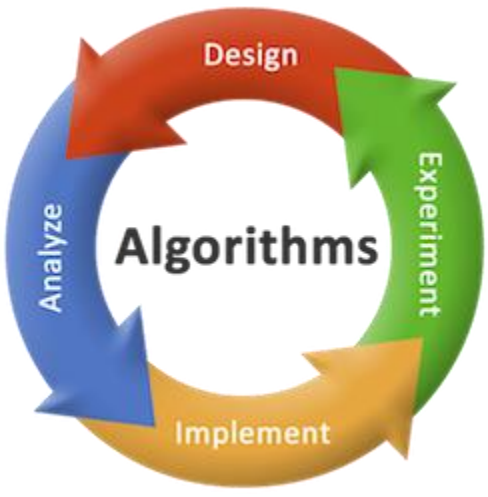
DAA Tutorial



Our DAA Tutorial is designed for beginners and professionals both.

Our DAA Tutorial includes all topics of algorithm, asymptotic analysis, algorithm control structure, recurrence, master method, recursion tree method, simple sorting algorithm, bubble sort, selection sort, insertion sort, divide and conquer, binary search, merge sort, counting sort, lower bound theory etc.

What is Algorithm?

A finite set of instruction that specifies a sequence of operation is to be carried out in order to solve a specific problem or class of problems is called an Algorithm.

Why study Algorithm?

As the speed of processor increases, performance is frequently said to be less central than other software quality characteristics (e.g. security, extensibility, reusability etc.). However, large problem sizes are commonplace in the area of computational science, which makes performance a very important factor. This is because longer computation time, to name a few mean slower results, less through research and higher cost of computation (if buying CPU Hours from an external party). The study of Algorithm, therefore, gives us a language to express performance as a function of problem size.

DAA Algorithm

The word algorithm has been derived from the Persian author's name, Abu Ja 'far Mohammed ibn Musa al Khowarizmi (c. 825 A.D.), who has written a textbook on Mathematics. The word is taken based on providing a special significance in computer science. The algorithm is understood as a method that can be utilized by the computer as when required to provide solutions to a particular problem.

An algorithm can be defined as a finite set of steps, which has to be followed while carrying out a particular problem. It is nothing but a process of executing actions step by step.

An algorithm is a distinct computational procedure that takes input as a set of values and results in the output as a set of values by solving the problem. More precisely, an algorithm is correct, if, for each input instance, it gets the correct output and gets terminated.

An algorithm unravels the computational problems to output the desired result. An algorithm can be described by incorporating a natural language such as English, Computer language, or a hardware language.

Characteristics of Algorithms

- **Input:** It should externally supply zero or more quantities.
- **Output:** It results in at least one quantity.
- **Definiteness:** Each instruction should be clear and ambiguous.
- **Finiteness:** An algorithm should terminate after executing a finite number of steps.
- **Effectiveness:** Every instruction should be fundamental to be carried out, in principle, by a person using only pen and paper.
- **Feasible:** It must be feasible enough to produce each instruction.
- **Flexibility:** It must be flexible enough to carry out desired changes with no efforts.
- **Efficient:** The term efficiency is measured in terms of time and space required by an algorithm to implement. Thus, an algorithm must ensure that it takes little time and less memory space meeting the acceptable limit of development time.
- **Independent:** An algorithm must be language independent, which means that it should mainly focus on the input and the procedure required to derive the output instead of depending upon the language.

Advantages of an Algorithm

- **Effective Communication:** Since it is written in a natural language like English, it becomes easy to understand the step-by-step delineation of a solution to any particular problem.
- **Easy Debugging:** A well-designed algorithm facilitates easy debugging to detect the logical errors that occurred inside the program.
- **Easy and Efficient Coding:** An algorithm is nothing but a blueprint of a program that helps develop a program.
- **Independent of Programming Language:** Since it is a language-independent, it can be easily coded by incorporating any high-level language.

Disadvantages of an Algorithm

- Developing algorithms for complex problems would be time-consuming and difficult to understand.
- It is a challenging task to understand complex logic through algorithms.

Pseudocode

Pseudocode refers to an informal high-level description of the operating principle of a computer program or other algorithm. It uses structural conventions of a standard programming language intended for human reading rather than the machine reading.

Advantages of Pseudocode

- Since it is similar to a programming language, it can be quickly transformed into the actual programming language than a flowchart.
- The layman can easily understand it.
- Easily modifiable as compared to the flowcharts.
- Its implementation is beneficial for structured, designed elements.
- It can easily detect an error before transforming it into a code.

Disadvantages of Pseudocode

- Since it does not incorporate any standardized style or format, it can vary from one company to another.
- Error possibility is higher while transforming into a code.
- It may require a tool for extracting out the Pseudocode and facilitate drawing flowcharts.
- It does not depict the design.

Difference between Algorithm and the Pseudocode

An algorithm is simply a problem-solving process, which is used not only in computer science to write a program but also in our day to day life. It is nothing but a series of instructions to solve a problem or get to the problem's solution. It not only helps in simplifying the problem but also to have a better understanding of it.

However, Pseudocode is a way of writing an algorithm. Programmers can use informal, simple language to write pseudocode without following any strict syntax. It encompasses semi-mathematical statements.

Problem: Suppose there are 60 students in the class. How will you calculate the number of absentees in the class?

Pseudo Approach:

1. Initialize a variable called as **Count** to zero, **absent** to zero, **total** to 60
2. FOR EACH Student PRESENT DO the following:
   Increase the **Count** by One
3. Then Subtract **Count** from **total** and store the result in **absent**
4. Display the number of absent students

Algorithmic Approach:

1. Count <- 0, absent <- 0, total <- 60
2. REPEAT till all students counted
   Count <- Count + 1

3. absent <- total - Count
4. Print "Number absent is:" , absent

Need of Algorithm

1. To understand the basic idea of the problem.

2. To find an approach to solve the problem.

3. To improve the efficiency of existing techniques.

4. To understand the basic principles of designing the algorithms.

5. To compare the performance of the algorithm with respect to other techniques.

6. It is the best method of description without describing the implementation detail.

7. The Algorithm gives a clear description of requirements and goal of the problem to the designer.

8. A good design can produce a good solution.

9. To understand the flow of the problem.

10. To measure the behavior (or performance) of the methods in all cases (best cases, worst cases, average cases)

11. With the help of an algorithm, we can also identify the resources (memory, input-output) cycles required by the algorithm.

12. With the help of algorithm, we convert art into a science.

13. To understand the principle of designing.

14. We can measure and analyze the complexity (time and space) of the problems concerning input size without implementing and running it; it will reduce the cost of design.

Analysis of algorithm

The analysis is a process of estimating the efficiency of an algorithm. There are two fundamental parameters based on which we can analysis the algorithm:

- **Space Complexity:** The space complexity can be understood as the amount of space required by an algorithm to run to completion.
- **Time Complexity:** Time complexity is a function of input size **n** that refers to the amount of time needed by an algorithm to run to completion.

Let's understand it with an example.

Suppose there is a problem to solve in Computer Science, and in general, we solve a program by writing a program. If you want to write a program in some programming language like C, then before writing a program, it is necessary to write a blueprint in an informal language.

Or in other words, you should describe what you want to include in your code in an English-like language for it to be more readable and understandable before implementing it, which is nothing but the concept of Algorithm.

In general, if there is a problem **P1**, then it may have many solutions, such that each of these solutions is regarded as an algorithm. So, there may be many algorithms such as $\mathbf{A_1, A_2, A_3}$, ..., $\mathbf{A_n}$.

Before you implement any algorithm as a program, it is better to find out which among these algorithms are good in terms of time and memory.

It would be best to analyze every algorithm in terms of **Time** that relates to which one could execute faster and **Memory** corresponding to which one will take less memory.

So, the Design and Analysis of Algorithm talks about how to design various algorithms and how to analyze them. After designing and analyzing, choose the best algorithm that takes the least time and the least memory and then implement it as a program in C.

In this course, we will be focusing more on time rather than space because time is instead a more limiting parameter in terms of the hardware. It is not easy to take a computer and change its speed. So, if we are running an algorithm on a particular platform, we are more or less stuck with the performance that platform can give us in terms of speed.

However, on the other hand, memory is relatively more flexible. We can increase the memory as when required by simply adding a memory card. So, we will focus on time than that of the space.

The running time is measured in terms of a particular piece of hardware, not a robust measure. When we run the same algorithm on a different computer or use different programming languages, we will encounter that the same algorithm takes a different time.

Generally, we make three types of analysis, which is as follows:

- **Worst-case time complexity:** For '**n**' input size, the worst-case time complexity can be defined as the maximum amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the maximum number of steps performed on an instance having an input size of n.
- **Average case time complexity:** For '**n**' input size, the average-case time complexity can be defined as the average amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the average number of steps performed on an instance having an input size of n.
- **Best case time complexity:** For '**n**' input size, the best-case time complexity can be defined as the minimum amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the minimum number of steps performed on an instance having an input size of n.

Complexity of Algorithm

The term algorithm complexity measures how many steps are required by the algorithm to solve the given problem. It evaluates the order of count of operations executed by an algorithm as a function of input data size.

To assess the complexity, the order (approximation) of the count of operation is always considered instead of counting the exact steps.

O(f) notation represents the complexity of an algorithm, which is also termed as an Asymptotic notation or "**Big O**" notation. Here the f corresponds to the function whose size is the same as that of the input data. The complexity of the asymptotic computation O(f) determines in which order the resources such as CPU time, memory, etc. are consumed by the algorithm that is articulated as a function of the size of the input data.

The complexity can be found in any form such as constant, logarithmic, linear, n*log(n), quadratic, cubic, exponential, etc. It is nothing but the order of constant, logarithmic, linear and so on, the number of steps encountered for the completion of a particular algorithm. To make it even more precise, we often call the complexity of an algorithm as "running time".

Typical Complexities of an Algorithm

- **Constant Complexity:**
  It imposes a complexity of **O(1)**. It undergoes an execution of a constant number of steps like 1, 5, 10, etc. for solving a given problem. The count of operations is independent of the input data size.
- **Logarithmic Complexity:**
  It imposes a complexity of **O(log(N))**. It undergoes the execution of the order of log(N) steps. To perform operations on N elements, it often takes the logarithmic base as 2.
  For N = 1,000,000, an algorithm that has a complexity of O(log(N)) would undergo 20 steps (with a constant precision). Here, the logarithmic base does not hold a necessary consequence for the operation count order, so it is usually omitted.
- **Linear Complexity:**
  - It imposes a complexity of **O(N)**. It encompasses the same number of steps as that of the total number of elements to implement an operation on N elements.
    For example, if there exist 500 elements, then it will take about 500 steps. Basically, in linear complexity, the number of elements linearly depends on the number of steps. For example, the number of steps for N elements can be N/2 or 3*N.
  - It also imposes a run time of **O(n*log(n))**. It undergoes the execution of the order N*log(N) on N number of elements to solve the given problem.
    For a given 1000 elements, the linear complexity will execute 10,000 steps for solving a given problem.

- **Quadratic Complexity:** It imposes a complexity of **O(n^2)**. For N input data size, it undergoes the order of N^2 count of operations on N number of elements for solving a given problem.
  If N = 100, it will endure 10,000 steps. In other words, whenever the order of operation tends to have a quadratic relation with the input data size, it results in quadratic complexity. For example, for N number of elements, the steps are found to be in the order of $3*N^2/2$.
- **Cubic Complexity:** It imposes a complexity of **O(n^3)**. For N input data size, it executes the order of N^3 steps on N elements to solve a given problem.
  For example, if there exist 100 elements, it is going to execute 1,000,000 steps.
- **Exponential Complexity:** It imposes a complexity of **O(2^n), O(N!), O(n^k), ...**. For N elements, it will execute the order of count of operations that is exponentially dependable on the input data size.
  For example, if N = 10, then the exponential function 2^N will result in 1024. Similarly, if N = 20, it will result in 1048 576, and if N = 100, it will result in a number having 30 digits. The exponential function N! grows even faster; for example, if N = 5 will result in 120. Likewise, if N = 10, it will result in 3,628,800 and so on.

Since the constants do not hold a significant effect on the order of count of operation, so it is better to ignore them. Thus, to consider an algorithm to be linear and equally efficient, it must undergo N, N/2 or 3*N count of operation, respectively, on the same number of elements to solve a particular problem.

How to approximate the time taken by the Algorithm?

So, to find it out, we shall first understand the types of the algorithm we have. There are two types of algorithms:

1. **Iterative Algorithm:** In the iterative approach, the function repeatedly runs until the condition is met or it fails. It involves the looping construct.
2. **Recursive Algorithm:** In the recursive approach, the function calls itself until the condition is met. It integrates the branching structure.

However, it is worth noting that any program that is written in iteration could be written as recursion. Likewise, a recursive program can be converted to iteration, making both of these algorithms equivalent to each other.

But to analyze the iterative program, we have to count the number of times the loop is going to execute, whereas in the recursive program, we use recursive equations, i.e., we write a function of F(n) in terms of F(n/2).

Suppose the program is neither iterative nor recursive. In that case, it can be concluded that there is no dependency of the running time on the input data size, i.e., whatever is the input size, the running time is going to be a constant value. Thus, for such programs, the complexity will be **O(1)**.

For Iterative Programs

Consider the following programs that are written in simple English and does not correspond to any syntax.

Example1:

In the first example, we have an integer i and a for loop running from i equals 1 to n. Now the question arises, how many times does the name get printed?

```
1. A()
2. {
3. int i;
4. for (i=1 to n)
5. printf("Edward");
6. }
```

Since i equals 1 to n, so the above program will print Edward, n number of times. Thus, the complexity will be **O(n)**.

Example2:

```
1. A()
2. {
3. int i, j:
4. for (i=1 to n)
5. for (j=1 to n)
6. printf("Edward");
```

```
7. }
```

In this case, firstly, the outer loop will run n times, such that for each time, the inner loop will also run n times. Thus, the time complexity will be **$O(n^2)$**.

Example3:

```
1. A()
2. {
3. i = 1; S = 1;
4. while (S<=n)
5. {
6. i++;
7. SS = S + i;
8. printf("Edward");
9. }
10. }
```

As we can see from the above example, we have two variables; i, S and then we have while S<=n, which means S will start at 1, and the entire loop will stop whenever S value reaches a point where S becomes greater than n.

Here i is incrementing in steps of one, and S will increment by the value of i, i.e., the increment in i is linear. However, the increment in S depends on the i.

Initially;

i=1, S=1

After 1st iteration;

i=2, S=3

After 2nd iteration;

i=3, S=6

After 3rd iteration;

i=4, S=10 … and so on.

Since we don't know the value of n, so let's suppose it to be k. Now, if we notice the value of S in the above case is increasing; for i=1, S=1; i=2, S=3; i=3, S=6; i=4, S=10; …

Thus, it is nothing but a series of the sum of first n natural numbers, i.e., by the time i reaches k, the value of S will be k(k+1)/2.

To stop the loop, $\frac{k(k+1)}{2}$ has to be greater than n, and when we solve this equation, we will get $\frac{k^2+k}{2} > n$. Hence, it can be concluded that we get a complexity of **$O(\sqrt{n})$** in this case.

For Recursive Program

Consider the following recursive programs.

Example1:

```
1. A(n)
2. {
3. if (n>1)
4. return (A(n-1))
5. }
```

Solution;

Here we will see the simple Back Substitution method to solve the above problem.

T(n) = 1 + T(n-1) …Eqn. (1)

Step1: Substitute n-1 at the place of n in Eqn. (1)

T(n-1) = 1 + T(n-2) ...Eqn. (2)

Step2: Substitute n-2 at the place of n in Eqn. (1)

T(n-2) = 1 + T(n-3) ...Eqn. (3)

Step3: Substitute Eqn. (2) in Eqn. (1)

T(n)= 1 + 1+ T(n-2) = 2 + T(n-2) ...Eqn. (4)

Step4: Substitute eqn. (3) in Eqn. (4)

T(n) = 2 + 1 + T(n-3) = 3 + T(n-3) = ... = k + T(n-k) ...Eqn. (5)

Now, according to Eqn. (1), i.e. T(n) = 1 + T(n-1), the algorithm will run until n>1. Basically, n will start from a very large number, and it will decrease gradually. So, when T(n) = 1, the algorithm eventually stops, and such a terminating condition is called anchor condition, base condition or stopping condition.

Thus, for k = n-1, the T(n) will become.

Step5: Substitute k = n-1 in eqn. (5)

T(n) = (n-1) + T(n-(n-1)) = (n-1) + T(1) = n-1+1

Hence, T(n) = n or **O(n).**

Algorithm Design Techniques

The following is a list of several popular design approaches:

1. Divide and Conquer Approach: It is a top-down approach. The algorithms which follow the divide & conquer techniques involve three steps:

- Divide the original problem into a set of subproblems.
- Solve every subproblem individually, recursively.
- Combine the solution of the subproblems (top level) into a solution of the whole original problem.

2. Greedy Technique: Greedy method is used to solve the optimization problem. An optimization problem is one in which we are given a set of input values, which are required either to be maximized or minimized (known as objective), i.e. some constraints or conditions.

- Greedy Algorithm always makes the choice (greedy criteria) looks best at the moment, to optimize a given objective.
- The greedy algorithm doesn't always guarantee the optimal solution however it generally produces a solution that is very close in value to the optimal.

3. Dynamic Programming: Dynamic Programming is a bottom-up approach we solve all possible small problems and then combine them to obtain solutions for bigger problems.

This is particularly helpful when the number of copying subproblems is exponentially large. Dynamic Programming is frequently related to **Optimization Problems**.

4. Branch and Bound: In Branch & Bound algorithm a given subproblem, which cannot be bounded, has to be divided into at least two new restricted subproblems. Branch and Bound algorithm are methods for global optimization in non-convex problems. Branch and Bound algorithms can be slow, however in the worst case they require effort that grows exponentially with problem size, but in some cases we are lucky, and the method coverage with much less effort.

5. Randomized Algorithms: A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased random bits, and it is then allowed to use these random bits to influence its computation.

6. Backtracking Algorithm: Backtracking Algorithm tries each possibility until they find the right one. It is a depth-first search of the set of possible solution. During the search, if an alternative doesn't work, then backtrack to the choice point, the place which presented different alternatives, and tries the next alternative.

7. Randomized Algorithm: A randomized algorithm uses a random number at least once during the computation make a decision.

Example 1: In Quick Sort, using a random number to choose a pivot.

Example 2: Trying to factor a large number by choosing a random number as possible divisors.

Loop invariants

This is a justification technique. We use loop invariant that helps us to understand why an algorithm is correct. To prove statement S about a loop is correct, define S concerning series of smaller statement S_0 S_1....S_k where,

- The initial claim so is true before the loop begins.
- If S_{i-1} is true before iteration i begin, then one can show that S_i will be true after iteration i is over.
- The final statement S_k implies the statement S that we wish to justify as being true.

Asymptotic Analysis of algorithms (Growth of function)

Resources for an algorithm are usually expressed as a function regarding input. Often this function is messy and complicated to work. To study Function growth efficiently, we reduce the function down to the important part.

Let f (n) = an^2+bn+c

In this function, the n^2 term dominates the function that is when n gets sufficiently large.

Dominate terms are what we are interested in reducing a function, in this; we ignore all constants and coefficient and look at the highest order term concerning n.

Asymptotic notation:

The word **Asymptotic** means approaching a value or curve arbitrarily closely (i.e., as some sort of limit is taken).

Asymptotic analysis

It is a technique of representing limiting behavior. The methodology has the applications across science. It can be used to analyze the performance of an algorithm for some large data set.

1. In computer science in the analysis of algorithms, considering the performance of algorithms when applied to very large input datasets

The simplest example is a function $f(n) = n^2+3n$, the term 3n becomes insignificant compared to n^2 when n is very large. The function "$f(n)$ is said to be **asymptotically equivalent** to n^2 as $n \to \infty$", and here is written symbolically as $f(n) \sim n^2$.

Asymptotic notations are used to write fastest and slowest possible running time for an algorithm. These are also referred to as 'best case' and 'worst case' scenarios respectively.

"In asymptotic notations, we derive the complexity concerning the size of the input. (Example in terms of n)"

"These notations are important because without expanding the cost of running the algorithm, we can estimate the complexity of the algorithms."

Why is Asymptotic Notation Important?

1. They give simple characteristics of an algorithm's efficiency.

2. They allow the comparisons of the performances of various algorithms.

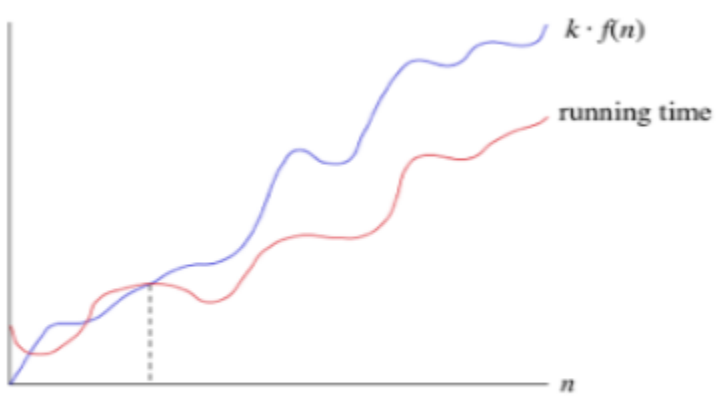
Asymptotic Notations:

Asymptotic Notation is a way of comparing function that ignores constant factors and small input sizes. Three notations are used to calculate the running time complexity of an algorithm:

1. Big-oh notation: Big-oh is the formal method of expressing the upper bound of an algorithm's running time. It is the measure of the longest amount of time. The function **f (n) = O (g (n))** [read as "f of n is big-oh of g of n"] if and only if exist positive constant c and such that

1. f (n) ⩽ k.g (n)f(n)⩽k.g(n) **for** n>n0n>n0 in all **case**

Hence, function g (n) is an upper bound for function f (n), as g (n) grows faster than f (n)



ASYMPTOTIC UPPER BOUND

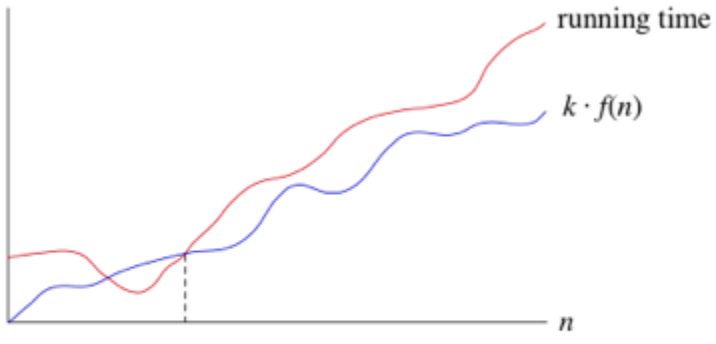
For Example:

1. 1. $3n+2=O(n)$ as $3n+2 \leq 4n$ **for** all $n \geq 2$
2. 2. $3n+3=O(n)$ as $3n+3 \leq 4n$ **for** all $n \geq 3$

Hence, the complexity of **f(n)** can be represented as O (g (n))

2. Omega () Notation: The function f (n) = Ω (g (n)) [read as "f of n is omega of g of n"] if and only if there exists positive constant c and n_0 such that

$F(n) \geq k^* g(n)$ for all n, $n \geq n_0$



ASYMPTOTIC LOWER BOUND

For Example:

$f(n) = 8n^2+2n-3 \geq 8n^2-3$

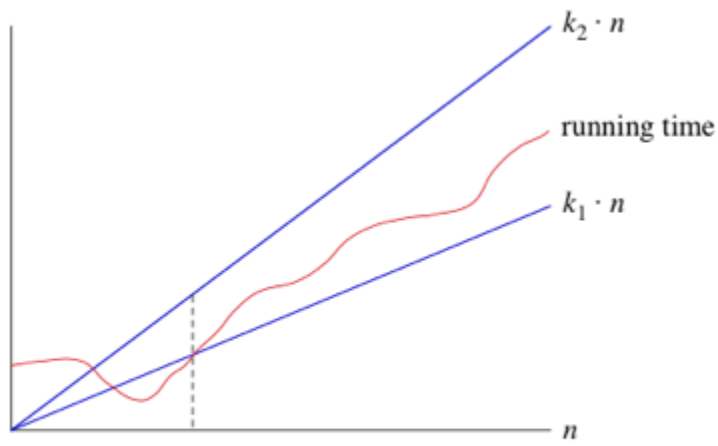
$= 7n^2+(n^2-3) \geq 7n^2$ (g(n))

Thus, $k_1=7$

Hence, the complexity of **f (n)** can be represented as Ω (g (n))

3. Theta (θ): The function f (n) = θ (g (n)) [read as "f is the theta of g of n"] if and only if there exists positive constant k_1, k_2 and k_0 such that

$k_1 * g(n) \leq f(n) \leq k_2\ g(n)$ for all n, $n \geq n_0$



ASYMPTOTIC TIGHT BOUND

For Example:

$3n+2 = \theta(n)$ as $3n+2 \geq 3n$ and $3n+2 \leq 4n$, for n

$k_1=3, k_2=4$, and $n_0=2$

Hence, the complexity of f (n) can be represented as θ (g(n)).

The Theta Notation is more precise than both the big-oh and Omega notation. The function f (n) = θ (g (n)) if g(n) is both an upper and lower bound.

Analyzing Algorithm Control Structure

To analyze a programming code or algorithm, we must notice that each instruction affects the overall performance of the algorithm and therefore, each instruction must be analyzed separately to analyze overall performance. However, there are some algorithm control structures which are present in each programming code and have a specific asymptotic analysis.

Some Algorithm Control Structures are:

1. Sequencing
2. If-then-else
3. for loop
4. While loop

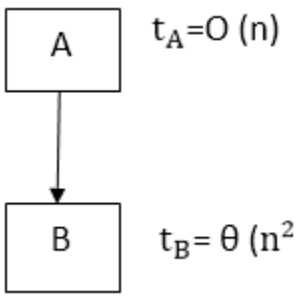
---

1. Sequencing:

Suppose our algorithm consists of two parts A and B. A takes time t_A and B takes time t_B for computation. The total computation "$t_A + t_B$" is according to the sequence rule. According to maximum rule, this computation time is (max (t_A,t_B)).

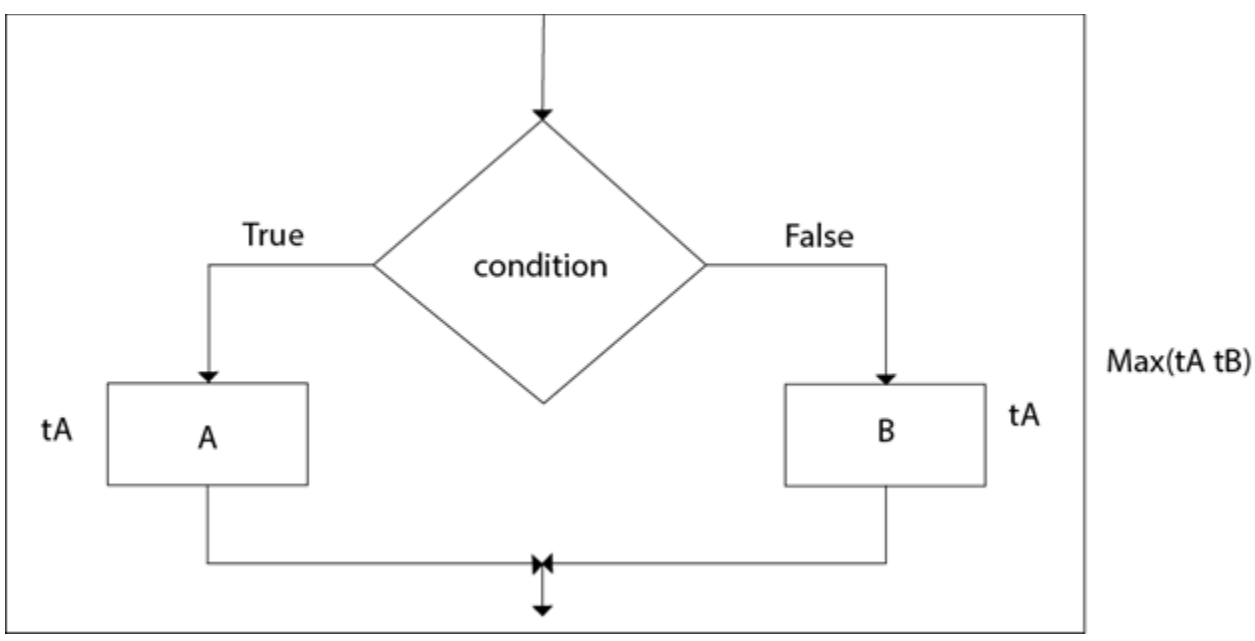
Example:

Suppose t_A =O (n) and t_B = θ (n^2).
Then, the total computation time can be calculated as



Computation Time = t_A + t_B
= (max (t_A,t_B)
= (max (O (n), θ (n^2)) = θ (n^2)

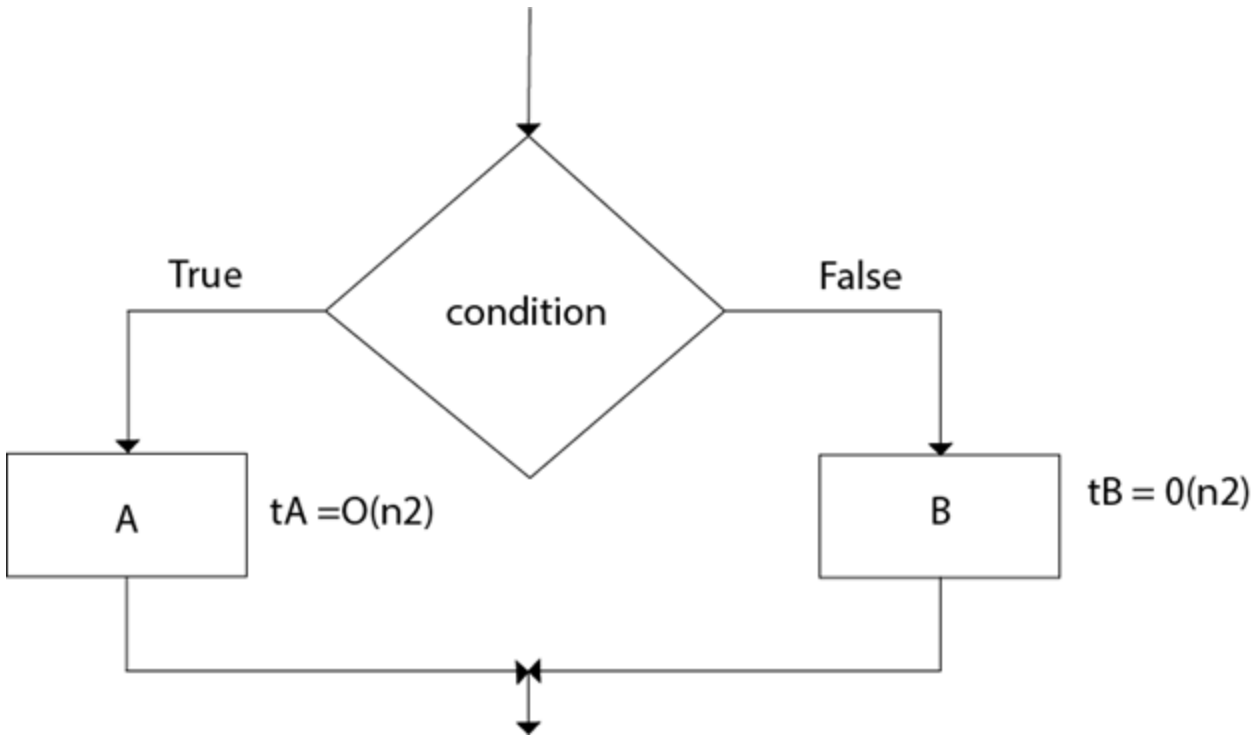
2. If-then-else:



The total time computation is according to the condition rule-"if-then-else." According to the maximum rule, this computation time is max (t_A,t_B).

Example:

Suppose t_A = O (n^2) and t_B = θ (n^2)
Calculate the total computation time for the following:



Total Computation = (max (t_A,t_B))
= max (O (n^2), θ (n^2) = θ (n^2)

3. For loop:

The general format of for loop is:

1. For (initialization; condition; updation)
2.
3. Statement(s);

Complexity of for loop:

The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times. As a result, the statements in the **inner** loop execute a total of N * M times. Thus, the total complexity for the two loops is O (N2)

Consider the following loop:

1. **for** i ← 1 to n
2. {
3. P (i)
4. }

If the computation time t_i for (P_I) various as a function of "i", then the total computation time for the loop is given not by a multiplication but by a sum i.e.

1. For i ← 1 to n
2. {
3. P (i)
4. }

Takes $\sum_{i=1}^{n} t_i$ time, i.e.$\sum_{j=1}^{n} \theta(1) = \theta\sum_{i=1}^{n} \theta(n)$

If the algorithms consist of nested "for" loops, then the total computation time is

```
For i ← 1 to n
 {
                                  ∑ⁿᵢ₌₁ ∑ⁿⱼ₌₁ tᵢⱼ
     For j ←  1 to n
   {
     P (ij)
   }
 }
```

Example:

Consider the following "for" loop, Calculate the total computation time for the following:

1. For i ← 2 to n-1

```
2.    {
3.        For j ← 3 to i
4.        {
5.                Sum ← Sum+A [i] [j]
6.            }
7.          }
```

Solution:

The total Computation time is:

$$\sum_{i=2}^{n-1}\sum_{j=3}^{i} t_{ij} = \sum_{i=2}^{n-1}\sum_{j=3}^{i} \theta\,(1)$$

$$\sum_{i=2}^{n-1} \theta(i)$$

$$= \theta\left(\sum_{i=2}^{n-1} i\right) = \theta\left(\frac{m^2}{2}\right) + \theta(m)$$

$$= \theta(m^2)$$

4. While loop:

The Simple technique for analyzing the loop is to determine the function of variable involved whose value decreases each time around. Secondly, for terminating the loop, it is necessary that value must be a positive integer. By keeping track of how many times the value of function decreases, one can obtain the number of repetition of the loop. The other approach for analyzing "while" loop is to treat them as recursive algorithms.

Algorithm:

1. 1. [Initialize] Set k: =1, LOC: =1 and MAX: = DATA [1]
2. 2. Repeat steps 3 and 4 **while** K≤N
3. 3. **if** MAX<DATA [k],then:
4. Set LOC: = K and MAX: = DATA [k]
5. 4. Set k: = k+1
6. [End of step 2 loop]
7. 5. Write: LOC, MAX
8. 6. EXIT

Example:

The running time of algorithm array Max of computing the maximum element in an array of n integer is O (n).

Solution:

1. array Max (A, n)
2. 1. Current max ← A [0]
3. 2. For i ← 1 to n-1
4. 3. **do if** current max < A [i]
5. 4. then current max ← A [i]
6. 5. **return** current max.

The number of primitive operation t (n) executed by this algorithm is at least.

1. 2 + 1 + n +4 (n-1) + 1=5n
2. 2 + 1 + n + 6 (n-1) + 1=7n-2

The best case T(n) =5n occurs when A [0] is the maximum element. The worst case T(n) = 7n-2 occurs when element are sorted in increasing order.

We may, therefore, apply the big-Oh definition with c=7 and n_0=1 and conclude the running time of this is O (n).

Recurrence Relation

A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs. To solve a Recurrence Relation means to obtain a function defined on the natural numbers that satisfy the recurrence.

For Example, the Worst Case Running Time T(n) of the MERGE SORT Procedures is described by the recurrence.

$$T(n) = \theta(1) \text{ if } n=1$$
$$2T\left(\frac{n}{2}\right) + \theta(n) \text{ if } n>1$$

There are four methods for solving Recurrence:

1. Substitution Method
2. Iteration Method
3. Recursion Tree Method
4. Master Method

---

1. Substitution Method:

The Substitution Method Consists of two main steps:

1. Guess the Solution.
2. Use the mathematical induction to find the boundary condition and shows that the guess is correct.

For Example1 Solve the equation by Substitution Method.

$$T(n) = T\left(\frac{n}{2}\right) + n$$

We have to show that it is asymptotically bound by O (log n).

Solution:

For $T(n) = O(\log n)$

We have to show that for some constant c

1. $T(n) \leq c \log n$.

Put this in given Recurrence Equation.

$$T(n) \leq c \log\left(\frac{n}{2}\right) + 1$$
$$\leq c \log\left(\frac{n}{2}\right) + 1 = c \log n - c \log_2 2 + 1$$
$$\leq c \log n \text{ for } c \geq 1$$

Thus **T (n) =O logn**.

Example2 Consider the Recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + n \quad n>1$$

Find an Asymptotic bound on T.

Solution:

We guess the solution is O (n (logn)).Thus for constant 'c'.
$T(n) \leq c\, n \log n$
Put this in given Recurrence Equation.
Now,

$$T(n) \leq 2c\left(\frac{n}{2}\right)\log\left(\frac{n}{2}\right) + n$$
$$\leq cn\log n - cn\log 2 + n$$
$$= cn\log n - n(c\log 2 - 1)$$

```
      ≤cn logn for (c≥1)
Thus T (n) = O (n logn).
```

2. Iteration Methods

It means to expand the recurrence and express it as a summation of terms of n and initial condition.

Example1: Consider the Recurrence

1. T (n) = 1 **if** n=1
2. = 2T (n-1) **if** n>1

Solution:

```
T (n) = 2T (n-1)
      = 2[2T (n-2)] = 2²T (n-2)
      = 4[2T (n-3)] = 2³T (n-3)
      = 8[2T (n-4)] = 2⁴T (n-4)    (Eq.1)

Repeat the procedure for i times

T (n) = 2ⁱ T (n-i)
Put n-i=1 or i= n-1 in    (Eq.1)
T (n) = 2ⁿ⁻¹ T (1)
      = 2ⁿ⁻¹ .1    {T (1) =1 .....given}
      = 2ⁿ⁻¹
```

Example2: Consider the Recurrence

1. T (n) = T (n-1) +1 and T (1) = θ (1).

Solution:

```
 T (n) = T (n-1) +1
       = (T (n-2) +1) +1 = (T (n-3) +1) +1+1
       = T (n-4) +4 = T (n-5) +1+4
       = T (n-5) +5= T (n-k) + k
Where k = n-1
   T (n-k) = T (1) = θ (1)
   T (n) = θ (1) + (n-1) = 1+n-1=n= θ (n).
```

Recursion Tree Method

1. Recursion Tree Method is a pictorial representation of an iteration method which is in the form of a tree where at each level nodes are expanded.

2. In general, we consider the second term in recurrence as root.

3. It is useful when the divide & Conquer algorithm is used.

4. It is sometimes difficult to come up with a good guess. In Recursion tree, each root and child represents the cost of a single subproblem.

5. We sum the costs within each of the levels of the tree to obtain a set of pre-level costs and then sum all pre-level costs to determine the total cost of all levels of the recursion.

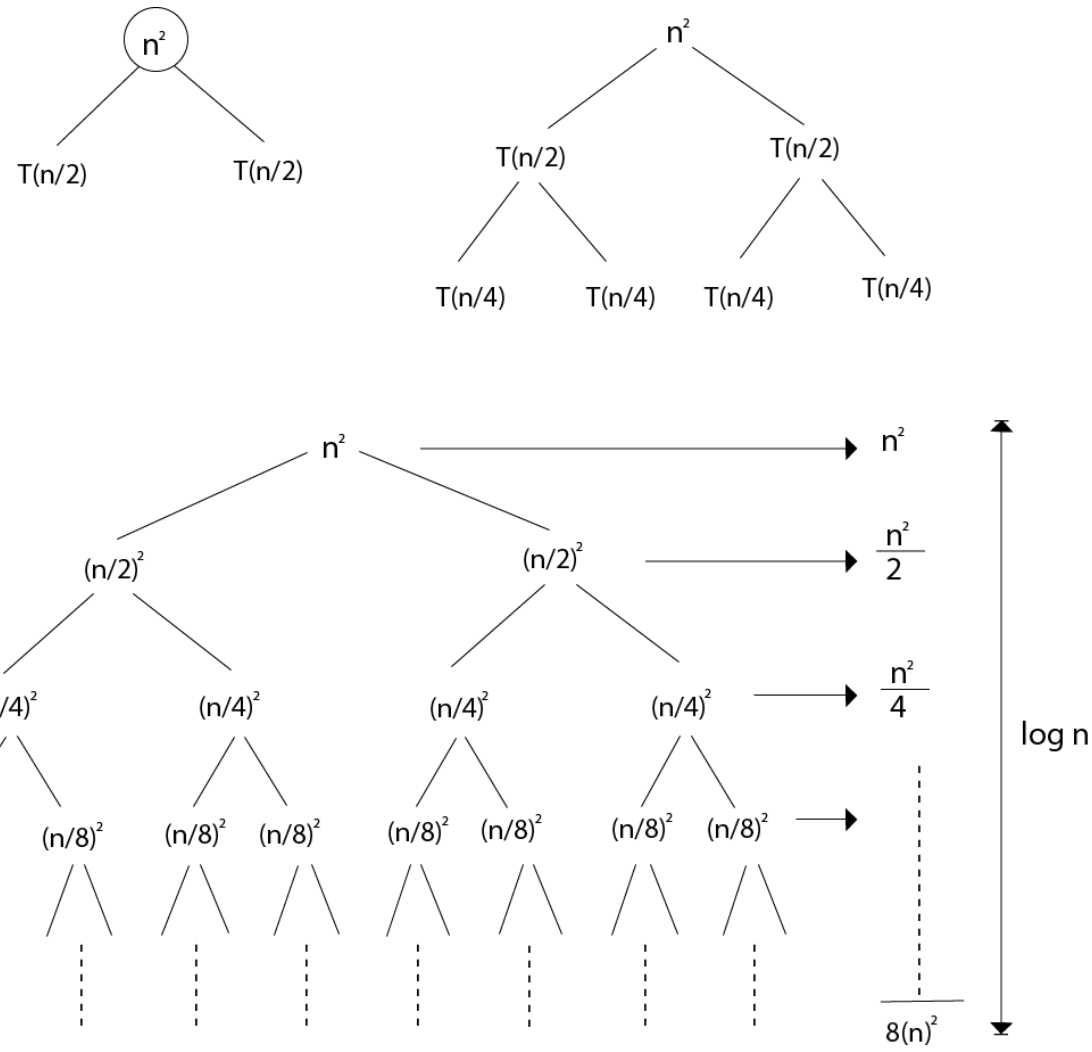
6. A Recursion Tree is best used to generate a good guess, which can be verified by the Substitution Method.

Example 1

Consider $T(n) = 2T\left(\frac{n}{2}\right) + n^2$

We have to obtain the asymptotic bound using recursion tree method.

Solution: The Recursion tree for the above recurrence is



$$T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \ldots\ldots \log n \text{ times.}$$

$$\leq n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2^i}\right)$$

$$\leq n^2 \left(\frac{1}{1-\frac{1}{2}}\right) \leq 2n^2$$

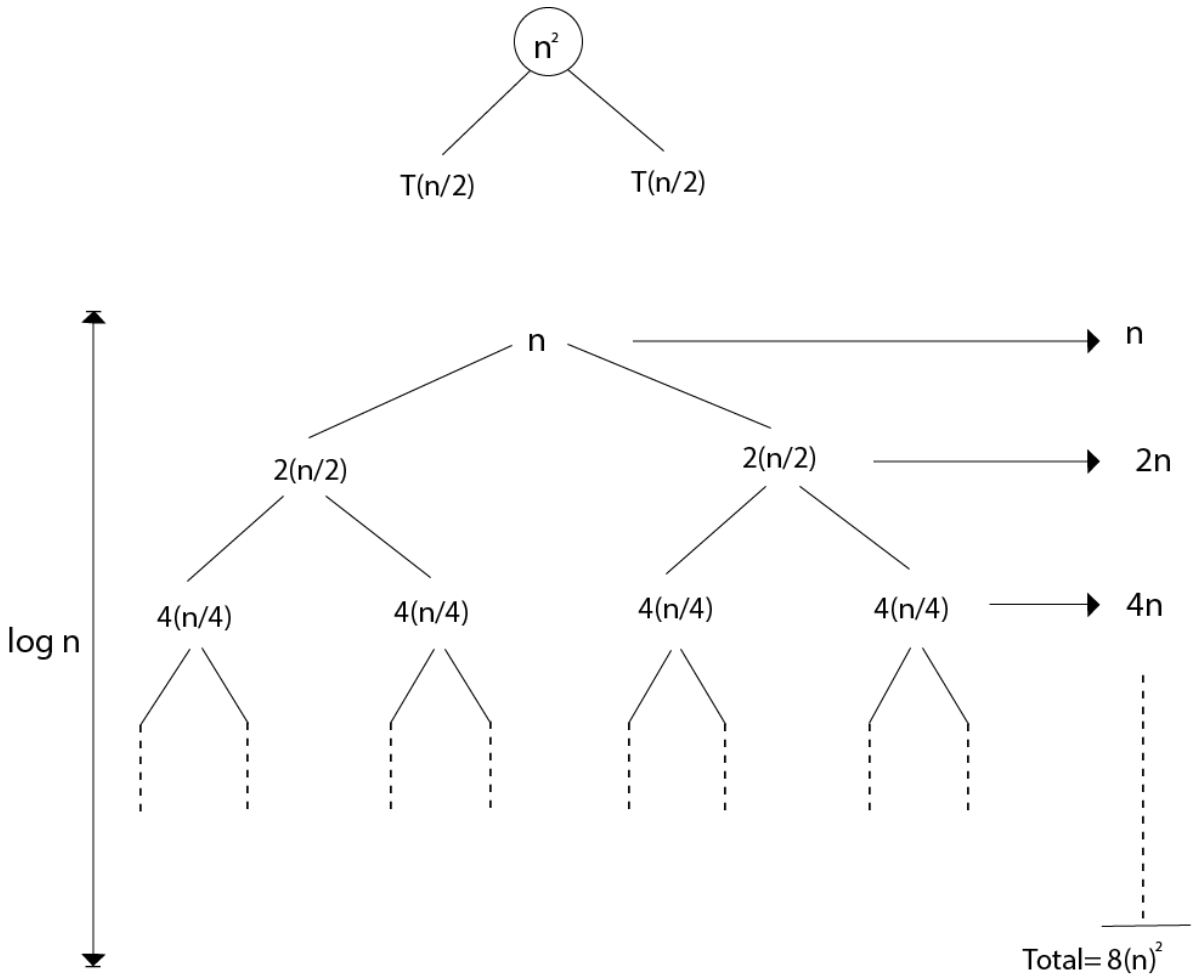
$$\mathbf{T(n) = \theta n^2}$$

Example 2: Consider the following recurrence

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

Obtain the asymptotic bound using recursion tree method.

Solution: The recursion trees for the above recurrence



We have $n + 2n + 4n + \ldots\ldots \log_2 n$ times

$$= n\,(1 + 2 + 4 + \ldots\ldots \log_2 n \text{ times})$$

$$= n\frac{(2^{\log_2 n} - 1)}{(2-1)} = \frac{n(n-1)}{1} = n^2 - n = \theta(n^2)$$

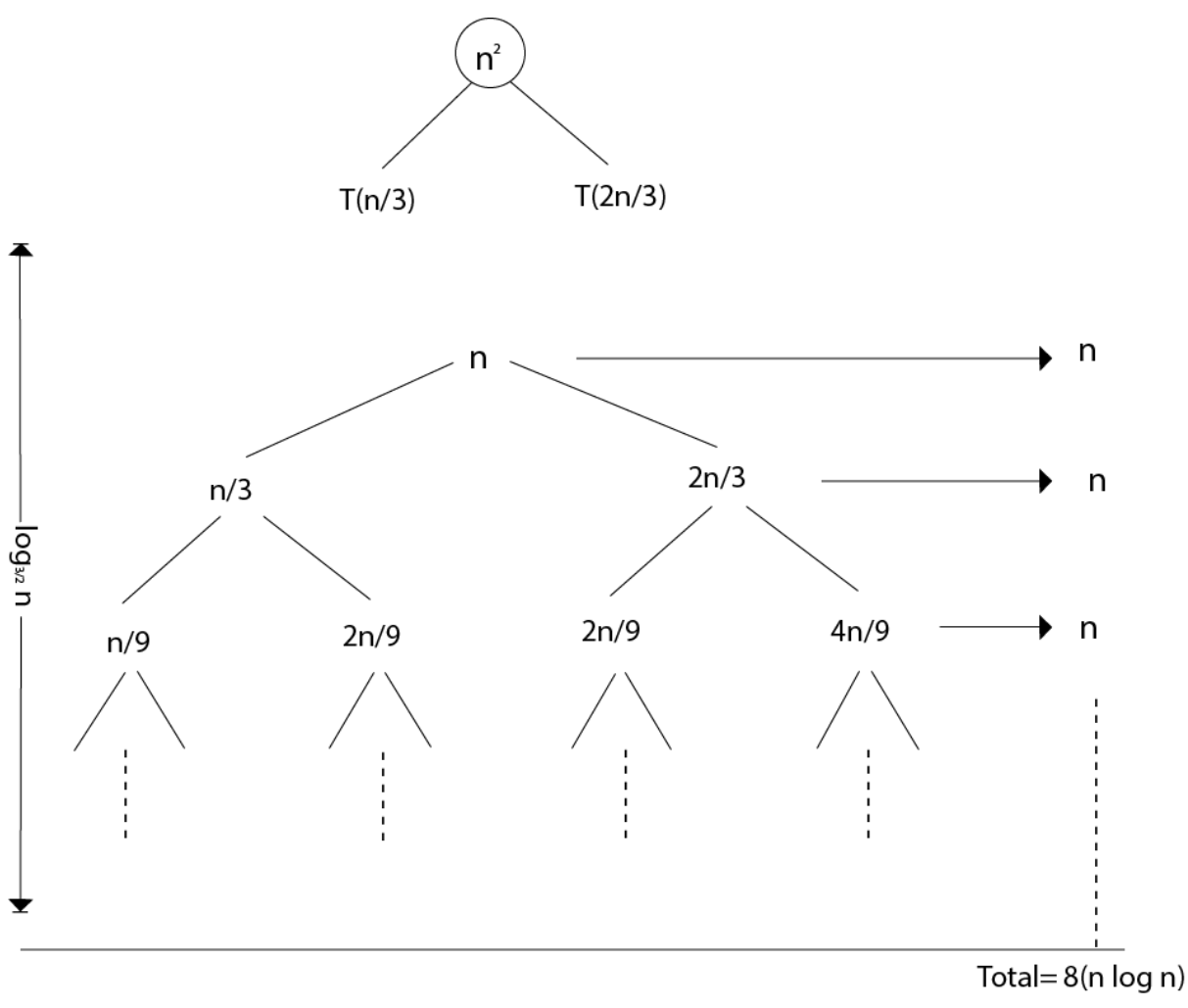
$$\mathbf{T\,(n) = \theta(n^2)}$$

Example 3: Consider the following recurrence

$$T\,(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$$

Obtain the asymptotic bound using recursion tree method.

Solution: The given Recurrence has the following recursion tree



When we add the values across the levels of the recursion trees, we get a value of n for every level. The longest path from the root to leaf is

$$n \longrightarrow \frac{2}{3}n \longrightarrow \left(\frac{2}{3}\right) n \longrightarrow \ldots 1$$

Since $\left(\frac{2}{3}\right) n=1$ when $i=\log \frac{3}{2}\ n$.

Thus the height of the tree is $\log \frac{3}{2}\ n$.

$$T(n) = n + n + n + \ldots\ldots + \log \frac{3}{2}\ n \text{ times.} = \boldsymbol{\theta(n \log n)}$$

Master Method

The Master Method is used for solving the following types of recurrence

T (n) = a $T\left(\frac{n}{b}\right)$ + f (n) with a≥1 and b≥1 be constant & f(n) be a function and $\frac{n}{b}$ can be interpreted as

Let T (n) is defined on non-negative integers by the recurrence.

$$T(n) = a\ T\left(\frac{n}{b}\right) + f(n)$$

In the function to the analysis of a recursive algorithm, the constants and function take on the following significance:

- n is the size of the problem.
- a is the number of subproblems in the recursion.
- n/b is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)
- f (n) is the sum of the work done outside the recursive calls, which includes the sum of dividing the problem and the sum of combining the solutions to the subproblems.
- It is not possible always bound the function according to the requirement, so we make three cases which will tell us what kind of bound we can apply on the function.

Master Theorem:

It is possible to complete an asymptotic tight bound in these three cases:

$$T(n)=\begin{cases} \Theta\left(n^{\log_b a}\right) & f(n)=O\left(n^{\log_b a-\varepsilon}\right) \\ \Theta\left(n^{\log_b a}\log n\right) & f(n)=\Theta\left(n^{\log_b a}\right) \\ \Theta(f(n)) & f(n)=\Omega\left(n^{\log_b a+\varepsilon}\right)\text{AND} \\ & af(n/b)<cf(n)\ \text{forlarge}n \end{cases} \begin{matrix} \\ \\ \varepsilon>0 \\ c<1 \end{matrix}$$

Case1: If f (n) = $O\left(n^{\log_b a-\varepsilon}\right)$ for some constant ε >0, then it follows that:

$$T(n) = \Theta\left(n^{\log_b a}\right)$$

Example:

$$T(n) = 8\ T\left(\frac{n}{2}\right)+1000n^2$$ apply master theorem on it.

Solution:

Compare T (n) = 8 T $\left(\frac{n}{2}\right)+1000n^2$ with

T (n) = a T $\left(\frac{n}{b}\right)+f(n)\ with\ a\geq 1\ and\ b>1$

a = 8, b=2, f (n) = 1000 n^2, $\log_b a = \log_2 8 = 3$

Put all the values in: f (n) = $O\left(n^{\log_b a-\varepsilon}\right)$

1000 n^2 = O ($n^{3-\varepsilon}$)

If we choose ε=1, we get: 1000 n^2 = O (n^{3-1}) = O (n^2)

Since this equation holds, the first case of the master theorem applies to the given recurrence relation, thus resulting in the conclusion:

T (n) = Θ $\left(n^{\log_b a}\right)$

Therefore: T (n) = Θ (n^3)

Case 2: If it is true, for some constant k ≥ 0 that:

F (n) = Θ $\left(n^{\log_b a}\log^k n\right)$ then it follows that: T (n) = Θ $\left(n^{\log_b a}\log^{k+1} n\right)$

Example:

T (n) = 2 $T\left(\frac{n}{2}\right)+10n$, solve the recurrence by using the master method.

As compare the given problem with T (n) = a T $\left(\frac{n}{b}\right)+f(n)\ with\ a\geq 1\ and\ b>1$ a = 2, b=2, k=0, f (n) = 10n, $\log_b a = \log_2 2 = 1$

Put all the values in f (n) =Θ $\left(n^{\log_b a}\log^k n\right)$, we will get

10n = Θ (n^1) = Θ (n) which is true.

Therefore: T (n) = Θ $\left(n^{\log_b a}\log^{k+1} n\right)$

= Θ (n log n)

Case 3: If it is true f(n) = Ω $\left(n^{\log_b a+\varepsilon}\right)$ for some constant ε >0 and it also true that: a f $\left(\frac{n}{b}\right)\leq cf(n)$ for some constant c<1 for large value of n ,then :

1. T (n) = Θ((f (n))

Example: Solve the recurrence relation:

T (n) = 2 $T\left(\frac{n}{2}\right)+n^2$

Solution:

Compare the given problem with T (n) = a T $\left(\frac{n}{b}\right)+f(n)\ with\ a\geq 1\ and\ b>1$

a= 2, b =2, f (n) = n^2, $\log_b a = \log_2 2 = 1$

Put all the values in f (n) = Ω $\left(n^{\log_b a+\varepsilon}\right)$ ..... (Eq. 1)

If we insert all the value in (Eq.1), we will get

n^2 = Ω($n^{1+\varepsilon}$) put ε =1, then the equality will hold.

n^2 = Ω(n^{1+1}) = Ω(n^2)

Now we will also check the second condition:

2 $\left(\frac{n}{2}\right)^2\leq cn^2 \Rightarrow \frac{1}{2}n^2\leq cn^2$

If we will choose c =1/2, it is true:

$\frac{1}{2}n^2\leq\frac{1}{2}n^2$ ∀ n ≥1

So it follows: T (n) = Θ ((f (n))

$T(n) = \Theta(n^2)$

DAA Bubble Sort

Bubble Sort, also known as Exchange Sort, is a simple sorting algorithm. It works by repeatedly stepping throughout the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. The pass through the list is duplicated until no swaps are desired, which means the list is sorted.

This is the easiest method among all sorting algorithms.

Algorithm

Step 1 ➤ Initialization

1. set 1 ← n, p ← 1

Step 2 ➤ loop,

1. Repeat through step 4 **while** (p ≤ n-1)
2. set E ← 0 ➤ Initializing exchange variable.

Step 3 ➤ comparison, loop.

1. Repeat **for** i ← 1, 1, ... l-1.
2. **if** (A [i] > A [i + 1]) then
3. set A [i] ↔ A [i + 1] ➤ Exchanging values.
4. Set E ← E + 1

Step 4 ➤ Finish, or reduce the size.

1. **if** (E = 0) then
2. exit
3. **else**
4. set l ← l - 1.

How Bubble Sort Works

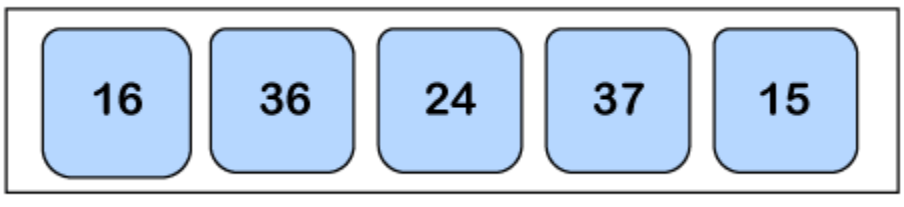
1. The bubble sort starts with the very first index and makes it a bubble element. Then it compares the bubble element, which is currently our first index element, with the next element. If the bubble element is greater and the second element is smaller, then both of them will swap.
   After swapping, the second element will become the bubble element. Now we will compare the second element with the third as we did in the earlier step and swap them if required. The same process is followed until the last element.
2. We will follow the same process for the rest of the iterations. After each of the iteration, we will notice that the largest element present in the unsorted array has reached the last index.

For each iteration, the bubble sort will compare up to the last unsorted element.

Once all the elements get sorted in the ascending order, the algorithm will get terminated.

Consider the following example of an unsorted array that we will sort with the help of the Bubble Sort algorithm.

