CORE COURSE III

DESIGN AND ANALYSIS OF ALGORITHMS

Objectives :

To study the concepts of algorithms and analysis of algorithms using divide and conquer, greedy method, dynamic programming, backtracking, and branch and bound techniques

UNIT I

Introduction: Algorithm Definition – Algorithm Specification – Performance Analysis. Elementary Data Structures: Stacks and Queues – Trees – Dictionaries – Priority Queues – Sets and Disjoint Set Union – Graphs

UNIT II

Divide and Conquer: The General Method – Defective Chessboard – Binary Search – Finding The Maximum And Minimum – Merge Sort – Quick Sort – Selection - Strassen's Matrix Multiplication.

UNIT III

The Greedy Method: General Method - Container Loading - Knapsack Problem - Tree Vertex Splitting – Job Sequencing With Deadlines - Minimum Cost Spanning Trees - Optimal Storage On Tapes – Optimal Merge Patterns - Single Source Shortest Paths.

UNIT IV

Dynamic Programming: The General Method – Multistage Graphs – All-Pairs Shortest Paths – Single-Source Shortest Paths - Optimal Binary Search Trees - String Editing - 0/1 Knapsack - Reliability Design - The Traveling Salesperson Problem - Flow Shop Scheduling. Basic Traversal and Search Techniques: Techniques for Binary Trees – Techniques for Graphs – Connected Components and Spanning Trees – Biconnected Components and DFS.

UNIT V

Backtracking: The General Method – The 8-Queens Problem – Sum of Subsets – Graph Coloring – Hamiltonian Cycles – Knapsack Problem Branch and Bound: The Method - 0/1 Knapsack Problem.

UNIT 1

ALGORITHM:

An algorithm is a finite set of instructions that , if followed , accomplishes a particular task. In addition , all algorithms must satisfy the following criteria

Input

Output

Definiteness

Finiteness

Effectiveness

STACK AND QUEUE

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following diagram depicts a stack and its operations −

A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions −

peek()

Algorithm of peek() function −

```
begin procedure peek
   return stack[top]
end procedure
```

Implementation of peek() function in C programming language −

Example

```
int peek() {
   return stack[top];
}
```

isfull()

Algorithm of isfull() function −

```
begin procedure isfull

   if top equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

Implementation of isfull() function in C programming language −

Example

```
bool isfull() {
   if(top == MAXSIZE)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function −

```
begin procedure isempty

   if top less than 1
      return true
   else
      return false
   endif

end procedure
```

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code −

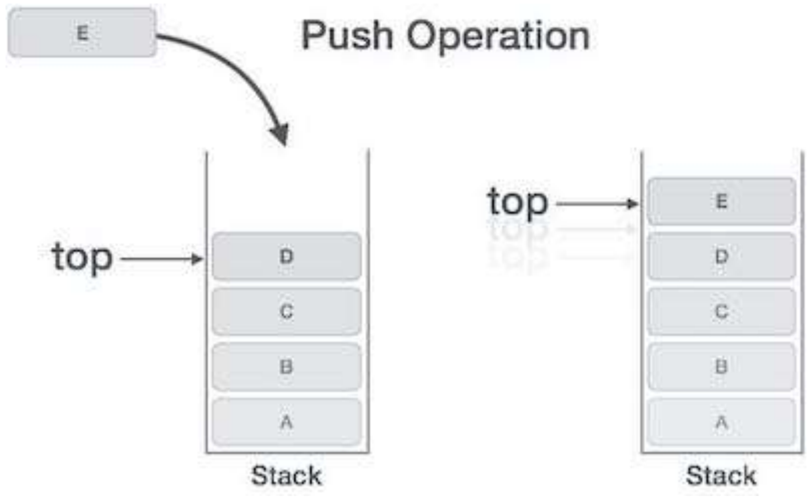
Example

```
bool isempty() {
   if(top == -1)
      return true;
   else
      return false;
}
```

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows −

```
begin procedure push: stack, data

   if stack is full
```

```
    return null
  endif

  top ← top + 1
  stack[top] ← data

end procedure
```

Implementation of this algorithm in C, is very easy. See the following code −

Example

```
void push(int data) {
  if(!isFull()) {
    top = top + 1;
    stack[top] = data;
  } else {
    printf("Could not insert data, Stack is full.\n");
  }
}
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

UNIT 2

Divide & Conquer

Many algorithms are recursive in nature to solve a given problem recursively dealing with sub-problems.

In **divide and conquer approach**, a problem is divided into smaller problems, then the smaller problems are solved independently, and finally the solutions of smaller problems are combined into a solution for the large problem.

Generally, divide-and-conquer algorithms have three parts −

- **Divide the problem** into a number of sub-problems that are smaller instances of the same problem.
- **Conquer the sub-problems** by solving them recursively. If they are small enough, solve the sub-problems as base cases.
- **Combine the solutions** to the sub-problems into the solution for the original problem.

Pros and cons of Divide and Conquer Approach

Divide and conquer approach supports parallelism as sub-problems are independent. Hence, an algorithm, which is designed using this technique, can run on the multiprocessor system or in different machines simultaneously.

In this approach, most of the algorithms are designed using recursion, hence memory management is very high. For recursive function stack is used, where function state needs to be stored.

Application of Divide and Conquer Approach

Following are some problems, which are solved using divide and conquer approach.

- Finding the maximum and minimum of a sequence of numbers
- Strassen's matrix multiplication
- Merge sort
- Binary search

Max-Min Problem

Let us consider a simple problem that can be solved by divide and conquer technique.

Problem Statement

The Max-Min Problem in algorithm analysis is finding the maximum and minimum value in an array.

Solution

To find the maximum and minimum numbers in a given array ***numbers[]*** of size **n**, the following algorithm can be used. First we are representing the **naive method** and then we will present **divide and conquer approach**.

Naïve Method

Naïve method is a basic method to solve any problem. In this method, the maximum and minimum number can be found separately. To find the maximum and minimum numbers, the following straightforward algorithm can be used.

Algorithm: Max-Min-Element (numbers[])

```
max := numbers[1]
min := numbers[1]

for i = 2 to n do
   if numbers[i] > max then
      max := numbers[i]
   if numbers[i] < min then
      min := numbers[i]
return (max, min)
```

Analysis

The number of comparison in Naive method is **2n - 2**.

The number of comparisons can be reduced using the divide and conquer approach. Following is the technique.

Divide and Conquer Approach

In this approach, the array is divided into two halves. Then using recursive approach maximum and minimum numbers in each halves are found. Later, return the maximum of two maxima of each half and the minimum of two minima of each half.

In this given problem, the number of elements in an array is $y-x+1$, where **y** is greater than or equal to **x**.
Max−Min(x,y) will return the maximum and minimum values of an array numbers[x...y].

Algorithm: Max - Min(x, y)

```
if y – x ≤ 1 then
   return (max(numbers[x], numbers[y]), min((numbers[x], numbers[y]))
else
   (max1, min1):= maxmin(x, ⌊((x + y)/2)⌋)
   (max2, min2):= maxmin(⌊((x + y)/2) + 1)⌋,y)
return (max(max1, max2), min(min1, min2))
```

Analysis

Let ***T(n)*** be the number of comparisons made by Max−Min(x,y), where the number of elements $n=y-x+1$.

If ***T(n)*** represents the numbers, then the recurrence relation can be represented as

$T(n)=\{T(\lfloor n2\rfloor)+T(\lceil n2\rceil)+2 for n>2 1 for n=2 0 for n=1$

Let us assume that ***n*** is in the form of power of **2**. Hence, $\mathbf{n = 2^k}$ where **k** is height of the recursion tree.

So,

$T(n)=2.T(n2)+2=2.(2.T(n4)+2)+2.....=3n2-2$

Compared to Naïve method, in divide and conquer approach, the number of comparisons is less. However, using the asymptotic notation both of the approaches are represented by **O(n)**.

Merge Sort

In this chapter, we will discuss merge sort and analyze its complexity.

Problem Statement

The problem of sorting a list of numbers lends itself immediately to a divide-and-conquer strategy: split the list into two halves, recursively sort each half, and then merge the two sorted sub-lists.

Solution

In this algorithm, the numbers are stored in an array ***numbers[]***. Here, ***p*** and ***q*** represents the start and end index of a sub-array.

```
Algorithm: Merge-Sort (numbers[], p, r)
if p < r then
q = ⌊(p + r) / 2⌋
Merge-Sort (numbers[], p, q)
    Merge-Sort (numbers[], q + 1, r)
    Merge (numbers[], p, q, r)
Function: Merge (numbers[], p, q, r)
$n_1$ = q – p + 1
```

```
$n_2$ = r – q
declare leftnums[1…$n_1$ + 1] and rightnums[1…$n_2$ + 1] temporary arrays
for i = 1 to $n_1$
   leftnums[i] = numbers[p + i - 1]
for j = 1 to $n_2$
   rightnums[j] = numbers[q+ j]
leftnums[$n_1$ + 1] = $\infty$
rightnums[$n_2$ + 1] = $\infty$
i = 1
j = 1
for k = p to r
   if leftnums[i] $\leq$ rightnums[j]
      numbers[k] = leftnums[i]
      i = i + 1
   else
      numbers[k] = rightnums[j]
      j = j + 1
```

Analysis

Let us consider, the running time of Merge-Sort as ***T(n)***. Hence,

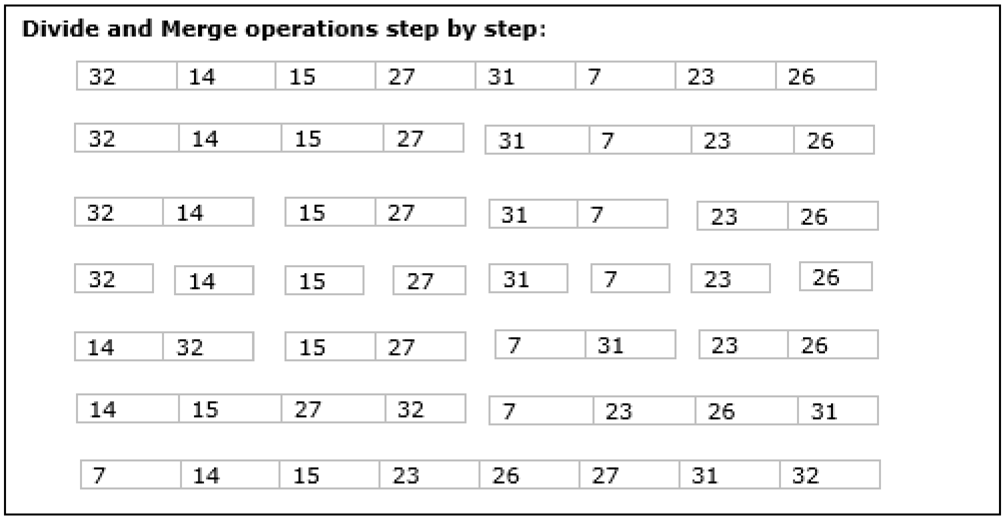
$T(n)=\{cifn\leqslant 12xT(n2)+dxnotherwise$ where c and d are constants

Therefore, using this recurrence relation,

$T(n)=2iT(n2i)+i.d.n$
As, $i=logn,T(n)=2lognT(n2logn)+logn.d.n$
$=c.n+d.n.logn$
Therefore, $T(n)=O(nlogn)$

Example

In the following example, we have shown Merge-Sort algorithm step by step. First, every iteration array is divided into two sub-arrays, until the sub-array contains only one element. When these sub-arrays cannot be divided further, then merge operations are performed.



Binary Search

In this chapter, we will discuss another algorithm based on divide and conquer method.

Problem Statement

Binary search can be performed on a sorted array. In this approach, the index of an element **x** is determined if the element belongs to the list of elements. If the array is unsorted, linear search is used to determine the position.

Solution

In this algorithm, we want to find whether element **x** belongs to a set of numbers stored in an array ***numbers[]***. Where ***l*** and ***r*** represent the left and right index of a sub-array in which searching operation should be performed.

Algorithm: Binary-Search(numbers[], x, l, r)

```
if l = r then
   return l
else
   m := $\lfloor (l + r) / 2 \rfloor$
   if $x \le$ numbers[m]  then
      return Binary-Search(numbers[], x, l, m)
   else
      return Binary-Search(numbers[], x, m+1, r)
```

Analysis

Linear search runs in ***O(n)*** time. Whereas binary search produces the result in ***O(log n)*** time

Let **T(n)** be the number of comparisons in worst-case in an array of **n** elements.

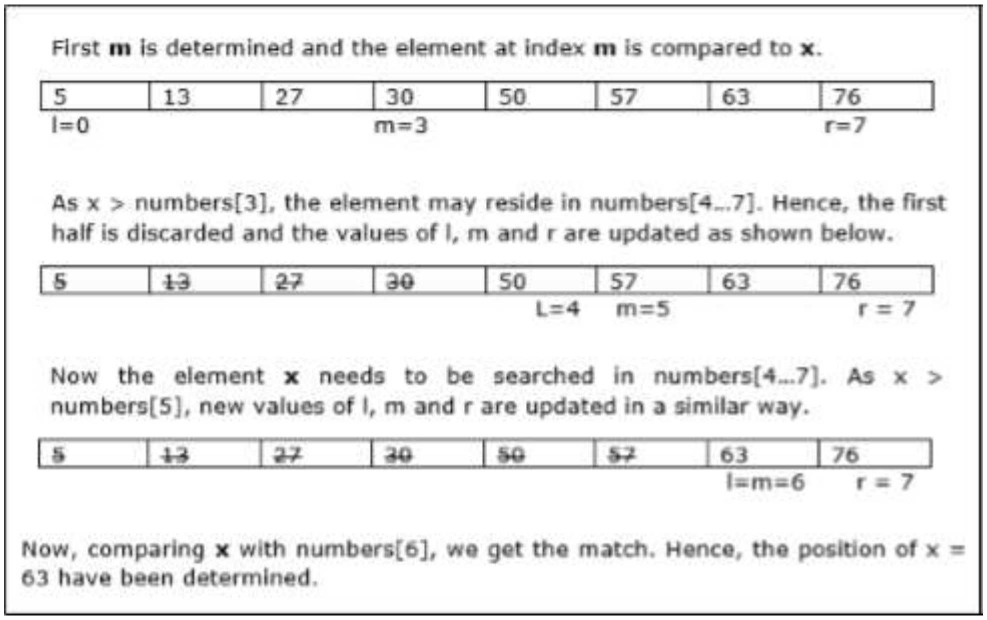
Hence,

T(n)={0T(n2)+1ifn=1otherwiseT(n)={0ifn=1T(n2)+1otherwise

Using this recurrence relation T(n)=lognT(n)=logn.
Therefore, binary search uses O(logn)O(logn) time.

Example

In this example, we are going to search element 63.


First **m** is determined and the element at index **m** is compared to **x**.

As x > numbers[3], the element may reside in numbers[4...7]. Hence, the first half is discarded and the values of l, m and r are updated as shown below.

Now the element **x** needs to be searched in numbers[4...7]. As x > numbers[5], new values of l, m and r are updated in a similar way.

Now, comparing **x** with numbers[6], we get the match. Hence, the position of x = 63 have been determined.

Strassen's Matrix Multiplication

In this chapter, first we will discuss the general method of matrix multiplication and later we will discuss Strassen's matrix multiplication algorithm.

Problem Statement

Let us consider two matrices ***X*** and ***Y***. We want to calculate the resultant matrix ***Z*** by multiplying ***X*** and ***Y***.

Naïve Method

First, we will discuss naïve method and its complexity. Here, we are calculating ***Z* = *X* × *Y***. Using Naïve method, two matrices (***X*** and ***Y***) can be multiplied if the order of these matrices are ***p*** × ***q*** and ***q*** × ***r***. Following is the algorithm.

Algorithm: Matrix-Multiplication (X, Y, Z)

