9? No.
Is 13 divisible by 10? No.
Is 13 divisible by 11? No.
Is 13 divisible by 12? No.

So, 13 is a prime number.

Although repetitive, you can use this process to write a program that determines if a number is prime. You may have spotted that some of those calculations are not really necessary – after all, you don't need to divide 13 by 7, 8, or any other number up to 12. The way to work out if a number is prime is to divide it by the numbers that go up to half its value.

Fibonacci Sequence

The Fibonacci sequence begins with 0 and 1:

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 …

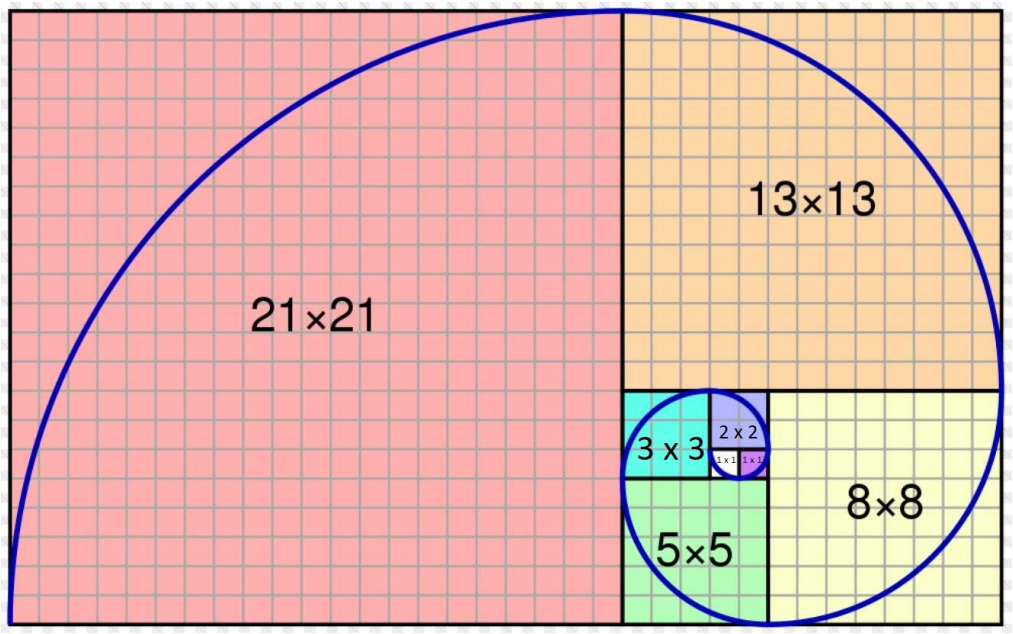
The next number will always be the total of the previous two numbers added together, as indicated below:

0 1 1 2 **3 5** 8 13 21 **34 55** 89 144 **233 277** 610

0+1=1 3+5=8 34+55=89 233+377=610

When you write the Fibonacci sequence as a rule, it is $X_{n-1} + X_{n-2}$

As you can see from the diagram below, the Fibonacci sequence may also be drawn like a spiral:



This sequence is clearly a progressive pattern because each number except the first two, is found by adding the previous two numbers.

One algorithm that could be used to output this sequence is a for loop and some variables that hold the following information:

- The number you want – X_n
- The number before it – X_{n-1}
- The number before that – X_{n-2}

Algorithm Design Techniques

These are some of the more popular algorithm design approaches and are the ones we will dedicate the rest of this book to:

1. **Divide and Conquer** – a top-down approach and algorithms following this technique contain three steps:
    a. The original problem is divided into subproblems
    b. Every subproblem is solved individually and recursively
    c. The subproblem solutions are combined into the solution for the entire problem
2. **Greedy Technique** – this solves the optimization problem, which involves a set of input values that need to

be minimized or maximized – constraints or conditions.
    a. The greedy algorithms used greedy criteria to choose the best solution for optimizing a specified object at any given moment
    b. The optimal solution is not always guaranteed, but the solution produced is typically close to the optimal value.
3. **Dynamic Programming –** a bottom-up approach that solves smaller problems before combining them to get the solution for a larger problem. This is helpful when you have a larger number of subproblems.
4. **Branch and Bound –** when the algorithm is given a subproblem that cannot be bound, it divides it into a minimum of two restricted subproblems. These algorithms are global optimization methods used in non-convex problems. They can be a little slow in the worst case because the effort required grows with the size of the problem.
5. **Randomized Algorithms –** these algorithms are defined with permission to access sources of unbiased, independent random bits. Those bits then influence the algorithm's computation.
6. **Backtracking Algorithm –** this algorithm will try every possibility in order until they reach the correct one. It searches for the solution depth-first and, where an alternative doesn't work, they backtrack to where other alternatives were presented and try again.

With all that said, let's move on to the first design technique – divide and conquer.

Chapter 2: Divide and Conquer

This chapter will discuss the Divide and Conquer technique and some of the algorithms under it, along with how the approach helps solve problems.

We can split this technique into three subparts:

1. **Divide –** the problem is divided into smaller problems
2. **Conquer –** the smaller problems are solved by recursive calling
3. **Combine –** the solutions for the smaller problems are combined to get the solution for the whole problem.

These are the standard algorithms used under Divide and Conquer:

Quicksort

Quicksort is a DAC algorithm that chooses a specific element as its pivot and then partitions an array around it. There are several quicksort versions that each use a different method to choose their pivot.:

- The first element is always picked as the pivot
- The last element is always picked as the pivot – demonstrated below
- A random element is picked as the pivot
- The median is picked as the pivot

Recursive Quicksort Function Pseudocode

```
/* low  —> Starting index,  high  —> Ending index */
Quicksort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);
```

```
        Quicksort(arr, low, pi - 1);  // Before pi
        Quicksort(arr, pi + 1, high); // After pi
    }
}
```

The Partition Algorithm

Partitioning can be done using several methods. The pseudocode below uses simple logic. Starting from the leftmost element, we track the indices of equal to or smaller elements as i. If a smaller element is found while traversing, the current element is swapped with arr[i]. Otherwise, the current element is ignored:

```
/* low  —> Starting index,  high  —> Ending index */
Quicksort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        Quicksort(arr, low, pi - 1);  // Before pi
        Quicksort(arr, pi + 1, high); // After pi
    }
}
```

Pseudocode for partition()

```
/* the last element is chosen as the pivot and placed
at its right position in the sorted array. all the
elements smaller than the pivot are placed
```

```
to the left of the pivot and the bigger elements on the right. */
partition (arr[], low, high)
{
    // pivot (the element placed at the correct position in the array)
    pivot = arr[high];

     i = (low - 1)  // The smaller element's index, indicating the pivot's correct position so far

    for (j = low; j <= high- 1; j++)
    {
       // If the current element is smaller than the pivot
       if (arr[j] < pivot)
       {
                  i++;      // the smaller element's index is incremented
           swap arr[i] and arr[j]
       }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

partition() illustrated:

arr[] = {10, 80, 30, 90, 40, 50, 70}

Indexes: 0 1 2 3 4 5 6

low = 0, high = 6, pivot = arr[h] = 70

Initialize the smaller element's index, **i = -1**

Traverse the elements from j = low to high-1

j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 0

arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j

// are same

j = 1 : Since arr[j] > pivot, don't do anything

// No change in i and arr[]

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 1

arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

j = 3 : Since arr[j] > pivot, don't do anything

// No change in i and arr[]

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 2

arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped

j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]

i = 3

arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

Because j is equal to high-1, we exited the loop.

Lastly, we swap arr[i+1] with arr[high] to put the pivot in its correct position:

arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

70 is now in the right place and all the smaller elements are placed in front of it, and the bigger elements after it.

Implementation

Below, you can see the Quicksort implementations in C++:

```
/* C++ implementation of QuickSort */
#include <bits/stdc++.h>
using namespace std;

// Utility function that swaps two elements
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* the last element is chosen as the pivot and placed
at its right position in the sorted array. all the
elements smaller than the pivot are placed
to the left of the pivot and the bigger elements on the right. */
int partition (int arr[], int low, int high)
```

```
{
    int pivot = arr[high]; // pivot
     int i = (low - 1); // The smaller element's index, indicating the pivot's correct position so far


    for (int j = low; j <= high - 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
                    i++; // the smaller element's index is incremented
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

/* The main function implementing QuickSort
arr[] —> Array to be sorted,
low —> Starting index,
high —> Ending index */
void Quicksort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
```

```
        at right place */
        int pi = partition(arr, low, high);

        // sort the elements separately before
        // and after partition
        Quicksort(arr, low, pi - 1);
        Quicksort(arr, pi + 1, high);
    }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver Code
int main()
{
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    Quicksort(arr, 0, n - 1);
    cout << "Sorted array: \n";
    printArray(arr, n);
```

```
    return 0;
}
```

Output

```
Sorted array:
1 5 7 8 9 10
```

Quicksort Analysis

We can write the time complexity of Quicksort as:

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

The first two terms indicate recursive calls, while the last indicates the partition process. The number of elements is indicated by k, and these are smaller than the pivot. The time complexity is entirely dependent on the partition strategy used and the input array. Below are details of three different cases:

Worst Case

This happens when the partition chooses the smallest or greatest element as the pivot. Going back to the partition strategy where the pivot is always the last element, the worst case would happen when the array has already been sorted into order, either descending or ascending. Here is the recurrence for the worst case:

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

This is the equivalent of:

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

And the solution is:

$$(n^2)$$

Best Case

This occurs when the middle element is always chosen as the pivot. Here is the best case recurrence:

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

The solution is:

(n Log n)

Average Case

When doing an average case analysis, all possible array permutations must be considered, and we must also calculate each permutation's time, which isn't all that easy. However, we get a rough idea of the average case by considering when O(n/9) elements are placed into one set and O(9n/10) elements in another by the partition process. Here is the recurrence:

T(n) = T(n/9) + T(9n/10) + (n)

And the solution is:

O(n Log n)

Quicksort's worst-case time complexity is $O(n^2)$, which is better than some of the other sorting algorithms, such as Heapsort and Mergesort. In practice, Quicksort is much faster because it has an inner loop that can be implemented efficiently on most architectures and real-world data. The pivot choice can be changed, so there are several ways Quicksort can be implemented. That way, the worst-case scenario will rarely happen for specified data types. However, where we deal with huge amounts of data stored externally, Mergesort is considered a better option.

Quicksort is considered an in-place sorting algorithm because extra space is only used to store recursive function calls and not manipulate the input.

3-Way Quicksort

A standard Quicksort algorithm chooses an element as the pivot, and the array is partitioned around it. We then recur for the subarrays to the left and right of the pivot.

Let's consider something different, an array with lots of redundant elements. For example:

{1, 4, 2, 4, 2, 4, 1, 2, 4, 1, 2, 2, 2, 2, 4, 1, 4, 4, 4}

If a simple Quicksort chose 4 as the pivot, only one instance of 4 is fixed, and the other recurrences are processed recursively. With 3-way Quicksort, we divide the array arr[l..r] into three:

1. arr[l..i] elements less than the pivot
2. arr[i+1…j-1] elements equal to the pivot
3. arr[j…r] elements greater than the pivot.

Quicksort vs. Mergesort

Quicksort is an in-place sorting algorithm, which means it doesn't need any extra storage space. On the other hand, Mergesort needs O9n) additional storage, where n indicates the array size, which can be quite expensive. Because the extra time has to be allocated and de-allocated, the algorithm's running time increases. When we compare the time complexity of both algorithms, we can see that they have an average complexity of O(n Log n) but different constants. In an array, Mergesort isn't so good because of the additional space needed.

Most practical Quicksort implementations use the randomized version with time complexity of O(n Log n). This version can also have a worst-case, but it doesn't tend to occur in patterns like the sorted array. In practice, randomized Quicksort works well and is a cache-friendly algorithm with a good locality of reference when used on arrays.

Things are different where the linked list is concerned, though. This is down to there being quite a difference in the memory allocation for linked lists and arrays. In an array, modes may be adjacent in memory, but this is not the case in linked lists. Conversely, items can be inserted into the middle of a linked list in O91) time and O(1) extra space, but we can't do this in an array. As such, Mergesort operations don't need extra space when implemented in a linked list.

Random access is possible in an array because the elements are continuous in memory. Let's assume we have a 40byte, integer array A with A[o]s address being x. Accessing A[i]

requires that the memory at (x + i*4) is directly accessed. Random access is not possible in a linked list. Accessing the ith degree in a linked list require traversing every node between the head and the ith node, because there is no continuous memory block. Therefore, Quicksort has a higher overhead.

Mergesort

The Mergesort algorithm also comes under the DAC technique and is one of the best for building recursive algorithms.

This technique splits a problem into two separate segments and solves them individually. Once we have a solution for each segment, we merge them to provide a solution for the whole problem.

Let's say we have an array A, and we want to sort the subsection. This starts at index p and goes to index r, represented by A[p…r].

- **Divide –** assume q to be the central point between p and r and split the subarray A[p..r] into two – A[p…q] and A[q+1, r]
- **Conquer –** once the array has been split, it's time to conquer. Both subarrays are individually sorted. If the base condition is not reached, the same procedure is followed, i.e., the subarrays are split down and sorted individually.
- **Combine –** when this step reaches the base condition, the sorted subarrays are merged into one sorted subarray.

Mergesort Algorithm

Mergesort will continue to split an array into subarrays until a specific condition is met, and Mergesort can be performed on a subarray of size 1, for example, p == r. Then the sorted subarrays are combined into increasingly larger arrays until the total array has been merged. Here's the algorithm:

If p<r

Then q → (p+ r)/2

MERGE-SORT (A, p, q)

MERGE-SORT (A, q+1,r)

MERGE (A, p, q, r)

MergeSort(A, o, length(A)-1) is called to sort the array. The algorithm continues dividing the array recursively until it meets the base condition and the array only contains a single element. The merge function will then merge those sorted subarrays to sort the whole array.

FUNCTIONS: MERGE (A, p, q, r)

n 1 = q-p+1

n 2= r-q

create the arrays [1…..n 1 + 1] and R [1…..n 2 +1]

for i ← 1 to n 1

do [i] ← A [p+ i-1]

for j ← 1 to n2

do R[j] ← A[q + j]

L [n 1+ 1] ← ∞

R[n 2+ 1] ← ∞

I ← 1

J ← 1

For k ← p to r

Do if L [i] ≤ R[j]

then A[k] ← L[i]

i ← i +1

else A[k] ← R[j]

j ← j+1

The Merge Step

Recursive algorithms depend on a base case and its merging ability on the results from the base cases. Mergesort is no exception; it's just that the merge step has far more importance.

This step is a solution that combines the sorted subarrays for any given problem into one large array. The algorithm maintains three pointers for each subarray and one for the final array's current index:

Did you get to the end of the array?

No:

Start by comparing both array's current elements

Then the smaller element is copied to the sorted arrays

Lastly, the pointer for the element with the smaller element is moved.

Yes:

Copy the remaining elements from the non-empty array

Step-By-Step – the merge() Function

Take the example below of an unsorted array. We are going to use the Mergesort algorithm to sort it:

A= (36,25,40,2,7,80,15)

Here's how we do it:

Step One – the algorithm divides the array in half iteratively until an atomic value is achieved. If the array contains an odd number of elements, one half will have more elements.

Step Two – once the array has been divided in half, you can see the order of elements in the original array was not changed. Now we can divide each subarray in half.

Step Three – we continue dividing the subarrays until an atomic value is achieved. This is a value that cannot be divided any further.

Step Four – next, the subarrays are merged. This must be done in the same way they were divided.

Step Five – the element on each list is compared and then combined into a single sorted list.

Step Six – the next iteration compares the lists containing the two data values and merges them into a single list of data values. These values are in sorted order.

And the array is fully sorted.

Mergesort Analysis

Let's say T(n) is the time the algorithm takes.

To sort the two divided halves will take 2T time at the most.

When the sorted lists are merged, we get a total comparison of n-1. This is because the last remaining element must be copied in the combined list, and there is no comparison.

As such, the relational formula is:

$$T(n) = 2T\frac{n}{2} + n - 1$$

However, the -1 is ignored because it takes a while to copy the element into the merge lists.

- **Best-Case –** for the already sorted array, the best-case time complexity is O(n*Log n)
- **Average Case –** the mergesort algorithm has an average case time complexity of O(n* Log n). This happens when at least 2 elements are jumbled, i.e., not in descending or ascending order
- **Worst-Case –** worst-case time complexity is O(n* Log n), occurring when the array's descending order is sorted into ascending order

The mergesort algorithm also has a space complexity is O(n).

Closest Pair of Points

Let's say we have an array of n points in the plane. We want to work out a problem that often occurs in applications, finding the closest pair of points. Take air traffic control, for example. You may want to monitor which planes are flying too close to one another because this could be an indication of a potential collision.

The formula for the distance between two points, p and q, is:

$$\|pq\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

O(n^2) is the brute force solution – the distance is computed between each pair, and the smallest is returned. Using the Divide and Conquer strategy, O(n Log n) time calculates the smallest distance.

Next, we'll look at another approach, O(n x (Log n)^2).

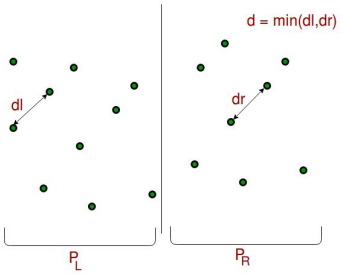
The Algorithm

Below, you can see the steps required for an O(n x (Log n)^2) algorithm.

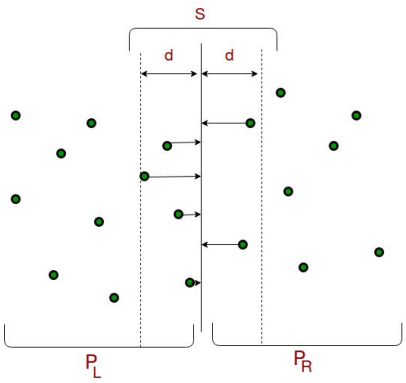
- **Input –** an array containing n points P[]
- **Output –** the smallest distance between two of the given array's points

In terms of pre-processing, we sort the input array per the x coordinates:

1. First, the array's middle point must be found. We'll take the middle point as P[n/2]
2. The array is divided in half. The first has points from P[o] to P[n/2], and the second has points from P[nm/2+1] and P[n-1].
3. The smallest distance is found recursively in each subarray. We'll say the distances are dl and dr, so we need the minimum of dl and dr – for this, the minimum is d.



4. From these three steps, we have obtained an upper bound d, which is the minimum distance. Now the pairs must be considered so that a point in the pair comes from the left subarray and one is from the right subarray. Consider the vertical line in the image below – this runs through P[n/2], so you need to find every point with an x coordinate nearer to that line than d. Then you could build an array strip[] of all these points:



5. The array strip is sorted per the y coordinates with the time complexity of O(n Log n). However, we can optimize this to O(n) by sorting and merging recursively.
6. Now we need to find the smallest distance in strip[]. This won't be easy - at first glance, it would appear to be O(m^2), but it is really O(n). We can geometrically prove that, for each point in the array, we only have to check a maximum of 7 points after it, assuming that the array has been sorted per the y coordinate.
7. Lastly, the minim of d is returned, along with the distance calculated in step 6.

Implementation

Below is this algorithm implemented in C++:

```
// A divide and conquer program in C++
```

```
// to find the smallest distance from a
// specified set of points.