Initially,

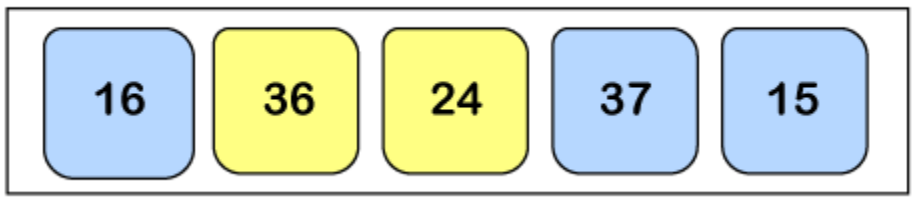


Pass 1:

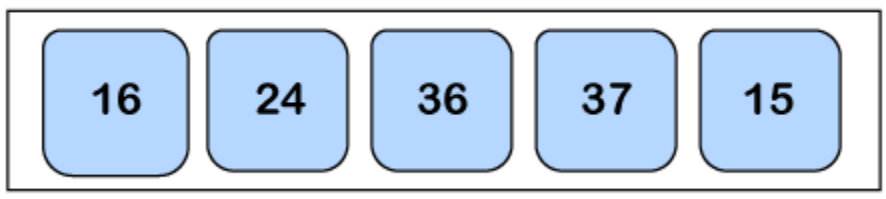
- **Compare a_0 and a_1**

As $a_0 < a_1$ so the array will remain as it is.

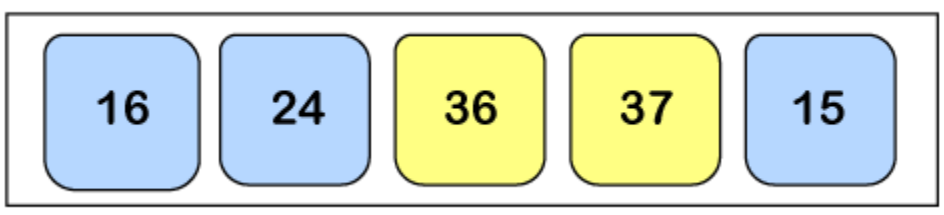
- **Compare a_1 and a_2**



Now $a_1 > a_2$, so we will swap both of them.

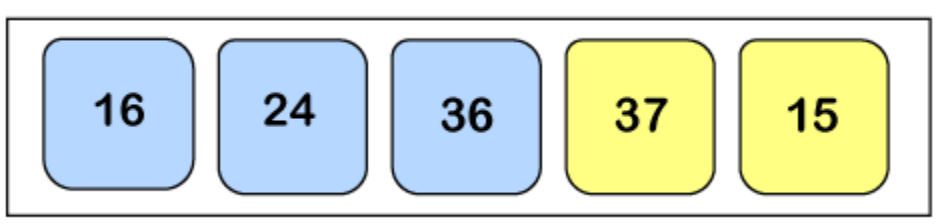


- **Compare a_2 and a_3**

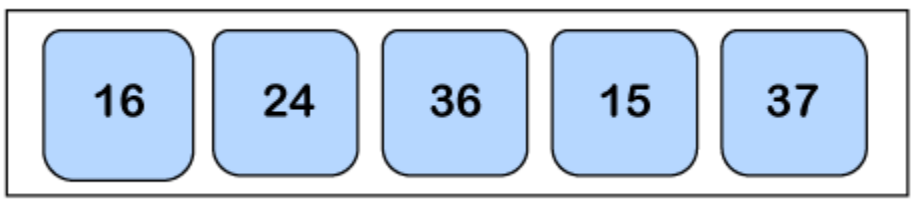


As $a_2 < a_3$ so the array will remain as it is.

- **Compare a_3 and a_4**



Here $a_3 > a_4$, so we will again swap both of them.

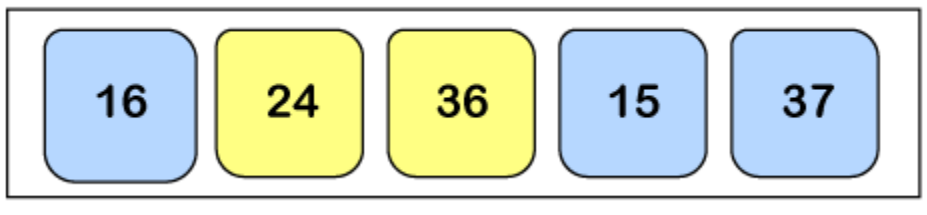


Pass 2:

- **Compare a_0 and a_1**

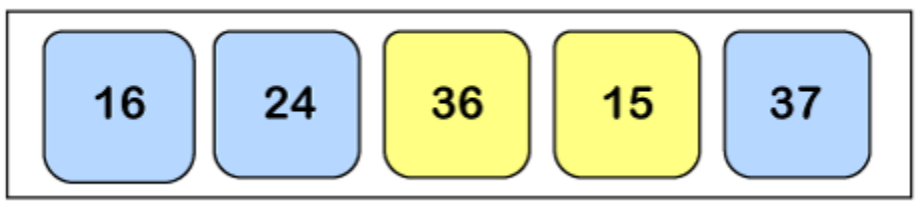
As $a_0 < a_1$ so the array will remain as it is.

- **Compare a_1 and a_2**

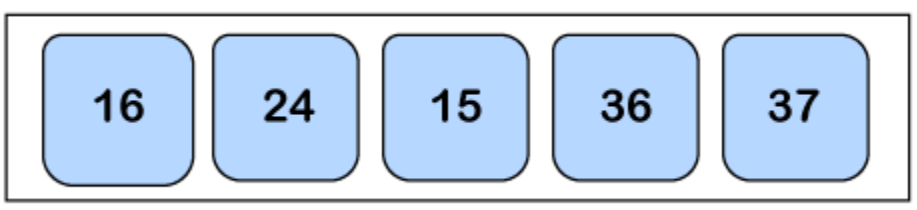


Here $a_1 < a_2$, so the array will remain as it is.

- **Compare a_2 and a_3**

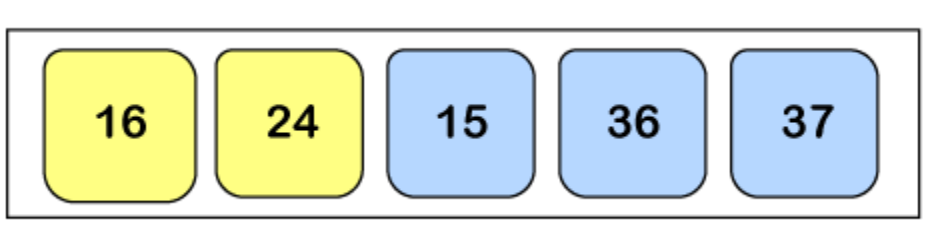


In this case, $a_2 > a_3$, so both of them will get swapped.



Pass 3:

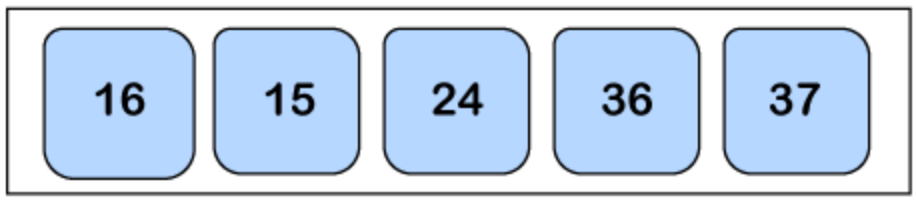
- **Compare a_0 and a_1**



As $a_0 < a_1$ so the array will remain as it is.

- **Compare a_1 and a_2**

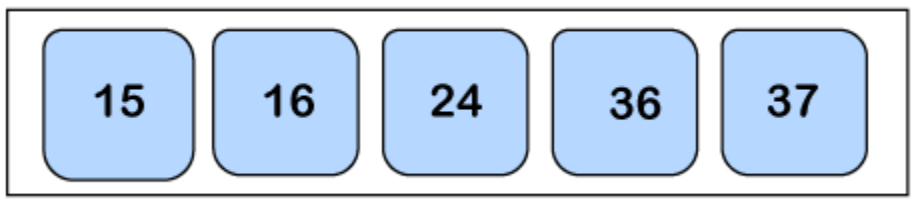
Now $a_1 > a_2$, so both of them will get swapped.



Pass 4:

- **Compare a_0 and a_1**

Here $a_0 > a_1$, so we will swap both of them.



Hence the array is sorted as no more swapping is required.

Complexity Analysis of Bubble Sort

Input: Given n input elements.

Output: Number of steps incurred to sort a list.

Logic: If we are given n elements, then in the first pass, it will do n-1 comparisons; in the second pass, it will do n-2; in the third pass, it will do n-3 and so on. Thus, the total number of comparisons can be found by;

Output;

$(n-1) + (n-2) + (n-3) + (n-4) + \ldots\ldots + 1$

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

i.e., $O(n^2)$

Therefore, the bubble sort algorithm encompasses a time complexity of **$O(n^2)$** and a space complexity of **O(1)** because it necessitates some extra memory space for temp variable for swapping.

Time Complexities:

- **Best Case Complexity**: The bubble sort algorithm has a best-case time complexity of **O(n)** for the already sorted array.
- **Average Case Complexity**: The average-case time complexity for the bubble sort algorithm is **$O(n^2)$**, which happens when 2 or more elements are in jumbled, i.e., neither in the ascending order nor in the descending order.
- **Worst Case Complexity**: The worst-case time complexity is also **$O(n^2)$**, which occurs when we sort the descending order of an array into the ascending order.

Advantages of Bubble Sort

1. Easily understandable.
2. Does not necessitates any extra memory.
3. The code can be written easily for this algorithm.
4. Minimal space requirement than that of other sorting algorithms.

Disadvantages of Bubble Sort

1. It does not work well when we have large unsorted lists, and it necessitates more resources that end up taking so much of time.
2. It is only meant for academic purposes, not for practical implementations.
3. It involves the n^2 order of steps to sort an algorithm.

Selection Sort

The selection sort enhances the bubble sort by making only a single swap for each pass through the rundown. In order to do this, a selection sort searches for the biggest value as it makes a pass and, after finishing the pass, places it in the best possible area. Similarly, as with a bubble sort, after the first pass, the biggest item is in the right place. After the second pass, the following biggest is set up. This procedure proceeds and requires n-1 goes to sort n item since the last item must be set up after the (n-1) th pass.

ALGORITHM: SELECTION SORT (A)

1. 1. k ← length [A]
2. 2. **for** j ←1 to n-1
3. 3. smallest ← j
4. 4. **for** I ← j + 1 to k
5. 5. **if** A [i] < A [smallest]
6. 6. then smallest ← i
7. 7. exchange (A [j], A [smallest])

How Selection Sort works

1. In the selection sort, first of all, we set the initial element as a **minimum**.
2. Now we will compare the minimum with the second element. If the second element turns out to be smaller than the minimum, we will swap them, followed by assigning to a minimum to the third element.
3. Else if the second element is greater than the minimum, which is our first element, then we will do nothing and move on to the third element and then compare it with the minimum. We will repeat this process until we reach the last element.
4. After the completion of each iteration, we will notice that our minimum has reached the start of the unsorted list.
5. For each iteration, we will start the indexing from the first element of the unsorted list. We will repeat the Steps from 1 to 4 until the list gets sorted or all the elements get correctly positioned. Consider the following example of an unsorted array that we will sort with the help of the Selection Sort algorithm.

   A [] = (7, 4, 3, 6, 5).
   A [] =

1st Iteration:

Set minimum = 7

- Compare a_0 and a_1

As, $a_0 > a_1$, set minimum = 4.

- Compare a_1 and a_2

As, $a_1 > a_2$, set minimum = 3.

- Compare a_2 and a_3

As, $a_2 < a_3$, set minimum= 3.

- Compare a_2 and a_4

As, $a_2 < a_4$, set minimum =3.

Since 3 is the smallest element, so we will swap a_0 and a_2.

2nd Iteration:

Set minimum = 4

- Compare a_1 and a_2

As, $a_1 < a_2$, set minimum = 4.

- Compare a_1 and a_3

As, A[1] < A[3], set minimum = 4.

- Compare a_1 and a_4

Again, $a_1 < a_4$, set minimum = 4.

Since the minimum is already placed in the correct position, so there will be no swapping.

3rd Iteration:

Set minimum = 7

- Compare a_2 and a_3

As, $a_2 > a_3$, set minimum = 6.

- Compare a_3 and a_4

As, $a_3 > a_4$, set minimum = 5.

Since 5 is the smallest element among the leftover unsorted elements, so we will swap 7 and 5.

4th Iteration:

Set minimum = 6

- Compare a_3 and a_4

As $a_3 < a_4$, set minimum = 6.

Since the minimum is already placed in the correct position, so there will be no swapping.

Complexity Analysis of Selection Sort

Input: Given **n** input elements.

Output: Number of steps incurred to sort a list.

Logic: If we are given n elements, then in the first pass, it will do **n-1** comparisons; in the second pass, it will do **n-2**; in the third pass, it will do **n-3** and so on. Thus, the total number of comparisons can be found by;

Output;

(n-1) + (n-2) + (n-3) + (n-4) + ... + 1

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

i.e., $O(n^2)$

Therefore, the selection sort algorithm encompasses a time complexity of **$O(n^2)$** and a space complexity of **O(1)** because it necessitates some extra memory space for temp variable for swapping.

Time Complexities:

- **Best Case Complexity:** The selection sort algorithm has a best-case time complexity of **$O(n^2)$** for the already sorted array.
- **Average Case Complexity:** The average-case time complexity for the selection sort algorithm is **$O(n^2)$**, in which the existing elements are in jumbled ordered, i.e., neither in the ascending order nor in the descending order.
- **Worst Case Complexity:** The worst-case time complexity is also **$O(n^2)$**, which occurs when we sort the descending order of an array into the ascending order.

In the selection sort algorithm, the time complexity is **$O(n^2)$** in all three cases. This is because, in each step, we are required to find **minimum** elements so that it can be placed in the correct position. Once we trace the complete array, we will get our minimum element.

Insertion Sort

Insertion sort is one of the simplest sorting algorithms for the reason that it sorts a single element at a particular instance. It is not the best sorting algorithm in terms of performance, but it's slightly more efficient than selection sort and bubble sort in practical scenarios. It is an intuitive sorting technique.

Let's consider the example of cards to have a better understanding of the logic behind the insertion sort.

Suppose we have a set of cards in our hand, such that we want to arrange these cards in ascending order. To sort these cards, we have a number of intuitive ways.

One such thing we can do is initially we can hold all of the cards in our left hand, and we can start taking cards one after other from the left hand, followed by building a sorted arrangement in the right hand.

Assuming the first card to be already sorted, we will select the next unsorted card. If the unsorted card is found to be greater than the selected card, we will simply place it on the right side, else to the left side. At any stage during this whole process, the left hand will be unsorted, and the right hand will be sorted.

In the same way, we will sort the rest of the unsorted cards by placing them in the correct position. At each iteration, the insertion algorithm places an unsorted element at its right place.

ALGORITHM: INSERTION SORT (A)

1. 1. for j = 2 to A.length
2. 2. key = A[j]
3. 3. // Insert A[j] into the sorted sequence A[1.. j - 1]
4. 4. i = j - 1
5. 5. while i > 0 and A[i] > key
6. 6. A[i + 1] = A[i]
7. 7. ii = i -1
8. 8. A[i + 1] = key

How Insertion Sort Works

1. We will start by assuming the very first element of the array is already sorted. Inside the **key**, we will store the second element.

Next, we will compare our first element with the **key**, such that if **the key** is found to be smaller than the first element, we will interchange their indexes or place the key at the first index. After doing this, we will notice that the first two elements are sorted.

2. Now, we will move on to the third element and compare it with the left-hand side elements. If it is the smallest element, then we will place the third element at the first index.

Else if it is greater than the first element and smaller than the second element, then we will interchange its position with the third element and place it after the first element. After doing this, we will have our first three elements in a sorted manner.

3. Similarly, we will sort the rest of the elements and place them in their correct position.

Consider the following example of an unsorted array that we will sort with the help of the Insertion Sort algorithm.

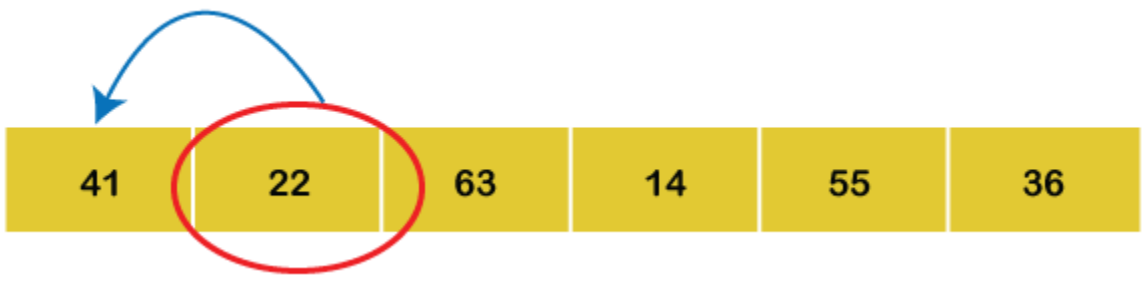
A = (41, 22, 63, 14, 55, 36)

Initially,

1st Iteration:

Set key = 22

Compare a1 with a0



Since a0 > a1, swap both of them.

2nd Iteration:

Set key = 63

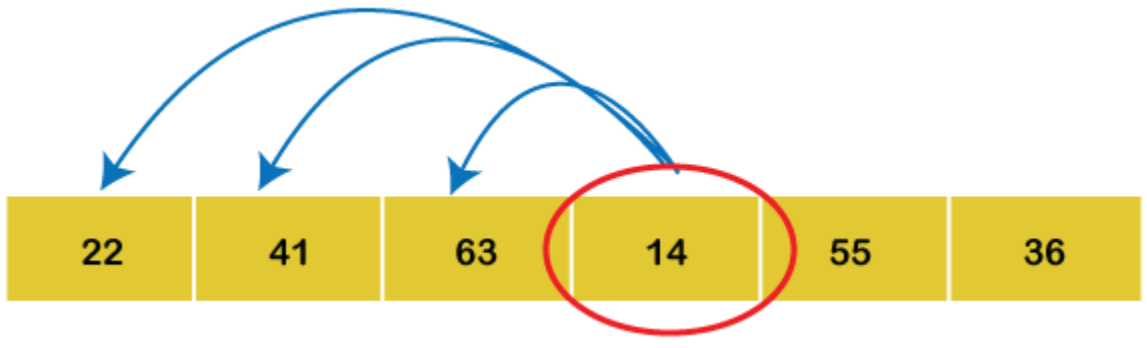
Compare a2 with a1 and a0

Since a2 > a1 > a0, keep the array as it is.

3rd Iteration:

Set key = 14

Compare a3 with a2, a1 and a0

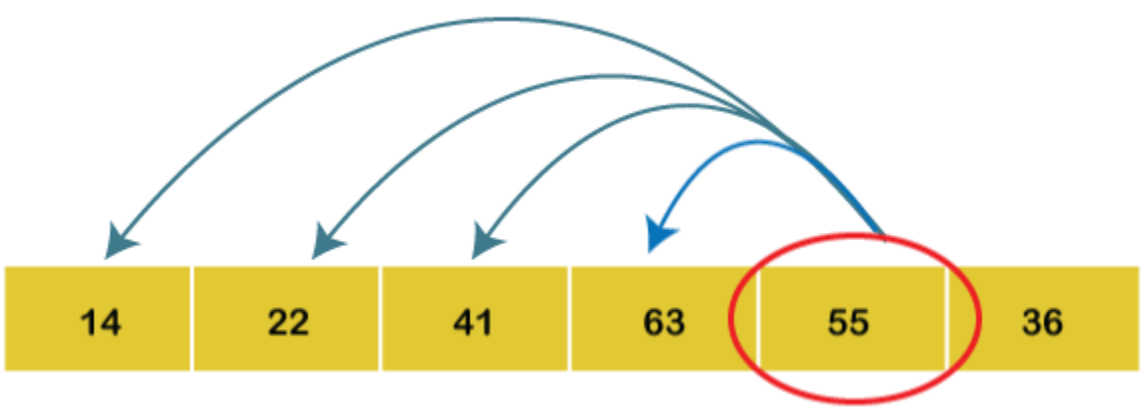


Since a3 is the smallest among all the elements on the left-hand side, place a3 at the beginning of the array.

4th Iteration:

Set key = 55

Compare a4 with a3, a2, a1 and a0.

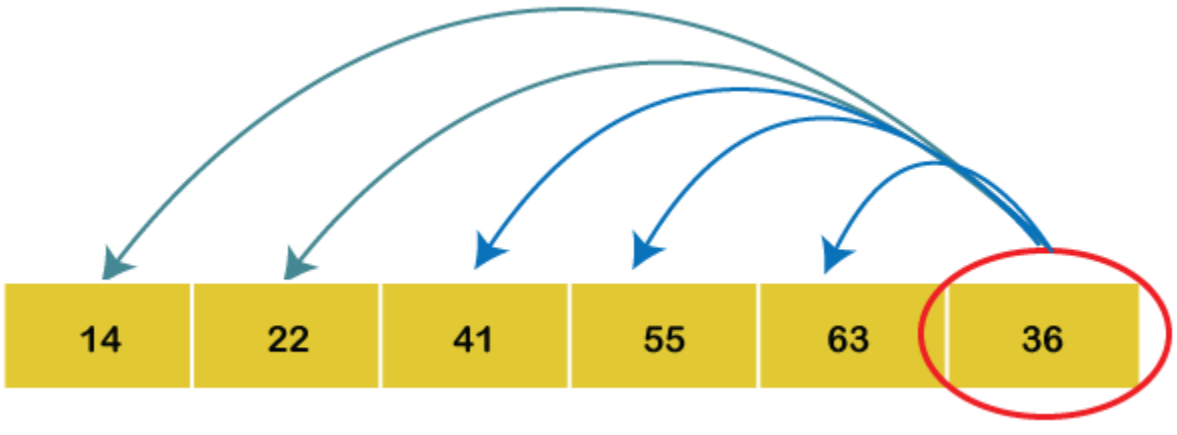


As a4 < a3, swap both of them.

5th Iteration:

Set key = 36

Compare a5 with a4, a3, a2, a1 and a0.



Since a5 < a2, so we will place the elements in their correct positions.

Hence the array is arranged in ascending order, so no more swapping is required.

Complexity Analysis of Insertion Sort

Input: Given n input elements.

Output: Number of steps incurred to sort a list.

Logic: If we are given **n** elements, then in the first pass, it will make **n-1** comparisons; in the second pass, it will do **n-2**; in the third pass, it will do **n-3** and so on. Thus, the total number of comparisons can be found by;

```
Output;
(n-1) + (n-2) + (n-3) + (n-4) + ..... + 1
```


```
Sum=
i.e., O(n²)
```

Therefore, the insertion sort algorithm encompasses a time complexity of **O(n^2)** and a space complexity of **O(1)** because it necessitates some extra memory space for a **key** variable to perform swaps.

Time Complexities:

- **Best Case Complexity:** The insertion sort algorithm has a best-case time complexity of **O(n)** for the already sorted array because here, only the outer loop is running n times, and the inner loop is kept still.
- **Average Case Complexity:** The average-case time complexity for the insertion sort algorithm is **O(n^2)**, which is incurred when the existing elements are in jumbled order, i.e., neither in the ascending order nor in the descending order.
- **Worst Case Complexity:** The worst-case time complexity is also **O(n^2)**, which occurs when we sort the ascending order of an array into the descending order.
  In this algorithm, every individual element is compared with the rest of the elements, due to which n-1 comparisons are made for every n^{th} element.

The insertion sort algorithm is highly recommended, especially when a few elements are left for sorting or in case the array encompasses few elements.

Space Complexity

The insertion sort encompasses a space complexity of **O(1)** due to the usage of an extra variable **key**.

Insertion Sort Applications

The insertion sort algorithm is used in the following cases:

- When the array contains only a few elements.

- When there exist few elements to sort.

Advantages of Insertion Sort

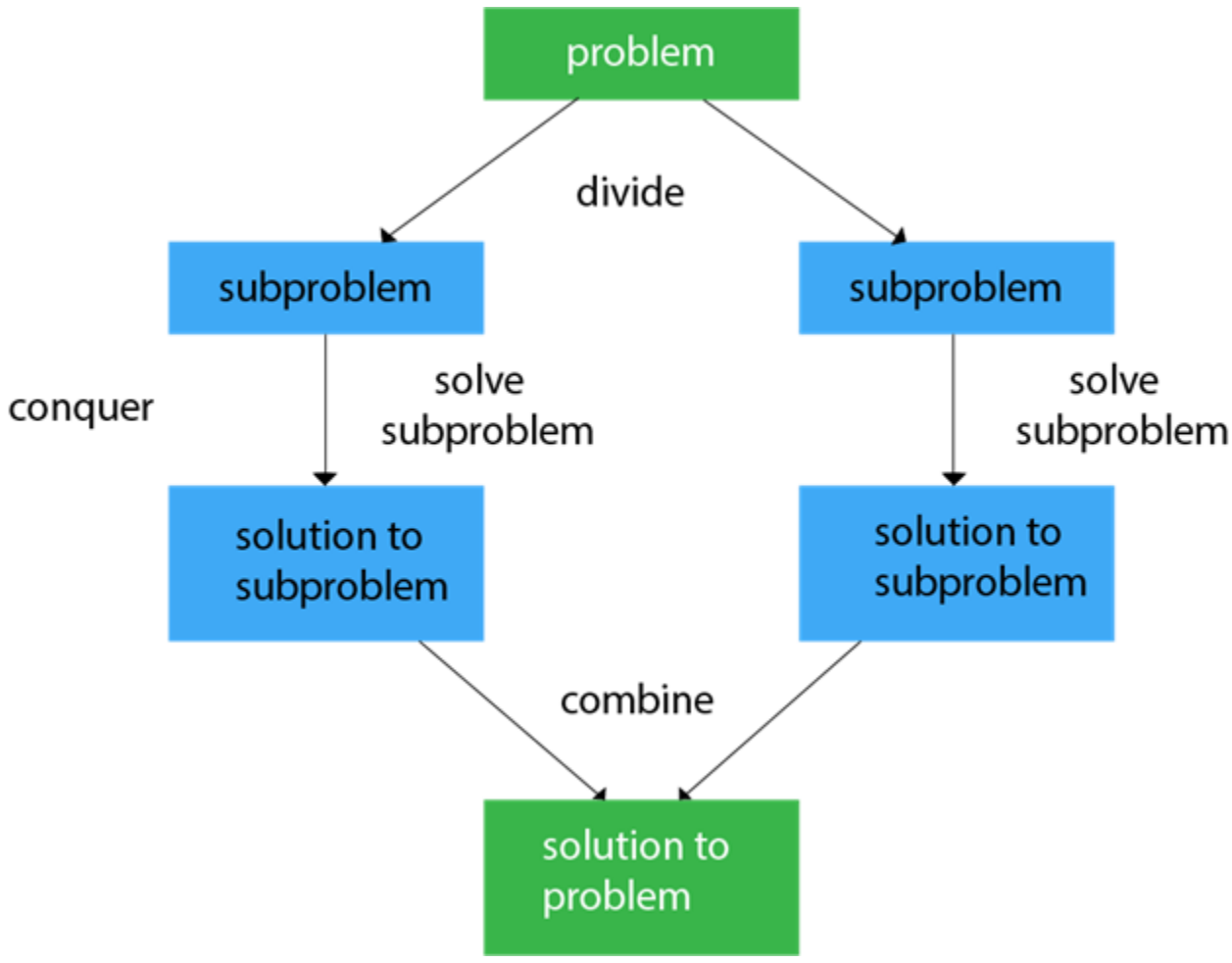
1. It is simple to implement.
2. It is efficient on small datasets.
3. It is stable (does not change the relative order of elements with equal keys)
4. It is in-place (only requires a constant amount O (1) of extra memory space).
5. It is an online algorithm, which can sort a list when it is received.

Divide and Conquer Introduction

Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy.

Divide and Conquer algorithm consists of a dispute using the following three steps.

1. **Divide** the original problem into a set of subproblems.
2. **Conquer:** Solve every subproblem individually, recursively.
3. **Combine:** Put together the solutions of the subproblems to get the solution to the whole problem.



Generally, we can follow the **divide-and-conquer** approach in a three-step process.

Examples: The specific computer algorithms are based on the Divide & Conquer approach:

1. Maximum and Minimum Problem
2. Binary Search
3. Sorting (merge sort, quick sort)
4. Tower of Hanoi.

Fundamental of Divide & Conquer Strategy:

There are two fundamental of Divide & Conquer Strategy:

1. Relational Formula
2. Stopping Condition

1. Relational Formula: It is the formula that we generate from the given technique. After generation of Formula we apply D&C Strategy, i.e. we break the problem recursively & solve the broken subproblems.

2. Stopping Condition: When we break the problem using Divide & Conquer Strategy, then we need to know that for how much time, we need to apply divide & Conquer. So the condition where the need to stop our recursion steps of D&C is called as Stopping Condition.

Applications of Divide and Conquer Approach:

Following algorithms are based on the concept of the Divide and Conquer Technique:

1. **Binary Search:** The binary search algorithm is a searching algorithm, which is also called a half-interval search or logarithmic search. It works by comparing the target value with the middle element existing in a sorted array. After making the comparison, if the value differs, then the half that cannot contain the target will eventually eliminate, followed by continuing the search on the other half. We will again consider the middle element and compare it with the target value. The process keeps on repeating until the target value is met. If we found the other half to be empty after ending the search, then it can be concluded that the target is not present in the array.
2. **Quicksort:** It is the most efficient sorting algorithm, which is also known as partition-exchange sort. It starts by selecting a pivot value from an array followed by dividing the rest of the array elements into two sub-arrays. The partition is made by comparing each of the elements with the pivot value. It compares whether the element holds a greater value or lesser value than the pivot and then sort the arrays recursively.
3. **Merge Sort:** It is a sorting algorithm that sorts an array by making comparisons. It starts by dividing an array into sub-array and then recursively sorts each of them. After the sorting is done, it merges them back.
4. **Closest Pair of Points:** It is a problem of computational geometry. This algorithm emphasizes finding out the closest pair of points in a metric space, given n points, such that the distance between the pair of points should be minimal.
5. **Strassen's Algorithm:** It is an algorithm for matrix multiplication, which is named after Volker Strassen. It has proven to be much faster than the traditional algorithm when works on large matrices.
6. **Cooley-Tukey Fast Fourier Transform (FFT) algorithm:** The Fast Fourier Transform algorithm is named after J. W. Cooley and John Turkey. It follows the Divide and Conquer Approach and imposes a complexity of O(nlogn).
7. **Karatsuba algorithm for fast multiplication:** It is one of the fastest multiplication algorithms of the traditional time, invented by Anatoly Karatsuba in late 1960 and got published in 1962. It multiplies two n-digit numbers in such a way by reducing it to at most single-digit.

Advantages of Divide and Conquer

- Divide and Conquer tend to successfully solve one of the biggest problems, such as the Tower of Hanoi, a mathematical puzzle. It is challenging to solve complicated problems for which you have no basic idea, but with the help of the divide and conquer approach, it has lessened the effort as it works on dividing the main problem into two halves and then solve them recursively. This algorithm is much faster than other algorithms.
- It efficiently uses cache memory without occupying much space because it solves simple subproblems within the cache memory instead of accessing the slower main memory.
- It is more proficient than that of its counterpart Brute Force technique.
- Since these algorithms inhibit parallelism, it does not involve any modification and is handled by systems incorporating parallel processing.

Disadvantages of Divide and Conquer

- Since most of its algorithms are designed by incorporating recursion, so it necessitates high memory management.
- An explicit stack may overuse the space.
- It may even crash the system if the recursion is performed rigorously greater than the stack present in the CPU.

Max - Min Problem

Problem: Analyze the algorithm to find the maximum and minimum element from an array.

Algorithm: Max ?Min Element (a [])

```
Max:  a [i]
Min:   a [i]
For i= 2 to n do
If a[i]> max then
max = a[i]
if a[i] < min then
min: a[i]
return (max, min)
```

Analysis:

Method 1: if we apply the general approach to the array of size n, the number of comparisons required are 2n-2.

Method-2: In another approach, we will divide the problem into sub-problems and find the max and min of each group, now max. Of each group will compare with the only max of another group and min with min.

Let n = is the size of items in an array

Let T (n) = time required to apply the algorithm on an array of size n. Here we divide the terms as T(n/2).

2 here tends to the comparison of the minimum with minimum and maximum with maximum as in above example.

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 2$$

$$T(n) = 2\,T\left(\frac{n}{2}\right) + 2 \rightarrow \text{Eq (i)}$$

T (2) = 1, time required to compare two elements/items. (Time is measured in units of the number of comparisons)

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + 2 \rightarrow \text{Eq (ii)}$$

Put eq (ii) in eq (i)

$$\begin{aligned} T(n) &= 2\left[2T\left(\frac{n}{2^2}\right) + 2\right] + 2 \\ &= 2^2 T\left(\frac{n}{2^2}\right) + 2^2 + 2 \end{aligned}$$

Similarly, apply the same procedure recursively on each subproblem or anatomy

{Use recursion means, we will use some stopping condition to stop the algorithm}

$$T(n) = 2^i T\left(\frac{n}{2^i}\right) + 2^i + 2^{i-1} + \ldots\ldots\ldots\ldots\ldots\ldots + 2 \quad \ldots\ldots \text{(Eq. 3)}$$

Recursion will stop, when $\frac{n}{2^i} = 2 \Rightarrow n = 2^{i+1}$ → (Eq. 4)

Put the equ.4 into equation3.

$$\begin{aligned} T(n) &= 2^i T(2) + 2^i + 2^{i-1} + \ldots\ldots\ldots\ldots\ldots\ldots + 2 \\ &= 2^i .1 + 2^i + 2^{i-1} + \ldots\ldots\ldots\ldots\ldots\ldots + 2 \\ &= 2^i + \frac{2\left(2^i - 1\right)}{2-1} \\ &= 2^{i+1} + 2^i - 2 \\ &= n + \frac{n}{2} - 2 \\ &= \frac{3n}{2} - 2 \end{aligned}$$

Number of comparisons requires applying the divide and conquering algorithm on n elements/items = $\frac{3n}{2}-2$

Number of comparisons requires applying general approach on n elements = (n-1) + (n-1) = 2n-2

From this example, we can analyze, that how to reduce the number of comparisons by using this technique.

Analysis: suppose we have the array of size 8 elements.

Method1: requires (2n-2), (2x8)-2=14 comparisons

Method2: requires $\frac{3\times 8}{2}-2=10 \ \ comparsions$

It is evident; we can reduce the number of comparisons (complexity) by using a proper technique.

Binary Search

1. In Binary Search technique, we search an element in a sorted array by recursively dividing the interval in half.

2. Firstly, we take the whole array as an interval.

3. If the Pivot Element (the item to be searched) is less than the item in the middle of the interval, We discard the second half of the list and recursively repeat the process for the first half of the list by calculating the new middle and last element.

4. If the Pivot Element (the item to be searched) is greater than the item in the middle of the interval, we discard the first half of the list and work recursively on the second half by calculating the new beginning and middle element.

5. Repeatedly, check until the value is found or interval is empty.

Analysis:

1. **Input:** an array A of size n, already sorted in the ascending or descending order.
2. **Output:** analyze to search an element item in the sorted array of size n.
3. **Logic:** Let T (n) = number of comparisons of an item with n elements in a sorted array.

- Set BEG = 1 and END = n
- Find mid = $\text{int}\left(\frac{beg+end}{2}\right)$
- Compare the search item with the mid item.

Case 1: item = A[mid], then LOC = mid, but it the best case and T (n) = 1

Case 2: item ≠A [mid], then we will split the array into two equal parts of size $\frac{n}{2}$.

And again find the midpoint of the half-sorted array and compare with search element.

Repeat the same process until a search element is found.

T (n) = $T\left(\frac{n}{2}\right)+1$ ..... (Equation 1)

{Time to compare the search element with mid element, then with half of the selected half part of array}

$T\left(\frac{n}{2}\right)$ = $T\left(\frac{n}{2^2}\right) + 1$, putting $\frac{n}{2}$ in place of n.

Then we get: T (n) = ($T\left(\frac{n}{2^2}\right) + 1$) +1.....By putting T $\frac{n}{2}$ in (1) equation

T (n) = $T\left(\frac{n}{2^2}\right) + 2$..... (Equation 2)

$T\left(\frac{n}{2^2}\right) = T\left(\frac{n}{2^3}\right) + 1$..... Putting $\frac{n}{2}$ in place of n in eq 1.

T (n) = $T\left(\frac{n}{2^3}\right) + 1 + 2$

T (n) =$T\left(\frac{n}{2^3}\right) + 3$..... (Equation 3)

$T\left(\frac{n}{2^3}\right) = T\left(\frac{n}{2^4}\right) + 1$..... Putting $\frac{n}{3}$ in place of n in eq1

Put $T\left(\frac{n}{2^3}\right)$ in eq (3)

T (n) = $T\left(\frac{n}{2^4}\right) + 4$

Repeat the same process ith times

T (n) = $T\left(\frac{n}{2^i}\right) + i$ .....

Stopping Condition: T (1) =1

At least there will be only one term left that's why that term will compare out, and only one comparison be done that's why

T (1) =1 (Item, Comparison)

Is the last term of the equation and it will be equal to 1

$\frac{n}{2^i} = 1$

$\frac{n}{2^i}$ Is the last term of the equation and it will be equal to 1

$n = 2^i$

Applying log both sides

$\log n = \log_2 i$

Log n = i log 2

$\frac{\log n}{\log 2} = 1$

$\log_2 n = i$

$T(n) = T\left(\frac{n}{2^i}\right) + i$

$\frac{n}{2^i}$ = 1 as in eq 5

= T (1) + i

= 1 + i..... T (1) =1 by stopping condition

= 1+ $\log_2 n$

= $\log_2 n$ ..... (1 is a constant that's why ignore it)

Therefore, binary search is of order o ($\log_2 n$)

Merge Sort

Merge sort is yet another sorting algorithm that falls under the category of Divide and Conquer technique. It is one of the best sorting techniques that successfully build a recursive algorithm.

Divide and Conquer Strategy

In this technique, we segment a problem into two halves and solve them individually. After finding the solution of each half, we merge them back to represent the solution of the main problem.

Suppose we have an array **A**, such that our main concern will be to sort the subsection, which starts at index **p** and ends at index **r**, represented by **A[p..r]**.

Divide

If assumed **q** to be the central point somewhere in between **p** and **r**, then we will fragment the subarray **A[p..r]** into two arrays **A[p..q]** and **A[q+1, r]**.

Conquer

After splitting the arrays into two halves, the next step is to conquer. In this step, we individually sort both of the subarrays **A[p..q]** and **A[q+1, r]**. In case if we did not reach the base situation, then we again follow the same procedure, i.e., we further segment these subarrays followed by sorting them separately.

Combine

As when the base step is acquired by the conquer step, we successfully get our sorted subarrays **A[p..q]** and **A[q+1, r]**, after which we merge them back to form a new sorted array **[p..r]**.

Merge Sort algorithm

The MergeSort function keeps on splitting an array into two halves until a condition is met where we try to perform MergeSort on a subarray of size 1, i.e., **p == r**.

And then, it combines the individually sorted subarrays into larger arrays until the whole array is merged.

1. ALGORITHM-MERGE SORT
2. 1. If p**<r**
3. 2. Then q → (p+ r)/2
4. 3. MERGE-SORT (A, p, q)
5. 4. MERGE-SORT (A, q+1,r)
6. 5. MERGE (A, p, q, r)

Here we called **MergeSort(A, 0, length(A)-1)** to sort the complete array.

As you can see in the image given below, the merge sort algorithm recursively divides the array into halves until the base condition is met, where we are left with only 1 element in the array. And then, the merge function picks up the sorted sub-arrays and merge them back to sort the entire array.

The following figure illustrates the dividing (splitting) procedure.

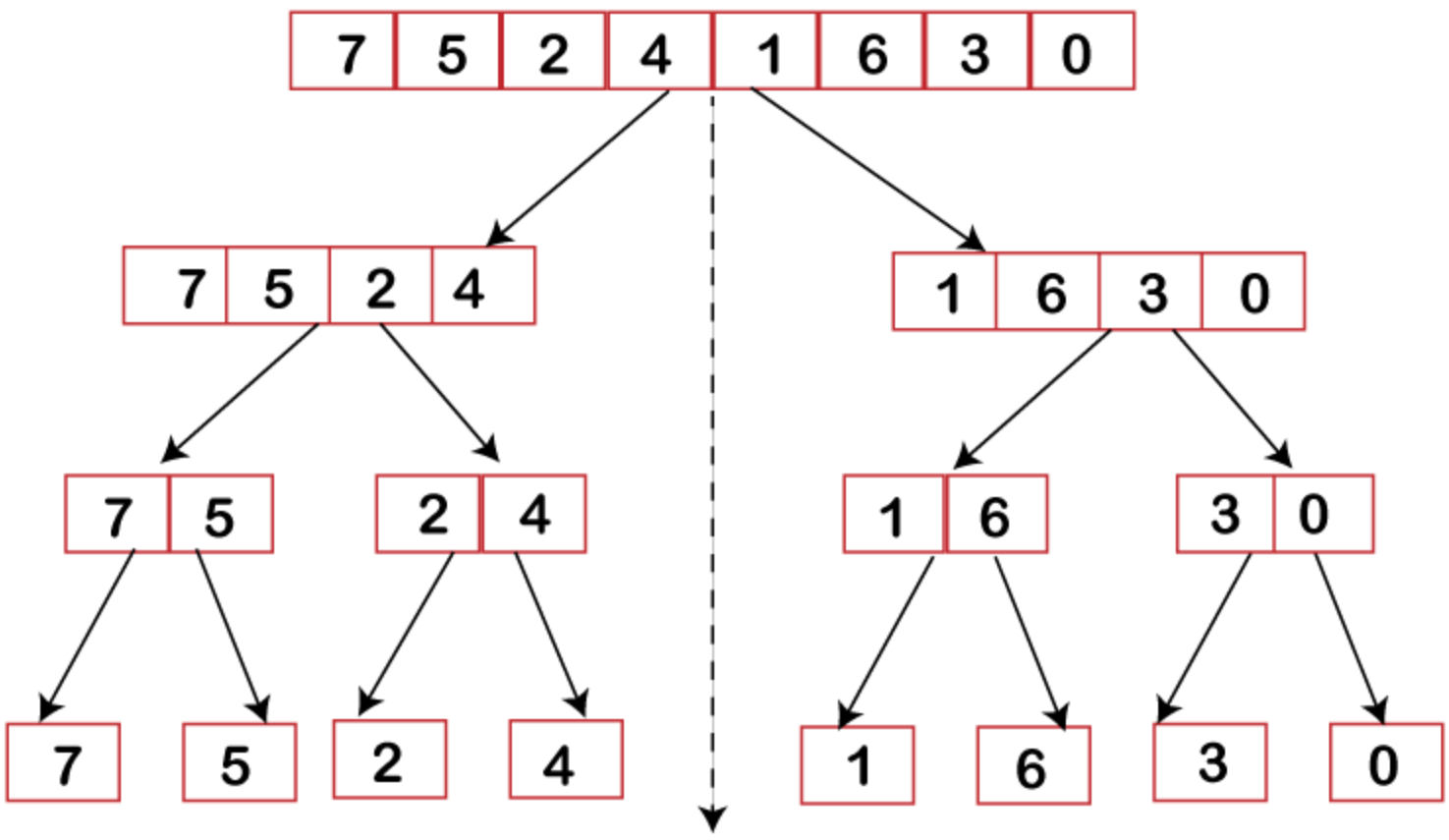


Figure 1: Merge Sort Divide Phase

1. FUNCTIONS: MERGE (A, p, q, r)
2.
3. 1. n 1 = q-p+1
4. 2. n 2= r-q
5. 3. create arrays [1.....n 1 + 1] and R [1.....n 2 +1]
6. 4. for i ← 1 to n 1
7. 5. do [i] ← A [p+ i-1]
8. 6. for j ← 1 to n2
9. 7. do R[j] ← A[q + j]
10. 8. L [n 1+ 1] ← ∞
11. 9. R[n 2+ 1] ← ∞
12. 10. I ← 1
13. 11. J ← 1
14. 12. For k ← p to r
15. 13. Do if L [i] ≤ R[j]
16. 14. then A[k] ← L[i]
17. 15. i ← i +1
18. 16. else A[k] ← R[j]
19. 17. j ← j+1

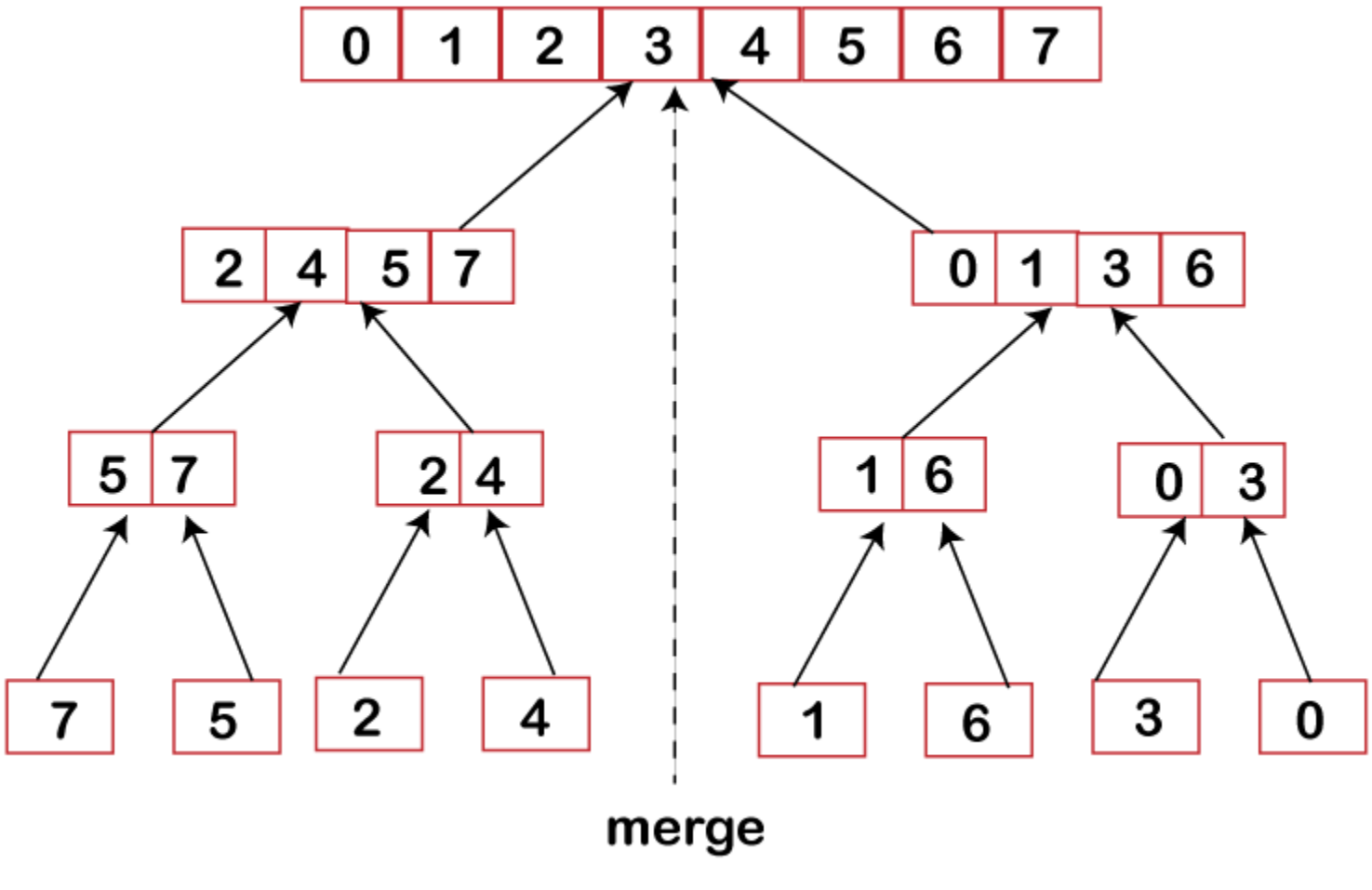


Figure 2: Merge Sort Combine Phase

The merge step of Merge Sort

Mainly the recursive algorithm depends on a base case as well as its ability to merge back the results derived from the base cases. Merge sort is no different algorithm, just the fact here the **merge** step possesses more importance.

To any given problem, the merge step is one such solution that combines the two individually sorted lists(arrays) to build one large sorted list(array).

The merge sort algorithm upholds three pointers, i.e., one for both of the two arrays and the other one to preserve the final sorted array's current index.

1. Did you reach the end of the array?
2. No:
3. Firstly, start with comparing the current elements of both the arrays.
4. Next, copy the smaller element into the sorted array.
5. Lastly, move the pointer of the element containing a smaller element.
6. Yes:
7. Simply copy the rest of the elements of the non-empty array

Merge( ) Function Explained Step-By-Step

Consider the following example of an unsorted array, which we are going to sort with the help of the Merge Sort algorithm.

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

Step1: The merge sort algorithm iteratively divides an array into equal halves until we achieve an atomic value. In case if there are an odd number of elements in an array, then one of the halves will have more elements than the other half.

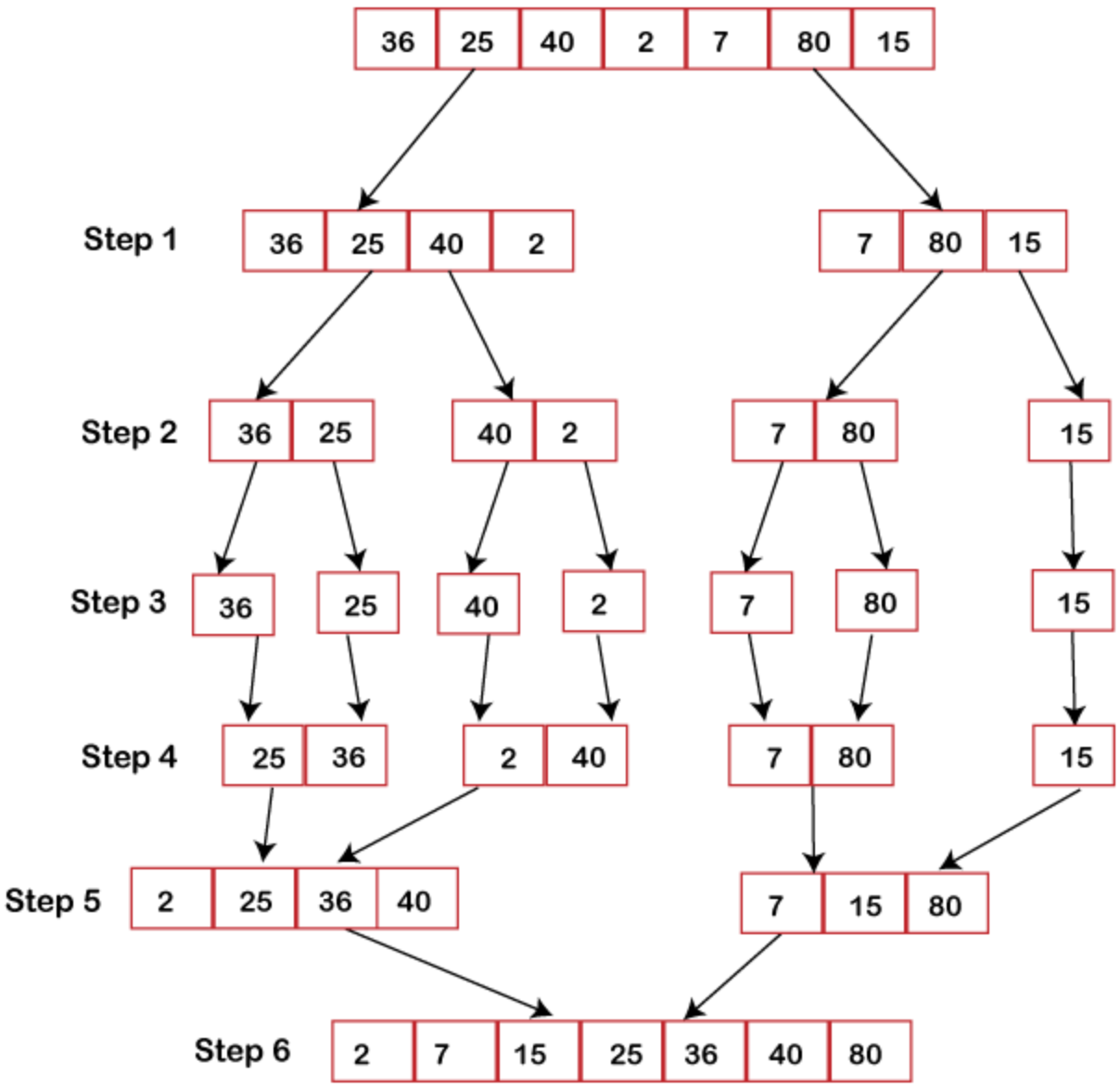
Step2: After dividing an array into two subarrays, we will notice that it did not hamper the order of elements as they were in the original array. After now, we will further divide these two arrays into other halves.

Step3: Again, we will divide these arrays until we achieve an atomic value, i.e., a value that cannot be further divided.

Step4: Next, we will merge them back in the same way as they were broken down.

Step5: For each list, we will first compare the element and then combine them to form a new sorted list.

Step6: In the next iteration, we will compare the lists of two data values and merge them back into a list of found data values, all placed in a sorted manner.



Hence the array is sorted.

Analysis of Merge Sort:

Let T (n) be the total time taken by the Merge Sort algorithm.

- Sorting two halves will take at the most $2T\frac{n}{2}$ time.
- When we merge the sorted lists, we come up with a total n-1 comparison because the last element which is left will need to be copied down in the combined list, and there will be no comparison.

Thus, the relational formula will be

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

But we ignore '-1' because the element will take some time to be copied in merge lists.

So $T(n) = 2T\left(\frac{n}{2}\right) + n$...equation 1

> ***Note: Stopping Condition T (1) =0 because at last, there will be only 1 element left that need to be copied, and there will be no comparison.***

Putting $n=\frac{n}{2}$ in place of n in .....equation 1

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + \frac{n}{2}$$.....equation2

Put 2 equation in 1 equation

$$T(n) = 2\left[2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right] + n$$

$$= 2^2T\left(\frac{n}{2^2}\right) + \frac{2n}{2} + n$$

$$T(n) = 2^2\,T\left(\frac{n}{2^2}\right) + 2n$$.....equation 3

Putting $n = \frac{n}{2^2}$ in equation 1

$$T\left(\frac{n}{2^3}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}$$.....equation4

Putting 4 equation in 3 equation

$$T(n) = 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + 2n$$

$$T(n) = 2^3\,T\left(\frac{n}{2^3}\right) + n + 2n$$

$$T(n) = 2^3\,T\left(\frac{n}{2^3}\right) + 3n$$.....equation5

From eq 1, eq3, eq 5.....we get

$$T(n) = 2^i\,T\left(\frac{n}{2^i}\right) + in$$.....equation6

From Stopping Condition:

$$\frac{n}{2^i} = 1 \text{ And } T\left(\frac{n}{2^i}\right)=0$$

$$n = 2^i$$

Apply log both sides:

log $n=\log_2 i$

$\log n = i \log 2$

$\frac{\log n}{\log 2} = i$

$\log_2 n = i$

From 6 equation

$$T(n) = 2^i\,T\left(\frac{n}{2^i}\right) + in$$

$$= 2^i \times 0 + \log_2 n \cdot n$$

$$= T(n) = n.\log n$$

Best Case Complexity: The merge sort algorithm has a best-case time complexity of **O(n*log n)** for the already sorted array.

Average Case Complexity: The average-case time complexity for the merge sort algorithm is **O(n*log n)**, which happens when 2 or more elements are jumbled, i.e., neither in the ascending order nor in the descending order.

Worst Case Complexity: The worst-case time complexity is also **O(n*log n)**, which occurs when we sort the descending order of an array into the ascending order.

Space Complexity: The space complexity of merge sort is **O(n)**.

Merge Sort Applications

The concept of merge sort is applicable in the following areas:

- Inversion count problem
- External sorting
- E-commerce applications

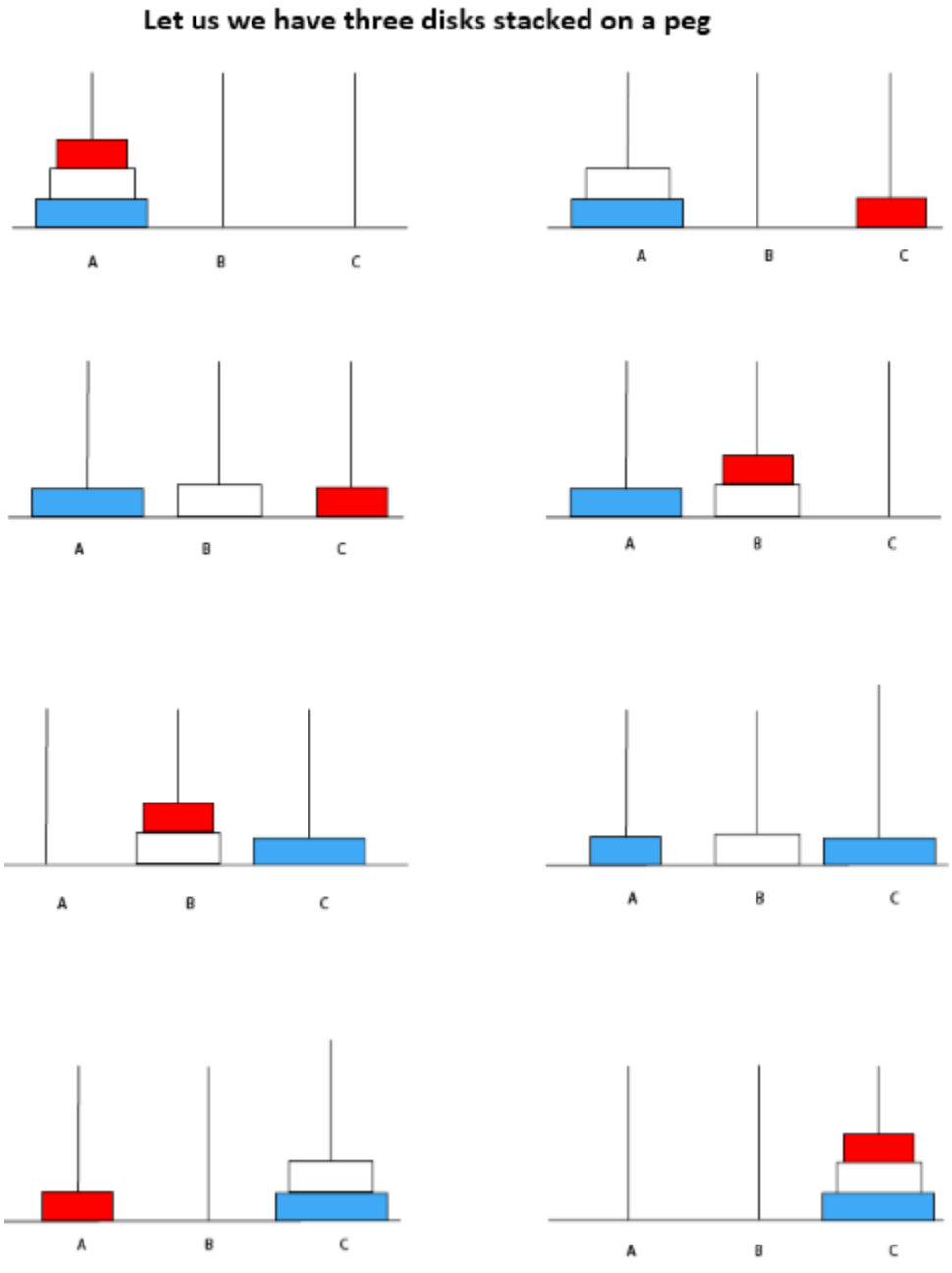
Tower of Hanoi

1. It is a classic problem where you try to move all the disks from one peg to another peg using only three pegs.

2. Initially, all of the disks are stacked on top of each other with larger disks under the smaller disks.

3. You may move the disks to any of three pegs as you attempt to relocate all of the disks, but you cannot place the larger disks over smaller disks and only one disk can be transferred at a time.