```
for i = 1 to p do
  for j = 1 to r do
    Z[i,j] := 0
    for k = 1 to q do
      Z[i,j] := Z[i,j] + X[i,k] × Y[k,j]
```

Complexity

Here, we assume that integer operations take ***O(1)*** time. There are three **for** loops in this algorithm and one is nested in other. Hence, the algorithm takes ***O(n^3)*** time to execute.

Strassen's Matrix Multiplication Algorithm

In this context, using Strassen's Matrix multiplication algorithm, the time consumption can be improved a little bit.

Strassen's Matrix multiplication can be performed only on **square matrices** where **n** is a **power of 2**. Order of both of the matrices are **n × n**.

Divide **X**, **Y** and **Z** into four (n/2)×(n/2) matrices as represented below −

Z=[IKJL]Z=[IJKL] X=[ACBD]X=[ABCD] and Y=[EGFH]Y=[EFGH]

Using Strassen's Algorithm compute the following −

$$M1:=(A+C)\times(E+F)M1:=(A+C)\times(E+F)$$

$$M2:=(B+D)\times(G+H)M2:=(B+D)\times(G+H)$$

$$M3:=(A-D)\times(E+H)M3:=(A-D)\times(E+H)$$

$$M4:=A\times(F-H)M4:=A\times(F-H)$$

$$M5:=(C+D)\times(E)M5:=(C+D)\times(E)$$

$$M6:=(A+B)\times(H)M6:=(A+B)\times(H)$$

$$M7:=D\times(G-E)M7:=D\times(G-E)$$

Then,

$$I:=M2+M3-M6-M7I:=M2+M3-M6-M7$$

$$J:=M4+M6J:=M4+M6$$

$$K:=M5+M7K:=M5+M7$$

$$L:=M1-M3-M4-M5L:=M1-M3-M4-M5$$

Analysis

$T(n)=\{c7xT(n2)+dxn2ifn=1otherwiseT(n)=\{cifn=17xT(n2)+dxn2otherwise$ where *c* and *d* are constants

Using this recurrence relation, we get $T(n)=O(nlog7)T(n)=O(nlog7)$

Hence, the complexity of Strassen’s matrix multiplication algorithm is $O(nlog7)O(nlog7)$.

UNIT 3

Greedy Method

Among all the algorithmic approaches, the simplest and straightforward approach is the Greedy method. In this approach, the decision is taken on the basis of current available information without worrying about the effect of the current decision in future.

Greedy algorithms build a solution part by part, choosing the next part in such a way, that it gives an immediate benefit. This approach never reconsiders the choices taken previously. This approach is mainly used to solve optimization problems. Greedy method is easy to implement and quite efficient in most of the cases. Hence, we can say that Greedy algorithm is an algorithmic paradigm based on heuristic that follows local optimal choice at each step with the hope of finding global optimal solution.

In many problems, it does not produce an optimal solution though it gives an approximate (near optimal) solution in a reasonable time.

Components of Greedy Algorithm

Greedy algorithms have the following five components −

- **A candidate set** − A solution is created from this set.
- **A selection function** − Used to choose the best candidate to be added to the solution.
- **A feasibility function** − Used to determine whether a candidate can be used to contribute to the solution.
- **An objective function** − Used to assign a value to a solution or a partial solution.

- **A solution function** − Used to indicate whether a complete solution has been reached.

Areas of Application

Greedy approach is used to solve many problems, such as

- Finding the shortest path between two vertices using Dijkstra's algorithm.
- Finding the minimal spanning tree in a graph using Prim's /Kruskal's algorithm, etc.

Where Greedy Approach Fails

In many problems, Greedy algorithm fails to find an optimal solution, moreover it may produce a worst solution. Problems like Travelling Salesman and Knapsack cannot be solved using this approach.

Fractional Knapsack

The Greedy algorithm could be understood very well with a well-known problem referred to as Knapsack problem. Although the same problem could be solved by employing other algorithmic approaches, Greedy approach solves Fractional Knapsack problem reasonably in a good time. Let us discuss the Knapsack problem in detail.

Knapsack Problem

Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

The knapsack problem is in combinatorial optimization problem. It appears as a subproblem in many, more complex mathematical models of real-world problems. One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.

Applications

In many cases of resource allocation along with some constraint, the problem can be derived in a similar way of Knapsack problem. Following is a set of example.

- Finding the least wasteful way to cut raw materials

- portfolio optimization
- Cutting stock problems

Problem Scenario

A thief is robbing a store and can carry a maximal weight of ***W*** into his knapsack. There are n items available in the store and weight of ***i**^th^* item is $\boldsymbol{w_i}$ and its profit is $\boldsymbol{p_i}$. What items should the thief take?

In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.

Based on the nature of the items, Knapsack problems are categorized as

- Fractional Knapsack
- Knapsack

Fractional Knapsack

In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.

According to the problem statement,

- There are **n** items in the store
- Weight of **i**th item $wi>0wi>0$
- Profit for **i**th item $pi>0pi>0$ and
- Capacity of the Knapsack is **W**

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction $\boldsymbol{x_i}$ of **i**th item.

$$0\leqslant xi\leqslant 10\leqslant xi\leqslant 1$$

The **i**th item contributes the weight $xi.wixi.wi$ to the total weight in the knapsack and profit $xi.pixi.pi$ to the total profit.

Hence, the objective of this algorithm is to

$$\text{maximize}\sum n=1n(xi.pi)\text{maximize}\sum n=1n(xi.pi)$$

subject to constraint,

∑n=1n(xi.wi)⩽W∑n=1n(xi.wi)⩽W

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

∑n=1n(xi.wi)=W∑n=1n(xi.wi)=W

In this context, first we need to sort those items according to the value of piwipiwi, so that pi+1wi+1pi+1wi+1 ≤ piwipiwi . Here, ***x*** is an array to store the fraction of items.

Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

```
for i = 1 to n
  do x[i] = 0
weight = 0
for i = 1 to n
  if weight + w[i] ≤ W then
    x[i] = 1
    weight = weight + w[i]
  else
    x[i] = (W - weight) / w[i]
    weight = W
    break
return x
```

Analysis

If the provided items are already sorted into a decreasing order of piwipiwi, then the whileloop takes a time in ***O(n)***; Therefore, the total time including the sort is in ***O(n logn)***.

Example

Let us consider that the capacity of the knapsack ***W* = 60** and the list of provided items are shown in the following table −

Item	A	B	C	D
Profit	280	100	120	120

Weight	40	10	20	24
Ratio (piwi)(piwi)	7	10	6	5

As the provided items are not sorted based on piwipiwi. After sorting, the items are as shown in the following table.

Item	**B**	**A**	**C**	**D**
Profit	100	280	120	120
Weight	10	40	20	24
Ratio (piwi)(piwi)	10	7	6	5

Solution

After sorting all the items according to piwipiwi. First all of ***B*** is chosen as weight of ***B*** is less than the capacity of the knapsack. Next, item ***A*** is chosen, as the available capacity of the knapsack is greater than the weight of ***A***. Now, ***C*** is chosen as the next item. However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of ***C***.

Hence, fraction of ***C*** (i.e. (60 − 50)/20) is chosen.

Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected.

The total weight of the selected items is **10 + 40 + 20 * (10/20) = 60**

And the total profit is **100 + 280 + 120 * (10/20) = 380 + 60 = 440**

This is the optimal solution. We cannot gain more profit selecting any different combination of items.

Job Sequencing with Deadline

Problem Statement

In job sequencing problem, the objective is to find a sequence of jobs, which is completed within their deadlines and gives maximum profit.

Solution

Let us consider, a set of ***n*** given jobs which are associated with deadlines and profit is earned, if a job is completed by its deadline. These jobs need to be ordered in such a way that there is maximum profit.

It may happen that all of the given jobs may not be completed within their deadlines.

Assume, deadline of **i**th job J_i is d_i and the profit received from this job is p_i. Hence, the optimal solution of this algorithm is a feasible solution with maximum profit.

Thus, D(i)>0D(i)>0 for 1⩽i⩽n1⩽i⩽n.
Initially, these jobs are ordered according to profit, i.e. p1⩾p2⩾p3⩾...⩾pnp1⩾p2⩾p3⩾...⩾pn.

Algorithm: Job-Sequencing-With-Deadline (D, J, n, k)

```
D(0) := J(0) := 0
k := 1
J(1) := 1   // means first job is selected
for i = 2 … n do
  r := k
  while D(J(r)) > D(i) and D(J(r)) ≠ r do
    r := r – 1
  if D(J(r)) ≤ D(i) and D(i) > r then
    for l = k … r + 1 by -1 do
      J(l + 1) := J(l)
      J(r + 1) := i
      k := k + 1
```

Analysis

In this algorithm, we are using two loops, one is within another. Hence, the complexity of this algorithm is O(n2)O(n2).

Example

Let us consider a set of given jobs as shown in the following table. We have to find a sequence of jobs, which will be completed within their deadlines and will give maximum profit. Each job is associated with a deadline and profit.

Job	**J_1**	**J_2**	**J_3**	**J_4**	**J_5**
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

Solution

To solve this problem, the given jobs are sorted according to their profit in a descending order. Hence, after sorting, the jobs are ordered as shown in the following table.

Job	**J_2**	**J_1**	**J_4**	**J_3**	**J_5**
Deadline	1	2	2	3	1
Profit	100	60	40	20	20

From this set of jobs, first we select J_2, as it can be completed within its deadline and contributes maximum profit.

- Next, J_1 is selected as it gives more profit compared to J_4.
- In the next clock, J_4 cannot be selected as its deadline is over, hence J_3 is selected as it executes within its deadline.
- The job J_5 is discarded as it cannot be executed within its deadline.

Thus, the solution is the sequence of jobs (J_2, J_1, J_3), which are being executed within their deadline and gives maximum profit.

Total profit of this sequence is **100 + 60 + 20 = 180**.

Optimal Merge Pattern

Merge a set of sorted files of different length into a single sorted file. We need to find an optimal solution, where the resultant file will be generated in minimum time.

If the number of sorted files are given, there are many ways to merge them into a single sorted file. This merge can be performed pair wise. Hence, this type of merging is called as **2-way merge patterns**.

As, different pairings require different amounts of time, in this strategy we want to determine an optimal way of merging many files together. At each step, two shortest sequences are merged.

To merge a **p-record file** and a **q-record file** requires possibly **p + q** record moves, the obvious choice being, merge the two smallest files together at each step.

Two-way merge patterns can be represented by binary merge trees. Let us consider a set of **n** sorted files **{f_1, f_2, f_3, …, f_n}**. Initially, each element of this is considered as a single node binary tree. To find this optimal solution, the following algorithm is used.

Algorithm: TREE (n)

```
for i := 1 to n – 1 do
   declare new node
   node.leftchild := least (list)
   node.rightchild := least (list)
   node.weight) := ((node.leftchild).weight) + ((node.rightchild).weight)
   insert (list, node);
return least (list);
```

At the end of this algorithm, the weight of the root node represents the optimal cost.

Example

Let us consider the given files, f_1, f_2, f_3, f_4 and f_5 with 20, 30, 10, 5 and 30 number of elements respectively.

If merge operations are performed according to the provided sequence, then

M_1 = merge f_1 and f_2 => 20 + 30 = 50

M_2 = merge M_1 and f_3 => 50 + 10 = 60

M_3 = merge M_2 and f_4 => 60 + 5 = 65

M_4 = merge M_3 and f_5 => 65 + 30 = 95

Hence, the total number of operations is

50 + 60 + 65 + 95 = 270

Now, the question arises is there any better solution?

Sorting the numbers according to their size in an ascending order, we get the following sequence −

f_4, f_3, f_1, f_2, f_5

Hence, merge operations can be performed on this sequence

M_1 = merge f_4 and f_3 => 5 + 10 = 15

M_2 = merge M_1 and f_1 => 15 + 20 = 35

M_3 = merge M_2 and f_2 => 35 + 30 = 65

M_4 = merge M_3 and f_5 => 65 + 30 = 95

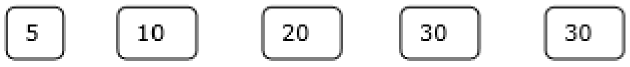
Therefore, the total number of operations is

15 + 35 + 65 + 95 = 210

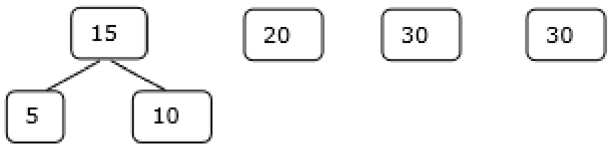
Obviously, this is better than the previous one.

In this context, we are now going to solve the problem using this algorithm.

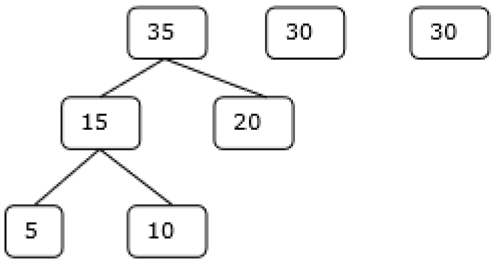
Initial Set



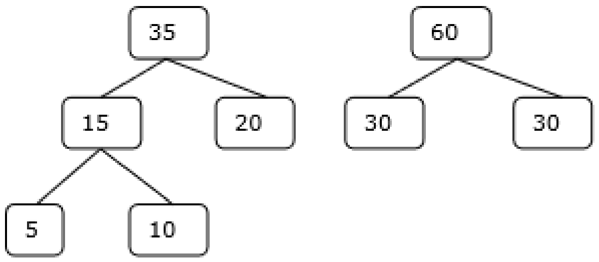
Step-1



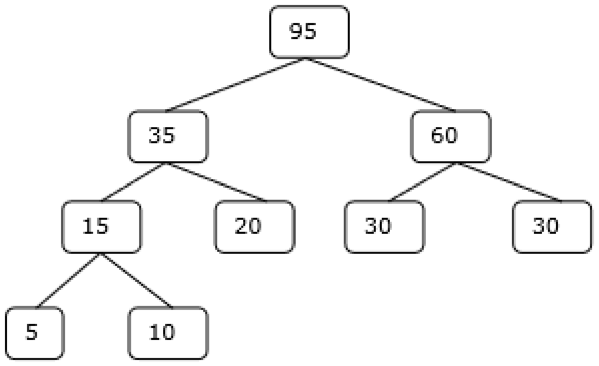
Step-2



Step-3



Step-4



Hence, the solution takes 15 + 35 + 60 + 95 = 205 number of comparisons.

Dynamic Programming

Dynamic Programming is also used in optimization problems. Like divide-and-conquer method, Dynamic Programming solves problems by combining the solutions of subproblems. Moreover, Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time.

Two main properties of a problem suggest that the given problem can be solved using Dynamic Programming. These properties are **overlapping sub-problems and optimal substructure**.

Overlapping Sub-Problems

Similar to Divide-and-Conquer approach, Dynamic Programming also combines solutions to sub-problems. It is mainly used where the solution of one sub-problem is needed repeatedly. The computed solutions are stored in a table, so that these don’t have to be re-computed. Hence, this technique is needed where overlapping sub-problem exists.

For example, Binary Search does not have overlapping sub-problem. Whereas recursive program of Fibonacci numbers have many overlapping sub-problems.

Optimal Sub-Structure

A given problem has Optimal Substructure Property, if the optimal solution of the given problem can be obtained using optimal solutions of its sub-problems.

For example, the Shortest Path problem has the following optimal substructure property −

If a node **x** lies in the shortest path from a source node **u** to destination node **v**, then the shortest path from **u** to **v** is the combination of the shortest path from **u** to **x**, and the shortest path from **x** to **v**.

The standard All Pair Shortest Path algorithms like Floyd-Warshall and Bellman-Ford are typical examples of Dynamic Programming.

Steps of Dynamic Programming Approach

Dynamic Programming algorithm is designed using the following four steps −

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from the computed information.

Applications of Dynamic Programming Approach

- Matrix Chain Multiplication
- Longest Common Subsequence
- Travelling Salesman Problem

Knapsack

In this tutorial, earlier we have discussed Fractional Knapsack problem using Greedy approach. We have shown that Greedy approach gives an optimal solution for Fractional Knapsack. However, this chapter will cover 0-1 Knapsack problem and its analysis.

In 0-1 Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack.

Hence, in case of 0-1 Knapsack, the value of $\boldsymbol{x_i}$ can be either ***0*** or ***1***, where other constraints remain the same.

0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution.

The following examples will establish our statement.

Example-1

Let us consider that the capacity of the knapsack is W = 25 and the items are as shown in the following table.

Item	**A**	**B**	**C**	**D**
Profit	24	18	18	10
Weight	24	10	10	7

Without considering the profit per unit weight ($\boldsymbol{p_i/w_i}$), if we apply Greedy approach to solve this problem, first item ***A*** will be selected as it will contribute max$_i$mum profit among all the elements.

After selecting item ***A***, no more item will be selected. Hence, for this given set of items total profit is ***24***. Whereas, the optimal solution can be achieved by selecting items, ***B*** and C, where the total profit is 18 + 18 = 36.

Example-2

Instead of selecting the items based on the overall benefit, in this example the items are selected based on ratio p_i/w_i. Let us consider that the capacity of the knapsack is W = 60 and the items are as shown in the following table.

Item	**A**	**B**	**C**
Price	100	280	120

Weight	10	40	20
Ratio	10	7	6

Using the Greedy approach, first item ***A*** is selected. Then, the next item ***B*** is chosen. Hence, the total profit is **100 + 280 = 380**. However, the optimal solution of this instance can be achieved by selecting items, ***B*** and ***C***, where the total profit is **280 + 120 = 400**.

Hence, it can be concluded that Greedy approach may not give an optimal solution.

To solve 0-1 Knapsack, Dynamic Programming approach is required.

Problem Statement

A thief is robbing a store and can carry a max$_i$mal weight of ***W*** into his knapsack. There are ***n*** items and weight of **i**th item is $\boldsymbol{w_i}$ and the profit of selecting this item is $\boldsymbol{p_i}$. What items should the thief take?

Dynamic-Programming Approach

Let ***i*** be the highest-numbered item in an optimal solution **S** for **W** dollars. Then $\boldsymbol{S' = S - \{i\}}$ is an optimal solution for $\boldsymbol{W - w_i}$ dollars and the value to the solution ***S*** is $\boldsymbol{V_i}$ plus the value of the sub-problem.

We can express this fact in the following formula: define **c[i, w]** to be the solution for items **1,2, … , i** and the maximum weight **w**.

The algorithm takes the following inputs

- The max$_i$mum weight **W**
- The number of items **n**
- The two sequences $\mathbf{v = <v_1, v_2, \ldots, v_n>}$ and $\mathbf{w = <w_1, w_2, \ldots, w_n>}$

```
Dynamic-0-1-knapsack (v, w, n, W)
for w = 0 to W do
  c[0, w] = 0
for i = 1 to n do
  c[i, 0] = 0
  for w = 1 to W do
    if $w_i \leq w$ then
```