#include <bits/stdc++.h>
using namespace std;

// A structure representing a Point in the 2D plane
class Point
{
    public:
    int x, y;
};

/* The two functions below are required for the library
function qsort().*/

// Required to sort the array of points
// per the X coordinate
int compareX(const void* a, const void* b)
{
    Point *p1 = (Point *)a, *p2 = (Point *)b;
    return (p1->x - p2->x);
}

// Required to sort the array of points per the Y
coordinate
int compareY(const void* a, const void* b)
{
```

```
    Point *p1 = (Point *)a, *p2 = (Point *)b;
    return (p1->y - p2->y);
}

// A utility function that finds the
// distance between two points
float dist(Point p1, Point p2)
{
    return sqrt( (p1.x - p2.x)*(p1.x - p2.x) +
                 (p1.y - p2.y)*(p1.y - p2.y)
               );
}

// A Brute Force method that returns the
// smallest distance between two points
// in P[] of size n
float bruteForce(Point P[], int n)
{
    float min = FLT_MAX;
    for (int i = 0; i < n; ++i)
        for (int j = i+1; j < n; ++j)
            if (dist(P[i], P[j]) < min)
                min = dist(P[i], P[j]);
    return min;
}

// A utility function that finds the
```

```
// minimum of two float values
float min(float x, float y)
{
    return (x < y)? x : y;
}



// A utility function that finds the
// distance between the closest points of
// strip of a specified size. All points in
// strip[] are sorted per the
// y coordinate. They all have an upper
// bound d on the minimum distance.
// Note that this method appears to be
// a O(n^2) method, but it's actually an O(n)
// method because the inner loop runs no more than 6 times
float stripClosest(Point strip[], int size, float d)
{
    float min = d; // Initializes the minimum distance as d


    qsort(strip, size, sizeof(Point), compareY);


     // Pick all the points one by one and try the next points until the difference
    // between the y coordinates is smaller than d.
     // It is geometrically proven that this loop runs at most 6 times
```

```
    for (int i = 0; i < size; ++i)
        for (int j = i+1; j < size && (strip[j].y - strip[i].y) <
min; ++j)
            if (dist(strip[i],strip[j]) < min)
                min = dist(strip[i], strip[j]);

    return min;
}

// A recursive function that finds the
// smallest distance. The array P contains
// all the points sorted per the x coordinate
float closestUtil(Point P[], int n)
{
    // If there are 2 or 3 points, use brute force
    if (n <= 3)
        return bruteForce(P, n);

    // Find the middle point
    int mid = n/2;
    Point midPoint = P[mid];

    // Consider the vertical line that passes
    // through the middle point and calculate
    // the smallest distance dl on the left
    // of middle point and dr on the right side
    float dl = closestUtil(P, mid);
```

```
    float dr = closestUtil(P + mid, n - mid);

    // Find the smaller of two distances
    float d = min(dl, dr);

    // Build an array strip[] containing the
    // points close (closer than d)
    // to the line passing through the middle point
    Point strip[n];
    int j = 0;
    for (int i = 0; i < n; i++)
        if (abs(P[i].x - midPoint.x) < d)
            strip[j] = P[i], j++;

    // Find the closest points in strip.
    // Return the minimum of d and closest
    // distance is strip[]
    return min(d, stripClosest(strip, j, d) );
}

// The main function to find the smallest distance
// This method primarily uses closestUtil()
float closest(Point P[], int n)
{
    qsort(P, n, sizeof(Point), compareX);

    // Use recursive function closestUtil()
```

```
    // to find the smallest distance
    return closestUtil(P, n);
}

// Driver code
int main()
{
    Point P[] = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12,
10}, {3, 4}};
    int n = sizeof(P) / sizeof(P[0]);
    cout << "The smallest distance is " << closest(P, n);
    return 0;
}
```

Output:

```
The smallest distance is 1.414214
```

Time Complexity

Let's say that this algorithm's time complexity will be T(n) and assume that the sorting algorithm is O(n Log n). The algorithm above divides the points into two sets and calls recursively for both. Once the division has been done, the strip is found in O(n) time and sorted in O(n Log n) time. The closest points are found in O(n) time.

So, we can express T(n) as:

$T(n) = 2T(n/2) + O(n) + O(n \text{ Log } n) + O(n)$
$T(n) = 2T(n/2) + O(n \text{ Log } n)$
$T(n) = T(n \text{ x Log } n \text{ x Log } n)$

Notes

1. If we optimize the fifth step in the above algorithm, we can improve the time complexity to O(n Log n).

2. The code will find the smallest distance and can be modified to find all points with the smallest distance.
3. The Quicksort algorithm is used, which can have a worst-case of O(n ^2). You can also use O(Log n) sorting algorithms such as Heapsort or Mergesort to provide an upper bound of O(n (Log n)^2).

What Doesn't Qualify as Divide and Conquer?

Binary Search is not a divide and conquer algorithm. Instead, it is a sorting algorithm where, in each step, the input element x is compared with the array's middle element value. If the values match, the middle index is returned. If x is smaller than the middle element, the algorithm will recur for the middle element's left side. Otherwise, it recurs for the right side.

While many people believe that this is a divide and conquer example, it isn't because each step only has one subproblem, while divide and conquer requires at least two. As such, Binary Search falls under Decrease and Conquer.

Here's the Divide and Conquer algorithm:

Divide and Conquer algorithm:

```
DAC(a, i, j)
{
   if(small(a, i, j))
     return(Solution(a, i, j))
   else
     m = divide(a, i, j)              // f1(n)
     b = DAC(a, i, mid)                 // T(n/2)
     c = DAC(a, mid+1, j)             // T(n/2)
     d = combine(b, c)                 // f2(n)
  return(d)
}
```

Below, you can see the recurrence relation for the divide and conquer algorithm:

O(1) if n is small

T(n) = f1(n) + 2T(n/2) + f2(n)

For example:

You want to find a specified array's minimum and maximum element:

Input: { 70, 250, 50, 80, 140, 12, 14 }

Output: The minimum number in a given array is: 12

The maximum number in a given array is: 250

The Approach

Finding the minimum and maximum elements from a specified array is a divide and conquer application, and there are three steps – divide, conquer, and combine.

For the Maximum:

The recursive approach is used to find the maximum. There will only be two elements on the left, and it will be easy to use a condition to find the maximum, for example, if(a([index]>a[index+1]. The program's a[index] and a[index+1] conditions ensure there are just two elements on the left.

```
if(index >= l-2)
{
if(a[index]>a[index+1])
{
// (a[index]
// We can now say that the last element will be the
maximum in a specified array.
}
else
{
//(a[index+1]
// We can now say that the last element will be the
```

```
maximum in a specified array.
}
}
```

The left side condition has been checked to find the maximum in this condition. Next, we check the right side condition for the same thing, and the recursive function to do this on the array's current index is:

```
max = DAC_Max(a, index+1, l);
// Recursive call
```

Next, the condition is compared, and the right side is checked at the current index's right side.

We will implement this logic in our program to check the right side condition:

```
// The right element will be the maximum.
if(a[index]>max)
return a[index];
// max will be the maximum element in a specified array.
else
return max;
}
```

For the Minimum:

Here, we want to find the minimum number in a specified array, and our approach will be recursive.

```
int DAC_Min(int a[], int index, int l)
//A recursive call function to find the minimum number in a specified array.
if(index >= l-2)
// to check the condition to see if there will be two elements in the left
We can then find the minimum element easily in a specified array.
{
// here we check the condition
if(a[index]<a[index+1])
```

return a[index];
else
return a[index+1];
}

Next, we check the right-side condition in a specified array:

// A recursive call for the right side in the specified array.
min = DAC_Min(a, index+1, l);

Now, the condition is checked to find the right side's minimum:

// The right element will be the minimum
if(a[index]<min)
return a[index];
// Here, min will be the minimum in a specified array.
else
return *min;*

// C++ code to demonstrate Divide and

// Conquer Algorithm#include<iostream>

include<iostream>

using namespace std;

// function to find the maximum no.

// in a given array.

int DAC_Max(int arr[], int index, int l)

{

int max;

if(index >= l - 2)

{

if(arr[index] > arr[index + 1])

return arr[index];

```
        else
          return arr[index + 1];
    }
    max = DAC_Max(arr, index + 1, l);
    if(arr[index] > max)
      return arr[index];
    else
      return max;
}

// Function to find the minimum no.
// in a given array
int DAC_Min(int arr[], int index, int l)
{
    int min;
    if(index >= l - 2)
    {
        if(arr[index] < arr[index + 1])
          return arr[index];
        else
          return arr[index + 1];
    }

    min = DAC_Min(arr, index + 1, l);
   if(arr[index] < min)
     return arr[index];
    else
```

```
        return min;
    }

    // Driver code
    int main()
    {
        int arr[] = {120, 34, 54, 32, 58, 11, 90};
        int n = sizeof(arr) / sizeof(arr[0]);
        int max, min;
        max = DAC_Max(arr, 0, n);
        min = DAC_Min(arr, 0, n);
        cout << “Maximum: ” << max << endl;
        cout << “Minimum: ” << min << endl;
        return 0;
    }
```

Output

Maximum: 120

Minimum: 11

Advantages of Divide and Conquer

- The divide and conquer technique successfully solves some of the biggest problems, such as the popular mathematical puzzle, the Tower of Hanoi. Trying to solve complicated problems when you don’t have any real idea where to start is difficult, but the divide and conquer approach reduces the effort by dividing the problem in two and working on each piece, and recursively solving it. This is a much faster algorithm than many others.
- It uses cache memory efficiently without taking up too much space. This is because simple subproblems can be

solved in the cache memory rather than having to access the main memory, which is much slower.

- It is far more proficient than the brute force technique.
- Because divide and conquer algorithms prohibit parallelism, no modification is needed, and systems can handle it with parallel processing.

Disadvantages of Divide and Conquer

- Because most divide and conquer algorithms incorporate recursion in their design, high memory management is required.
- Explicit stacks may overuse space
- The system may crash if rigorous recursion is performed greater than the current CPU stack.

In the next chapter, we look at greedy algorithms

Chapter 3: Greedy Algorithms

Algorithm design does not provide us with a "silver bullet" cure for every problem. The technique you use depends entirely on the type of problem and, while there may be several options in some cases, there will always be a best technique. A good programmer will know which technique to use, and greedy algorithms are just one of those.

What Is a Greedy Algorithm?

First off, a greedy algorithm is a technique rather than an algorithm. As the name indicates, it will make its choice based on what appears to be the best one at the time. This means its choice will always be logically optimal, with the hope that it will result in a globally optimal solution.

So, how does the technique determine an optimal choice?

Let's say we have an objective function, and it needs optimizing at a specific point, i.e., it needs to be maximized or minimized. A greedy algorithm will make sure the objective function has been optimized by making a greedy choice at every step. This technique only gets one go at computing the solution, so it cannot go back and change its decision.

Like all techniques, greedy algorithms have their advantages and disadvantages:

1. Greedy algorithms are pretty easy to find for a problem, and you may even find multiple ones that work
2. It is typically easier to analyze runtime for greedy algorithms than for others. For example, in the Divide and Conquer technique, it isn't always clear if the time is slow or fast because it gets smaller at each recursion level, while the subproblems increase in number.
3. The hardest part with greedy algorithms is that it requires more work to understand the correctness problems. Even when you have the correct algorithm, it isn't always easy to prove why it is right. It is actually more art than

science to prove a greedy algorithm is correct, and you need a whole lot of creativity.

As an aside, it's worth noting that very few greedy algorithms are actually correct, but we'll cover that later on.

Creating a Greedy Algorithm

Let's say that you are incredibly busy and you only have T time to do certain things. Obviously, you want to do as many of those things as you can in that time.

You have an array A containing integers. Each element indicates the completion time for a certain thing, and you want to know the maximum number of things you can do in the specified time.

This is one of the simpler greedy algorithm problems. During every iteration, you need to greedily choose what will take the minimum time to finish while continuously maintaining two variables – numberOfThings and currentTime. To do this calculation, you need to follow these steps:

1. First, array A must be sorted in non-decreasing order.
2. Second, each t-do item is selected, one at a time
3. Third, you must add the time it takes to do each item into the currentTime variable
4. Lastly, one is added to the numberOfThings variable

This should be repeated while currentTime is equal to or less than T.

A = {5, 3, 4, 2, 1} and T = 6

After sorting into a non-decreasing order:

A = {1, 2, 3, 4, 5}

After the first iteration:

currentTime = 1

numberOfThings = 1

After the second iteration:

currentTime is 1 + 2 = 3

numberOfThings = 2

After the third iteration:

currentTime is 3 + 3 = 6

numberOfThings = 3

After the fourth iteration:

currentTime is 6 + 4 = 10,

Because this is now greater than T, the answer is 3.

Implementation

Here's that problem implemented:

```
#include <iostream>
    #include <algorithm>

    using namespace std;
    const int MAX = 105;
    int A[MAX];

    int main()
    {
       int T, N, numberOfThings = 0, currentTime = 0;
       cin >> N >> T;
       for(int i = 0;i < N;++i)
          cin >> A[i];
       sort(A, A + N);
       for(int i = 0;i < N;++i)
       {
          currentTime += A[i];
          if(currentTime > T)
```

```
            break;
        numberOfThings++;
    }
    cout << numberOfThings << endl;
    return 0;
}
```

This is a trivial example, and once you see the problem, you should clearly see that the greedy algorithm can be applied.

Let's look at a more complex Scheduling problem.

Here's what you have:

- A list of all the tasks that must be completed today
- The time needed to do each task
- The priority for each task – in computer science, this is called the weight

You need to work out the order the tasks should be done in to get the optimum result.

First, the inputs must be analyzed. The following are the inputs for this problem:

- **Integer N** indicates how many jobs you want to finish
- **Lists P:** Priority or weight
- **List T:** the time needed for the completion of each job

Next, you need to understand the criteria that should be optimized. To do this, you need to determine the total time required to finish each task:

$C(j) = T[1] + T[2] + \ldots. + T[j]$ where $1 <= j <= N$

Why? Because jth work needs to wait until the first (j-1) tasks have been done. After that, jth will need T(j) time to complete.

For example, let's say T = {1, 2, 3}. In that case, it will take the following completion time:

This is because **jth** work has to wait till the first (j-1) tasks are completed, after which it requires T[j] time for completion.

For example, if T = {1, 2, 3}, the completion time will be:

- C(1) = T[1] = 1
- C(2) = T[1] + T[2] = 1 + 2 = 3
- C(3) = T[1] + T[2] + T[3] = 1 + 2 + 3 = 6

You want the shortest possible completion time but it isn't quite as simple as that.

In any specified sequence, the jobs at the start of the queue have much shorter completion times, while those at the back of the queue have longer times.

So, what is the best way to complete all these tasks?

Well, that depends on what your objective function is. The scheduling problem has many different objective functions but the objective function F = the weighted sum of all the completion times.

F = P[1] * C(1) + P[2] * C(2) + …… + P[N] * C(N)

We need to minimize this objective function.

Special Cases

Let's consider a couple of special cases that are somewhat intuitive in terms of the optimal solution. When we look at these special cases, some natural greedy algorithms will appear, leading to the next step – figuring out how to choose the right one, which you then need to prove correct.

These are the two special cases:

1. The time needed to complete all the tasks is the same – T[i] = T[j]. In this, 1 < = i, j < = N. However, while the tasks have the same completion time, they don't have the same priorities. So, what is the sensible order to schedule these tasks?
2. All the tasks have the same priorities – P[i] = P[j]. In this, 1 < = N. However, each task takes a different completion time, so what order should they be scheduled in?

Where the completion time for each task is the same, preference should be given to the task with the highest priority.

Let's dive a bit deeper into each case.

Case 1

You need to minimize an objective function so, let's assume that the time needed to complete the tasks is t.

T[i] = t where 1 <= i <= N

Regardless of the sequence you use, each task will have the following completion time:

C(1) = T[1] = t
C(2) = T[1] + T[2] = 2 * t
C(3) = T[1] + T[2] + T[3] = 3 * t
…
C(N) = N * t

Making the objective function as small as possible requires that the highest priority and the shortest completion time are associated with one another.

Case 2

In this case, if each task has a different priority, you are required to favor the one that requires the shortest amount of completion time. Let's assume that the tasks all have a priority of P.

F = P[1] * C(1) + P[2] * C(2) + …… + P[N] * C(N)
F = p * C(1) + p * C(2) + …… + p * C(N)
F = p * (C(1) + C(2) + …… + C(N))

Minimizing the value of objective function F requires that (C(1) + … + C(N)) is minimized. You can do this if you begin working on those tasks that need the shortest completion time.

In terms of giving preference to tasks, there are two rules. Preference must be given to those tasks that:

1. Have higher priority
2. Take less completion time

This leads us to the next step, which is moving past the special cases and looking at the general case. In the general case, the completion time and priority are different for each task.

Let's say you have two tasks, and those two rules above provide the same advice. In that case, the task with the lowest completion time and the highest priority is clearly the one that should be completed first. But what if you get conflicting advice from the rules? What if, out of your two tasks, one needs a longer completion time while the other has the higher priority? For example, P[i] > P[j] but T[i] > T[j]. Which task should you do first?

Can the time and priority parameters be aggregated into one score? Can this be done in such a way that sorting the tasks from high to low scores would always give us the best solution?

Don't forget those rules:

1. Higher priority tasks should be given preference, so they lead to a higher score
2. Tasks that don't take so long to complete should be given priority so that the score decreases the more time is required.

A simple mathematical function can be used. This takes the two numbers (for priority and time) as the input, and a single number (score) is returned as the output. At the same time, the function meets both rules. There are many functions like this, but we'll look at two of the simple ones:

- **Algorithm 1** : the jobs are ordered in decreasing value - (P[i] - T[i])
- **Algorithm 2** : the jobs are ordered in decreasing value - (P[i] / T[i])

To keep this simple, we assume there are no ties.

We now have two algorithms, but at least one is wrong. We need to rule out the incorrect algorithm that doesn't do what we want.

T = {5, 2} and P = {3, 1}

Algorithm 1 tells us that(P[1] – T[1]) < P[2] – T[2]) so task 2 is the first task to be completed. In this case, the objective

function would be:

F = P[1] * C(1) + P[2] * C(2) = 1 * 2 + 3 * 7 = 23

Algorithm 2 tells us that (P[1] / T[1]) > (P[2] / T[2]), so task 1 is the first task to be completed. In this case, the objective function would be:

F = P[1] * C(1) + P[2] * C(2) = 3 * 5 + 1 * 7 = 22

Because we won't always get the right answer from the first algorithm, we must assume it is sometimes incorrect.

Note

Don't forget that greedy algorithms are wrong in more cases than right. Even though we ruled out algorithm 1 as not being correct, it in no way implies a guarantee that the second one will be correct. In our case, though, the second algorithm does turn out to be correct all the time.

Below, you can see the algorithm that will return the objective function's optimal value:

```
Algorithm (P, T, N)
  {
      let S be an array of pairs ( C++ STL pair ) storing the scores and their indices
      , C be the completion times and F be the objective function
      for i from 1 to N:
        S[i] = ( P[i] / T[i], i )  // Algorithm 2
      sort(S)
      C = 0
      F = 0
      for i from 1 to N:    // Greedily make the best choice
        C = C + T[S[i].second]
        F = F + P[S[i].second]*C
```

```
    return F
}
```

Time Complexity

We have two loops taking O(N) time and a sorting function that takes O(N * Log N) time. That means the time complexity is O(2 * N + N * Log N) = O(N * Log N).

Proof of Correctness

So, how do we prove that the second algorithm is correct? To do this, we can use proof by contradiction. Let's assume that what we're attempting to prove is false. From that, we can derive something else that is definitely false.

So we can assume that the greedy algorithm doesn't give us an optimal solution as its output, which means there is a different solution that the greedy algorithm doesn't output that is much better than the algorithm.

A = Greedy schedule (not an optimal schedule)

B = Optimal Schedule (the best schedule you can make)

Assumption 1: all (P[i] / T[i]) are different.

Assumption 2: (for simplicity, this won't affect the generality) (P[1] / T[1]) > (P[2] / T[2]) > …. > (P[N] / T[N])

Assumption 2 ensures that the greedy schedule is A = (1, 2, 3). Because we now know that A isn't optimal, and we also know that A isn't equal to B, because B is optimal, we can claim that there must be two consecutive jobs n B (i, j) and that the earlier job has the large index of (i > j). This is true because A = (1, 2, 3, …, N) is the only schedule with the property where the indices only go up.

That means

B = (1, 2, …, i, j, …, N) where i > j.

One more thing you must consider if you swap the jobs is the profit or loss impact of the action. Consider the effect the swap

will have on these completion times:

1. Work on k other than i and j
2. Work on i
3. Work on j

There are 2 cases for k:

1. When k is to the left of i and j in B – if i and j are swapped, it will not affect k's completion time
2. When k is to the right of i and j in B – after i and j are swapped, k will have a completion time of C(k) = T[1] + T[2] + … + T[j] to T[i] – k stays the same.

The completion time for i is :

- **Before swapping –** C(i) = T[1] + T[2] + … + T[i]
- **After swapping -** C(i) = T[1] + T[2] + … + T[j] + T[i]

It's clear that i's completion time increases by T[j] and j's completion time decreases by T[i].

As a result of the swap, the loss is (P[i] / T[i]) < (P[j] / T[j]).

Using the second assumption from above, i > j implies that (P[i] / T[i]) < (P[j] / T[j]). As such, (P[i] * T[j]) < (P[j] * T[i]), which means that Loss < Profit. While B is improved by the swap, it is a contradiction, as we first thought, because we assumed that B would be optimal. That completes the proof.

Where Are Greedy Algorithms Used?

For a greedy algorithm to work, a problem must have the following two components:

1. **Optimal substructures.** An optimal solution for any problem will always contain an optimal solution to each subproblem.
2. **A greedy property.** If the choice you make seems the best one for the given time, and you leave the subproblems to be solved until later, you will still get an optimal solution and will not have to go back and reconsider your previous choices.

For example:

Activity Selection Problem

This problem is known as a combinatorial optimization problem, and it concerns selecting non-conflicting activities that need to be performed in a specific time frame. The specified activities are all marked with a start time of S_i and a finish time of f_i.

The problem requires that we select the largest (maximum) number of activities one person or machine can perform, assuming only one activity is worked on at a time. The activity selection problem is also known by another name, the interval scheduling maximization problem, or ISMP for short. This is a special type of interval scheduling problem.

One of the most common applications this problem is used in is to schedule one room for several competing events. Each event has time requirements, i.e., a start time and end time, and, within the operations research framework, more will arise.

Let's say, for example, that we have n activities. Each has a start time, $S_{i,}$ and a finish time, $f_{i.}$ Two of those activities, i and j, are considered non-conflicting, so long as $S_i \geq f_i$ or $S_j \geq f_{i.}$ In the activity selection problem, we need to find S, which is the maximal solution set, of the non-conflicting activities. Put simply, no solution set S' must exist, where |S' |>|S|, in a case where several maximal solutions are equal in size.

This kind of selection problem is notable because when we use a greedy algorithm to get our solution, it will result in an optimal solution every time. Below you can see the pseudocode showing the iterative algorithm version and proof of the result's optimality.

Greedy-Iterative-Activity-Selector(A, s, f):