This problem can be easily solved by Divide & Conquer algorithm



In the above 7 step all the disks from peg A will be transferred to C given Condition:

1. Only one disk will be shifted at a time.
2. Smaller disk can be placed on larger disk.

Let T (n) be the total time taken to move n disks from peg A to peg C

1. Moving n-1 disks from the first peg to the second peg. This can be done in T (n-1) steps.
2. Moving larger disks from the first peg to the third peg will require first one step.
3. Recursively moving n-1 disks from the second peg to the third peg will require again T (n-1) step.

So, total time taken T (n) = T (n-1)+1+ T(n-1)

Relation formula for Tower of Hanoi is:

T (n) = 2T (n-1) +1

Note: Stopping Condition: T (1) =1

Because at last there will be one disk which will have to move from one peg to another.

T (n) = 2T (n-1) +1.....eq1

Put n = n-1 in eq 1

T (n-1) = 2T (n-2) +1.....eq2

Putting 2 eq in 1 eq

T (n) = 2(2T (n-2) +1) +1

T (n) = 2^2T (n-2) +2+1.....eq3

Put n = n-2 in eq 1

T (n-2) = 2T (n-3) +1.....eq4

Putting 4 eq in 3 eq

$T(n) = 2^2[2T(n-3)+1] + 2 + 1$

$T(n) = 2^3T(n-3) + 2^2 + 2 + 1$.....eq5

From 1 eq, 3 eq, 5 eq

We get,

$$T(n) = 2^iT(n-i) + 2^{i-1} + 2^{i-2} + \ldots\ldots\ldots\ldots + 2^0$$

Now n-i=1 from stopping condition

And T (n-i) =1

n-1 =i ← A equation

Now, $T(n) = 2^i(1) + 2^{i-1} + 2^{i-2} + \ldots\ldots\ldots\ldots + 2^0$

It is a Geometric Progression Series with common ratio, r=2
First term, a=1(2^0)

Sum of n terms in G.P= $S_n = \frac{a(1-r^n)}{1-r}$

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

Because exponents are from 0 to 1

n=i+1

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

$T(n) = 2^{i+1} - 1$

From A

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

$T(n) = 2^n - 1$.....B Equation

B equation is the required complexity of technique tower of Hanoi when we have to move n disks from one peg to another.

T (3) = 2^3- 1
= 8 - 1 = **7 Ans**

[As in concept we have proved that there will be 7 steps now proved by general equation]

Program of Tower of Hanoi:

```c
#include<stdio.h>
void towers(int, char, char, char);
 int main()
 {
      int num;
      printf ("Enter the number of disks : ");
       scanf ("%d", &num);
     printf ("The sequence of moves involved in the Tower of Hanoi are :\n");
     towers (num, 'A', 'C', 'B');
     return 0;

}
   void towers( int num, char from peg, char topeg, char auxpeg)
{
        if (num == 1)
{
     printf ("\n Move disk 1 from peg %c to peg %c", from peg, topeg);
        return;
 }
  Towers (num - 1, from peg, auxpeg, topeg);
   Printf ("\n Move disk %d from peg %c to peg %c", num, from peg, topeg);
  Towers (num - 1, auxpeg, topeg, from peg);
 }
```

Output:

```
Enter the number of disks: 3
The sequence of moves involved in the Tower of Hanoi is
Move disk 1 from peg A to peg C
Move disk 2 from peg A to peg B
Move disk 1 from peg C to peg B
Move disk 3 from peg A to peg C
Move disk 1 from peg B to peg A
Move disk 2 from peg B to peg C
Move disk 1 from peg A to peg
```

METHOD NAME	EQUATION	STOPPING CONDITION	COMPLEXITIES
1.Max&Min	$T(n) = 2T\left(\frac{n}{2}\right)+2$	T(2)=1	$T(n) = \frac{3N}{2} - 2$
2.Binary Search	$T(n) = T\left(\frac{n}{2}\right) + 1$	T(1)=1	T(n)=log n
3.Merge Sort	$T(n)=2T\left(\frac{n}{2}\right)+n$	T(1)=0	T(n)=nlog n
4. Tower of Hanoi	T(n)=2T(n-1)+1	T(1)=1	$T(n)=2^n-1$

Heap Sort

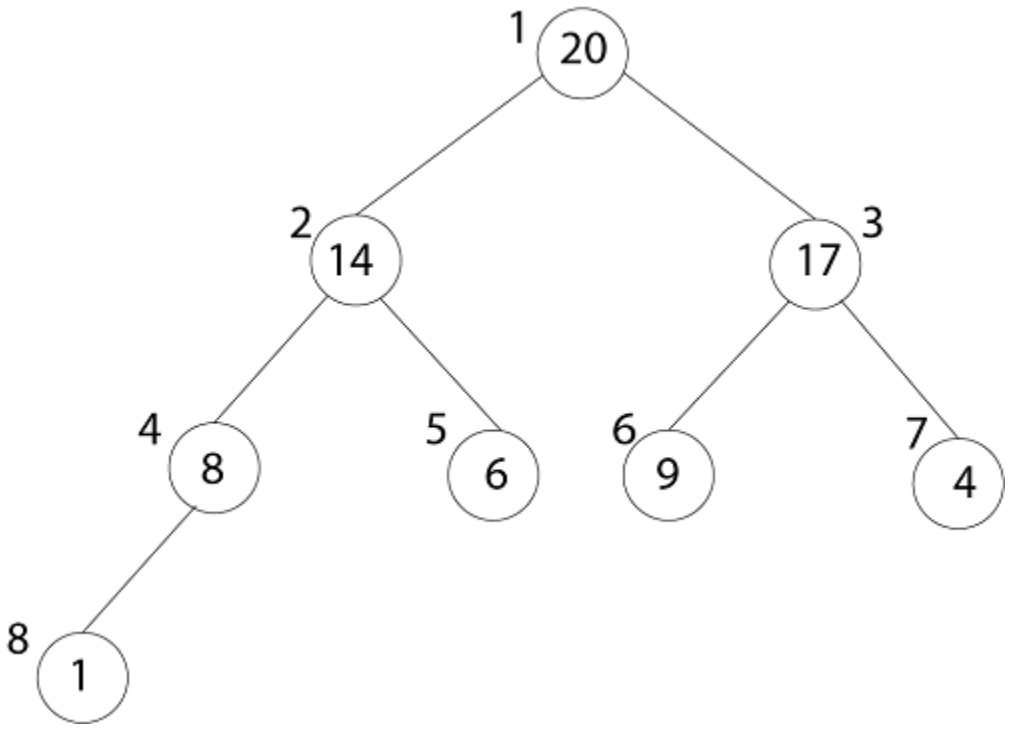
Binary Heap:

Binary Heap is an array object can be viewed as Complete Binary Tree. Each node of the Binary Tree corresponds to an element in an array.

1. Length [A],number of elements in array
2. Heap-Size[A], number of elements in a heap stored within array A.

The root of tree A [1] and gives index 'i' of a node that indices of its parents, left child, and the right child can be computed.

1. PARENT (i)
2. Return floor (i/2)
3. LEFT (i)
4. Return 2i
5. RIGHT (i)
6. Return 2i+1



Representation of an array of the above figure is given below:

1	2	3	4	5	6	7	8	9
20	14	17	8	6	9	4	1	

The index of 20 is 1

To find the index of the left child, we calculate 1*2=2

This takes us (correctly) to the 14.

Now, we go right, so we calculate 2*2+1=5

This takes us (again, correctly) to the 6.

Now, 4's index is 7, we want to go to the parent, so we calculate 7/2 =3 which takes us to the 17.

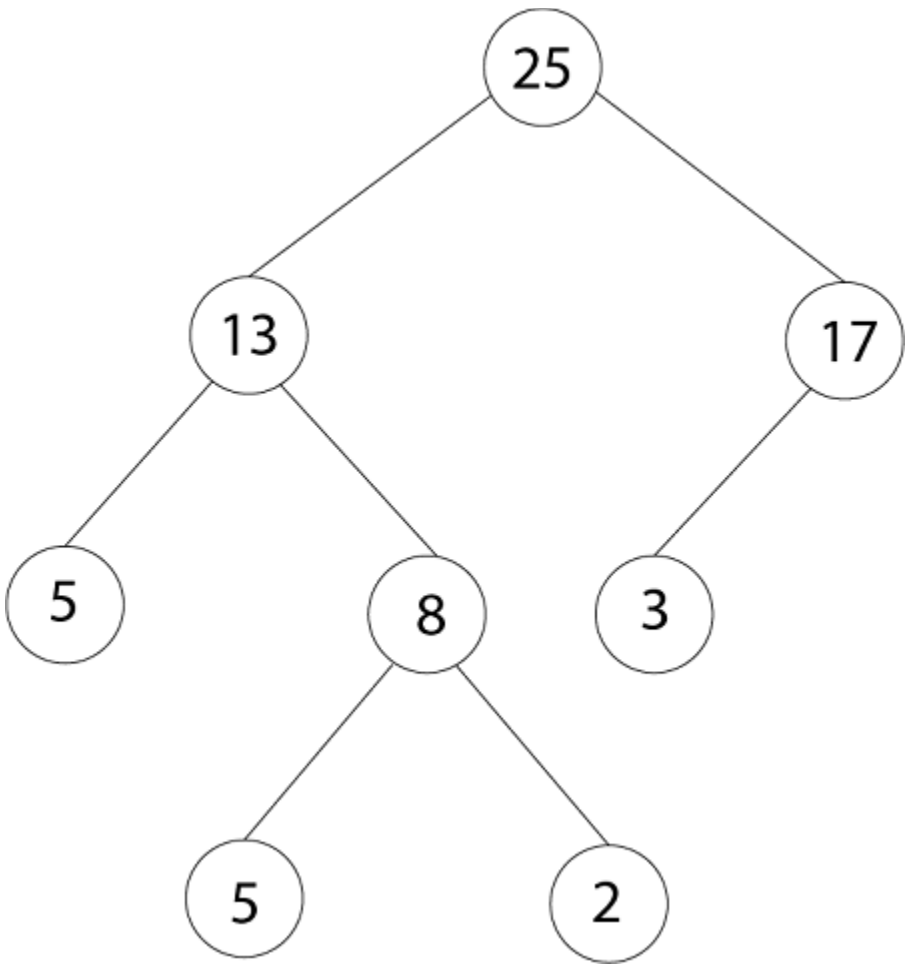
Heap Property:

A binary heap can be classified as Max Heap or Min Heap

1. Max Heap: In a Binary Heap, for every node I other than the root, the value of the node is greater than or equal to the value of its highest child

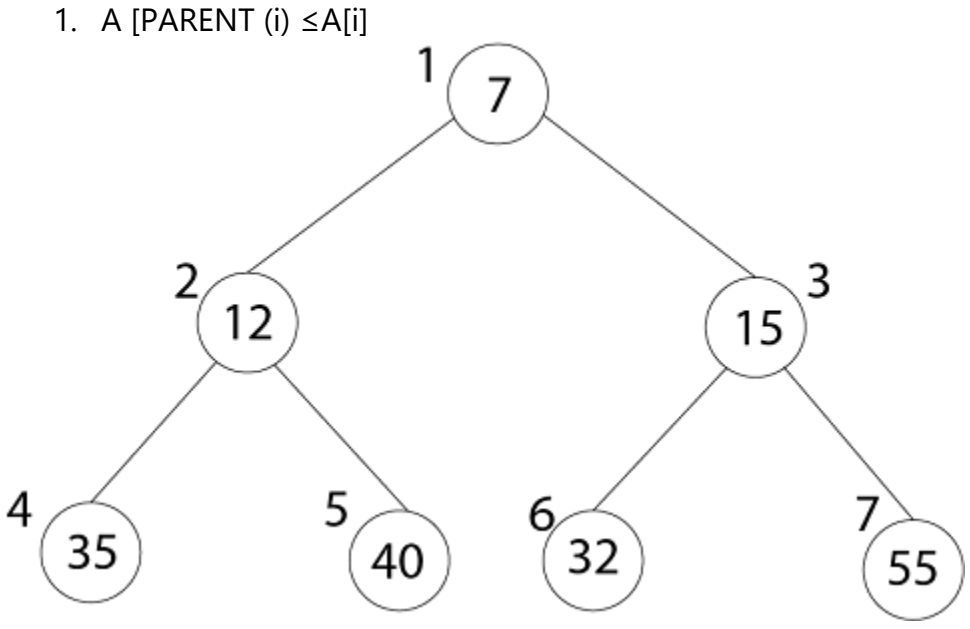
1. A [PARENT (i) ≥A[i]

Thus, the highest element in a heap is stored at the root. Following is an example of MAX-HEAP



2. MIN-HEAP: In MIN-HEAP, the value of the node is lesser than or equal to the value of its lowest child.

1. A [PARENT (i) ≤A[i]



Heapify Method:

1. Maintaining the Heap Property: Heapify is a procedure for manipulating heap Data Structure. It is given an array A and index I into the array. The subtree rooted at the children of A [i] are heap but node A [i] itself may probably violate the heap property i.e. A [i] < A [2i] or A [2i+1]. The procedure 'Heapify' manipulates the tree rooted as A [i] so it becomes a heap.

MAX-HEAPIFY (A, i)

```
1. l ← left [i]
2. r ← right [i]
3. if l≤ heap-size [A] and A[l] > A [i]
4. then largest ← l
5. Else largest ← i
6. If r≤ heap-size [A] and A [r] > A[largest]
7. Then largest ← r
8. If largest ≠ i
9. Then exchange A [i]    A [largest]
10. MAX-HEAPIFY (A, largest)
```

Analysis:

The maximum levels an element could move up are Θ (log n) levels. At each level, we do simple comparison which O (1). The total time for heapify is thus O (log n).

Building a Heap:

```
BUILDHEAP (array A, int n)
```

```
1 for i ← n/2 down to 1
2 do
3 HEAPIFY (A, i, n)
```

HEAP-SORT ALGORITHM:

HEAP-SORT (A)

```
1. BUILD-MAX-HEAP (A)
2. For I ← length[A] down to Z
3. Do exchange A [1] ⟷ A [i]
4. Heap-size [A] ← heap-size [A]-1
5. MAX-HEAPIFY (A,1)
```

Analysis: Build max-heap takes O (n) running time. The Heap Sort algorithm makes a call to 'Build Max-Heap' which we take O (n) time & each of the (n-1) calls to Max-heap to fix up a new heap. We know 'Max-Heapify' takes time O (log n)

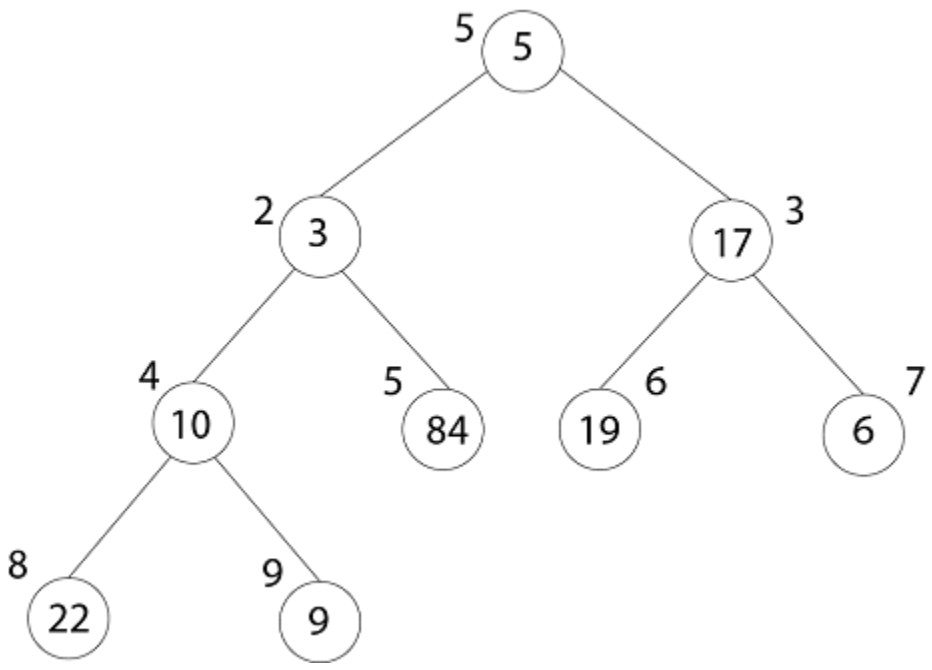
The total running time of Heap-Sort is O (n log n).

Example: Illustrate the Operation of BUILD-MAX-HEAP on the array.

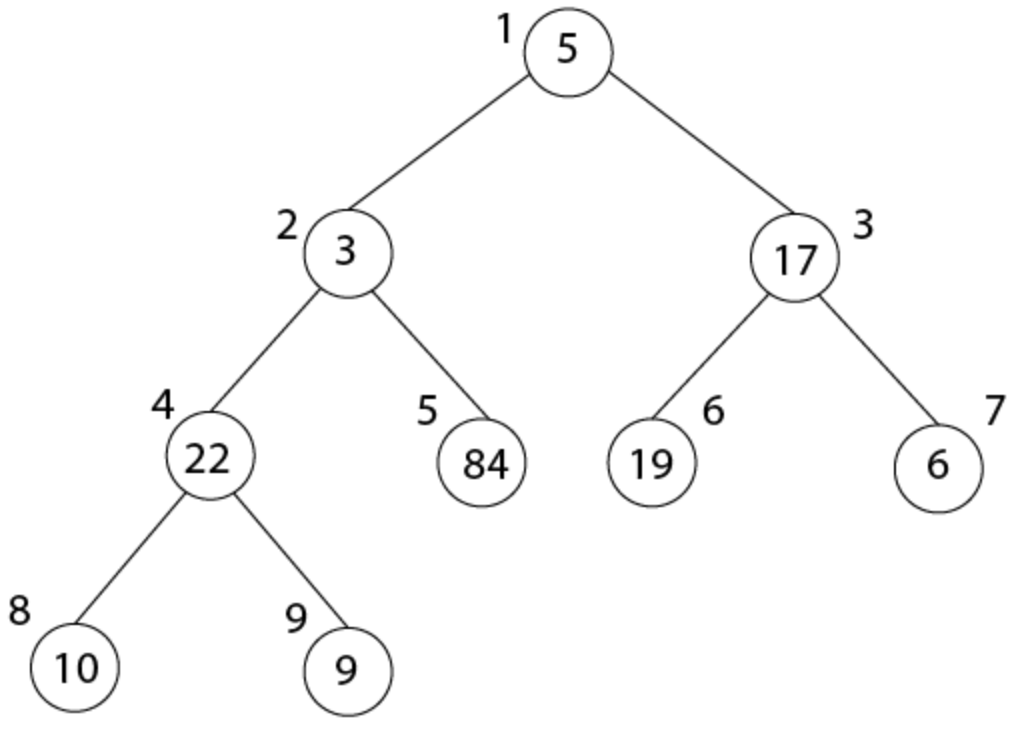
1. A = (5, 3, 17, 10, 84, 19, 6, 22, 9)

Solution: Originally:

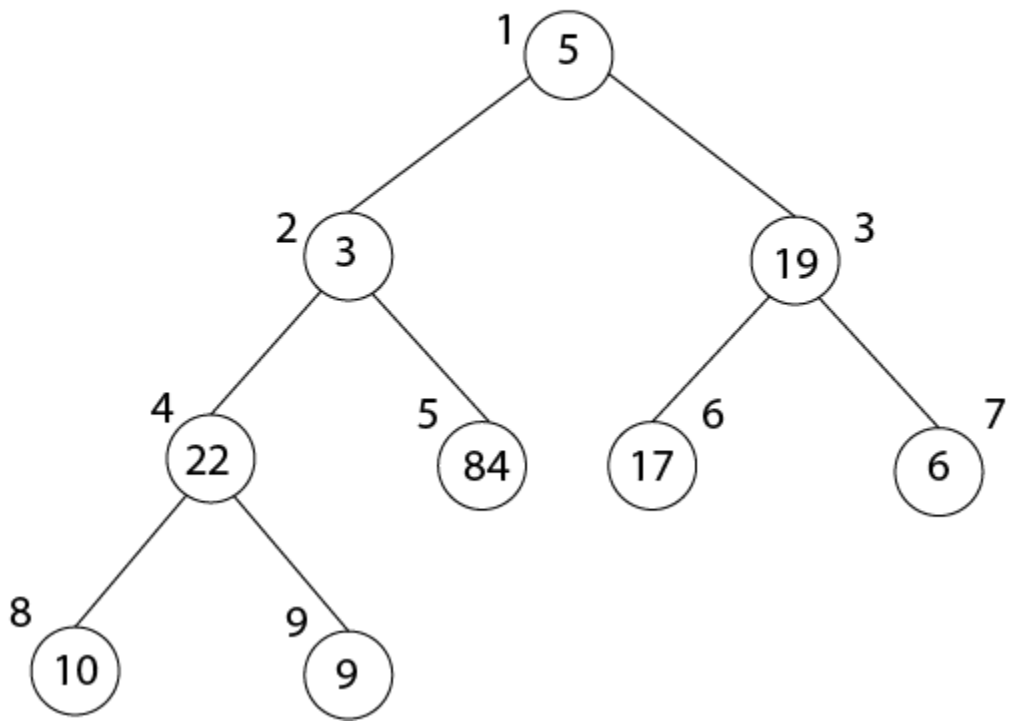
1. Heap-Size (A) =9, so first we call MAX-HEAPIFY (A, 4)
2. And I = 4.5= 4 to 1



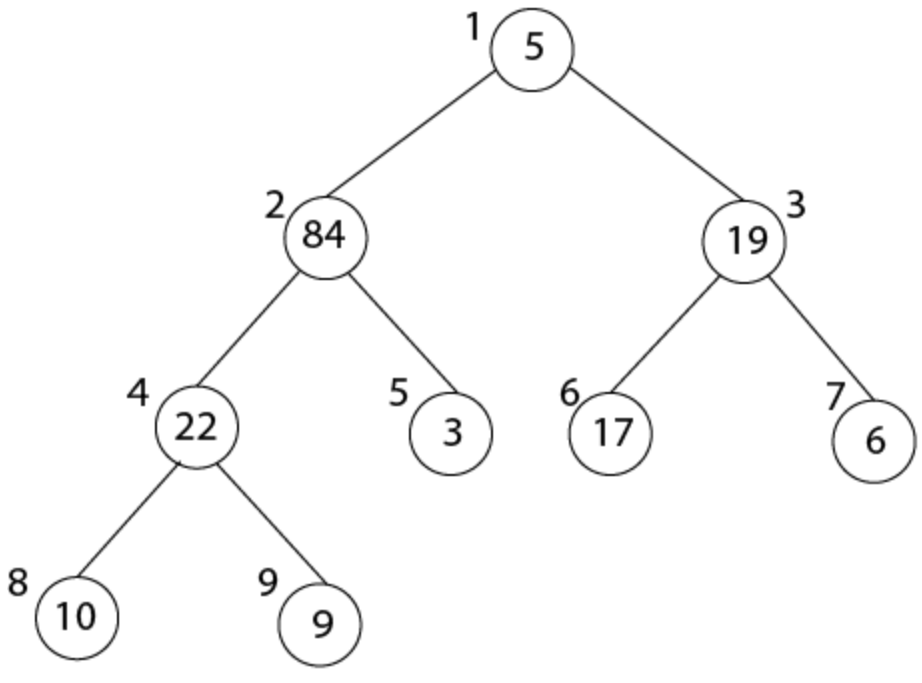
1. After MAX-HEAPIFY (A, 4) and i=4
2. L ← 8, r ← 9
3. l≤ heap-size[A] and A [l] >A [i]
4. 8 ≤9 and 22>10
5. Then Largest ← 8
6. If r≤ heap-size [A] and A [r] > A [largest]
7. 9≤9 and 9>22
8. If largest (8) ≠4
9. Then exchange A [4] ←→ A [8]
10. MAX-HEAPIFY (A, 8)



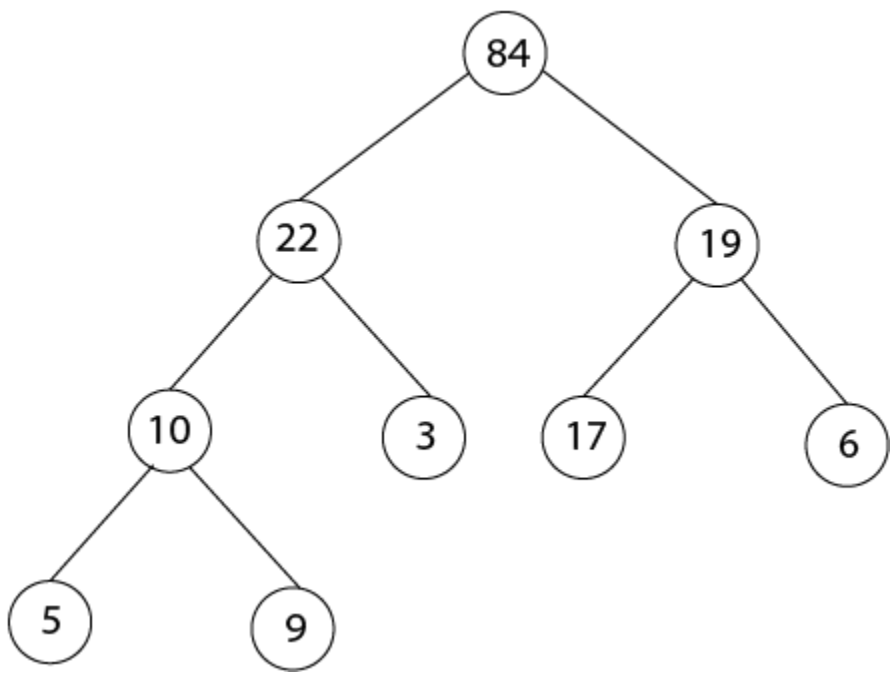
1. After MAX-HEAPIFY (A, 3) and i=3
2. l← 6, r ← 7
3. l≤ heap-size[A] and A [l] >A [i]
4. 6≤ 9 and 19>17
5. Largest ← 6
6. If r≤ heap-size [A] and A [r] > A [largest]
7. 7≤9 and 6>19
8. If largest (6) ≠3
9. Then Exchange A [3] ←→ A [6]
10. MAX-HEAPIFY (A, 6)



1. After MAX-HEAPIFY (A, 2) and i=2
2. l ← 4, r ← 5
3. l≤ heap-size[A] and A [l] >A [i]
4. 4≤9 and 22>3
5. Largest ← 4
6. If r≤ heap-size [A] and A [r] > A [largest]
7. 5≤9 and 84>22
8. Largest ← 5
9. If largest (4) ≠2
10. Then Exchange A [2] ←→ A [5]
11. MAX-HEAPIFY (A, 5)



1. After MAX-HEAPIFY (A, 1) and i=1
2. l ← 2, r ← 3
3. l≤ heap-size[A] and A [l] >A [i]
4. 2≤9 and 84>5
5. Largest ← 2
6. If r≤ heap-size [A] and A [r] > A [largest]
7. 3≤9 and 19<84
8. If largest (2) ≠1
9. Then Exchange A [1] ←→ A [2]
10. MAX-HEAPIFY (A, 2)



Priority Queue:

As with heaps, priority queues appear in two forms: max-priority queue and min-priority queue.

A priority queue is a data structure for maintaining a set S of elements, each with a combined value called a key. A max-priority queue guides the following operations:

INSERT(S, x): inserts the element x into the set S, which is proportionate to the operation S=S∪[x].

MAXIMUM (S) returns the element of S with the highest key.

EXTRACT-MAX (S) removes and returns the element of S with the highest key.

INCREASE-KEY(S, x, k) increases the value of element x's key to the new value k, which is considered to be at least as large as x's current key value.

Let us discuss how to implement the operations of a max-priority queue. The procedure HEAP-MAXIMUM consider the MAXIMUM operation in θ (1) time.

HEAP-MAXIMUM (A)

1. return A [1]

The procedure HEAP-EXTRACT-MAX implements the EXTRACT-MAX operation. It is similar to the for loop of Heap-Sort procedure.

HEAP-EXTRACT-MAX (A)

```
1 if A. heap-size < 1
2 error "heap underflow"
3 max ← A [1]
4 A [1] ← A [heap-size [A]]
5 heap-size [A] ← heap-size [A]-1
6 MAX-HEAPIFY (A, 1)
7 return max
```

The procedure HEAP-INCREASE-KEY implements the INCREASE-KEY operation. An index i into the array identify the priority-queue element whose key we wish to increase.

HEAP-INCREASE-KEY.A, i, key)

```
1 if key < A[i]
2 errors "new key is smaller than current key"
3 A[i] = key
4 while i>1 and A [Parent (i)] < A[i]
5 exchange A [i] with A [Parent (i)]
6 i =Parent [i]
```

The running time of HEAP-INCREASE-KEY on an n-element heap is O (log n) since the path traced from the node updated in line 3 to the root has length O (log n).

The procedure MAX-HEAP-INSERT implements the INSERT operation. It takes as an input the key of the new item to be inserted into max-heap A. The procedure first expands the max-heap by calculating to the tree a new leaf whose key is - ∞. Then it calls HEAP-INCREASE-KEY to set the key of this new node to its right value and maintain the max-heap property

MAX-HEAP-INSERT (A, key)

```
1 A. heap-size = A. heap-size + 1
2 A [A. heap-size] = - ∞
3 HEAP-INCREASE-KEY (A, A. heap-size, key)
```

The running time of MAX-HEAP-INSERT on an n-element heap is O (log n).

Example: Illustrate the operation of HEAP-EXTRACT-MAX on the heap

1. A= (15,13,9,5,12,8,7,4,0,6,2,1)

Fig: Operation of HEAP-INCREASE-KEY

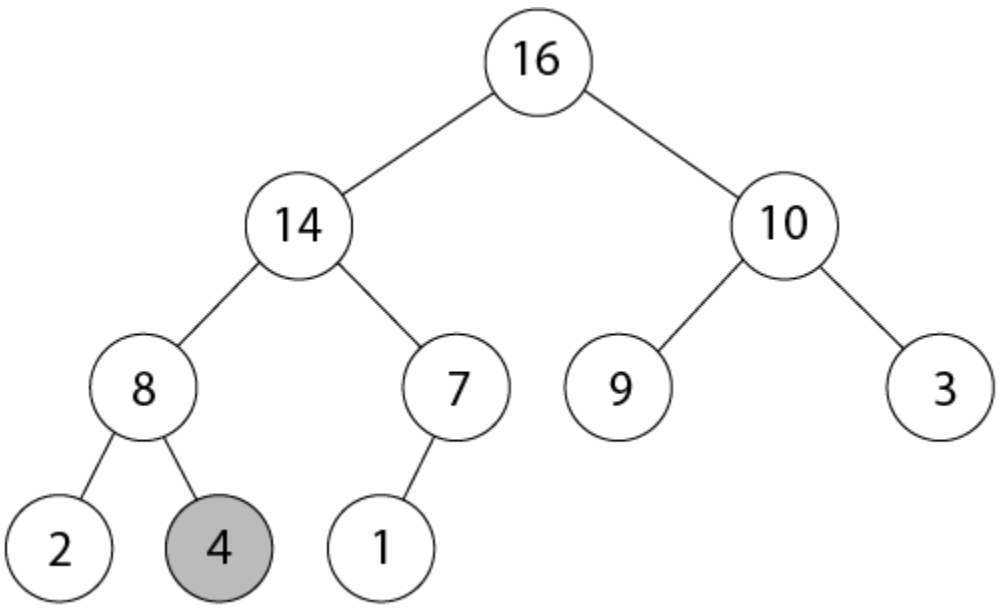


Fig: (a)

In this figure, that max-heap with a node whose index is 'i' heavily shaded

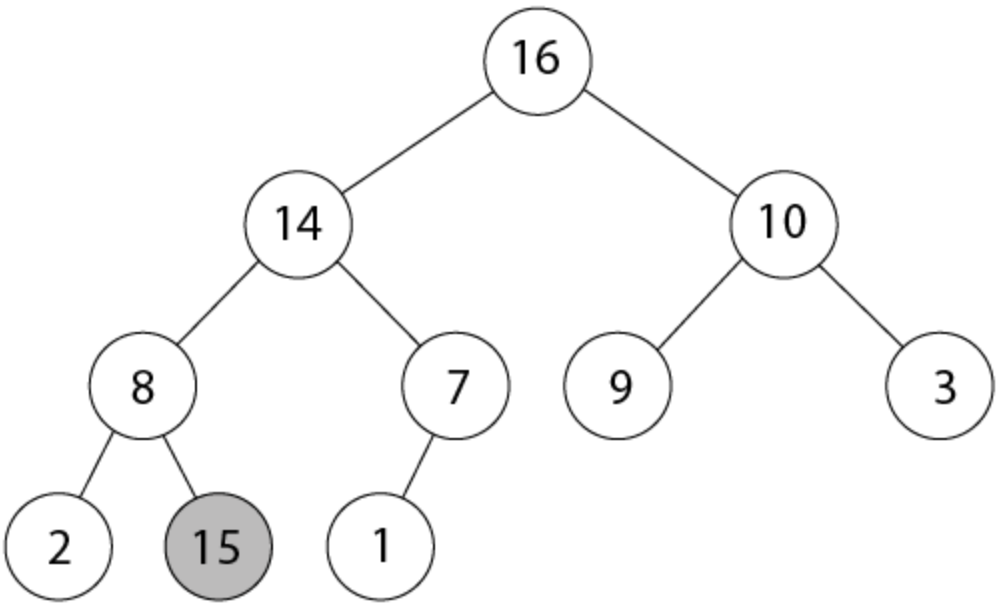


Fig: (b)

In this Figure, this node has its key increased to 15.

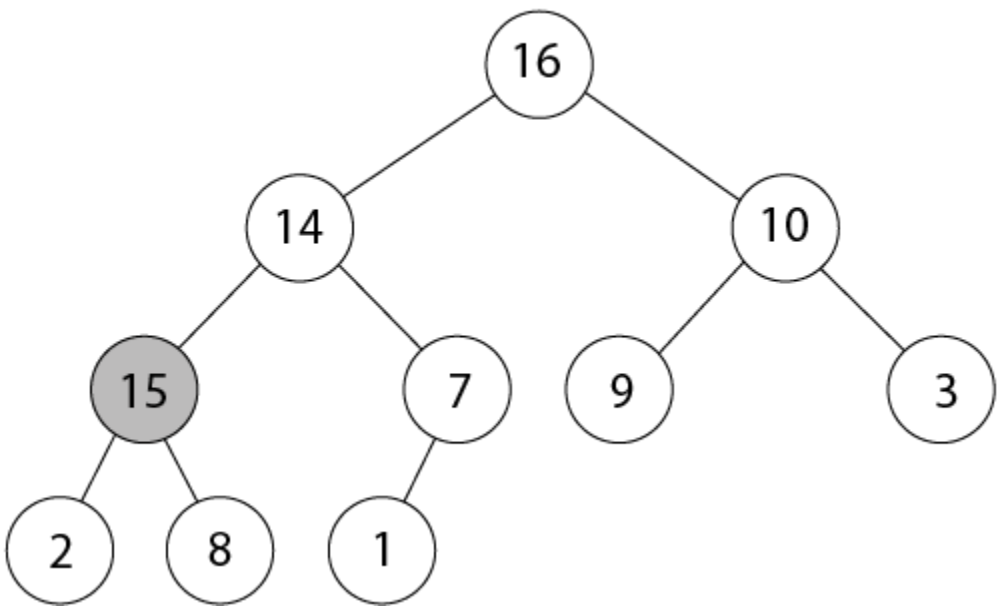


Fig: (c)

After one iteration of the while loop of lines 4-6, the node and its parent have exchanged keys, and the index i moves up to the parent.

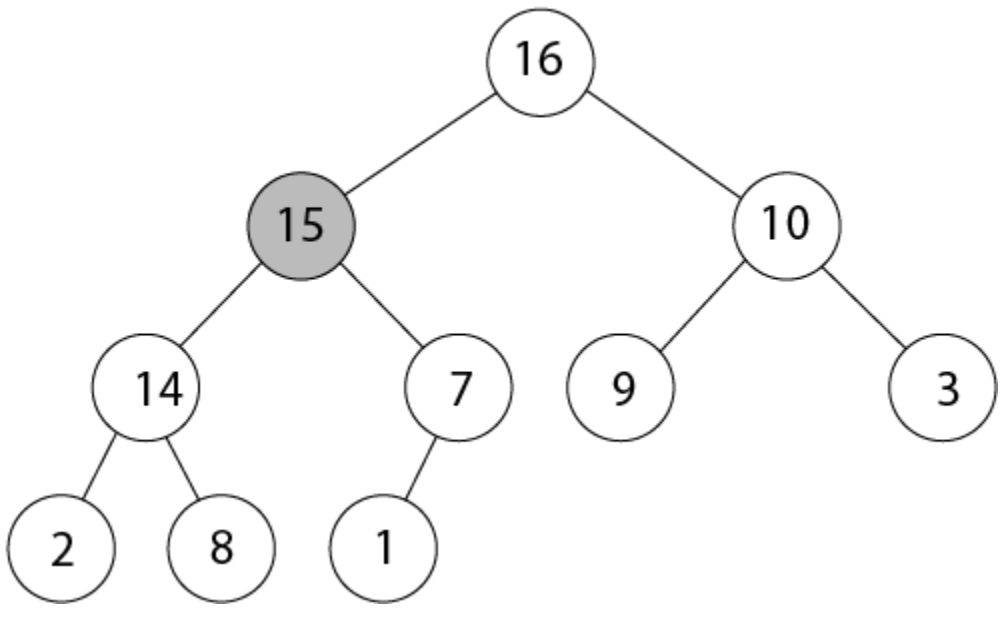


Fig: (d)

The max-heap after one more iteration of the while loops, the A [PARENT (i) ≥A (i)] the max-heap property now holds and the procedure terminates.

Heap-Delete:

Heap-DELETE (A, i) is the procedure, which deletes the item in node 'i' from heap A, HEAP-DELETE runs in O (log n) time for n-element max heap.

HEAP-DELETE (A, i)

```
1. A [i] ← A [heap-size [A]]
2. Heap-size [A] ← heap-size [A]-1
3. MAX-HEAPIFY (A, i)
```

Quick sort

It is an algorithm of Divide & Conquer type.

Divide: Rearrange the elements and split arrays into two sub-arrays and an element in between search that each element in left sub array is less than or equal to the average element and each element in the right sub- array is larger than the middle element.

Conquer: Recursively, sort two sub arrays.

Combine: Combine the already sorted array.

Algorithm:

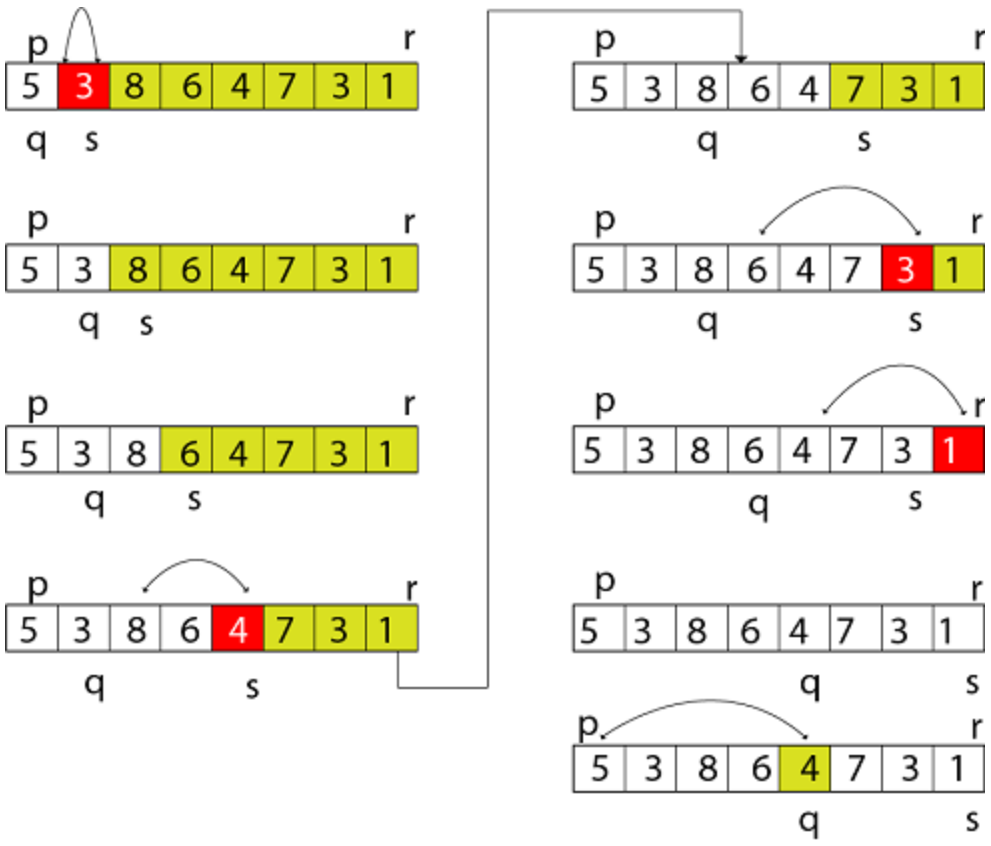
1. QUICKSORT (array A, **int** m, **int** n)
2. 1 **if** (n > m)
3. 2 then
4. 3 i ← a random index from [m,n]
5. 4 swap A [i] with A[m]
6. 5 o ← PARTITION (A, m, n)
7. 6 QUICKSORT (A, m, o - 1)
8. 7 QUICKSORT (A, o + 1, n)

Partition Algorithm:

Partition algorithm rearranges the sub arrays in a place.

1. PARTITION (array A, **int** m, **int** n)
2. 1 x ← A[m]
3. 2 o ← m
4. 3 **for** p ← m + 1 to n
5. 4 **do if** (A[p] < x)
6. 5 then o ← o + 1
7. 6 swap A[o] with A[p]
8. 7 swap A[m] with A[o]
9. 8 **return** o

Figure: shows the execution trace partition algorithm



Example of Quick Sort:

1. 44 33 11 55 77 90 40 60 99 22 88

Let **44** be the **Pivot** element and scanning done from right to left

Comparing **44** to the right-side elements, and if right-side elements are **smaller** than **44**, then swap it. As **22** is smaller than **44** so swap them.

22 33 11 55 77 90 40 60 99 **44** 88

Now comparing **44** to the left side element and the element must be **greater** than 44 then swap them. As **55** are greater than **44** so swap them.

22 33 11 **44** 77 90 40 60 99 **55** 88

Recursively, repeating steps 1 & steps 2 until we get two lists one left from pivot element **44** & one right from pivot element.

22 33 11 **40** 77 90 **44** 60 99 55 88

Swap with 77:

22 33 11 40 **44** 90 **77** 60 99 55 88

Now, the element on the right side and left side are greater than and smaller than **44** respectively.

Now we get two sorted lists:

22 33 11 40 **44** 90 77 66 99 55 88

Sublist1 **Sublist2**

And these sublists are sorted under the same process as above done.

These two sorted sublists side by side.

22 33 11 40 **44** **90** 77 60 99 55 88

11 33 **22** 40 **44** 88 77 60 99 55 **90**

11 **22** 33 40 **44** 88 77 60 **90** 55 **99**

First sorted list

88 77 60 **55** **90** 99

Sublist3 **Sublist4**

55 77 60 **88** 90 99

Sorted

55 **77** **60**

55 60 77

Sorted

Merging Sublists:

11	**22**	**33**	**40**	**44**	**55**	**60**	**77**	**88**	**90**	**99**

SORTED LISTS

Worst Case Analysis: It is the case when items are already in sorted form and we try to sort them again. This will takes lots of time and space.

Equation:

1. T (n) =T(1)+T(n-1)+n

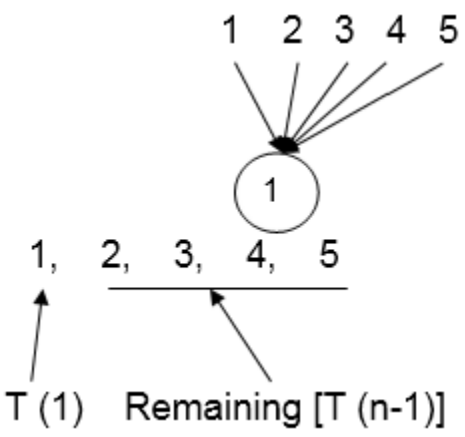
T (1) is time taken by pivot element.

T (n-1) is time taken by remaining element except for pivot element.

N: the number of comparisons required to identify the exact position of itself (every element)

If we compare first element pivot with other, then there will be 5 comparisons.

It means there will be n comparisons if there are n items.



Relational Formula for Worst Case:

T (n) =T (1) + T (n-1) + n…… (1)

T (n-1) =T (1) +T (n-1-1) + (n-1)

Put T (n-1) in equation1

> By putting (n-1) in place of n in equation1

T(n)=T(1)+T(1)+(T(n-2)+(n-1)+n……(ii)

T (n) =2T (1) +T (n-2) + (n-1) +n

T (n-2) =T (1) +T (n-3) + (n-2)

Put T (n-2) in equation (ii)

> By putting (n-2) in place of n in equation1

T(n)=2T(1)+T(1)+T(n-3)+(n-2)+(n-1)+n

T(n)=3T(1)+T(n-3)+)+(n-2)+(n-1)+n

T (n-3) =T (1) +T (n-4) +n-3

> By putting (n-3) in place of n in equation1

T(n)=3T(1)+T(1)+T(n-4)+(n-3)+(n-2)+(n-1)+n

=4T(1)+T(n-4))+(n-3)+(n-2)+(n-1)+n……(iii)

Note: for making T (n-4) as T (1) we will put (n-1) in place of '4' and if We put (n-1) in place of 4 then we have to put (n-2) in place of 3 and (n-3) In place of 2 and so on.

T(n)=(n-1) T(1) + T(n-(n-1))+(n-(n-2))+(n-(n-3))+(n-(n-4))+n

T (n) = (n-1) T (1) + T (1) + 2 + 3 + 4+.....n

T (n) = (n-1) T (1) +T (1) +2+3+4+.....+n+1-1

[Adding 1 and subtracting 1 for making AP series]

T (n) = (n-1) T (1) +T (1) +1+2+3+4+..... + n-1

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

Stopping Condition: T (1) =0

Because at last there is only one element left and no comparison is required.

T (n) = (n-1) (0) +0+ $\frac{n(n+1)}{2}$ -1

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

> Avoid all the terms expect higher terms n^2

T (n) =O (n^2)

Worst Case Complexity of Quick Sort is T (n) =O (n^2)

Randomized Quick Sort [Average Case]:

Generally, we assume the first element of the list as the pivot element. In an average Case, the number of chances to get a pivot element is equal to the number of items.

1. Let total time taken =T (n)
2. For eg: In a given list
3. p 1, p 2, p 3, p 4.....pn
4. If p 1 is the pivot list then we have 2 lists.
5. I.e. T (0) and T (n-1)
6. If p2 is the pivot list then we have 2 lists.
7. I.e. T (1) and T (n-2)
8. p 1, p 2, p 3, p 4.....pn
9. If p3 is the pivot list then we have 2 lists.
10. I.e. T (2) and T (n-3)
11. p 1, p 2, p 3, p 4.....p n

So in general if we take the **Kth** element to be the pivot element.

Then,

$$T(n) = \sum_{K=1}^{n} T(K-1) + T(n-k)$$

Pivot element will do n comparison and we are doing average case so,

$$T(n) = n+1 + \frac{1}{n}\left(\sum_{K=1}^{n} T(K-1) + T(n-k)\right)$$

N comparisons — Average of n elements

So Relational Formula for Randomized Quick Sort is:

$$T(n) = n+1 + \frac{1}{n}\left(\left(\sum_{K=1}^{n} T(K-1) + T(n-k)\right)\right)$$

$$= n+1 + \frac{1}{n}(T(0)+T(1)+T(2)+...T(n-1)+T(n-2)+T(n-3)+...T(0))$$

$$= n+1 + \frac{1}{n}x2\ (T(0)+T(1)+T(2)+...T(n-2)+T(n-1))$$

1. $n\,T(n) = n(n+1) + 2\ (T(0)+T(1)+T(2)+...T(n-1)$.....eq 1

Put n=n-1 in eq 1

1. $(n-1)\,T(n-1) = (n-1)n + 2\,(T(0)+T(1)+T(2)+...T(n-2)$.....eq2

From eq1 and eq 2

$n\,T(n) - (n-1)\,T(n-1) = n(n+1)-n(n-1)+2\,(T(0)+T(1)+T(2)+?T(n-2)+T(n-1))-2(T(0)+T(1)+T(2)+...T(n-2))$

$n\,T(n) - (n-1)\,T(n-1) = n[n+1-n+1]+2T(n-1)$

$n\,T(n) = [2+(n-1)]T(n-1)+2n$

$n\,T(n) = n+1\,T(n-1)+2n$

$$\frac{n}{n+1}T(n) = \frac{2n}{n+1} + T(n-1) \qquad \text{[Divide by n+1]}$$

$$\frac{1}{n+1}T(n) = \frac{2}{n+1} + \frac{T(n-1)}{n} \qquad \text{[Divide by n] .....eq 3}$$

Put n=n-1 in eq 3

$$\frac{1}{n}T(n-1) = \frac{2}{n} + \frac{T(n-2)}{n-1}\text{.....eq4}$$

Put 4 eq in 3 eq

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{T(n-2)}{n-1}\text{.....eq5}$$

Put n=n-2 in eq 3

$$\frac{T(n-2)}{n-1} = \frac{2}{n-1} + \frac{2}{n} + \frac{T(n-3)}{n-2}\text{.....eq6}$$

Put 6 eq in 5 eq

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \frac{T(n-3)}{n-2}\text{.....eq7}$$

Put n=n-3 in eq 3

$$\frac{T(n-3)}{n-2} = \frac{2}{n-2} + \frac{T(n-4)}{n-3}\text{.....eq8}$$

Put 8 eq in 7 eq

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \frac{2}{n-2} + \frac{T(n-4)}{n-3} \ldots\ldots\ldots\ldots \text{eq9}$$

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

3 terms of $\frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} = T(n-3)$

4 terms of $\frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \frac{2}{n-2}$ =T(n-4)

From 3eq, 5eq, 7eq, 9 eq we get

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \ldots\ldots\ldots + \frac{2}{3} + \frac{T(n-(n-1))}{n-(n-2)} \ldots\ldots\ldots \text{eq10}$$

From 3 eq $\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{T(n-1)}{n}$

Put n=1

$$\frac{T(1)}{2} = \frac{2}{2} + \frac{T(0)}{1}$$

$$\frac{T(1)}{2} = 1$$

From 10 eq

$$\frac{T(n)}{n+1} = \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \ldots\ldots\ldots\ldots + \frac{2}{3} + \frac{T(1)}{2}$$

$$= \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \ldots\ldots\ldots\ldots + \frac{2}{3} + 1$$

Multiply and divide the last term by 2

$$= \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \ldots\ldots\ldots\ldots + \frac{2}{3} + 2 \times \frac{1}{2}$$

$$= 2\left[\frac{1}{2} + \frac{1}{3} + \frac{1}{4} \ldots\ldots + \frac{1}{n} + \frac{1}{n+1}\right]$$

$$= 2 \sum_{2 \le k \le n+1}^{n} \frac{1}{k} = 2 \int_{2}^{n+1} \frac{1}{k}$$

Multiply & divide k by n

$$= 2 \int_{2}^{n+1} \frac{1}{\frac{k.n}{n}}$$

Put $\frac{k}{n} = x$ and $\frac{1}{n} = dx$

$$\frac{T(n)}{n+1} = 2 \int_{2}^{n+1} \frac{1}{x}\, dx$$

$$= 2 \log x \Big|_{2}^{n+1}$$

=2[log (n+1)-log2]

T (n) =2(n+1) [log (n+1)-log2]

Ignoring Constant we get

T (n) =n log n

T (n) =O (n log n)

NOTE: $\int \frac{1}{x} dx = \log x$

Is the average case complexity of quick sort for sorting n elements.

3. Quick Sort [Best Case]: In any sorting, best case is the only case in which we don't make any comparison between elements that is only done when we have only one element to sort.

Method Name	Equation	Stopping Condition	Complexities
1.Quick Sort[Worst Case]	T(n)=T(n-1)+T(0)+n	T(1)=0	$T(n)=n^2$
2.Quick Sort[Average Case]	$T(n)=n+1+\frac{1}{n}(\sum_{K=1}^{n} T(k-1)+T(n-k))$		T(n)=nlogn

Stable Sorting

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array.

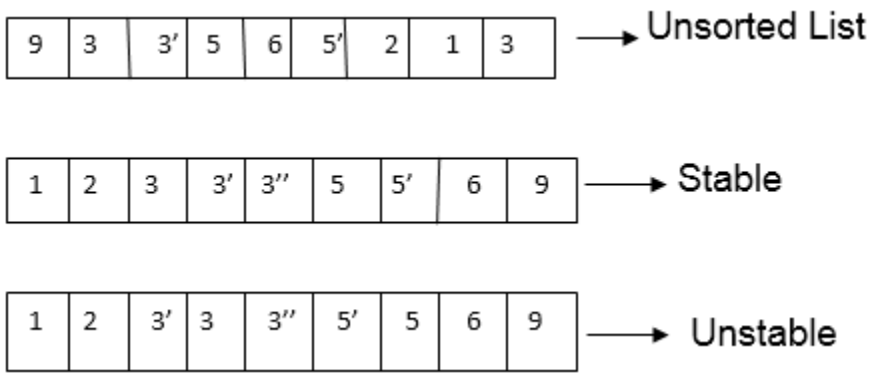
Some Sorting Algorithm is stable by nature like Insertion Sort, Merge Sort and Bubble Sort etc.

Sorting Algorithm is not stable like Quick Sort, Heap Sort etc.

Another Definition of Stable Sorting:

A Stable Sort is one which preserves the original order of input set, where the comparison algorithm does not distinguish between two or more items. A Stable Sort will guarantee that the original order of data having the same rank is preserved in the output.

In Place Sorting Algorithm:



1. An In-Place Sorting Algorithm directly modifies the list that is received as input instead of creating a new list that is then modified.
2. In this Sorting, a small amount of extra space it uses to manipulate the input set. In other Words, the output is placed in the correct position while the algorithm is still executing, which means that the input will be overwritten by the desired output on run-time.
3. In-Place, Sorting Algorithm updates input only through replacement or swapping of elements.
4. An algorithm which is not in-place is sometimes called not-in-Place or out of Place.
5. An Algorithm can only have a constant amount of extra space, counting everything including function call and Pointers, Usually; this space is O (log n).

Note:

1. Bubble sort, insertion sort, and selection sort are in-place sorting algorithms. Because only swapping of the element in the input array is required.
2. Bubble sort and insertion sort can be applying as stable algorithms but selection sort cannot (without significant modifications).
3. Merge sort is a stable algorithm but not an in-place algorithm. It requires extra array storage.
4. Quicksort is not stable but is an in-place algorithm.
5. Heap sort is an in-place algorithm but is not stable.

Lower Bound Theory

Lower Bound Theory Concept is based upon the calculation of minimum time that is required to execute an algorithm is known as a lower bound theory or Base Bound Theory.

Lower Bound Theory uses a number of methods/techniques to find out the lower bound.

Concept/Aim: The main aim is to calculate a minimum number of comparisons required to execute an algorithm.

Techniques:

The techniques which are used by lower Bound Theory are:

1. Comparisons Trees.
2. Oracle and adversary argument
3. State Space Method

1. Comparison trees:

In a comparison sort, we use only comparisons between elements to gain order information about an input sequence (a1; a2.....an).

Given a_i,a_j from (a_1, a_2.....a_n)We Perform One of the Comparisons

- $a_i < a_j$ less than
- $a_i \leq a_j$ less than or equal to
- $a_i > a_j$ greater than
- $a_i \geq a_j$ greater than or equal to
- $a_i = a_j$ equal to

To determine their relative order, if we assume all elements are distinct, then we just need to consider $a_i \leq a_j$ '=' is excluded &, ≥,≤,>,< are equivalent.

Consider sorting three numbers a1, a2, and a3. There are 3! = 6 possible combinations:

1. (a1, a2, a3), (a1, a3, a2),
2. (a2, a1, a3), (a2, a3, a1)
3. (a3, a1, a2), (a3, a2, a1)

The Comparison based algorithm defines a decision tree.

Decision Tree: A decision tree is a full binary tree that shows the comparisons between elements that are executed by an appropriate sorting algorithm operating on an input of a given size. Control, data movement, and all other conditions of the algorithm are ignored.

In a decision tree, there will be an array of length n.

So, total leaves will be n! (I.e. total number of comparisons)

If tree height is h, then surely

$$n! \leq 2^n \text{ (tree will be binary)}$$

Taking an Example of comparing a1, a2, and a3.