```
        if $v_i + c[i-1, w-w_i]$ then
           $c[i, w] = v_i + c[i-1, w-w_i]$
        else $c[i, w] = c[i-1, w]$
    else
        $c[i, w] = c[i-1, w]$
```

The set of items to take can be deduced from the table, starting at **c[n, w]** and tracing backwards where the optimal values came from.

If *c[i, w] = c[i-1, w]*, then item ***i*** is not part of the solution, and we continue tracing with **c[i-1, w]**. Otherwise, item ***i*** is part of the solution, and we continue tracing with **c[i-1, w-W]**.

Analysis

This algorithm takes $\theta(n, w)$ times as table *c* has $(n + 1).(w + 1)$ entries, where each entry requires $\theta(1)$ time to compute.

Longest Common Subsequence

The longest common subsequence problem is finding the longest sequence which exists in both the given strings.

Subsequence

Let us consider a sequence $S = <s_1, s_2, s_3, s_4, \ldots, s_n>$.

A sequence $Z = <z_1, z_2, z_3, z_4, \ldots, z_m>$ over S is called a subsequence of S, if and only if it can be derived from S deletion of some elements.

Common Subsequence

Suppose, ***X*** and ***Y*** are two sequences over a finite set of elements. We can say that ***Z*** is a common subsequence of ***X*** and ***Y***, if ***Z*** is a subsequence of both ***X*** and ***Y***.

Longest Common Subsequence

If a set of sequences are given, the longest common subsequence problem is to find a common subsequence of all the sequences that is of maximal length.

The longest common subsequence problem is a classic computer science problem, the basis of data comparison programs such as the diff-utility, and has applications in bioinformatics. It is also widely used by revision control systems, such as SVN and Git, for reconciling multiple changes made to a revision-controlled collection of files.

Naïve Method

Let $\boldsymbol{X}$ be a sequence of length $\boldsymbol{m}$ and $\boldsymbol{Y}$ a sequence of length $\boldsymbol{n}$. Check for every subsequence of $\boldsymbol{X}$ whether it is a subsequence of $\boldsymbol{Y}$, and return the longest common subsequence found.

There are $\boldsymbol{2^m}$ subsequences of $\boldsymbol{X}$. Testing sequences whether or not it is a subsequence of $\boldsymbol{Y}$ takes $\boldsymbol{O(n)}$ time. Thus, the naïve algorithm would take $\boldsymbol{O(n2^m)}$ time.

UNIT 4

Dynamic Programming

Let $X = < x_1, x_2, x_3, \ldots, x_m >$ and $Y = < y_1, y_2, y_3, \ldots, y_n >$ be the sequences. To compute the length of an element the following algorithm is used.

In this procedure, table ***C[m, n]*** is computed in row major order and another table ***B[m,n]*** is computed to construct optimal solution.

Algorithm: LCS-Length-Table-Formulation (X, Y)

```
m := length(X)
n := length(Y)
for i = 1 to m do
  C[i, 0] := 0
for j = 1 to n do
  C[0, j] := 0
for i = 1 to m do
  for j = 1 to n do
    if $x_i = y_j$
      C[i, j] := C[i - 1, j - 1] + 1
      B[i, j] := 'D'
    else
      if C[i -1, j] ≥ C[i, j -1]
        C[i, j] := C[i - 1, j] + 1
        B[i, j] := 'U'
      else
      C[i, j] := C[i, j - 1]
      B[i, j] := 'L'
return C and B
```

Algorithm: Print-LCS (B, X, i, j)

```
if i = 0 and j = 0
  return
if B[i, j] = 'D'
  Print-LCS(B, X, i-1, j-1)
  Print($x_i$)
else if B[i, j] = 'U'
  Print-LCS(B, X, i-1, j)
else
  Print-LCS(B, X, i, j-1)
```

This algorithm will print the longest common subsequence of **X** and **Y**.

Analysis

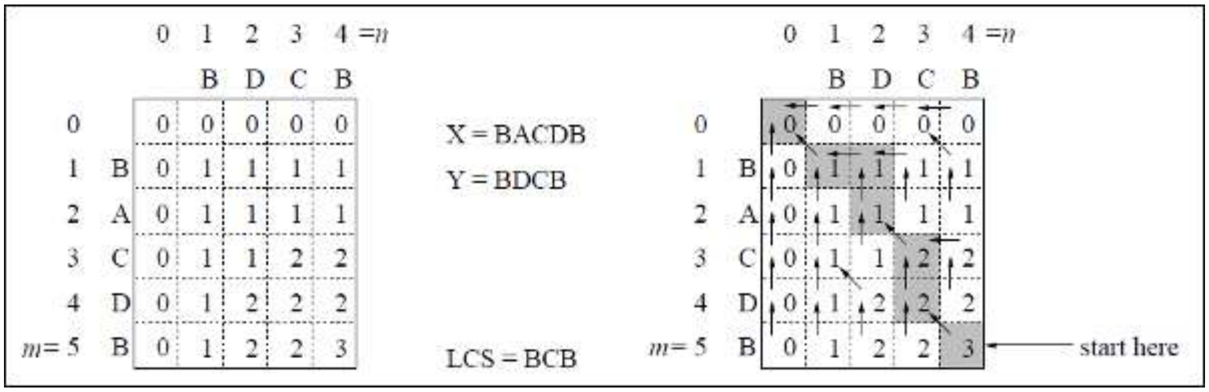
To populate the table, the outer **for** loop iterates ***m*** times and the inner **for** loop iterates ***n*** times. Hence, the complexity of the algorithm is *O(m, n)*, where ***m*** and ***n*** are the length of two strings.

Example

In this example, we have two strings ***X = BACDB*** and ***Y = BDCB*** to find the longest common subsequence.

Following the algorithm LCS-Length-Table-Formulation (as stated above), we have calculated table C (shown on the left hand side) and table B (shown on the right hand side).

In table B, instead of ‘D’, ‘L’ and ‘U’, we are using the diagonal arrow, left arrow and up arrow, respectively. After generating table B, the LCS is determined by function LCS-Print. The result is BCB.



		0	1	2	3	4 = n
			B	D	C	B
0		0	0	0	0	0
1	B	0	1	1	1	1
2	A	0	1	1	1	1
3	C	0	1	1	2	2
4	D	0	1	2	2	2
m = 5	B	0	1	2	2	3

X = BACDB

Y = BDCB

LCS = BCB

Spanning Tree

A **spanning tree** is a subset of an undirected Graph that has all the vertices connected by minimum number of edges.

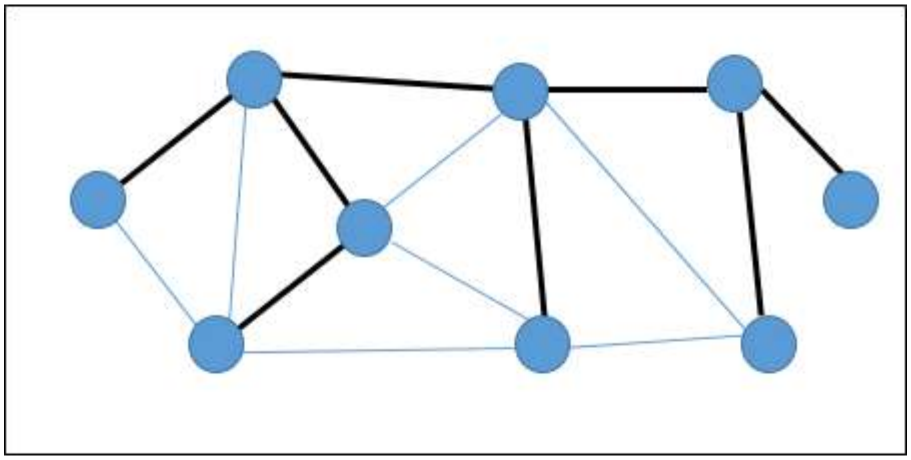
If all the vertices are connected in a graph, then there exists at least one spanning tree. In a graph, there may exist more than one spanning tree.

Properties

- A spanning tree does not have any cycle.
- Any vertex can be reached from any other vertex.

Example

In the following graph, the highlighted edges form a spanning tree.

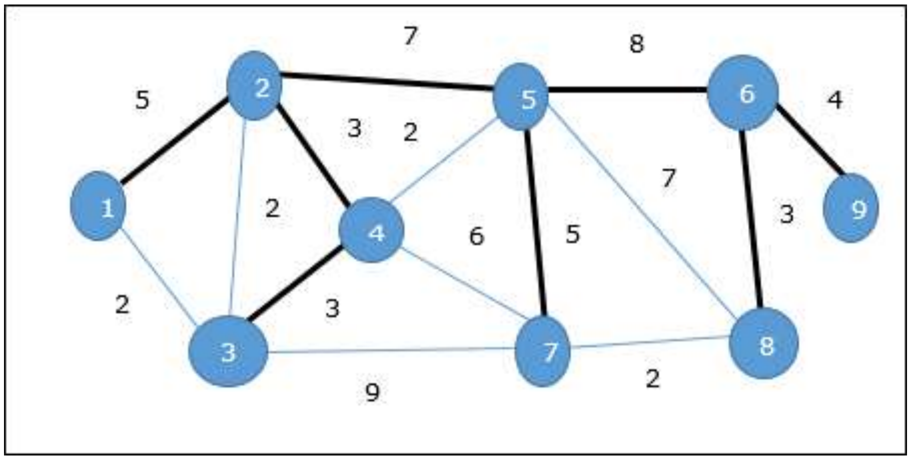


Minimum Spanning Tree

A **Minimum Spanning Tree (MST)** is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight. To derive an MST, Prim's algorithm or Kruskal's algorithm can be used. Hence, we will discuss Prim's algorithm in this chapter.

As we have discussed, one graph may have more than one spanning tree. If there are **n** number of vertices, the spanning tree should have ***n* - 1** number of edges. In this context, if each edge of the graph is associated with a weight and there exists more than one spanning tree, we need to find the minimum spanning tree of the graph.

Moreover, if there exist any duplicate weighted edges, the graph may have multiple minimum spanning tree.



In the above graph, we have shown a spanning tree though it's not the minimum spanning tree. The cost of this spanning tree is (5 + 7 + 3 + 3 + 5 + 8 + 3 + 4) = 38.

We will use Prim's algorithm to find the minimum spanning tree.

Prim's Algorithm

Prim's algorithm is a greedy approach to find the minimum spanning tree. In this algorithm, to form a MST we can start from an arbitrary vertex.

Algorithm: MST-Prim's (G, w, r)

```
for each u ϵ G.V
   u.key = ∞
   u.∏ = NIL
r.key = 0
Q = G.V
while Q ≠ Φ
   u = Extract-Min (Q)
   for each v ϵ G.adj[u]
      if each v ϵ Q and w(u, v) < v.key
         v.∏ = u
         v.key = w(u, v)
```

The function Extract-Min returns the vertex with minimum edge cost. This function works on min-heap.

Example

Using Prim's algorithm, we can start from any vertex, let us start from vertex **1**.

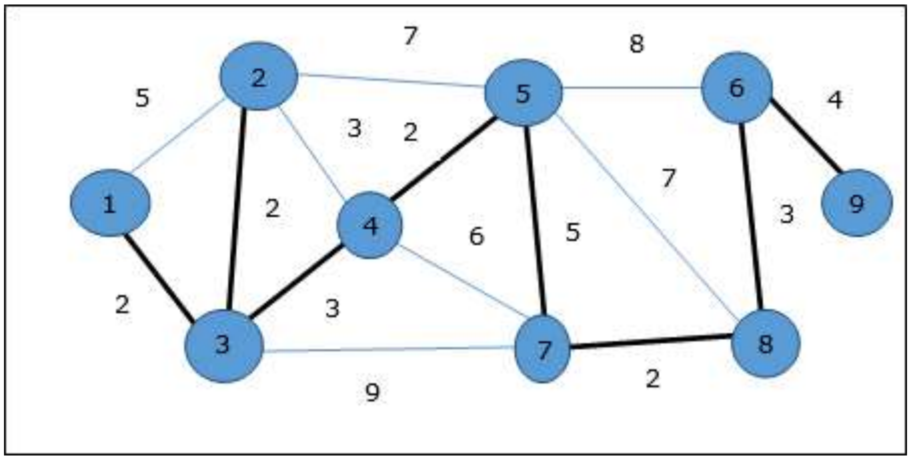
Vertex **3** is connected to vertex **1** with minimum edge cost, hence edge **(1, 2)** is added to the spanning tree.

Next, edge **(2, 3)** is considered as this is the minimum among edges **{(1, 2), (2, 3), (3, 4), (3, 7)}**.

In the next step, we get edge **(3, 4)** and **(2, 4)** with minimum cost. Edge **(3, 4)** is selected at random.

In a similar way, edges **(4, 5), (5, 7), (7, 8), (6, 8)** and **(6, 9)** are selected. As all the vertices are visited, now the algorithm stops.

The cost of the spanning tree is (2 + 2 + 3 + 2 + 5 + 2 + 3 + 4) = 23. There is no more spanning tree in this graph with cost less than **23**.



Shortest Paths

Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a directed weighted graph *G = (V, E)*, where all the edges are non-negative (i.e., *w(u, v)* ≥ 0 for each edge *(u, v) Є E*).

In the following algorithm, we will use one function ***Extract-Min()***, which extracts the node with the smallest key.

Algorithm: Dijkstra's-Algorithm (G, w, s)

```
for each vertex v Є G.V
   v.d := ∞
   v.∏ := NIL
s.d := 0
S := Φ
Q := G.V
while Q ≠ Φ
   u := Extract-Min (Q)
   S := S U {u}
   for each vertex v Є G.adj[u]
      if v.d > u.d + w(u, v)
         v.d := u.d + w(u, v)
         v.∏ := u
```

Analysis

The complexity of this algorithm is fully dependent on the implementation of Extract-Min function. If extract min function is implemented using linear search, the complexity of this algorithm is ***$O(V^2 + E)$***.

In this algorithm, if we use min-heap on which ***Extract-Min()*** function works to return the node from ***Q*** with the smallest key, the complexity of this algorithm can be reduced further.

Example

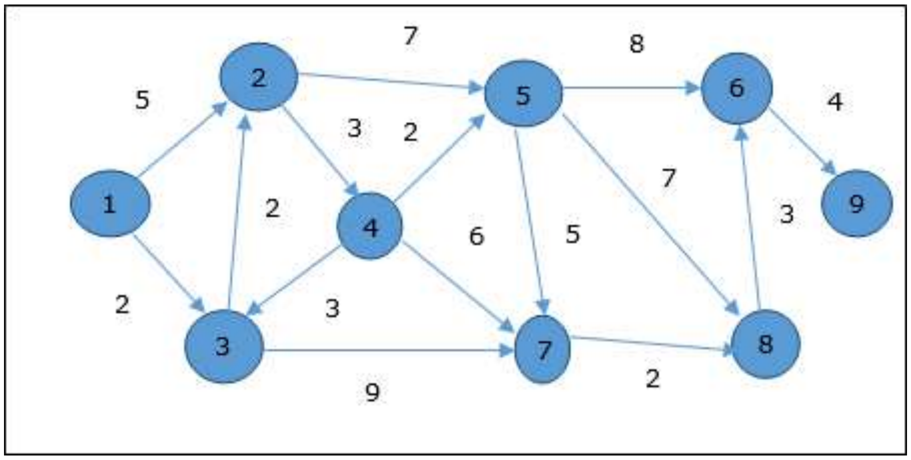
Let us consider vertex ***1*** and ***9*** as the start and destination vertex respectively. Initially, all the vertices except the start vertex are marked by ∞ and the start vertex is marked by ***0***.

Vertex	**Initial**	**Step1 V_1**	**Step2 V_3**	**Step3 V_2**	**Step4 V_4**	**Step5 V_5**	**Step6 V_7**	**Step7 V_8**	**Step8 V_6**
1	0	0	0	0	0	0	0	0	0
2	∞	5	4	4	4	4	4	4	4
3	∞	2	2	2	2	2	2	2	2
4	∞	∞	∞	7	7	7	7	7	7
5	∞	∞	∞	11	9	9	9	9	9
6	∞	∞	∞	∞	∞	17	17	16	16
7	∞	∞	11	11	11	11	11	11	11
8	∞	∞	∞	∞	∞	16	13	13	13
9	∞	∞	∞	∞	∞	∞	∞	∞	20

Hence, the minimum distance of vertex ***9*** from vertex ***1*** is ***20***. And the path is

1→ 3→ 7→ 8→ 6→ 9

This path is determined based on predecessor information.



Bellman Ford Algorithm

This algorithm solves the single source shortest path problem of a directed graph ***G = (V, E)*** in which the edge weights may be negative. Moreover, this algorithm can be applied to find the shortest path, if there does not exist any negative weighted cycle.

Algorithm: Bellman-Ford-Algorithm (G, w, s)