```
    Sort A by finish times stored in f
  S = {A[1]}
```

```
k = 1

n = A.length

for i = 2 to n:
    if s[i] ≥ f[k]:
        S = S U {A[i]}
        k = i

return S
```

Let's break this pseudocode down.

Line 1 – We call this algorithm the Greedy Iterative Activity Selector, firstly because it is a greedy algorithm, and secondly because it is an iterative algorithm. There is a recursive version too.

- Array *A* contains the activities
- Array *8* contains the start times of the activities listed in *A*
- Array *f* contains the finish times of the activities listed in *A*

Note – all these arrays begin at index 1 and run to the maximum length of the array

Line 3 – this line sorts array *A* in increasing order of the finish times. To do this, it uses the times stored in array *f* . Using something like Heapsort, Mergesort, Quicksort, or a similar algorithm, this operation can be done in O (n Log n) time.

Line 4 – this line creates a set, *S* , which stores the specified activities. Activity *A* [1], which is the one with the earliest finish time, is used to initialize the set.

Line 5 - this line creates a variable, *k* , that tracks the last selected activity's index.

Line 9 – this line begins iterating from array *A's* second element to the last one.

Lines 10 and 11 – if s[i] (start time) of (A[i]) (the ith activity) is greater than or equal to *f* [k] (the finish time) of (A[k]) (the last selected activity), A[i] is compatible to those activities selected in set S and can be added to S.

Line 12 – this line updates the last selected activity's index to the activity just added.

Proof of Optimality

Let's say that S = {1, 2, …, n} is the set of activities placed in order of the finish time. We assume that the optimal solution is A $\subseteq$ S, in order of finish time and that A's first activity has an index of k ≠ 1 – the greedy choice is not used to start this optimal solution. We will also show that B = (A \ {k}) ∪ {1} is an optimal solution starting with the greedy choice. Because $f1 \leq f$ k, and array A's activities are disjoint, B's activities are disjoint too. Because there are the same number of activities in A and B - |A| = |B|, we can assume that B is optimal, too.

When the greedy choice has been made, the problem is reduced to finding the subproblem's optimal solution. Let's say that the optimal solution to our primary problem S, with the greedy choice, if A, then the optimal solution to the selection problem

$$S' = \{i \in S : s_i \geq f_1\}$$

is

$$A' = A \setminus \{1\}$$

Why? If this wasn't the case, a solution B' to S' must be picked containing more activities than A' with the greedy choice for S'. If 1 is added to B', we get a solution B to S containing more activities than A, which is more than feasible, but this is contradictory to the optimality.

Weighted Activity Selection

A generalized version of the selection problem requires an optimal set is selected of non-overlapping activities where the

total weight has been maximized. However, there is no greedy solution this time, unlike the unweighted problem. However, we can use the following approach to form a dynamic programming solution.

Let's say we have an optimal solution with activity k. On the left and right of k, we have non-overlapping solutions, and we can recursively find solutions for both sets because of the optimal substructure. We don't know k at this stage, but each activity can be tried and will lead to an $O(n^3)$ solution. We can optimize this further because we can find the optimal solution for each of the activity sets in (i, j) if we know the solution for (i, t), in which t is the final non-overlapping interval where j is in (i, j). This would give us an O(n2) solution, and we can even optimize that further. Not all the ranges (i, j) need to be considered, only (1, j). In the algorithm below, we get an O(n Log n) solution:

```
Weighted-Activity-Selection(S):  // S = list of activities

    sort S by finish time
    opt[0] = 0  // opt[j] represents the optimal solution (sum of weights of selected activities) for S[1,2..,j]

    for i = 1 to n:
        t = binary search to find the activity with a finish time <= start time for i
            // if there is more than one activity like this, choose the one that finishes last
        opt[i] = MAX(opt[i-1], opt[t] + w(i))

    return opt[n]
```

Fractional Knapsack Problem

The fractional knapsack problem, also called the continuous knapsack problem, is a combinatorial optimization algorithmic problem, where a container needs to be filled with fractions of different types of material. These fractions are chosen to ensure the value of the materials is maximized.

This problem is similar to the classic knapsack problem, where the items that go in the container are indivisible. However, where that problem can be solved in NP-hard time, the fractional knapsack problem is solvable in polynomial time. This is just one example of how making a tiny change in a problem's formation can significantly impact its computational complexity.

Let's define our problem.

We can specify an instance of the knapsack problem with its numerical capacity W and the collection of materials to go in it. Each material has two numbers – w_i, the weight of the material available for selection, and vi, which is the material's total value. The problem goal is to select an amount of each material $x_i \leq w_{i,}$ subject to constraints in terms of the knapsack's capacity:

$$\sum_i x_i \leq W$$

Ensuring the total benefit is maximized

$$\sum_i x_i v_i$$

The classic knapsack problem requires that x_i must be zero or w_i for each amount, but the fractional knapsack problem is different in that x_i is allowed to range from zero to wi continuously.

Formulations of this specific problem rescale the x_i variable to be in the 0 to 1 range, resulting in a capacity constraint of:

$$\sum_{i} x_i w_i \leq W$$

And, once again, our goal is to ensure the total benefit is maximized:

$$\sum_{i} x_i v_i$$

One way of solving the fractional knapsack problem is to use a greedy algorithm. The algorithm must consider the sorted materials in terms of their values per the unit weight. X_i is chosen to be as big as possible for each material:

- If the sum of all the choices made up to this point is equal to W's capacity, the algorithm will set $x_i = 0$
- If d (the difference between the sum of all the choices made up to this point) and W is less than w_i, the algorithm will set $x_i = d$
- For remaining cases, the algorithm will set $x_i = w_i$

Because the materials must be sorted, the time complexity for this greedy algorithm is O(n Log n) on those inputs that have n materials. However, this can be optimized to O(n) by adapting the algorithm, so it finds weighted medians.

Greedy Algorithms Examples

The greedy technique is one of the most powerful, and it works very well for many different problems. A few algorithms are considered greedy algorithm applications, including the following.

Minimum Spanning Tree

The minimum spanning tree, or MST, is also known as the minimum weight spanning tree. It is a subset containing all the edges of an undirected graph that is connected and edge-weighted, with all the vertices connected, no cycles, and the minimum total edge weight possible.

MSTs have many use cases, and one example would be a telecommunications company supplying a new neighborhood with cable. If the company is constrained by where it can lay the cable, i.e., along certain routes, a graph will show the points (houses) that route connects. Some might be more expensive than others because the cable has to go deeper, or they are longer, and these would show on the graph as having larger weighted edges.

That graph would produce a spanning tree containing a subset of the paths. It would not have any cycles but would still connect all the points. While many spanning trees may be available, an MST would have the lowest total cost, indicating the least expensive route.

Let's say we have a connected, undirected graph G = (V, E). This graph's spanning tree would be one that spans G, which means every vertex of G is included. It would also be a subgraph of G, which means all the tree's edges belong to G.

The spanning tree's cost is the sum of weights of every edge in the tree. Out of all the spanning trees, the MST is the one with the lowest cost.

The MST can be directly applied in network design. It is used to great effect in the following types of problems:

- Traveling salesman
- Multi-terminal minimum cut
- Minimum-cost weighted perfect matching
- Cluster analysis
- Image segmentation
- Handwriting recognition

Two of the most famous algorithms in finding MSTs are Kruskal's and Prim's algorithms. Let's look into them, along with a couple of others.

Kruskal's Algorithm

This algorithm builds a spanning tree by adding edges into a tree, one at a time. Kruskal's follows the greedy approach because an edge is found with the lowest weight in each

iteration, which is then added to the ever-increasing spanning tree.

The steps to the algorithm are:

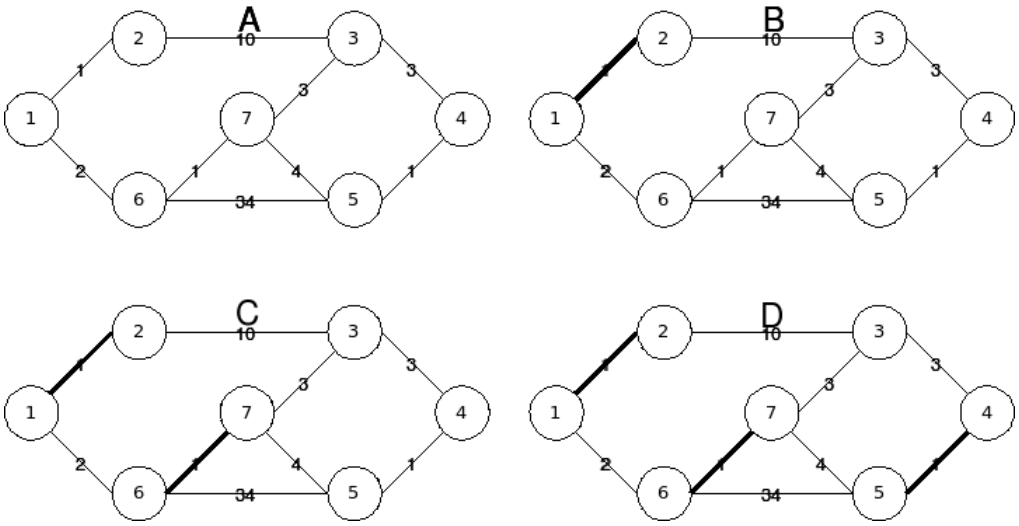
1. The graph's edges are sorted as per their weights
2. Begin to add edges to the spanning tree, starting from the smallest weight edge until you reach the largest weight edge
3. The only edges that should be added are those that don't form a cycle and only have disconnected components.

The next question is, how do we check if two vertices are connected?

We could use a DFS algorithm (depth-first search). This would start at the first vertex, then check to see if the second is visited. However, the downside to DFS is that time complexity is much larger. This is because it has an O(V + E) order, where V indicates the vertices and E indicates the edges.

So, the best solution is probably Disjoint Sets. These sets have an empty set as the intersection, which means they don't share any common elements.

Have a look at this example of Kruskal's algorithm:



In this algorithm, the edge that has the lowest weight is selected at each iteration. We begin with the lowest weighted edge, those with a weight 1. Then we go to the next lowest weighted, i.e., weight 2. These edges are disjoint. Then we

move to weight 3, which connected the graph's two disjoint edges. We can't choose any of the weight 4 edges as that would create something we cannot have – a cycle. So, we go to weight 5, but the next two edges will also create a cycle, so they too are ignored. We continue like this until we get our minimum spanning tree.

Here's an implementation of this:

```
#include <iostream>
#include <vector>
#include <utility>
#include <algorithm>

using namespace std;
const int MAX = 1e4 + 5;
int id[MAX], nodes, edges;
pair <long long, pair<int, int> > p[MAX];

void initialize()
{
    for(int i = 0;i < MAX;++i)
        id[i] = i;
}

int root(int x)
{
    while(id[x] != x)
    {
        id[x] = id[id[x]];
        x = id[x];
```

```
    }
    return x;
}

void union1(int x, int y)
{
    int p = root(x);
    int q = root(y);
    id[p] = id[q];
}

long long kruskal(pair<long long, pair<int, int> > p[])
{
    int x, y;
    long long cost, minimumCost = 0;
    for(int i = 0;i < edges;++i)
    {
        // Selecting the edges one by one in increasing order
from the start
        x = p[i].second.first;
        y = p[i].second.second;
        cost = p[i].first;
        // Check the selected edge is not creating a cycle
        if(root(x) != root(y))
        {
            minimumCost += cost;
            union1(x, y);
```

```
        }
    }
    return minimumCost;
}

int main()
{
    int x, y;
    long long weight, cost, minimumCost;
    initialize();
    cin >> nodes >> edges;
    for(int i = 0;i < edges;++i)
    {
        cin >> x >> y >> weight;
        p[i] = make_pair(weight, make_pair(x, y));
    }
    // Sort the edges in ascending order
    sort(p, p + edges);
    minimumCost = kruskal(p);
    cout << minimumCost << endl;
    return 0;
}
```

Time Complexity

The operation that takes the most time in this algorithm is sorting. This is because the Disjoint operations have a total complexity of O(E Log V), which is also Kruskal's overall time complexity.

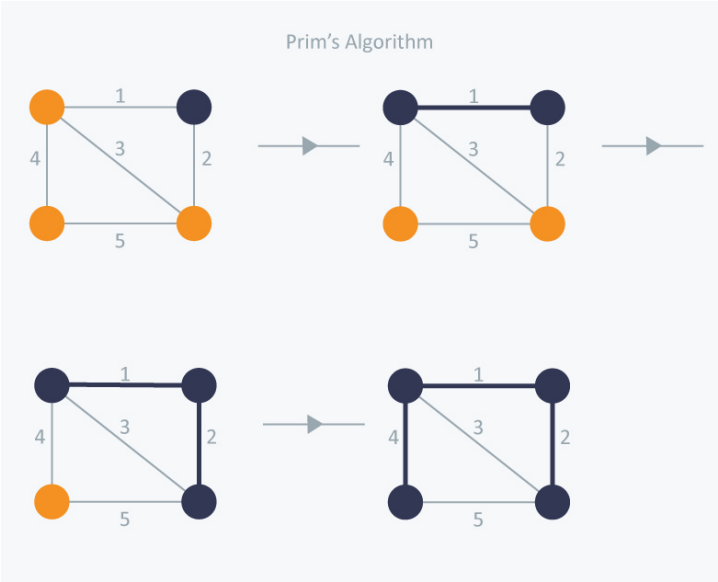
Prim's Algorithm

Prim's algorithm is also known as Jarnik's algorithm, and it follows a greedy approach to find a weighted, undirected graph's minimum spanning tree. This means it will find a subset containing the edges forming a tree with every vertex. In this tree, the total weight of the edges is minimized. Prim's builds a tree one vertex at a time, starting from an arbitrary vertex. Each step adds the least expensive connection to another vertex.

Here are the steps in the algorithm:

1. Two disjoint vertices sets are maintained. One has vertices from the increasing spanning tree, while the other contains those not in the tree.
2. The least expensive vertex is selected from those connected to the spanning tree but not in the tree. This is added to the tree, typically using a Priority Queue. The vertices to be added to the tree are inserted into the Priority Queue.
3. The last step is to check there are no cycles. This is done by marking the already selected nodes and inserting the unmarked nodes into the Priority Queue.

Have a look at this example:



In this algorithm, we begin with an arbitrary node, any one will do, and mark it. A new vertex is marked in each iteration – this vertex must be adjacent to an already-marked vertex.

Because Prim's is a greedy algorithm, it selects the least expensive edge and marks the vertex. Now, we have three choices – the edges with the weights 3, 4, and 5. However, choosing weight 3 will result in a cycle, which we can't have. In that case, we need to select those with weight 4, which results in a minimum spanning tree with a total cost of 7, which is = 1 + 2 + 4.

Here's an implementation of this algorithm:

```
#include <iostream>
#include <vector>
#include <queue>
#include <functional>
#include <utility>

using namespace std;
const int MAX = 1e4 + 5;
typedef pair<long long, int> PII;
bool marked[MAX];
vector <PII> adj[MAX];

long long prim(int x)
{
    priority_queue<PII, vector<PII>, greater<PII> > Q;
    int y;
    long long minimumCost = 0;
    PII p;
    Q.push(make_pair(0, x));
    while(!Q.empty())
    {
```

```
        // Select the edge that has the minimum weight
        p = Q.top();
        Q.pop();
        x = p.second;
        // Checking for a cycle
        if(marked[x] == true)
            continue;
        minimumCost += p.first;
        marked[x] = true;
        for(int i = 0;i < adj[x].size();++i)
        {
            y = adj[x][i].second;
            if(marked[y] == false)
                Q.push(adj[x][i]);
        }
    }
    return minimumCost;
}

int main()
{
    int nodes, edges, x, y;
    long long weight, minimumCost;
    cin >> nodes >> edges;
    for(int i = 0;i < edges;++i)
    {
        cin >> x >> y >> weight;
```

```
        adj[x].push_back(make_pair(weight, y));
        adj[y].push_back(make_pair(weight, x));
    }
    // Select 1 as your starting node
    minimumCost = prim(1);
    cout << minimumCost << endl;
    return 0;
}
```

Time Complexity

Prim's algorithm has a time complexity of O((V + E) Log V). This is because every vertex that goes in the Priority Queue is only inserted once, and these insertions take logarithmic time.

Dijkstra's Algorithm

While Dijkstra's algorithm has many variants, the commonest use is to find the shortest path in a graph between two vertices, ensuring that the total sum of the edge weights is minimized.

Here are the algorithm steps:

- All the vertices distances are set as = infinity. The exception is the source vertex, which is set to = 0
- The source vertex is pushed into a min-priority queue. This is in the form of (distance, vertex) because the min-priority queue comparison is done according to the vertex distances
- The vertex that is the minimum distance from the priority queue is popped –(popped vertex = source)
- The distances between the popped vertex and the connected vertices are updated where current vertex distance + edge weight < next vertex distance. The vertex that has the new distance is pushed to the priority queue
- If the vertex that got popped had previously been visited, continue without it

- The same algorithm should be applied repeatedly until there is nothing left in the priority queue

Here's the implementation, assuming we have source vertex = 1:

```
#define SIZE 100000 + 1


vector < pair < int , int > > v [SIZE];    // each vertex contains the connected vertices with the edges weights
int dist [SIZE];
bool vis [SIZE];


void Dijkstra(){
                                        // the vertices distances are set as infinity
    memset(vis, false , sizeof vis);          // all vertices are set as unvisited
    dist[1] = 0;
    multiset < pair < int , int > > s;          // multiset does the job as a min-priority queue


    s.insert({0 , 1});                      // the source node is inserted with a distance = 0


    while(!s.empty()){


        pair <int , int> p = *s.begin();        // the vertex with the minimum distance is popped
        s.erase(s.begin());


        int x = p.s; int wei = p.f;
```

```
            if( vis[x] ) continue;                    // check if the
popped vertex has previously been visited
            vis[x] = true;


          for(int i = 0; i < v[x].size(); i++){
             int e = v[x][i].f; int w = v[x][i].s;
              if(dist[x] + w < dist[e]  ){             // check if we
can minimize the next vertex distance
                 dist[e] = dist[x] + w;
                  s.insert({dist[e],  e} );           // the next vertex
with the updated distance is inserted
             }
          }
       }
    }
```

Time Complexity

Dijkstra's algorithm has a total time complexity of O(V2). However, when the min-priority queue is used, that reduces to O(V + ElogV).

However, this would prove expensive in time should we need to find the shortest path between all vertex pairs. We'll discuss an algorithm designed for that in the dynamic programming chapter.

Graph Coloring Greedy Algorithm

Graph coloring is a special type of graph labeling covered under graph theory. It involves assigning labels, also called colors, to elements on a graph with specified constraints. Put simply, it is a method whereby a graph's vertices are colored in such a way that no vertices adjacent to one another share a color. This is known as vertex coloring. In the same way, edge coloring gives each edge a color in such a way that no two

adjacent edges are colored the same. Face coloring on planar graphs assigns colors to regions or faces in a way that no two faces sharing a boundary are colored the same.

- **Chromatic Number –** this indicates the least number of colors needed to color the graph, for example, 3
- **Vertex Coloring –** this indicates the subject's starting point. Many coloring problems can be turned into vertex versions. For example, a graph's edge coloring is simply the line graph being vertex colored, and a plane graph's face coloring is simply the dual being vertex colored. However, more often than not, we state and study non-vertex coloring problems exactly as they are. This is partly due to perspective and partly due to the fact that some problems should be studied as non-vertex, such as edge coloring.

Origin

The use of colors comes from when colors were used to color maps of the world, where each country is completely filled in with color. This was then generalized to coloring the faces on plane-embedded graphs. It then became coloring the vertices by planar duality, and in this form can generalize to any graph. The colors are typically indicated by the first few non-negative or positive integers in terms of computer and mathematical representation. Generally, the coloring problem's nature is dependent on how many colors there are but not what they are.

Pseudocode

- The first vertex is colored with the first color
- The remaining vertices are colored in the same way
- The currently chosen vertex should be colored with the color of the lowest number not used on adjacent colored vertices
- If the colors have been used on any vertex adjacent to v, a new color should be assigned to it.

Here's an implementation:

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

// A class representing an undirected graph
class Graph
{
    int V;    // No. of vertices
    list<int> *adj;    // A dynamic array of adjacency lists
public:
    // Constructor and destructor
    Graph(int V)   { this->V = V; adj = new list<int>[V]; }
    ~Graph()       { delete [] adj; }

    // function that adds an edge to a graph
    void addEdge(int v, int w);

    // Prints the greedy coloring of the vertices
    void greedyColoring();
};

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v);  // Note: the graph is undirected
}
```

```
// Assigns colors from 0 upwards to all vertices and prints
// the assignment of colors
void Graph::greedyColoring()
{
    int result[V];

    // Assign the first color to the first vertex
    result[0]  = 0;

    // Initialize remaining V-1 vertices as unassigned
    for (int u = 1; u < V; u++)
        result[u] = -1;  // u does not have a color assigned to
it

     // A temporary array that stores the available colors.
The true
    // value of available[cr] means the color cr is
    // assigned to an adjacent vertex
    bool available[V];
    for (int cr = 0; cr < V; cr++)
        available[cr] = false;

    // Assign colors to the remaining V-1 vertices
    for (int u = 1; u < V; u++)
    {
        // Process the adjacent vertices and flag their colors
        // as unavailable
        list<int>::iterator i;
```

```
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
            if (result[*i] != -1)
                available[result[*i]] = true;

        // Find the first available color
        int cr;
        for (cr = 0; cr < V; cr++)
            if (available[cr] == false)
                break;

        result[u] = cr; // Assign the found color

        // Reset the values to false for the next iteration
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
            if (result[*i] != -1)
                available[result[*i]] = false;
    }

    // print the result
    for (int u = 0; u < V; u++)
        cout << "Vertex " << u << " ->  Color "
             << result[u] << endl;
}

// Driver program to test the function above
int main()
{
```

```
    Graph g1(5);
    g1.addEdge(0, 1);
    g1.addEdge(0, 2);
    g1.addEdge(1, 2);
    g1.addEdge(1, 3);
    g1.addEdge(2, 3);
    g1.addEdge(3, 4);
    cout << "Coloring of graph 1 \n";
    g1.greedyColoring();

    Graph g2(5);
    g2.addEdge(0, 1);
    g2.addEdge(0, 2);
    g2.addEdge(1, 2);
   g2.addEdge(1, 4);
    g2.addEdge(2, 4);
    g2.addEdge(4, 3);
    cout << "\nColoring of graph 2 \n";
    g2.greedyColoring();

    return 0;
}
```

Time Complexity

In the worst case, the time complexity is O(V^2 + E), where the vertex is V, and the edge is E.