Left subtree will be true condition i.e. $a_i \leq a_j$

Right subtree will be false condition i.e. $a_i > a_j$

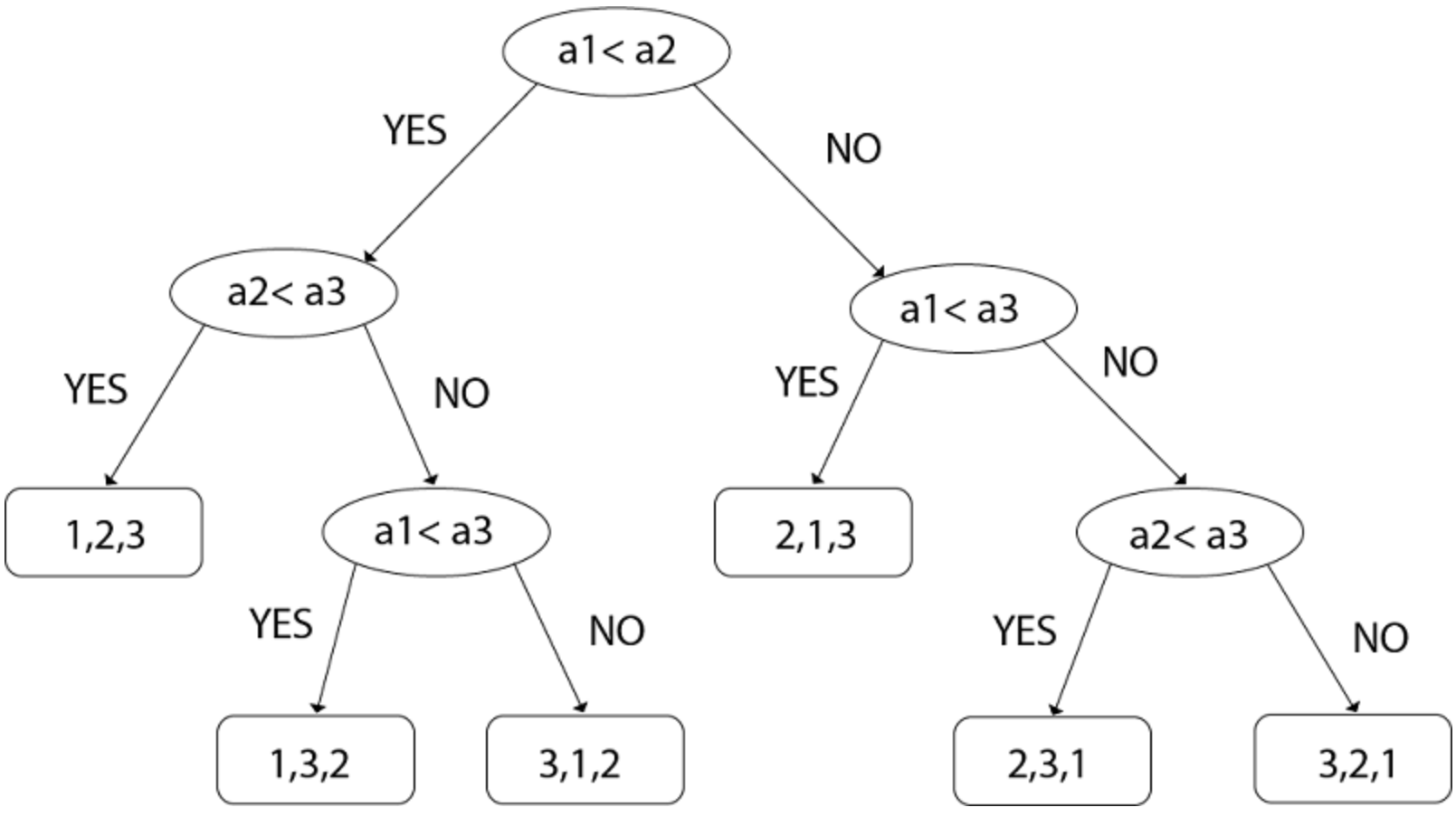


Fig: Decision Tree

So from above, we got

$N! \leq 2^n$

Taking Log both sides

$$\text{Log } n! \leq h \log 2$$

$$h \log 2 \geq \log n!$$

$$h \geq \log 2\ n!$$

$$h \geq \log 2\ [1, 2, 3 \ldots\ldots n]$$

$$h \geq \log_2 1 + \log_2 2 + \log_2 3 + \cdots \log_2 n$$

$$h \geq \sum_{i=1}^{n} \log_2 i$$

$$h \geq \int_i^n \log_2 i - 1\ d^i$$

$$h \geq \log_2 i \,.\, x^\circ \int_1^n - \int_1^n \frac{1}{i} xi\ di$$

$$h \geq n\log_2 n - \int_1^n 1\ di$$

$$h \geq n\log_2 n - i \int_1^n$$

$$h \geq n \log_2 n - n + 1$$

Ignoring the Constant terms

$$h \geq n\log_2 n$$

h=π n (log n)

Comparison tree for Binary Search:

Example: Suppose we have a list of items according to the following Position:

1. 1,2,3,4,5,6,7,8,9,10,11,12,13,14

Mid = $\left[\left(\frac{1+14}{2}\right)\right] = \frac{15}{2} = 7.5 = \mathbf{7}$

Note: Choose the greatest integer

1, 2, 3, 4, 5, 6	8, 9, 10, 11, 12, 13, 14
Mid=$\left(\frac{1+6}{2}\right)$=**3**	Mid =$\left(\frac{8+14}{2}\right)$ = **11**

1, 2	4, 5, 6	8, 9, 10	12, 13, 14
Mid=$\left(\frac{1+2}{2}\right)$=**1**	Mid=$\left(\frac{4+6}{2}\right)$=**5**	Mid= $\left(\frac{8+10}{2}\right)$=**9**	Mid=$\left(\frac{12+14}{2}\right)$=**13**

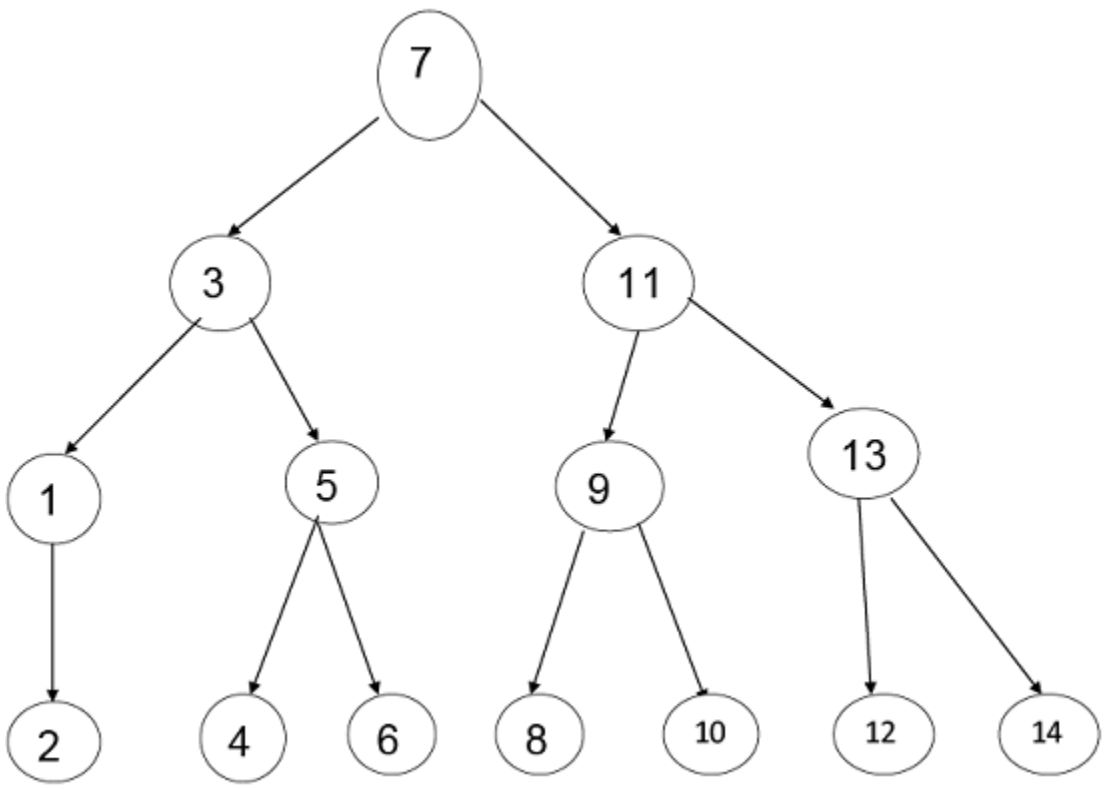
And the last midpoint is:

1. 2, 4, 6, 8, 10, 12, 14

Thus, we will consider all the midpoints and we will make a tree of it by having stepwise midpoints.

The Bold letters are Mid-Points Here

According to Mid-Point, the tree will be:



Step1: Maximum number of nodes up to k level of the internal node is 2^k-1

For Example

```
 2^k-1
 2^3-1= 8-1=7
Where k = level=3
```

Step2: Maximum number of internal nodes in the comparisons tree is n!

> ***Note: Here Internal Nodes are Leaves.***

Step3: From Condition1 & Condition 2 we get

```
N! ≤ 2^k-1
 14 < 15
 Where N = Nodes
```

Step4: Now, $n+1 \leq 2^k$

Here, Internal Nodes will always be less than 2^k in the Binary Search.

Step5:

$n+1 <= 2^k$

$\text{Log}\ (n+1) = k \log 2$

$k >= \frac{\log(n+1)}{\log 2}$

$k >= \log_2(n+1)$

Step6:

1. T (n) = k

Step7:

$T\ (n) >= \log_2(n+1)$

Here, the minimum number of Comparisons to perform a task of the search of n terms using Binary Search

---

2. Oracle and adversary argument:

Another technique for obtaining lower bounds consists of making use of an "oracle."

Given some model of estimation such as comparison trees, the oracle tells us the outcome of each comparison.

In order to derive a good lower bound, the oracle efforts it's finest to cause the algorithm to work as hard as it might.

It does this by deciding as the outcome of the next analysis, the result which matters the most work to be needed to determine the final answer.

And by keeping step of the work that is finished, a worst-case lower bound for the problem can be derived.

Example: (Merging Problem) given the sets A (1: m) and B (1: n), where the information in A and in B are sorted. Consider lower bounds for algorithms combining these two sets to give an individual sorted set.

Consider that all of the m+n elements are specific and A (1) < A (2) < ....< A (m) and B (1) < B (2) < ....< B (n).

Elementary combinatory tells us that there are C ((m+n), n)) ways that the A's and B's may merge together while still preserving the ordering within A and B.

Thus, if we need comparison trees as our model for combining algorithms, then there will be C ((m+n), n)) external nodes and therefore at least log C ((m+n), m) comparisons are needed by any comparison-based merging algorithm.

If we let MERGE (m, n) be the minimum number of comparisons used to merge m items with n items then we have the inequality

1. Log C ((m+n), m) MERGE (m, n) m+n-1.

The upper bound and lower bound can get promptly far apart as m gets much smaller than n.

---

3. State Space Method:

1. State Space Method is a set of rules that show the possible states (n-tuples) that an algorithm can assume from a given state of a single comparison.

2. Once the state transitions are given, it is possible to derive lower bounds by arguing that the finished state cannot be reached using any fewer transitions.

3. Given n distinct items, find winner and loser.

4. Aim: When state changed count it that is the aim of State Space Method.

5. In this approach, we will count the number of comparison by counting the number of changes in state.

6. Analysis of the problem to find out the smallest and biggest items by using the state space method.

7. State: It is a collection of attributes.

8. Through this we sort out two types of Problems:

1. Find the largest & smallest element from an array of elements.
2. To find out largest & second largest elements from an array of an element.

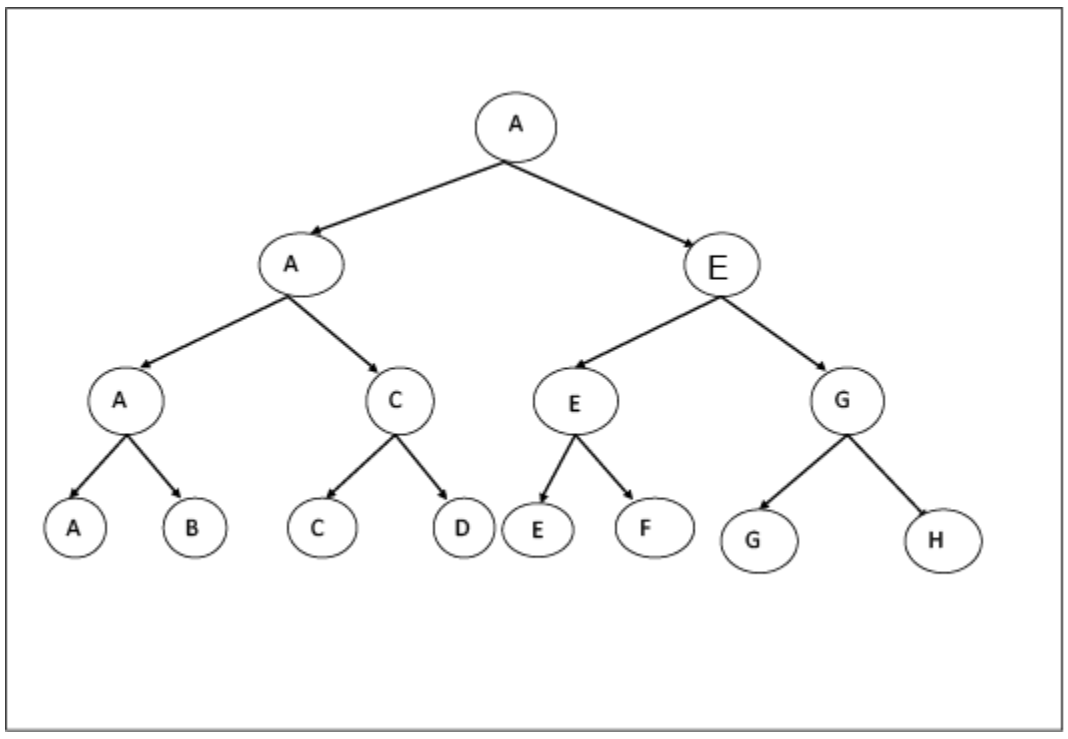
9. For the largest item, we need 7 comparisons and what will be the second largest item?

```
Now we count those teams who lose the match with team A
    Teams are: B, D, and E
So the total no of comparisons are: 7
Let n is the total number of items, then
    Comparisons = n-1 (to find the biggest item)
No of Comparisons to find out the 2nd biggest item = log2n-1
```

10. In this **no of comparisons** are equal to the **number of changes of states** during the execution of the algorithm.

Example: State (A, B, C, D)

A. No of items that can never be compared.
B. No of items that win but never lost (ultimate winner).
C. No of items that lost but never win (ultimate loser)
D. No of items who sometimes lost & sometimes win.



Firstly, A, B compare out or match between them. A wins and in C, D.C wins and so on. We can assume that B wins and so on. We can assume that B wins in place of A it can be anything depending on our self.

Phase-1:

A	B	C	D	
8	**0**	**0**	**0**	No match occurs; there are total 8 external nodes or 8 teams
6	**1**	**1**	**0**	Match between A & B,6 left in A state as only one match is done between A & B.1 add to B and 1 add to C .
4	**2**	**2**	**0**	C & D match out 4 left out 1 for C wins and 1 for D's lost and 0 for there is no team that sometimes wins or lost.
2	**3**	**3**	**0**	Match between E & F
0	**4**	**4**	**0**	Match between E & F and Match between G & H are compare out.

In Phase-1 there are 4 states
If the team is 8 then 4 states
As if n team the n/2 states.

Phase-2

A	B	C	D
0	4	4	0

Started at point where Phase 1 ends. Initial States for Phase-2

0	3	4	1

A & C both wins from lower level. But now A wins and C lost. But, C wins from lower level so C will be counted as sometimes wins & sometimes lost.

0	2	4	2

E & G are match E wins but G lost but G wins from lower level. It will be counted as sometimes wins & sometimes lost.

0	1	4	3

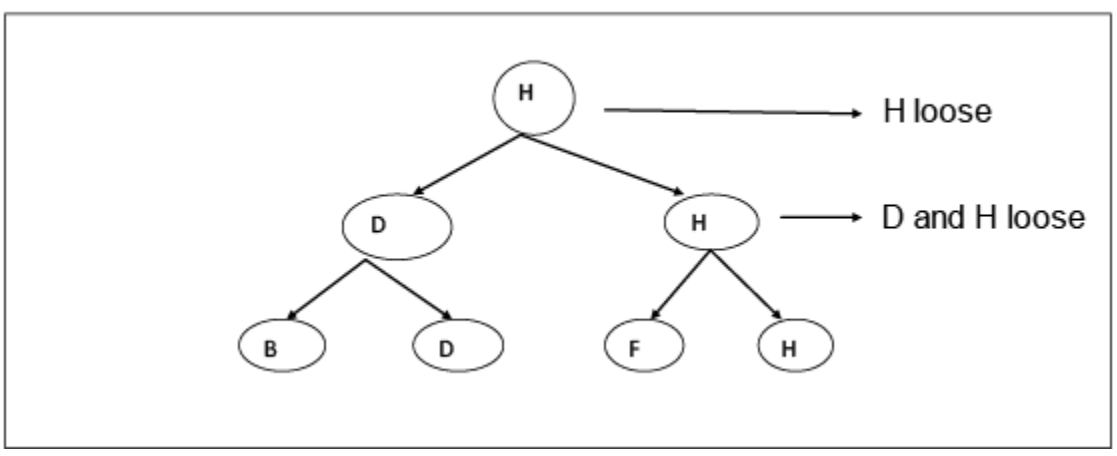
A & C both wins at Phase-1 but now A wins and C lost. So, C wins before and last now A will come in state- D

4 is Constant in C-State as B, D, F, H are lost teams that never win.

Thus there are 3 states in Phase-2,
If n is 8 then states are 3
If n teams that are $\left(\frac{n}{2}-1\right)$ states.

Phase-3: This is a Phase in which teams which come under C-State are considered and there will be matches between them to find out the team which is never winning at all.

In this Structure, we are going to move upward for denoting who is not winning after the match.



Here H is the team which is never winning at all. By this, we fulfill our second aim to.

A	B	C	D

0 1 4 3

Here by match out of B & D .We find that B wins and D looses But B at lower level looses.∴ It will be counted as sometimes wins and lost and it will be removed from lost team.∴ 4 will turn to 3 from lost team.

0 1 3 4

After match out F & H, F wins and H looses but F looses at lower level ∴F will be counted as sometimes wins & lost also removed from lost team thus 2 turn to 1 from lost team.

0 1 2 5

Match between F & H

0 1 1 6

After match out of D & H.H wins and D lost but H looses from lower level ∴it will be counted as sometimes wins & lost also removed from lost team thus 2 turn to 1 from lost team.

Thus there are 3 states in Phase -3

If n teams that are $\left(\frac{n}{2}-1\right)$ states

Note: the total of all states value is always equal to 'n'.

Thus, by adding all phase's states we will get:-

Phase1 + Phase 2 + Phase 3

$$\frac{n}{2}+\left(\frac{n}{2}-1\right)+\left(\frac{n}{2}\right)-1$$

$$\frac{n}{2}+\frac{n}{2}-1+\frac{n}{2}-1$$

$$\frac{n+n-2+n-2=}{2}\quad\frac{3n-4}{2}$$

$$\frac{3n-2}{2}$$

Suppose we have 8 teams then $\left[3\left(\frac{8}{2}\right)-2\right]$=10 states are there (as minimum) to find out which one is never wins team.

Thus, Equation is:

$$\mathbf{T(n)=}\frac{3n-2}{2}$$

Lower bound (L (n)) is a property of the particular issue i.e. the sorting problem, matrix multiplication not of any particular algorithm solving that problem.

Lower bound theory says that no calculation can carry out the activity in less than that of (L (n)) times the units for arbitrary inputs i.e. that for every comparison based sorting algorithm must take at least **L (n)** time in the worst case.

L (n) is the base overall conceivable calculation which is greatest finished.

Trivial lower bounds are utilized to yield the bound best alternative is to count the number of elements in the problems input that must be prepared and the number of output items that need to be produced.

The lower bound theory is the method that has been utilized to establish the given algorithm in the most efficient way which is possible. This is done by discovering a function **g (n)** that is a lower bound on the time that any algorithm must take to solve the given problem. Now if we have an algorithm whose computing time is the same order as **g (n)** , then we know that asymptotically we cannot do better.

If **f (n)** is the time for some algorithm, then we write **f (n) = Ω (g (n))** to mean that **g (n)** is the **lower bound of f (n)** . This equation can be formally written, if there exists positive constants **c** and **n0** such that **|f (n)| >= c|g (n)|** for all **n > n0**. In

addition for developing lower bounds within the constant factor, we are more conscious of the fact to determine more exact bounds whenever this is possible.

Deriving good **lower bounds** is more challenging than arrange efficient algorithms. This happens because a lower bound states a fact about all possible algorithms for solving a problem. Generally, we cannot enumerate and analyze all these algorithms, so lower bound proofs are often hard to obtain.

Linear Time Sorting

We have sorting algorithms that can sort "n" numbers in O (n log n) time.

Merge Sort and Heap Sort achieve this upper bound in the worst case, and Quick Sort achieves this on Average Case.

Merge Sort, Quick Sort and Heap Sort algorithm share an interesting property: the sorted order they determined is based only on comparisons between the input elements. We call such a sorting algorithm **"Comparison Sort"**.

There is some algorithm that runs faster and takes linear time such as Counting Sort, Radix Sort, and Bucket Sort but they require the special assumption about the input sequence to sort.

Counting Sort and Radix Sort assumes that the input consists of an integer in a small range.

Bucket Sort assumes that a random process that distributes elements uniformly over the interval generates the input.

Counting Sort

It is a linear time sorting algorithm which works faster by not making a comparison. It assumes that the number to be sorted is in range 1 to k where k is small.

Basic idea is to determine the "rank" of each number in the final sorted array.

Counting Sort uses three arrays:

1. A [1, n] holds initial input.
2. B [1, n] holds sorted output.
3. C [1, k] is the array of integer. C [x] is the rank of x in A where x ∈ [1, k]

Firstly C [x] to be a number of elements of A [j] that is equal to x

- Initialize C to zero
- For each j from 1 to n increment C [A[j]] by 1

We set B[C [x]] =A[j]

If there are duplicates, we decrement C [i] after copying.

1. Counting Sort (array P, array Q, **int** k)
2. 1. For i ← 1 to k
3. 2. **do** C [i] ← 0 [θ(k) times]
4. 3. **for** j ← 1 to length [A]
5. 4. **do** C[A[j]] ← C [A [j]]+1 [θ(n) times]
6. 5. // C [i] now contain the number of elements equal to i
7. 6. **for** i ← 2 to k
8. 7. **do** C [i] ← C [i] + C[i-1] [θ(k) times]
9. 8. //C[i] now contain the number of elements ≤ i
10. 9. **for** j ← length [A] down to 1 [θ(n) times]
11. 10. **do** B[C[A[j] ← A [j]
12. 11. C[A[j] ← C[A[j]-1

Explanation:

Step1: for loop initialize the array R to 'o'. But there is a contradict in the first step initialize of loop variable 1 to k or 0 to k. As 0&1 are based on the minimum value comes in array A (input array). Basically, we start I with the value which is minimum in input array 'A'

For loops of steps **3 to 4** inspects each input element. If the value of an input element is 'i', we increment C [i]. Thus, after step 5, C [i] holds the number of input element equal to I for each integer i=0, 1, 2.....k

Step **6 to 8** for loop determines for each i=0, 1.....how many input elements are less than or equal to i

For loop of step **9 to 11** place each element A [j] into its correct sorted position in the output array B. for each A [j],the value C [A[j]] is the correct final position of A [j] in the output array B, since there are C [A[j]] element less than or equal to A [i].

Because element might not be distinct, so we decrement C[A[j] each time we place a value A [j] into array B decrement C[A[j] causes the next input element with a value equal to A [j], to go to the position immediately before A [j] in the output array.

Analysis of Running Time:

- For a loop of step 1 to 2 take θ(k) times
- For a loop of step 3 to 4 take θ(n) times
- For a loop of step 6 to 7 take θ(k) times
- For a loop of step 9 to 11 take θ(n) times

Overall time is **θ(k+n)** time.

Note:

1. Counting Sort has the important property that it is stable: numbers with the same value appears in the output array in the same order as they do in the input array.
2. Counting Sort is used as a subroutine in Radix Sort.

Example: Illustration the operation of Counting Sort in the array.

1. A= (7,1,3,1,2,4,5,7,2,4,3)

Solution:

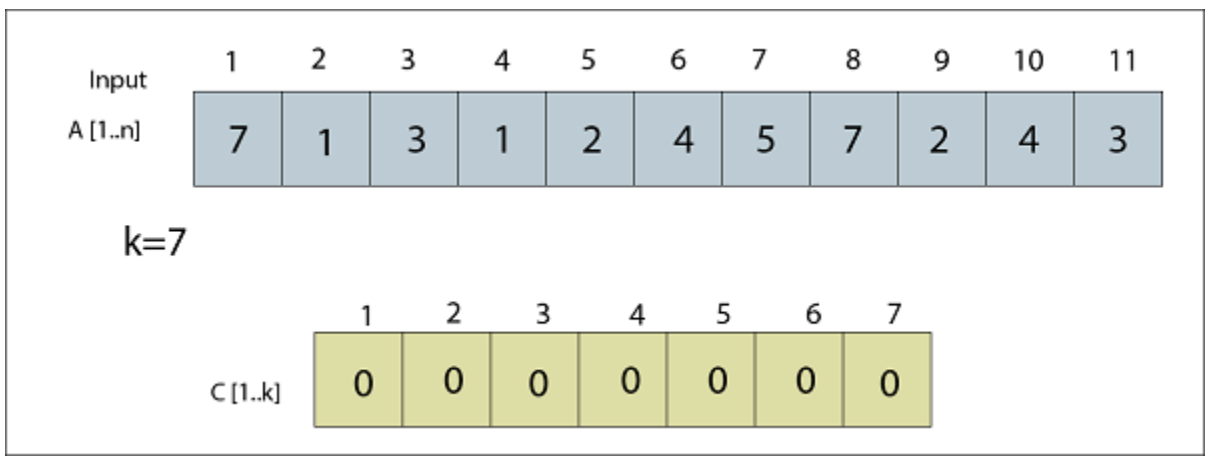


Fig: Initial A and C Arrays

1. For j=1 to 11
2. J=1, C [1, k] =

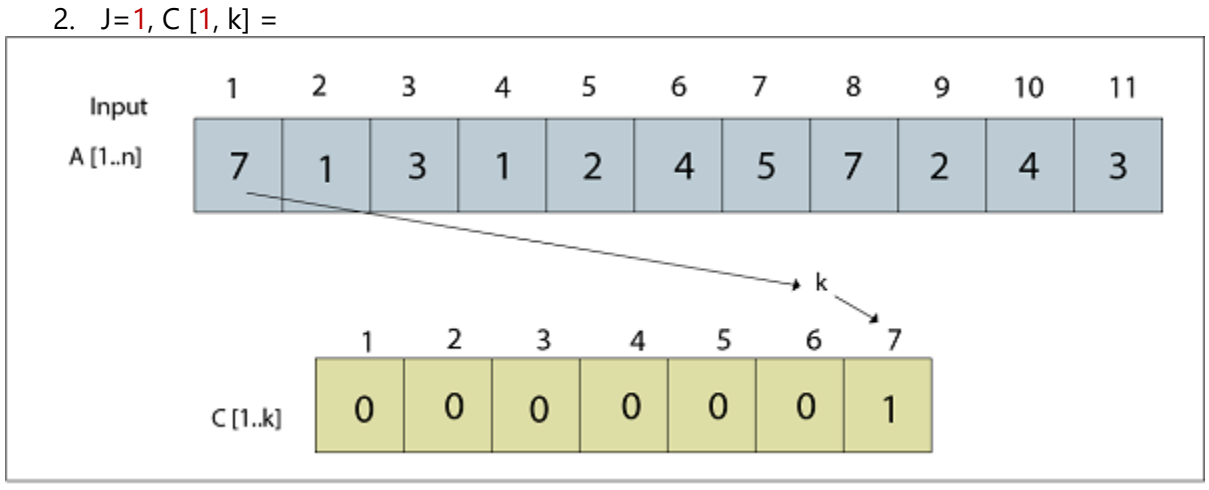


Fig: A [1] = 7 Processed

1. J=2, C [1, k] =

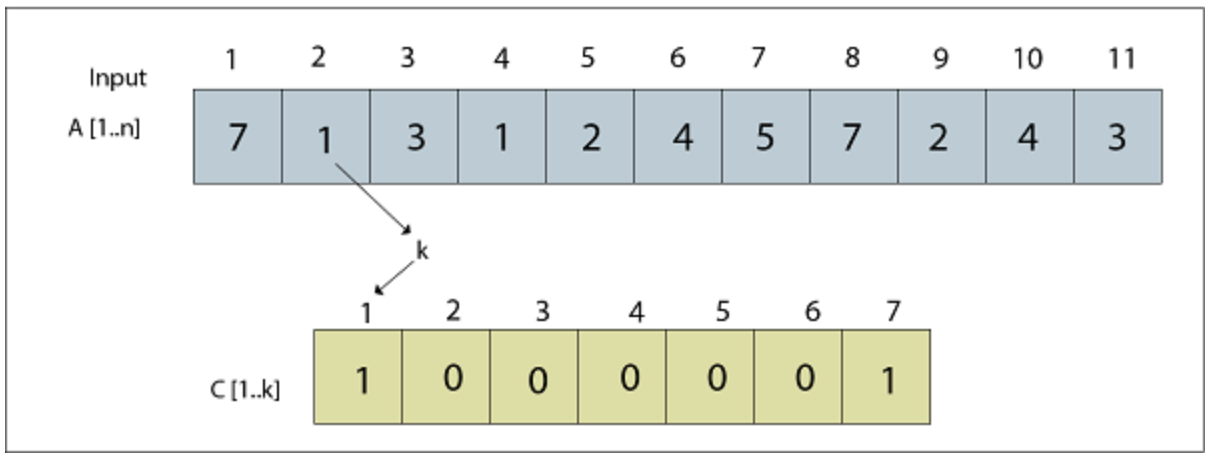


Fig: A [2] = 1 Processed

1. J=3, C [1, k]

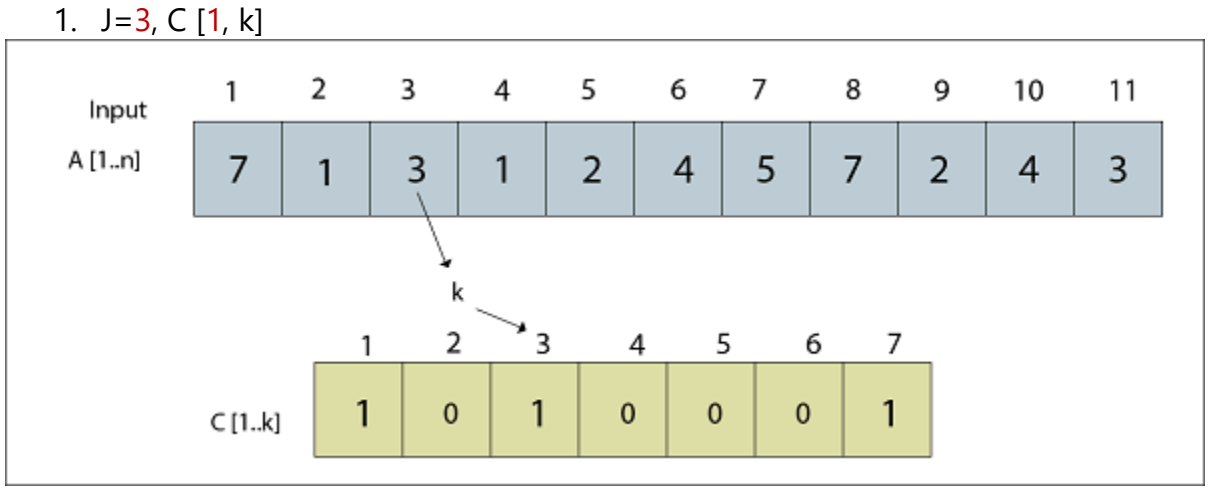


Fig: A [3] = 3 Processed

1. J=4, C [1, k]

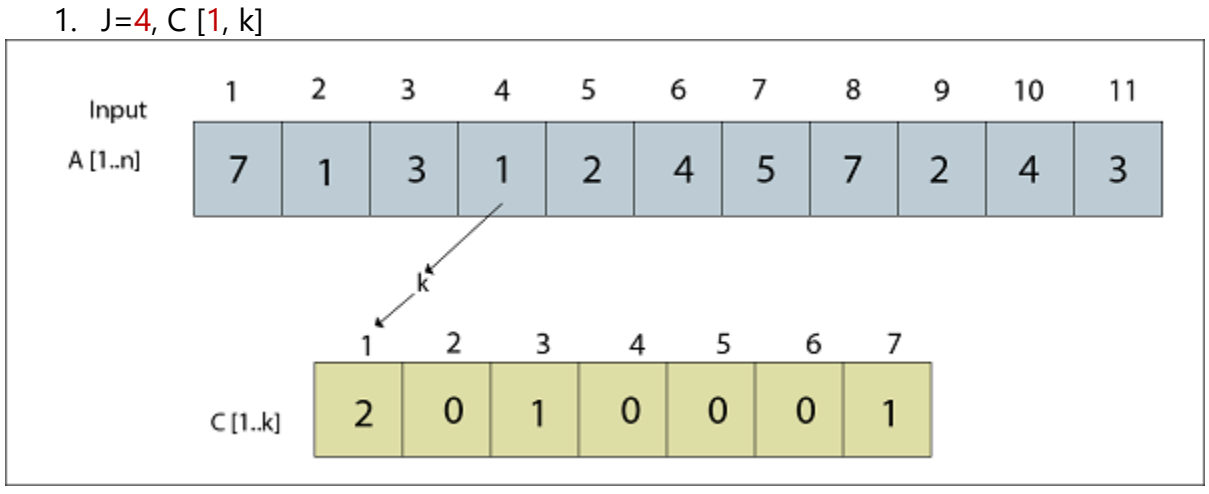


Fig: A [4] = 1 Processed

1. J=5, C [1, k]

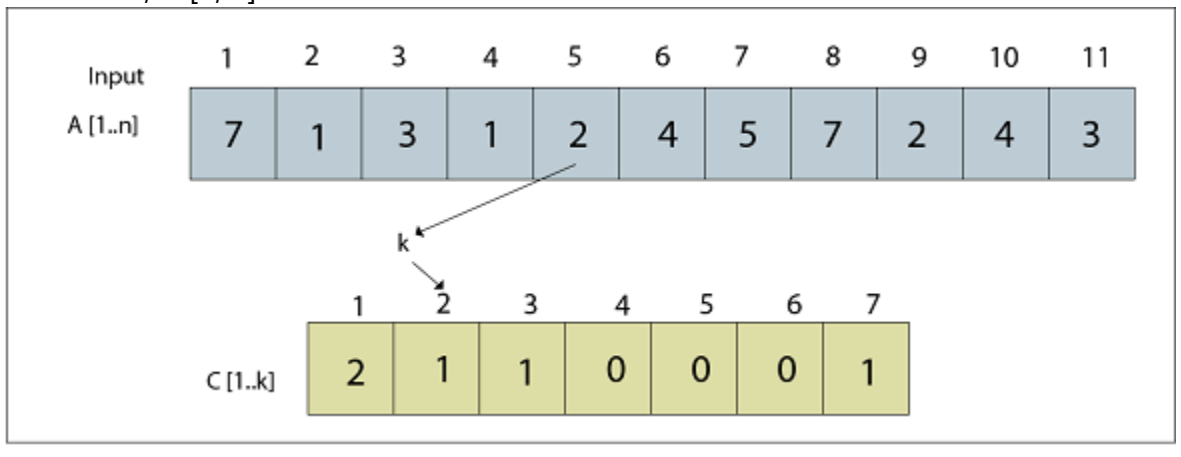


Fig: A [5] = 2 Processed

UPDATED C is:

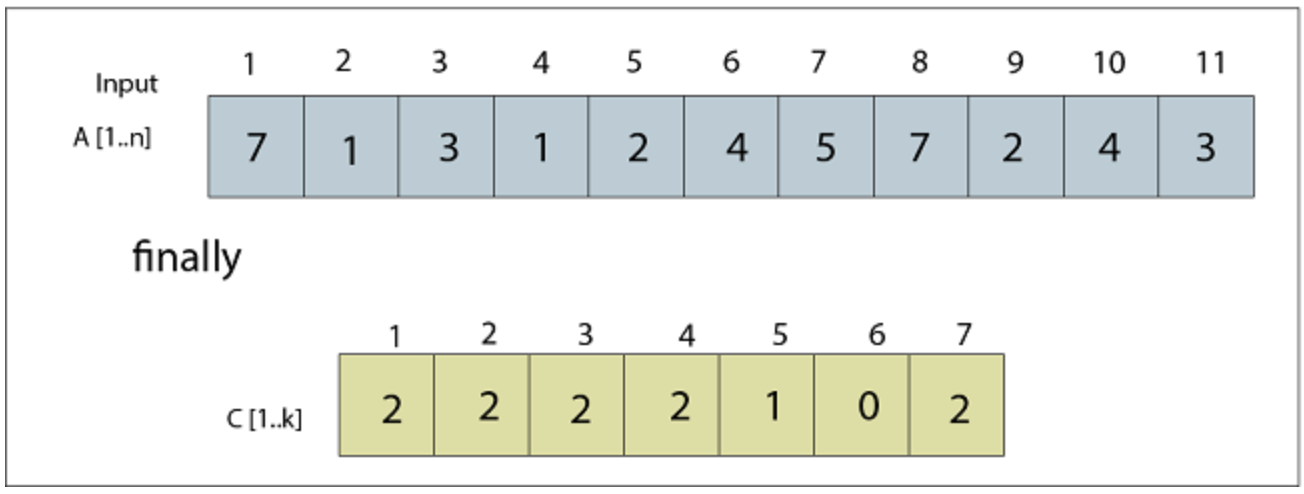


Fig: C now contains a count of elements of A

> **Note: here the item of 'A' one by one get scanned and will become a position in 'C' and how many times the item get accessed will be mentioned in an item in 'C' Array and it gets updated or counter increased by 1 if any item gets accessed again.**

Now, the for loop i= 2 to 7 will be executed having statement:

1. C [i] = C [i] + C [i-1]

By applying these conditions we will get C updated as i stated from 2 up to 7

```
C [2] = C [2] + C [1]
C [2] = 2 + 2
```
C [2] = 4

```
C [3] = C [3] + C [2]
C [3] = 2 + 4
```
C [3] = 6

```
C [4] = C [4] + C [3]
C [4] = 2 + 6
```
C [4] = 8

```
C [5] = C [5] + C [4]
C [5] = 1 +8
```
C [5] = 9

```
C [6] = C [6] + C [5]
C [6] = 0 + 9
```
C [6] = 9

```
C [7] = C [7] + C [6]
C [7] = 2 + 9
```
C [7] = 11

Thus the Updated C is:

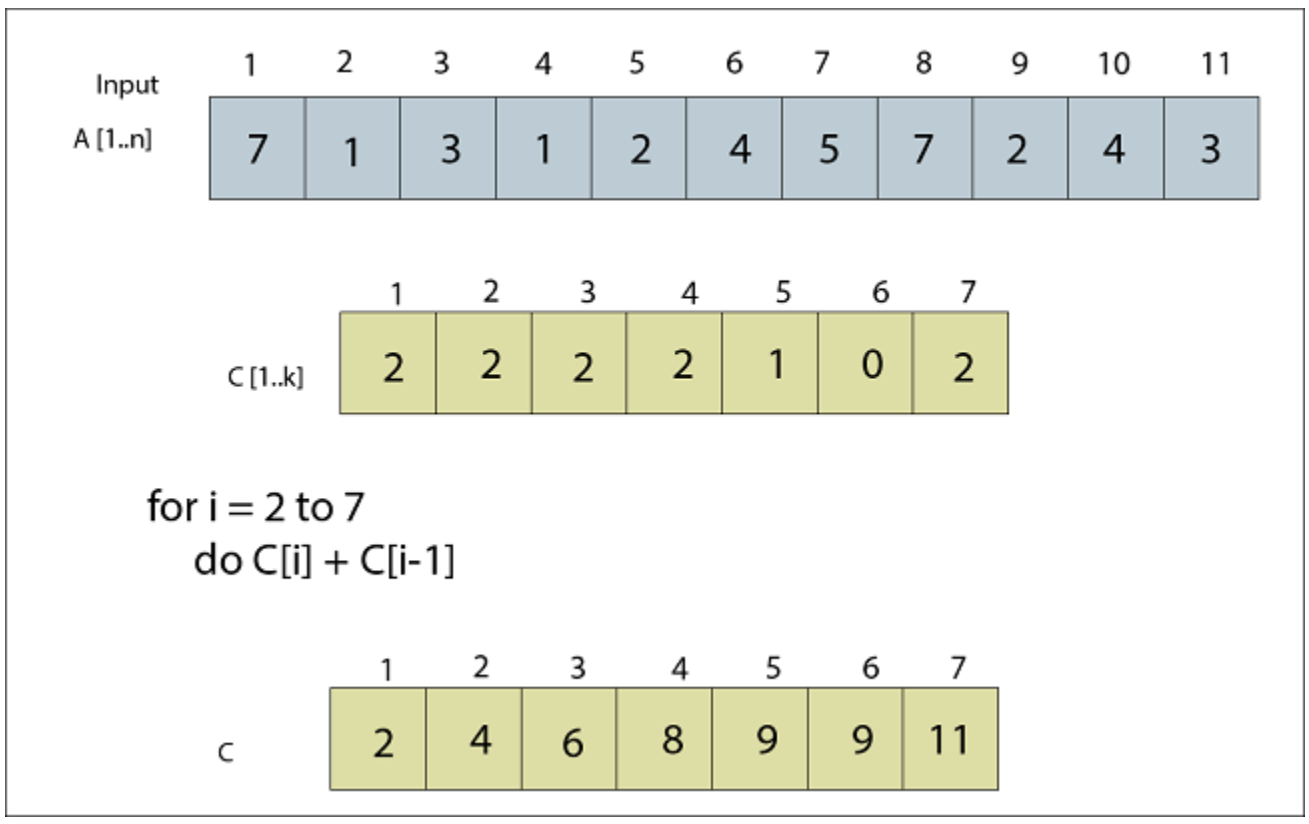


Fig: C set to rank each number of A

Now, we will find the new array B

	1	2	3	4	5	6	7	8	9	10	11
A =	7	1	3	1	2	4	5	7	2	4	3

	1	2	3	4	5	6	7	8	9	10	11
B=											

- B has the same size as A
- It will give sorted output.

	1	2	3	4	5	6	7
C=	2	4	6	8	9	9	11

C is an intermediate between A & B.

Now two Conditions will apply:

1. B[C[A[j] ← A [j]
2. C[A[j] ← C[A[j]-1

We decrease counter one by one by '1'
We start scanning of the element in A from the last position.
Element in A became a position in C

1. For j ← 11 to 1

Step 1

```
B [C [A [11]]] = A [11]      C [A [11] = C [A [11]-1
B [C [3] = 3                 C [3] = C [3] -1
```

B [6] = 3 **C [3] = 5**

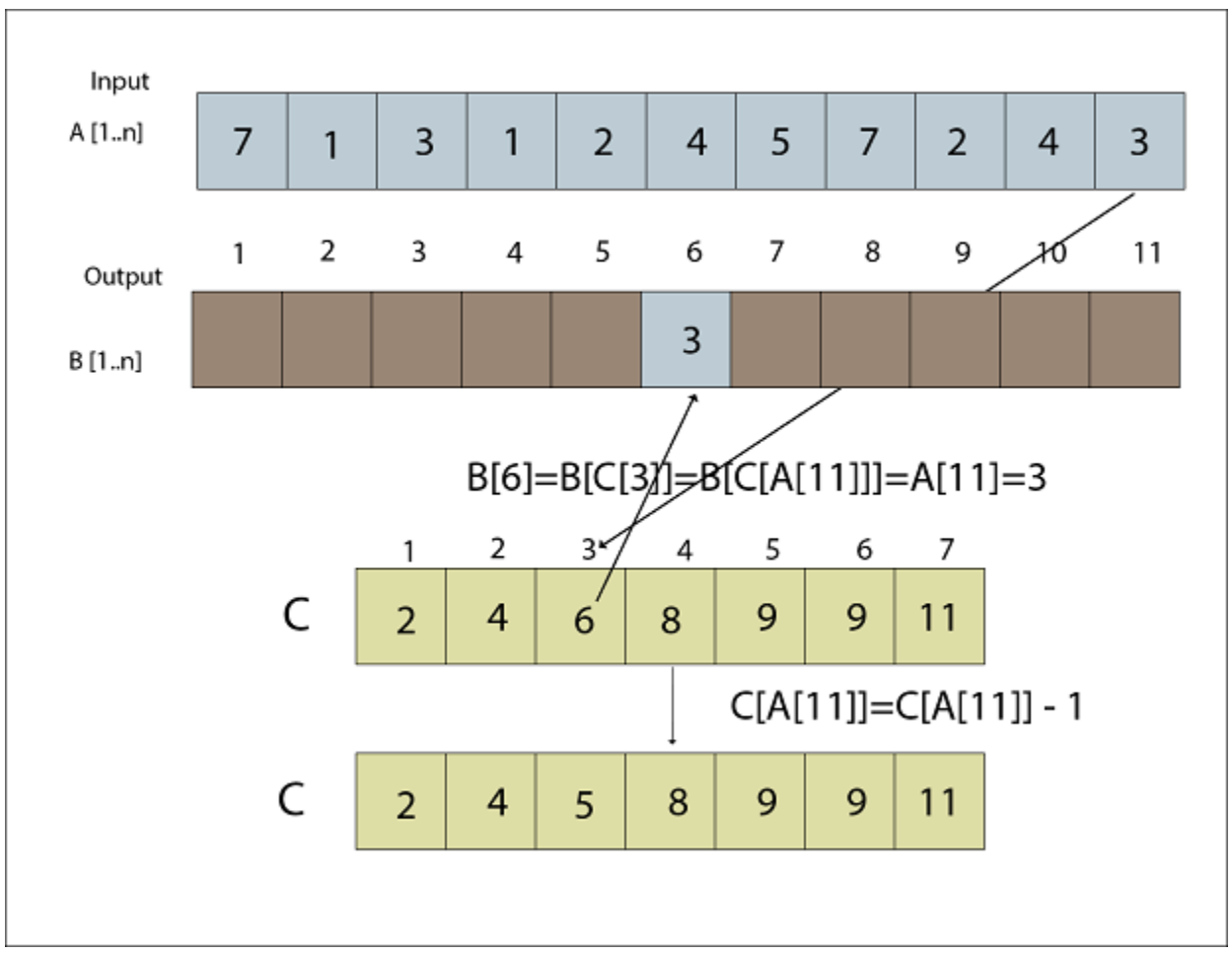


Fig: A [11] placed in Output array B

Step 2

```
B [C [A [10]]] = A [10]      C [A [10]] = C [A [10]]-1
B [C [4]] =4                 C [4] = C [4] -1
```

B [8] = 4 **C [4] = 7**

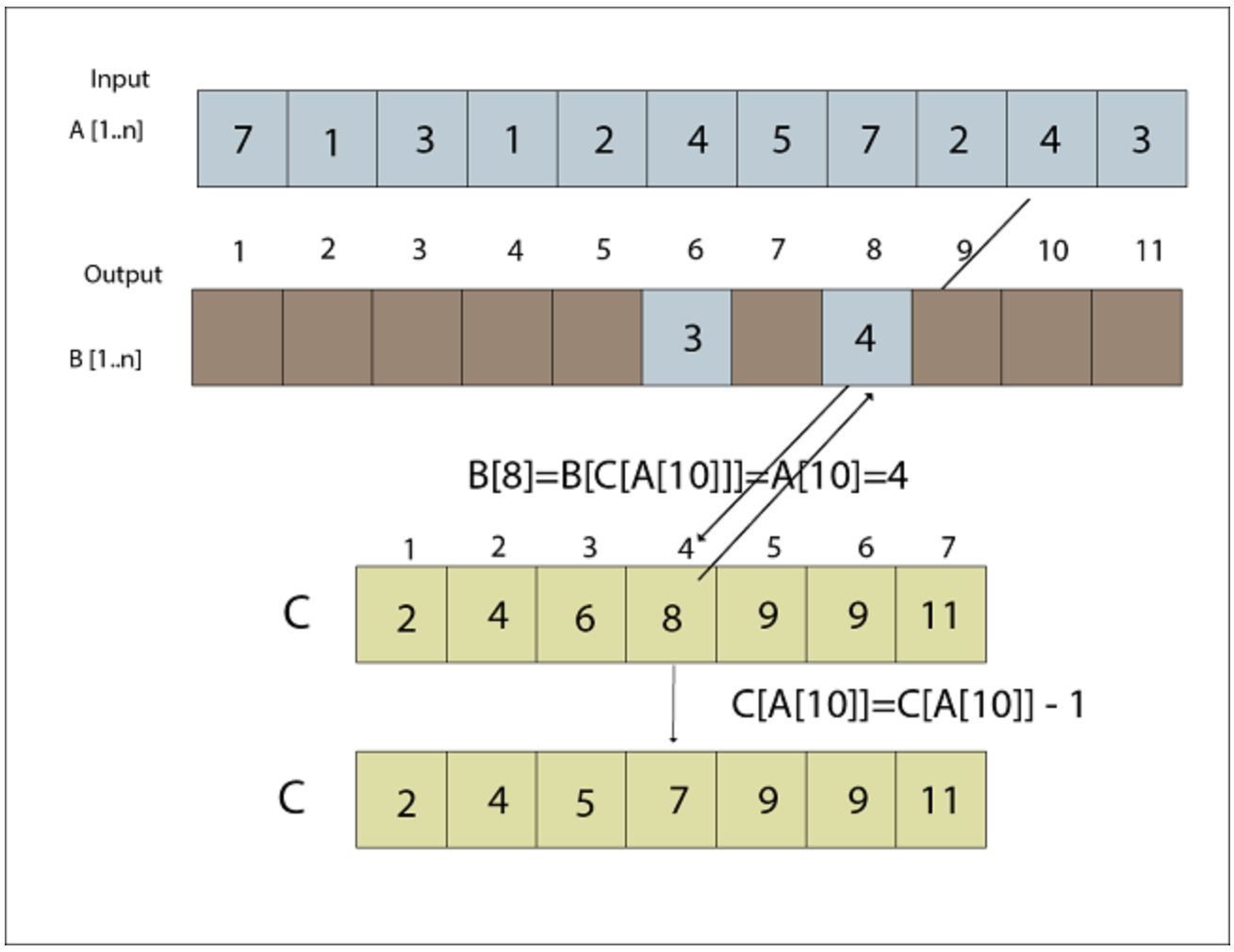


Fig: A [10] placed in Output array B

Step 3

```
B [C [A [9]] = A [9]          C [A [9] = C [A [9]]-1
B [C [2]] = A [2]             C [2] = C [2]-1
```

B [4] = 2 **C [2] = 3**

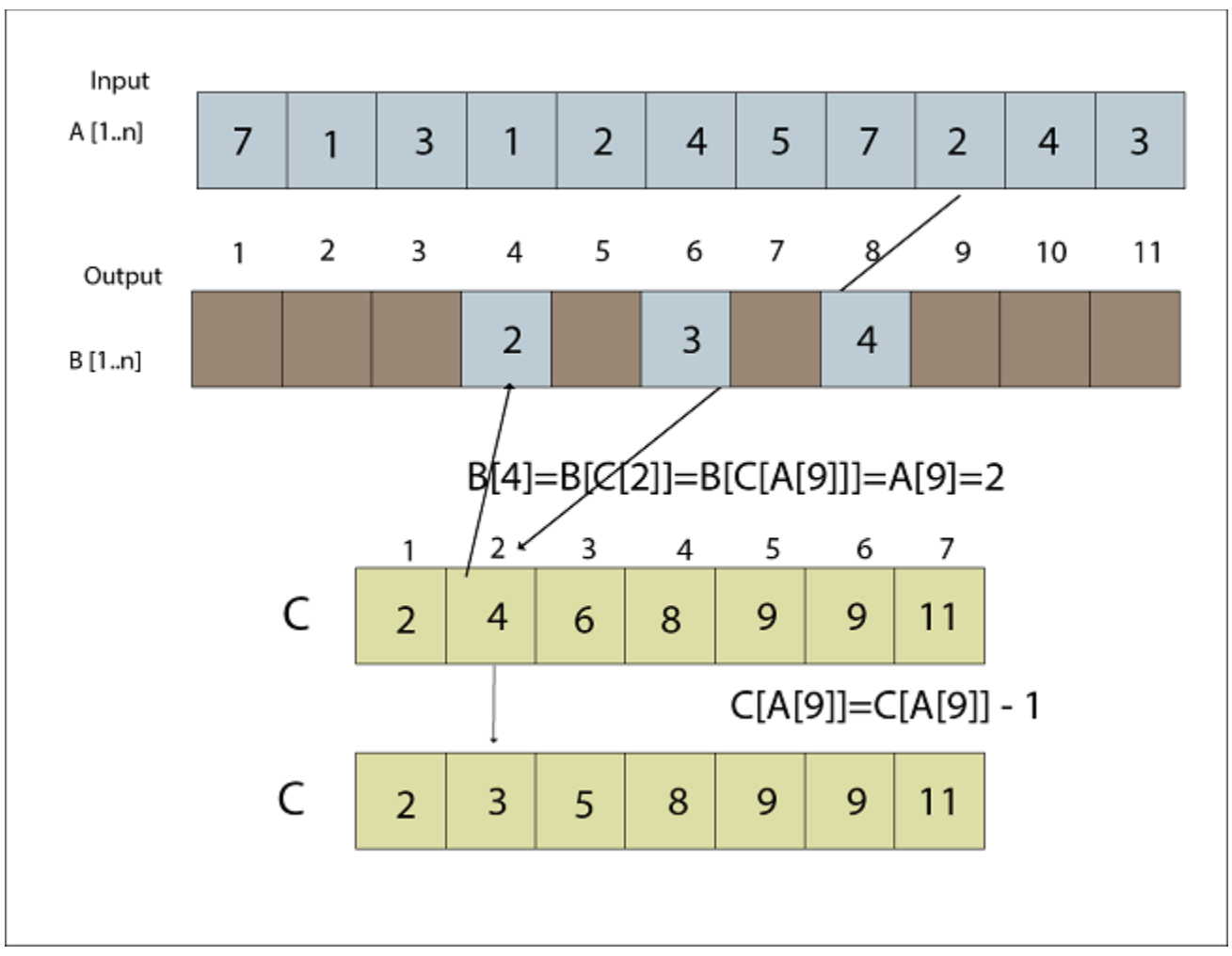


Fig: A [9] placed in Output array B

Step 4

```
B [C [A [8]]] = A [8]         C [A [8]] =C [A [8]] -1
B [C [7]] =7                  C [A [8]] = C [7]-1
```

B [11] =7 **C [7] = 10**

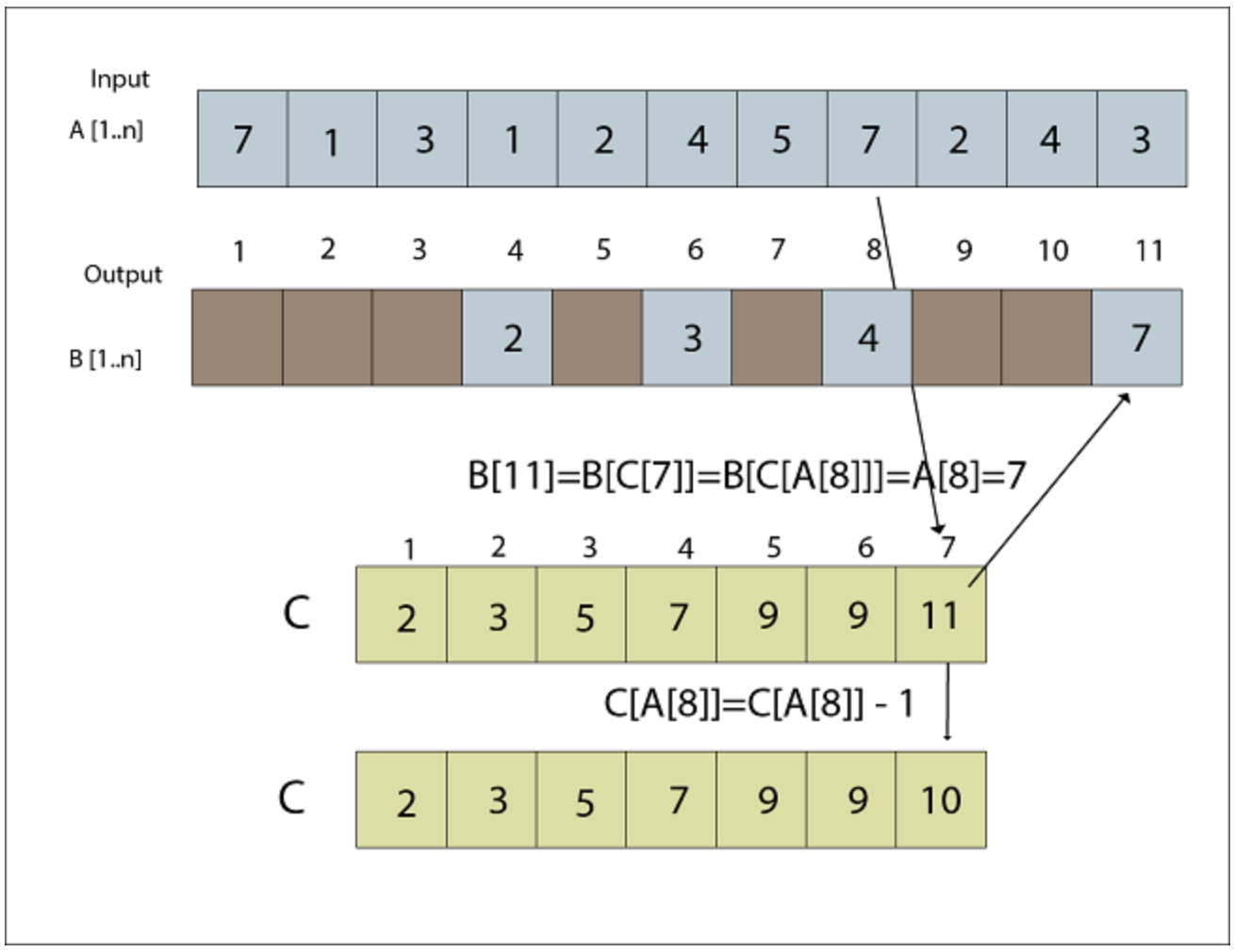


Fig: A [8] placed in Output array B

Step 5

```
B [C [A [7]]] = A [7]          C [A [7]] = C [A [7]] - 1
B [C [5]] = 5                  C [5] = C [5] - 1
```