```
for each vertex v Є G.V
   v.d := ∞
   v.∏ := NIL
s.d := 0
for i = 1 to |G.V| - 1
   for each edge (u, v) Є G.E
      if v.d > u.d + w(u, v)
         v.d := u.d +w(u, v)
         v.∏ := u
for each edge (u, v) Є G.E
   if v.d > u.d + w(u, v)
      return FALSE
return TRUE
```

Analysis

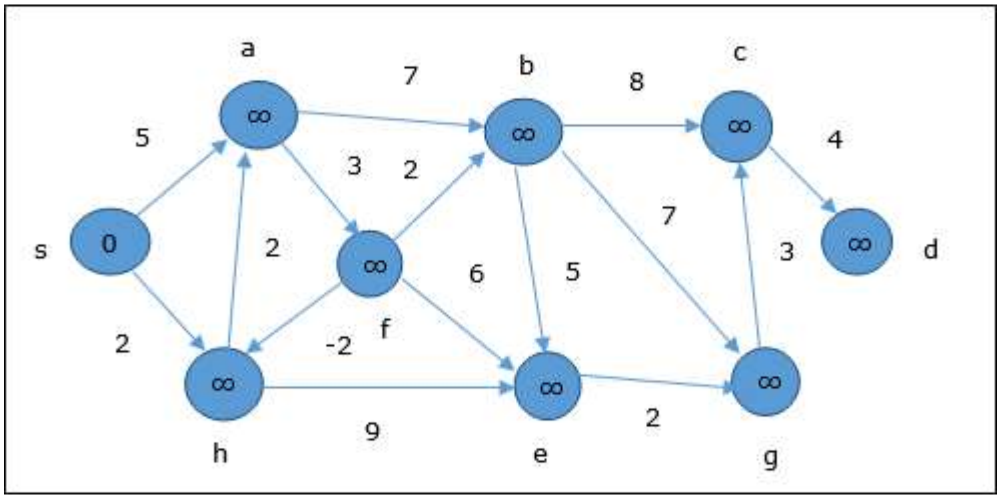
The first **for** loop is used for initialization, which runs in ***O(V)*** times. The next **for** loop runs |***V - 1***| passes over the edges, which takes ***O(E)*** times.

Hence, Bellman-Ford algorithm runs in ***O(V, E)*** time.

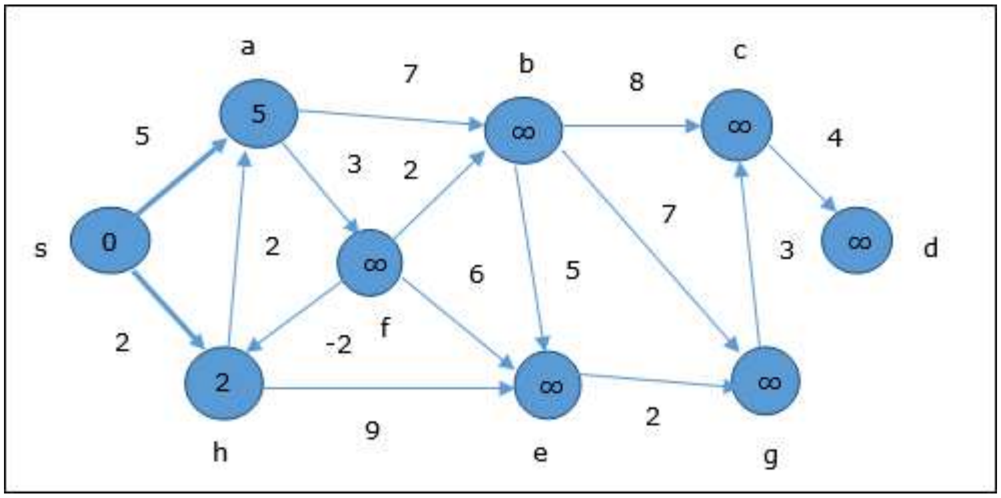
Example

The following example shows how Bellman-Ford algorithm works step by step. This graph has a negative edge but does not have any negative cycle, hence the problem can be solved using this technique.

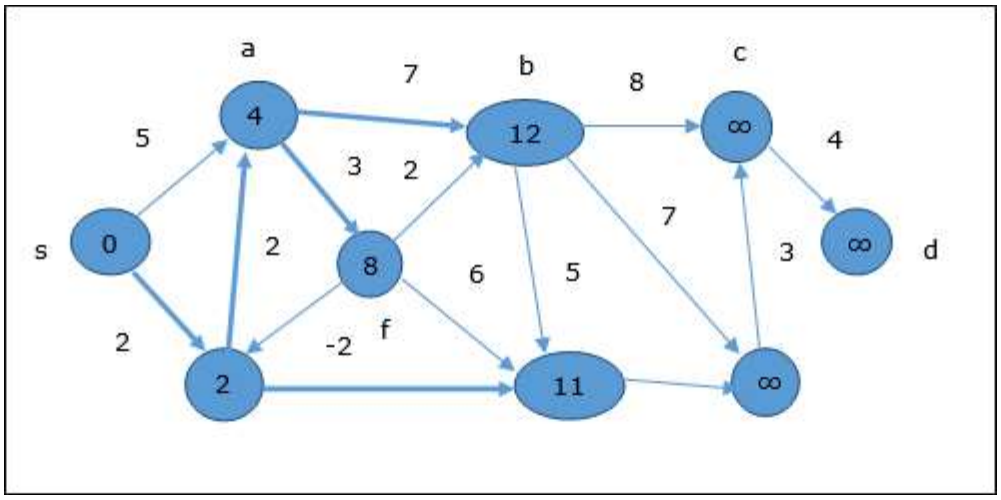
At the time of initialization, all the vertices except the source are marked by ∞ and the source is marked by **0**.



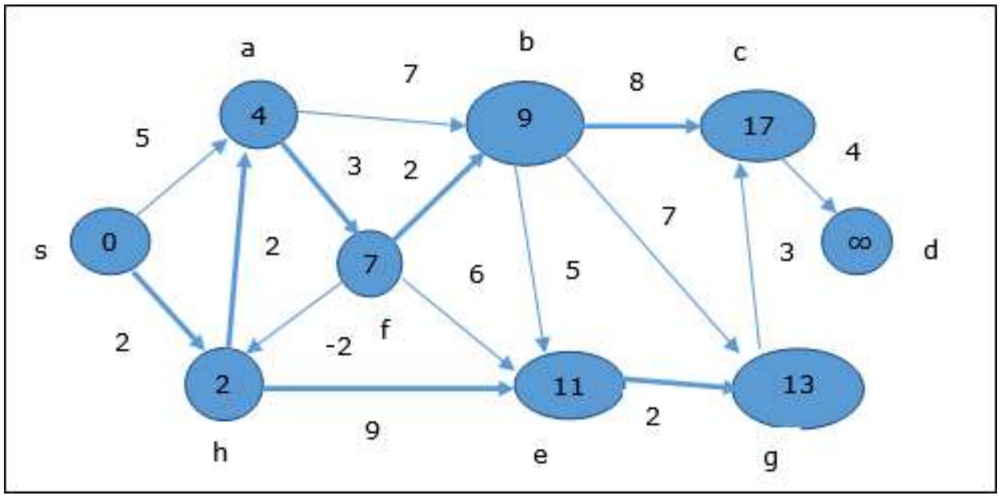
In the first step, all the vertices which are reachable from the source are updated by minimum cost. Hence, vertices ***a*** and ***h*** are updated.



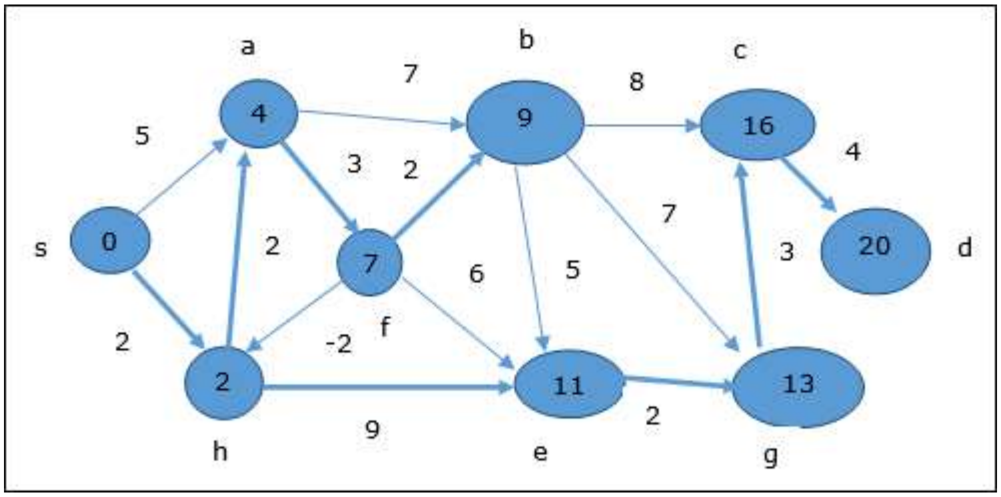
In the next step, vertices ***a, b, f*** and ***e*** are updated.



Following the same logic, in this step vertices ***b, f, c*** and ***g*** are updated.



Here, vertices ***c*** and ***d*** are updated.



Hence, the minimum distance between vertex **s** and vertex **d** is **20**.

Based on the predecessor information, the path is s→ h→ e→ g→ c→ d

Multistage Graph

A multistage graph **G = (V, E)** is a directed graph where vertices are partitioned into **k** (where ***k*** > **1**) number of disjoint subsets ***S* = {*s_1,s_2,…,s_k*}** such that edge *(u, v)* is in E, then *u* Є *s_i* and *v* Є *s_{1+1}* for some subsets in the partition and |***s_1***| = |***s_k***| = 1.

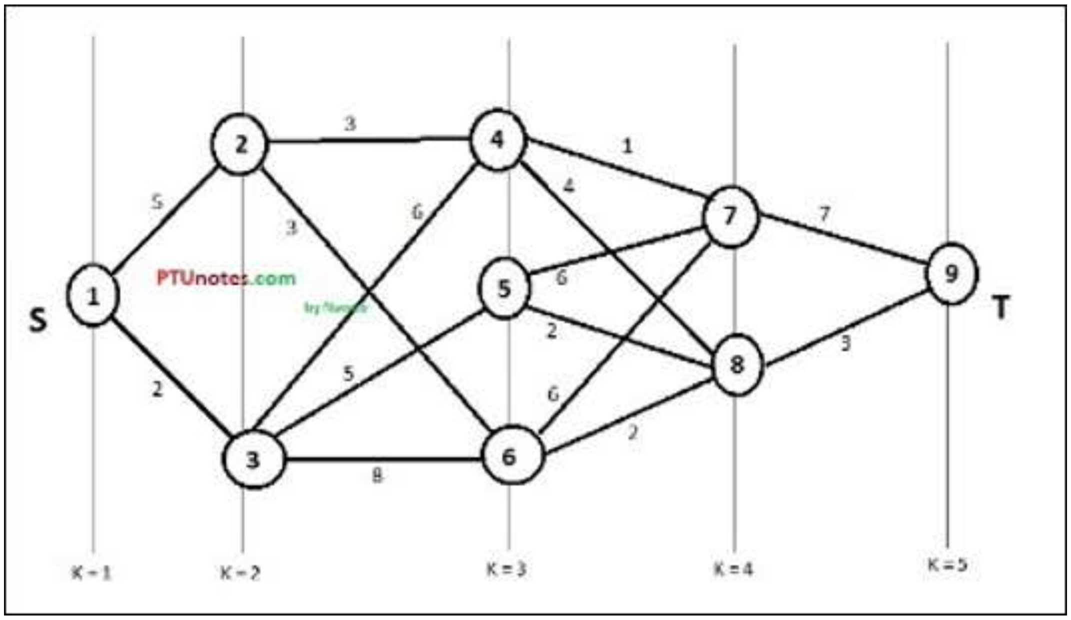
The vertex ***s* Є *s_1*** is called the **source** and the vertex ***t* Є *s_k*** is called **sink**.

G is usually assumed to be a weighted graph. In this graph, cost of an edge *(i, j)* is represented by *c(i, j)*. Hence, the cost of path from source ***s*** to sink ***t*** is the sum of costs of each edges in this path.

The multistage graph problem is finding the path with minimum cost from source ***s*** to sink ***t***.

Example

Consider the following example to understand the concept of multistage graph.



According to the formula, we have to calculate the cost **(i, j)** using the following steps

Step-1: Cost (K-2, j)

In this step, three nodes (node 4, 5. 6) are selected as **j**. Hence, we have three options to choose the minimum cost at this step.

Cost(3, 4) = min {c(4, 7) + Cost(7, 9),c(4, 8) + Cost(8, 9)} = 7

Cost(3, 5) = min {c(5, 7) + Cost(7, 9),c(5, 8) + Cost(8, 9)} = 5

Cost(3, 6) = min {c(6, 7) + Cost(7, 9),c(6, 8) + Cost(8, 9)} = 5

Step-2: Cost (K-3, j)

Two nodes are selected as j because at stage k - 3 = 2 there are two nodes, 2 and 3. So, the value i = 2 and j = 2 and 3.

Cost(2, 2) = min {c(2, 4) + Cost(4, 8) + Cost(8, 9),c(2, 6) +

Cost(6, 8) + Cost(8, 9)} = 8

Cost(2, 3) = {c(3, 4) + Cost(4, 8) + Cost(8, 9), c(3, 5) + Cost(5, 8)+ Cost(8, 9), c(3, 6) + Cost(6, 8) + Cost(8, 9)} = 10

Step-3: Cost (K-4, j)

Cost (1, 1) = {c(1, 2) + Cost(2, 6) + Cost(6, 8) + Cost(8, 9), c(1, 3) + Cost(3, 5) + Cost(5, 8) + Cost(8, 9))} = 12

c(1, 3) + Cost(3, 6) + Cost(6, 8 + Cost(8, 9))} = 13

Hence, the path having the minimum cost is **1→ 3→ 5→ 8→ 9**.

Travelling Salesman Problem

Problem Statement

A traveler needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once. What is the shortest possible route that he visits each city exactly once and returns to the origin city?

Solution

Travelling salesman problem is the most notorious computational problem. We can use brute-force approach to evaluate every possible tour and select the best one. For **n** number of vertices in a graph, there are **(*n* - 1)!** number of possibilities.

Instead of brute-force using dynamic programming approach, the solution can be obtained in lesser time, though there is no polynomial time algorithm.

Let us consider a graph ***G = (V, E)***, where ***V*** is a set of cities and ***E*** is a set of weighted edges. An edge ***e(u, v)*** represents that vertices ***u*** and ***v*** are connected. Distance between vertex ***u*** and ***v*** is ***d(u, v)***, which should be non-negative.

Suppose we have started at city ***1*** and after visiting some cities now we are in city ***j***. Hence, this is a partial tour. We certainly need to know ***j***, since this will determine which cities are most convenient to visit next. We also need to know all the cities visited so far, so that we don't repeat any of them. Hence, this is an appropriate sub-problem.

For a subset of cities ***S Є {1, 2, 3, ... , n}*** that includes ***1***, and ***j Є S***, let ***C(S, j)*** be the length of the shortest path visiting each node in **S** exactly once, starting at ***1*** and ending at ***j***.

When |***S***| > 1, we define ***C(S, 1)*** = ∝ since the path cannot start and end at **1**.

Now, let express **C(S, j)** in terms of smaller sub-problems. We need to start at ***1*** and end at **j**. We should select the next city in such a way that

C(S,j)=minC(S−{j},i)+d(i,j)wherei∈Sandi≠jc(S,j)=minC(s−{j},i)+d(i,j)wherei∈Sandi≠jC(S,j)=minC(S−{j},i)+d(i,j)wherei∈Sandi≠jc(S,j)=minC(s−{j},i)+d(i,j)wherei∈Sandi≠j

Algorithm: Traveling-Salesman-Problem

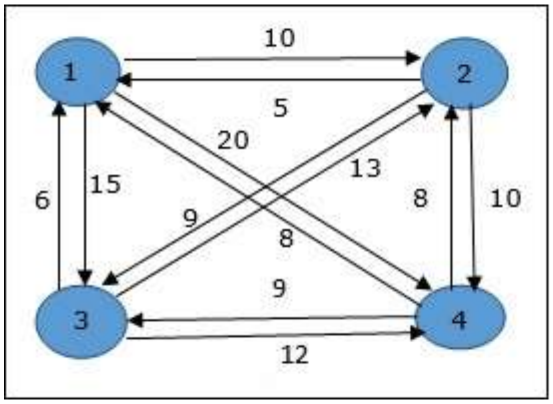
```
C ({1}, 1) = 0
for s = 2 to n do
   for all subsets S Є {1, 2, 3, … , n} of size s and containing 1
      C (S, 1) = ∞
   for all j Є S and j ≠ 1
      C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}
Return minj C ({1, 2, 3, …, n}, j) + d(j, i)
```

Analysis

There are at the most 2n.n2n.n sub-problems and each one takes linear time to solve. Therefore, the total running time is O(2n.n2)O(2n.n2).

Example

In the following example, we will illustrate the steps to solve the travelling salesman problem.



From the above graph, the following table is prepared.

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

S = Φ

Cost(2,Φ,1)=d(2,1)=5Cost(2,Φ,1)=d(2,1)=5Cost(2,Φ,1)=d(2,1)=5Cost(2,Φ,1)=d(2,1)=5

Cost(3,Φ,1)=d(3,1)=6Cost(3,Φ,1)=d(3,1)=6Cost(3,Φ,1)=d(3,1)=6Cost(3,Φ,1)=d(3,1)=6

Cost(4,Φ,1)=d(4,1)=8Cost(4,Φ,1)=d(4,1)=8Cost(4,Φ,1)=d(4,1)=8Cost(4,Φ,1)=d(4,1)=8

S = 1

Cost(i,s)=min{Cost(j,s–(j))+d[i,j]}Cost(i,s)=min{Cost(j,s)−(j))+d[i,j]}Cost(i,s)=min{Cost(j,s–(j))+d[i,j]}Cost(i,s)=min{Cost(j,s)−(j))+d[i,j]}

Cost(2,{3},1)=d[2,3]+Cost(3,Φ,1)=9+6=15cost(2,{3},1)=d[2,3]+cost(3,Φ,1)=9+6=15Cost(2,{3},1)=d[2,3]+Cost(3,Φ,1)=9+6=15cost(2,{3},1)=d[2,3]+cost(3,Φ,1)=9+6=15

Cost(2,{4},1)=d[2,4]+Cost(4,Φ,1)=10+8=18cost(2,{4},1)=d[2,4]+cost(4,Φ,1)=10+8=18Cost(2,{4},1)=d[2,4]+Cost(4,Φ,1)=10+8=18cost(2,{4},1)=d[2,4]+cost(4,Φ,1)=10+8=18

Cost(3,{2},1)=d[3,2]+Cost(2,Φ,1)=13+5=18cost(3,{2},1)=d[3,2]+cost(2,Φ,1)=13+5=18Cost(3,{2},1)=d[3,2]+Cost(2,Φ,1)=13+5=18cost(3,{2},1)=d[3,2]+cost(2,Φ,1)=13+5=18

Cost(3,{4},1)=d[3,4]+Cost(4,Φ,1)=12+8=20cost(3,{4},1)=d[3,4]+cost(4,Φ,1)=12+8=20Cost(3,{4},1)=d[3,4]+Cost(4,Φ,1)=12+8=20cost(3,{4},1)=d[3,4]+cost(4,Φ,1)=12+8=20

Cost(4,{3},1)=d[4,3]+Cost(3,Φ,1)=9+6=15cost(4,{3},1)=d[4,3]+cost(3,Φ,1)=9+6=15Cost(4,{3},1)=d[4,3]+Cost(3,Φ,1)=9+6=15cost(4,{3},1)=d[4,3]+cost(3,Φ,1)=9+6=15

Cost(4,{2},1)=d[4,2]+Cost(2,Φ,1)=8+5=13cost(4,{2},1)=d[4,2]+cost(2,Φ,1)=8+5=13Cost(4,{2},1)=d[4,2]+Cost(2,Φ,1)=8+5=13cost(4,{2},1)=d[4,2]+cost(2,Φ,1)=8+5=13

S = 2

Cost(2,{3,4},1)=⎧⎩⎭ d[2,3]+Cost(3,{4},1)=9+20=29d[2,4]+Cost(4,{3},1)=10+15=25=25Cost(2,{3,4},1){d[2,3]+cost(3,{4},1)=9+20=29d[2,4]+Cost(4,{3},1)=10+15=25=25Cost(2,{3,4},1)={d[2,3]+Cost(3,{4},1)=9+20=29d[2,4]+Cost(4,{3},1)=10+15=25=25Cost(2,{3,4},1){d[2,3]+cost(3,{4},1)=9+20=29d[2,4]+Cost(4,{3},1)=10+15=25=25

Cost(3,{2,4},1)=⎧⎩⎭ d[3,2]+Cost(2,{4},1)=13+18=31d[3,4]+Cost(4,{2},1)=12+13=25=25Cost(3,{2,4},1){d[3,2]+cost(2,{4},1)=13+18=31d[3,4]+Cost(4,{2},1)=12

+13=25=25Cost(3,{2,4},1)={d[3,2]+Cost(2,{4},1)=13+18=31d[3,4]+Cost(4,{2},1)=12+13=25=25Cost(3,{2,4},1){d[3,2]+cost(2,{4},1)=13+18=31d[3,4]+Cost(4,{2},1)=12+13=25=25

Cost(4,{2,3},1)=⎧⎩⎨ d[4,2]+Cost(2,{3},1)=8+15=23d[4,3]+Cost(3,{2},1)=9+18=27=23Cost(4,{2,3},1){d[4,2]+cost(2,{3},1)=8+15=23d[4,3]+Cost(3,{2},1)=9+18=27=23Cost(4,{2,3},1)={d[4,2]+Cost(2,{3},1)=8+15=23d[4,3]+Cost(3,{2},1)=9+18=27=23Cost(4,{2,3},1){d[4,2]+cost(2,{3},1)=8+15=23d[4,3]+Cost(3,{2},1)=9+18=27=23

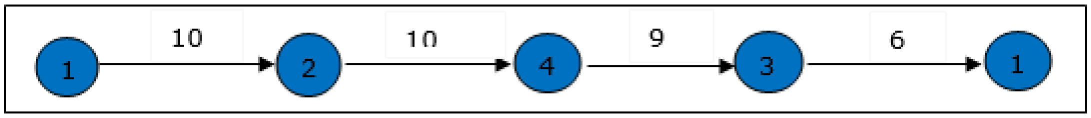
S = 3

Cost(1,{2,3,4},1)=⎧⎩⎨ | | | | | | | | | | | | | | | | d[1,2]+Cost(2,{3,4},1)=10+25=35d[1,3]+Cost(3,{2,4},1)=15+25=40d[1,4]+Cost(4,{2,3},1)=20+23=43=35cost(1,{2,3,4}),1)d[1,2]+cost(2,{3,4},1)=10+25=35d[1,3]+cost(3,{2,4},1)=15+25=40d[1,4]+cost(4,{2,3},1)=20+23=43=35Cost(1,{2,3,4},1)={d[1,2]+Cost(2,{3,4},1)=10+25=35d[1,3]+Cost(3,{2,4},1)=15+25=40d[1,4]+Cost(4,{2,3},1)=20+23=43=35cost(1,{2,3,4}),1)d[1,2]+cost(2,{3,4},1)=10+25=35d[1,3]+cost(3,{2,4},1)=15+25=40d[1,4]+cost(4,{2,3},1)=20+23=43=35

The minimum cost path is 35.

Start from cost **{1, {2, 3, 4}, 1}**, we get the minimum value for **d [1, 2]**. When **s = 3**, select the path from 1 to 2 (cost is 10) then go backwards. When **s = 2**, we get the minimum value for **d [4, 2]**. Select the path from 2 to 4 (cost is 10) then go backwards.

When **s = 1**, we get the minimum value for **d [4, 3]**. Selecting path 4 to 3 (cost is 9), then we shall go to then go to **s = Φ** step. We get the minimum value for **d [3, 1]** (cost is 6).



Optimal Cost Binary Search Trees

A Binary Search Tree (BST) is a tree where the key values are stored in the internal nodes. The external nodes are null nodes. The keys are ordered lexicographically, i.e. for each internal node all the keys in the left sub-tree are less than the keys in the node, and all the keys in the right sub-tree are greater.

When we know the frequency of searching each one of the keys, it is quite easy to compute the expected cost of accessing each node in the tree. An optimal binary search tree is a BST, which has minimal expected cost of locating each node

Search time of an element in a BST is ***O(n)***, whereas in a Balanced-BST search time is ***O(log n)***. Again the search time can be improved in Optimal Cost Binary Search Tree, placing the most frequently used data in the root and closer to the root element, while placing the least frequently used data near leaves and in leaves.

Here, the Optimal Binary Search Tree Algorithm is presented. First, we build a BST from a set of provided **n** number of distinct keys **<** $\boldsymbol{k_1, k_2, k_3, \ldots k_n}$ **>**. Here we assume, the probability of accessing a key $\boldsymbol{K_i}$ is $\boldsymbol{p_i}$. Some dummy keys ($\boldsymbol{d_0, d_1, d_2, \ldots d_n}$) are added as some searches may be performed for the values which are not present in the Key set ***K***. We assume, for each dummy key $\boldsymbol{d_i}$ probability of access is $\boldsymbol{q_i}$.

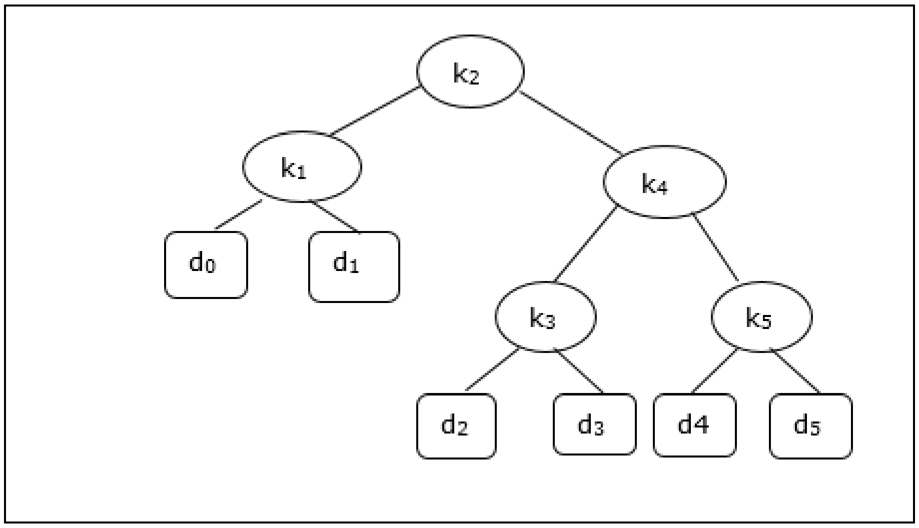
```
Optimal-Binary-Search-Tree(p, q, n)
e[1…n + 1, 0…n],
w[1…n + 1, 0…n],
root[1…n + 1, 0…n]
for i = 1 to n + 1 do
   e[i, i - 1] := $q_i$ - 1
   w[i, i - 1] := $q_i$ - 1
for l = 1 to n do
   for i = 1 to n – l + 1 do
      j = i + l – 1 e[i, j] := ∞
      w[i, i] := w[i, i -1] + $p_j$ + $q_j$
      for r = i to j do
         t := e[i, r - 1] + e[r + 1, j] + w[i, j]
         if t < e[i, j]
            e[i, j] := t
            root[i, j] := r
return e and root
```

Analysis

The algorithm requires **O ($\mathbf{n^3}$)** time, since three nested **for** loops are used. Each of these loops takes on at most **n** values.

Example

Considering the following tree, the cost is 2.80, though this is not an optimal result.



Node	Depth	Probability	Contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30

d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

To get an optimal solution, using the algorithm discussed in this chapter, the following tables are generated.

In the following tables, column index is **i** and row index is **j**.

e	**1**	**2**	**3**	**4**	**5**	**6**
5	2.75	2.00	1.30	0.90	0.50	0.10
4	1.75	1.20	0.60	0.30	0.05	
3	1.25	0.70	0.25	0.05		
2	0.90	0.40	0.05			
1	0.45	0.10				
0	0.05					

w	1	2	3	4	5	6
5	1.00	0.80	0.60	0.50	0.35	0.10
4	0.70	0.50	0.30	0.20	0.05	
3	0.55	0.35	0.15	0.05		
2	0.45	0.25	0.05			
1	0.30	0.10				
0	0.05					

root	1	2	3	4	5
5	2	4	5	5	5
4	2	2	4	4	
3	2	2	3		
2	1	2			
1	1				

From these tables, the optimal tree can be formed.

Binary Heap

There are several types of heaps, however in this chapter, we are going to discuss binary heap. A **binary heap** is a data structure, which looks similar to a complete binary tree. Heap data structure obeys ordering properties discussed below. Generally, a Heap is represented by an array. In this chapter, we are representing a heap by ***H***.

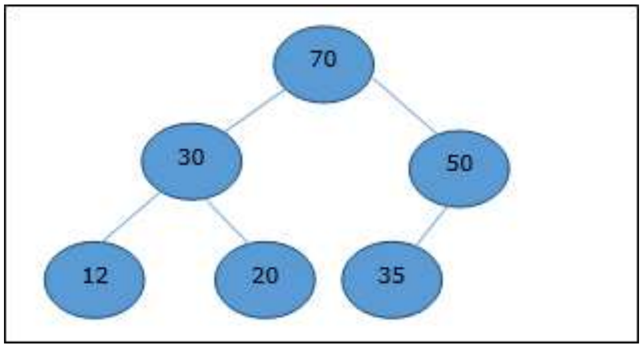
As the elements of a heap is stored in an array, considering the starting index as ***1***, the position of the parent node of **i**$^{\mathbf{th}}$ element can be found at $\lfloor i/2 \rfloor$. Left child and right child of **i**$^{\mathbf{th}}$ node is at position ***2i*** and ***2i* + *1***.

A binary heap can be classified further as either a ***max-heap*** or a ***min-heap*** based on the ordering property.

Max-Heap

In this heap, the key value of a node is greater than or equal to the key value of the highest child.

Hence, ***H[Parent(i)] ≥ H[i]***

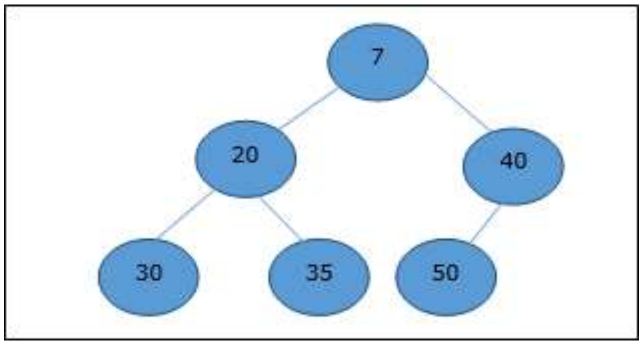


Min-Heap

In mean-heap, the key value of a node is lesser than or equal to the key value of the lowest child.

Hence, ***H[Parent(i)] ≤ H[i]***

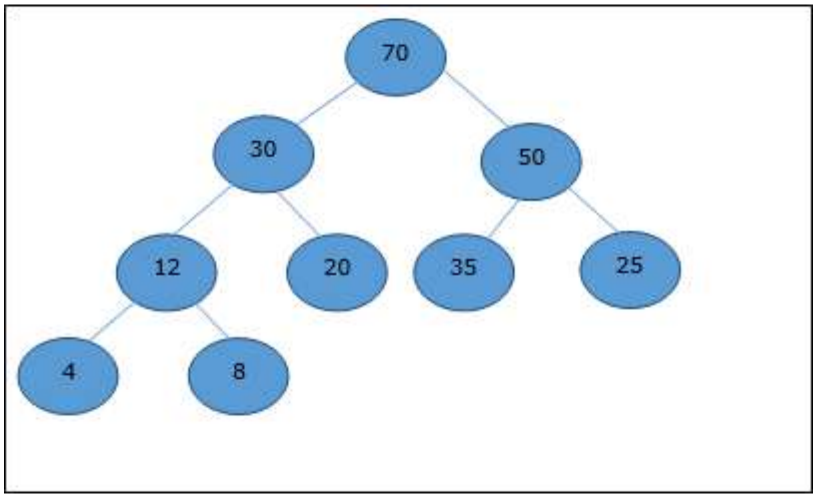
In this context, basic operations are shown below with respect to Max-Heap. Insertion and deletion of elements in and from heaps need rearrangement of elements. Hence, **Heapify** function needs to be called.



Array Representation

A complete binary tree can be represented by an array, storing its elements using level order traversal.

Let us consider a heap (as shown below) which will be represented by an array **H**.



Considering the starting index as **0**, using level order traversal, the elements are being kept in an array as follows.

Index	0	1	2	3	4	5	6	7	8	...
elements	70	30	50	12	20	35	25	4	8	...

In this context, operations on heap are being represented with respect to Max-Heap.

To find the index of the parent of an element at index **i**, the following algorithm ***Parent (numbers[], i)*** is used.

Algorithm: Parent (numbers[], i)