Applications

Greedy coloring algorithms have a number of applications:

1. **Making a timetable or schedule.** Let's say that we want to create a schedule for a local university to track exams. We list the different subjects and all the students taking each subject. Some students will be taking common subjects. How can we schedule the exams in a way that no student is down to take two exams at the same time? We also need to know the minimum number of time slots needed for all the exams to be scheduled. We can present this as a graph, where the subjects are vertices, and an edge between two of them indicates a common student. This graph coloring problem has a minimum number of time slots equal to the graph's chromatic number.
2. **Assigning Mobile Radio Frequencies.** When radio towers are assigned with frequencies, every tower in the same location must have a different one. How do we assign the frequencies when we are bound by this constraint? What's the minimum required number of frequencies? This is also a graph coloring problem where the vertices are towers, and an edge between two indicates they are close to one another, within a certain range.
3. **Mobile Radio Frequency Assignment** : When frequencies are assigned to towers, frequencies assigned to all towers at the same location must be different. How to assign frequencies with this constraint? What is the minimum number of frequencies needed? This problem is also an instance of a graph coloring problem where every tower represents a vertex, and an edge between two towers represents that they are in range of each other.
4. **Sudoku.** This is another version of a graph coloring problem where the vertices represent cells. If two vertices have an edge between them, it means they are in the same block, row, or column.
5. **Register Allocation.** Register allocation is a compiler organization process where large numbers of target variables are assigned to a smaller number of CPU registers.

6. **Bipartite Graphs** . Graph coloring with two colors is one way of testing if a graph is a bipartite graph or not. If a graph can be colored in two colors, it is bipartite. If not, it isn't.
7. **Map Coloring** . Geographical maps showing states or countries where no two adjacent cities can have the same color. Four colors are enough to color a map.

Huffman Codes

Huffman coding falls under the lossless data compression algorithm family, and the goal with this is to assign input characters with variable-length code. The lengths of the codes are based on the corresponding character's frequencies. The smallest code goes to the most frequent character, while the largest code goes to the least frequent character.

Any variable-length code assigned to an input character is known as a prefix code. This means that the bit sequences or codes are assigned in a way that a code assigned to a character cannot be a prefix to another character. Huffman coding uses this to ensure that when the generated bitstream is decoded, there is no ambiguity.

Let's look at a counter example to understand these prefix codes better. Let's say we have four characters – a, b, c, and d. The variable-length codes corresponding to these characters are 00, 0, 0, and 1. This causes no small amount of ambiguity because the prefix code we assigned to c is the same as the one assigned to a and b. Where the compressed bit stream is 0001, we could have a compressed output of cccd, ccb, acd, or ab.

Huffman coding falls into two parts:

1. The input characters are used to build the Huffman tree
2. The Huffman tree is traversed, and codes are assigned to the characters.

Building a Huffman Tree

The steps involved in building a Huffman tree are below. The input is an array containing unique characters and the

frequency of each one's occurrence. The output is the Huffman tree.

1. A leaf node must be created for every unique character, and a min-heap is then built out of all the leaf nodes. The min-heap is used as a priority queue, and the frequency field value is used for comparing two leaf nodes in the heap. To start with, the least frequent character will be at the root.
2. The two nodes with the smallest frequency are extracted from the min-heap.
3. A new internal node is created. Its frequency is equal to the sum of both the node frequencies. The first extracted node is the left child and the second extracted node is the right child. The node is added to the min-heap.
4. Repeat the second and third steps until there is only one node left in the heap. This node is the root node, and your Huffman tree is completed.

Let's look at an example to understand the logic:

character	**Frequency**
a	5
b	9
c	12
d	13
e	16
f	45

Step One – a min-heap is built with 6 nodes. Each node represents a tree's root where the tree has just one node.

Step Two – two minimum frequency nodes are extracted from the min-heap. A new internal node is added with a frequency of 5 + 9 = 14.

Min-heap now contains five nodes. Four of those are the roots of trees with only one element each, while the fifth node is the root of a tree with three elements.

character	Frequency
c	12
d	13
Internal Node	14
e	16
f	45

Step Three – two minimum frequency nodes are extracted from the heap. A new internal node is added with a frequency of 12 + 13 = 25.

The min-heap now contains four nodes. Two of these are the roots of trees with only one element each, while the other two are the roots of trees with more than a single node.

character	Frequency
Internal Node	14
e	16
Internal Node	25
f	45

Step Four – two minimum frequency nodes are extracted. A new internal node is added with a frequency of 14 + 16 = 30.

The min-heap now has three nodes.

Step 4: two minimum frequency nodes are extracted. Add a new internal node with frequency 14 + 16 = 30

character	Frequency
Internal Node	25
Internal Node	30
f	45

Step Five – two minimum frequency nodes are extracted. A new internal node is added with a frequency of 25 + 30 = 55.

The min-heap now contains two nodes.

character Frequency

f 45

Internal Node 55

Step Six – two minimum frequency nodes are extracted. A new internal node is added with a frequency of 45 + 55 = 100.

character Frequency

Internal Node 100

The algorithm will stop because there is only one node in the heap.

Printing the Codes from the Huffman Tree

First, the tree must be traversed, starting at the root, and maintaining an auxiliary array. As you move to the left child, 0 must be written to the array, and, as you move to the right child, 1 must be written to the array. When you encounter a leaf node, the array must be printed.

Here's the code:

character code-word

f 0

c 100

d 101

a 1100

b 1101

e 111

Here is this approach implemented in C:

```
// C program for Huffman Coding
#include <stdio.h>
#include <stdlib.h>
```

```
// We can avoid this constant by explicitly
// calculating the height of the Huffman Tree
#define MAX_TREE_HT 100

// A Huffman tree node
struct MinHeapNode {

    // One of the input characters
    char data;

    // The frequency of the character
    unsigned freq;

    // The left and right child of this node
    struct MinHeapNode *left, *right;
};

// A Min Heap:  Collection of
// min-heap (or Huffman tree) nodes
struct MinHeap {

    // The current size of the min-heap
    unsigned size;

    // The capacity of the min-heap
    unsigned capacity;
```

```
    // An array of minheap node pointers
    struct MinHeapNode** array;
};

// A utility function that allocates a new
// min-heap node with a given character
// and the frequency of the character
struct MinHeapNode* newNode(char data, unsigned freq)
{
        struct MinHeapNode* temp = (struct MinHeapNode*)malloc(
        sizeof(struct MinHeapNode));

    temp->left = temp->right = NULL;
    temp->data = data;
    temp->freq = freq;

    return temp;
}

// A utility function that creates
// a min-heap of a given capacity
struct MinHeap* createMinHeap(unsigned capacity)

{

    struct MinHeap* minHeap
```

```
                = (struct MinHeap*)malloc(sizeof(struct
MinHeap));

    // current size is 0
    minHeap->size = 0;

    minHeap->capacity = capacity;

    minHeap->array = (struct MinHeapNode**)malloc(
                    minHeap->capacity * sizeof(struct
MinHeapNode*));
    return minHeap;
}

// A utility function that
// swaps two min heap nodes
void swapMinHeapNode(struct MinHeapNode** a,
                struct MinHeapNode** b)

{

    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

// The standard minHeapify function.
void minHeapify(struct MinHeap* minHeap, int idx)
```

```
{

    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;

    if (left < minHeap->size
        && minHeap->array[left]->freq
               < minHeap->array[smallest]->freq)
        smallest = left;

    if (right < minHeap->size
        && minHeap->array[right]->freq
               < minHeap->array[smallest]->freq)
        smallest = right;

    if (smallest != idx) {
         swapMinHeapNode(&minHeap->array[smallest],
                         &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
}

// A utility function that checks
// if the size of the heap is 1 or not
int isSizeOne(struct MinHeap* minHeap)
```

```
{

    return (minHeap->size == 1);
}

// A standard function that extracts
// the minimum value node from the heap
struct MinHeapNode* extractMin(struct MinHeap*
minHeap)

{

    struct MinHeapNode* temp = minHeap->array[0];
      minHeap->array[0] = minHeap->array[minHeap-
>size - 1];

    —minHeap->size;
    minHeapify(minHeap, 0);

    return temp;
}

// A utility function that inserts
// a new node to the min-heap
void insertMinHeap(struct MinHeap* minHeap,
                   struct MinHeapNode* minHeapNode)

{
```

```
    ++minHeap->size;
    int i = minHeap->size - 1;

    while (i
           && minHeapNode->freq
                < minHeap->array[(i - 1) / 2]->freq) {

       minHeap->array[i] = minHeap->array[(i - 1) / 2];
       i = (i - 1) / 2;
    }

    minHeap->array[i] = minHeapNode;
}

// A standard function that builds the min-heap
void buildMinHeap(struct MinHeap* minHeap)

{

    int n = minHeap->size - 1;
    int i;

    for (i = (n - 1) / 2; i >= 0; —i)
       minHeapify(minHeap, i);
}
```

```
// A utility function that prints an array of size n
void printArr(int arr[], int n)
{
    int i;
    for (i = 0; i < n; ++i)
        printf("%d", arr[i]);

    printf("\n");
}

// Utility function that checks if this node is a leaf
int isLeaf(struct MinHeapNode* root)

{

    return !(root->left) && !(root->right);
}

// Creates a min-heap with a capacity
// equal to size and inserts all the characters of
// the data[] in min heap. Initially the size of
// min-heap is equal to the capacity
struct MinHeap* createAndBuildMinHeap(char data[],
                                      int freq[], int size)

{
```

```
    struct MinHeap* minHeap = createMinHeap(size);

    for (int i = 0; i < size; ++i)
        minHeap->array[i] = newNode(data[i], freq[i]);

    minHeap->size = size;
    buildMinHeap(minHeap);

    return minHeap;
}

// The main function that builds the Huffman tree
struct MinHeapNode* buildHuffmanTree(char data[],
                                     int freq[], int size)

{
    struct MinHeapNode *left, *right, *top;

    // Step 1: Create a min heap of a capacity
    // equal to size.  To start with, there are
    // nodes equal to size.
    struct MinHeap* minHeap
        = createAndBuildMinHeap(data, freq, size);

    // Iterate while the size of heap isn't 1
    while (!isSizeOne(minHeap)) {
```

```
        // Step 2: Extract the two minimum
        // frequency items from the min-heap
        left = extractMin(minHeap);
        right = extractMin(minHeap);

        // Step 3:  Create a new internal
        // node with a frequency equal to the
        // sum of the two nodes frequencies.
        // Make the two extracted nodes the
        // left and right children of this new node.
        // Add this node to the min-heap
        // '$' is a special value for the internal nodes, not
        // used
        top = newNode('$', left->freq + right->freq);

        top->left = left;
        top->right = right;

        insertMinHeap(minHeap, top);
    }

    // Step 4: The remaining node is the
    // root node and that completes the tree.
    return extractMin(minHeap);
}
```

```
// Prints the huffman codes from the root of the
Huffman Tree.
// It uses arr[] to store codes
void printCodes(struct MinHeapNode* root, int arr[],
                int top)

{

    // Assign 0 to left edge and recur
    if (root->left) {

        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }

    // Assign 1 to right edge and recur
    if (root->right) {

        arr[top] = 1;
        printCodes(root->right, arr, top + 1);
    }

    // If this is a leaf node, then
    // it will contain one of the input
    // characters. Print the character
    // and its code from arr[]
    if (isLeaf(root)) {
```

```
        printf("%c: ", root->data);
        printArr(arr, top);
    }
}

// The main function that builds a
// Huffman Tree and print the codes by traversing
// the built Huffman Tree
void HuffmanCodes(char data[], int freq[], int size)

{
    // Construct Huffman Tree
    struct MinHeapNode* root
        = buildHuffmanTree(data, freq, size);

    // Print the Huffman codes using
    // the Huffman tree built above
    int arr[MAX_TREE_HT], top = 0;

    printCodes(root, arr, top);
}

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

```
    char arr[] = { ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’ };
    int freq[] = { 5, 9, 12, 13, 16, 45 };

    int size = sizeof(arr) / sizeof(arr[0]);

    HuffmanCodes(arr, freq, size);

    return 0;
}
```

Output:

```
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111
```

Time Complexity

This has a time complexity of O(n Log n), where n indicates how many unique characters there are. When there are n nodes, we call extractMin() 2*(n – 1) times. The extractMin() function has a time complexity of O(log n) because the minHeapify() function has to be called. The overall time complexity of the Huffman codes is O(n Log n).

Applications

The Huffman coding algorithm is used mostly in cases where characters frequently appear in a series. They are also used to transmit texts and faxes and by compression formats such as GZIP, PKZIP, etc.

In the next chapter, we will discuss another design technique called Dynamic Programming.

Chapter 4: Dynamic Programming

Dynamic programming is a fancy way of saying that we take an optimization problem, break it down into smaller subproblems, and store each subproblem's solution. That way, we only solve a subproblem once.

Okay, so that may not make a whole lot of sense, but it will when you see how it works. At this stage, you need to understand that dynamic programming is useful when used on optimization problems where you need to find a minimum or maximum solution within given constraints. It looks through every potential subproblem and will not recompute a solution, guaranteeing efficiency and correctness, which isn't true for many algorithm-solving techniques.

First, we'll define a subproblem and then discuss why storing the solutions, known as memorization, is important in dynamic programming.

Subproblems

A subproblem is a smaller version of a primary problem. They look much like the primary problem but in different words in many cases. If a subproblem is formulated correctly, it will build on other subproblems to provide the solution to the total problem.

Let's look at an example. We're going to look at a dynamic programming problem to find the subproblem.

Go back in time to the 1950s. You have an old IBM-650 computer, and that means punch cards. It is your job to work that computer for one day, and you must run n number of punch cards. Every punch card i needs to run at a predetermined time s_i and stop at a predetermined finish time f_i. You can only run a single punch card at a time on the IBM-650, and each one has v_i, an associated value based on its importance to the company.

- **Problem –** it's your job to work out the best schedule to run the punch cards, ensuring the total value of the

processed punch cards is maximized.

- **Subproblem –** I'm only going to give you a brief idea because we'll be running through this in more detail later. You need the maximum value schedule for the punch cards i through n with the punch cards sorted by start time.

Note that the subproblem breaks the main problem down into separate components that help build the solution. The subproblem allows you to find the maximum value schedule for the punch cards 1 to n, and because it looks much like the original problem, the subproblem can help solve the main problem.

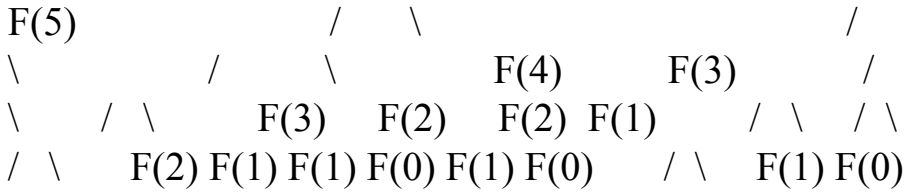
The solution must be stored or memoized for each subproblem solved in a dynamic programming problem.

Memoization with Fibonacci Numbers

When you need an algorithm implemented to calculate the Fibonacci number for a specified number, which one do you choose? Most would choose a recursive algorithm that might look like this Python implementation:

```
def fibonacciVal(n):  if n == 0:    return 0  elif n == 1:
return 1      else:          return fibonacciVal(n-1) +
fibonacciVal(n-2)
```

Sure, this does the job, but it comes with a huge cost. Here's what the algorithm needs to calculate to solve n=5, which is abbreviated to F(5)):



This tree represents all the computations needed to find the Fibonacci value of n = 5. Note that the n = 2 subproblem is solved three times – that's an awful lot of wasted computation for a small example.

Rather than calculating that value three times, what if we came up with an algorithm that only did the calculation once, stored the value, and then accessed it for all other occurrences of n = 2? Well, that's memoization for you.

So, here's the solution to the value problem written for the dynamic programming technique:

```
def fibonacciVal(n):  memo = [0] * (n+1)  memo[0], memo[1] = 0, 1  for i in range(2, n+1):  memo[i] = memo[i-1] + memo[i-2]  return memo[n]
```

Note that the return value's solution comes from memo[ ], the memorization array, filled in iteratively by the for loop. What do we mean by "iteratively?" You can see that memo[2] was calculated and then stored before memo[3], memo[4], …, memo[n]. Because this is how memo[] is filled, each solution's subproblem is solvable using the solutions from the already solved subproblems because their values had previously been stored in memo[].

We don't need to recompute any solutions using memorization, which means the algorithm is more efficient. Guaranteeing that a dynamic problem considers every possibility before choosing a solution can only be done by using memorization.

Now we've got that out of the way, we can dive into dynamic programming.

Dynamic Programming Process

The dynamic programming process requires several steps, each of which is detailed here.

Step One – Identify the subproblem in words

Many programmers get straight down to writing their code before even thinking about the problem they are trying to solve. This is not good. One of the best ways to get your brain in gear before you start writing code is to describe the subproblem using words.

If your problem can only be solved using dynamic programming, think about what you need to solve the problem and use that to write down the subproblems.

Take the punch card problem. As we found, we could write the subproblem as needing to find the maximum value schedule for the punch cards i to n with the punch cards sorted by the start time. We can reach this subproblem by realizing that we need to answer the following subproblems to answer that subproblem:

- the maximum value schedule for the punch cards n-1 to n with the punch cards sorted by the start time
- the maximum value schedule for the punch cards n-2 to n with the punch cards sorted by the start time
- the maximum value schedule for the punch cards n-3 to n with the punch cards sorted by the start time
- and so on
- the maximum value schedule for the punch cards 2 to n, with the punch cards sorted by the start time

You know you are on the right path when you can identify one subproblem that builds on other ones already solved.

Step Two – Write your subproblem as a recurring mathematical decision

When you have the words for your subproblem, you can think about writing it mathematically. Why would you do this? Because the repeated decision, or recurrence, will be what goes into your code. Plus, it gives you a way of vetting what you wrote in step one. It may not be right if your written subproblem cannot be easily coded into math.

When you try to find a recurrence, ask yourself these two questions:

1. At each step, what decision do I need to make?
2. My algorithm is at step i, so what information is needed to determine what should be done in i+1?

Going back to our punch card problem, we can ask those questions:

- **At each step, what decision do I need to make?**

Let's assume that, as we already mentioned, the punch cards have been sorted by their start time. For every punch card compatible with the current schedule, i.e., it has a start time after the finish time of the one already running, the algorithm needs to choose whether or not to run the punch card.

- **My algorithm is at step i, so what information is needed to determine what should be done in i+1?**

Now the algorithm needs to decide between those two options, which means it needs to know what the next compatible punch card is. For a specified punch card p, the next compatible one is q, such that s_q, which indicates the start time predetermined for p, comes after f _p, which is the finish time predetermined for p. We then get the minimized difference between s_q and f _p. In simple terms, it's the card that starts after the current one finishes.

By now, you should have begun to formulate that recurring mathematical decision. If not, don't worry. It gets easier as you go along.

So, here's the recurrence for this problem:

OPT(i) = max(v_i + OPT(next[i]), OPT(i+1))

So, we may need to explain this, especially if you have never written one before.

OPT(i) represents our subproblem (the maximum value schedule) from the first step.

We then mathematically represent the two options – whether to run or not – as:

v_i + OPT(next[i])

This represents the algorithm's decision to run punch card i, and the value gained from running it is added to OPT(next[i]). In this, next[i] represents the compatible card that follows i,

and OPT(next[i]) provides the maximum value schedule for the next i to n. When these two values are added together, we get the maximum value schedule that we were looking for should punch card i run.

OPT(i+1)

On the other hand, this clause is for when the algorithm decides not to run punch card i., which means we do not gain its value. OPT(i+1) provides the maximum value schedule for the punch cards i+1 to n, so it gives us that maximum value schedule should punch card i not be run.

The decision is mathematically encoded at each step to reflect our subproblem from the first step.

Step Three – Use steps one and two to solve the primary problem

Step one is where the punch card problem subproblem was written, and in step two, the recurring mathematical decision corresponding to the subproblem was written. Now we need to use this information to solve the main problem. How do we do that?

OPT(1)

It's as simple as that. The subproblem from step one is the maximum value schedule for the punch cards i to n, where the cards have been sorted in start time order. Because of that, the primary problem's solution is written in the same way as the subproblem, except the punch cards are 1 to n. Because the first two steps go together, we can write the primary problem as

OPT(1)

Step Four – Work out the memoization's array dimensions and the direction it should be filled

If you found the third a bit deceptive, you would be forgiven. You might wonder how OPT(1) can be the solution to the dynamic program when it relies on OPT(2), OPT(next[1], etc.

You would be right to spot that OPT(1) is reliant on the OPT(2) solution. The following comes right after step two:

$$OPT(1) = \max(v\_1 + OPT(next[1]), OPT(2))$$

But this isn't such a pressing issue. Remember the Fibonacci memoization we covered earlier? To find n = 5's Fibonacci value, the algorithm relies on the n = 0, n = 1, n = 2, n = 3 and n = 4 Fibonacci values already being memoized. If the memoization table is completed in the right order, the fact that OPT(1) relies on other subproblems isn't an issue.

How do we identify the right way to fill the table? Because we know that OPT(1) is relying on the OPT(2) and OPT(next[1]) solutions, and we know that the sorting means punch cards 2 and next[1] will both start after punch card 1, it should be easy to determine that the memorization table should be filled from OPT(n) to OPT(1).

The next thing to work out is what dimensions the memoization array should be. There is a trick to this – the array's dimension is equal to the number of variables and their size that OPT(.) relies on. The punch card problem has OPT(i), which means OPT(.)is only reliant on variable i, representing the number of the punch card. This suggests that we will have a one-dimensional memoization array, and because there are a total of n punch cards, the array's size is n.

So, if we have n = 5, the array would look something like this:

memo = [OPT(1), OPT(2), OPT(3), OPT(4), OPT(5)]

However, remember that most programming languages are indexed from 0, so it may be better to create the array aligning its indices with the punch card numbers:

memo = [0, OPT(1), OPT(2), OPT(3), OPT(4), OPT(5)]

Step Five – Time to code it

Coding the dynamic program requires steps two through four to be combined. You only need one more piece of information

– a base case – and you'll find this as you play about with the algorithm.

A dynamic program written for our punch card problem would look like this:

```
def punchcardSchedule(n, values, next): # Initialize memoization array - Step 4  memo = [0] * (n+1)  # Set base case  memo[n] = values[n]  # Build memoization table from n to 1 - Step 2  for i in range(n-1, 0, -1):  memo[i] = max(v_i + memo[next[i]], memo[i+1])  # Return solution to original problem OPT(1) - Step 3  return memo[1]
```

Now you know how to write a dynamic program, we'll move on to another type of dynamic programming.

Paradox of Choice: Multiple Options Dynamic Programming

While our punch card problem had to make a decision between two options – run or don't run a punch card. However, many problems have several options to choose from before a decision is made at every step.

Here's a new example.

Let's say you are selling friendship bracelets. You have n customers, and the product's value monotonically increases. This means the product prices {p_1, …, p_n} change such that, if customer j follows customer i, then $p_i \leq p_j$. The values these n customers have are {v_1, …, v_n}.

A customer, i, buys a bracelet at a price, p_1, ONLY IF $p_i \leq$ v_i}. If it isn't, the revenue from that specific customer is 0. For this problem, we will assume that all prices are natural numbers.

- **Problem –** We need to find the set prices that guarantee the maximum possible revenue from the sales

Before we dive into the steps, take a minute to think about how this could be addressed.

Step One – Identify the subproblem in words

- **Subproblem –** maximum revenue from customers i to n, where customer i-1's price was set at q

How did we did identify that subproblem? By realizing that we need the answers to the subproblems below to determine the maximum revenue for customers i to n.

- The maximum revenue from customers n-2 to n, where n-2 customer's price was q
- The maximum revenue from customers n-2 to n, where n-3 customer's price was q
- And so on

Note that the subproblem now has a new variable, q. Solving the subproblems requires knowing the price set for each customer before the subproblem. The new variable, q, ensures that the set prices are monotonic and that the current customer is tracked by variable i.

Step Two – Write your subproblem as a recurring mathematical decision

As before, there are two questions to ask yourself when trying to find the recurrence:

3. At each step, what decision do I need to make?
4. My algorithm is at step i, so what information is needed to determine what should be done in i+1?

Using our friendship bracelet problem, let's answer these questions.

- **At each step, what decision do I need to make?**

First, decide the bracelet's price for the current customer. Because this must be a natural number, it should be set in the range from q (the set price for customer i-1) to v_1 for customer i. v_i indicates the maximum price customer i will pay for the bracelet.

- **My algorithm is at step i, so what information is needed to determine what should be done in i+1?**

The algorithm has to know customer i's set price and customer i+1's value to decide on the natural number for customer i+1's set price.

With the answers to both these questions, the recurrence can be mathematically written as:

$$OPT(i,q) = \max\sim([Revenue(v\_i, a) + OPT(i+1, a)])$$

where max~ will find the overall maximum a in the range of $q \leq a \leq v\_i$

Again, we need to explain this recurrence. Because customer i-1 has a price of q, customer i has a price, a, that stays at integer q or changes to one in the range q+1 to v_i. If we want the total revenue, customer i's revenue is added to the maximum revenue from the customers i+1 to n, where customer i's price was set to a.

In simple terms, maximizing the total revenue requires the algorithm to find customer i's optimal price by checking every possible price from q to v_i, and where $v\_i \leq q$, price a stays as q.

And the Other Steps?

The first two steps are the hardest part, so go through the other steps, as detailed in the first problem, yourself, to make sure you understand them.

Runtime Analysis of Dynamic Programs

Runtime analysis is the fun part of algorithm writing. For this, we will be using big -O notation, a mathematical notation used to describe a function's limiting behavior when the argument leans towards a specific infinity or value. It is used for classifying algorithms in terms of how their space requirements and/or run time increases exponentially with the input size.

Typically, the runtime for a dynamic program is made up of these features:

- Pre-processing

- The number of times the for loop runs
- How long the recurrence takes to run in an iteration of the for loop
- Post-processing

Below you can see the form for the overall runtime:

Pre-processing + Loop * Recurrence + Post-processing

We'll use the punch card problem to do a runtime analysis. This will give you an idea of how the big-o notation works in a dynamic program. Here's the program:

```
def punchcardSchedule(n, values, next): # Initialize memoization array - Step 4  memo = [0] * (n+1)  # Set base case  memo[n] = values[n]  # Build memoization table from n to 1 - Step 2  for i in range(n-1, 0, -1):  memo[i] = max(v_i + memo[next[i]], memo[i+1])  # Return solution to original problem OPT(1) - Step 3  return memo[1]
```

Here's the runtime broken down:

- **Pre-processing** : this is all about building O(n), the memoization array.
- **The number of times the for loop runs** : O(*n*).
- **How long the recurrence takes to run in an iteration of the for loop** : recurrence run time is constant time because it has to decide between two options at every iteration - O(1).
- **Post-processing** : there isn't any O(1).

So, the overall runtime for this problem is O(n) O(n) * O(1) + O(1). In its simplified form, that is O(n).

Dynamic Programming Algorithms

While there are several dynamic algorithms, we will discuss two of them here – Floyd-Warshall and Bellman-Ford. Both of these are classed as All Pair Shortest Path Algorithms.

Floyd-Warshall Algorithm

Floyd-Warshall is an algorithm used to find the shortest path in a weighted graph between all of the pairs of vertices. It works on undirected and directed weighted graphs, but it will not work for graphs that have negative cycles, i.e., the cycle's sum of edges is negative.

Weighted graphs are those where a numerical value is associated with each edge.

Floyd-Warshall is also known as the Roy-Floyd algorithm, Floyd's algorithm, WFI algorithm, or Roy-Warshall algorithm, and it finds the shortest paths by following the dynamic programming technique.

How It Works

Shortly, you will see an image showing the graph and the solutions to the following steps. Read through this and make sense of it before you read through it again with the images:

We want to find the shortest path between every pair of vertices:

Step One – First, all the self-loops, along with the parallel edges, need to be removed from the graph. Don't forget to ensure the lowest weight edge is retained.

Note that, in our graph (shown below), there are no parallel edges or self-edges.

Step Two – the initial distance matrix needs to be written, and this uses weights to represent the distance between each pair of vertices.

The diagonal elements represent self-loops and have a distance value = 0

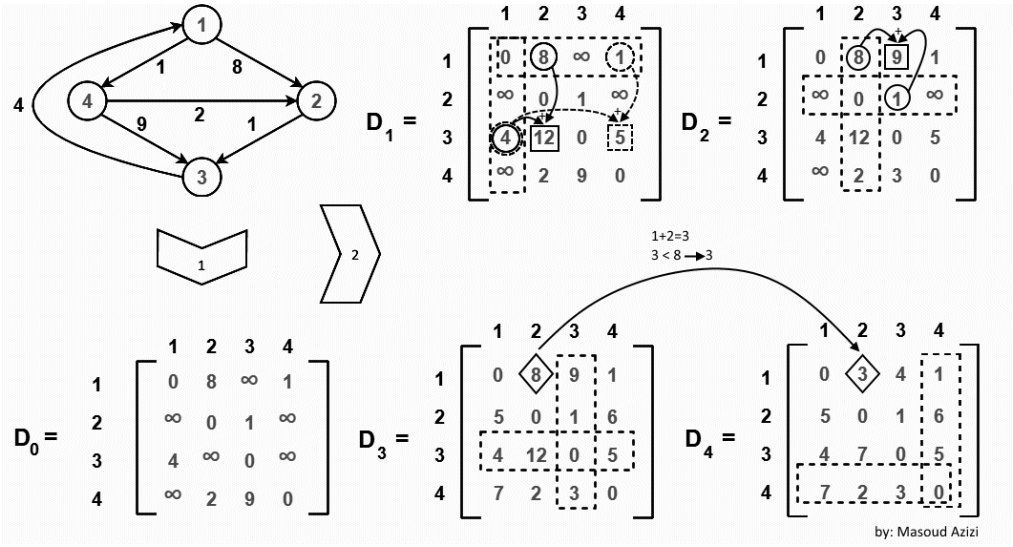
Those separated by a direct edge have a distance value = the weight of the direct edge

Those that do not have a direct edge between them have a distance value = ∞

The initial matrix for the graph can be seen in the image below as matrix D_0

Step Three – Four matrices now need to be written using the Floyd-Warshall algorithm – D_1, D_2, D_3, and D_4.

Take a look at the image below to see the graph and the matrices.



The shortest distance path between each pair of vertices is represented in the final matrix, D_4.

Note

The above problem's weighted graph has four vertices. This means the solution will contain four matrices, each 4 x 4, not including the initial distance matrix. And each matrix's diagonal elements will always be 0.

Implementation

Below you can see the Floyd-Warshall algorithm:

```
n = no of vertices
A = matrix of dimension n*n
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            Ak[i, j] = min (Ak-1[i, j], Ak-1[i, k] + Ak-1[k, j])
return A
```

And this is a C++ implementation example, showing you how it all works:

```
// Floyd-Warshall Algorithm in C++

#include <iostream>
using namespace std;

// defining the number of vertices
#define nV 4

#define INF 999

void printMatrix(int matrix[][nV]);

// Implementing floyd warshall algorithm
void floydWarshall(int graph[][nV]) {
  int matrix[nV][nV], i, j, k;

  for (i = 0; i < nV; i++)
    for (j = 0; j < nV; j++)
      matrix[i][j] = graph[i][j];

  // Adding vertices individually
  for (k = 0; k < nV; k++) {
    for (i = 0; i < nV; i++) {
      for (j = 0; j < nV; j++) {
        if (matrix[i][k] + matrix[k][j] < matrix[i][j])
```

```
            matrix[i][j] = matrix[i][k] + matrix[k][j];
          }
        }
      }
      printMatrix(matrix);
    }

    void printMatrix(int matrix[][nV]) {
      for (int i = 0; i < nV; i++) {
        for (int j = 0; j < nV; j++) {
          if (matrix[i][j] == INF)
            printf("%4s", "INF");
          else
            printf("%4d", matrix[i][j]);
        }
        printf("\n");
      }
    }

    int main() {
      int graph[nV][nV] = {{0, 3, INF, 5},
                {2, 0, INF, 4},
                {INF, 1, 0, INF},
                {INF, INF, 2, 0}};
      floydWarshall(graph);
    }
```

Time Complexity

Three loops traverse all the nodes. The innermost loop only has constant complexity operations, so the asymptotic complexity is $O(n_3)$, where n is the number of nodes in the graph.

Advantages of Floyd-Warshall Algorithm

- It's a great algorithm for finding a weighted graph's shortest path where the graph has negative or positive edge weights.
- You only need to execute the algorithm once to find the shortest path length between all the pairs of vertices
- The algorithm can easily be modified and used n path reconstruction
- You can also use a version of Floyd-Warshall to find a transitive closure of a relation R and find the widest path in the weighted graph.

Disadvantages of Floyd-Warshall

- The algorithm can only find the shortest path when the graph doesn't have any negative cycles
- It will not return path details

Applications

- To find the shortest path in directed weighted graphs
- To find the transitive closure in directed weighted graphs
- To find the real matrices inversion
- To see if an undirected graph is bipartite.

Bellman-Ford Algorithm

The Bellman-Ford algorithm is another that finds the shortest path between vertices in weighted graphs.

Let's say we have a graph, and the graph contains a source vertex src. We want to find the shortest path from src to all the other vertices. There may be negative weight edges in this graph.

Dijkstra's algorithm works for this problem, but that is a greedy algorithm with a time complexity of O((V + E)Log V), and it doesn't work on graphs containing negative weight cycles.

The Algorithm

- **Input –** the graph, and src, a source vertex
- **Output –** the shortest distance from src to all other vertices. If the graph has a negative weight cycle, the shortest distances cannot be calculated

Here are the algorithm steps:

Step One – Initialize the distances as infinite between src and all other vertices and the distance to the src as 0. Then an array dist[], size |V| is created with all values set as infinite. The only exception to that is dist[src], where src is the source vertex.

Step Two – Calculate the shortest distances – the following steps should be done |V| - 1 times, where |V| indicates how many vertices the graph contains:

a. Do this for every given edge u-v – if dist[v] > dist[u] + edge uv's weight, dist[v] needs updating – dist[v] = dist[u] + uv edge's weight

Step Three – If there is a negative cycle in the graph, this step reports it. The following should be done for every uv edge:

a. If dist[v] > dist[u] + edge uv's weight, then "graph contains a negative cycle." The idea behind this step is that the second step will guarantee the shortest distances, only where there is no negative weight cycle in the graph. If all the edges are iterated through once more, and a shorter path is obtained for any vertex, the graph has a negative weight cycle.

So, how does this work? In much the same way as any other dynamic programming problem, Bellman-Ford does its shortest distance calculations from the bottom up. First, the distances, where the path has no more than one edge, are calculated. Next, the shortest distances with no more than two

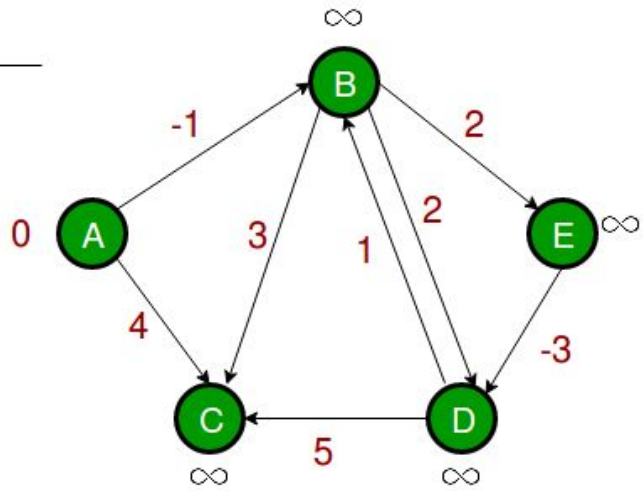
edges on the path are calculated, and so on. The shortest distance with no more than i edges is calculated after the ith iteration. A simple path can have no more than |V| - 1 edges, which is why the outer loop will run |V| – 1 times.

The idea is that assuming the graph does not have a negative weight cycle and the shortest path with no more than i edges, iterating over all the edges is guaranteed to give us the shortest path with no more than (i+1) edges.

Here's an example graph that will help us understand this algorithm a little better.

The given src vertex is 0, and all distances are initialized as 0, except for the source distance. There are five vertices in the graph, which means that all the edges will need to be processed four times.

A	B	C	D	E
0	∞	∞	∞	∞

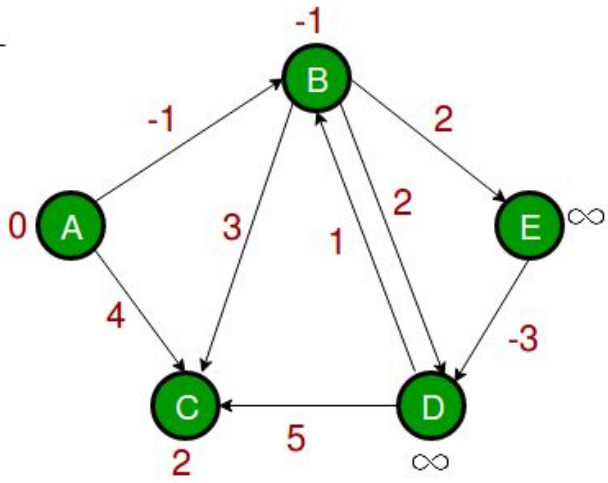


The edges are processed in this order:

(B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D)

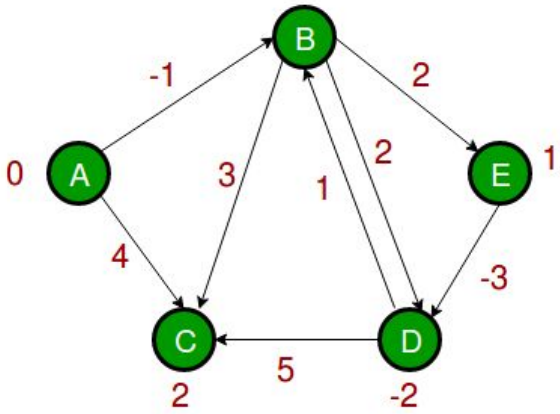
The following distances are obtained after the first processing of the edges. Initial distances are shown in the first row, while the distances in the second row are when the edges (B, E), (D, B), and (A, B) are processed. The third row shows the distances from (A, C), and the final row shows them for (D, C|), (B, C), and (D, E).

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞



The initial iteration will guarantee the shortest distances of the paths, no more than one edge long. When the edges are all processed the second time, we get the distances below – the last row indicates the final values:

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



Iterating a second time guarantees the shortest distances for the paths, no more than two edges long. Bellman-Ford processes all the edges twice more, and then the distances are minimized. That way, the distances will not be updated after the third and fourth iterations.

Here's the implementation in C++:

```
// A C++ program for Bellman-Ford's single source
// shortest path algorithm.
```

```
#include <bits/stdc++.h>

// The structure represents a weighted edge in the graph
struct Edge {
    int src, dest, weight;
};

// The structure represents a connected, directed and
// weighted graph
struct Graph {
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // The graph is represented as an array of edges.
    struct Edge* edge;
};

// This creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}
```

```
// A utility function used to print the solution
void printArr(int dist[], int n)
{
    printf("Vertex   Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

// The main function used to find the shortest distances from src to
// all other vertices using the Bellman-Ford algorithm. The function
// also detects a negative weight cycle in the graph
void BellmanFord(struct Graph* graph, int src)
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];

    // Step 1: Initialize the distances from src to all other vertices
    // as INFINITE
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src] = 0;

    // Step 2: Relax all the edges |V| - 1 times. A simple shortest
```

```
    // path from src to any other vertex can have at-most |V| - 1
    // edges
    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
             if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }

     // Step 3: check for negative-weight cycles.  The above step
     // guarantees the shortest distances if the graph doesn't contain
    // a negative weight cycle.  If we get a shorter path, then there
    // is a negative weight cycle.
    for (int i = 0; i < E; i++) {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
          if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
            printf("contains a negative weight cycle");
```

```
            return; // If a negative cycle is detected, simply
return
        }
    }

    printArr(dist, V);

    return;
}

// Driver program to test above functions
int main()
{
    /* Let us create the graph given in the above example
*/
    int V = 5; // Number of vertices in graph
    int E = 8; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1 (or A-B in above figure)
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = -1;

    // add edge 0-2 (or A-C in above figure)
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 4;
```

```
// add edge 1-2 (or B-C in above figure)
graph->edge[2].src = 1;
graph->edge[2].dest = 2;
graph->edge[2].weight = 3;

// add edge 1-3 (or B-D in above figure)
graph->edge[3].src = 1;
graph->edge[3].dest = 3;
graph->edge[3].weight = 2;

// add edge 1-4 (or A-E in above figure)
graph->edge[4].src = 1;
graph->edge[4].dest = 4;
graph->edge[4].weight = 2;

// add edge 3-2 (or D-C in above figure)
graph->edge[5].src = 3;
graph->edge[5].dest = 2;
graph->edge[5].weight = 5;

// add edge 3-1 (or D-B in above figure)
graph->edge[6].src = 3;
graph->edge[6].dest = 1;
graph->edge[6].weight = 1;

// add edge 4-3 (or E-D in above figure)
```

```
    graph->edge[7].src = 4;
    graph->edge[7].dest = 3;
    graph->edge[7].weight = -3;

    BellmanFord(graph, 0);

    return 0;
}
```

Output:

Vertex	Distance from Source
0	0
1	-1
2	2
3	-2
4	1

Note

1. You will find negative cycles in many different graph applications. For example, we might get more advantage following a path rather than paying cost for it.
2. The Bellman-Ford algorithm works better than some others, including Dijkstra's, for distributed systems. In Dijkstra's, we have to find the minimum value of every vertex but, in Bellman-Ford, we consider each edge individually.
3. Bellman-Ford won't work with undirected graphs with negative edges. This is because they are declared negative cycles.

That completes our look at dynamic programming techniques so, now, we can turn our attention to the branch and bound technique.

Chapter 5: Branch and Bound

Images courtesy of https://geeksforgeeks.org

The branch and bound technique is a design paradigm typically used to solve problems of a combinatorial optimization nature. These problems tend to be exponential as far as time complexity is concerned, and you may need to explore every permutation in the worst-case. Branch and bound can be used to solve problems like this quickly.

We'll consider the 0/1 knapsack problem to help us understand branch and bound. The knapsack problem is solvable by a number of algorithms, including:

- Dynamic programming for 0/1 knapsack
- Greedy algorithm for fractional knapsack
- Backtracking solution for 0/1 knapsack

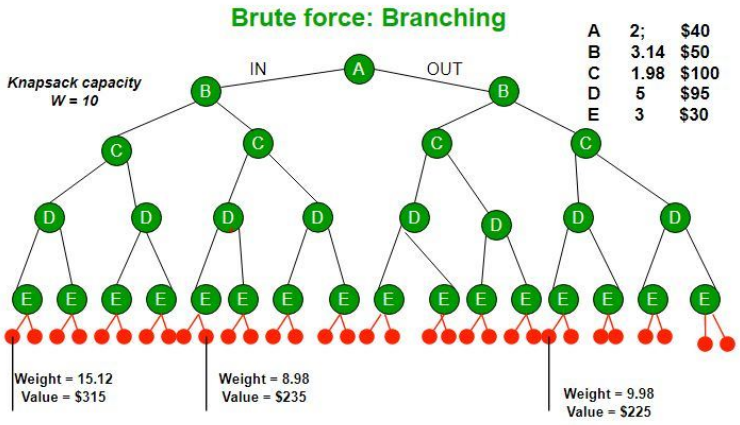
Knapsack Using Branch and Bound

Let's look at the following 0/1 Knapsack problem to understand Branch and Bound.

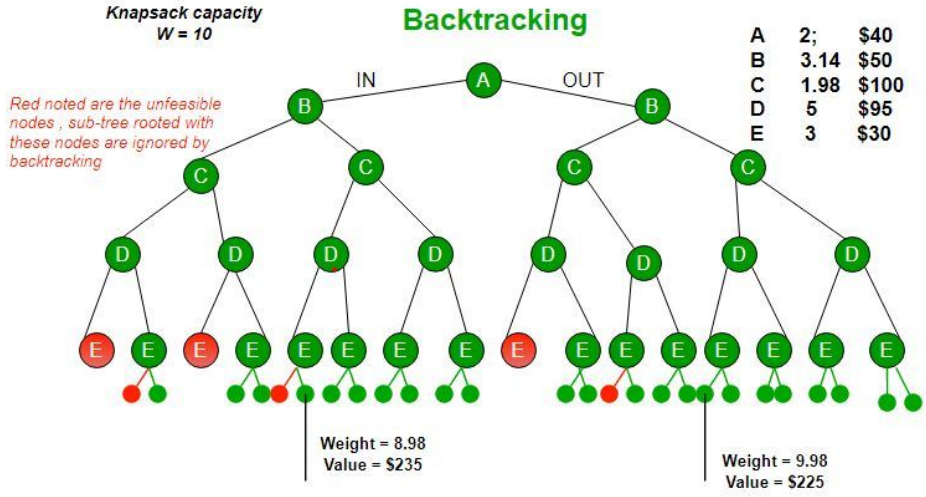
We have two integer arrays, val[0…n-1] and wt[0…n-1]. These arrays represent values and weights associated with n items. We want to find the maximum value subset of val[], where the subset's sum of weights is equal to or smaller than W, which is the Knapsack capacity.

There are several ways we can approach this problem:

1. **Greedy Approach –** this technique will choose the items in decreasing order of the value-per-unit weight. This approach will only work for the fractional knapsack problem and may not give us the right answer for 0/1 knapsack.
2. **Dynamic Programming -** uses a 2D table with a size of n x W. This will not work where the item weights are not integers.
3. **Brute Force –** the above solution will not always work, but you can use brute force. 2_n solutions can be generated where you have n items, and each is checked to ensure they satisfy the constraint before the maximum solution satisfying the constraint is saved. We can express this solution as tree:



The brute force solution can be optimized by backtracking. We can do a depth-first search on the tree and, if we get to a point where a solution isn't feasible any longer, we don't need to explore any further. In our example, it would be more effective to use backtracking if the knapsack capacity were smaller or we had more items.



Branch and Bound

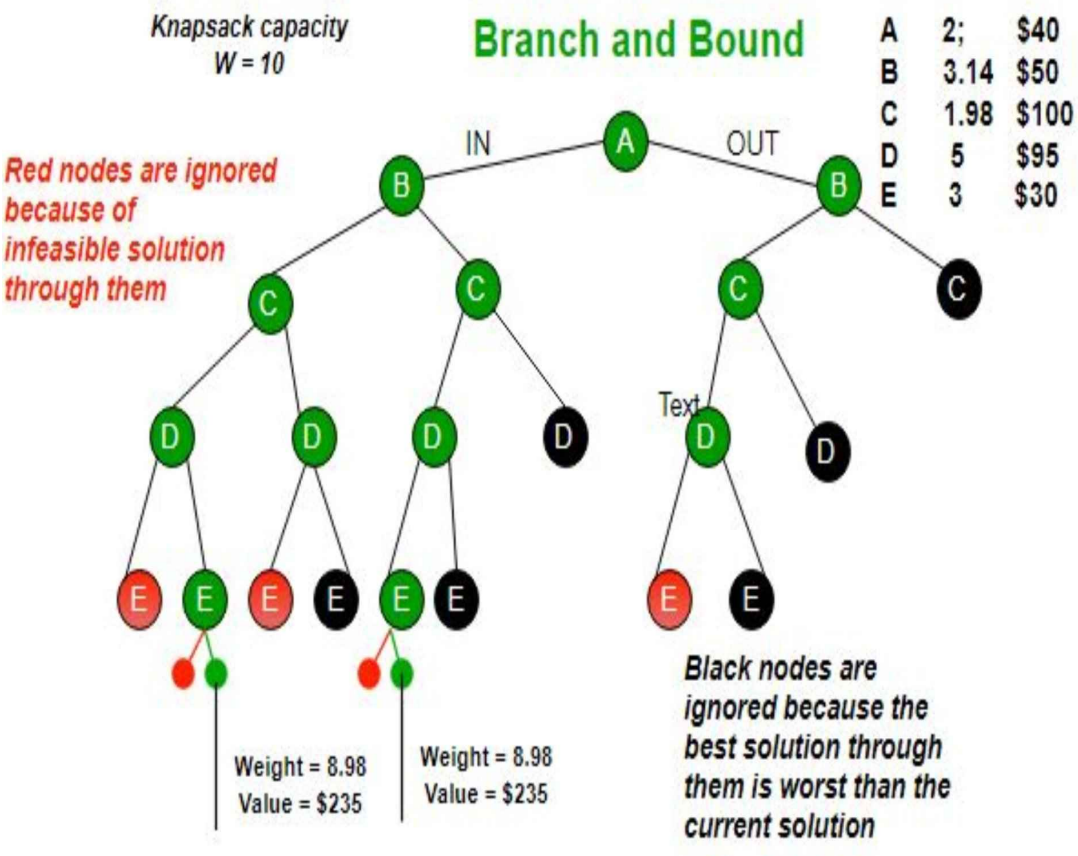
We can optimize the backtracking solution if we know a bound on the best-possible subtree that is rooted with all nodes. However, the node and subtrees can be ignored if the subtree's best is worse than the current best. Bound (the best solution) is computed for each node and compared with the current bound before the node is explored.

In the diagram below, the example bounds are:

- **A down –** can provide $315
- **B down –** can provide $275
- **C down –** can provide $225

- **D down –** can provide $125
- **E down -** can provide $30

We'll discuss how to get these bounds shortly:



While the branch and bounds technique is useful to search for a solution, the entire tree must be fully calculated in the worst-case scenario. In the best case, only one path must be calculated fully through the tree, while the rest can be pruned.

Implementation

Here is the implementation of the 0/1 knapsack problem using the branch and bound technique. We looked at several techniques and determined that branch and bound work the best when the item weights are not integers.

We want to find the bound for all the nodes in the 0/1 knapsack problem, using the fact that the greedy approach works best on the fractional knapsack problem.

The optimal solution is computed through the node, using the greedy approach to find out if a specific node provides a better solution. If

the solution is better than the current best, there is no way to get an even better solution through the node.

Here's the algorithm:

1. All the items are sorted in decreasing order of ratio of value per unit weight. This way, we can compute the upper bound using the greedy approach.
2. The maximum profit maxProfit = 0 is initialized
3. An empty queue, Q, is created
4. A dummy node of the decision tree is created and enqueued to Q. The dummy node's profit and weight are 0.
5. While Q isn't empty, do the following:
    a. Extract an item, u, from Q
    b. Compute the next level node's profit. If it is greater than maxProfit, maxProfit must be updated
    c. Compute the next level node's bound. If it is greater than maxProfit, the next level node must be added to Q
    d. Consider a case where the next level node is not included in the solution – a node should be added to the queue with the level set as next but without considering the weight and profit.

Here's an illustration:

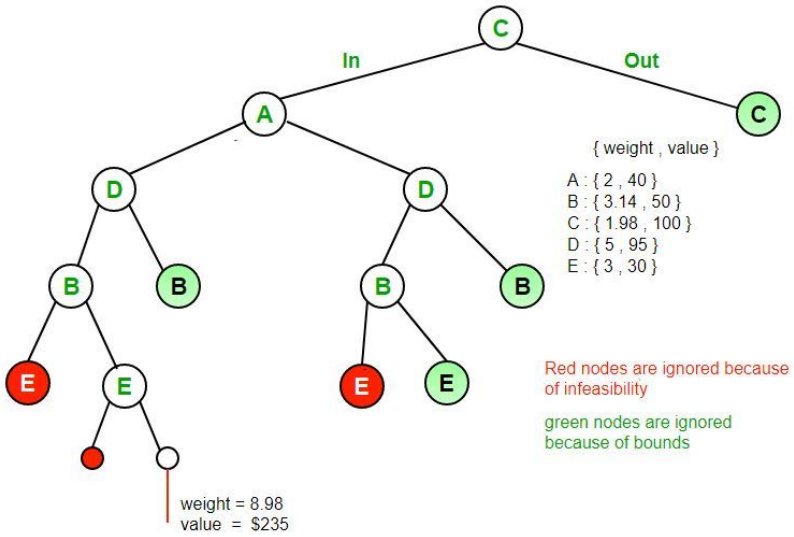
Input:

```
// In every pair, the first thing is the weight of the item
// and the second thing is the value of item
Item arr[] = {{2, 40}, {3.14, 50}, {1.98, 100},
              {5, 95}, {3, 30}};
Knapsack Capacity W = 10
```

Output :

```
The maximum possible profit = 235
```

You can see this in the diagram below, where the items are considered to be sorted by value/weight.



Note:

This image doesn't follow the algorithm strictly because there is no dummy node, but it does give you an idea. Here's a C++ implementation of the above:

```
// C++ program to solve knapsack problem using
// branch and bound
#include <bits/stdc++.h>
using namespace std;


// The structure for the Item which stores the weight and the corresponding
// value of Item
struct Item
{
    float weight;
    int value;
};


// Node structure that stores the information of the decision
// tree
struct Node
```

```
{
    // level  —> Level of the node in the decision tree (or index
    //            in arr[]
    // profit —> Profit of the nodes on the path from the root to this
    //           node (including this node)
    // bound –> The upper bound of the maximum profit in the subtree
    //           of this node/
    int level, profit, bound;
    float weight;
};

// A comparison function to sort Item according to
// val/weight ratio
bool cmp(Item a, Item b)
{
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
}

// Returns the bound of profit in the subtree rooted with u.
// This function mainly uses the Greedy solution to find
// an upper bound on maximum profit.
int bound(Node u, int n, int W, Item arr[])
{
    // if weight overcomes the knapsack capacity, return
    // 0 as expected bound
    if (u.weight >= W)
```

```
        return 0;

    // initialize the bound on profit by current profit
    int profit_bound = u.profit;

    // start including items from index 1 more to current
    // item index
    int j = u.level + 1;
    int totweight = u.weight;

    // checking the index condition and knapsack capacity
    // condition
    while ((j < n) && (totweight + arr[j].weight <= W))
    {
        totweight    += arr[j].weight;
        profit_bound += arr[j].value;
        j++;
    }

    // If k is not n, include last item partially for
    // upper bound on profit
    if (j < n)
        profit_bound += (W - totweight) * arr[j].value /
                                    arr[j].weight;

    return profit_bound;
}

// Returns the maximum profit we can get with capacity W
```

```
int knapsack(int W, Item arr[], int n)
{
    // sorting Item on basis of value per unit
    // weight.
    sort(arr, arr + n, cmp);

    // make a queue for traversing the node
    queue<Node> Q;
    Node u, v;

    // dummy node at starting
    u.level = -1;
    u.profit = u.weight = 0;
    Q.push(u);

    // Extract the items one by one from the decision tree
     // and compute the profit of all children of the extracted item
    // and keep saving maxProfit
    int maxProfit = 0;
    while (!Q.empty())
    {
        // Dequeue a node
        u = Q.front();
        Q.pop();

        // If it is the starting node, assign level 0
        if (u.level == -1)
            v.level = 0;
```

```
// If there is nothing on the next level
if (u.level == n-1)
    continue;

// Else if not last node, then increment level,
// and compute profit of children nodes.
v.level = u.level + 1;

// Taking the current level's item add current
// level's weight and value to node u's
// weight and value
v.weight = u.weight + arr[v.level].weight;
v.profit = u.profit + arr[v.level].value;

// If the cumulated weight is less than W and
// profit is greater than previous profit,
// update maxProfit
if (v.weight <= W && v.profit > maxProfit)
    maxProfit = v.profit;

// Get the upper bound on profit to decide
// whether to add v to Q or not.
v.bound = bound(v, n, W, arr);

// If the bound value is greater than profit,
// then only push into the queue for further
// consideration
if (v.bound > maxProfit)
    Q.push(v);
```

```
            // Do the same thing,  but without taking
            // the item in knapsack
            v.weight = u.weight;
            v.profit = u.profit;
            v.bound = bound(v, n, W, arr);
            if (v.bound > maxProfit)
                Q.push(v);
        }

        return maxProfit;
    }

    // driver program to test above function
    int main()
    {
        int W = 10;   // Weight of knapsack
        Item arr[] = {{2, 40}, {3.14, 50}, {1.98, 100},
                    {5, 95}, {3, 30}};
        int n = sizeof(arr) / sizeof(arr[0]);

        cout << "Maximum possible profit = "
            << knapsack(W, arr, n);

        return 0;
    }
```

Output:

```
Maximum possible profit = 235
```

Using Branch and Bound to Generate Binary Strings of Length N

Examples:

The input is N, and the output is:

000
001
010
011
100
101
110
111

Explanation:

Numbers that have three binary digits are 0, 1, 2, 3, 4, 5, 6, 7.

The input is N, and the output is:

00
01
10
11

Approach :

Use branch and bound to generate the combinations:

- We have an empty solution vector to start with
- While Queue isn't empty, the partial vector must be removed from the queue
- If this is the last vector, the combination is printed, else
- For the next partial vector component, k child vectors are created by fixing all the possible states for that component, and the vectors are inserted into the queue

Here is the implementation of this approach:

```
// CPP Program to generate

// Binary Strings using Branch and Bound

#include <bits/stdc++.h>

using namespace std;
```

```
// Creating a Node class
class Node
{
public:
    int *soln;
    int level;
    vector<Node *> child;
    Node *parent;

    Node(Node *parent, int level, int N)
    {
        this->parent = parent;
        this->level = level;
        this->soln = new int[N];
    }
};

// A utility function to generate the binary strings of length n
void generate(Node *n, int &N, queue<Node *> &Q)
{
    // If the list is full, print combination
    if (n->level == N)
    {
        for (int i = 0; i < N; i++)
            cout << n->soln[i];
        cout << endl;
    }
    else
```

```
    {
        int l = n->level;

        // iterate while length is not equal to n
        for (int i = 0; i <= 1; i++)
        {
            Node *x = new Node(n, l + 1, N);
            for (int k = 0; k < l; k++)
                x->soln[k] = n->soln[k];
            x->soln[l] = i;
            n->child.push_back(x);
            Q.push(x);
        }
    }
}

// Driver Code
int main()
{
    // Initiate Generation
    // Create a root Node
    int N = 3;
    Node *root;
    root = new Node(NULL, 0, N);

    // Queue that maintains the list of live Nodes
    queue<Node *> Q;

    // Instantiate the Queue
```

```
        Q.push(root);

        while (!Q.empty())
        {
            Node *E = Q.front();
            Q.pop();
            generate(E, N, Q);
        }

        return 0;
    }
```

Output:

```
000
001
010
011
100
101
110
111
```

This approach has a time complexity of $O(2^n)$

Chapter 6: Randomized Algorithm

Images courtesy of https://geeksforgeeks.org

A randomized algorithm makes a decision in its logic using random numbers. For example, the randomized Quicksort algorithm chooses the next pivot using a random number, or the array is randomly shuffled. This randomness helps reduce the time or space complexity in standard algorithms.

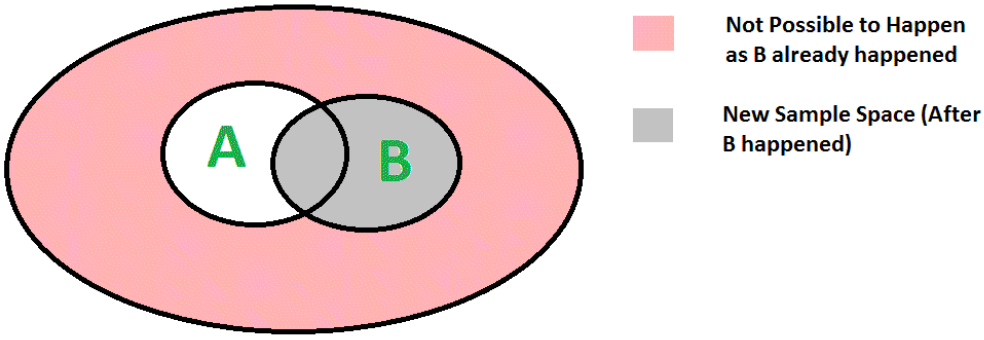
Let's see how it all works:

Conditional Probability

Conditional probability P(A | B) indicates a probability of A happening, given that B happens.

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}$$

The diagram below explains this formula. B has happened already, so the sample space is reduced to B. That means the probability that A will happen is P(A ⋂ B) divided by P(B).



Below, you can see Bayes' conditional probability formula:

$$P(A|B) = \frac{PB|A)P(A)}{P(B)}$$

This formula tells us the relationship that exists between P(A|B) and P(B|A).

Have a look at the following conditional probability formulas P(A|B) and P(B|A):

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)} \quad \text{- - - 1}$$

$$P(B \mid A) = \frac{P(B \cap A)}{P(A)} \quad \text{- - - 2}$$

Because $P(B \cap A) = P(A \cap B)$, $P(A \cap B)$ can be replaced with $P(B|A)P(A)$ in the first formula, we get the given formula.

Random Variables

Random variables are functions that map random event outcomes to real values. Here's an example of a coin-tossing game.

If a coin toss results in "Heads," a player will pay $50. If the result is "Tails," they are paid $50.

We can define a random variable profit for the person as:

Profit = +50 if Heads

-50 if Tails

Typically, players will not find gambling games fair because the organizer will always take a share of the profits for every event. So, normally, the expected profit will be positive for the organizer and negative for the player, and that's how an event organizer makes their money.

We can define a random variable's expected value like this:

$E[R] = r1*p1 + r2*p2 + \ldots rk*pk$

ri ==> the value of R with a probability of pi

Therefore, the expected value is the sum of the product of the two terms below, for every possible event:

a. The probability of an event
b. The value of R such that even:

Example 1:

In the above coin toss example

Expected value of profit = 50 * (1/2) +

(-50) * (1/2)

= 0

Example 2:

The expected value for a six-faced dice throw is

= 1*(1/6) + 2*(1/6) + …. + 6*(1/6)

= 3.5

The Linearity of the Expectation:

Here R_1 and R_2 are discrete random variables on a probability space, so:

$E[R_1 + R_2] = E[R_1] + E[R_2]$

Let's say that the sum of three dice throws gives us an expected value of 3*7/2=7

How Many Trials Are Expected Until Success?

If p is the probability of success in every trial, the expected number of trials to succeed would be 1/p. For example, let's say we have a 6-faced dice, and we throw it until we get a 5. We would expect to throw six times to get a five, with a probability of 1/6, so the number of expected trials is 1/(1/6) = 6.

Let's consider a Quicksort version that looks for pivots until one of the n/2 elements in the middle is chosen. The expected number of trials is 2 because the probability of choosing one of the n/2 elements in the middle is 1 / 2.

How to Analyze Randomized Algorithms

The time complexity for some randomized algorithms is deterministic, and these are known as Monte Carlo algorithms. In terms of the worst-case, these are far easier to analyze. Other algorithms have a time complexity that depends on a random variable's value, and these are called Las Vegas

algorithms. Typically, these are analyzed for the expected worst-case. To compute the expected time in the worst-case, we need to consider every possible value of the used random variable in the worst case and the time taken by each value. The average of the times becomes the time complexity for worst-case.

Here's an example of a randomized quicksort algorithm:

Central pivots divide arrays so that one side has a minimum of 1 / 4 elements:

```
// Sorts an array arr[low..high]
randQuickSort(arr[], low, high)
```

1 . If low >= high, then EXIT.

2 . While pivot 'x' is not a Central Pivot.

(i) Choose a random number uniformly from [low..high].

Let the randomly picked number be x .

(ii) Count the elements in arr[low..high] smaller than arr[x]. Let this count be sc .

(iii) Count the elements in arr[low..high] greater than arr[x]. Let this count be gc.

(iv) Let n = (high-low+1). If sc >= n/4 and gc >= n/4, then x is a central pivot.

3 . Partition arr[low..high] around the pivot x.

4 . // Recur for smaller elements

```
randQuickSort(arr, low, sc-1)
```

5 . // Recur for greater elements

randQuickSort(arr, high-gc+1, high)

The important thing to take away from this is that step two has O(n) time complexity.

So, how many times does the while loop need to run before it finds a central pivot? There is a 1/n probability that the element chosen randomly is a central pivot. That means the loop is expected to run n times, which leads to the expected time complexity of O(n) for step two.

In the worst-case, the array is divided by each partition so that one side has n/4 elements and the other has 3n/4 elements. In the worst-case scenario, the recursion tree's height is Log 3/4nm, equating to O(Log n).

$T(n) < T(n/4) + T(3n/4) + O(n)$

$T(n) < 2T(3n/4) + O(n)$

The recurrence above has a solution of O(Log n).

It's worth bearing in mind that the randomized algorithm above isn't the best method for randomized Quicksort. All I wanted to do was simplify the analysis. Typically, we implement randomized Quicksort by randomly choosing a pivot with no loop or shuffling the array elements. This algorithm has an expected time complexity of O(n Log n), and the analysis is more complicated than we will go into here.

Randomized Algorithms – Classification and Application

Classification

Typically, randomized algorithms fall into two classifications, as we briefly explained in the last section:

- **Las Vegas** – algorithms in this classification always produce an optimum or correct result. They have a time complexity based on random values, and it is usually evaluated as the expected value. Take the randomized

Quicksort algorithm, for example. It will always sort input arrays, and Quicksort has an O(n Log n) time complexity in the worst-case scenario.

- **Monte Carlo** – algorithms in this classification always produce optimum or correct results with a degree of probability. They have a deterministic running time, and working out the worst-case time complexity is typically easier. Take Karger's algorithm, for example. Some implementations produce a minimum cut with a probability equal to or greater than $1/n^2$, where n indicates the number of vertices, and time complexity is O (E).

Understanding Classification

Let's take a binary array where exactly 50% of its elements are 0, and the rest are 1. We want to find the index of any of the 1's.

If we used a Las Vegas algorithm, it would choose random elements until it finds a 1. Conversely, a Monte Carlo algorithm would choose random elements until it finds a 1 or the maximum allowable times (k) to choose a random element have been used. The Las Vegas algorithm will always find an index of 1, but the expected value will always determine the time complexity. The expected number of trials to succeed is 2, so O(1) is the expected time complexity.

The Solovay-Strassen Primality Test

Primality testers are algorithms that look at whether a specified integer is prime. This is one of the more common problems, especially in cryptography and cyber security, and lots of other different applications where the special properties in prime numbers are relied on.

Volker Strassen and Robert M Solovay developed this probabilistic test to see if a number is prime or composite. Before we look at the algorithm, we need to talk about some modular arithmetic terms and mathematics.

Key Concepts

- **Legendre symbol (a/p):**

This function is defined on two integers, a and p, where:

(a/p) = 0; if a≡0(mod p)
(a/p) = 1; if k exists such that k^2 = a mod p exists
(a/p) = -1; otherwise

The function follows Euler's criterion.

- **Jacobi symbol (a/n):**

This is Legendre's symbol generalization. For an integer a and a positive odd integer n, the Jacobi symbol (a/n) is the product of the symbols that correspond to n's prime factors:

$$(a/n) = (a/p_1)^{k_1}.(a/p_2)^{k_2}.(a/p_3)^{k_3} \ldots\ldots\ldots..(a/p_n)^{k_n}$$

Where $n = p_1^{k_1}.p_2^{k_2} \ldots\ldots p_n^{k_n}$

The Jacobi symbol has some useful properties for determining if a number is prime. For an odd prime, (a/n) = (a/p).

A number's Jacobi symbol may be computed in time complexity of $O(\text{Log } n)_2$ time.

Algorithm

This is how the test works:

1. First, a number n is selected to test for primality, and a random number a, from the (2, n-1) range. The Jacobi symbol (a/n) is then computed
2. If n is prime, the Jacobi symbol is equal to the Legendre, and this satisfies Euler's criterion
3. If the given condition is not satisfied, n is considered composite, and the program stops
4. Like most probabilistic primality tests, the accuracy and number of iterations are directly proportionate to one another. More accurate results can be gained by running the test several times.

Random number a is known as a Euler witness for the compositeness of n. When we use a fast algorithm for modular

exponentiation, we get a running time of $O(k.\log^3 n)$, where k indicates how many different values in a that we test.

Below you can see the Solvay-Strassen Primality test pseudocode:

```
# check if the integer p is prime or not, run for k iterations
boolean SolovayStrassen(double p, int k):
    if(p<2):
        return false
    if(p!=2 and p%2 == 0):
        return false

    for(int i = 0; i<k; i++):
        double a = rand() % (p-1) + 1
        # Jacobi() calculates the Jacobi symbol (a/n)
        # modExp() performs modular exponentiation
        double jacobian = (p + Jacobi(a, p)) % p
        double mod = modExp(a, (p-1)/2, p)

        if(!jacobian || mod != jacobian)
            return false
    return true
```

By their nature, a randomized algorithm uses randomness in its logic. They can make random choices, which allows some problems to be solved quicker than if they were solved deterministically.

Are properties such as randomness used in machine learning? Yes, they are. Many machine learning applications use randomness, for example:

- Algorithms can be initialized to random states. One example is the initial weights in ANNs – artificial neural networks
- Internal decisions in deterministic methods sometimes resolve ties using randomness
- When data is split randomly into training and testing data sets
- When a training dataset is randomly shuffled in stochastic gradient descent

All this makes randomness requires, be it random number generation or harnessing randomness. If you have worked with machine learning models, you may have spotted that you get different results whenever you run them. This is down to the degree of randomness and random behavior in ranges, which is known as stochastic behavior – most machine learning models are stochastic by their nature.

The Monte Carlo algorithms are fast and almost likely correct, while Las Vegas algorithms are not always fast but are always correct.

Applications and Scope:

- Let's say we have a tool used for sorting, and many users make use of this tool, while a few use it in arrays that have already been sorted. Lat's say that tool uses simple Quicksort, not randomized; in that case, those few users will always come up against the worst-case scenario. Conversely, if it used randomized Quicksort, no user will always face the worst-case, and everyone will get the time complexity of O(n Log n).
- Randomized algorithms work very well in cryptography
- They work well in load balancing
- They are used for primality testing and other number-theoretic applications
- They are used in data structures, such as sorting, hashing, order statistics, searching, and computational geometry
- They are used in algebraic identities, such as interactive proof systems, matrix, and polynomial identity verification

- They are used in counting and enumeration, such as combinatorial structures and matrix permanent counting
- They are used in graph algorithms, such as shortest paths, minimum spanning trees, and minimum cuts
- They are used in parallel and distributed computing, such as deadlock avoidance distributed consensus
- They are used in probabilistic existence proofs, i.e., to show that, when objects are drawn from suitable probability spaces, a combinatorial object will also arise with a non-zero probability
- They are also used in derandomization, i.e., a randomized algorithm is devised and then it can be argued that it can yield a deterministic algorithm by being derandomized.

Time to move on to our final chapter, a discussion on recursion and backtracking.

Chapter 7: Recursion and Backtracking

In our final chapter, we'll look at the final two techniques – recursion and backtracking.

Recursion

Recursion is a process in which a function directly or indirectly calls itself. The function corresponding to this is known as a recursive function. Recursive algorithms are used to easily solve certain problems, such as the TOH (Towers of Hanoi), DFS of Graph, Inorder/Preorder/Postorder Tree Traversals, and more.

Mathematical Interpretation

Let's look at an example of a problem – a programmer needs to determine the sum of the first n natural numbers, and there are several approaches they can choose. The simplest one is nothing more than adding the numbers 1 to n. The function would look like this:

approach(1) – Adding one by one

f(n) = 1 + 2 + 3 +……..+ n

However, we could representing this using a different mathematical approach:

approach(2) – Recursive adding

f(n) = 1 n=1

f(n) = n + f(n-1) n>1

The difference between these two approaches is quite simple. In the second approach, we call the function f() inside the function, which is called recursion. The function that has the function being called is known as the recursive function, and this is a great tool for programmers to use for more efficient coding of certain problems.

What Is the Base Condition?

The base case solution is provided in a recursive program, and smaller programs are used to express the bigger

problem's solution.

```
int fact(int n)
{
    if (n < = 1) // base case
        return 1;
    else
        return n*fact(n-1);
}
```

In this example, we have defined the n < = 1 base case, and we can solve the number's larger value by converting it to a smaller one until we reach the base case.

One question that gets asked is how does recursion work to solve a problem? The idea is that a problem is presented as at least one smaller problem, and base conditions are added to stop the recursion. For example, factorial n can be computed if we know (n-1)s factorial. In that case, the factorial's base case would be n = 0, and when n = 0, we return 1.

The next question is, why do we get stack overflow errors in recursion? If we do not define the base case or we cannot reach it, the stack overflow problem may occur. Here's an example to help us understand this:

```
int fact(int n)
{
    // wrong base case (it can cause
    // stack overflow).
    if (n == 100)
        return 1;

    else
        return n*fact(n-1);
```

```
}
```

If we call fact(10), then fact(9), fact(), fact(7), etc., will also be called, but that number will not get to 100. That means we do not reach the base case. A stack overflow error arises if these functions on the stack exhaust the memory.

You might also wonder about the difference between direct recursion and indirect. A function, fun, is directly recursive if it calls the same function fun. However, it becomes indirectly recursive if it calls a different function, perhaps fun_new, and this function then directly or indirectly calls function fun.

Here's an example:

```
// An example of direct recursion
void directRecFun()
{
    // Some code….

    directRecFun();

    // Some code…
}

// An example of indirect recursion
void indirectRecFun1()
{
    // Some code…

  indirectRecFun2();

    // Some code…
```

```
}
void indirectRecFun2()
{
    // Some code…

    indirectRecFun1();

    // Some code…
}
```

How Is Memory Allocated to Function Calls?

When you call a function from main(), memory is allocated on the stack. When a recursive function calls itself, the called function's memory gets allocated on the top of the memory that was allocated to the calling function. Then the local variables are copied, one for each function call. When we reach the base case, the value is returned from the function to the calling function, the memory gets deallocated, and the process goes on.

Let's use a simple function to see how recursion works:

```
// A C++ program to demonstrate working of
// recursion
#include <bits/stdc++.h>
using namespace std;

void printFun(int test)
{
    if (test < 1)
        return;
    else {
```

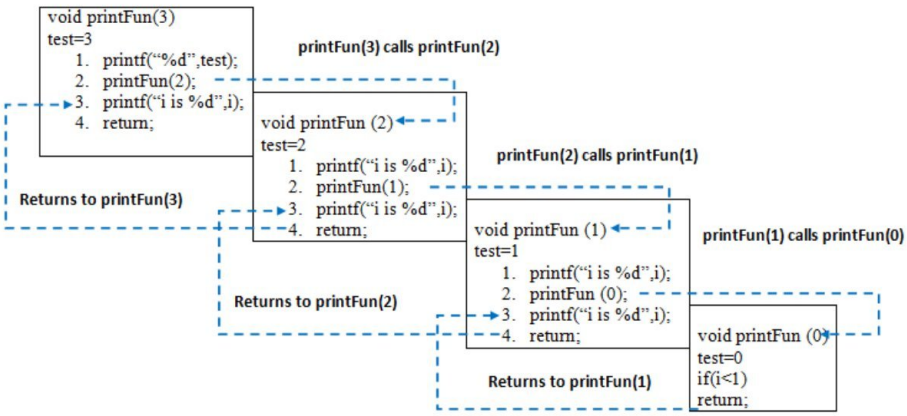
```
        cout << test << " ";
        printFun(test - 1); // statement 2
        cout << test << " ";
        return;
    }
}

// Driver Code
int main()
{
    int test = 3;
  printFun(test);
}
```

Output:

3 2 1 1 2 3

Let's break this down a little.

When we call printFun(3) from main(), the memory is allocated to it. Then, we initialize a local variable test to 3 and push statements 1 to 4 onto the stack, as you can see in the diagram below.



First, 3 is printed. Then, printFun(2) is called in statement 2, and memory is allocated to it. We initialize the local variable test to 2 and push statements 1 to 4 in the stack. printFun(2) will called printFun(1) which calls printFun(o). The latter goes to the if statement and returns to printFun(1).

The remaining printFun(1) statements are then executed, printFun(2) is called, and so on.

The output prints the values 3 to 1 and then 1 to 3. You can see the memory stack in the above diagram.

Now we can look at some practical problems recursion can solve and understand how it works.

Problem 1

Write a program and the recurrence relation that finds n's Fibonacci sequence, where n>2.

Mathematical Equation:

n if n == 0, n == 1;

fib(n) = fib(n-1) + fib(n-2) otherwise;

Recurrence Relation:

$$T(n) = T(n-1) + T(n-2) + O(1)$$

Here's the recursive problem implementation:

- **Input** : n = 5
- **Output** :

A Fibonacci series of 5 numbers is : 0 1 1 2 3

```
// C++ code to implement Fibonacci series
#include <bits/stdc++.h>
using namespace std;

// Function for Fibonacci

int fib(int n)
```

```
{
    // Stop condition
    if (n == 0)
        return 0;

    // Stop condition
    if (n == 1 || n == 2)
        return 1;

    // Recursion function
    else
        return (fib(n - 1) + fib(n - 2));
}

// Driver Code
int main()
{
    // Initialize variable n.
    int n = 5;
    cout<<“Fibonacci series of 5 numbers is: “;

    // for loop to print the Fibonacci series.
    for (int i = 0; i < n; i++)
    {
        cout<<fib(i)<<” “;
    }
    return 0;
```

}

Output:

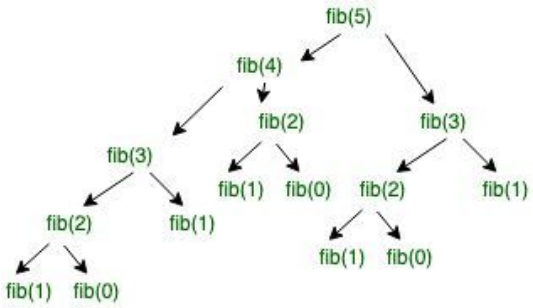
Fibonacci series of 5 numbers is: 0 1 1 2 3

Below, you can see input 5's recursive tree, showing clearly how we can use smaller problems to solve a larger one. The Fibonacci function is fib(n), and the time complexity is dependent on the function call.

fib(n) -> level CBT (UB) -> 2^n-1 nodes -> 2^n function call -> 2^n*O(1) -> T(n) = O(2^n)

For the best case, the time complexity is:

T(n) = θ(2^n\2)



Problem 2

Write a program and the recurrence relation to find n's factorial, where n>2.

Mathematical Equation:

1 if n == 0 or n == 1;

f(n) = n*f(n-1) if n> 1;

Recurrence Relation:

T(n) = 1 for n = 0

T(n) = 1 + T(n-1) for n > 0

Here's the implementation:

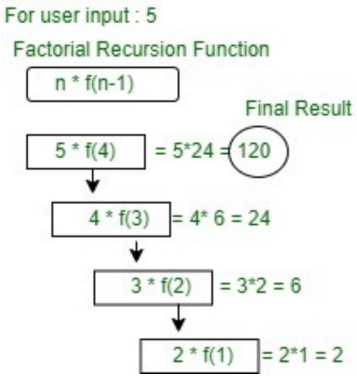
- **Input –** n=5
- **Output –**

Factorial of 5 is:

```
// C++ code to implement factorial
#include <bits/stdc++.h>
using namespace std;

// Factorial function
int f(int n)
{
    // Stop condition
    if (n == 0 || n == 1)
        return 1;