B [9] = 5 **C [5] =8**

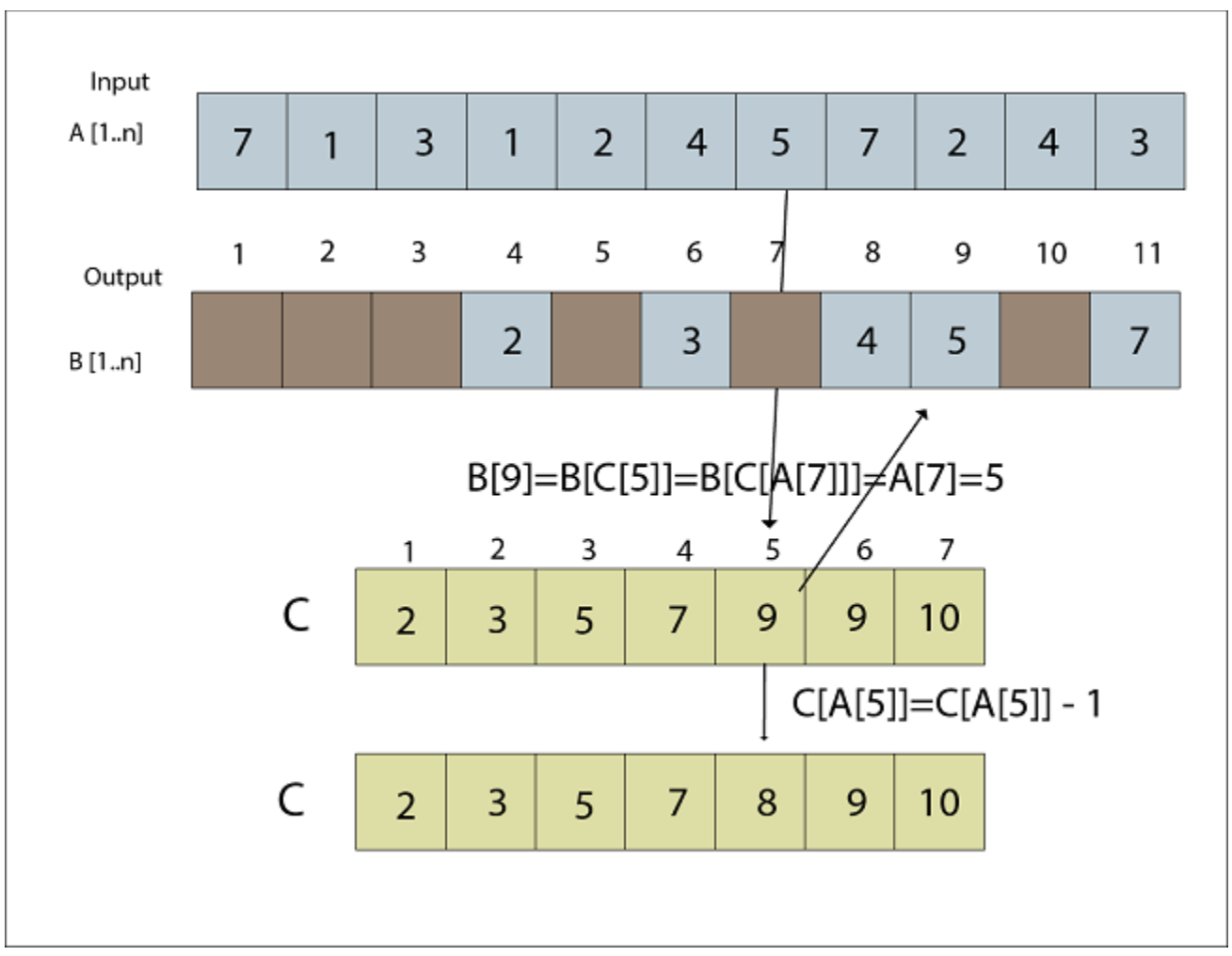


Fig: A [7] placed in Output array B

Step 6

```
B [C [A [6]]] = A [6]          C [A [6]] = C [A [6]] - 1
B [C [4]] = 4                  C [4] = C [4] - 1
```

B [7] = 4 **C [4] = 6**

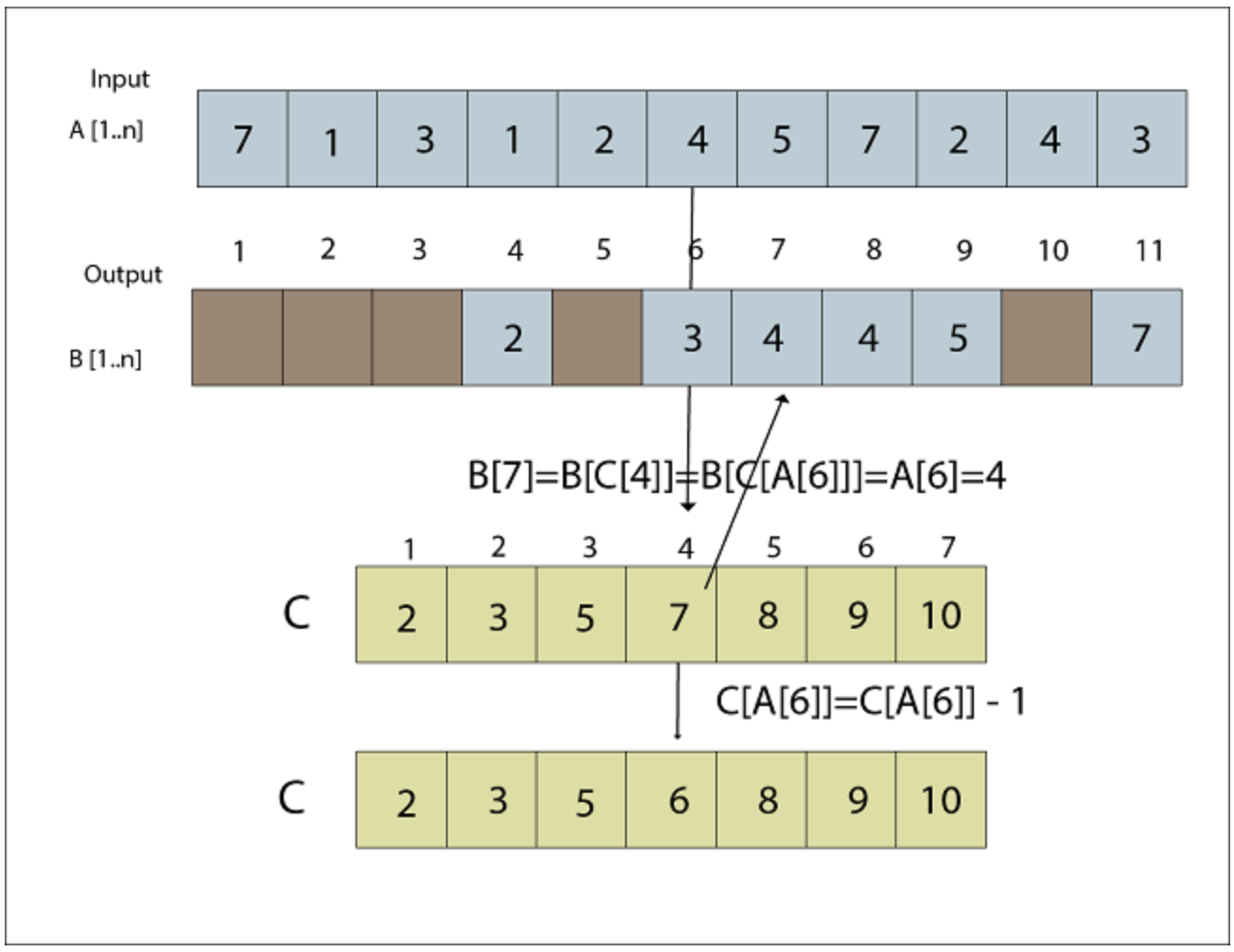


Fig: A [6] placed in Output array B

Step 7

```
B [C [A [5]]] = A [5]          C [A [5] = C [A [5]] -1
B [C [2] =2                    C [2] = C [2] - 1
```

B [3] = 2 **C [2] = 2**

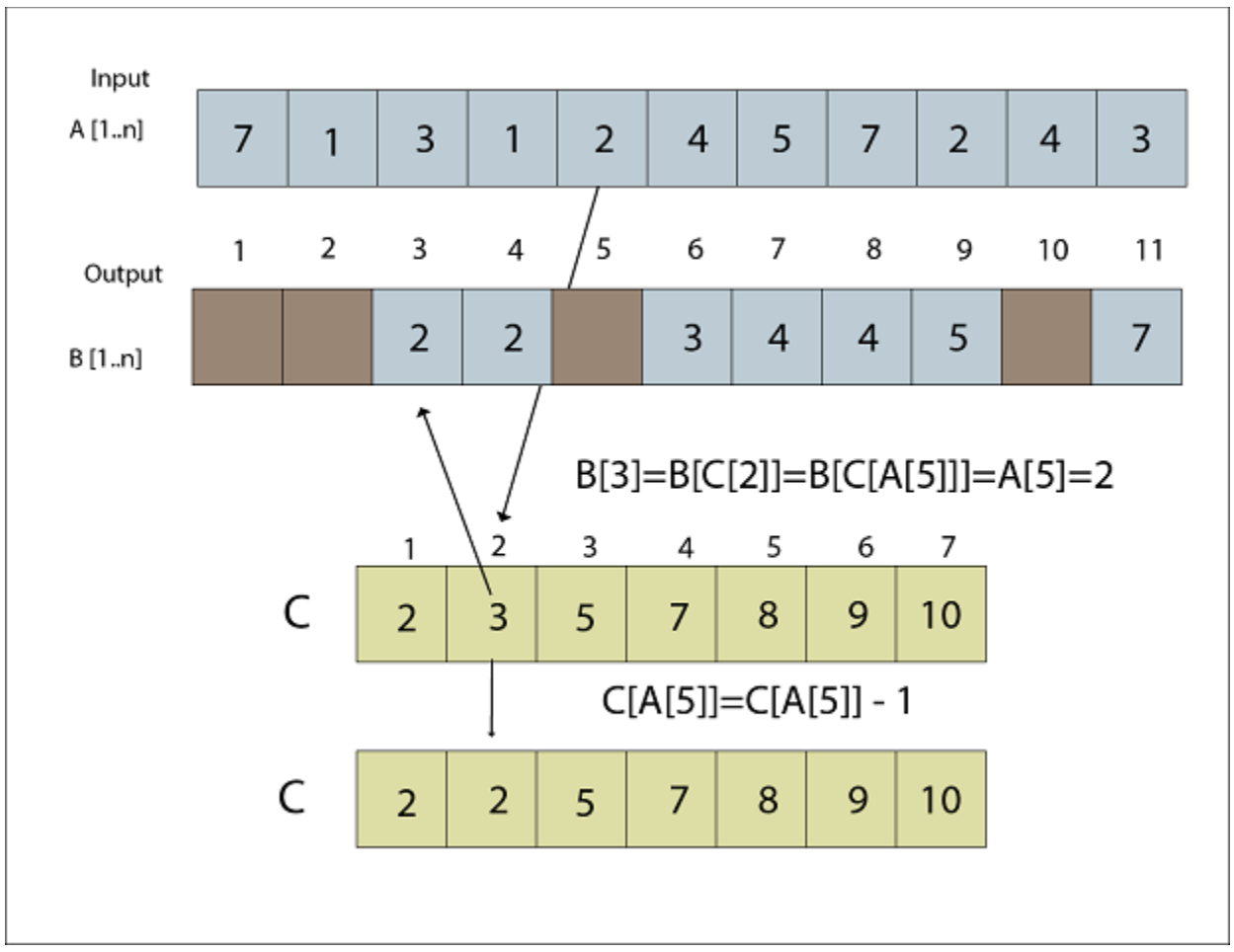


Fig: A [5] placed in Output array B

Step 8

```
B [C [A [4]]] = A [4]          C [A [4]] = C [A [4]] - 1
B [C [1] = 1                   C [1] = C [1] - 1
```

B [2] = 1 **C [1] = 1**

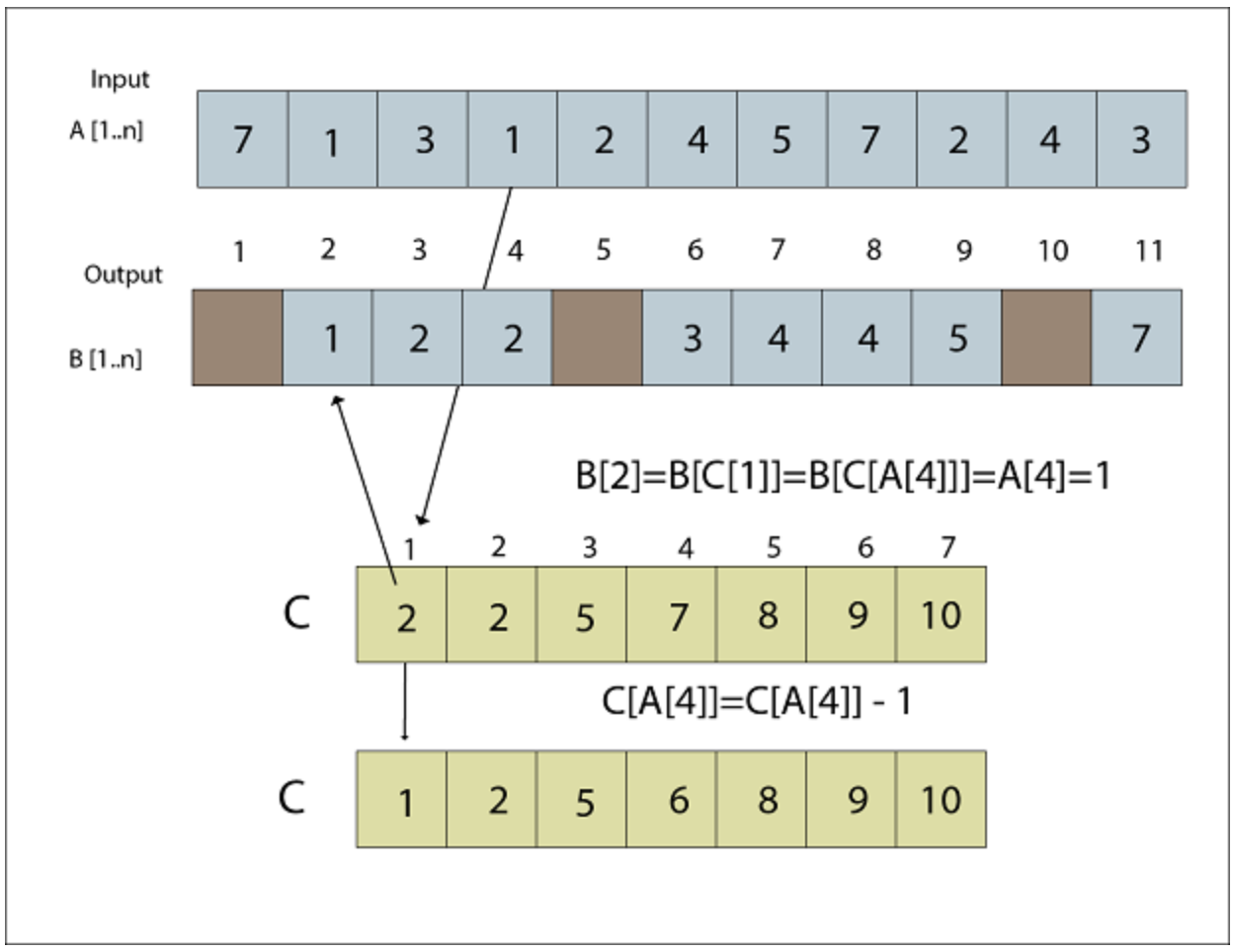


Fig: A [4] placed in Output array B

Step 9

```
B [C [A [3]]] = A [3]          C [A [3]] = C [A [3]] - 1
B [C [3] = 3                   C [3] = C [3] - 1
```

B [5] = 3 **C [3] = 4**

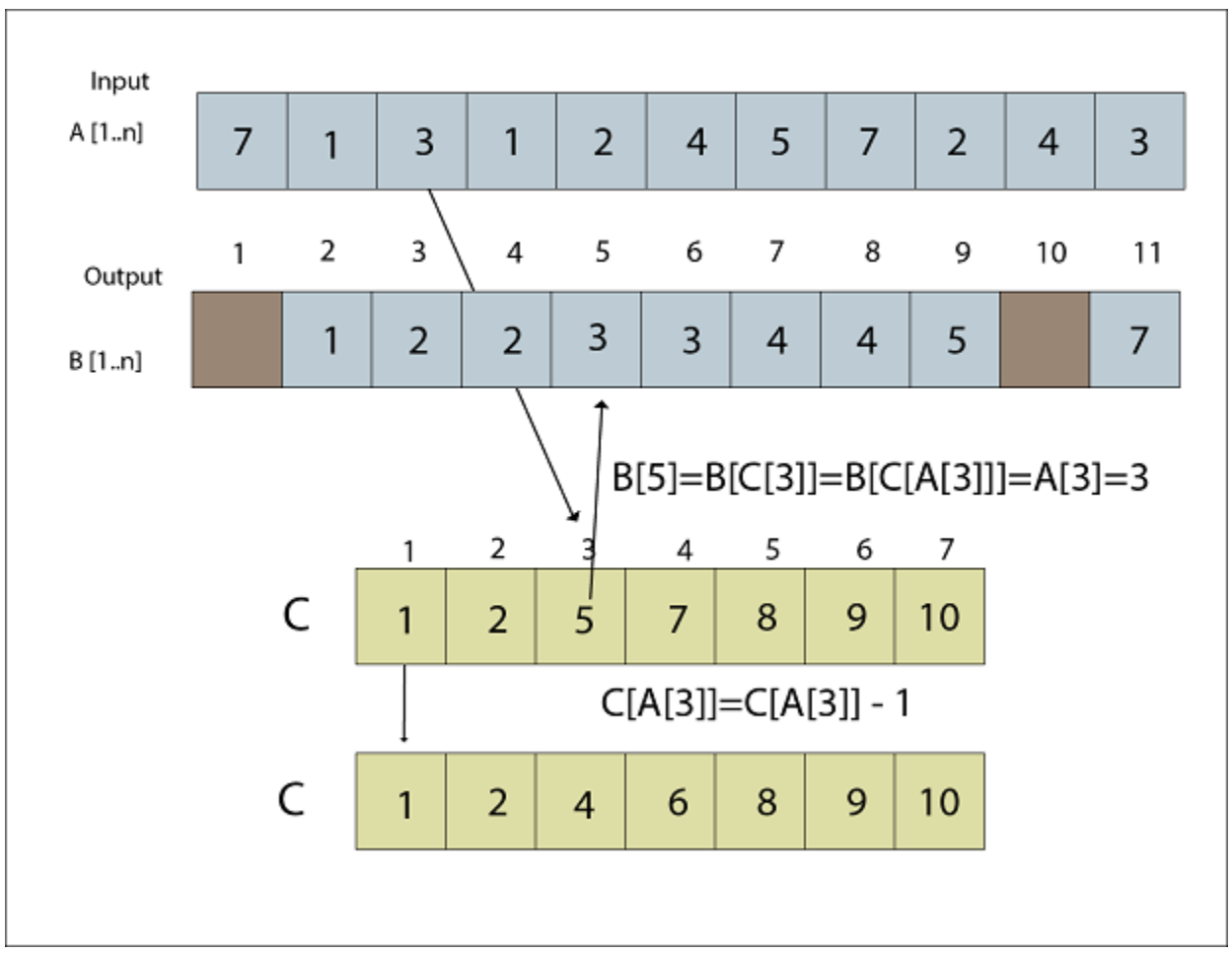


Fig: A [3] placed in Output array B

Step 10

```
B [C [A [2]]] = A [2]          C [A [2]] = C [A [2]] - 1
B [C [1]] = 1                  C [1] = C [1] - 1
```

B [1] = 1 **C [1] = 0**

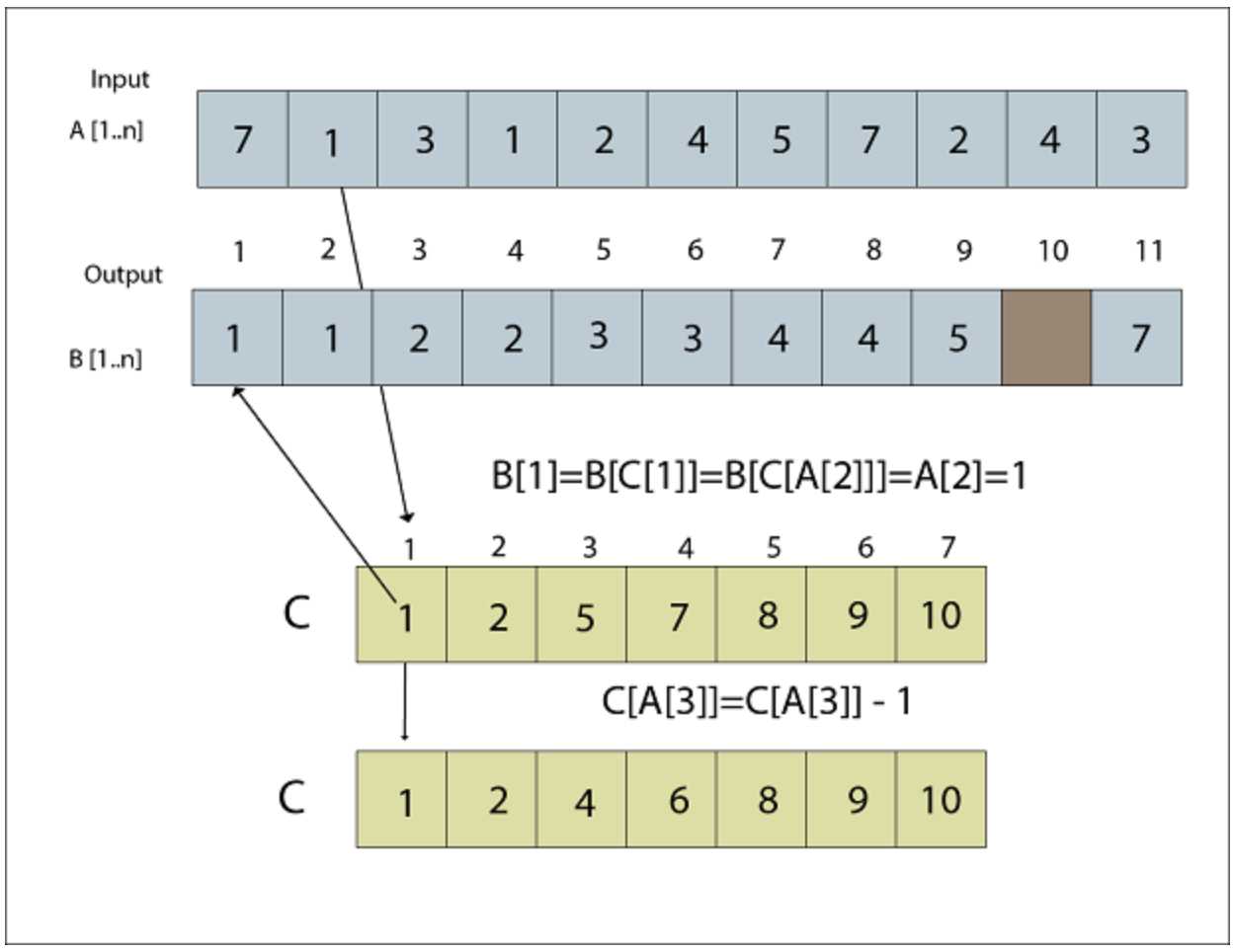


Fig: A [2] placed in Output array B

Step 11

```
B [C [A [1]]] = A [1]        C [A [1]] = C [A [1]] - 1
B [C [7]] = 7                C [7] = C [7] - 1
```

B [10] = 7 **C [7] = 9**

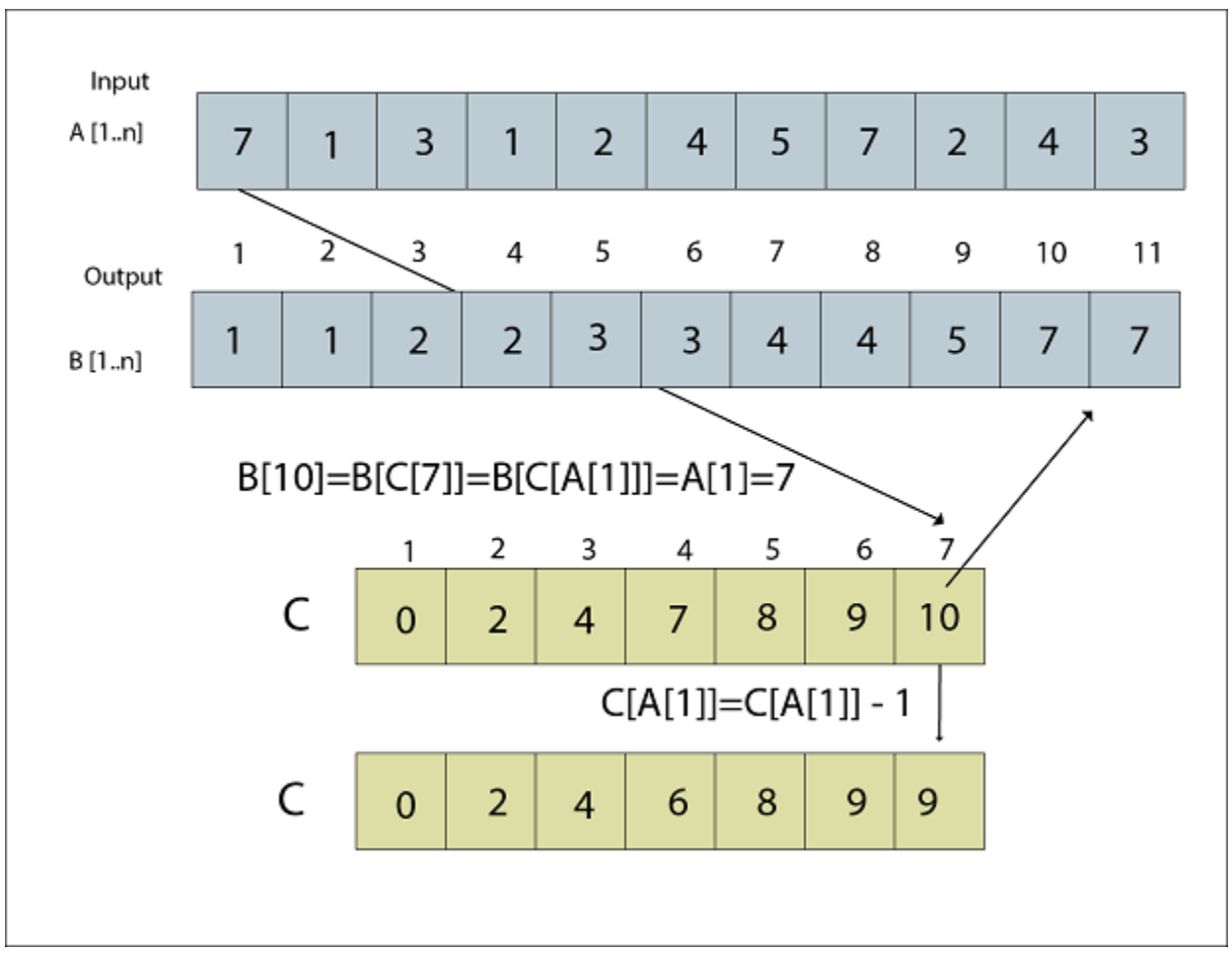


Fig: B now contains the final sorted data.

Bucket Sort

Bucket Sort runs in linear time on average. Like Counting Sort, bucket Sort is fast because it considers something about the input. Bucket Sort considers that the input is generated by a random process that distributes elements uniformly over the intervalμ=[0,1].

To sort n input numbers, Bucket Sort

1. Partition μ into n non-overlapping intervals called buckets.
2. Puts each input number into its buckets
3. Sort each bucket using a simple algorithm, e.g. Insertion Sort and then
4. Concatenates the sorted lists.

Bucket Sort considers that the input is an n element array A and that each element A [i] in the array satisfies 0≤A [i] <1. The code depends upon an auxiliary array B [0....n-1] of linked lists (buckets) and considers that there is a mechanism for maintaining such lists.

BUCKET-SORT (A)

```
1. n ← length [A]
2. for i ← 1 to n
3. do insert A [i] into list B [n A[i]]
4. for i ← 0 to n-1
5. do sort list B [i] with insertion sort.
6. Concatenate the lists B [0], B [1] ...B [n-1] together in order.
```

Example: Illustrate the operation of BUCKET-SORT on the array.

1. A = (0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 068)

Solution:

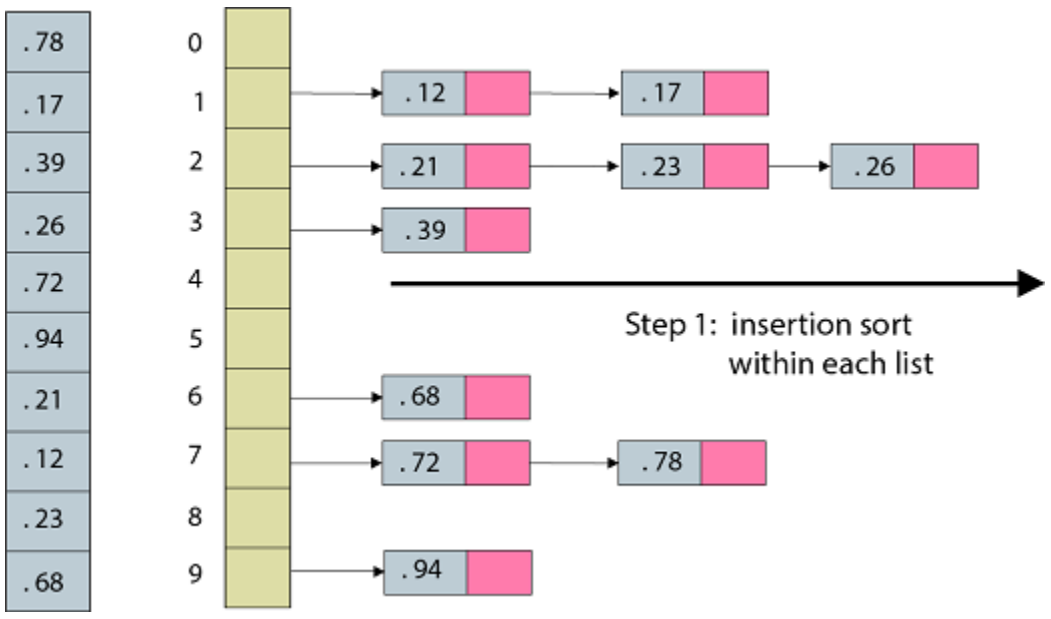


Fig: Bucket sort: step 1, placing keys in bins in sorted order

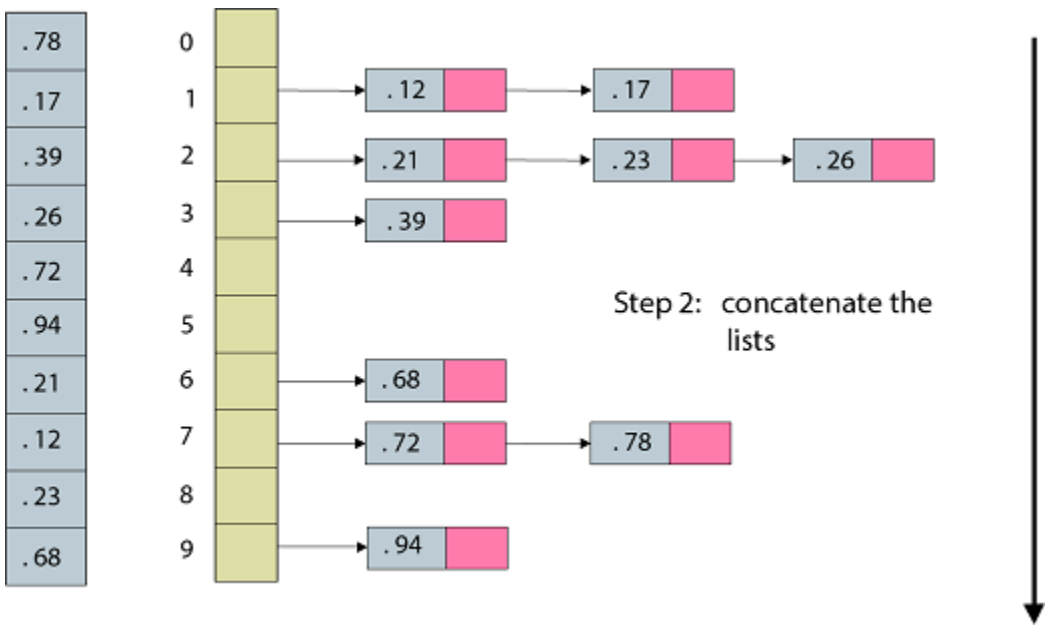


Fig: Bucket sort: step 2, concatenate the lists

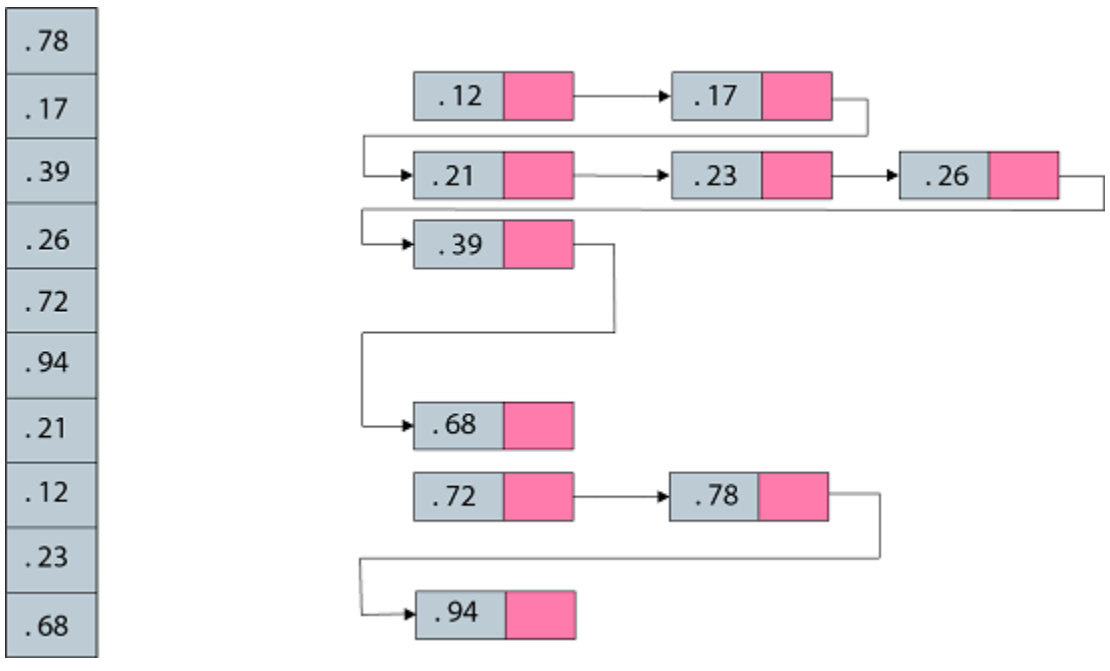


Fig: Bucket sort: the final sorted sequence

Next →← Prev

Radix Sort

Radix Sort is a Sorting algorithm that is useful when there is a constant'd' such that all keys are d digit numbers. To execute Radix Sort, for p =1 towards 'd' sort the numbers with respect to the Pth digits from the right using any linear time stable sort.

The Code for Radix Sort is straightforward. The following procedure assumes that each element in the n-element array A has d digits, where digit 1 is the lowest order digit and digit d is the highest-order digit.

Here is the algorithm that sorts A [1.n] where each number is d digits long.

RADIX-SORT (array A, int n, int d)

```
1 for i ← 1 to d
2 do stably sort A to sort array A on digit i
```

Example: The first Column is the input. The remaining Column shows the list after successive sorts on increasingly significant digit position. The vertical arrows indicate the digits position sorted on to produce each list from the previous one.

1. 576 49[4] 9[5]4 [1]76 176
2. 494 19[4] 5[7]6 [1]94 194
3. 194 95[4] 1[7]6 [2]78 278
4. 296 → 57[6] → 2[7]8 → [2]96 → 296
5. 278 29[6] 4[9]4 [4]94 494
6. 176 17[6] 1[9]4 [5]76 576
7. 954 27[8] 2[9]6 [9]54 954

Hashing

Hashing is the transformation of a string of character into a usually shorter fixed-length value or key that represents the original string.

Hashing is used to index and retrieve items in a database because it is faster to find the item using the shortest hashed key than to find it using the original value. It is also used in many encryption algorithms.

A hash code is generated by using a key, which is a unique value.

Hashing is a technique in which given key field value is converted into the address of storage location of the record by applying the same operation on it.

The advantage of hashing is that allows the execution time of basic operation to remain constant even for the larger side.

Why we need Hashing?

Suppose we have 50 employees, and we have to give 4 digit key to each employee (as for security), and we want after entering a key, direct user map to a particular position where data is stored.

If we give the location number according to 4 digits, we will have to reserve 0000 to 9999 addresses because anybody can use anyone as a key. There is a lot of wastage.

In order to solve this problem, we use hashing which will produce a smaller value of the index of the hash table corresponding to the key of the user.

Universal Hashing

Let H be a finite collection of hash functions that map a given universe U of keys into the range {0, 1..... m-1}. Such a collection is said to be universal if for each pair of distinct keys k,l∈U, the number of hash functions h∈ H for which h(k)= h(l) is at most |H|/m. In other words, with a hash function randomly chosen from H, the chance of a collision between distinct keys k and l is no more than the chance 1/m of a collision if h(k) and h(l)were randomly and independently chosen from the set {0,1,...m-1}.

Rehashing

If any stage the hash table becomes nearly full, the running time for the operations of will start taking too much time, insert operation may fail in such situation, the best possible solution is as follows:

1. Create a new hash table double in size.
2. Scan the original hash table, compute new hash value and insert into the new hash table.
3. Free the memory occupied by the original hash table.

Example: Consider inserting the keys 10, 22, 31,4,15,28,17,88 and 59 into a hash table of length m = 11 using open addressing with the primary hash function h' (k) = k mod m .Illustrate the result of inserting these keys using linear probing, using quadratic probing with c_1=1 and c_2=3, and using double hashing with $h_2(k)$ = 1 + (k mod (m-1)).

Solution: Using Linear Probing the final state of hash table would be:

0	22
1	88
2	/
3	/
4	4
5	15
6	28
7	17
8	59
9	31
10	10

Using Quadratic Probing with $c_1=1$, $c_2=3$, the final state of hash table would be h (k, i) = (h' (k) $+c_1*i+ c_2 *i^2$) mod m where m=11 and h' (k) = k mod m.

0	22
1	88
2	/
3	17
4	4
5	/
6	28
7	59
8	15
9	31
10	10

Using Double Hashing, the final state of the hash table would be:

0	22
1	/
2	59
3	17
4	4
5	15
6	28
7	88
8	/
9	31
10	10

Hash Tables

It is a collection of items which are stored in such a way as to make it easy to find them later.

Each position in the hash table is called slot, can hold an item and is named by an integer value starting at 0.

The mapping between an item and a slot where the item belongs in a hash table is called a Hash Function. A hash Function accepts a key and returns its hash coding, or hash value.

Assume we have a set of integers 54, 26, 93, 17, 77, 31. Our first hash function required to be as "remainder method" simply takes the item and divide it by table size, returning remainder as its hash value i.e.

1. h item = item % (size of table)
2. Let us say the size of table = 11, then
3. 54 % 11 = 10 26 % 11 = 4 93 % 11 = 5
4. 17 % 11 = 6 77 % 11 = 0 31 % 11 = 9

ITEM	HASH VALUE
54	10
26	4
93	5
17	6
77	0
31	9

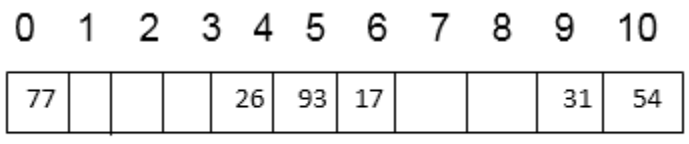


Fig: Hash Table

Now when we need to search any element, we just need to divide it by the table size, and we get the hash value. So we get the O (1) search time.

Now taking one more element 44 when we apply the hash function on 44, we get (44 % 11 = 0), But 0 hash value already has an element 77. This Problem is called as Collision.

Collision: According to the Hash Function, two or more item would need in the same slot. This is said to be called as Collision.

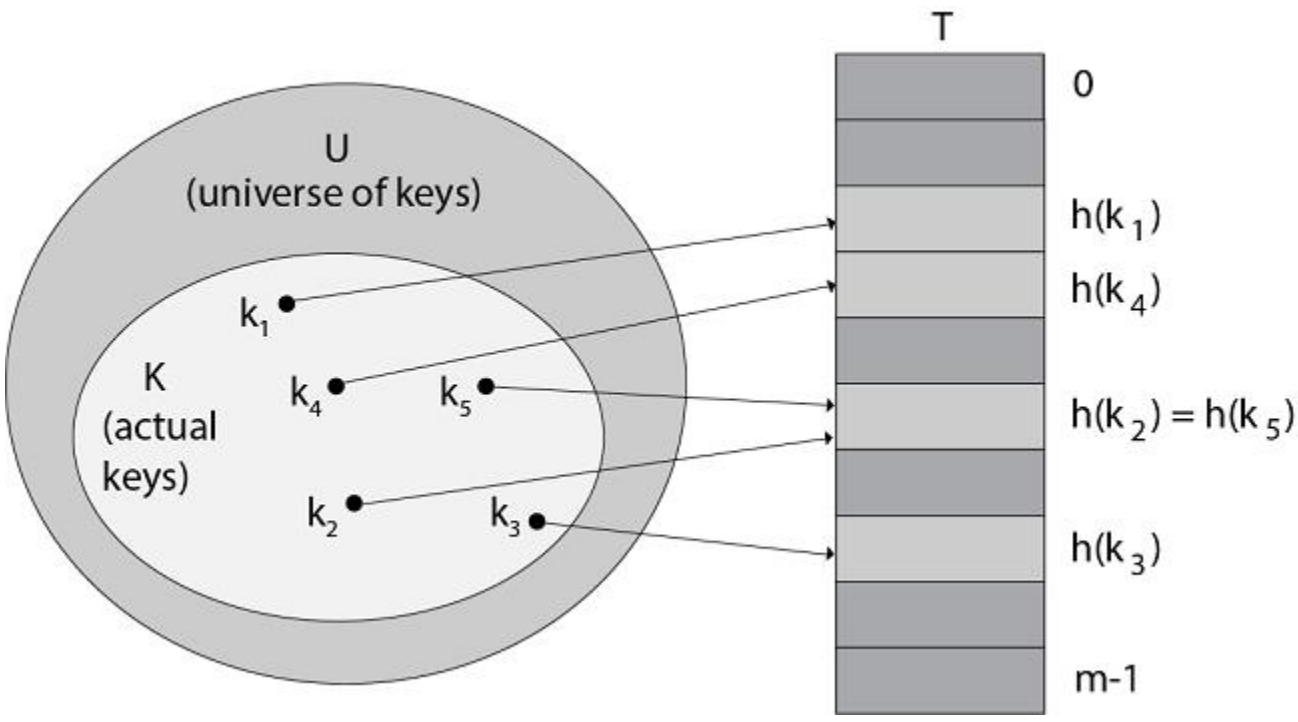


Figure: using a hash function h to map keys to hash-table slots. Because keys K2 and k5 map to the same slot, they collide.

Why use HashTable?

1. If **U** (Universe of keys) is large, storing a table T of size [U] may be impossible.
2. Set **k** of keys may be small relative to **U** so space allocated for **T** will waste.

So Hash Table requires less storage. Indirect addressing element with key k is stored in slot k with hashing it is stored in h (k) where h is a hash f^n and hash (k) is the value of key k. Hash f^n required array range.

Application of Hash Tables:

Some application of Hash Tables are:

1. **Database System:** Specifically, those that are required efficient random access. Usually, database systems try to develop between two types of access methods: sequential and random. Hash Table is an integral part of efficient random access because they provide a way to locate data in a constant amount of time.
2. **Symbol Tables:** The tables utilized by compilers to maintain data about symbols from a program. Compilers access information about symbols frequently. Therefore, it is essential that symbol tables be implemented very efficiently.
3. **Data Dictionaries:** Data Structure that supports adding, deleting, and searching for data. Although the operation of hash tables and a data dictionary are similar, other Data Structures may be used to implement data dictionaries.
4. **Associative Arrays:** Associative Arrays consist of data arranged so that n^{th} elements of one array correspond to the n^{th} element of another. Associative Arrays are helpful for indexing a logical grouping of data by several key fields.

Methods of Hashing

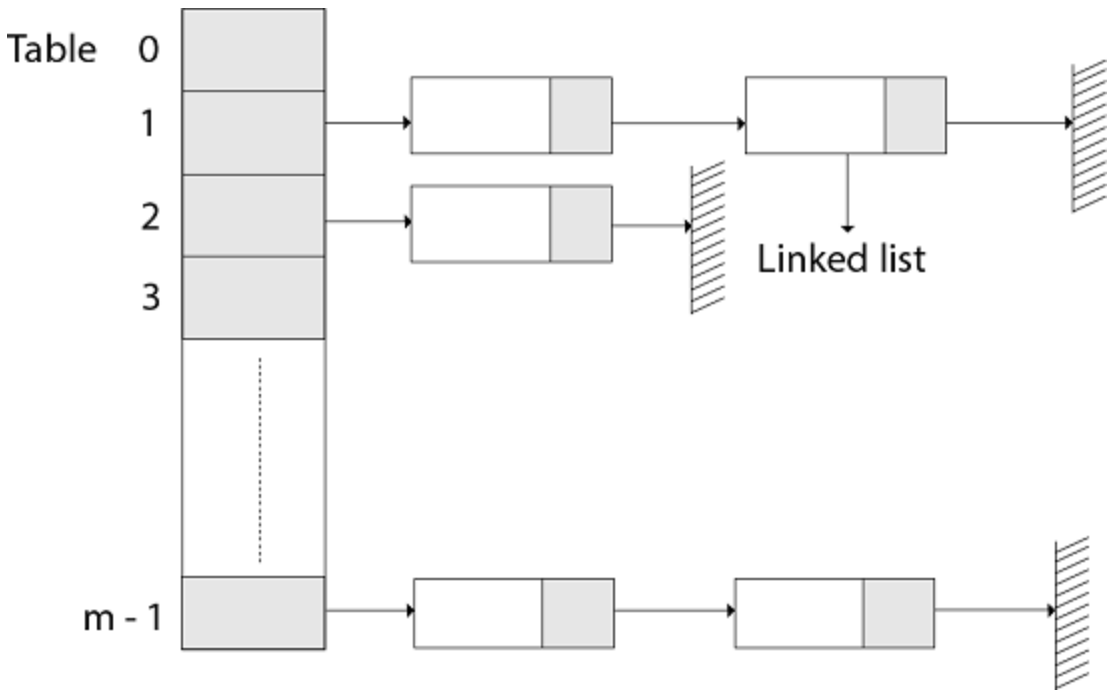
There are two main methods used to implement hashing:

1. Hashing with Chaining
2. Hashing with open addressing

1. Hashing with Chaining

In Hashing with Chaining, the element in S is stored in Hash table T [0...m-1] of size m, where m is somewhat larger than n, the size of S. The hash table is said to have m slots. Associated with the hashing scheme is a hash function h which is mapping from U to {0...m-1}.Each key k ∈S is stored in location T [h (k)], and we say that k is hashed into slot h (k). If more than one key in S hashed into the same slot then we have a **collision**.

In such case, all keys that hash into the same slot are placed in a linked list associated with that slot, this linked list is called the chain at slot. The load factor of a hash table is defined to be ∝=n/m it represents the average number of keys per slot. We typically operate in the range m=θ(n), so ∝ is usually a constant generally ∝<1.



Collision Resolution by Chaining:

In chaining, we place all the elements that hash to the same slot into the same linked list, As fig shows that Slot j contains a pointer to the head of the list of all stored elements that hash to j ; if there are no such elements, slot j contains NIL.

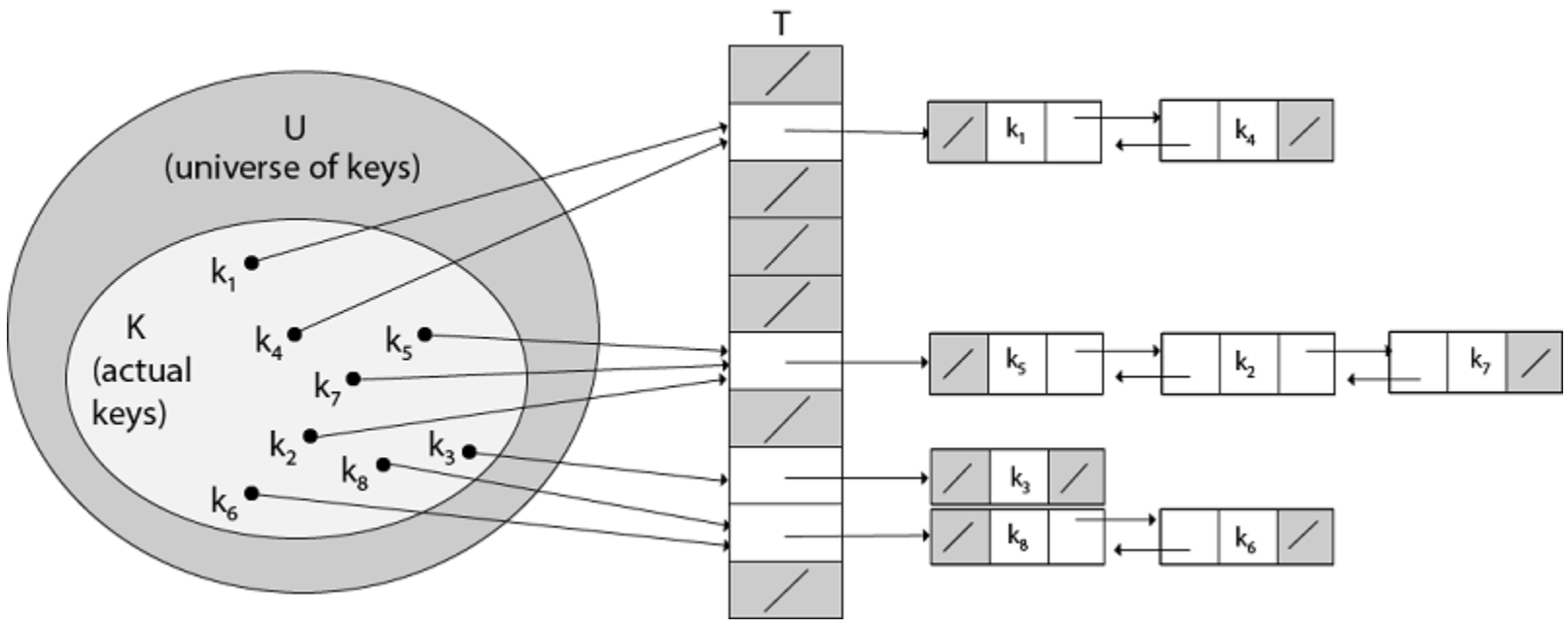


Fig: Collision resolution by chaining.

Each hash-table slot T [j] contains a linked list of all the keys whose hash value is j.

For example, h (k1) = h (k4) and h (k5) = h (k7) =h (K2). The linked list can be either singly or doubly linked; we show it as doubly linked because deletion is faster that way.

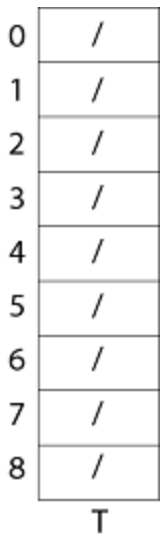
Analysis of Hashing with Chaining:

Given a hash table T with m slots that stores n elements, we define the load factors α for T as n/m that is the average number of elements stored in a chain. The worst case running time for searching is thus θ(n) plus the time to compute the hash function- no better than if we used one linked list for all the elements. Clearly, hash tables are not used for their worst-case performance.

The average performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average.

Example: let us consider the insertion of elements 5, 28, 19,15,20,33,12,17,10 into a chained hash table. Let us suppose the hash table has 9 slots and the hash function be h (k) =k mod 9.

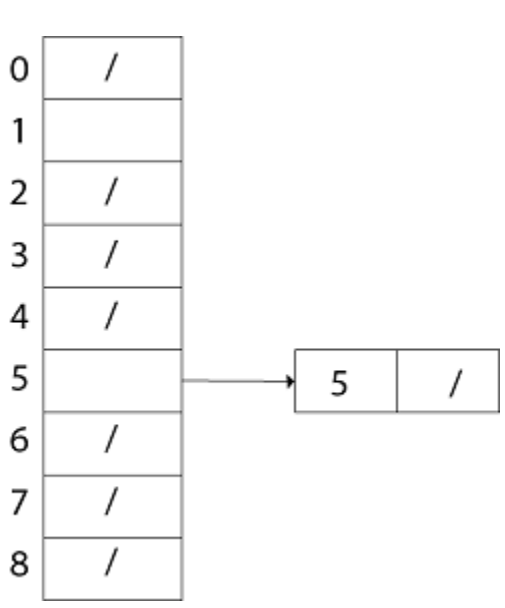
Solution: The initial state of chained-hash table



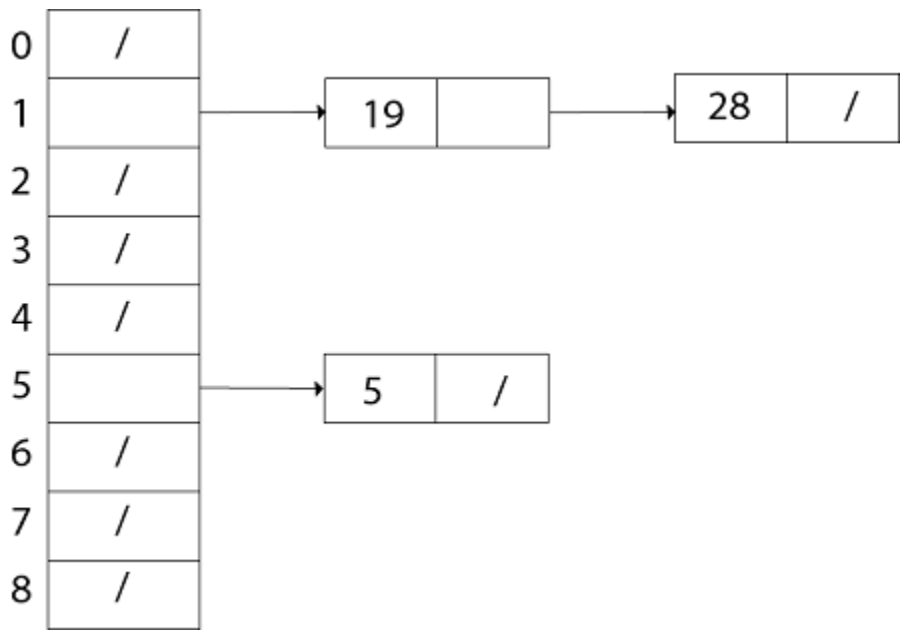
Insert 5:

1. h (5) = 5 mod 9 =5

Create a linked list for T [5] and store value 5 in it.

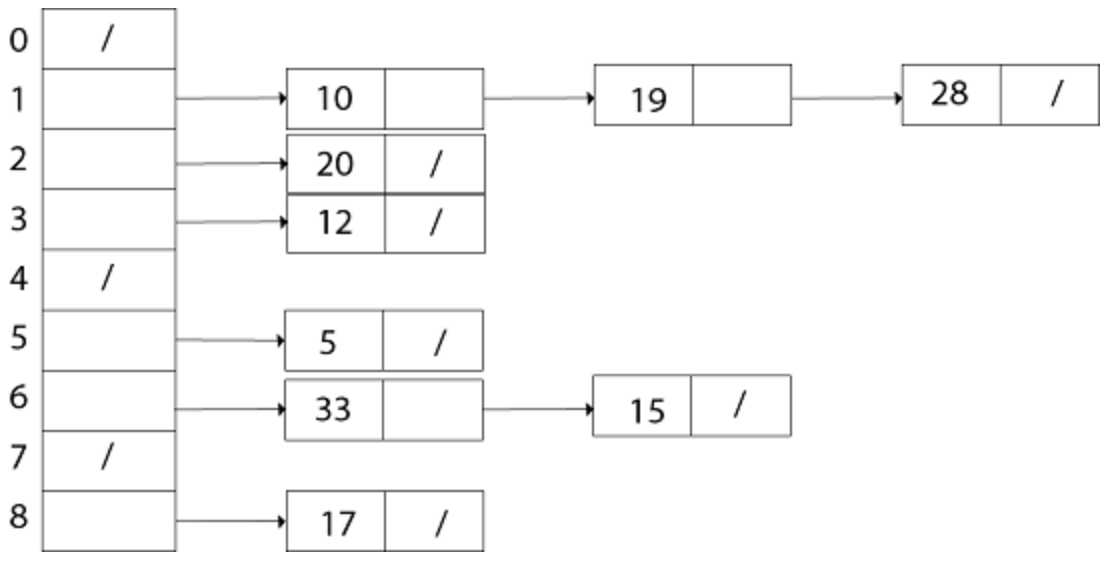


Similarly, insert 28. h (28) = 28 mod 9 =1. Create a Linked List for T [1] and store value 28 in it. Now insert 19 h (19) = 19 mod 9 = 1. Insert value 19 in the slot T [1] at the beginning of the linked-list.



1. Now insert h 15, h (15) = 15 mod 9 = 6. Create a link list **for** T [6] and store value 15 in it.
2. Similarly, insert 20, h (20) = 20 mod 9 = 2 in T [2].
3. Insert 33, h (33) = 33 mod 9 = 6
4. In the beginning of the linked list T [6]. Then,
5. Insert 12, h (12) = 12 mod 9 = 3 in T [3].
6. Insert 17, h (17) = 17 mod 9 = 8 in T [8].
7. Insert 10, h (10) = 10 mod 9 = 1 in T [1].

Thus the chained- hash- table after inserting key 10 is



2. Hashing with Open Addressing

In Open Addressing, all elements are stored in hash table itself. That is, each table entry consists of a component of the dynamic set or NIL. When searching for an item, we consistently examine table slots until either we find the desired object or we have determined that the element is not in the table. Thus, in open addressing, the load factor α can never exceed 1.