```
if i == 1
   return NULL
else
   [i / 2]
```

The index of the left child of an element at index **i** can be found using the following algorithm, ***Left-Child (numbers[], i)***.

Algorithm: Left-Child (numbers[], i)

```
If 2 * i ≤ heapsize
   return [2 * i]
else
   return NULL
```

The index of the right child of an element at index **i** can be found using the following algorithm, ***Right-Child(numbers[], i)***.

Algorithm: Right-Child (numbers[], i)

```
if 2 * i < heapsize
   return [2 * i + 1]
else
   return NULL
```

Insert Method

To insert an element in a heap, the new element is initially appended to the end of the heap as the last element of the array.

After inserting this element, heap property may be violated, hence the heap property is repaired by comparing the added element with its parent and moving the added element up a level, swapping positions with the parent. This process is called ***percolation up***.

The comparison is repeated until the parent is larger than or equal to the percolating element.

Algorithm: Max-Heap-Insert (numbers[], key)

```
heapsize = heapsize + 1
numbers[heapsize] = -∞
i = heapsize
numbers[i] = key
while i > 1 and numbers[Parent(numbers[], i)] < numbers[i]
   exchange(numbers[i], numbers[Parent(numbers[], i)])
```

```
i = Parent (numbers[], i)
```

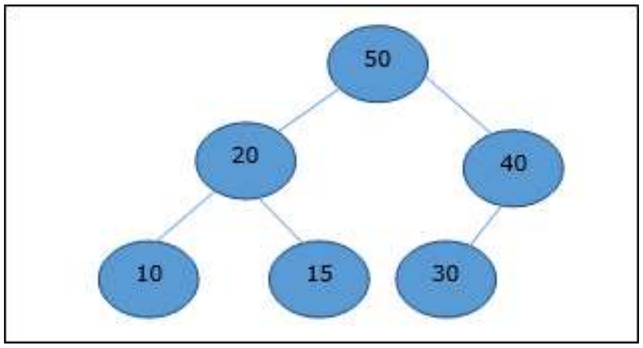
Analysis

Initially, an element is being added at the end of the array. If it violates the heap property, the element is exchanged with its parent. The height of the tree is ***log n***. Maximum ***log n*** number of operations needs to be performed.

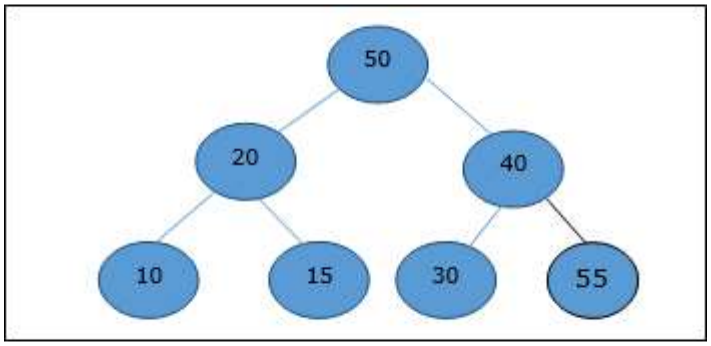
Hence, the complexity of this function is ***O(log n)***.

Example

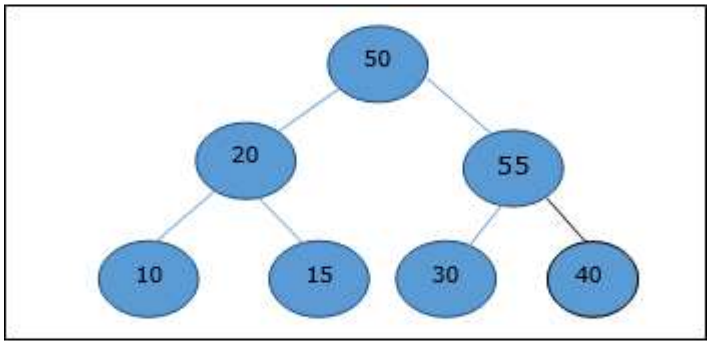
Let us consider a max-heap, as shown below, where a new element 5 needs to be added.



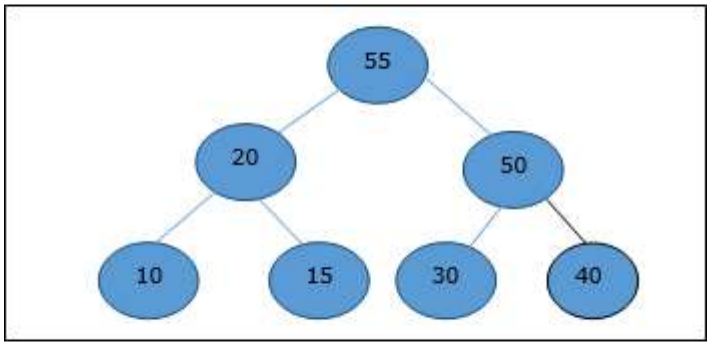
Initially, 55 will be added at the end of this array.



After insertion, it violates the heap property. Hence, the element needs to swap with its parent. After swap, the heap looks like the following.



Again, the element violates the property of heap. Hence, it is swapped with its parent.



Now, we have to stop.

Heapify Method

Heapify method rearranges the elements of an array where the left and right sub-tree of **i**th element obeys the heap property.

Algorithm: Max-Heapify(numbers[], i)