    // Recursive condition
    else
        return n * f(n - 1);
}

// Driver code
int main()
{
    int n = 5;
    cout<<“factorial of “<<n<<” is: “<<f(n);
    return 0;
}
```

Output:

factorial of 5 is: 120



Advantages and Disadvantages of Recursive Programming vs. Iterative Programming

Iterative and recursive programs both share the same powers for solving problems. That means we can write recursive problems iteratively and iterative programs recursively. Recursive programs require more space than iterative programs because the functions will all stay in the stack until we reach the base case. Recursive programming also has greater requirements in terms of space because of the return overheads and function calls.

However, recursion does provide a cleaner and easier way of writing code. Towe of Hanoi, tree traversals, and some other problems are inherently recursive, and it is best to write recursive code for them. We can also use a stack data structure to write these codes iteratively.

Backtracking

Backtracking is a technique used to recursively solve problems by incrementally building a solution. This is done one piece at a time, and the solutions that don't satisfy the problem's constraints at any point of time are removed. By time, we are referring to how much time passes until any level of the search tree is reached.

It is defined as a technique that solves a computational problem by considering a search on all possible combinations, and there are three problem types in backtracking:

- **Decision Problems –** we look for feasible solutions

- **Optimization Problems –** we look for the best solution
- **Enumeration Problems –** we find every feasible solution

How Do We Know if Backtracking Can Be Used?

Typically, where a constraint satisfaction problem has well-defined, clear constraints on an objective solution, where candidates are incrementally built to the solution, and it backtracks or abandons the candidate once it is determined that the candidate cannot complete and provide a valid solution, backtracking can be used to solve it.

However, we can solve most problems using greedy or dynamic programming algorithms. This way, they are solved in linear, logarithmic, linear-logarithmic time complexity in input size order. Therefore they outclass backtracking in just about every respect, especially because backtracking algorithms are typically exponential in space and time. However, there are still some problems that can only be solved by backtracking.

Let's consider an example. We have three boxes, and there is a gold coin in one of them – which one, we don't know at this stage. So, to find the coin, each box needs to be opened one at a time. If the first box doesn't have the coin in it, you close it and look in the second box. This continues until the coin is found. That is backtracking in its simplest form, reaching the best solution by solving the subproblems one at a time.

Now let's look at a more formal example to help us understand backtracking.

Let's say we have a computational problem, P, and data, D, that corresponds to the problem. Solving the problem requires that certain restraints be satisfied, which are represented by C. The backtracking algorithm works like this:

The algorithm starts building the solution, beginning with S, the empty solution set – S = {}.

1. The first move left is added to S – all moves are added one at a time. A new subtree, S, is now created in the algorithm's subtree
2. Make sure S satisfies the three C constraints:
    a. If yes, subtree S can add more children
    b. Else, subtree S is useless. Use argument S to recur back to step one
3. If the new subtree is eligible to add children, argument S + $_s$ is used to recur to step one
4. If a check for S + $_s$ confirms it is the solution for D, the program is output and terminated. If not, it is discarded.

Backtracking vs. Recursion

Recursion requires the function to call itself until the base case is reached. In backtracking, recursion is used to explore every possibility until the best solution is found.

Here is the pseudocode for backtracking:

1. **Recursive backtracking solution.**

```
void findSolutions(n, other params) :
    if (found a solution) :
        solutionsFound = solutionsFound + 1;
        displaySolution();
        if (solutionsFound >= solutionTarget) :
            System.exit(0);
        return

    for (val = first to last) :
        if (isValid(val, n)) :
            applyValue(val, n);
            findSolutions(n+1, other params);
```

```
            removeValue(val, n);
```

2. **Determining if a solution exists**

```
boolean findSolutions(n, other params) :
    if (found a solution) :
        displaySolution();
        return true;

    for (val = first to last) :
        if (isValid(val, n)) :
            applyValue(val, n);
            if (findSolutions(n+1, other params))
                return true;
            removeValue(val, n);
        return false;
```

The Knight's Tour Problem

We now know that backtracking is an algorithmic technique for finding the right solution by trying all possible solutions. Typically, problems solved using this technique are solved the same way – each possible solution is tried and tried only once. A Naïve solution would try all solutions and output one that follows the specified constraints in the problem.

Backtracking is incremental in how it works and is a Naïve solution optimization. One of the most popular problems is the Knights Tour.

Problem

Let's say we have a chessboard, N*N. The Knight is on the first block of the empty board. Using the rules of chess, the Knight visits each square just once. You must print the order the squares are visited in.

Here's an example:

- **Input –** N = 8
- **Output**

  0 59 38 33 30 17 8 63

  37 34 31 60 9 62 29 16

  58 1 36 39 32 27 18 7

  35 48 41 26 61 10 15 28

  42 57 2 49 40 23 6 19

  47 50 45 54 25 20 11 14

  56 43 52 3 22 13 24 5

  51 46 55 44 53 4 21 12

Below you can see the chessboard with 8*8 squares, and the number in each square indicates the Knight's move number:

0	5 9	3 8	3 3	3 0	1 7	8	6 3
3 7	3 4	3 1	6 0	9	6 2	2 9	1 6
5 8	1	3 6	39	32	27	18	7
35	48	41	26	61	10	15	28
42	57	2	49	40	23	6	19
47	50	45	54	25	20	11	14
56	43	52	3	22	13	24	5
51	46	55	44	53	4	21	12

First, we'll look at the Naïve algorithm and then Backtracking.

Naïve Algorithm

The Naïve algorithm will generate each tour one at a time and see if it satisfies the specified constraints:

```
while there are untried tours
{
  generate the next tour
  if this tour covers all squares
  {
     print this path;
  }
}
```

Backtracking Algorithm

Backtracking works incrementally. Typically an empty solution vector is the start, and items are added one at a time. Items are different depending on the program, but, in this case, an item is a Knight's move. When an item is added, we check if the addition violates the constraints. If yes, the item is removed, and the next alternative is tried. If nothing works, we backtrack to the previous stage and remove the added item. If we get all the way back to the initial stage, we can say that there is no solution. If the constraints are not violated when an item is added, more items are added recursively, one at a time. If we get a complete solution vector, the solution can be printed.

Here is the algorithm for the Knights Tour problem:

```
If all the squares are visited
    print the solution
Else
  a) Add another to the solution vector and recursively
  check if it leads to a solution. (A Knight can make no more than
  eight moves. One of those eight moves is chosen in this step).
  b) If the move chosen doesn't lead to a solution
```

then remove it from the solution vector and try another move.

c) If none of the moves work, return false (Returning false

removes the previously added item in recursion, and if false is

returned by the initial call of recursion, “no solution exists”)

Below you can see the implementation for this problem. One of the solutions is printed in a 2D matrix. In short, the output is a 2D matrix, 8*8, showing numbers from 0 to 63, which indicate the Knight’s moves:

Following are implementations for Knight’s tour problem. It prints one of the possible solutions in 2D matrix form. Basically, the output is a 2D 8*8 matrix with numbers from 0 to 63, and these numbers show steps made by Knight.

```
// C++ program for Knight Tour problem
#include <bits/stdc++.h>
using namespace std;

#define N 8

int solveKTUtil(int x, int y, int movei, int sol[N][N],
                int xMove[], int yMove[]);

/* A utility function that checks if i,j are
valid indexes for N*N chessboard */
int isSafe(int x, int y, int sol[N][N])
{
    return (x >= 0 && x < N && y >= 0 && y < N
```

```
            && sol[x][y] == -1);
}

/* A utility function that prints the
solution matrix sol[N][N] */
void printSolution(int sol[N][N])
{
    for (int x = 0; x < N; x++) {
        for (int y = 0; y < N; y++)
            cout << " " << setw(2) << sol[x][y] << " ";
        cout << endl;
    }
}

/* This function will solve the Knight Tour problem using
Backtracking. It mainly uses solveKTUtil()
to solve the problem. It returns false if a complete
tour is not possible, otherwise it returns true and prints
the
tour.
Please note - there may be more than one solution,
this function prints only one of the possible solutions. */
int solveKT()
{
    int sol[N][N];

    /* Initialization of the solution matrix */
```

```
    for (int x = 0; x < N; x++)
        for (int y = 0; y < N; y++)
            sol[x][y] = -1;

    /* xMove[] and yMove[] define next move of Knight.
    xMove[] is for next value of x coordinate
    yMove[] is for next value of y coordinate */
    int xMove[8] = { 2, 1, -1, -2, -2, -1, 1, 2 };
    int yMove[8] = { 1, 2, 2, 1, -1, -2, -2, -1 };

    // Because the knight starts at the first block
    sol[0][0] = 0;

    /* Start from 0,0 and explore all tours using
    solveKTUtil() */
    if (solveKTUtil(0, 0, 1, sol, xMove, yMove) == 0) {
        cout << "Solution does not exist";
        return 0;
    }
    else
        printSolution(sol);

    return 1;
}

/* A recursive utility function that solves the Knight Tour
problem */
```

```
int solveKTUtil(int x, int y, int movei, int sol[N][N],
                int xMove[8], int yMove[8])
{
    int k, next_x, next_y;
    if (movei == N * N)
        return 1;

    /* Try all the next moves from
    the current coordinate x, y */
    for (k = 0; k < 8; k++) {
        next_x = x + xMove[k];
        next_y = y + yMove[k];
        if (isSafe(next_x, next_y, sol)) {
            sol[next_x][next_y] = movei;
            if (solveKTUtil(next_x, next_y, movei + 1, sol,
                            xMove, yMove)
                == 1)
                return 1;
            else

              // backtracking
                sol[next_x][next_y] = -1;
        }
    }
    return 0;
}
```

```
// Driver Code
int main()
{
    // Function Call
    solveKT();
    return 0;
}
```

Output:

```
 0  59  38  33  30  17   8  63
37  34  31  60   9  62  29  16
58   1  36  39  32  27  18   7
35  48  41  26  61  10  15  28
42  57   2  49  40  23   6  19
47  50  45  54  25  20  11  14
56 43  52   3  22  13  24   5
51  46  55  44  53   4  21  12
```

Time Complexity

We have N^2 squares, and there are eight possible moves for each one. That makes the worst-case running time $O(N^{\wedge 2})$. In terms of space complexity, we have an auxiliary space of $O(N^2)$

Auxiliary Space: $O(N^2)$

Subset Sum

The subset sum problem finds a subset of elements selected from a specified set where the sum adds up to a specified number, K. We'll assume the values in the set are non-negative and that we are using a unique input set with no duplicates.

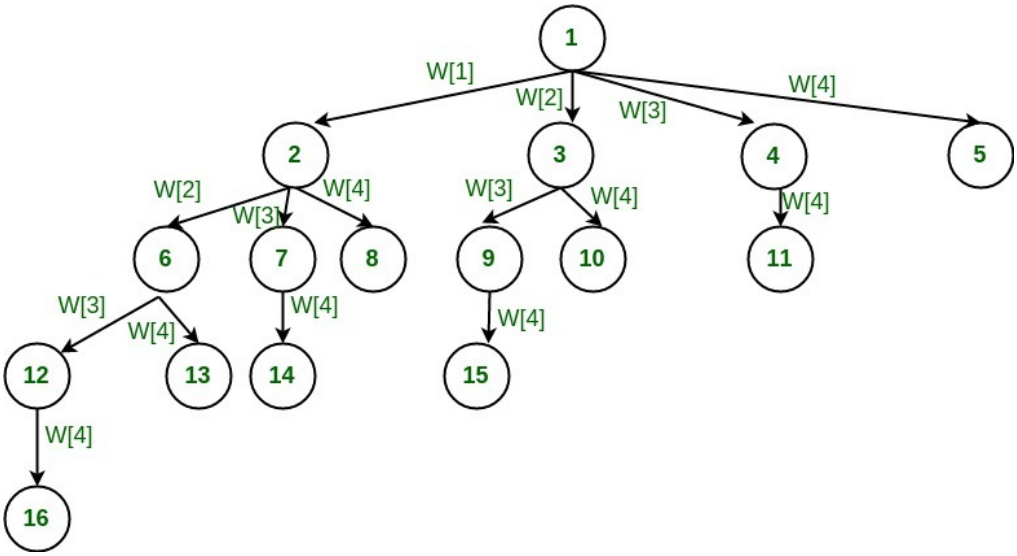
Exhaustive Algorithm

One way of finding the subsets that add up to K is to look at every possible subset. A powerset of size 2^N will contain all the generated subsets from a specified subset.

Backtracking Algorithm

When we use the exhaustive search, all subsets are considered, regardless of whether the constraints are satisfied or not. With backtracking, we can systematically consider all the elements to be selected.

Let's assume that our set has four elements, w[1] … w[4]. We can design a backtracking algorithm using a tree diagram, and the diagram below illustrates an approach where a variable-sized tuple is generated:



In this tree, a function call is represented by a node, and a candidate element is represented by a branch. There are four children in the root node. Simply put, each element in the set is considered as a separate branch by the root.

The subtrees on the next level correspond to the one with the parent node, and each level's branches represent a tuple element for consideration. For example, if we were at level 1, tuple_vector[1] would be able to take any of the values from the four generated branches. From level 2 of the left node, tuple_vector[2] would be able to take any of the values from the three generated branches, and so on,

The root's left-most child will generate every subset with w[1]. In the same way, the root's second child generates those subsets with 1[2] but without w[1].

Elements are added as we go down the tree's depth. If the sum we added satisfies the constraints, we can generate the child nodes. If the constraints were not met, we don't generate any more subtrees of the node and backtrack to the previous node, where unexplored nodes can be considered. Many times, this will save a serious amount of processing time.

We should get a clue from the tree to help us implement the algorithm. The subsets whose sum equals the specified number are printed. We need to look at the trees' depth and breadth nodes. A loop controls the nodes generated on the breadth, and recursion generates the nodes on the depth.

Here is the pseudocode:

```
if(subset is satisfying the constraint)
    print the subset
     exclude the current element and consider the next element
else
     generate the nodes of the present level along the breadth of the tree and
    recur for the next levels
```

Now we can look at a subset sum implementation that uses a variable size tuple vector. In this program, we look at every possibility in much the same way as the exhaustive search does. This demonstrates the use of backtracking, and the code will verify how the backtracking solution can be optimized.

The power in backtracking is obvious when implicit and explicit constraints are combined and, when the checks fail, we stop generating the nodes. The algorithm can be improved by strengthening the checks on the constraints and using presorted data. The rest of the array can be ignored once the sum is significantly bigger than the target number if the initial

array is sorted. Instead, we can backtrack and look at other possibilities.

So let's assume that we have a presorted array and we have found one subset. The next node can be generated by excluding the present one when the constraints are satisfied by including the next node.

Below, you can see an optimized implementation, where the subtree is pruned if the constraints are not satisfied.

```
#include <bits/stdc++.h>
using namespace std;

#define ARRAYSIZE(a) (sizeof(a))/(sizeof(a[0]))
static int total_nodes;

// prints subset found
void printSubset(int A[], int size)
{
    for(int i = 0; i < size; i++)
    {
       cout<<" "<< A[i];
    }
    cout<<"\n";
}

// qsort compare function
int comparator(const void *pLhs, const void *pRhs)
{
    int *lhs = (int *)pLhs;
    int *rhs = (int *)pRhs;
```

```
    return *lhs > *rhs;
}

// inputs
// s           - set vector
// t           - tuplet vector
// s_size      - set size
// t_size      - tuplet size so far
// sum         - sum so far
// ite         - nodes count
// target_sum  - sum to be found
void subset_sum(int s[], int t[],
                int s_size, int t_size,
                int sum, int ite,
                int const target_sum)
{
    total_nodes++;

    if( target_sum == sum )
    {
        // We found sum
        printSubset(t, t_size);

        // constraint check
         if( ite + 1 < s_size && sum - s[ite] + s[ite + 1] <=
target_sum )
        {
```

```
            // Exclude the previously added item and consider
the next candidate
                subset_sum(s, t, s_size, t_size - 1, sum - s[ite], ite
+ 1, target_sum);
        }
        return;
    }
    else
    {

        // constraint check
        if( ite < s_size && sum + s[ite] <= target_sum )
        {

            // generate the nodes along the breadth
            for( int i = ite; i < s_size; i++ )
            {
                t[t_size] = s[i];
                if( sum + s[i] <= target_sum )
                {

                    // consider the next level node (along depth)
                        subset_sum(s, t, s_size, t_size + 1, sum +
s[i], i + 1, target_sum);
                }
            }
        }
```

```
    }
}

// Wrapper that prints the subsets that sum to target_sum
void generateSubsets(int s[], int size, int target_sum)
{
    int *tuplet_vector = (int *)malloc(size * sizeof(int));
    int total = 0;

    // sort the set
    qsort(s, size, sizeof(int), &comparator);
    for( int i = 0; i < size; i++ )
    {
       total += s[i];
    }
    if( s[0] <= target_sum && total >= target_sum )
    {
                subset_sum(s, tuplet_vector, size, 0, 0, 0,
target_sum);
    }
    free(tuplet_vector);
}

// Driver code
int main()
{
    int weights[] = {15, 22, 14, 26, 32, 9, 16, 8};
```

```
    int target = 53;
    int size = ARRAYSIZE(weights);
    generateSubsets(weights, size, target);
    cout << “Nodes generated ” << total_nodes;
    return 0;
}
```

Output:

8 9 14 22n 8 14 15 16n 15 16 22nNodes generated 68

We could go down another route – the tree could be generated in fixed-size tuple analogs in a binary pattern and, when the constraints are not satisfied, the subtrees would be killed.

Conclusion

Thank you for taking the time to read my guide. The world revolves around algorithms now, and they are used to solve many common problems. Most of the time, you don't even realize that what you are doing requires an algorithm – it's all done neatly behind the scenes, giving you the results you want as if by magic.

We started with a quick look at how to design an algorithm, something you need to know and understand before you even attempt to get started on your own. We then moved on to some of the most popular design techniques, including backtracking, recursion, branch and bound, and the divide and conquer algorithm, providing plenty of code examples for you to follow and see how they work.

From here, you can enhance your knowledge further by digging even deeper into these algorithms and taking your learning another step closer to your goal. It could be that you want to understand what goes on behind the scenes better, or you are aiming for a career in computer science. Either way, this book has given you the foundation knowledge you need.

Thank you once again for reading. I hope you found the book useful.

References

“Basics of Greedy Algorithms Tutorials & Notes | Algorithms | HackerEarth.” *HackerEarth* , 2016, www.hackerearth.com/practice/algorithms/greedy/basics-of-greedy-algorithms/tutorial/.

Datta, Subham. “Branch and Bound Algorithm | Baeldung on Computer Science.” *Www.baeldung.com* , 23 Aug. 2020, www.baeldung.com/cs/branch-and-bound.

“Designing an Algorithm - Revision 3 - KS3 Computer Science - BBC Bitesize.” *BBC Bitesize* , 2019, www.bbc.co.uk/bitesize/guides/z3bq7ty/revision/3.

“Divide and Conquer.” *GeeksforGeeks* , www.geeksforgeeks.org/divide-and-conquer/.

February 12, and 2019. “Algorithm Design Techniques | How Is Algorithm Design Applied? | WLU.” *Online.wlu.ca* , online.wlu.ca/news/2019/02/12/how-algorithm-design-applied.

“Greedy Algorithms - GeeksforGeeks.” *GeeksforGeeks* , 2017, www.geeksforgeeks.org/greedy-algorithms/.

“Huffman Coding | Greedy Algo-3.” *GeeksforGeeks* , 3 Nov. 2012, www.geeksforgeeks.org/huffman-coding-greedy-algo-3/.

“ICS3U.” *Lah.elearningontario.ca* , lah.elearningontario.ca/CMS/public/exported_courses/ICS3U/exported/ICS3UU03/ICS3UU03/ICS3UU03A02/_content.html.

Miyaki, Keita. “Basic Algorithms — Finding the Closest Pair.” *Medium* , 10 Feb. 2020, towardsdatascience.com/basic-algorithms-finding-the-closest-pair-5fbef41e9d55.

Nehra, Pulkit. “Quicksort vs Merge Sort Which Is Better?” *Medium* , 14 Sept. 2021, medium.com/@pulkitnehra/quicksort-vs-merge-sort-which-is-better-6eb208ab27c5.

“Randomized Algorithms.” *OpenGenus IQ: Computing Expertise & Legacy* , 13 Nov. 2020, iq.opengenus.org/randomized-algorithms-introduction/.

“Randomized Algorithms | Brilliant Math & Science Wiki.” *Brilliant.org* , brilliant.org/wiki/randomized-algorithms-overview/.

“Recursion and Backtracking Tutorials & Notes | Basic Programming.” *HackerEarth* , www.hackerearth.com/practice/basic-programming/recursion/recursion-and-backtracking/tutorial/.

“Shortest Path Algorithms Tutorials & Notes | Algorithms | HackerEarth.” *HackerEarth* , 2016, www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/.

“Spanning Tree in Data Structure.” *TechVidvan* , 20 Aug. 2021, techvidvan.com/tutorials/spanning-tree/.

“Subset Sum | Backtracking-4.” *GeeksforGeeks* , 24 Sept. 2011, www.geeksforgeeks.org/subset-sum-backtracking-4/.

www.freecodecamp.org/news/demystifying-dynamic-programming-3efafb8d4296/.