The advantage of open addressing is that it avoids Pointer. In this, we compute the sequence of slots to be examined. The extra memory freed by not sharing pointers provides the hash table with a larger number of slots for the same amount of memory, potentially yielding fewer collision and faster retrieval.

The process of examining the location in the hash table is called Probing.

Thus, the hash function becomes

1. h : U x {0,1,....m-1} → {0,1,....,m-1}.

With open addressing, we require that for every key k, the probe sequence

1. {h, (k, 0), h (k, 1)....h (k, m-1)}
2. Be a Permutation of (0, 1..... m-1)

The HASH-INSERT procedure takes as input a hash table T and a key k

HASH-INSERT (T, k)

```
1. i ← 0
2. repeat j ← h (k, i)
3. if T [j] = NIL
4. then T [j] ← k
5. return j
6. else ← i= i +1
7. until i=m
8. error "hash table overflow"
```

The procedure HASH-SEARCH takes as input a hash table T and a key k, returning j if it finds that slot j contains key k or NIL if key k is not present in table T.

HASH-SEARCH.T (k)

```
1. i ← 0
2. repeat j ← h (k, i)
3. if T [j] =j
4. then return j
5. i ← i+1
6. until T [j] = NIL or i=m
7. return NIL
```

Open Addressing Techniques

Three techniques are commonly used to compute the probe sequence required for open addressing:

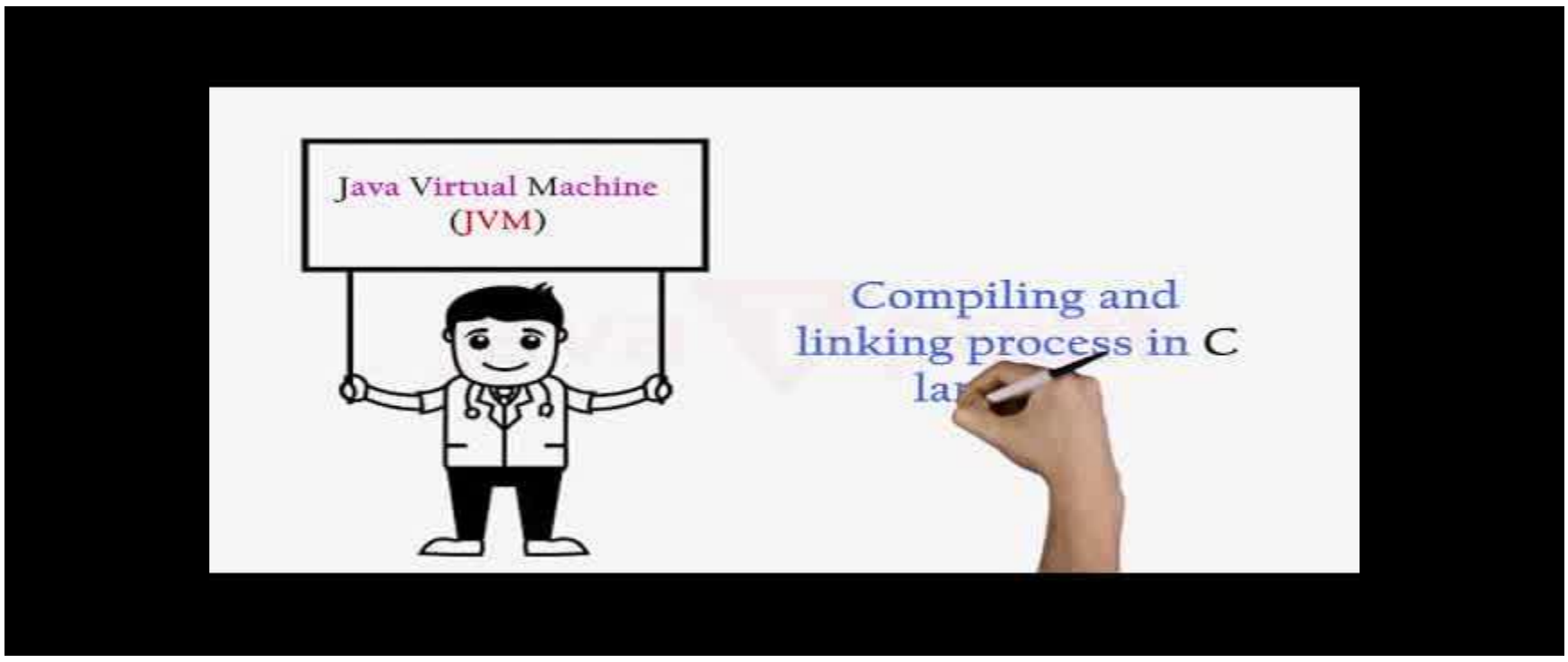
1. Linear Probing.
2. Quadratic Probing.
3. Double Hashing.

1. Linear Probing:

It is a Scheme in Computer Programming for resolving collision in hash tables.

Suppose a new record R with key k is to be added to the memory table T but that the memory locations with the hash address H (k). H is already filled.

Our natural key to resolve the collision is to crossing R to the first available location following T (h). We assume that the table T with m location is circular, so that T [i] comes after T [m].



The above collision resolution is called "Linear Probing".

Linear probing is simple to implement, but it suffers from an issue known as primary clustering. Long runs of occupied slots build up, increasing the average search time. Clusters arise because an empty slot proceeded by i full slots gets filled next with probability (i + 1)/m. Long runs of occupied slots tend to get longer, and the average search time increases.

Given an ordinary hash function h': U {0, 1...m-1}, the method of linear probing uses the hash function.

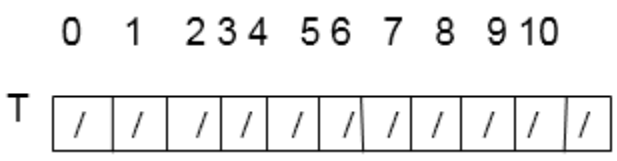
1. h (k, i) = (h' (k) + i) mod m

Where 'm' is the size of hash table and h' (k) = k mod m. for i=0, 1....m-1.

Given key k, the first slot is T [h' (k)]. We next slot T [h' (k) +1] and so on up to the slot T [m-1]. Then we wrap around to slots T [0], T [1]....until finally slot T [h' (k)-1]. Since the initial probe position dispose of the entire probe sequence, only m distinct probe sequences are used with linear probing.

Example: Consider inserting the keys 24, 36, 58,65,62,86 into a hash table of size m=11 using linear probing, consider the primary hash function is h' (k) = k mod m.

Solution: Initial state of hash table



Insert 24. We know h (k, i) = [h' (k) + i] mod m
Now h (24, 0) = [24 mod 11 + 0] mod 11
= (2+0) mod 11 = 2 mod 11 = 2
Since T [2] is free, insert key 24 at this place.

Insert 36. Now h (36, 0) = [36 mod 11 + 0] mod 11
= [3+0] mod 11 = 3
Since T [3] is free, insert key 36 at this place.

Insert 58. Now h (58, 0) = [58 mod 11 +0] mod 11
= [3+0] mod 11 =3
Since T [3] is not free, so the next sequence is
h (58, 1) = [58 mod 11 +1] mod 11
= [3+1] mod 11= 4 mod 11=4

T [4] is free; Insert key 58 at this place.

Insert 65. Now h (65, 0) = [65 mod 11 +0] mod 11
= (10 +0) mod 11= 10
T [10] is free. Insert key 65 at this place.

Insert 62. Now h (62, 0) = [62 mod 11 +0] mod 11
= [7 + 0] mod 11 = 7
T [7] is free. Insert key 62 at this place.

Insert 86. Now h (86, 0) = [86 mod 11 + 0] mod 11
= [9 + 0] mod 11 = 9
T [9] is free. Insert key 86 at this place.
Thus,

	0	1	2	3	4	5	6	7	8	9	10
T	/	/	24	36	58	/	/	62	/	86	65

2. Quadratic Probing:

Suppose a record R with key k has the hash address H (k) = h then instead of searching the location with addresses h, h+1, and h+ 2...We linearly search the locations with addresses

$h, h+1, h+4, h+9...h+i^2$

Quadratic Probing uses a hash function of the form

$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$

Where (as in linear probing) h' is an auxiliary hash function c_1 and $c_2 \neq 0$ are auxiliary constants and i=0, 1...m-1. The initial position is T [h' (k)]; later position probed is offset by the amount that depend in a quadratic manner on the probe number i.

Example: Consider inserting the keys 74, 28, 36,58,21,64 into a hash table of size m =11 using quadratic probing with c_1=1 and c_2=3. Further consider that the primary hash function is h' (k) = k mod m.

Solution: For Quadratic Probing, we have

$h(k, i) = [k \bmod m + c_1 i + c_2 i^2) \bmod m$

0	1	2	3	4	5	6	7	8	9	10
/	/	/	/	/	/	/	/	/	/	/

This is the initial state of hash table

Here c_1= 1 and c_2=3
h (k, i) = [k mod m + i + $3i^2$] mod m

Insert 74.

h (74,0)= (74 mod 11+0+3x0) mod 11
= (8 +0+0) mod 11 = 8
T [8] is free; insert the key 74 at this place.

Insert 28.

h (28, 0) = (28 mod 11 + 0 + 3 x 0) mod 11
= (6 +0 + 0) mod 11 = 6.
T [6] is free; insert key 28 at this place.

Insert 36.

h (36, 0) = (36 mod 11 + 0 + 3 x 0) mod 11
= (3 + 0+0) mod 11=3
T [3] is free; insert key 36 at this place.

Insert 58.

h (58, 0) = (58 mod 11 + 0 + 3 x 0) mod 11

```
                = (3 + 0 + 0) mod 11 = 3
T [3] is not free, so next probe sequence is computed as
      h (59, 1) = (58 mod 11 + 1 + 3 x1²) mod 11
                  = (3 + 1 + 3) mod 11
                =7 mod 11= 7
T [7] is free; insert key 58 at this place.
```

Insert 21.

```
      h (21, 0) = (21 mod 11 + 0 + 3 x 0)
                = (10 + 0) mod 11 = 10
T [10] is free; insert key 21 at this place.

      Insert 64.

      h (64, 0) = (64 mod 11 + 0 + 3 x 0)
                = (9 + 0+ 0) mod 11 = 9.
T [9] is free; insert key 64 at this place.
```

Thus, after inserting all keys, the hash table is

0	1	2	3	4	5	6	7	8	9	10
/	/	/	36	/	/	28	58	74	64	21

3. Double Hashing:

Double Hashing is one of the best techniques available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations.

Double hashing uses a hash function of the form

$h(k, i) = (h_1(k) + i\, h_2(k)) \bmod m$

Where h_1 and h_2 are auxiliary hash functions and m is the size of the hash table.

$h_1(k) = k \bmod m$ or $h_2(k) = k \bmod m'$. Here m' is slightly less than m (say m-1 or m-2).

Example: Consider inserting the keys 76, 26, 37,59,21,65 into a hash table of size m = 11 using double hashing. Consider that the auxiliary hash functions are $h_1(k)=k \bmod 11$ and $h_2(k) = k \bmod 9$.

Solution: Initial state of Hash table is

	0	1	2	3	4	5	6	7	8	9	10
T	/	/	/	/	/	/	/	/	/	/	/

1. Insert 76.

```
          h₁(76) = 76 mod 11 = 10
          h₂(76) = 76 mod 9 = 4
                    h (76, 0) = (10 + 0 x 4) mod 11
                              = 10 mod 11 = 10
T [10] is free, so insert key 76 at this place.
```

2. Insert 26.

```
       h₁(26) = 26 mod 11 = 4
           h₂(26) = 26 mod 9 = 8
                     h (26, 0) = (4 + 0 x 8) mod 11
                               = 4 mod 11 = 4
T [4] is free, so insert key 26 at this place.
```

3. Insert 37.

```
       h₁(37) = 37 mod 11 = 4
           h₂(37) = 37 mod 9 = 1
                      h (37, 0) = (4 + 0 x 1) mod 11 = 4 mod 11 = 4
T [4] is not free, the next probe sequence is
                   h (37, 1) = (4 + 1 x 1) mod 11 = 5 mod 11 = 5
T [5] is free, so insert key 37 at this place.
```

4. Insert 59.

$h_1(59) = 59 \bmod 11 = 4$

$h_2(59) = 59 \bmod 9 = 5$

$h\ (59, 0) = (4 + 0 \times 5) \bmod 11 = 4 \bmod 11 = 4$

Since, T [4] is not free, the next probe sequence is

$h\ (59, 1) = (4 + 1 \times 5) \bmod 11 = 9 \bmod 11 = 9$

T [9] is free, so insert key 59 at this place.

5. Insert 21.

$h_1(21) = 21 \bmod 11 = 10$

$h_2(21) = 21 \bmod 9 = 3$

$h\ (21, 0) = (10 + 0 \times 3) \bmod 11 = 10 \bmod 11 = 10$

T [10] is not free, the next probe sequence is

$h\ (21, 1) = (10 + 1 \times 3) \bmod 11 = 13 \bmod 11 = 2$

T [2] is free, so insert key 21 at this place.

6. Insert 65.

$h_1(65) = 65 \bmod 11 = 10$

$h_2(65) = 65 \bmod 9 = 2$

$h\ (65, 0) = (10 + 0 \times 2) \bmod 11 = 10 \bmod 11 = 10$

T [10] is not free, the next probe sequence is

$h\ (65, 1) = (10 + 1 \times 2) \bmod 11 = 12 \bmod 11 = 1$

T [1] is free, so insert key 65 at this place.

Thus, after insertion of all keys the final hash table is

0	1	2	3	4	5	6	7	8	9	10
/	65	21	/	26	37	/	/	/	59	76

Hash Function

Hash Function is used to index the original value or key and then used later each time the data associated with the value or key is to be retrieved. Thus, hashing is always a one-way operation. There is no need to "reverse engineer" the hash function by analyzing the hashed values.

Characteristics of Good Hash Function:

1. The hash value is fully determined by the data being hashed.
2. The hash Function uses all the input data.
3. The hash function "uniformly" distributes the data across the entire set of possible hash values.
4. The hash function generates complicated hash values for similar strings.

Some Popular Hash Function is:

1. Division Method:

Choose a number m smaller than the number of n of keys in k (The number m is usually chosen to be a prime number or a number without small divisors, since this frequently a minimum number of collisions).

The hash function is:

$$h(k) = k \bmod m \qquad h(k) = k \bmod m + 1$$

For Example: if the hash table has size m = 12 and the key is k = 100, then h (k) = 4. Since it requires only a single division operation, hashing by division is quite fast.

2. Multiplication Method:

The multiplication method for creating hash functions operates in two steps. First, we multiply the key k by a constant A in the range 0 < A < 1 and extract the fractional part of kA. Then, we increase this value by m and take the floor of the result.

The hash function is:

$$h(k) = \lfloor m(k\,A \bmod 1) \rfloor$$

Where "k A mod 1" means the fractional part of k A, that is, k A -⌊k A⌋.

3. Mid Square Method:

The key k is squared. Then function H is defined by

1. H (k) = L

Where L is obtained by deleting digits from both ends of k^2. We emphasize that the same position of k^2 must be used for all of the keys.

4. Folding Method:

The key k is partitioned into a number of parts k_1, k_2.... k_n where each part except possibly the last, has the same number of digits as the required address.

Then the parts are added together, ignoring the last carry.

$$H\ (k)\ =\ k^1+\ k^2+.....+k^n$$

Example: Company has 68 employees, and each is assigned a unique four- digit employee number. Suppose L consist of 2-digit addresses: 00, 01, and 02....99. We apply the above hash functions to each of the following employee numbers:

1. 3205, 7148, 2345

(a) Division Method: Choose a Prime number m close to 99, such as m =97, Then

1. H (3205) = 4, H (7148) = 67, H (2345) = 17.

That is dividing 3205 by 17 gives a remainder of 4, dividing 7148 by 97 gives a remainder of 67, dividing 2345 by 97 gives a remainder of 17.

(b) Mid-Square Method:

k =	3205	7148	2345
k^2=	10272025	51093904	5499025
h (k) =	72	93	99

Observe that fourth & fifth digits, counting from right are chosen for hash address.

(c) Folding Method: Divide the key k into 2 parts and adding yields the following hash address:

1. H (3205) = 32 + 50 = 82 H (7148) = 71 + 84 = 55
2. H (2345) = 23 + 45 = 68

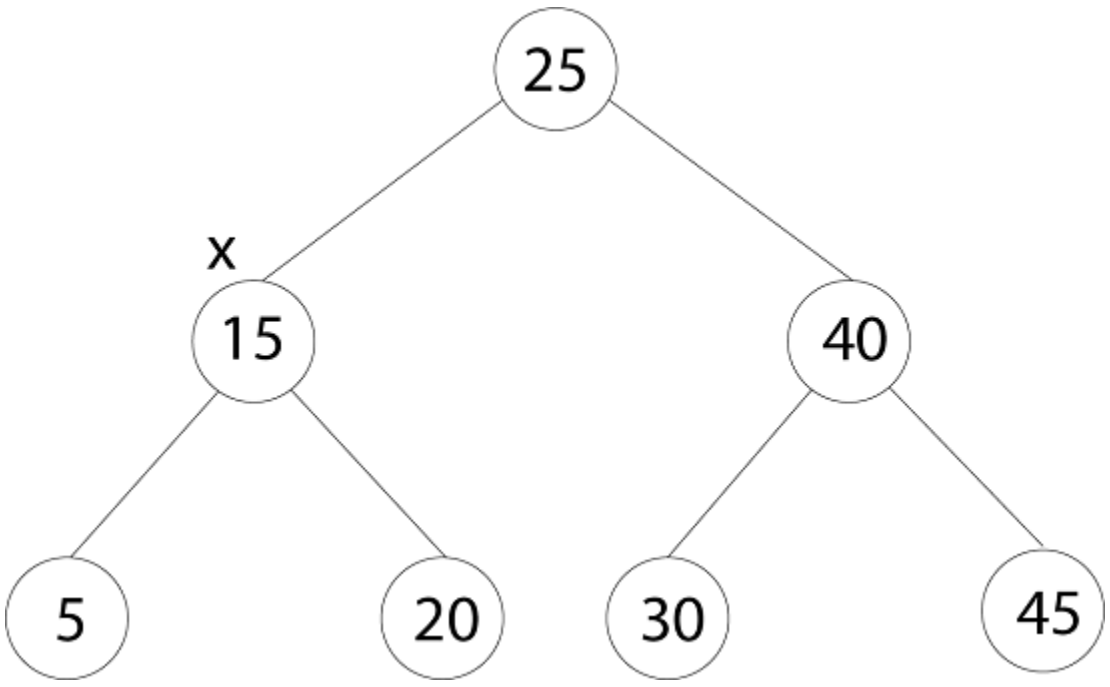
Binary Search Trees

A Binary Search tree is organized in a Binary Tree. Such a tree can be defined by a linked data structure in which a particular node is an object. In addition to a key field, each node contains field left, right, and p that point to the nodes corresponding to its left child, its right child, and its parent, respectively. If a child or parent is missing, the appropriate field contains the value NIL. The root node is the only node in the tree whose parent field is Nil.

Binary Search Tree Property

Let x be a node in a binary search tree.

- If y is a node in the left subtree of x, then key [y] ≤key [k].
- If z is a node in the right subtree of x, then key [x] ≤ key [y].



In this tree key [x] = 15

- If y is a node in the left subtree of x, then key [y] = 5.

1. i.e. key [y] ≤ key[x].

- If y is a node in the right subtree of x, then key [y] = 20.

1. i.e. key [x] ≤ key[y].

Traversal in Binary Search Trees:

1. In-Order-Tree-Walk (x): Always prints keys in binary search tree in sorted order.

INORDER-TREE-WALK (x) - Running time is θ(n)

```
1. If x ≠ NIL.
2. then INORDER-TREE-WALK (left [x])
3. print key [x]
4. INORDER-TREE-WALK (right [x])
```

2. PREORDER-TREE-WALK (x): In which we visit the root node before the nodes in either subtree.

PREORDER-TREE-WALK (x):

```
1. If x ≠ NIL.
2. then print key [x]
3. PREORDER-TREE-WALK (left [x]).
4. PREORDER-TREE-WALK (right [x]).
```

3. POSTORDER-TREE-WALK (x): In which we visit the root node after the nodes in its subtree.

POSTORDER-TREE-WALK (x):

```
1. If x ≠ NIL.
2. then POSTORDER-TREE-WALK (left [x]).
3. POSTORDER-TREE-WALK (right [x]).
```

```
4. print key [x]
```

Querying a Binary Search Trees:

1. Searching: The TREE-SEARCH (x, k) algorithm searches the tree node at x for a node whose key value equal to k. It returns a pointer to the node if it exists otherwise NIL.

TREE-SEARCH (x, k)

```
1. If x = NIL or k = key [x].
2. then return x.
3. If k < key [x].
4. then return TREE-SEARCH (left [x], k)
5. else return TREE-SEARCH (right [x], k)
```

Clearly, this algorithm runs in O (h) time where h is the height of the tree. The iterative version of the above algorithm is very easy to implement

ITERATIVE-TREE- SEARCH (x, k)

```
1. while x ≠ NIL and k ≠ key [k].
2. do if k < key [x].
3. then x ← left [x].
4. else x ← right [x].
5. return x.
```

2. Minimum and Maximum: An item in a binary search tree whose key is a minimum can always be found by following left child pointers from the root until a NIL is encountered. The following procedure returns a pointer to the minimum element in the subtree rooted at a given node x.

TREE- MINIMUM (x)

```
1. While left [x] ≠ NIL.
2. do x←left [x].
3. return x.
```

TREE-MAXIMUM (x)

```
1. While left [x] ≠ NIL
2. do x←right [x].
3. return x.
```

3. Successor and predecessor: Given a node in a binary search tree, sometimes we used to find its successor in the sorted form determined by an in order tree walk. If all keys are specific, the successor of a node x is the node with the smallest key greater than key[x]. The structure of a binary search tree allows us to rule the successor of a node without ever comparing keys. The following action returns the successor of a node x in a binary search tree if it exists, and NIL if x has the greatest key in the tree:

TREE SUCCESSOR (x)

```
1. If right [x] ≠ NIL.
2. Then return TREE-MINIMUM (right [x]))
3. y←p [x]
4. While y ≠ NIL and x = right [y]
5. do x←y
6. y←p[y]
7. return y.
```

The code for TREE-SUCCESSOR is broken into two cases. If the right subtree of node x is nonempty, then the successor of x is just the leftmost node in the right subtree, which we find in line 2 by calling TREE-MINIMUM (right [x]). On the other hand, if the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. To find y, we quickly go up the tree from x until we encounter a node that is the left child of its parent; lines 3-7 of TREE-SUCCESSOR handle this case.

The running time of TREE-SUCCESSOR on a tree of height h is O (h) since we either follow a simple path up the tree or follow a simple path down the tree. The procedure TREE-PREDECESSOR, which is symmetric to TREE-SUCCESSOR, also runs in time O (h).

4. Insertion in Binary Search Tree: To insert a new value into a binary search tree T, we use the procedure TREE-INSERT. The procedure takes a node ´ for which key [z] = v, left [z] NIL, and right [z] = NIL. It modifies T and some of the attributes of z in such a way that it inserts into an appropriate position in the tree.

TREE-INSERT (T, z)

```
1. y ←NIL.
2. x←root [T]
3. while x ≠ NIL.
4. do y←x
5. if key [z]< key [x]
```

```
6. then x←left [x].
7. else x←right [x].
8. p [z]←y
9. if y = NIL.
10. then root [T]←z
11. else if key [z] < key [y]
12. then left [y]←z
```

For Example:

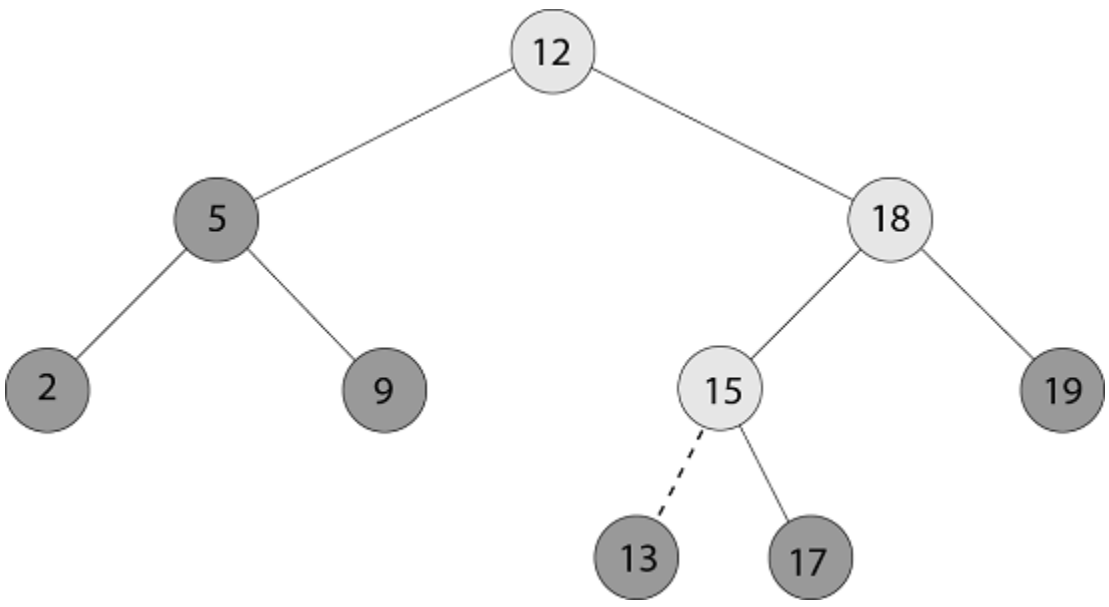


Fig: Working of TREE-INSERT

Suppose we want to insert an item with key 13 into a Binary Search Tree.

1. x = 1
2. y = 1 as x ≠ NIL.
3. Key [z] < key [x]
4. 13 < not equal to 12.
5. x ←right [x].
6. x ←3
7. Again x ≠ NIL
8. y ←3
9. key [z] < key [x]
10. 13 < 18
11. x←left [x]
12. x←6
13. Again x ≠ NIL, y←6
14. 13 < 15
15. x←left [x]
16. x←NIL
17. p [z]←6

Now our node z will be either left or right child of its parent (y).

1. key [z] < key [y]
2. 13 < 15
3. Left [y] ← z
4. Left [6] ← z

So, insert a node in the left of node index at 6.

5. Deletion in Binary Search Tree: When Deleting a node from a tree it is essential that any relationships, implicit in the tree can be maintained. The deletion of nodes from a binary search tree will be considered:

There are three distinct cases:

1. **Nodes with no children:** This case is trivial. Simply set the parent's pointer to the node to be deleted to nil and delete the node.
2. **Nodes with one child:** When z has no left child then we replace z by its right child which may or may not be NIL. And when z has no right child, then we replace z with its right child.

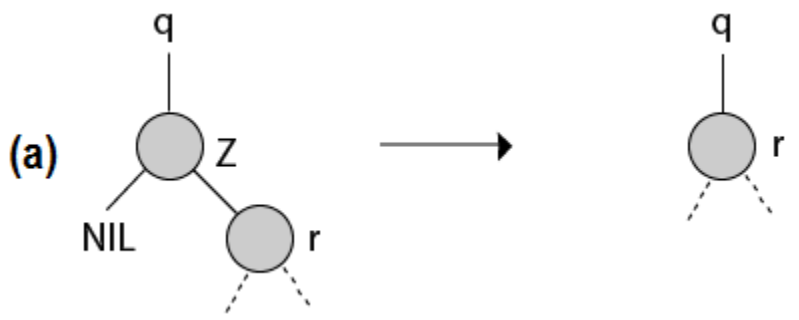
3. **Nodes with both Childs:** When z has both left and right child. We find z's successor y, which lies in right z's right subtree and has no left child (the successor of z will be a node with minimum value its right subtree and so it has no left child).
    - If y is z's right child, then we replace z.
    - Otherwise, y lies within z's right subtree but not z's right child. In this case, we first replace z by its own right child and the replace z by y.

TREE-DELETE (T, z)

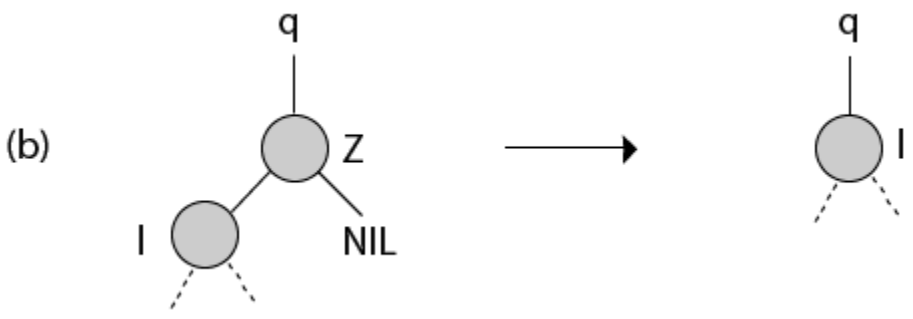
```
1. If left [z] = NIL or right [z] = NIL.
2. Then y ← z
3. Else y  ← TREE- SUCCESSOR (z)
4. If left [y] ≠ NIL.
5. Then x ← left [y]
6. Else x ← right [y]
7. If x ≠NIL
8. Then p[x] ← p [y]
9. If p[y] = NIL.
10. Then root [T] ← x
11. Else if y = left [p[y]]
12. Then left [p[y]] ← x
13. Else right [p[y]] ← y
14. If y ≠ z.
15. Then key [z] ← key [y]
16. If y has other fields, copy them, too.
17. Return y
```

The Procedure runs in O (h) time on a tree of height h.

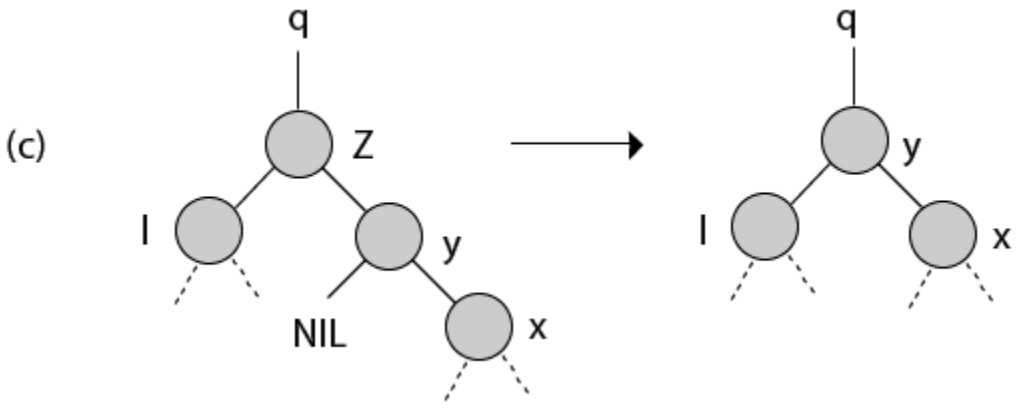
For Example: Deleting a node z from a binary search tree. Node z may be the root, a left child of node q, or a right child of q.



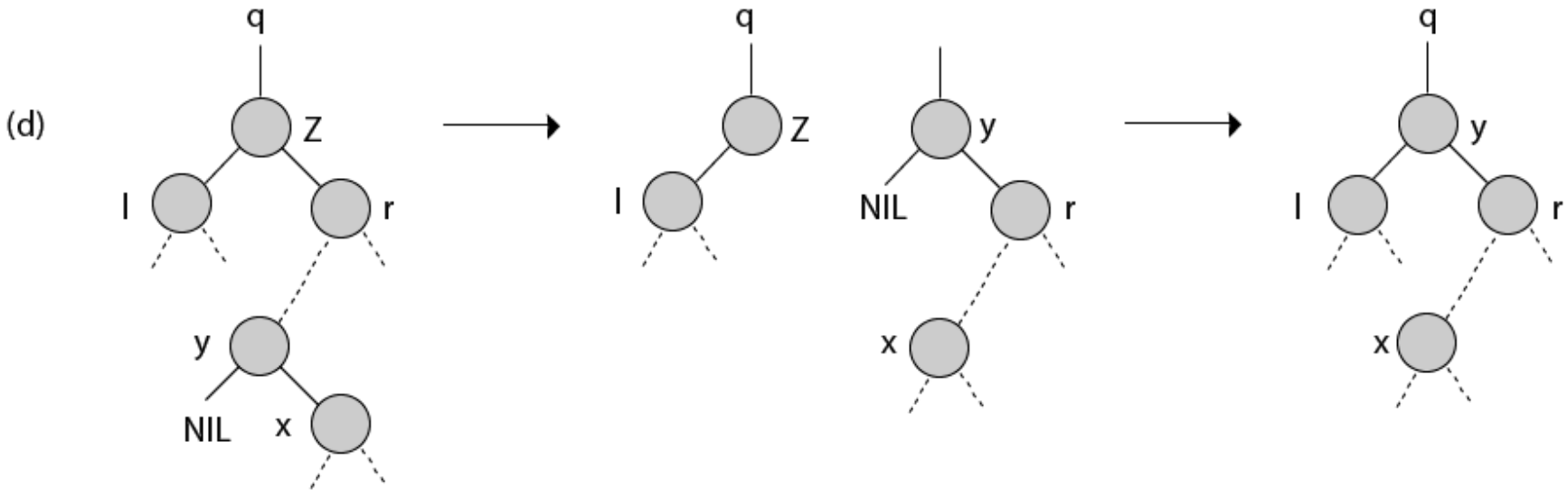
Z has the only right child.



Z has the only left child. We replace z by l.



Node z has two children; its left child is node l, its right child is its successor y, and y's right child is node x. We replace z by y, updating y's left child to become l, but leaving x as y's right child.



Node z has two children (left child l and right child r), and its successor y ≠ r lies within the subtree rooted at r. We replace y with its own right child x, and we set y to be r's parent. Then, we set y to be q's child and the parent of l.

Red Black Tree

A Red Black Tree is a category of the self-balancing binary search tree. It was created in 1972 by Rudolf Bayer who termed them **"symmetric binary B-trees**."

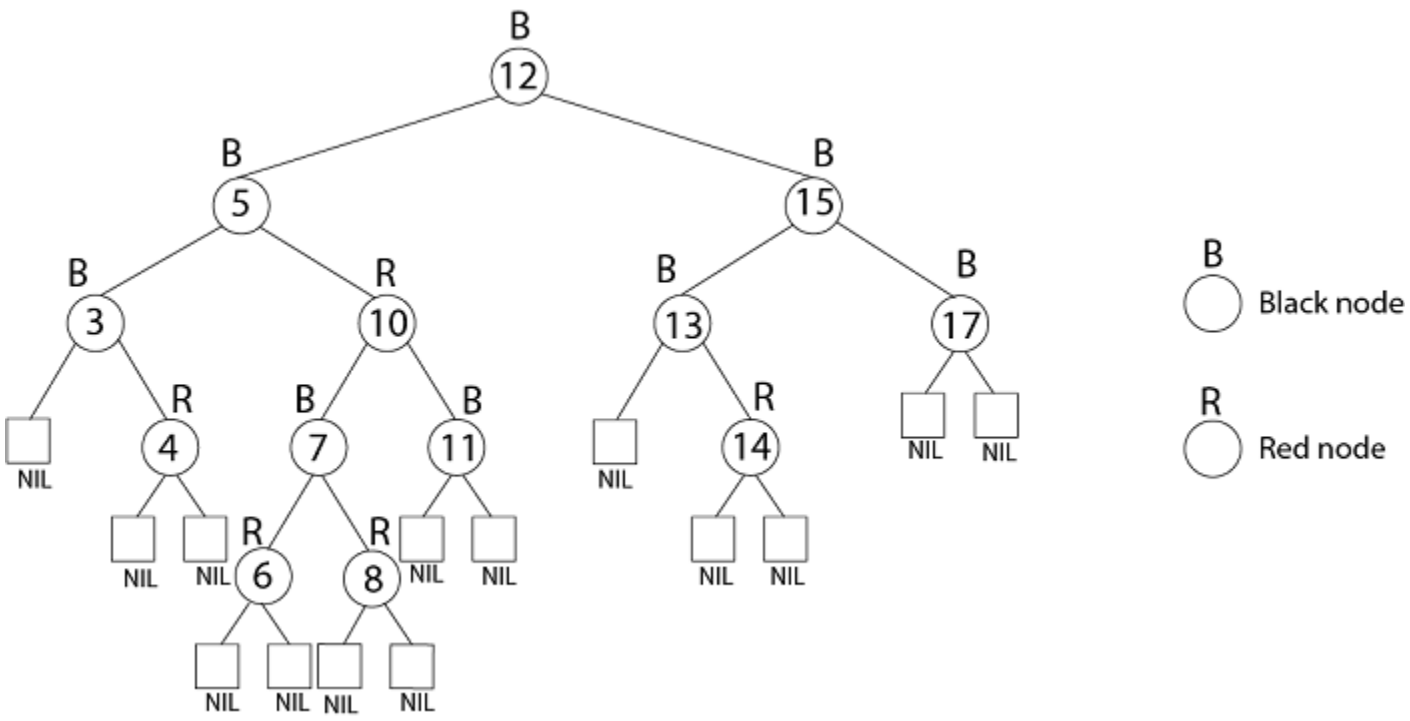
A red-black tree is a Binary tree where a particular node has color as an extra attribute, either red or black. By check the node colors on any simple path from the root to a leaf, red-black trees secure that no such path is higher than twice as long as any other so that the tree is generally balanced.

Properties of Red-Black Trees

A red-black tree must satisfy these properties:

1. The root is always black.
2. A nil is recognized to be black. This factor that every non-NIL node has two children.
3. **Black Children Rule:** The children of any red node are black.
4. **Black Height Rule:** For particular node v, there exists an integer bh (v) such that specific downward path from v to a nil has correctly bh (v) black real (i.e. non-nil) nodes. Call this portion the black height of v. We determine the black height of an RB tree to be the black height of its root.

A tree T is an almost red-black tree (ARB tree) if the root is red, but other conditions above hold.

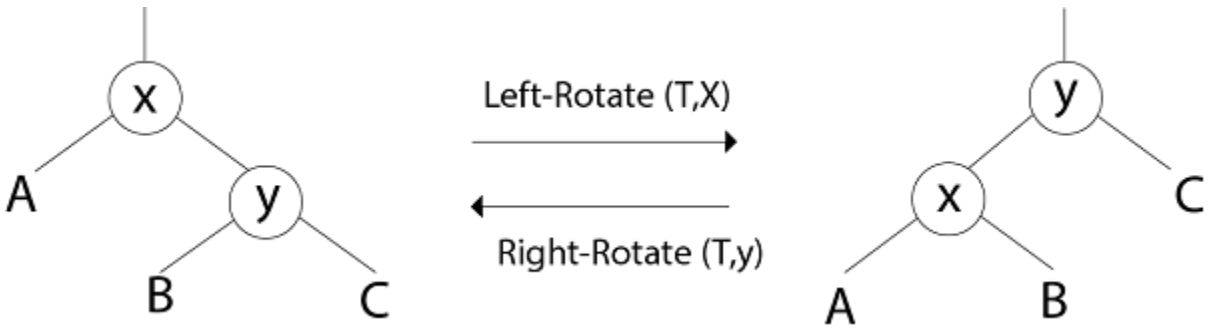


Operations on RB Trees:

The search-tree operations TREE-INSERT and TREE-DELETE, when runs on a red-black tree with n keys, take O (log n) time. Because they customize the tree, the conclusion may violate the red-black properties. To restore these properties, we must change the color of some of the nodes in the tree and also change the pointer structure.

1. Rotation:

Restructuring operations on red-black trees can generally be expressed more clearly in details of the rotation operation.



Clearly, the order (Ax By C) is preserved by the rotation operation. Therefore, if we start with a BST and only restructure using rotation, then we will still have a BST i.e. rotation do not break the BST-Property.

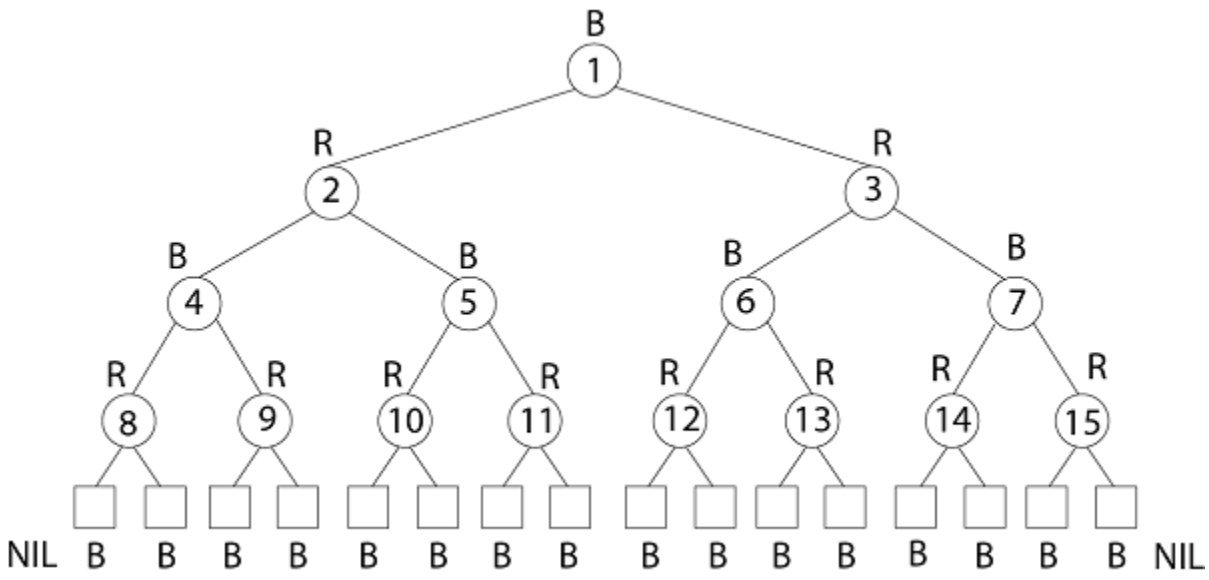
LEFT ROTATE (T, x)

```
1. y ← right [x]
1. y ← right [x]
2. right [x] ← left [y]
3. p [left[y]] ← x
```

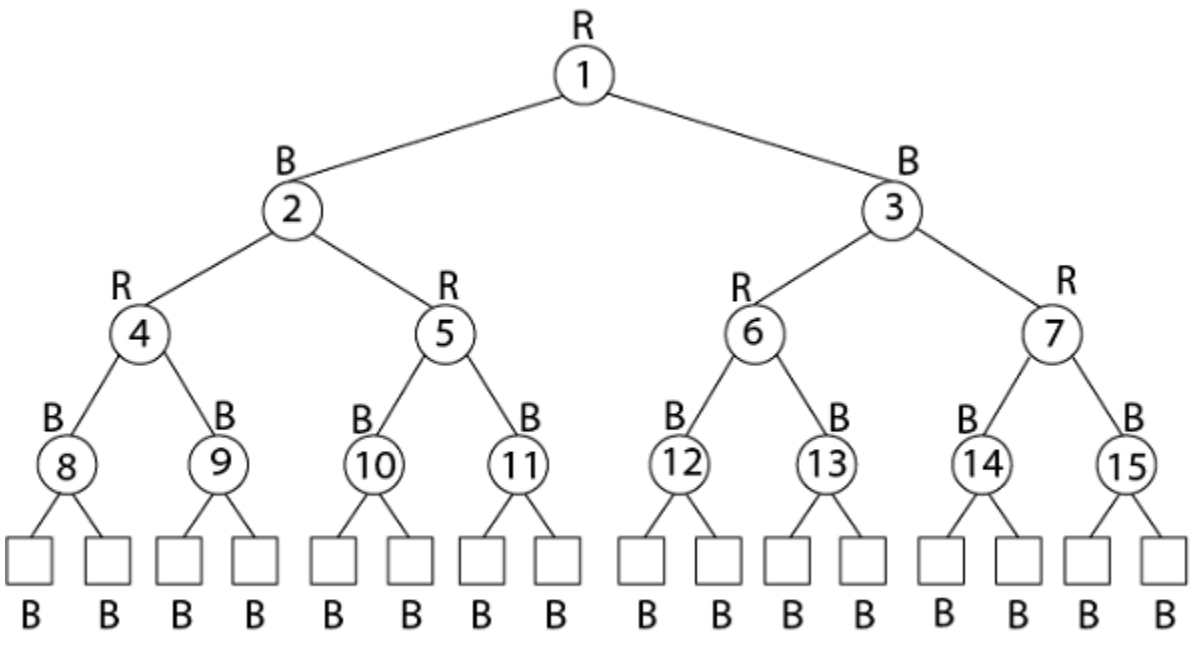
```
4. p[y] ← p[x]
5. If p[x] = nil [T]
  then root [T] ← y
   else if x = left [p[x]]
     then left [p[x]] ← y
   else right [p[x]] ← y
6. left [y] ← x.
7. p [x] ← y.
```

Example: Draw the complete binary tree of height 3 on the keys {1, 2, 3... 15}. Add the NIL leaves and color the nodes in three different ways such that the black heights of the resulting trees are: 2, 3 and 4.

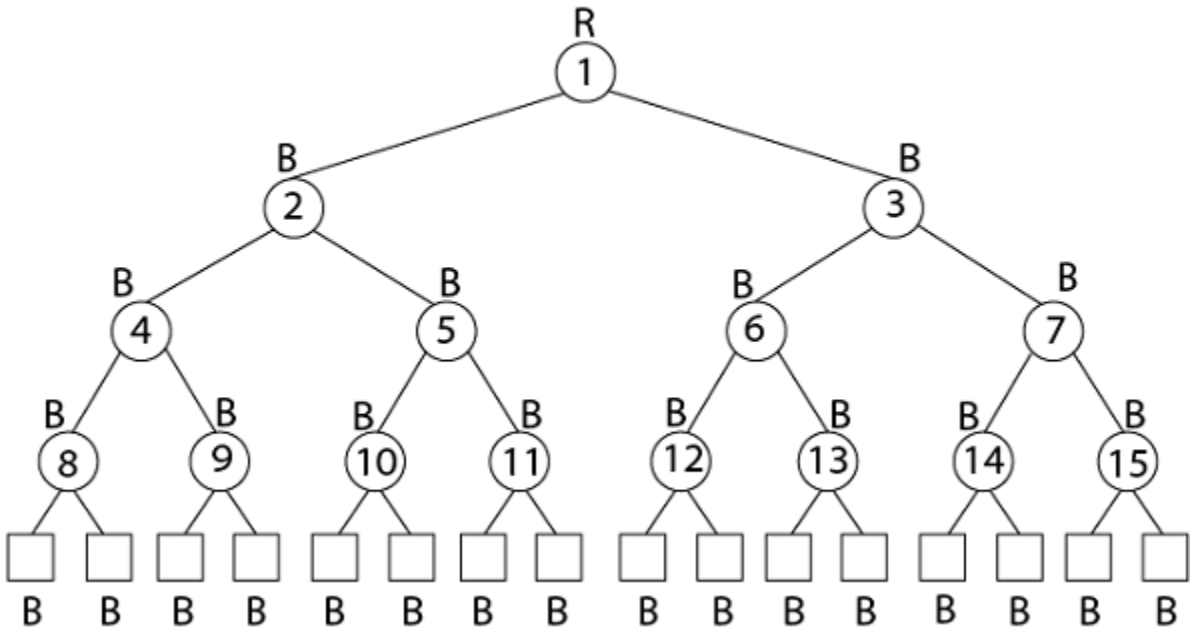
Solution:



Tree with black-height-2



Tree with black-height-3



Tree with black-height-4

2. Insertion:

- Insert the new node the way it is done in Binary Search Trees.
- Color the node red
- If an inconsistency arises for the red-black tree, fix the tree according to the type of discrepancy.

A discrepancy can decision from a parent and a child both having a red color. This type of discrepancy is determined by the location of the node concerning grandparent, and the color of the sibling of the parent.

RB-INSERT (T, z)

```
1. y ← nil [T]
2. x ← root [T]
3. while x ≠ NIL [T]
4. do y ← x
5. if key [z] < key [x]
6. then x  ← left [x]
7. else x ←  right [x]
8. p [z] ← y
9. if y = nil [T]
10. then root [T] ← z
11. else if key [z] < key [y]
12. then left [y] ← z
13. else right [y] ← z
14. left [z] ← nil [T]
15. right [z] ← nil [T]
16. color [z] ← RED
17. RB-INSERT-FIXUP (T, z)
```

After the insert new node, Coloring this new node into black may violate the black-height conditions and coloring this new node into red may violate coloring conditions i.e. root is black and red node has no red children. We know the black-height violations are hard. So we color the node red. After this, if there is any color violation, then we have to correct them by an RB-INSERT-FIXUP procedure.

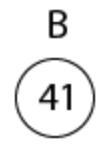
RB-INSERT-FIXUP (T, z)

```
1. while color [p[z]] = RED
2. do if p [z] = left [p[p[z]]]
3. then y ← right [p[p[z]]]
4. If color [y] = RED
5. then color [p[z]] ← BLACK    //Case 1
6. color [y] ← BLACK            //Case 1
7. color [p[z]] ← RED           //Case 1
8. z  ← p[p[z]]                 //Case 1
9. else if z= right [p[z]]
10. then z ← p [z]              //Case 2
11. LEFT-ROTATE (T, z)          //Case 2
12. color [p[z]] ← BLACK        //Case 3
13. color [p [p[z]]] ← RED      //Case 3
14. RIGHT-ROTATE  (T,p [p[z]])  //Case 3
15. else (same as then clause)
     With "right" and "left" exchanged
16. color [root[T]] ← BLACK
```

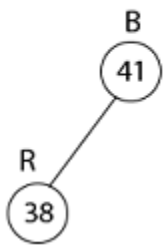
Example: Show the red-black trees that result after successively inserting the keys 41,38,31,12,19,8 into an initially empty red-black tree.

Solution:

Insert 41

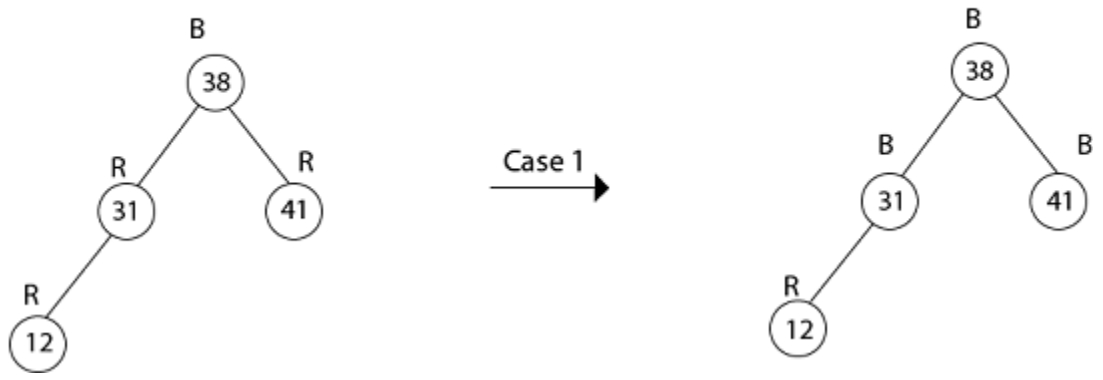


Insert 38

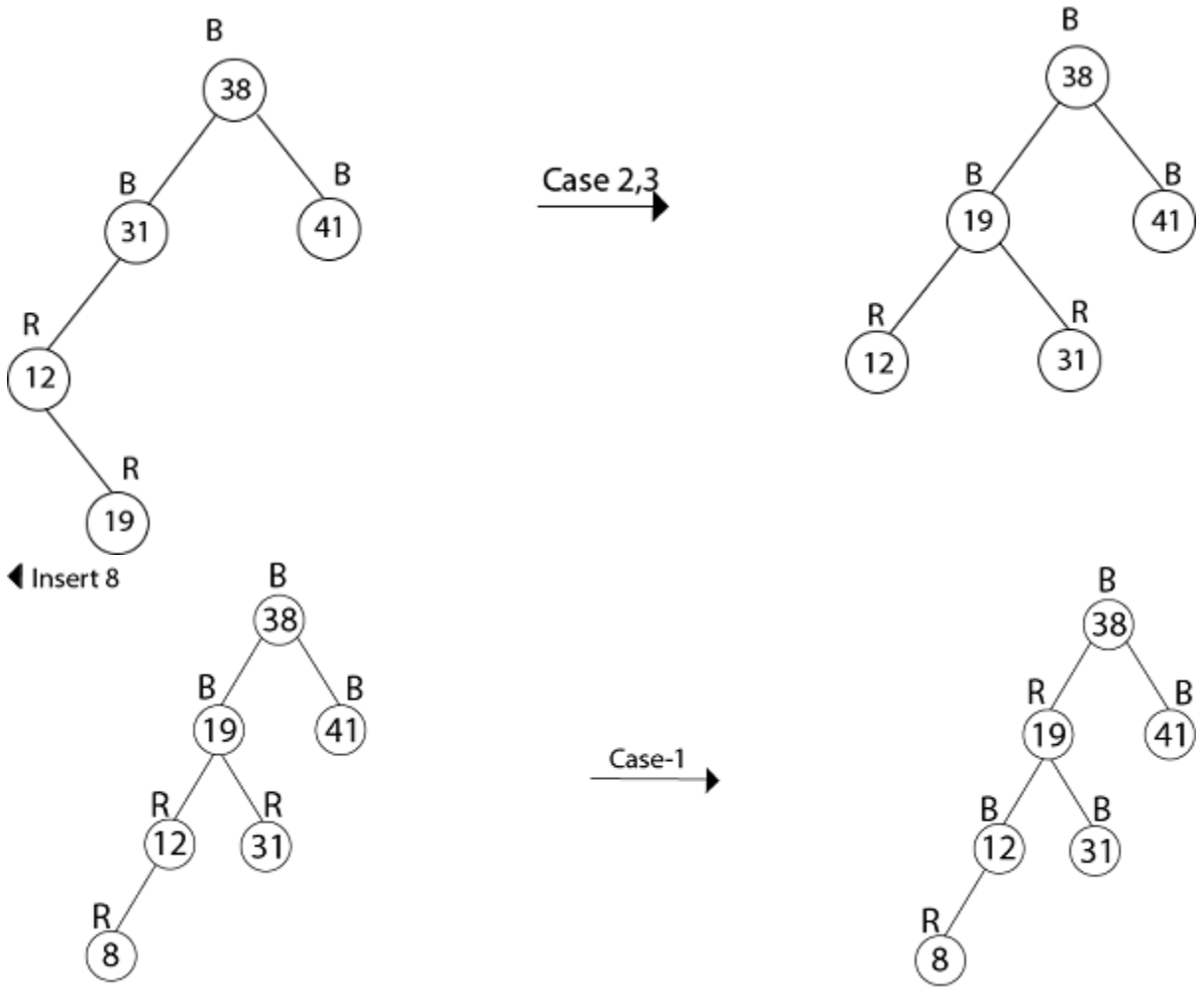


Insert 31

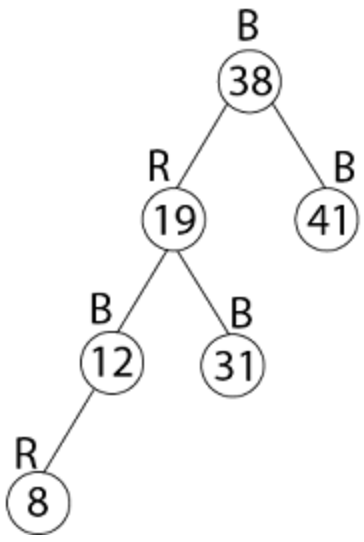
Insert 12



Insert 19



Thus the final tree is



---

3. Deletion:

First, search for an element to be deleted

- If the element to be deleted is in a node with only left child, swap this node with one containing the largest element in the left subtree. (This node has no right child).
- If the element to be deleted is in a node with only right child, swap this node with the one containing the smallest element in the right subtree (This node has no left child).
- If the element to be deleted is in a node with both a left child and a right child, then swap in any of the above two ways. While swapping, swap only the keys but not the colors.
- The item to be deleted is now having only a left child or only a right child. Replace this node with its sole child. This may violate red constraints or black constraint. Violation of red constraints can be easily fixed.
- If the deleted node is black, the black constraint is violated. The elimination of a black node y causes any path that contained y to have one fewer black node.
- Two cases arise:
  - The replacing node is red, in which case we merely color it black to make up for the loss of one black node.
  - The replacing node is black.

The strategy RB-DELETE is a minor change of the TREE-DELETE procedure. After splicing out a node, it calls an auxiliary procedure RB-DELETE-FIXUP that changes colors and performs rotation to restore the red-black properties.