```
leftchild := numbers[2i]
rightchild := numbers [2i + 1]
if leftchild ≤ numbers[].size and numbers[leftchild] > numbers[i]
   largest := leftchild
else
   largest := i
if rightchild ≤ numbers[].size and numbers[rightchild] > numbers[largest]
   largest := rightchild
if largest ≠ i
   swap numbers[i] with numbers[largest]
   Max-Heapify(numbers, largest)
```

When the provided array does not obey the heap property, Heap is built based on the following algorithm ***Build-Max-Heap (numbers[])***.

Algorithm: Build-Max-Heap(numbers[])

```
numbers[].size := numbers[].length
fori = ⌊  numbers[].length/2 ⌋  to 1 by -1
  Max-Heapify (numbers[], i)
```

Extract Method

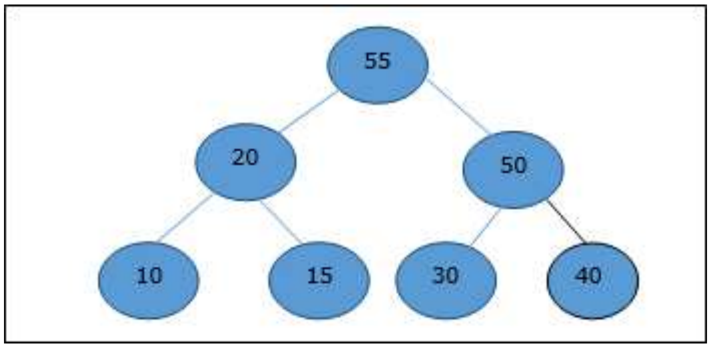
Extract method is used to extract the root element of a Heap. Following is the algorithm.

Algorithm: Heap-Extract-Max (numbers[])

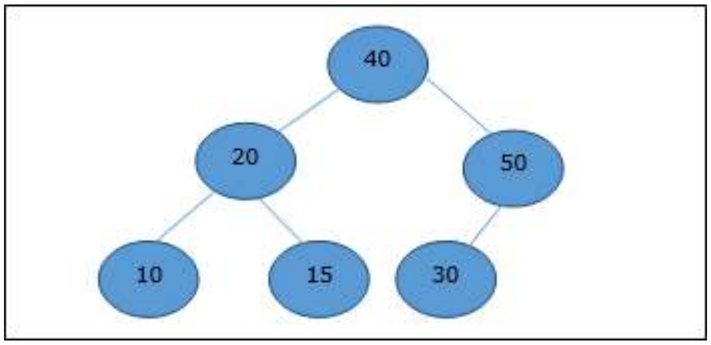
```
max = numbers[1]
numbers[1] = numbers[heapsize]
heapsize = heapsize – 1
Max-Heapify (numbers[], 1)
return max
```

Example

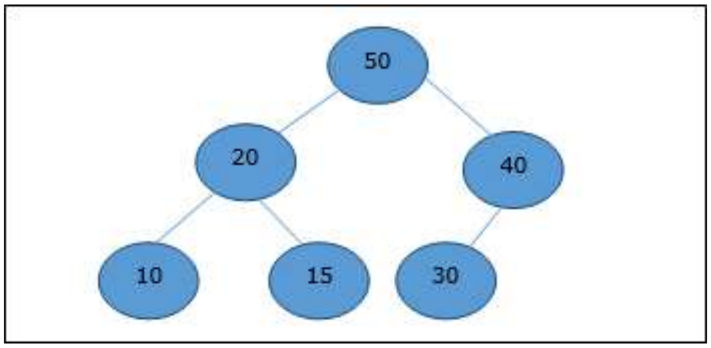
Let us consider the same example discussed previously. Now we want to extract an element. This method will return the root element of the heap.



After deletion of the root element, the last element will be moved to the root position.



Now, Heapify function will be called. After Heapify, the following heap is generated.



SOTRING METHODS

Bubble Sort

Bubble Sort is an elementary sorting algorithm, which works by repeatedly exchanging adjacent elements, if necessary. When no exchanges are required, the file is sorted.

This is the simplest technique among all sorting algorithms.

Algorithm: Sequential-Bubble-Sort (A)

```
fori← 1 to length [A] do
for j ← length [A] down-to i +1 do
  if A[A] < A[j - 1] then
    Exchange A[j] ↔ A[j-1]
```

Implementation

```
voidbubbleSort(int numbers[], intarray_size) {
  inti, j, temp;
  for (i = (array_size - 1); i >= 0; i--)
  for (j = 1; j <= i; j++)
    if (numbers[j - 1] > numbers[j]) {
      temp = numbers[j-1];
      numbers[j - 1] = numbers[j];
      numbers[j] = temp;
    }
}
```

Analysis

Here, the number of comparisons are

$1 + 2 + 3 + ... + (n - 1) = n(n - 1)/2 = O(n^2)$

Clearly, the graph shows the ***n^2*** nature of the bubble sort.

In this algorithm, the number of comparison is irrespective of the data set, i.e. whether the provided input elements are in sorted order or in reverse order or at random.

Memory Requirement

From the algorithm stated above, it is clear that bubble sort does not require extra memory.

Example

Unsorted list:

5	2	1	4	3	7	6

,

Insertion Sort

Insertion sort is a very simple method to sort numbers in an ascending or descending order. This method follows the incremental method. It can be compared with the technique how cards are sorted at the time of playing a game.

The numbers, which are needed to be sorted, are known as **keys**. Here is the algorithm of the insertion sort method.

Algorithm: Insertion-Sort(A)

```
for j = 2 to A.length
```

```
key = A[j]
i = j – 1
while i > 0 and A[i] > key
   A[i + 1] = A[i]
   i = i -1
A[i + 1] = key
```

Analysis

Run time of this algorithm is very much dependent on the given input.

If the given numbers are sorted, this algorithm runs in ***O(n)*** time. If the given numbers are in reverse order, the algorithm runs in ***$O(n^2)$*** time.

Example

Unsorted list:

2	13	5	18	14

1st iteration:

Key = a[2] = 13

a[1] = 2 < 13

Swap, no swap

2	13	5	18	14

2nd iteration:

Key = a[3] = 5

a[2] = 13 > 5

Swap 5 and 13

2	5	13	18	14

Next, a[1] = 2 < 13

2	5	13	18	14

3rd iteration:

Key = a[4] = 18

a[3] = 13 < 18,

a[2] = 5 < 18,

a[1] = 2 < 18

Swap, no swap	2	5	13	18	14

4th iteration:

Key = a[5] = 14

a[4] = 18 > 14

Swap 18 and 14	2	5	13	14	18

Next, a[3] = 13 < 14,

a[2] = 5 < 14,

a[1] = 2 < 14

So, no swap	2	5	13	14	18

Finally,

the sorted list is	2	5	13	14	18

Selection Sort

This type of sorting is called **Selection Sort** as it works by repeatedly sorting elements. It works as follows: first find the smallest in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continue in this way until the entire array is sorted.

Algorithm: Selection-Sort (A)

```
fori ← 1 to n-1 do
  min j ← i;
```

```
 min x ← A[i]
 for j ←i + 1 to n do
   if A[j] < min x then
     min j ← j
     min x ← A[j]
 A[min j] ← A [i]
 A[i] ← min x
```

Selection sort is among the simplest of sorting techniques and it works very well for small files. It has a quite important application as each item is actually moved at the most once.

Section sort is a method of choice for sorting files with very large objects (records) and small keys. The worst case occurs if the array is already sorted in a descending order and we want to sort them in an ascending order.

Nonetheless, the time required by selection sort algorithm is not very sensitive to the original order of the array to be sorted: the test if ***A[j]*** < ***min x*** is executed exactly the same number of times in every case.

Selection sort spends most of its time trying to find the minimum element in the unsorted part of the array. It clearly shows the similarity between Selection sort and Bubble sort.

- Bubble sort selects the maximum remaining elements at each stage, but wastes some effort imparting some order to an unsorted part of the array.
- Selection sort is quadratic in both the worst and the average case, and requires no extra memory.

For each ***i*** from ***1*** to ***n - 1***, there is one exchange and ***n - i*** comparisons, so there is a total of ***n - 1*** exchanges and

$(n - 1) + (n - 2) + ... + 2 + 1 = n(n - 1)/2$ comparisons.

These observations hold, no matter what the input data is.

In the worst case, this could be quadratic, but in the average case, this quantity is ***O(n log n)***. It implies that the **running time of Selection sort is quite insensitive to the input**.

Implementation

```
Void Selection-Sort(int numbers[], int array_size) {
 int i, j;
 int min, temp;
 for (i = 0; I < array_size-1; i++) {
        min = i;
        for (j = i+1; j < array_size; j++)
          if (numbers[j] < numbers[min])
            min = j;
        temp = numbers[i];
        numbers[i] = numbers[min];
        numbers[min] = temp;
    }
}
```

Example

Unsorted list:

5	2	1	4	3

1st iteration:

Smallest = 5

2 < 5, smallest = 2

1 < 2, smallest = 1

4 > 1, smallest = 1

3 > 1, smallest = 1

Swap 5 and 1

1	2	5	4	3

2nd iteration:

Smallest = 2

2 < 5, smallest = 2

2 < 4, smallest = 2

2 < 3, smallest = 2

No Swap

1	2	5	4	3

3rd iteration:

Smallest = 5

4 < 5, smallest = 4

3 < 4, smallest = 3

Swap 5 and 3

1	2	3	4	5

4th iteration:

Smallest = 4

4 < 5, smallest = 4

No Swap

1	2	3	4	5

Finally,

the sorted list is

1	2	3	4	5

Quick Sort

It is used on the principle of divide-and-conquer. Quick sort is an algorithm of choice in many situations as it is not difficult to implement. It is a good general purpose sort and it consumes relatively fewer resources during execution.

Advantages

- It is in-place since it uses only a small auxiliary stack.
- It requires only ***n (log n)*** time to sort **n** items.
- It has an extremely short inner loop.
- This algorithm has been subjected to a thorough mathematical analysis, a very precise statement can be made about performance issues.

Disadvantages

- It is recursive. Especially, if recursion is not available, the implementation is extremely complicated.
- It requires quadratic (i.e., n2) time in the worst-case.

- It is fragile, i.e. a simple mistake in the implementation can go unnoticed and cause it to perform badly.

Quick sort works by partitioning a given array ***A[p ... r]*** into two non-empty sub array ***A[p ... q]*** and ***A[q+1 ... r]*** such that every key in ***A[p ... q]*** is less than or equal to every key in ***A[q+1 ... r]***.

Then, the two sub-arrays are sorted by recursive calls to Quick sort. The exact position of the partition depends on the given array and index ***q*** is computed as a part of the partitioning procedure.

Algorithm: Quick-Sort (A, p, r)

```
if p < r then
  q Partition (A, p, r)
  Quick-Sort (A, p, q)
  Quick-Sort (A, q + r, r)
```

Note that to sort the entire array, the initial call should be ***Quick-Sort (A, 1, length[A])***

As a first step, Quick Sort chooses one of the items in the array to be sorted as pivot. Then, the array is partitioned on either side of the pivot. Elements that are less than or equal to pivot will move towards the left, while the elements that are greater than or equal to pivot will move towards the right.

Partitioning the Array

Partitioning procedure rearranges the sub-arrays in-place.

Function: Partition (A, p, r)

```
x ← A[p]
i ← p-1
j ← r+1
while TRUE do
  Repeat j ← j - 1
  until A[j] ≤ x
  Repeat i← i+1
  until A[i] ≥ x
  if i < j then
    exchange A[i] ↔ A[j]
  else
    return j
```

Analysis

The worst case complexity of Quick-Sort algorithm is ***O(n^2)***. However using this technique, in average cases generally we get the output in ***O(n log n)*** time.

Radix Sort

Radix sort is a small method that many people intuitively use when alphabetizing a large list of names. Specifically, the list of names is first sorted according to the first letter of each name, that is, the names are arranged in 26 classes.

Intuitively, one might want to sort numbers on their most significant digit. However, Radix sort works counter-intuitively by sorting on the least significant digits first. On the first pass, all the numbers are sorted on the least significant digit and combined in an array. Then on the second pass, the entire numbers are sorted again on the second least significant digits and combined in an array and so on.

Algorithm: Radix-Sort (list, n)

```
shift = 1
for loop = 1 to keysize do
   for entry = 1 to n do
      bucketnumber = (list[entry].key / shift) mod 10
      append (bucket[bucketnumber], list[entry])
   list = combinebuckets()
   shift = shift * 10
```

Analysis

Each key is looked at once for each digit (or letter if the keys are alphabetic) of the longest key. Hence, if the longest key has **m** digits and there are **n** keys, radix sort has order **O(m.n)**.

However, if we look at these two values, the size of the keys will be relatively small when compared to the number of keys. For example, if we have six-digit keys, we could have a million different records.

Here, we see that the size of the keys is not significant, and this algorithm is of linear complexity **O(n)**.

Example

Following example shows how Radix sort operates on seven 3-digits number.

Input	**1st Pass**	**2nd Pass**	**3rd Pass**
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

In the above example, the first column is the input. The remaining columns show the list after successive sorts on increasingly significant digits position. The code for Radix sort assumes that each element in an array ***A*** of ***n*** elements has ***d*** digits, where digit ***1*** is the lowest-order digit and ***d*** is the highest-order digit.

UNIT 5

Backtracking

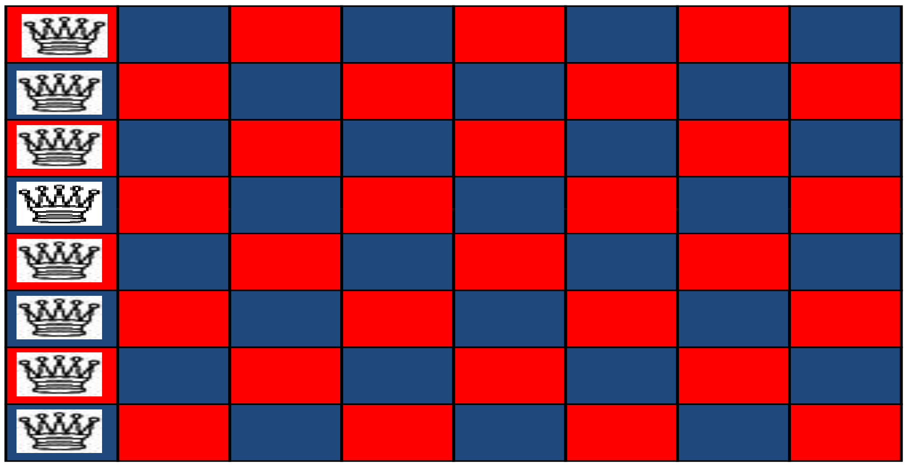
i) EightQueens Problem

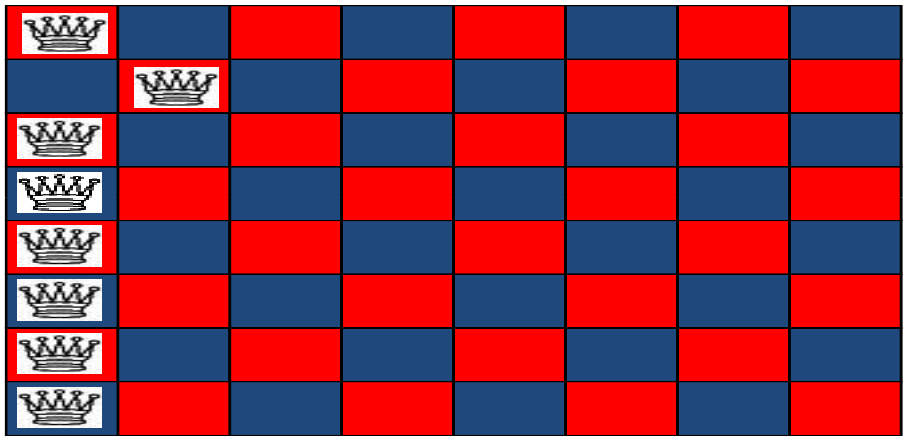
ii) Graph Coloring

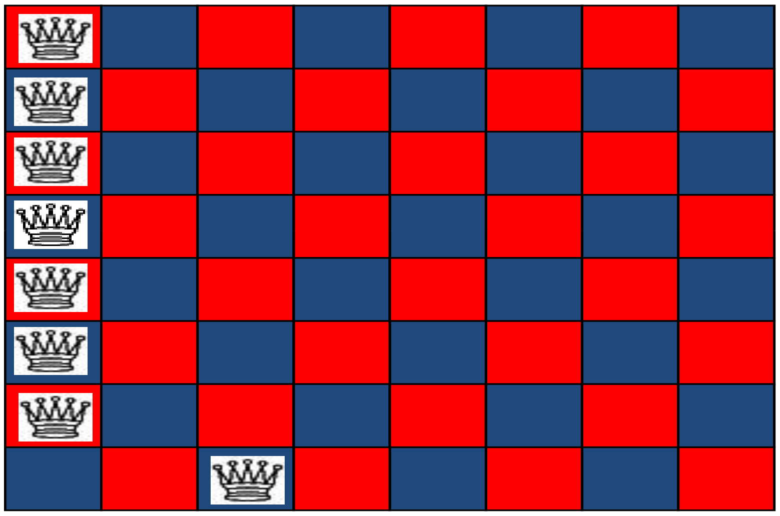
iii)Hamilton Cycles

iv)Knapsack Problem

EIGHT QUEENS PROBLEM:







The first solution:

Consider every possible placements

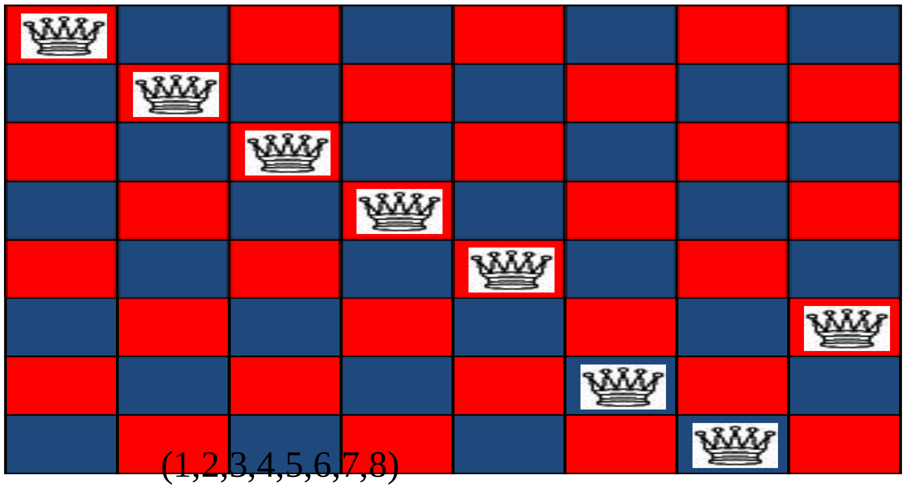
Second solution idea:

- Don’t place 2 queens in the same row.
  - Now how many positions must be checked? Represent a positioning as a vector [x_1, …, x_8] Where each element is an integer 1, …, 8.

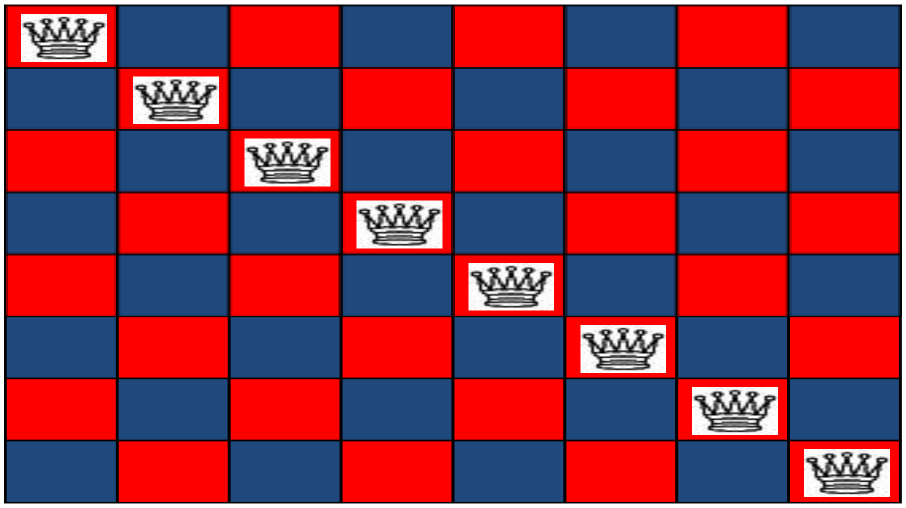
$$n^n = 8^8 = 16{,}777{,}216$$

Third solution idea:

- Don’t place 2 queens in the same row or in the same column.
- Generate all permutations of (1,2…8) Now how many positions must be checked



(1,2,3,4,5,6,7,8)



(1.2.3.4.5.8.6.7)

Third Solution Idea

- *Don't place 2 queens in the same row or in the same column.*
- Generate all permutations of (1,2…8)
- – *Now how many positions must be checked? We went from*

$C(n^2, n)$ to n^n to $n!$

  - And we're happy about it!

- We applied *explicit constraints* to shrink our search space.

N!=8!=40,320

Eight queens problem – Place

Return true if a queen can be placed in Kth row and ith column otherwise false x[] is a global array whose first (k-1) value have been set. Abs(x) returns absolute value of r

Eight Queens problem

Two queens are placed at positions (i ,j) and (k ,l).

They are on the same diagonal

GRAPH COLORING

Graph Coloring Problem

- Assign colors to the vertices of a graph so that no adjacent vertices share the same color
  - Vertices i, j are adjacent if there is an edge from vertex *i* to vertex *j*.
- Find all *m*-colorings of a graph
  - Find all ways to color a graph with at most *m*

    colors.
  - m is called chromatic number

The *m*-Coloring problem

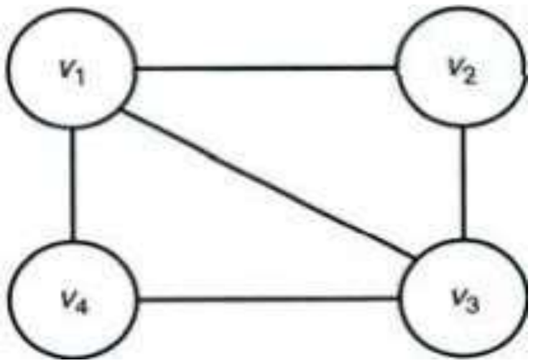
Finding all ways to color an undirected graph using at most *m* different colors, so that no two adjacent vertices are the same color.

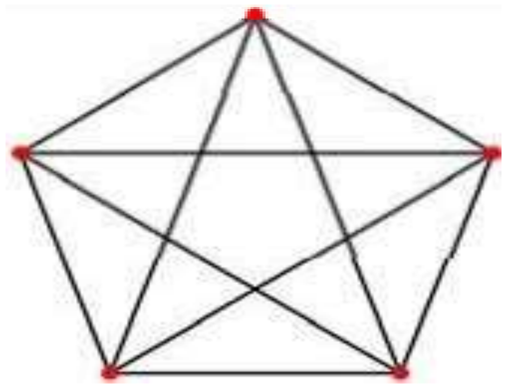
Usually the *m*-Coloring problem consider as a

unique problem for each value of *m.*

Planar graph

It can be drawn in a plane in such a way that no two edges cross each other.

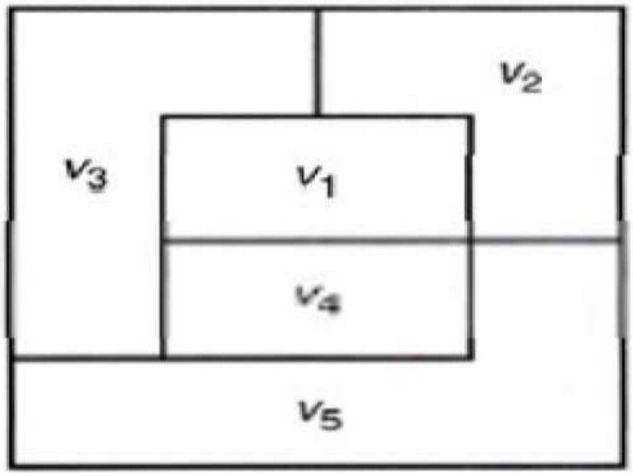




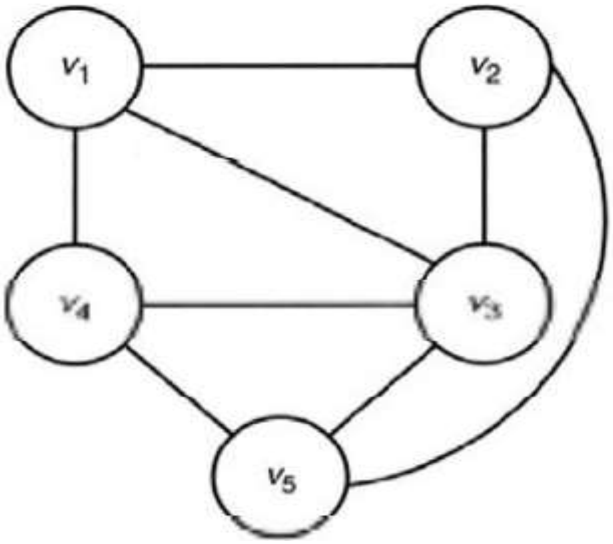
m_coloring(0)

- `The number of nodes in the state space tree for this algorithm

CORRESPONDED PLAN GRAPH



This algorithm was formed using recursive backtracking schema. The graph is



represented by its boolean adjacency matrix G[1:n,1:n]. All assignment of 1,2..,m to the vertices of the graph such that adjacent vertices are assigned distinct integerare printed. K is the index of the next vertex to color

```
1.    Algorithm mcoloring(k)
2.    {     repeat
3.          {                              // generate all legal
            assignment for x[k]
4.    nextvalue(k);                        //assign to x[k] a legal value
5.    If (x[k] = 0 ) then return ; //no new color possible
7.    If ( k = n) then                // at most m color have been used to
      color the n vertices
8.    Write (x[1 : n];
9.    Else mcoloring(k+1)
10.       } until (false)
11.   }
```

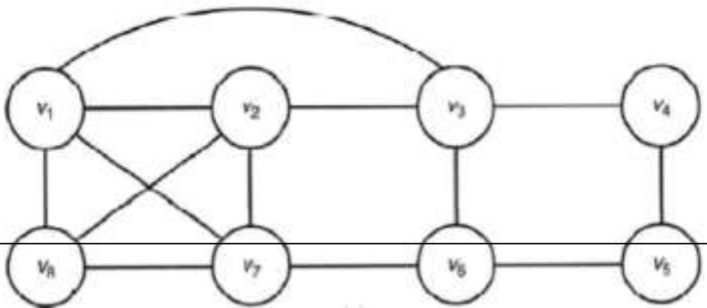
X[1]…x[k-1] have been assigned integer value in the range [1,m] such that adjacent vertices have distinct integer. A value for x[k] is determined in the range [0,m]. X[k] is assigned the next highest numbered color while maintaining distinctness from the adj. vertices of vertex k. if no such color exists, then x[k] is 0

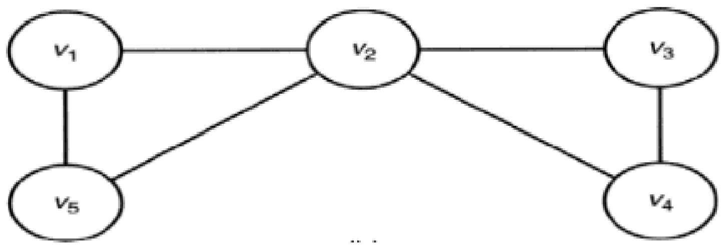
```
1.    Algorithm Nextvalue(k)
2.    {     repeat
3.          {      X [k]:= (x[k] +1) mod (m+1) ;
          //next higher color
4.                 If ( x[k] = 0) then return;                  // all
                   colors have been used
5.         for j := 1 to n do
6.          {                          // check if this color is distinct from
adjacent colors 7.                            If ((G[k, j] != 0) and (x[k] = x[ j ]
))
8.                          //if (k , j)is and edge if adj. vertices have the
                            samecolor.
9.          then break;
10.        }
11.        If (j = n+1) then return;                    //new color
           found
12.       } until (false);                              //otherwise try
          to find another color
}
```

Hamiltonian cycle (HC)

Definitions

- Hamiltonian cycle (HC): is a cycle which passes once and exactly once through every vertex of G and returns to starting position
- Hamiltonian path: is a path which passes once and exactly once through every vertex of G (G can be digraph).
- A graph is Hamiltonian iff a Hamiltonian cycle (HC) exists.
  - Hamiltonian Circuit
  - [v1, v2, v8, v7, v6, v5, v4, v3, v2]





BACKTRACKM ALGORITHM:

- Search all the potential solutions
- Employ pruning of some kind to restrict the amount of researching
- Advantage:

  Find all solution, can decide HC exists or not

- Disadvantage

  Worst case, needs exponential time. Normally, take a long time

  APPLICATION:

- Hamiltonian cycles in fault random geometric network
- In a network, if Hamiltonian cycles exist, the fault tolerance is better.

HEURISTIC ALGORITHM

{

```
Find new unvisited node.
If found { Extend path P and pruning on the graph. If
   this choice does not permit HC, remove the
   extendednode.
} else
    Transform Path. Try all possible endpoints of this path
 Form cycle. Try to find HC
```

This algorithm uses the recursive formulation of backtracking to find all the hamiltonion cycles of a graph. The graph is stored as an adjacency matrix G[1:n, 1:n]. All cycles begins at node 1.

```
{
3.      repeat
4.          {
            //generate values for x[k]
5.              Nextvalue(k);        //assign a legal next value to x[k]
6.              if ( x[k] = 0 ) then return
7.              if (k = n) then write ( x[1:n]);
8.                  else Hamiltonian(k + 1);
9.          } until(false);
10.  }
```

Knapsack:

Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

The knapsack problem is in combinatorial optimization problem. It appears as a subproblem in many, more complex mathematical models of real-world problems. One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.

Applications

In many cases of resource allocation along with some constraint, the problem can be derived in a similar way of Knapsack problem. Following is a set of example.

- Finding the least wasteful way to cut raw materials
- portfolio optimization
- Cutting stock problems

Problem Scenario

A thief is robbing a store and can carry a maximal weight of ***W*** into his knapsack. There are n items available in the store and weight of $\boldsymbol{i^{th}}$ item is $\boldsymbol{w_i}$ and its profit is $\boldsymbol{p_i}$. What items should the thief take?

In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.

Based on the nature of the items, Knapsack problems are categorized as

- Fractional Knapsack
- Knapsack

Fractional Knapsack

In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.

According to the problem statement,

- There are **n** items in the store

- Weight of **i**th item wi>0wi>0
- Profit for **i**th item pi>0pi>0 and
- Capacity of the Knapsack is **W**

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction ***x**i* of **i**th item.

0⩽xi⩽10⩽xi⩽1

The **i**th item contributes the weight xi.wixi.wi to the total weight in the knapsack and profit xi.pixi.pi to the total profit.

Hence, the objective of this algorithm is to

maximize∑n=1n(xi.pi)maximize∑n=1n(xi.pi)

subject to constraint,

∑n=1n(xi.wi)⩽W∑n=1n(xi.wi)⩽W

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

∑n=1n(xi.wi)=W∑n=1n(xi.wi)=W

In this context, first we need to sort those items according to the value of piwipiwi, so that pi+1wi+1pi+1wi+1 ≤ piwipiwi . Here, ***x*** is an array to store the fraction of items.

```
Algorithm: Greedy-Fractional-Knapsack (w[1..n],
p[1..n], W)
for i = 1 to n
   do x[i] = 0
weight = 0
for i = 1 to n
   if weight + w[i] ≤ W then
      x[i] = 1
      weight = weight + w[i]
   else
      x[i] = (W - weight) / w[i]
      weight = W
      break
```

```
return x
```

Analysis

If the provided items are already sorted into a decreasing order of piwipiwi, then the whileloop takes a time in ***O(n)***; Therefore, the total time including the sort is in ***O(n logn)***.

Example

Let us consider that the capacity of the knapsack ***W* = 60** and the list of provided items are shown in the following table −

Item	**A**	**B**
Profit	280	100
Weight	40	10
Ratio (piwi)(piwi)	7	10

As the provided items are not sorted based on piwipiwi. After sorting, the items are as shown in the following table.

Item	**B**	**A**
Profit	100	280
Weight	10	40

Ratio (piwi)(piwi)	10	7

Solution

After sorting all the items according to piwipiwi. First all of ***B*** is chosen as weight of ***B*** is less than the capacity of the knapsack. Next, item ***A*** is chosen, as the available capacity of the knapsack is greater than the weight of ***A***. Now, ***C*** is chosen as the next item. However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of ***C***.

Hence, fraction of ***C*** (i.e. (60 − 50)/20) is chosen.

Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected.

The total weight of the selected items is **10 + 40 + 20 * (10/20) = 60**

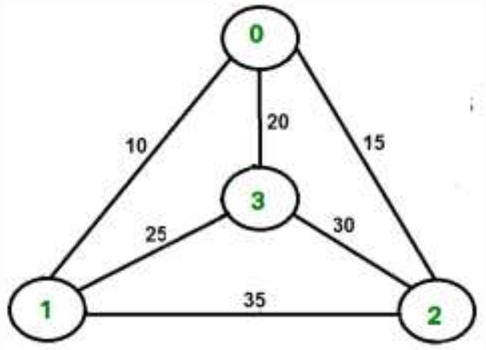
And the total profit is **100 + 280 + 120 * (10/20) = 380 + 60 = 440**

This is the optimal solution. We cannot gain more profit selecting any different combination of items.

BRANCH AND BOUND:

Traveling Salesman Problem using Branch And Bound

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible tour that visits every city exactly once and returns to the starting point.



For example, consider the graph shown in figure on right side. A TSP tour in the graph is 0-1-3-2-0. The cost of the tour is 10+25+30+15 which is 80.

We have discussed following solutions
1) Naive and Dynamic Programming
2) Approximate solution using MST

Branch and Bound Solution
As seen in the previous articles, in Branch and Bound method, for current node in tree, we compute a bound on best possible solution that we can get if we down this node. If the bound on best possible solution itself is worse than current best (best computed so far), then we ignore the subtree rooted with the node.

Note that the cost through a node includes two costs.
1) Cost of reaching the node from the root (When we reach a node, we have this cost computed)
2) Cost of reaching an answer from current node to a leaf (We compute a bound on this cost to decide whether to ignore subtree with this node or not).

- In cases of a **maximization problem**, an upper bound tells us the maximum possible solution if we follow the given node. For example in 0/1 knapsack we used Greedy approach to find an upper bound.
- In cases of a **minimization problem**, a lower bound tells us the minimum possible solution if we follow the given node. For example, in Job Assignment Problem, we get a lower bound by assigning least cost job to a worker.

In branch and bound, the challenging part is figuring out a way to compute a bound on best possible solution. Below is an idea used to compute bounds for Traveling salesman problem.

Cost of any tour can be written as below.