RB-DELETE (T, z)

```
1. if left [z] = nil [T] or right [z] = nil [T]
2. then y ← z
3. else y ← TREE-SUCCESSOR (z)
4. if left [y] ≠ nil [T]
5. then x ← left [y]
6. else x ← right [y]
7. p [x] ←  p [y]
8. if p[y] = nil [T]
9. then root [T]  ← x
10. else if y = left [p[y]]
11. then left [p[y]] ← x
12. else right [p[y]] ← x
13. if y≠ z
14. then key [z] ← key [y]
15. copy y's satellite data into z
16. if color [y] = BLACK
17. then RB-delete-FIXUP (T, x)
18. return y
```

RB-DELETE-FIXUP (T, x)

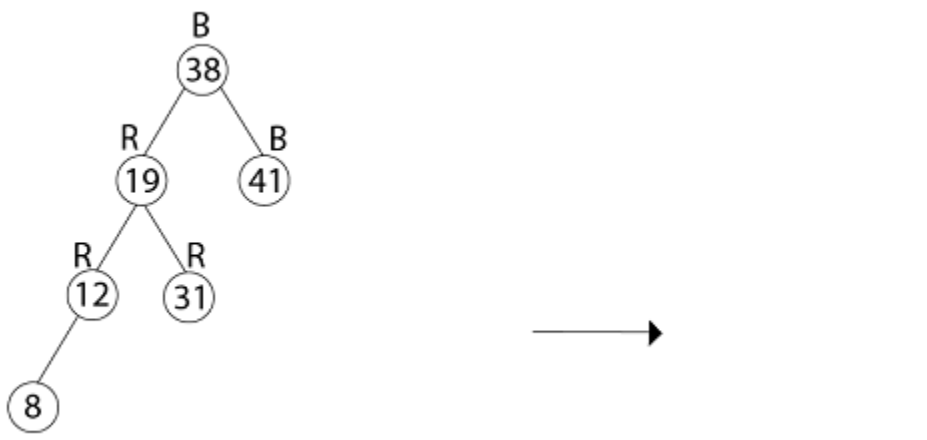
```
1. while x ≠ root [T] and color [x] = BLACK
2. do if x = left [p[x]]
3. then w ← right [p[x]]
4. if color [w] = RED
5. then color [w] ← BLACK        //Case 1
6. color [p[x]] ← RED            //Case 1
7. LEFT-ROTATE (T, p [x])        //Case 1
8. w ← right [p[x]]              //Case 1
```

```
9. If color [left [w]] = BLACK and color [right[w]] = BLACK
10. then color [w] ← RED          //Case 2
11. x ← p[x]                      //Case 2
12. else if color [right [w]] = BLACK
13. then color [left[w]] ← BLACK //Case 3
14. color [w] ← RED               //Case 3
15. RIGHT-ROTATE (T, w)           //Case 3
16. w ← right [p[x]]              //Case 3
17. color [w] ← color [p[x]]      //Case 4
18. color p[x] ← BLACK            //Case 4
19. color [right [w]] ← BLACK     //Case 4
20. LEFT-ROTATE (T, p [x])        //Case 4
21. x ← root [T]                  //Case 4
22. else (same as then clause with "right" and "left" exchanged)
23. color [x] ← BLACK
```

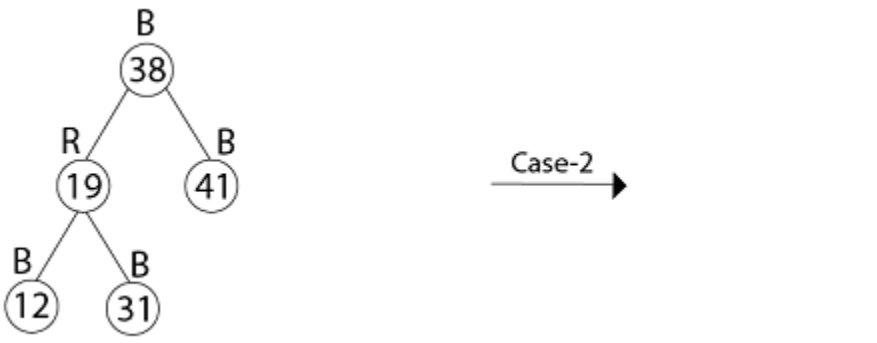
Example: In a previous example, we found that the red-black tree that results from successively inserting the keys 41,38,31,12,19,8 into an initially empty tree. Now show the red-black trees that result from the successful deletion of the keys in the order 8, 12, 19,31,38,41.

Solution:

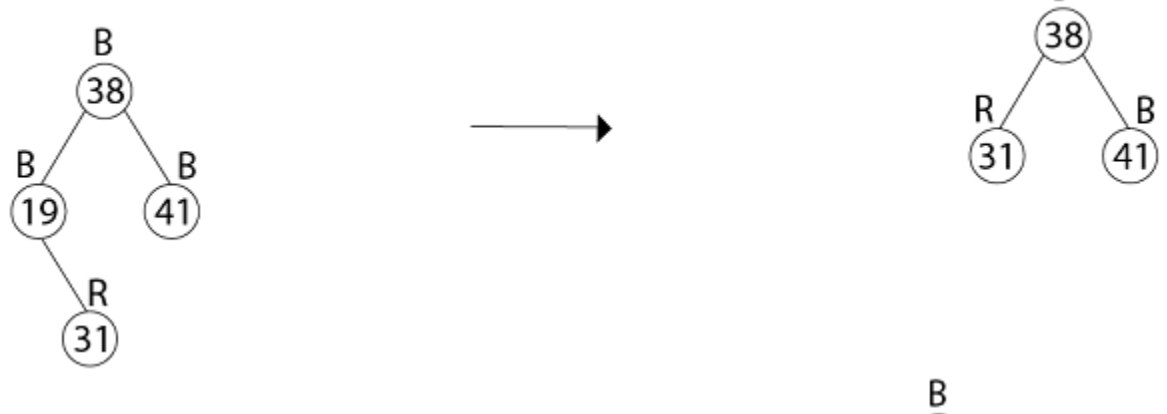
Delete 8



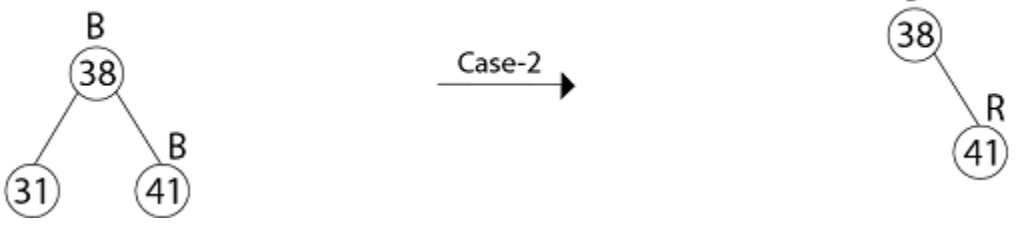
Delete 12



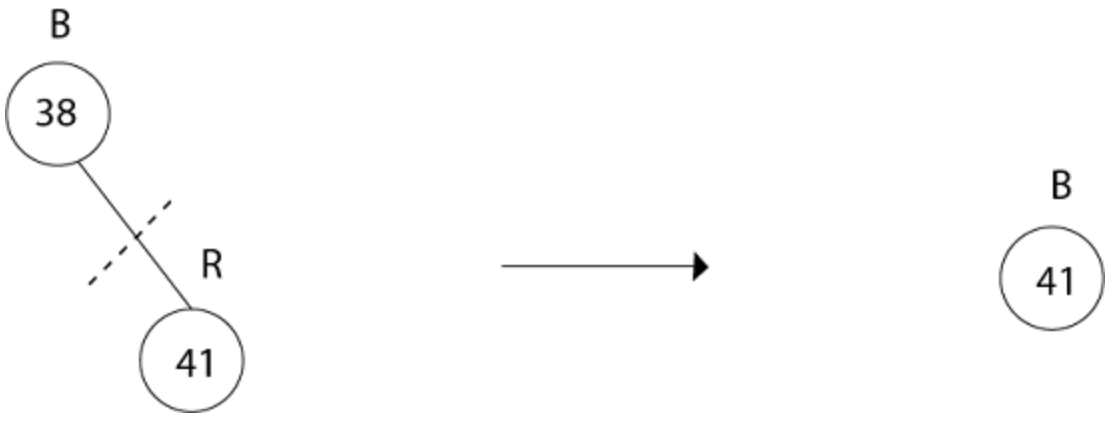
Delete 19



Delete 31



Delete 38



Delete 41

No Tree.

Introduction of Dynamic Programming

Dynamic Programming is the most powerful design technique for solving optimization problems.

Divide & Conquer algorithm partition the problem into disjoint subproblems solve the subproblems recursively and then combine their solution to solve the original problems.

Dynamic Programming is used when the subproblems are not independent, e.g. when they share the same subproblems. In this case, divide and conquer may do more work than necessary, because it solves the same sub problem multiple times.

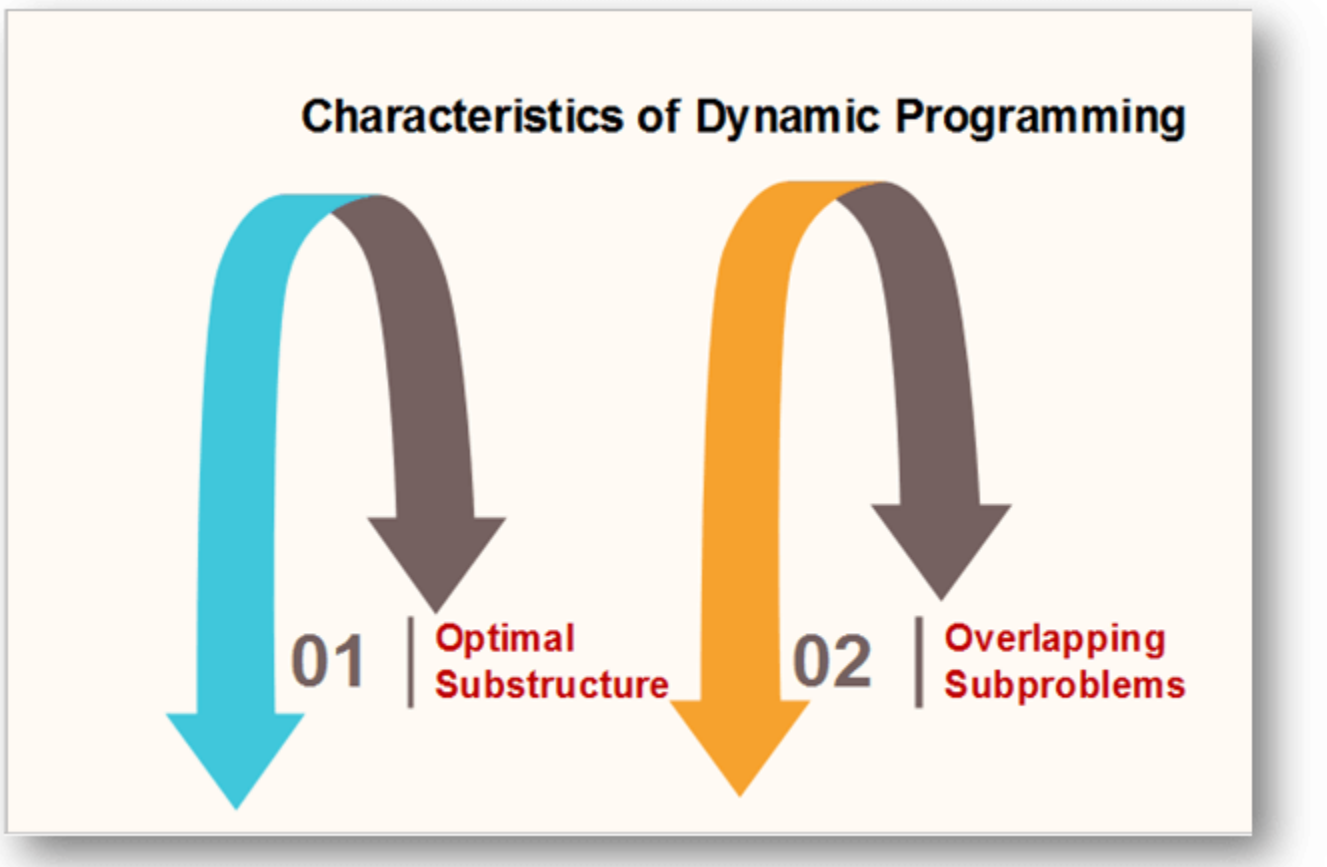
Dynamic Programming solves each subproblems just once and stores the result in a table so that it can be repeatedly retrieved if needed again.

Dynamic Programming is a **Bottom-up approach-** we solve all possible small problems and then combine to obtain solutions for bigger problems.

Dynamic Programming is a paradigm of algorithm design in which an optimization problem is solved by a combination of achieving sub-problem solutions and appearing to the "**principle of optimality**".

Characteristics of Dynamic Programming:

Dynamic Programming works when a problem has the following features:-



- **Optimal Substructure:** If an optimal solution contains optimal sub solutions then a problem exhibits optimal substructure.
- **Overlapping subproblems:** When a recursive algorithm would visit the same subproblems repeatedly, then a problem has overlapping subproblems.

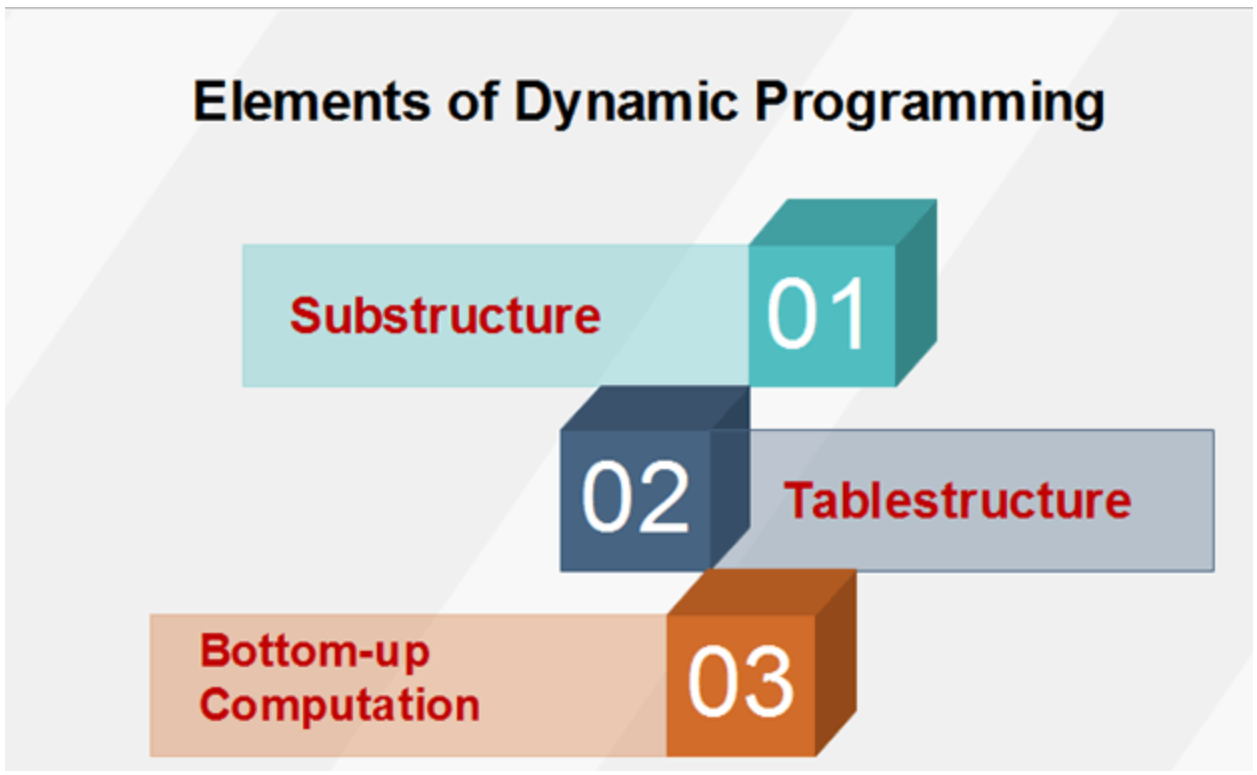
If a problem has optimal substructure, then we can recursively define an optimal solution. If a problem has overlapping subproblems, then we can improve on a recursive implementation by computing each subproblem only once.

If a problem doesn't have optimal substructure, there is no basis for defining a recursive algorithm to find the optimal solutions. If a problem doesn't have overlapping sub problems, we don't have anything to gain by using dynamic programming.

If the space of subproblems is enough (i.e. polynomial in the size of the input), dynamic programming can be much more efficient than recursion.

Elements of Dynamic Programming

There are basically three elements that characterize a dynamic programming algorithm:-

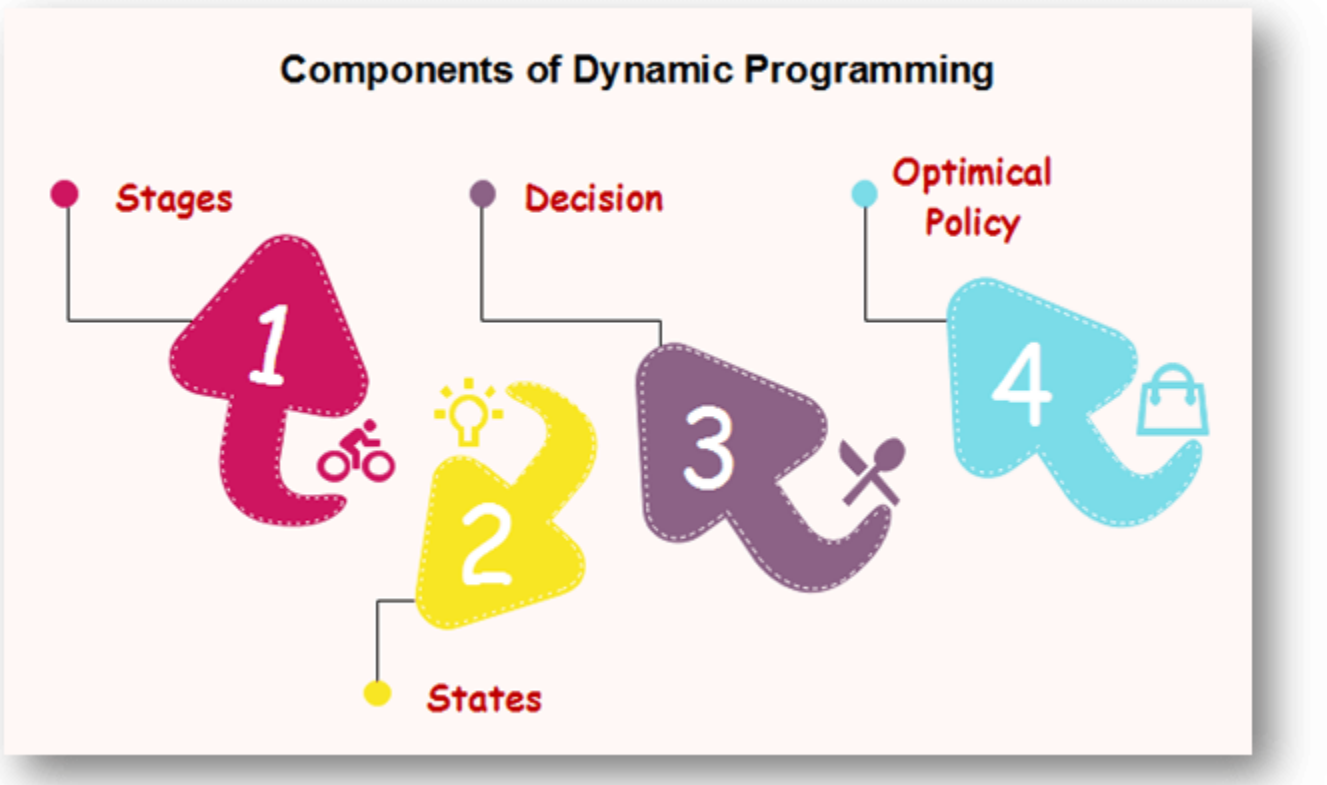


1. **Substructure:** Decompose the given problem into smaller subproblems. Express the solution of the original problem in terms of the solution for smaller problems.
2. **Table Structure:** After solving the sub-problems, store the results to the sub problems in a table. This is done because subproblem solutions are reused many times, and we do not want to repeatedly solve the same problem over and over again.
3. **Bottom-up Computation:** Using table, combine the solution of smaller subproblems to solve larger subproblems and eventually arrives at a solution to complete problem.

Note: Bottom-up means:-

i. Start with smallest subproblems.
ii. Combining their solutions obtain the solution to sub-problems of increasing size.
iii. Until solving at the solution of the original problem.

Components of Dynamic programming



1. **Stages:** The problem can be divided into several subproblems, which are called stages. A stage is a small portion of a given problem. For example, in the shortest path problem, they were defined by the structure of the graph.
2. **States:** Each stage has several states associated with it. The states for the shortest path problem was the node reached.
3. **Decision:** At each stage, there can be multiple choices out of which one of the best decisions should be taken. The decision taken at every stage should be optimal; this is called a stage decision.
4. **Optimal policy:** It is a rule which determines the decision at each stage; a policy is called an optimal policy if it is globally optimal. This is known as Bellman principle of optimality.
5. Given the current state, the optimal choices for each of the remaining states does not depend on the previous states or decisions. In the shortest path problem, it was not necessary to know how we got a node only that we did.

6. There exist a recursive relationship that identify the optimal decisions for stage j, given that stage j+1, has already been solved.
7. The final stage must be solved by itself.

Development of Dynamic Programming Algorithm

It can be broken into four steps:

1. Characterize the structure of an optimal solution.
2. Recursively defined the value of the optimal solution. Like Divide and Conquer, divide the problem into two or more optimal parts recursively. This helps to determine what the solution will look like.
3. Compute the value of the optimal solution from the bottom up (starting with the smallest subproblems)
4. Construct the optimal solution for the entire problem form the computed values of smaller subproblems.

Applications of dynamic programming

1. 0/1 knapsack problem
2. Mathematical optimization problem
3. All pair Shortest path problem
4. Reliability design problem
5. Longest common subsequence (LCS)
6. Flight control and robotics control
7. Time-sharing: It schedules the job to maximize CPU usage

Differentiate between Divide & Conquer Method vs Dynamic Programming.

Divide & Conquer Method	Dynamic Programming
1.It deals (involves) three steps at each level of recursion: **Divide** the problem into a number of subproblems. **Conquer** the subproblems by solving them recursively. **Combine** the solution to the subproblems into the solution for original subproblems.	**1.**It involves the sequence of four steps: ○ Characterize the structure of optimal solutions. ○ Recursively defines the values of optimal solutions. ○ Compute the value of optimal solutions in a Bottom-up minimum. ○ Construct an Optimal Solution from computed information.
2. It is Recursive.	**2.** It is non Recursive.
3. It does more work on subproblems and hence has more time consumption.	**3.** It solves subproblems only once and then stores in the table.
4. It is a top-down approach.	**4.** It is a Bottom-up approach.
5. In this subproblems are independent of each other.	**5.** In this subproblems are interdependent.
6. For example: Merge Sort & Binary Search etc.	**6. For example:** Matrix Multiplication.

Fibonacci sequence

Fibonacci sequence is the sequence of numbers in which every next item is the total of the previous two items. And each number of the Fibonacci sequence is called Fibonacci number.

Example: 0 ,1,1,2,3,5,8,13,21,..... is a Fibonacci sequence.

The Fibonacci numbers F_nare defined as follows:

$F_0 = 0$

$F_n=1$

$F_n=F_{(n-1)}+ F_{(n-2)}$

FIB (n)

```
1. If (n < 2)
2. then return n
3. else return FIB (n - 1) + FIB (n - 2)
```

Figure: shows four levels of recursion for the call fib (8):

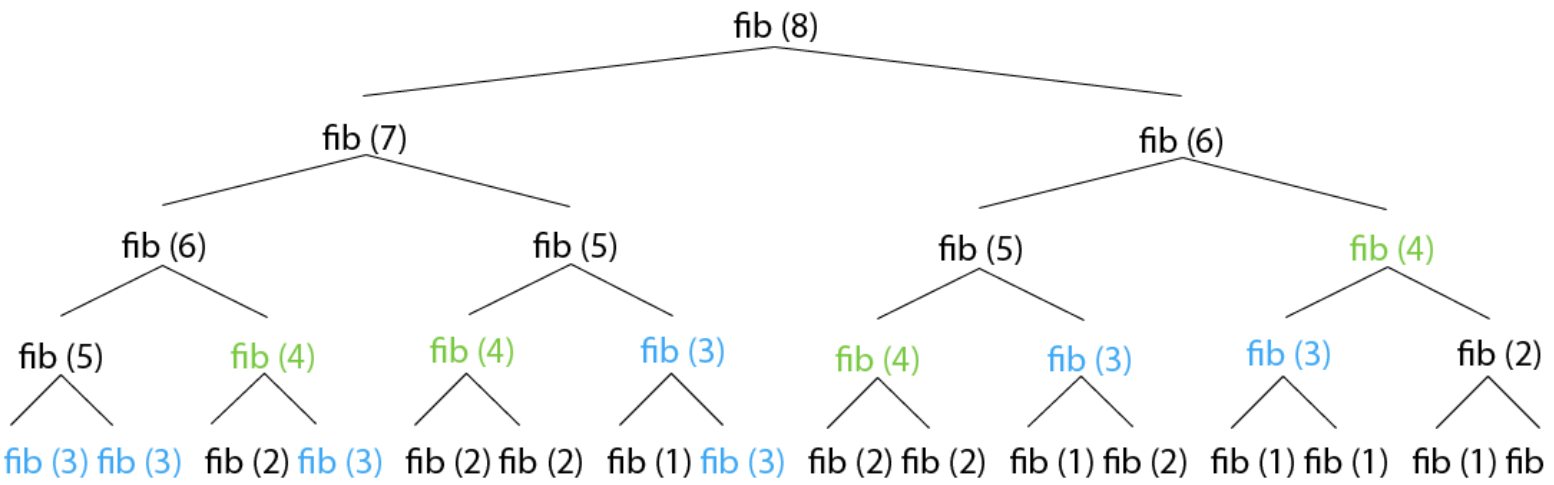


Figure: Recursive calls during computation of Fibonacci number

A single recursive call to fib (n) results in one recursive call to fib (n - 1), two recursive calls to fib (n - 2), three recursive calls to fib (n - 3), five recursive calls to fib (n - 4) and, in general, Fk-1 recursive calls to fib (n - k) We can avoid this unneeded repetition by writing down the conclusion of recursive calls and looking them up again if we need them later. This process is called memorization.

Here is the algorithm with memorization

MEMOFIB (n)

```
1 if (n < 2)
2 then return n
3 if (F[n] is undefined)
4 then F[n] ← MEMOFIB (n - 1) + MEMOFIB (n - 2)
5 return F[n]
```

If we trace through the recursive calls to MEMOFIB, we find that array F [] gets filled from bottom up. I.e., first F [2], then F [3], and so on, up to F[n]. We can replace recursion with a simple for-loop that just fills up the array F [] in that order

ITERFIB (n)

```
1 F [0] ← 0
2 F [1] ← 1
3 for i ← 2 to n
4 do
5 F[i] ← F [i - 1] + F [i - 2]
6 return F[n]
```

This algorithm clearly takes only O (n) time to compute Fn. By contrast, the original recursive algorithm takes O ($\emptyset^{n}$),$\emptyset$ = $\frac{1+\sqrt{5}}{2}$ = 1.618. ITERFIB conclude an exponential speedup over the original recursive algorithm.

Matrix Chain Multiplication

It is a Method under Dynamic Programming in which previous output is taken as input for next.

Here, Chain means one matrix's column is equal to the second matrix's row [always].

In general:

If $A = [a_{ij}]$ is a p x q matrix

$B = [b_{ij}]$ is a q x r matrix

$C = [c_{ij}]$ is a p x r matrix

Then

$$AB = C \text{ if } c_{ij} = \sum_{k=1}^{q} a_{ik}\ b_{kj}$$

Given following matrices {A_1,A_2,A_3,...A_n} and we have to perform the matrix multiplication, which can be accomplished by a series of matrix multiplications

A_1 xA_2 x,A_3 x.....x A_n

Matrix Multiplication operation is **associative** in nature rather commutative. By this, we mean that we have to follow the above matrix order for multiplication but we are free to **parenthesize** the above multiplication depending upon our need.

In general, for 1≤ i≤ p and 1≤ j ≤ r

$$C[i, j] = \sum_{k=1}^{q} A[i, k]B[k, j]$$

It can be observed that the total entries in matrix 'C' is 'pr' as the matrix is of dimension p x r Also each entry takes O (q) times to compute, thus the total time to compute all possible entries for the matrix 'C' which is a multiplication of 'A' and 'B' is proportional to the product of the dimension p q r.

It is also noticed that we can save the number of operations by reordering the parenthesis.

Example1: Let us have 3 matrices, A_1,A_2,A_3 of order (10 x 100), (100 x 5) and (5 x 50) respectively.

Three Matrices can be multiplied in two ways:

1. **A_1,(A_2,A_3):** First multiplying(A_2 and A_3) then multiplying and resultant withA_1.
2. **(A_1,A_2),A_3:** First multiplying(A_1 and A_2) then multiplying and resultant withA_3.

No of Scalar multiplication in Case 1 will be:

1. (100 x 5 x 50) + (10 x 100 x 50) = 25000 + 50000 = 75000

No of Scalar multiplication in Case 2 will be:

1. (100 x 10 x 5) + (10 x 5 x 50) = 5000 + 2500 = 7500

To find the best possible way to calculate the product, we could simply parenthesis the expression in every possible fashion and count each time how many scalar multiplication are required.

Matrix Chain Multiplication Problem can be stated as "find the optimal parenthesization of a chain of matrices to be multiplied such that the number of scalar multiplication is minimized".

Number of ways for parenthesizing the matrices:

There are very large numbers of ways of parenthesizing these matrices. If there are n items, there are (n-1) ways in which the outer most pair of parenthesis can place.

(A_1) (A_2,A_3,A_4,.....A_n)
Or (A_1,A_2) (A_3,A_4 .....A_n)
Or (A_1,A_2,A_3) (A_4 .....A_n)
.....

Or(A_1,A_2,A_3.....A_{n-1}) (A_n)

It can be observed that after splitting the kth matrices, we are left with two parenthesized sequence of matrices: one consist 'k' matrices and another consist 'n-k' matrices.

Now there are 'L' ways of parenthesizing the left sublist and 'R' ways of parenthesizing the right sublist then the Total will be L.R:

$$p(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} p(k)p(n-k) & \text{if } n \geq 2 \end{cases}$$

Also p (n) = c (n-1) where c (n) is the nth **Catalon number**

$$c(n) = \frac{1}{n+1}\binom{2n}{n}$$

On applying Stirling's formula we have

$c(n) = \Omega\left(\frac{4^n}{n^{1.5}}\right)$

Which shows that 4^n grows faster, as it is an exponential function, then $n^{1.5}$.

Development of Dynamic Programming Algorithm

1. Characterize the structure of an optimal solution.
2. Define the value of an optimal solution recursively.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct the optimal solution from the computed information.

Dynamic Programming Approach

Let $A_{i,j}$ be the result of multiplying matrices i through j. It can be seen that the dimension of $A_{i,j}$ is p_{i-1} x p_j matrix.

Dynamic Programming solution involves breaking up the problems into subproblems whose solution can be combined to solve the global problem.

At the greatest level of parenthesization, we multiply two matrices

$A_{1.....n}=A_{1....k}$ x $A_{k+1....n})$

Thus we are left with two questions:

- How to split the sequence of matrices?
- How to parenthesize the subsequence $A_{1.....k}$ and$A_{k+1.....n}$?

One possible answer to the first question for finding the best value of 'k' is to check all possible choices of 'k' and consider the best among them. But that it can be observed that checking all possibilities will lead to an exponential number of total possibilities. It can also be noticed that there exists only O (n^2) different sequence of matrices, in this way do not reach the exponential growth.

Step1: Structure of an optimal parenthesization: Our first step in the dynamic paradigm is to find the optimal substructure and then use it to construct an optimal solution to the problem from an optimal solution to subproblems.

Let $A_{i....j}$ where i≤ j denotes the matrix that results from evaluating the product

A_i A_{i+1}....A_j.

If i < j then any parenthesization of the product A_i A_{i+1} .....A_j must split that the product between A_k and A_{k+1} for some integer k in the range i ≤ k ≤ j. That is for some value of k, we first compute the matrices $A_{i.....k}$ & $A_{k+1....j}$ and then multiply them together to produce the final product $A_{i....j}$. The cost of computing $A_{i....k}$ plus the cost of computing $A_{k+1....j}$ plus the cost of multiplying them together is the cost of parenthesization.

Step 2: A Recursive Solution: Let m [i, j] be the minimum number of scalar multiplication needed to compute the matrix$A_{i....j}$.

If i=j the chain consist of just one matrix $A_{i....i}=A_i$ so no scalar multiplication are necessary to compute the product. Thus m [i, j] = 0 for i= 1, 2, 3....n.

If i<j we assume that to optimally parenthesize the product we split it between A_k and A_{k+1} where i≤ k ≤j. Then m [i,j] equals the minimum cost for computing the subproducts $A_{i....k}$ and $A_{k+1....j}$+ cost of multiplying them together. We know A_i has dimension p_{i-1} x p_i, so computing the product $A_{i....k}$ and $A_{k+1....j}$takes p_{i-1} p_k p_j scalar multiplication, we obtain

$m\ [i,j] = m\ [i,\ k] + m\ [k + 1,\ j] + p_{i-1}\ \ p_k\ p_j$

There are only (j-1) possible values for 'k' namely k = i, i+1.....j-1. Since the optimal parenthesization must use one of these values for 'k' we need only check them all to find the best.

So the minimum cost of parenthesizing the product A_i A_{i+1}.....A_j becomes

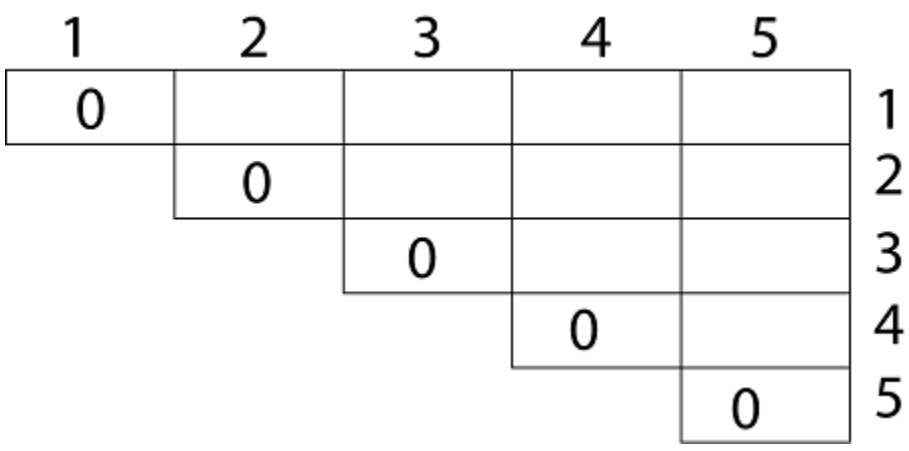
$$m\ [i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j}\{m\ [i,k] + m\ [k+1,j] + p_{i-1}\, p_k p_j\} & \text{if } i < j \end{cases}$$

To construct an optimal solution, let us define s [i,j] to be the value of 'k' at which we can split the product $A_i A_{i+1} \ldots A_j$ To obtain an optimal parenthesization i.e. s [i, j] = k such that

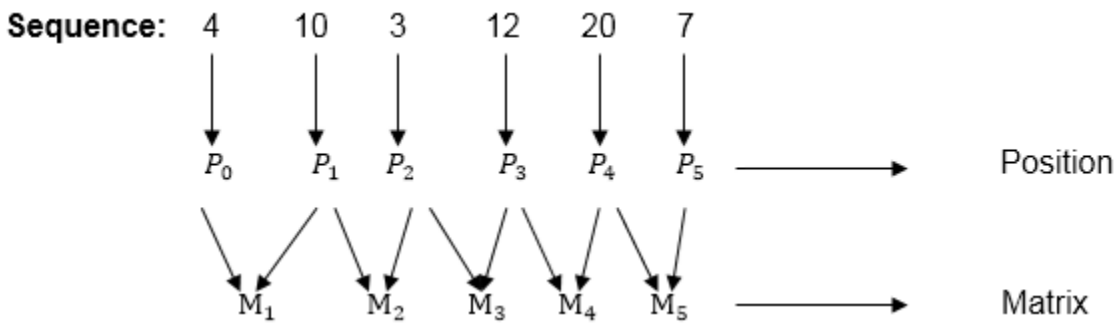
$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$$

Example of Matrix Chain Multiplication

Example: We are given the sequence {4, 10, 3, 12, 20, and 7}. The matrices have size 4 x 10, 10 x 3, 3 x 12, 12 x 20, 20 x 7. We need to compute M [i,j], 0 ≤ i, j≤ 5. We know M [i, i] = 0 for all i.



Let us proceed with working away from the diagonal. We compute the optimal solution for the product of 2 matrices.



Here P_0 to P_5 are Position and M_1 to M_5 are matrix of size (p_i to p_{i-1})

On the basis of sequence, we make a formula

For M_i ⟶ p [i] as column

p [i-1] as row

In Dynamic Programming, initialization of every method done by '0'.So we initialize it by '0'.It will sort out diagonally.

We have to sort out all the combination but the minimum output combination is taken into consideration.

Calculation of Product of 2 matrices:

1. $m(1,2) = m_1 \times m_2$
   $= 4 \times 10 \times 10 \times 3$
   $= 4 \times 10 \times 3 = 120$

2. $m(2,3) = m_2 \times m_3$
   $= 10 \times 3 \times 3 \times 12$
   $= 10 \times 3 \times 12 = 360$

3. $m(3,4) = m_3 \times m_4$
   $= 3 \times 12 \times 12 \times 20$
   $= 3 \times 12 \times 20 = 720$

4. $m(4,5) = m_4 \times m_5$
   $= 12 \times 20 \times 20 \times 7$
   $= 12 \times 20 \times 7 = 1680$

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

- We initialize the diagonal element with equal i,j value with '0'.
- After that second diagonal is sorted out and we get all the values corresponded to it

Now the third diagonal will be solved out in the same way.

Now product of 3 matrices:

`M [1, 3] = M1 M2 M3`

1. There are two cases by which we can solve this multiplication: (M_1 x M_2) + M_3, M_1+ (M_2x M_3)
2. After solving both cases we choose the case in which minimum output is there.

$$M[1,3] = \min \begin{cases} M[1,2] + M[3,3] + p_0 p_2 p_3 = 120 + 0 + 4.3.12 = 264 \\ M[1,1] + M[2,3] + p_0 p_1 p_3 = 0 + 360 + 4.10.12 = 840 \end{cases}$$

M [1, 3] =264

As Comparing both output **264** is minimum in both cases so we insert **264** in table and (M_1 x M_2) + M_3 this combination is chosen for the output making.

`M [2, 4] = M2 M3 M4`

1. There are two cases by which we can solve this multiplication: (M_2x M_3)+M_4, M_2+(M_3 x M_4)
2. After solving both cases we choose the case in which minimum output is there.

$$M[2,4] = \min \begin{cases} M[2,3] + M[4,4] + p_1 p_3 p_4 = 360 + 0 + 10.12.20 = 2760 \\ M[2,2] + M[3,4] + p_1 p_2 p_4 = 0 + 720 + 10.3.20 = 1320 \end{cases}$$

M [2, 4] = 1320

As Comparing both output **1320** is minimum in both cases so we insert **1320** in table and M_2+(M_3 x M_4) this combination is chosen for the output making.

`M [3, 5] = M3 M4 M5`

1. There are two cases by which we can solve this multiplication: (M_3 x M_4) + M_5, M_3+ (M_4xM_5)
2. After solving both cases we choose the case in which minimum output is there.

$$M[3,5] = \min \begin{cases} M[3,4] + M[5,5] + p_2 p_4 p_5 = 720 + 0 + 3.20.7 = 1140 \\ M[3,3] + M[4,5] + p_2 p_3 p_5 = 0 + 1680 + 3.12.7 = 1932 \end{cases}$$

`M [3, 5] = 1140`

As Comparing both output **1140** is minimum in both cases so we insert **1140** in table and (M_3 x M_4) + M_5this combination is chosen for the output making.

1	2	3	4	5	
0	120				1
	0	360			2
		0	720		3
			0	1680	4
				0	5

→

1	2	3	4	5	
0	120	264			1
	0	360	1320		2
		0	720	1140	3
			0	1680	4
				0	5

Now Product of 4 matrices:

`M [1, 4] = M1 M2 M3 M4`

There are three cases by which we can solve this multiplication:

1. (M_1 x M_2 x M_3) M_4
2. M_1 x(M_2 x M_3 x M_4)
3. (M_1 xM_2) x (M_3 x M_4)

After solving these cases we choose the case in which minimum output is there

$$M[1,4] = \min \begin{cases} M[1,3] + M[4,4] + p_0p_3p_4 = 264 + 0 + 4.12.20 = 1224 \\ M[1,2] + M[3,4] + p_0p_2p_4 = 120 + 720 + 4.3.20 = 1080 \\ M[1,1] + M[2,4] + p_0p_1p_4 = 0 + 1320 + 4.10.20 = 2120 \end{cases}$$

M [1, 4] =1080

As comparing the output of different cases then '**1080**' is minimum output, so we insert 1080 in the table and (M_1 xM_2) x (M_3 x M_4) combination is taken out in output making,

```
M [2, 5] = M2 M3 M4 M5
```

There are three cases by which we can solve this multiplication:

1. (M_2 x M_3 x M_4)x M_5
2. M_2 x(M_3 x M_4 x M_5)
3. (M_2 x M_3)x (M_4 x M_5)

After solving these cases we choose the case in which minimum output is there

$$M[2,5] = \min \begin{cases} M[2,4] + M[5,5] + p_1p_4p_5 = 1320 + 0 + 10.20.7 = 2720 \\ M[2,3] + M[4,5] + p_1p_3p_5 = 360 + 1680 + 10.12.7 = 2880 \\ M[2,2] + M[3,5] + p_1p_2p_5 = 0 + 1140 + 10.3.7 = 1350 \end{cases}$$

```
M [2, 5] = 1350
```

As comparing the output of different cases then '**1350**' is minimum output, so we insert 1350 in the table and M_2 x(M_3 x M_4 xM_5)combination is taken out in output making.

	1	2	3	4	5
1	0	120	264		
2		0	360	1320	
3			0	720	1140
4				0	1680
5					0

→

	1	2	3	4	5
1	0	120	264	1080	
2		0	360	1320	1350
3			0	720	1140
4				0	1680
5					0

Now Product of 5 matrices:

```
M [1, 5] = M1  M2 M3 M4 M5
```

There are five cases by which we can solve this multiplication:

1. (M_1 x M_2 xM_3 x M_4)x M_5
2. M_1 x(M_2 xM_3 x M_4 xM_5)
3. (M_1 x M_2 xM_3)x M_4 xM_5
4. M_1 x M_2x(M_3 x M_4 xM_5)

After solving these cases we choose the case in which minimum output is there

$$M[1,5] = \min \begin{cases} M[1,4] + M[5,5] + p_0p_4p_5 = 1080 + 0 + 4.20.7 = 1544 \\ M[1,3] + M[4,5] + p_0p_3p_5 = 264 + 1680 + 4.12.7 = 2016 \\ M[1,2] + M[3,5] + p_0p_2p_5 = 120 + 1140 + 4.3.7 = 1344 \\ M[1,1] + M[2,5] + p_0p_1p_5 = 0 + 1350 + 4.10.7 = 1630 \end{cases}$$

```
M [1, 5] = 1344
```

As comparing the output of different cases then '**1344**' is minimum output, so we insert 1344 in the table and M_1 x M_2 x(M_3 x M_4 x M_5)combination is taken out in output making.

Final Output is:

	1	2	3	4	5
1	0	120	264	1080	
2		0	360	1320	1350
3			0	720	1140
4				0	1680
5					0

→

	1	2	3	4	5
1	0	120	264	1080	1344
2		0	360	1320	1350
3			0	720	1140
4				0	1680
5					0

Step 3: Computing Optimal Costs: let us assume that matrix A_i has dimension p_{i-1}x p_i for i=1, 2, 3....n. The input is a sequence (p_0,p_1,.....p_n) where length [p] = n+1. The procedure uses an auxiliary table m [1....n, 1.....n] for storing m [i, j] costs an auxiliary table s [1.....n, 1.....n] that record which index of k achieved the optimal costs in computing m [i, j].

The algorithm first computes m [i, j] ← 0 for i=1, 2, 3.....n, the minimum costs for the chain of length 1.

Algorithm of Matrix Chain Multiplication

MATRIX-CHAIN-ORDER (p)

```
1. n   length[p]-1
2. for i ← 1 to n
3. do m [i, i] ← 0
4. for l ← 2 to n    // l is the chain length
5. do for i ← 1 to n-l + 1
6. do j ← i+ l -1
7. m[i,j] ←  ∞
8. for k  ← i to j-1
9. do q  ← m [i, k] + m [k + 1, j] + p_{i-1} p_k p_j
10. If q < m [i,j]
11. then m [i,j] ← q
12. s [i,j] ← k
13. return m and s.
```

We will use table s to construct an optimal solution.

Step 1: Constructing an Optimal Solution:

PRINT-OPTIMAL-PARENS (s, i, j)

```
1. if i=j
2. then print "A"
3. else print "("
4. PRINT-OPTIMAL-PARENS (s, i, s [i, j])
5. PRINT-OPTIMAL-PARENS (s, s [i, j] + 1, j)
6. print ")"
```

Analysis: There are three nested loops. Each loop executes a maximum n times.

1. l, length, O (n) iterations.
2. i, start, O (n) iterations.
3. k, split point, O (n) iterations

Body of loop constant complexity

Total Complexity is: O (n^3)

Algorithm with Explained Example

Question: P [7, 1, 5, 4, 2}

Solution: Here, P is the array of a dimension of matrices.

So here we will have 4 matrices:

A_{17x1} A_{21x5} A_{35x4} A_{44x2}

i.e.

First Matrix A_1 have dimension 7 x 1

Second Matrix A_2 have dimension 1 x 5

Third Matrix A_3 have dimension 5 x 4
Fourth Matrix A_4 have dimension 4 x 2

Let say,
From P = {7, 1, 5, 4, 2} - (Given)
And P is the Position
p_0 = 7, p_1 =1, p_2 = 5, p_3 = 4, p_4=2.
Length of array P = number of elements in P
∴length (p)= 5
From step 3
Follow the steps in Algorithm in Sequence
According to Step 1 of Algorithm Matrix-Chain-Order

Step 1:

n ← length [p]-1
Where n is the total number of elements
And length [p] = 5
∴ n = 5 - 1 = 4

n = 4

Now we construct two tables m and s.
Table m has dimension [1.....n, 1.......n]
Table s has dimension [1.....n-1, 2.......n]

	1	2	3	4
1	0	35 ∞	48 ∞	42
2		0	20 ∞	28 ∞
3			0	
4				0

m-Table
[1....n, 1.....n]

	2	3	4
1	1	1	1
2		2	3
3			3

n-Table
[1....n-1, 2.....n]

Now, according to step 2 of Algorithm

1. **for** i ← 1 to n
2. **this** means: **for** i ← 1 to 4 (because n =4)
3. **for** i=1
4. m [i, i]=0
5. m [1, 1]=0
6. Similarly **for** i = 2, 3, 4
7. m [2, 2] = m [3,3] = m [4,4] = 0
8. i.e. fill all the diagonal entries "0" in the table m
9. Now,
10. l ← 2 to n
11. l ← 2 to 4 (because n =4)

Case 1:

1. When l - 2

```
for (i ← 1 to n - l + 1)
```

i ← 1 to 4 - 2 + 1

i ← 1 to 3

When i = 1

do j ← i + l - 1

j ← 1 + 2 - 1

j ← 2

i.e. j = 2

Now, m [i, j] ← ∞

i.e. m [1,2] ← ∞

Put ∞ in m [1, 2] table

for k ← i to j-1

k ← 1 to 2 - 1

k ← 1 to 1

k = 1

Now q ← m [i, k] + m [k + 1, j] + p_{i-1} p_k p_j

for l = 2

i = 1

j =2

k = 1

q ← m [1,1] + m [2,2] + p_0x p_1x p_2

and m [1,1] = 0

for i ← 1 to 4

∴ q ← 0 + 0 + 7 x 1 x 5

q ← 35

We have m [i, j] = m [1, 2] = ∞

Comparing q with m [1, 2]

q < m [i, j]

i.e. **35 < m [1, 2]**

35 < ∞

True

then, **m [1, 2] ← 35 (∴ m [i,j] ← q)**

s [1, 2] ← k

and the value of **k = 1**

s [1,2] ← 1

Insert "1" at dimension s [1, 2] in table s. And 35 at m [1, 2]

2. l remains 2

L = 2

i ← 1 to n - l + 1

i ← 1 to 4 - 2 + 1

i ← 1 to 3

for i = 1 done before

Now value of i becomes 2

i = 2

j ← i + l - 1

j ← 2 + 2 - 1

j ← 3

j = 3

m [i , j] ← ∞

i.e. **m [2,3] ← ∞**

Initially insert ∞ at m [2, 3]

Now, for k ← i to j - 1

k ← 2 to 3 - 1

k ← 2 to 2

i.e. k =2

Now, q ← m [i, k] + m [k + 1, j] + p_{i-1} p_k p_j

For l =2

i = 2

j = 3

k = 2

q ← m [2, 2] + m [3, 3] + p_1x p_2 x p_3

q ← 0 + 0 + 1 x 5 x 4

q ← 20

Compare q with m [i ,j]

If q < m [i,j]

i.e. 20 < m [2, 3]

20 < ∞

True

Then m [i,j] ← q

m [2, 3] ← 20

and s [2, 3] ← k

and k = 2

s [2,3] ← 2

3. Now i become 3

i = 3

l = 2

j ← i + l - 1

j ← 3 + 2 - 1

j ← 4

j = 4

Now, m [i, j] ← ∞

m [3,4] ← ∞

Insert ∞ at m [3, 4]

for k ← i to j - 1

k ← 3 to 4 - 1

k ← 3 to 3

i.e. k = 3

Now, q ← m [i, k] + m [k + 1, j] + p_{i-1} p_k p_j

i = 3

l = 2

j = 4

k = 3

q ← m [3, 3] + m [4,4] + p_2 x p_3 x p_4

q ← 0 + 0 + 5 x 2 x 4

q 40

Compare q with m [i, j]

If q < m [i, j]

40 < m [3, 4]

40 < ∞

True

Then, m [i,j] ← q

m [3,4] ← 40

and s [3,4] ← k

s [3,4] ← 3

Case 2: l becomes 3

L = 3

for i = 1 to n - l + 1

i = 1 to 4 - 3 + 1

i = 1 to 2

When i = 1

j ← i + l - 1

j ← 1 + 3 - 1

j ← 3

j = 3

Now, m [i,j] ← ∞

m [1, 3] ← ∞

for k ← i to j - 1

k ← 1 to 3 - 1

k ← 1 to 2

Now we compare the value for both k=1 and k = 2. The minimum of two will be placed in m [i,j] or s [i,j] respectively.

(A) When k = 1	**(B)** When k =2
L =3,i=1,j=3	L =3,i=1,j=3
q ⟵ m [i, k] + m [k + 1, j] + p_{i-1} p_k p_j	q ⟵ m [i, k] + m [k + 1, j] + p_{i-1} p_k p_j
q ⟵ m [1,1] + m [2,3] + p_0x p_1x p_3	q ⟵ m [1,2] + m [3,3] +p_0x p_2x p_3
q ⟵ 0 + 20 + 7 x 1x 4	q ⟵ 35 + 0 + 7 x 5 x 3
q ⟵ 48	**q ⟵ 140**

Now from above

Value of q become minimum for **k=1**

∴ m [i,j] ← q

m [1,3] ← 48

Also m [i,j] > q

i.e. 48 < ∞

∴ m [i , j] ← q

m [i, j] ← 48

and s [i,j] ← k

i.e. m [1,3] ← 48

s [1, 3] ← 1

Now i become 2

i = 2

l = 3

then j ← i + l -1

j ← 2 + 3 - 1

j ← 4

j = 4

so m [i,j] ← ∞

m [2,4] ← ∞

Insert initially ∞ at m [2, 4]

for k ← i to j-1

k ← 2 to 4 - 1

k ← 2 to 3

Here, also find the minimum value of m [i,j] for two values of k = 2 and k =3

(A) When k =2	**(B)** When k = 3
i=2,l=3, j=4	i=2,l=3, j=4
q ⟵ m [i, k] + m [k + 1, j] + $p_{i-1}\ p_k\ p_j$	q ⟵ m [i, k] + m [k + 1, j] + $p_{i-1}\ p_k\ p_j$
q ⟵ m [2,2] + m [3,4] + $p_1 x\ p_2 x p_4$	q ⟵ m [2,3] + m [4,4] +$p_1 x p_3 x\ p_4$
q ⟵ 0 + 40 +7 x 5 x 2	q ⟵ 20 + 0 + 1 x 4 x 2
q ⟵ 110	**q ⟵ 28**

1. But 28 < ∞
2. So m [i,j] ← q
3. And q ← 28
4. m [2, 4] ← 28
5. and s [2, 4] ← 3
6. e. It means in s table at s [2,4] insert 3 and at m [2,4] insert 28.

Case 3: l becomes 4

L = 4

For i ← 1 to n-l + 1

i ← 1 to 4 - 4 + 1

i ← 1

i = 1

do j ← i + l - 1

j ← 1 + 4 - 1

j ← 4

j = 4

Now m [i,j] ← ∞

m [1,4] ← ∞

for k ← i to j -1

k ← 1 to 4 - 1

k ← 1 to 3

When k = 1

q ← m [i, k] + m [k + 1, j] + $p_{i-1}\ p_k\ p_j$

q ← m [1,1] + m [2,4] + $p_0 x p_4 x\ p_1$

q ← 0 + 28 + 7 x 2 x 1

q ← 42

Compare q and m [i, j]

m [i,j] was ∞
i.e. m [1,4]
if q < m [1,4]
   42< ∞
    True
Then m [i,j] ← q
        m [1,4] ← 42
and s [1,4] 1   ? k =1
When k = 2
       L = 4, i=1, j = 4
q ← m [i, k] + m [k + 1, j] + p_{i-1} p_k p_j
q ← m [1, 2] + m [3,4] + p_0 xp_2 xp_4
q ← 35 + 40 + 7 x 5 x 2
q ← 145
Compare q and m [i,j]
Now m [i, j]
      i.e. m [1,4] contains 42.
So if q < m [1, 4]
But 145 less than or not equal to m [1, 4]
    So 145 less than or not equal to 42.
So no change occurs.
When k = 3
       l = 4
       i = 1
       j = 4
q ← m [i, k] + m [k + 1, j] + p_{i-1} p_k p_j
q ← m [1, 3] + m [4,4] + p_0 xp_3 x p_4
q ← 48 + 0 + 7 x 4 x 2
q ← 114
Again q less than or not equal to m [i, j]
       i.e. 114 less than or not equal to m [1, 4]
            114 less than or not equal to 42

So no change occurs. So the value of m [1, 4] remains 42. And value of s [1, 4] = 1

Now we will make use of only s table to get an optimal solution.

Use of step 4 Algorithm

Initial call of step 4 is (s, 1, n)

Where i=1

j = n and n= 4

i≠ j

else part

print "("

Print-optimal-Parens (s, i, s [i, j])

here i==j yes

print "A_1"

Print-optimal-Parens (s, 1, 4)

Print-optimal-Parens (s, s [i,j] + 1,j]

Print-optimal-Parens (s, 2, 4)

print "("

Print-optimal (s, 2, 3)

Print ")"

Print-optimal (s, 4, 4)

Print ")"

Starting from the beginning we are parenthesizing Matrix Chain Multiplication:

$(A_1)\ ((A_2\ A_3)A_4))$

Longest Common Sequence (LCS)

A subsequence of a given sequence is just the given sequence with some elements left out.

Given two sequences X and Y, we say that the sequence Z is a common sequence of X and Y if Z is a subsequence of both X and Y.

In the longest common subsequence problem, we are given two sequences $X = (x_1\ x_2....x_m)$ and $Y = (y_1\ y_2\ y_n)$ and wish to find a maximum length common subsequence of X and Y. LCS Problem can be solved using dynamic programming.

Characteristics of Longest Common Sequence

A brute-force approach we find all the subsequences of X and check each subsequence to see if it is also a subsequence of Y, this approach requires exponential time making it impractical for the long sequence.

Given a sequence $X = (x_1\ x_2.....x_m)$ we define the ith prefix of X for i=0, 1, and 2...m as $X_i = (x_1\ x_2.....x_i)$. For example: if X = (A, B, C, B, C, A, B, C) then X_4= (A, B, C, B)

Optimal Substructure of an LCS: Let $X = (x_1\ x_2....x_m)$ and $Y = (y_1\ y_2.....)\ y_n)$ be the sequences and let $Z = (z_1\ z_2.....z_k)$ be any LCS of X and Y.

- If $x_m = y_n$, then $z_k=x\_m=y_n$ and Z_{k-1} is an LCS of $X_{m-1}$and Y_{n-1}
- If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of $X_{m-1}$and Y.
- If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}

Step 2: Recursive Solution: LCS has overlapping subproblems property because to find LCS of X and Y, we may need to find the LCS of X_{m-1} and Y_{n-1}. If $x_m \neq y_n$, then we must solve two subproblems finding an LCS of X and Y_{n-1}.Whenever of these LCS's longer is an LCS of x and y. But each of these subproblems has the subproblems of finding the LCS of X_{m-1} and Y_{n-1}.

Let c [i,j] be the length of LCS of the sequence $X_i$and Y_j.If either i=0 and j =0, one of the sequences has length 0, so the LCS has length 0. The optimal substructure of the LCS problem given the recurrence formula

$$c\ [i,\ j] = \begin{cases} 0 & \text{if } i = 0 \quad \text{or} \quad j = 0 \\ c\ [i-1, j-1]1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Step3: Computing the length of an LCS: let two sequences $X = (x_1\ x_2.....x_m)$ and $Y = (y_1\ y_2..... y_n)$ as inputs. It stores the c [i,j] values in the table c [0.....m,0.....n].Table b [1.....m, 1.....n] is maintained which help us to construct an optimal solution. c [m, n] contains the length of an LCS of X,Y.