```
Cost of a tour T = (1/2) * &Sum; (Sum of cost of two edges

                                adjacent to u and in the

                                tour T)

                  where u ∈ V
```

```
For every vertex u, if we consider two edges through it in
T,

and sum their costs.  The overall sum for all vertices would

be twice of cost of tour T (We have considered every edge

twice.)


(Sum of two tour edges adjacent to u) >= (sum of minimum
weight

                                          two edges adjacent
to

                                          u)


Cost of any tour >=  1/2) * &Sum; (Sum of cost of two
minimum

                              weight edges adjacent to u)

                   where u ∈ V
```

For example, consider the above shown graph. Below are minimum cost two edges adjacent to every node.

```
Node     Least cost edges   Total cost

0     (0, 1), (0, 2)            25

1     (0, 1), (1, 3)         35

2    (0, 2), (2, 3)           45

3     (0, 3), (1, 3)            45
```

```
Thus a lower bound on the cost of any tour =

         1/2(25 + 35 + 45 + 45)

       = 75

Refer this for one more example.
```

Now we have an idea about computation of lower bound. Let us see how to how to apply it state space search tree. We start enumerating all possible nodes (preferably in lexicographical order)

1. The Root Node: Without loss of generality, we assume we start at vertex "0" for which the lower bound has been calculated above.
Dealing with Level 2: The next level enumerates all possible vertices we can go to (keeping in mind that in any path a vertex has to occur only once) which are, 1, 2, 3… n (Note that the graph is complete). Consider we are calculating for vertex 1, Since we moved from 0 to 1, our tour has now included the edge 0-1. This allows us to make necessary changes in the lower bound of the root.

```
Lower Bound for vertex 1 =

   Old lower bound - ((minimum edge cost of 0 +

                    minimum edge cost of 1) / 2)

                  + (edge cost 0-1)
```

How does it work? To include edge 0-1, we add the edge cost of 0-1, and subtract an edge weight such that the lower bound remains as tight as possible which would be the sum of the minimum edges of 0 and 1 divided by 2. Clearly, the edge subtracted can't be smaller than this.

Dealing with other levels: As we move on to the next level, we again enumerate all possible vertices. For the above case going further after 1, we check out for 2, 3, 4, …n.
Consider lower bound for 2 as we moved from 1 to 1, we include the edge 1-2 to the tour and alter the new lower bound for this node.

```
Lower bound(2) =
     Old lower bound - ((second minimum edge cost of 1 +
                         minimum edge cost of 2)/2)
                     + edge cost 1-2)
```

0/1 KNAPSACK PROBLEM:

In this tutorial, earlier we have discussed Fractional Knapsack problem using Greedy approach. We have shown that Greedy approach gives an optimal solution for Fractional Knapsack. However, this chapter will cover 0-1 Knapsack problem and its analysis.

In 0-1 Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack.

Hence, in case of 0-1 Knapsack, the value of x_i can be either ***0*** or ***1***, where other constraints remain the same.

0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution.

The following examples will establish our statement.

Example-1

Let us consider that the capacity of the knapsack is W = 25 and the items are as shown in the following table.

Without considering the profit per unit weight (p_i/w_i), if we apply Greedy approach to solve this problem, first item ***A*** will be selected as it will contribute max$_i$mum profit among all the elements.

After selecting item ***A***, no more item will be selected. Hence, for this given set of items total profit is ***24***. Whereas, the optimal solution can be achieved by selecting items, ***B*** and C, where the total profit is 18 + 18 = 36.

Example-2

Instead of selecting the items based on the overall benefit, in this example the items are selected based on ratio p_i/w_i. Let us consider that the capacity of the knapsack is $W = 60$ and the items are as shown in the following table.

Using the Greedy approach, first item ***A*** is selected. Then, the next item ***B*** is chosen. Hence, the total profit is **100 + 280 = 380**. However, the optimal solution of this instance can be achieved by selecting items, ***B*** and ***C***, where the total profit is **280 + 120 = 400**.

Hence, it can be concluded that Greedy approach may not give an optimal solution.

To solve 0-1 Knapsack, Dynamic Programming approach is required.

Problem Statement

A thief is robbing a store and can carry a maxᵢmal weight of ***W*** into his knapsack. There are ***n*** items and weight of **i**th item is $\boldsymbol{w_i}$ and the profit of selecting this item is $\boldsymbol{p_i}$. What items should the thief take?

Dynamic-Programming Approach

Let ***i*** be the highest-numbered item in an optimal solution **S** for **W** dollars. Then $\boldsymbol{S' = S - \{i\}}$ is an optimal solution for $\boldsymbol{W - w_i}$ dollars and the value to the solution ***S*** is $\boldsymbol{V_i}$ plus the value of the sub-problem.

We can express this fact in the following formula: define **c[i, w]** to be the solution for items **1,2, … , i** and the maxᵢmum weight **w**.

The algorithm takes the following inputs

- The maxᵢmum weight **W**
- The number of items **n**
- The two sequences $\mathbf{v = <v_1, v_2, …, v_n>}$ and $\mathbf{w = <w_1, w_2, …, w_n>}$

```
Dynamic-0-1-knapsack (v, w, n, W)
for w = 0 to W do
```

```
   c[0, w] = 0
for i = 1 to n do
   c[i, 0] = 0
   for w = 1 to W do
      if $w_i \leq$ w then
         if $v_i$ + c[i-1, w-$w_i$] then
            c[i, w] = $v_i$ + c[i-1, w-$w_i$]
         else c[i, w] = c[i-1, w]
      else
         c[i, w] = c[i-1, w]
```

The set of items to take can be deduced from the table, starting at **c[n, w]** and tracing backwards where the optimal values came from.

If *c[i, w] = c[i-1, w]*, then item ***i*** is not part of the solution, and we continue tracing with **c[i-1, w]**. Otherwise, item ***i*** is part of the solution, and we continue tracing with **c[i-1, w-W]**.

Analysis

This algorithm takes θ(*n*, *w*) times as table *c* has (*n* + 1).(*w* + 1) entries, where each entry requires θ(1) time to compute.

Subsequence

Let us consider a sequence S = <s_1, s_2, s_3, s_4, …,s_n>.

A sequence Z = <z_1, z_2, z_3, z_4, …,z_m> over S is called a subsequence of S, if and only if it can be derived from S deletion of some elements.

Common Subsequence

Suppose, ***X*** and ***Y*** are two sequences over a finite set of elements. We can say that ***Z*** is a common subsequence of ***X*** and ***Y***, if ***Z*** is a subsequence of both ***X*** and ***Y***.

Longest Common Subsequence

If a set of sequences are given, the longest common subsequence problem is to find a common subsequence of all the sequences that is of maximal length.

The longest common subsequence problem is a classic computer science problem, the basis of data comparison programs such as the diff-utility, and has applications in bioinformatics. It is also widely used by revision control systems, such as SVN and Git, for reconciling multiple changes made to a revision-controlled collection of files.

Naïve Method

Let ***X*** be a sequence of length ***m*** and ***Y*** a sequence of length ***n***. Check for every subsequence of ***X*** whether it is a subsequence of ***Y***, and return the longest common subsequence found.

There are $\boldsymbol{2^m}$ subsequences of ***X***. Testing sequences whether or not it is a subsequence of ***Y*** takes ***O(n)*** time. Thus, the naïve algorithm would take $\boldsymbol{O(n2^m)}$ time

SUM OF SUBSETS:

The Subset-Sum Problem is to find a subset's' of the given set S = (S_1 S_2 S_3...S_n) where the elements of the set S are n positive integers in such a manner that s'∈S and sum of the elements of subset's' is equal to some positive integer 'X.'

The Subset-Sum Problem can be solved by using the backtracking approach. In this implicit tree is a binary tree. The root of the tree is selected in such a way that represents that no decision is yet taken on any input. We assume that the elements of the given set are arranged in increasing order:

$S_1 \leq S_2 \leq S_3 ... \leq S_n$

The left child of the root node indicated that we have to include 'S_1' from the set 'S' and the right child of the root indicates that we have to execute 'S_1'. Each node stores the total of the partial solution elements. If at any stage the sum equals to 'X' then the search is successful and terminates.

The dead end in the tree appears only when either of the two inequalities exists:

- The sum of s' is too large i.e.

$s' + S_i + 1 > X$

- The sum of s' is too small i.e.

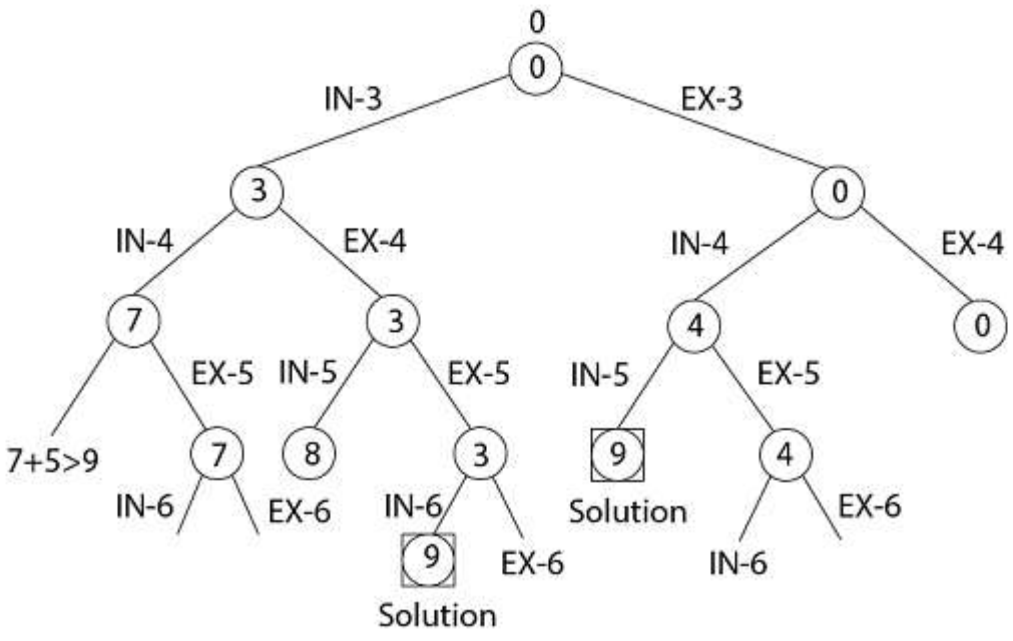
$s' + \sum_{j=i+1}^{n} S_j < X$

Example: Given a set S = (3, 4, 5, 6) and X =9. Obtain the subset sum using Backtracking approach.

Solution:

1. Initially S = (3, 4, 5, 6) and X =9.
2. S'= (∅)

The implicit binary tree for the subset sum problem is shown as fig:



The number inside a node is the sum of the partial solution elements at a particular level.

Thus, if our partial solution elements sum is equal to the positive integer 'X' then at that time search will terminate, or it continues if all the possible solution needs to be obtained.

Let, S = {S1 …. Sn} be a set of n positive integers, then we have to find a subset whose sum is equal to given positive integer d.It is always convenient to sort the set's elements in ascending order. That is, S1 ≤ S2 ≤…. ≤ Sn

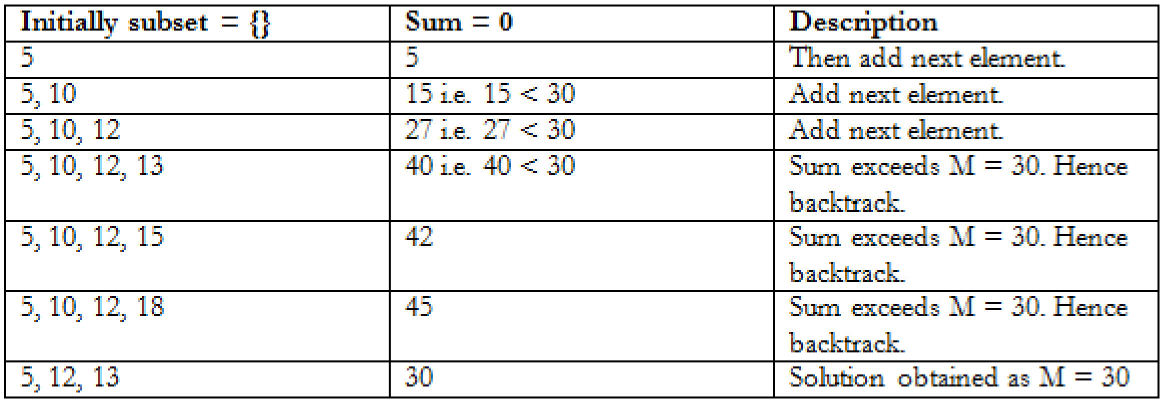
Algorithm:

Let, S is a set of elements and m is the expected sum of subsets. Then:

1. Start with an empty set.
2. Add to the subset, the next element from the list.
3. If the subset is having sum m then stop with that subset as solution.
4. If the subset is not feasible or if we have reached the end of the set then backtrack through the subset until we find the most suitable value.
5. If the subset is feasible then repeat step 2.
6. If we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without solution.

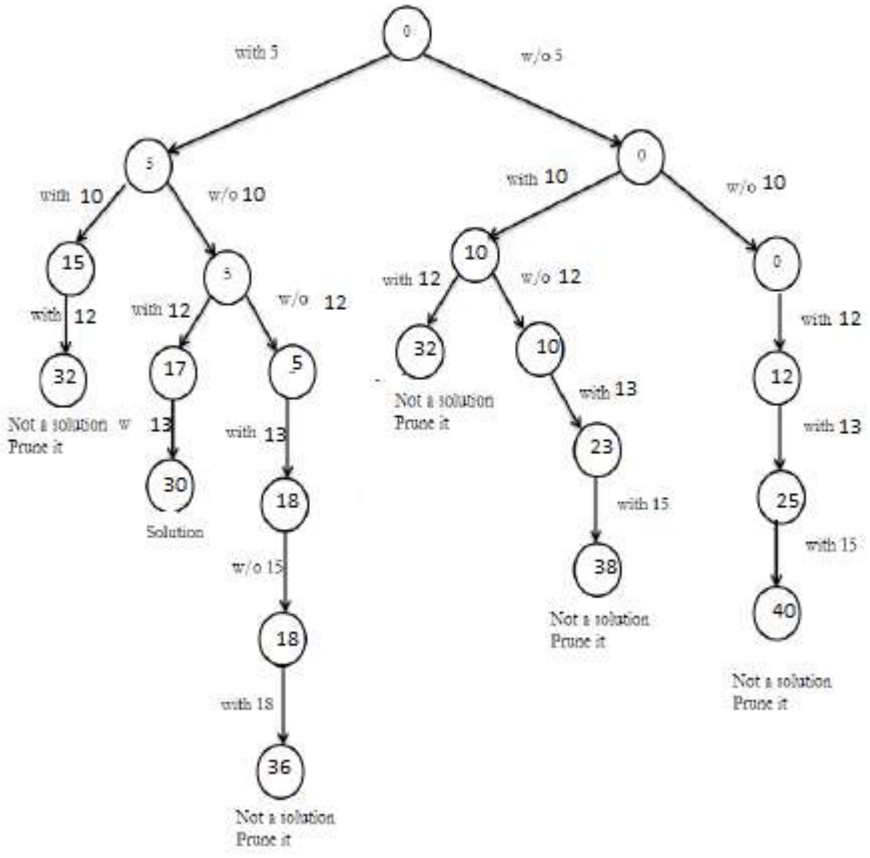
Example: Solve following problem and draw portion of state space tree M=30,W ={5, 10, 12, 13, 15, 18}

Solution:



Initially subset = {}	**Sum = 0**	**Description**
5	5	Then add next element.
5, 10	15 i.e. 15 < 30	Add next element.
5, 10, 12	27 i.e. 27 < 30	Add next element.
5, 10, 12, 13	40 i.e. 40 < 30	Sum exceeds M = 30. Hence backtrack.
5, 10, 12, 15	42	Sum exceeds M = 30. Hence backtrack.
5, 10, 12, 18	45	Sum exceeds M = 30. Hence backtrack.
5, 12, 13	30	Solution obtained as M = 30

The state space tree is shown as below in figure. {5, 10, 12, 13, 15, 18}


0
with 5
w/o 5
5
0
with 10
w/o 10
with 10
w/o 10
15
5
10
0
with 12
with 12
w/o 12
with 12
w/o 12
with 12
32
17
5
32
10
12
Not a solution
Prune it
w 13
with 13
Not a solution
Prune it
with 13
with 13
30
Solution
18
23
25
w/o 15
with 15
with 15
18
38
40
with 18
Not a solution
Prune it
Not a solution
Prune it
36
Not a solution
Prune it