Algorithm of Longest Common Sequence

LCS-LENGTH (X, Y)

```
1. m ← length [X]
2. n ← length [Y]
3. for i ← 1 to m
4. do c [i,0] ← 0
5. for j ← 0 to m
6. do c [0,j] ← 0
7. for i ← 1 to m
8. do for j ← 1 to n
9. do if xi= yj
10. then c [i,j] ← c [i-1,j-1] + 1
11. b [i,j] ← "↖"
12. else if c[i-1,j] ≥ c[i,j-1]
13. then c [i,j] ← c [i-1,j]
14. b [i,j] ← "↑"
15. else c [i,j] ← c [i,j-1]
16. b [i,j] ← "← "
17. return c and b.
```

Example of Longest Common Sequence

Example: Given two sequences X [1...m] and Y [1.....n]. Find the longest common subsequences to both.

x: A	B	C	B	D	A	B
y: B	D	C	A	B	A

here X = (A,B,C,B,D,A,B) and Y = (B,D,C,A,B,A)

m = length [X] and n = length [Y]

m = 7 and n = 6

Here x_1= x [1] = A y_1= y [1] = B

x_2= B y_2= D

x_3= C y_3= C

x_4= B y_4= A

x_5= D y_5= B

x_6= A y_6= A

x_7= B

Now fill the values of c [i, j] in m x n table

Initially, for i=1 to 7 c [i, 0] = 0

For j = 0 to 6 c [0, j] = 0

That is:

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	X_i	0	0	0	0	0	0	0
1	A	0						
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Now for i=1 and j = 1

x_1 and y_1 we get $x_1 \neq y_1$ i.e. A ≠ B

And c [i-1,j] = c [0, 1] = 0

c [i, j-1] = c [1,0] = 0

That is, c [i-1,j]= c [i, j-1] so c [1, 1] = 0 and b [1, 1] = ' ↑ '

Now for i=1 and j = 2

x_1 and y_2 we get $x_1 \neq y_2$ i.e. A ≠ D

c [i-1,j] = c [0, 2] = 0

c [i, j-1] = c [1,1] = 0

That is, c [i-1,j]= c [i, j-1] and c [1, 2] = 0 b [1, 2] = ' ↑ '

Now for i=1 and j = 3

x_1 and y_3 we get $x_1 \neq y_3$ i.e. A ≠ C

c [i-1,j] = c [0, 3] = 0

c [i, j-1] = c [1,2] = 0

so c [1,3] = 0 b [1,3] = ' ↑ '

Now for i=1 and j = 4

x_1 and y_4 we get. $x_1=y_4$ i.e A = A

c [1,4] = c [1-1,4-1] + 1

= c [0, 3] + 1

= 0 + 1 = 1

c [1,4] = 1

b [1,4] = ' ↖ '

Now for i=1 and j = 5

x_1 and y_5 we get $x_1 \neq y_5$

c [i-1,j] = c [0, 5] = 0

c [i, j-1] = c [1,4] = 1

Thus c [i, j-1] > c [i-1,j] i.e. c [1, 5] = c [i, j-1] = 1. So b [1, 5] = '←'

Now for i=1 and j = 6

x_1 and y_6 we get $x_1=y_6$

c [1, 6] = c [1-1,6-1] + 1

= c [0, 5] + 1 = 0 + 1 = 1

c [1,6] = 1

b [1,6] = ' ↖ '

i		j	0	1	2	3	4	5	6
			y_i	B	D	C	A	B	A
0	X_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0						
3	C		0						
4	B		0						
5	D		0						
6	A		0						
7	B		0						

Now for i=2 and j = 1

We get x_2 and y_1 B = B i.e. $x_2= y_1$

c [2,1] = c [2-1,1-1] + 1

= c [1, 0] + 1

= 0 + 1 = 1

c [2, 1] = 1 and b [2, 1] = ' ↖ '

Similarly, we fill the all values of c [i, j] and we get

i		j	0	1	2	3	4	5	6
			y_i	B	D	C	A	B	A
0	X_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Step 4: Constructing an LCS: The initial call is PRINT-LCS (b, X, X.length, Y.length)

PRINT-LCS (b, x, i, j)

```
1. if i=0 or j=0
2. then return
3. if b [i,j] = ' ↖ '
4. then PRINT-LCS (b,x,i-1,j-1)
5. print x_i
6. else if b [i,j] = ' ↑ '
7. then PRINT-LCS (b,X,i-1,j)
8. else PRINT-LCS (b,X,i,j-1)
```

Example: Determine the LCS of (1,0,0,1,0,1,0,1) and (0,1,0,1,1,0,1,1,0).

Solution: let X = (1,0,0,1,0,1,0,1) and Y = (0,1,0,1,1,0,1,1,0).

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \quad \text{or} \quad j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \quad \text{and} \quad x_i = y_i \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_i \end{cases}$$

We are looking for c [8, 9]. The following table is built.

			x =(1,0,0,1,0,1,0,1)					y =(0,1,0,1,1,0,1,1,0)			
i	j	0	1	2	3	4	5	6	7	8	9
		y_j	0	1	0	1	1	0	1	1	0
0	X_i	0	0	0	0	0	0	0	0	0	0
1	1	0	↑ 0	↖ (1)	↑ 1	↖ 1	← 1	↖ 1	↖ 0	↖ 1	← 1
2	0	0	↖ 1	↑ 1	↖ (2)	← 2	← 2	↖ 2	← 2	← 2	↖ 2
3	0	0	↖ 1	↑ 1	↖ 2	↑ 2	↑ 2	↖ (3)	← 3	← 3	↖ 3
4	1	0	↑ 1	↖ 2	↑ 2	↖ 3	↖ 3	↑ 3	↖ (4)	↖ 4	← 4
5	0	0	↖ 1	↑ 2	↖ 3	↑ 3	↑ 3	↖ 4	↑ 4	↑ 4	↖ 5
6	1	0	↑ 1	↖ 2	↑ 3	↖ 4	↖ 4	↑ 4	↖ 5	↖ (5)	↑ 5
7	0	0	↖ 1	↑ 2	↖ 3	↑ 4	↑ 4	↖ 5	↑ 5	↑ 5	↖ (6)
8	1	0	↑ 1	↖ 2	↑ 3	↖ 4	↖ 5	↑ 5	↖ 6	↖ 6	↑ 6

From the table we can deduct that LCS = 6. There are several such sequences, for instance (1,0,0,1,1,0) (0,1,0,1,0,1) and (0,0,1,1,0,1)

0/1 Knapsack Problem: Dynamic Programming Approach:

Knapsack Problem:

Knapsack is basically means bag. A bag of given capacity.

We want to pack n items in your luggage.

- The ith item is worth v_i dollars and weight w_i pounds.
- Take as valuable a load as possible, but cannot exceed W pounds.
- v_i w_i W are integers.

1. W ≤ capacity
2. Value ← Max

Input:

- Knapsack of capacity
- List (Array) of weight and their corresponding value.

Output: To maximize profit and minimize weight in capacity.

The knapsack problem where we have to pack the knapsack with maximum value in such a manner that the total weight of the items should not be greater than the capacity of the knapsack.

Knapsack problem can be further divided into two parts:

1. Fractional Knapsack: Fractional knapsack problem can be solved by **Greedy Strategy** where as 0 /1 problem is not.

It cannot be solved by **Dynamic Programming Approach.**

0/1 Knapsack Problem:

In this item cannot be broken which means thief should take the item as a whole or should leave it. That's why it is called **0/1 knapsack Problem**.

- Each item is taken or not taken.
- Cannot take a fractional amount of an item taken or take an item more than once.
- It cannot be solved by the Greedy Approach because it is enable to fill the knapsack to capacity.
- **Greedy Approach** doesn't ensure an Optimal Solution.

Example of 0/1 Knapsack Problem:

Example: The maximum weight the knapsack can hold is W is 11. There are five items to choose from. Their weights and values are presented in the following table:

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$												
$w_2 = 2\ v_2 = 6$												
$w_3 = 5\ v_3 = 18$												
$w_4 = 6\ v_4 = 22$												
$w_5 = 7\ v_5 = 28$												

The [i, j] entry here will be V [i, j], the best value obtainable using the first "i" rows of items if the maximum capacity were j. We begin by initialization and first row.

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0											
$w_3 = 5\ v_3 = 18$	0											
$w_4 = 6\ v_4 = 22$	0											
$w_5 = 7\ v_5 = 28$	0											

$$V\ [i, j] = \max\ \{V\ [i - 1, j],\ v_i + V\ [i - 1, j - w_i]$$

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0											
$w_4 = 6\ v_4 = 22$	0											
$w_5 = 7\ v_5 = 28$	0											

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6\ v_4 = 22$	0											
$w_5 = 7\ v_5 = 28$	0											

The value of V [3, 7] was computed as follows:

$$\begin{aligned} V\ [3, 7] &= \max\ \{V\ [3 - 1, 7],\ v_3 + V\ [3 - 1, 7 - w_3] \\ &= \max\ \{V\ [2, 7],\ 18 + V\ [2, 7 - 5]\} \\ &= \max\ \{7,\ 18 + 6\} \\ &= 24 \end{aligned}$$

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6\ v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7\ v_5 = 28$	0											

Finally, the output is:

Weight Limit (i):	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1\ v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2\ v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5\ v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
$w_4 = 6\ v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
$w_5 = 7\ v_5 = 28$	0	1	6	7	7	18	22	28	29	34	35	40

The maximum value of items in the knapsack is 40, the bottom-right entry). The dynamic programming approach can now be coded as the following algorithm:

Algorithm of Knapsack Problem

KNAPSACK (n, W)

```
1. for w = 0, W
2. do V [0,w] ← 0
3. for i=0, n
4. do V [i, 0] ← 0
5. for w = 0, W
6. do if (wi≤ w & vi + V [i-1, w -  wi]> V [i -1,W])
7. then V [i, W] ← vi + V [i - 1, w - wi]
8. else V [i, W] ← V [i - 1, w]
```

DUTCH NATIONAL FLAG

Dutch National Flag (DNF) - It is a programming problem proposed by Edsger Dijkstra. The flag of the Netherlands consists of three colors: white, red, and blue. The task is to randomly arrange balls of white, red, and blue in such a way that balls of the same color are placed together. For DNF (Dutch National Flag), we sort an array of 0, 1, and 2's in linear time that does not consume any extra space. We have to keep in mind that this algorithm can be implemented only on an array that has three unique elements.

ALGORITHM -

- Take three-pointers, namely - low, mid, high.
- We use low and mid pointers at the start, and the high pointer will point at the end of the given array.

CASES :

- If array [mid] =0, then swap array [mid] with array [low] and increment both pointers once.
- If array [mid] = 1, then no swapping is required. Increment mid pointer once.
- If array [mid] = 2, then we swap array [mid] with array [high] and decrement the high pointer once.

CODE -

(IN C LANGUAGE)

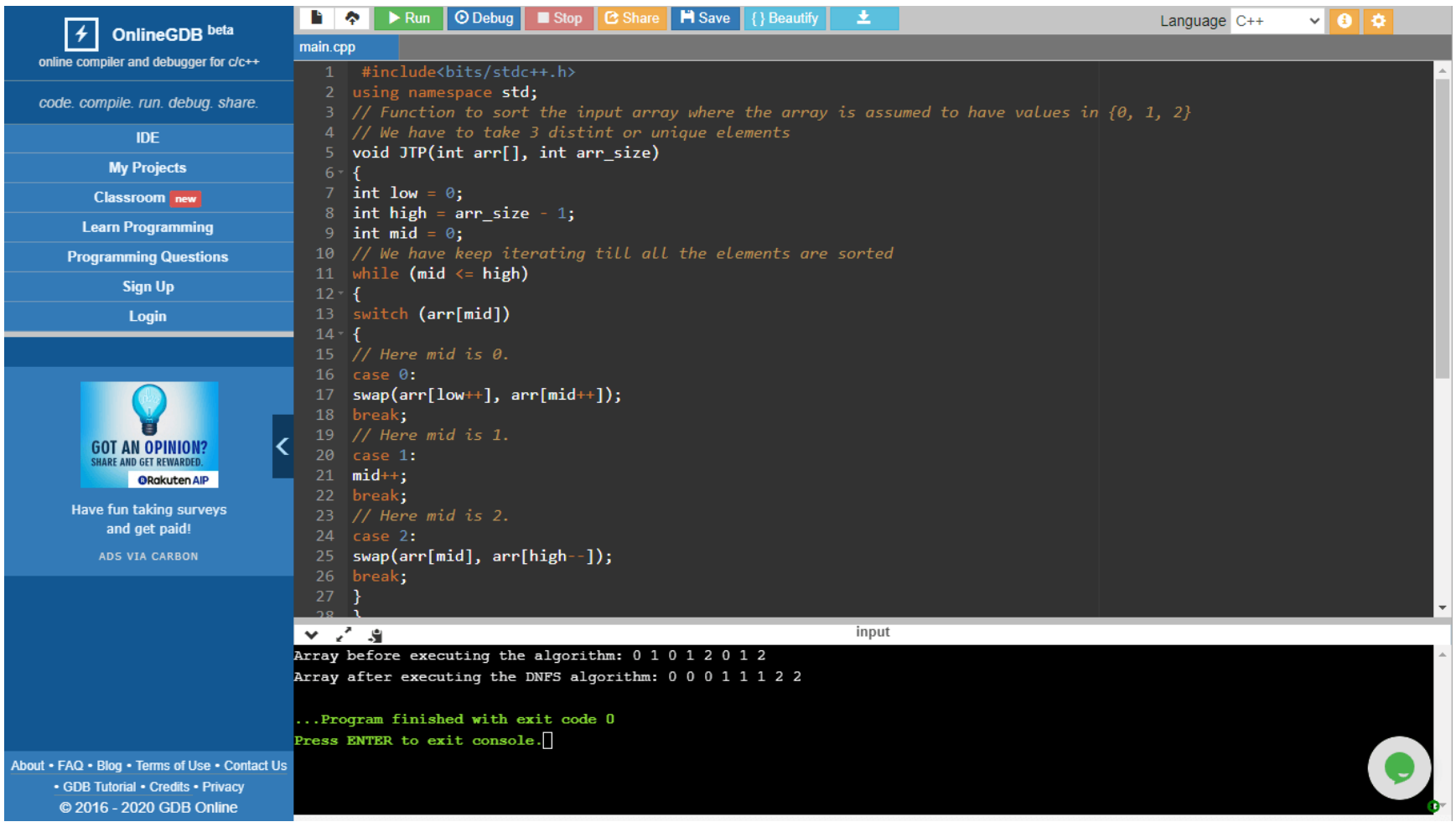
```
1. #include<bits/stdc++.h>
2. using namespace std;
3. // Function to sort the input array where the array is assumed to have values in {0, 1, 2}
4. // We have to take 3 distint or unique elements
5. void JTP(int arr[], int arr_size)
6. {
7. int low = 0;
8. int high = arr_size - 1;
9. int mid = 0;
```

```
// We have keep iterating till all the elements are sorted
while (mid <= high)
{
switch (arr[mid])
{
// Here mid is 0.
case 0:
swap(arr[low++], arr[mid++]);
break;
// Here mid is 1.
case 1:
mid++;
break;
// Here mid is 2.
case 2:
swap(arr[mid], arr[high--]);
break;
}
}
// Now, we write the function to print array arr[]
void printArray(int arr[], int arr_size)
{
// To iterate and print every element, we follow these steps
for (int i = 0; i < arr_size; i++)
cout << arr[i] << " ";
}
//Main Code
int main()
{
int arr[] = {0,1,0,1,2,0,1,2};
int n = sizeof(arr)/sizeof(arr[0]);
cout << "Array before executing the algorithm: ";
printArray(arr, n);
JTP(arr, n);
cout << "\nArray after executing the DNFS algorithm: ";
printArray(arr, n);
return 0;
}
```

OUTPUT -

Array before executing the algorithm: 0 1 0 1 2 0 1 2

Array after executing the DNFS algorithm: 0 0 0 1 1 1 2 2



CODE -

(IN JAVA)

```java
import java.io.*;
class DNF {
static void JTP(int arr[], int arr_size)
{
int low = 0;
int high = arr_size - 1;
int mid = 0, temp=0; // We use temporary variable for swapping
while (mid <= high)
{
switch (arr[mid])
{
case 0: // Here mid pointer points is at 0.
{
temp = arr[low];
arr[low] = arr[mid];
arr[mid] = temp;
low++;
mid++;
break;
}
case 1: // Here mid pointer points is at 1.
mid++;
break;
case 2: // Here mid pointer points is at 2.
{
temp = arr[mid];
arr[mid] = arr[high];
arr[high] = temp;
high--;
break;
}
}
}
```

```
35. // Now we have to call function to print array arr[]
36. static void printArray(int arr[], int arr_size)
37. {
38. int i;
39. for (i = 0; i < arr_size; i++)
40. System.out.print(arr[i]+" ");
41. System.out.println("");
42. }
43. //Now we use driver function to check for above functions
44. public static void main (String[] arguments)
45. {
46. int arr[] = {0, 1, 0, 1, 2, 0, 1, 2};
47. int arr_size = arr.length;
48. System.out.println("Array before executing the DNFS algorithm : ");
49. printArray(arr, arr_size);
50. JTP(arr, arr_size);
51. System.out.println("\nArray after executing the DNFS algorithm : ");
52. printArray(arr, arr_size);
53. }
54. }
```

OUTPUT -

Array before executing the DNFS algorithm : 0 1 0 1 2 0 1 2

Array after executing the DNFS algorithm : 0 0 0 1 1 1 2 2

C++ Java Python

```
1  import java.io.*;
2  class DNF {
3  static void JTP(int arr[], int arr_size)
4  {
5  int low = 0;
6  int high = arr_size - 1;
7  int mid = 0, temp=0; // We use temporary variable for swapping
8  while (mid <= high)
9  {
10 switch (arr[mid])
11 {
12 case 0: // Here mid pointer points is at 0.
13 {
14 temp = arr[low];
15 arr[low] = arr[mid];
16 arr[mid] = temp;
17 low++;
18 mid++;
19 break;
20 }
```

RUN

Close

```
Output                                                    2.741s

Array before executing the DNFS algorithm :
0 1 0 1 2 0 1 2

Array after executing the DNFS algorithm :
0 0 0 1 1 1 2 2
```

CODE -

(IN PYTHON)

```
1.  def JTP( arr, arr_size):
2.    low = 0
3.    high = arr_size - 1
4.    mid = 0
5.    while mid <= high:
6.      if arr[mid] == 0:
7.        arr[low],arr[mid] = arr[mid],arr[low]
8.        low = low + 1
```

```
9.        mid = mid + 1
10.     elif arr[mid] == 1:
11.       mid = mid + 1
12.     else:
13.       arr[mid],arr[high] = arr[high],arr[mid]
14.       high = high - 1
15.   return arr
16.
17. # Function to print array
18. def printArray(arr):
19.     for k in arr:
20.         print k,
21.     print
22.
23. # Driver Program
24. arr = [0, 1, 0, 1, 1, 2, 0, 1, 2]
25. arr_size = len(arr)
26. print " Array before executing the algorithm: ",
27. printArray(arr)
28. arr = JTP(arr, arr_size)
29. print "Array after executing the DNFS algorithm: ",
30. printArray(arr)
```

OUTPUT -

Array before executing the algorithm: 0 1 0 1 1 2 0 1 2

Array after executing the DNFS algorithm: 0 0 0 1 1 1 1 2 2

C++ Java Python

```
def JTP( arr, arr_size):
  low = 0
  high = arr_size - 1
  mid = 0
  while mid <= high:
    if arr[mid] == 0:
      arr[low],arr[mid] = arr[mid],arr[low]
      low = low + 1
      mid = mid + 1
    elif arr[mid] == 1:
      mid = mid + 1
    else:
      arr[mid],arr[high] = arr[high],arr[mid]
      high = high - 1
  return arr

# Function to print array
def printArray(arr):
    for k in arr:
        print k,
    print
```

RUN

Close

Output 0.649s

```
 Array before executing the algorithm:  0 1 0 1 1 2 0 1 2
Array after executing the DNFS algorithm:  0 0 0 1 1 1 1 2 2
```

Longest Palindrome Subsequence

It is a sequence of characters in a string that can be spelled and read the same both ways, forward and backward. Words like civic, redivider, deified, radar, level, madam, rotor, refer, kayak, racecar, and reviver. But in palindromic subsequence, a sequence can but not necessarily appear in the same relative order, but the chance of being necessarily contiguous and palindromic in nature is negligible.

Dynamic Programming Solution -

Example - We have been given a sequence as "BDBADBDCBDCADB." Then the longest palindrome will be eleven - "BDABDCDBADB", this is the longest palindromic subsequence here. 'BBABB,' 'DAD,' 'BBDBB' and many more are also

palindromic subsequences of the given sequence, but they are not the longest. In simpler words, sequences generate subsequences, then we compare their length and find which palindromic subsequence is the longest.

We follow these steps to achieve the most extended palindrome sequence using Dynamic Programming Solution -

First, we reverse the sequence.

Then we use the LCS algorithm (Longest Common Subsequence) to find the longest common subsequence among the original sequence and reversed sequence. Here original LCS and reverse LCS are a function that returns the longest common subsequence between the pair of strings, now the answer from LCS will be the longest palindromic subsequence.

Let LP (a, b) = Length of longest palindromic subsequence in array Z from index a to b

LP (a, b) = LP (a+1, b-1) + 2: if Z [a] = X [b]

= max [LP (a+1, b), LP (a, b-1)]: if Z [a]! = Z [b]

= 1 if a = b

= 1 if a = b - 1 and Z [a]! = Z [b]

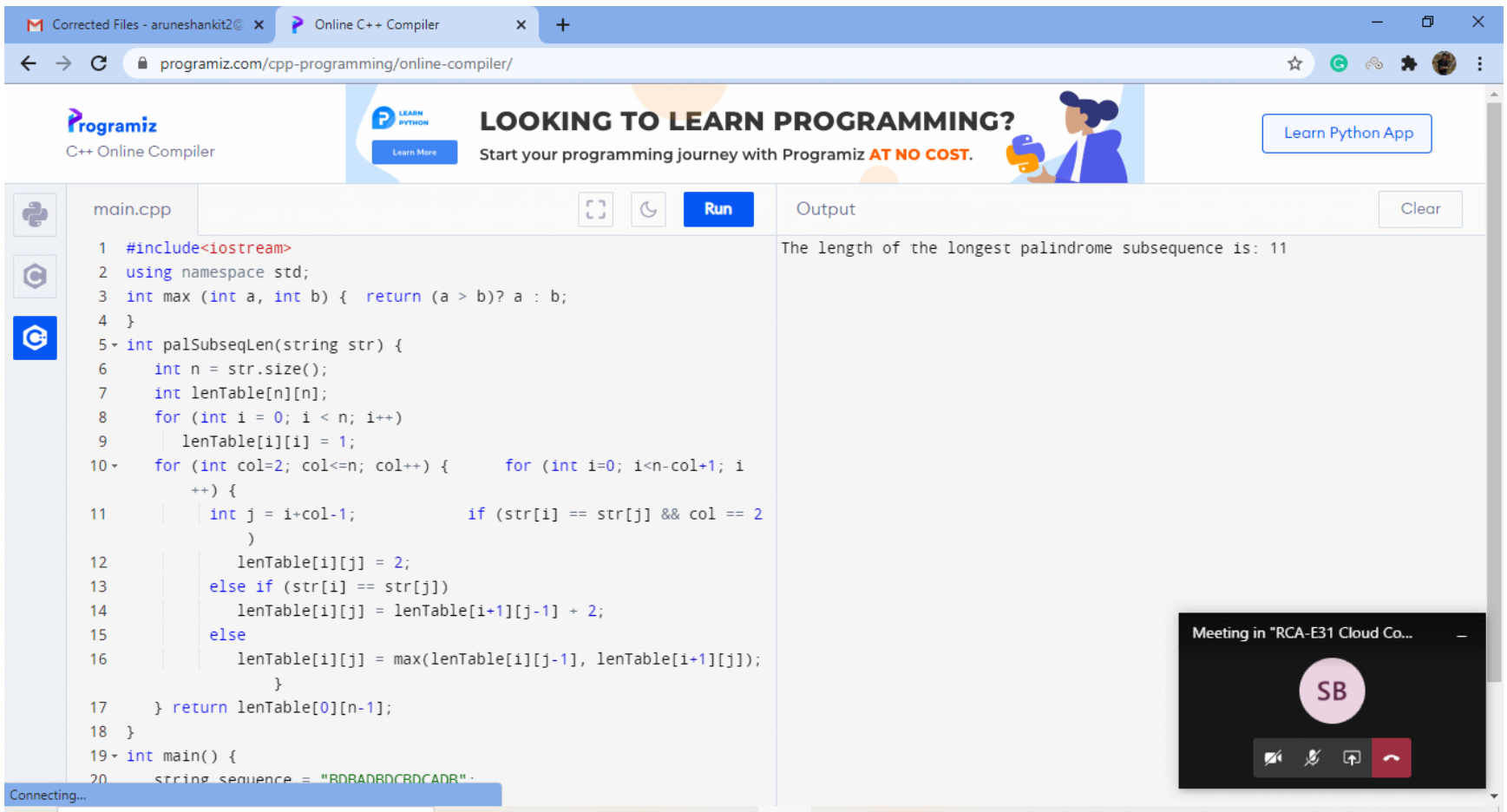
= 2 if a = b - 1 and Z [a] = Z [b]

Code -

```
#include<iostream>
using namespace std;
int max (int a, int b) {  return (a > b)? a : b;
}
int palSubseqLen(string str) {
   int n = str.size();
   int lenTable[n][n];
   for (int i = 0; i < n; i++)
      lenTable[i][i] = 1;
   for (int col=2; col<=n; col++) {      for (int i=0; i<n-col+1; i++) {
      int j = i+col-1;           if (str[i] == str[j] && col == 2)
         lenTable[i][j] = 2;
      else if (str[i] == str[j])
         lenTable[i][j] = lenTable[i+1][j-1] + 2;
      else
         lenTable[i][j] = max(lenTable[i][j-1], lenTable[i+1][j]); }
   } return lenTable[0][n-1];
}
int main() {
   string sequence = "BDBADBDCBDCADB";
   int n = sequence.size();
   cout << "The length of the longest palindrome subsequence is: " << palSubseqLen(sequence);
}
```

OUTPUT -

The length of the longest palindrome subsequence is: 11



Now, if we were to combine all the above cases into a mathematical equation:

We call the original sequence X = (x1x2 ... xm) and reverse as Y = (y1y2 ... ym). Here, the prefixes of X are X1, 2, 3 ... m and the prefixes of Y are Y1, 2, 3 ... m.

Let LCS (XI, Yj) represent the set of the longest common subsequence of prefixes Xi and Yj.

Then:

LCS (Xi , Yj) = ∅ ; if i = 0 or j = 0

LCS (Xi , Yj) = LCS(Xi-1, Yj-1) ^ xi ; if i > 0, j > 0 & xi = yj

LCS (Xi , Yj) = max{ LCS (Xi, Yj-1), LCS (Xi-1, Yj) } ; if i > 0, j > 0 & xi ≠ yj

If the last characters match, then the sequence LCS (Xi-1 , Yj-1) is extended by that matching character xi. Otherwise, the best result from LCS (Xi , Yj-1) and LCS (Xi-1 , Yj) is used.

In the recursive method, we compute some sub-problem, divide it, and repeatedly perform this kind of task. So it's a simple but very tedious method. The time complexity in recursive solution is more. The worst-case time complexity is exponential O (2^n), and auxiliary space used by the program is O (1).

In X, if the last and first characters are the same -

X (0, n - 1) = X (1, n - 2) + 2

If not, then

X (0, n - 1) = MAX (X (1, n - 1), X (0, n - 2)).

CODE -

(IN JAVA)

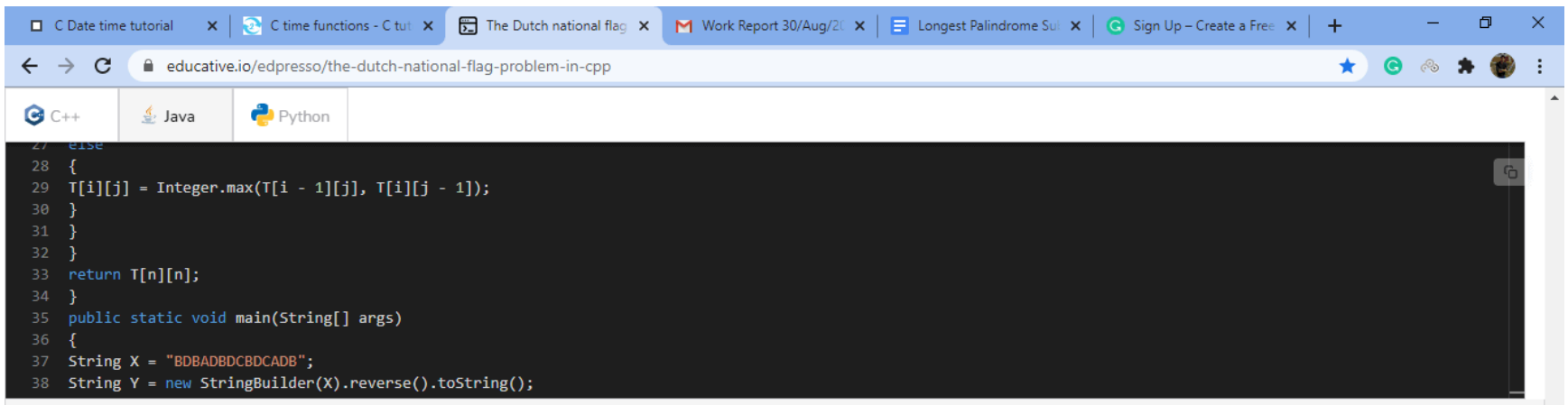
```
1. class Main
2. {
3. public static String longestPalindrome(String X, String Y, int m, int n, int[][] T)
4. {
5. if (m == 0 || n == 0) {
6. return "";
7. }
8. if (X.charAt(m - 1) == Y.charAt(n - 1))
9. {
10. return longestPalindrome(X, Y, m - 1, n - 1, T) + X.charAt(m - 1);
```

```
}
if (T[m - 1][n] > T[m][n - 1]) {
return longestPalindrome(X, Y, m - 1, n, T);
}
return longestPalindrome(X, Y, m, n - 1, T);
}
public static int LCSLength(String X, String Y, int n, int[][] T)
{
for (int i = 1; i <= n; i++)
{
for (int j = 1; j <= n; j++)
{
if (X.charAt(i - 1) == Y.charAt(j - 1))
{
T[i][j] = T[i - 1][j - 1] + 1;
}
else
{
T[i][j] = Integer.max(T[i - 1][j], T[i][j - 1]);
}
}
}
return T[n][n];
}
public static void main(String[] args)
{
String X = "BDBADBDCBDCADB";
String Y = new StringBuilder(X).reverse().toString();
int[][] T = new int[X.length() + 1][X.length() + 1];
System.out.println("The length of Longest Palindromic Subsequence is "
                    + LCSLength(X, Y, X.length(), T));
System.out.println("The Longest Palindromic Subsequence is "
                    + longestPalindrome(X, Y, X.length(), X.length(), T));
}
}
```

OUTPUT -

The length of the Longest Palindromic Subsequence is 11

The Longest Palindromic Subsequence is BDADBCBDADB



Now, if we were to combine all the above cases into a mathematical equation:

We call the original sequence X = (x1x2 ... xm) and reverse as Y = (y1y2 ... ym). Here, the prefixes of X are X1, 2, 3 ... m and the prefixes of Y are Y1, 2, 3 ... m.

Let LCS (XI, Yj) represent the set of the longest common subsequence of prefixes Xi and Yj.

Then:

LCS (Xi , Yj) = ∅ ; if i = 0 or j = 0

LCS (Xi , Yj) = LCS(Xi-1, Yj-1) ^ xi ; if i > 0, j > 0 & xi = yj

LCS (Xi , Yj) = max{ LCS (Xi, Yj-1), LCS (Xi-1, Yj) } ; if i > 0, j > 0 & xi ≠ yj

If the last characters match, then the sequence LCS (Xi-1 , Yj-1) is extended by that matching character xi. Otherwise, the best result from LCS (Xi , Yj-1) and LCS (Xi-1 , Yj) is used.

In the recursive method, we compute some sub-problem, divide it, and repeatedly perform this kind of task. So it's a simple but very tedious method. The time complexity in recursive solution is more. The worst-case time complexity is exponential O (2^n), and auxiliary space used by the program is O (1).

In X, if the last and first characters are the same -

X (0, n - 1) = X (1, n - 2) + 2

If not, then

X (0, n - 1) = MAX (X (1, n - 1), X (0, n - 2)).

CODE -

(IN JAVA)

```
class Main
{
public static String longestPalindrome(String X, String Y, int m, int n, int[][] T)
{
if (m == 0 || n == 0) {
return "";
}
if (X.charAt(m - 1) == Y.charAt(n - 1))
{
return longestPalindrome(X, Y, m - 1, n - 1, T) + X.charAt(m - 1);
```

```
}
if (T[m - 1][n] > T[m][n - 1]) {
return longestPalindrome(X, Y, m - 1, n, T);
}
return longestPalindrome(X, Y, m, n - 1, T);
}
public static int LCSLength(String X, String Y, int n, int[][] T)
{
for (int i = 1; i <= n; i++)
{
for (int j = 1; j <= n; j++)
{
if (X.charAt(i - 1) == Y.charAt(j - 1))
{
T[i][j] = T[i - 1][j - 1] + 1;
}
else
{
T[i][j] = Integer.max(T[i - 1][j], T[i][j - 1]);
}
}
}
return T[n][n];
}
public static void main(String[] args)
{
String X = "BDBADBDCBDCADB";
String Y = new StringBuilder(X).reverse().toString();
int[][] T = new int[X.length() + 1][X.length() + 1];
System.out.println("The length of Longest Palindromic Subsequence is "
                    + LCSLength(X, Y, X.length(), T));
System.out.println("The Longest Palindromic Subsequence is "
                    + longestPalindrome(X, Y, X.length(), X.length(), T));
}
}
```

OUTPUT -

The length of the Longest Palindromic Subsequence is 11

The Longest Palindromic Subsequence is BDADBCBDADB

C++ Java Python

```java
class Main
{
public static String longestPalindrome(String X, String Y, int m, int n, int[][] T)
{
if (m == 0 || n == 0) {
return "";
}
if (X.charAt(m - 1) == Y.charAt(n - 1))
{
return longestPalindrome(X, Y, m - 1, n - 1, T) + X.charAt(m - 1);
}
if (T[m - 1][n] > T[m][n - 1]) {
return longestPalindrome(X, Y, m - 1, n, T);
}
return longestPalindrome(X, Y, m, n - 1, T);
}
public static int LCSLength(String X, String Y, int n, int[][] T)
{
for (int i = 1; i <= n; i++)
{
for (int j = 1; j <= n; j++)
{
if (X.charAt(i - 1) == Y.charAt(j - 1))
{
T[i][j] = T[i - 1][j - 1] + 1;
}
else
```

RUN

Close

Output 1.439s

```
The length of Longest Palindromic Subsequence is 11
The Longest Palindromic Subsequence is BDADBCBDADB
```

Optimal Substructure -

It satisfies overlapping subproblem properties. In a two dimensional array, Longest Common Subsequence can be made as a memo, where LCS[X][Y] represents the length between original with length X and reverse, with length. The longest palindromic subsequence can be generated by backtracking technique, after filling and using the above algorithm

Greedy Algorithm Introduction

"Greedy Method finds out of many options, but you have to choose the best option."

In this method, we have to find out the best method/option out of many present ways.

In this approach/method we focus on the first stage and decide the output, don't think about the future.

This method may or may not give the best output.

Greedy Algorithm solves problems by making the best choice that seems best at the particular moment. Many optimization problems can be determined using a greedy algorithm. Some issues have no efficient solution, but a greedy algorithm may provide a solution that is close to optimal. A greedy algorithm works if a problem exhibits the following two properties:

1. **Greedy Choice Property:** A globally optimal solution can be reached at by creating a locally optimal solution. In other words, an optimal solution can be obtained by creating "greedy" choices.
2. **Optimal substructure:** Optimal solutions contain optimal subsolutions. In other words, answers to subproblems of an optimal solution are optimal.

Example:

1. machine scheduling
2. Fractional Knapsack Problem
3. Minimum Spanning Tree
4. Huffman Code
5. Job Sequencing
6. Activity Selection Problem

Steps for achieving a Greedy Algorithm are:

1. **Feasible:** Here we check whether it satisfies all possible constraints or not, to obtain at least one solution to our problems.
2. **Local Optimal Choice:** In this, the choice should be the optimum which is selected from the currently available
3. **Unalterable:** Once the decision is made, at any subsequence step that option is not altered.

An Activity Selection Problem

The activity selection problem is a mathematical optimization problem. Our first illustration is the problem of scheduling a resource among several challenge activities. We find a greedy algorithm provides a well designed and simple method for selecting a maximum- size set of manually compatible activities.

Suppose S = {1, 2....n} is the set of n proposed activities. The activities share resources which can be used by only one activity at a time, e.g., Tennis Court, Lecture Hall, etc. Each Activity "i" has **start time** s_i and a **finish time** f_i, where $s_i \leq f_i$. If selected activity "i" take place meanwhile the half-open time interval $[s_i, f_i)$. Activities i and j are **compatible** if the intervals (s_i, f_i) and $[s_i, f_i)$ do not overlap (i.e. i and j are compatible if $s_i \geq f_i$ or $s_i \geq f_i$). The activity-selection problem chosen the maximum- size set of mutually consistent activities.

Algorithm Of Greedy- Activity Selector:

GREEDY- ACTIVITY SELECTOR (s, f)

```
1. n ← length [s]
2. A ← {1}
3. j ← 1.
4. for i ← 2 to n
5. do if si ≥ fi
6. then A ← A ∪ {i}
7. j ← i
8. return A
```

Example: Given 10 activities along with their start and end time as

$S = (A_1\ A_2\ A_3\ A_4\ A_5\ A_6\ A_7\ A_8\ A_9\ A_{10})$

Si = (1,2,3,4,7,8,9,9,11,12)

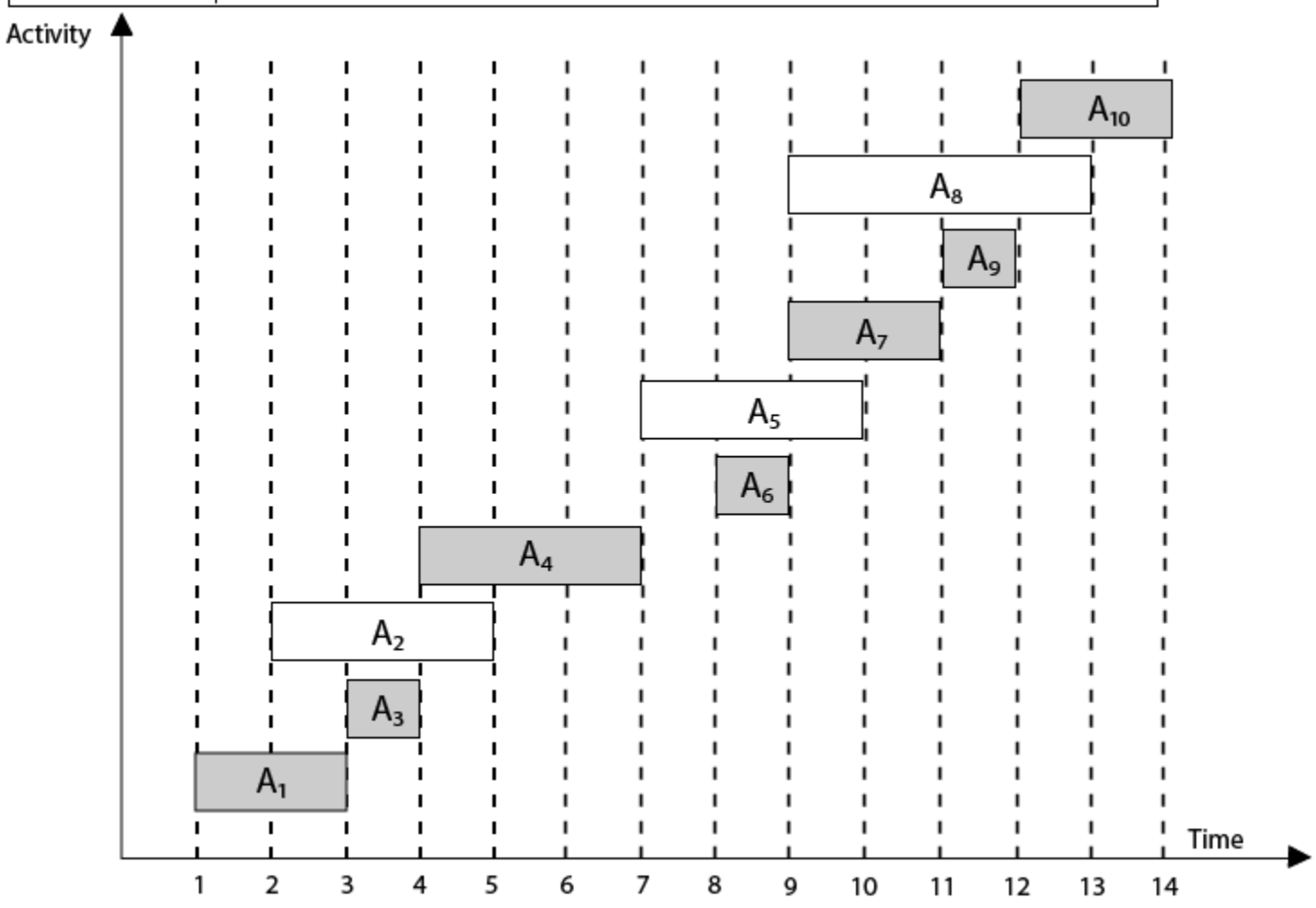
fi = (3,5,4,7,10,9,11,13,12,14)

Compute a schedule where the greatest number of activities takes place.

Solution: The solution to the above Activity scheduling problem using a greedy strategy is illustrated below:

Arranging the activities in increasing order of end time

Activity	A_1	A_3	A_2	A_4	A_6	A_5	A_7	A_9	A_8	A_{10}
Start	1	3	2	4	8	7	9	11	9	12
Finish	3	4	5	7	9	10	11	12	13	14



Now, schedule A_1

Next schedule A_3 as A_1 and A_3 are non-interfering.

Next **skip** A_2 as it is interfering.

Next, schedule A_4 as A_1 A_3 and A_4 are non-interfering, then next, schedule A_6 as A_1 A_3 A_4 and A_6 are non-interfering.

Skip A_5 as it is interfering.

Next, schedule A_7 as A_1 A_3 A_4 A_6 and A_7 are non-interfering.

Next, schedule A_9 as A_1 A_3 A_4 A_6 A_7 and A_9 are non-interfering.

Skip A_8 as it is interfering.

Next, schedule A_{10} as A_1 A_3 A_4 A_6 A_7 A_9 and A_{10} are non-interfering.

Thus the final Activity schedule is:

$$(A_1\ A_3\ A_4\ A_6\ A_7\ A_9\ A_{10})$$

Fractional Knapsack

Fractions of items can be taken rather than having to make binary (0-1) choices for each item.

Fractional Knapsack Problem can be solvable by greedy strategy whereas 0 - 1 problem is not.

Steps to solve the Fractional Problem:

1. Compute the value per pound v_i/w_i for each item.
2. Obeying a Greedy Strategy, we take as possible of the item with the highest value per pound.
3. If the supply of that element is exhausted and we can still carry more, we take as much as possible of the element with the next value per pound.

4. Sorting, the items by value per pound, the greedy algorithm run in O (n log n) time.

```
Fractional Knapsack (Array v, Array w, int W)
1. for i= 1 to size (v)
2. do p [i] = v [i] / w [i]
3. Sort-Descending (p)
4. i ← 1
5. while (W>0)
6. do amount = min (W, w [i])
7. solution [i] = amount
8. W= W-amount
9. i ← i+1
10. return solution
```

Example: Consider 5 items along their respective weights and values: -

$I = (I_1, I_2, I_3, I_4, I_5)$

w = (5, 10, 20, 30, 40)

v = (30, 20, 100, 90,160)

The capacity of knapsack W = 60

Now fill the knapsack according to the decreasing value of p_i.

First, we choose the item I_i whose weight is 5.

Then choose item I_3 whose weight is 20. Now,the total weight of knapsack is 20 + 5 = 25

Now the next item is I_5, and its weight is 40, but we want only 35, so we chose the fractional part of it,

$$\text{i.e., } 5 \times \frac{5}{5} + 20 \times \frac{20}{20} + 40 \times \frac{35}{40}$$

Weight = 5 + 20 + 35 = 60

Maximum Value:-

$$30 \times \frac{5}{5} + 100 \times \frac{20}{20} + 160 \times \frac{35}{40}$$

= 30 + 100 + 140 = 270 (Minimum Cost)

Solution:

ITEM	w_i	v_i
I_1	5	30
I_2	10	20
I_3	20	100
I_4	30	90
I_5	40	160

Taking value per weight ratio i.e. $p_i = \frac{v_i}{w_i}$

ITEM	w_i	v_i	$P_i=\frac{v_i}{w_i}$
I_1	5	30	6.0
I_2	10	20	2.0
I_3	20	100	5.0
I_4	30	90	3.0
I_5	40	160	4.0

Now, arrange the value of p_i in decreasing order.

ITEM	w_i	v_i	$p_i=\frac{v_i}{w_i}$
I_1	5	30	6.0
I_3	20	100	5.0
I_5	40	160	4.0
I_4	30	90	3.0
I_2	10	20	2.0

Huffman Codes

- (i) Data can be encoded efficiently using Huffman Codes.
- (ii) It is a widely used and beneficial technique for compressing data.
- (iii) Huffman's greedy algorithm uses a table of the frequencies of occurrences of each character to build up an optimal way of representing each character as a binary string.

Suppose we have 10^5 characters in a data file. Normal Storage: 8 bits per character (ASCII) - 8 x 10^5 bits in a file. But we want to compress the file and save it compactly. Suppose only six characters appear in the file:

	a	b	c	d	e	f	Total
Frequency	45	13	12	16	9	5	100

How can we represent the data in a Compact way?

(i) Fixed length Code: Each letter represented by an equal number of bits. With a fixed length code, at least 3 bits per character:

For example:

```
a                   000

b                   001

c                   010

d                   011

e                   100
```

```
f                    101
```

For a file with 10^5 characters, we need 3 x 10^5 bits.

(ii) A variable-length code: It can do considerably better than a fixed-length code, by giving many characters short code words and infrequent character long codewords.

For example:

```
a       0
b      101
c      100
d      111
e      1101
f      1100
Number of bits = (45 x 1 + 13 x 3 + 12 x 3 + 16 x 3 + 9 x 4 + 5 x 4) x 1000
```

= 2.24 x 10^5bits

Thus, 224,000 bits to represent the file, a saving of approximately 25%.This is an optimal character code for this file.

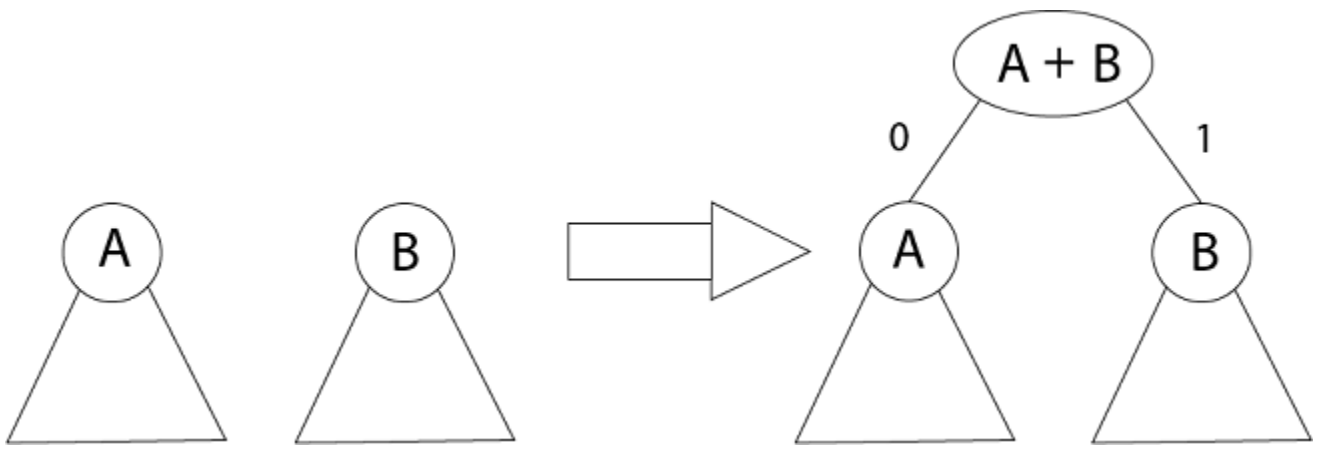
Prefix Codes:

The prefixes of an encoding of one character must not be equal to complete encoding of another character, e.g., 1100 and 11001 are not valid codes because 1100 is a prefix of some other code word is called prefix codes.

Prefix codes are desirable because they clarify encoding and decoding. Encoding is always simple for any binary character code; we concatenate the code words describing each character of the file. Decoding is also quite comfortable with a prefix code. Since no codeword is a prefix of any other, the codeword that starts with an encoded data is unambiguous.

Greedy Algorithm for constructing a Huffman Code:

Huffman invented a greedy algorithm that creates an optimal prefix code called a Huffman Code.



The algorithm builds the tree T analogous to the optimal code in a bottom-up manner. It starts with a set of |C| leaves (C is the number of characters) and performs |C| - 1 'merging' operations to create the final tree. In the Huffman algorithm 'n' denotes the quantity of a set of characters, z indicates the parent node, and x & y are the left & right child of z respectively.

Algorithm of Huffman Code

Huffman (C)

```
1. n=|C|
2. Q ← C
3. for i=1 to n-1
4. do
5. z= allocate-Node ()
6. x=  left[z]=Extract-Min(Q)
7. y= right[z] =Extract-Min(Q)
8. f [z]=f[x]+f[y]
9. Insert (Q, z)
10. return Extract-Min (Q)
```

Example: Find an optimal Huffman Code for the following set of frequencies:

1. a: 50 b: 25 c: 15 d: 40 e: 75

Solution:

Given that: C = {a, b, c, d, e}

f (C) = {50, 25, 15, 40, 75}

n = 5

Q ← c

i.e.

c	15		b	25		d	40		a	50		e	75

for i ← 1 to 4

i = 1 Z ← Allocate node

x ← Extract-Min (Q)

y ← Extract-Min (Q)

c	15		b	25		d	40		a	50		e	75

Left [z] ← x

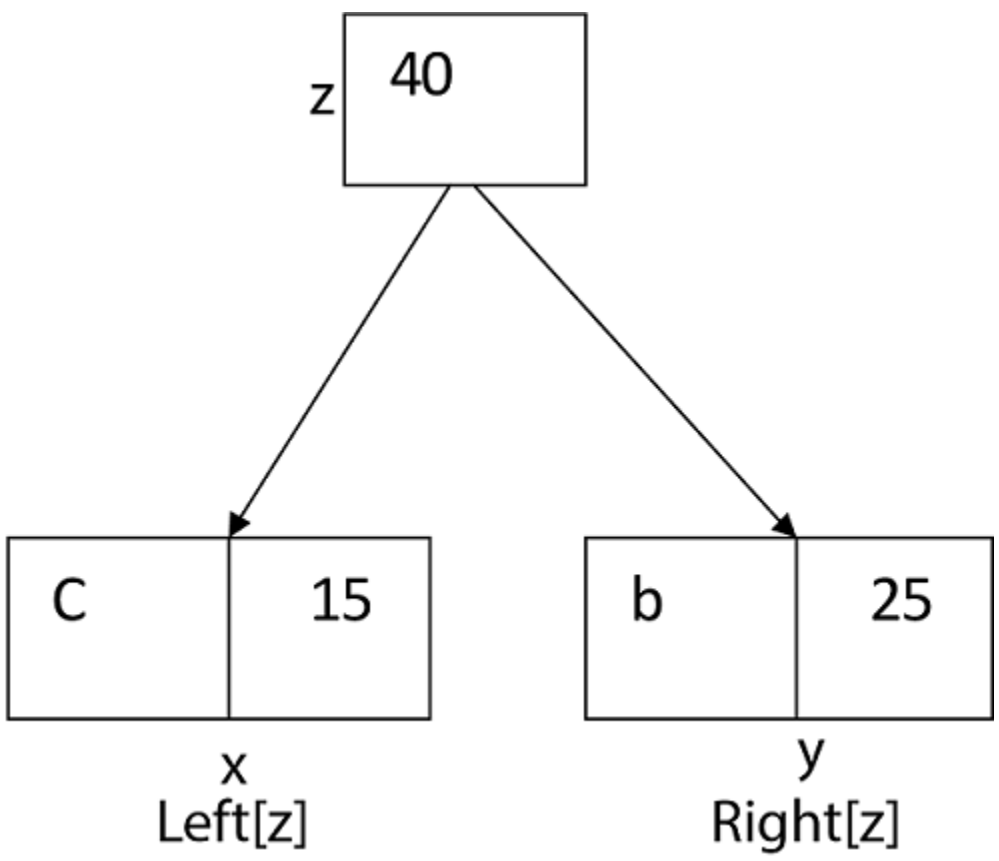
Right [z] ← y

f (z) ← f (x) + f (y) = 15 + 25

f (z) = 40

d	40		a	50		e	75

i.e.



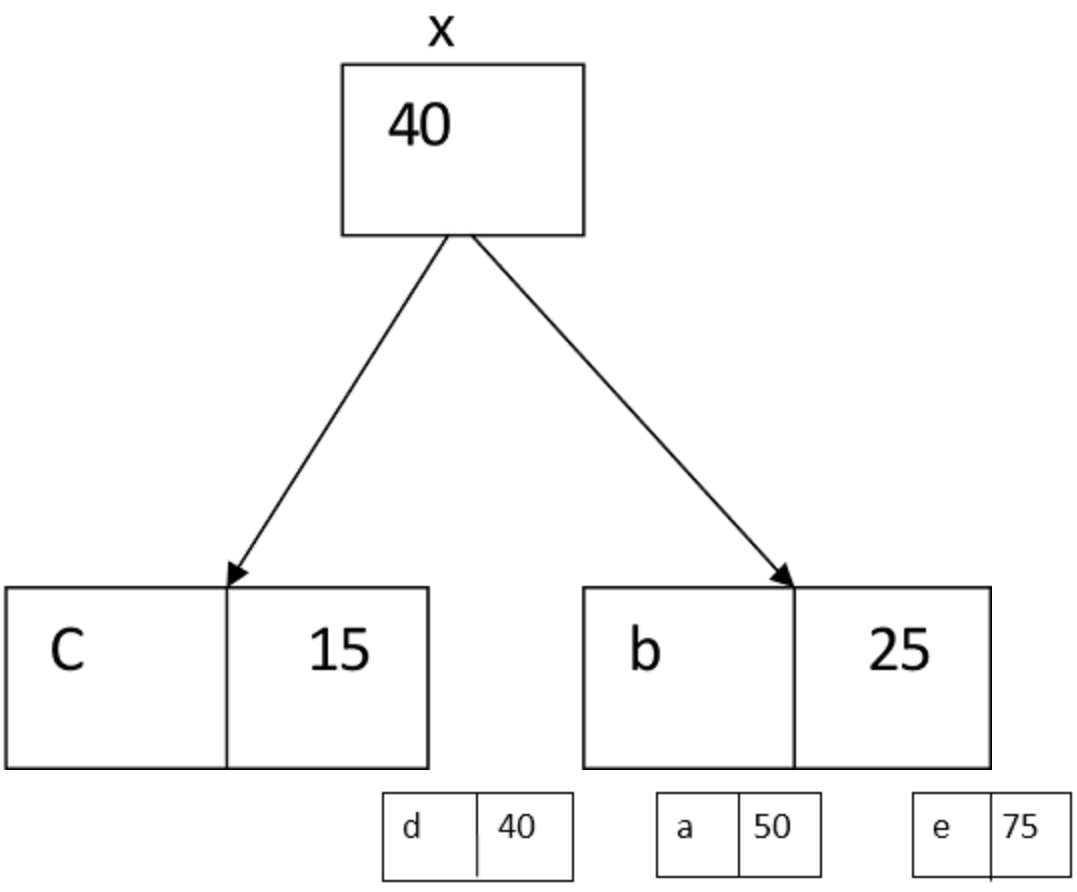
Again for i=2

Features of Java

The features of Java are also known as java *buzzwords*.

A list of most important features of Java language is given below.

1) Simple
2) Object-Oriented
3) Portable
4) Platfom Independent
5) Secured
6) Robust
7) Architecture neutral
8) Interpreted
9) High Performance
10) Multithreaded
11) Distributed
12)



z ⟵ Allocate node

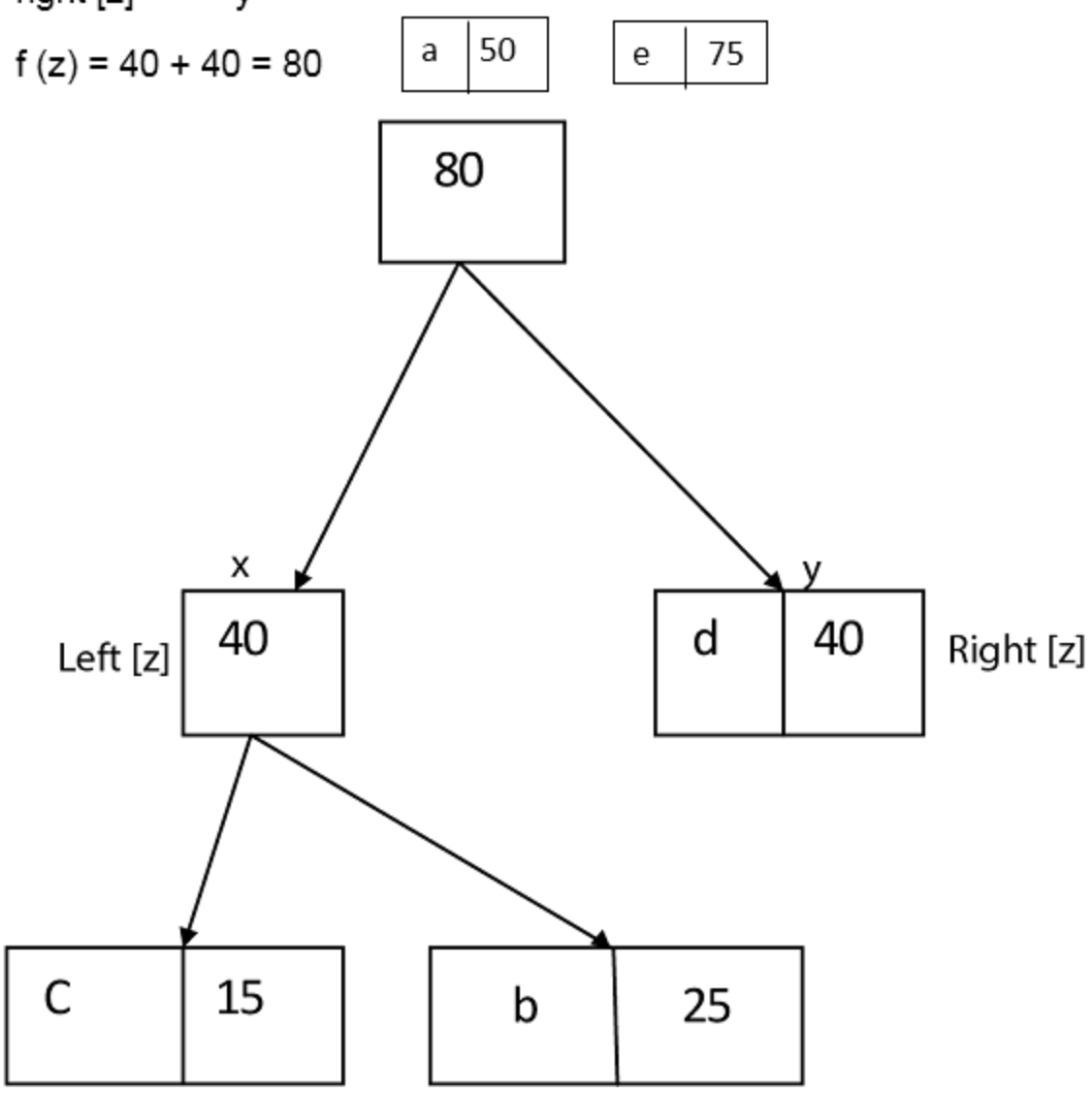
x ⟵ 40

y ⟵ 40

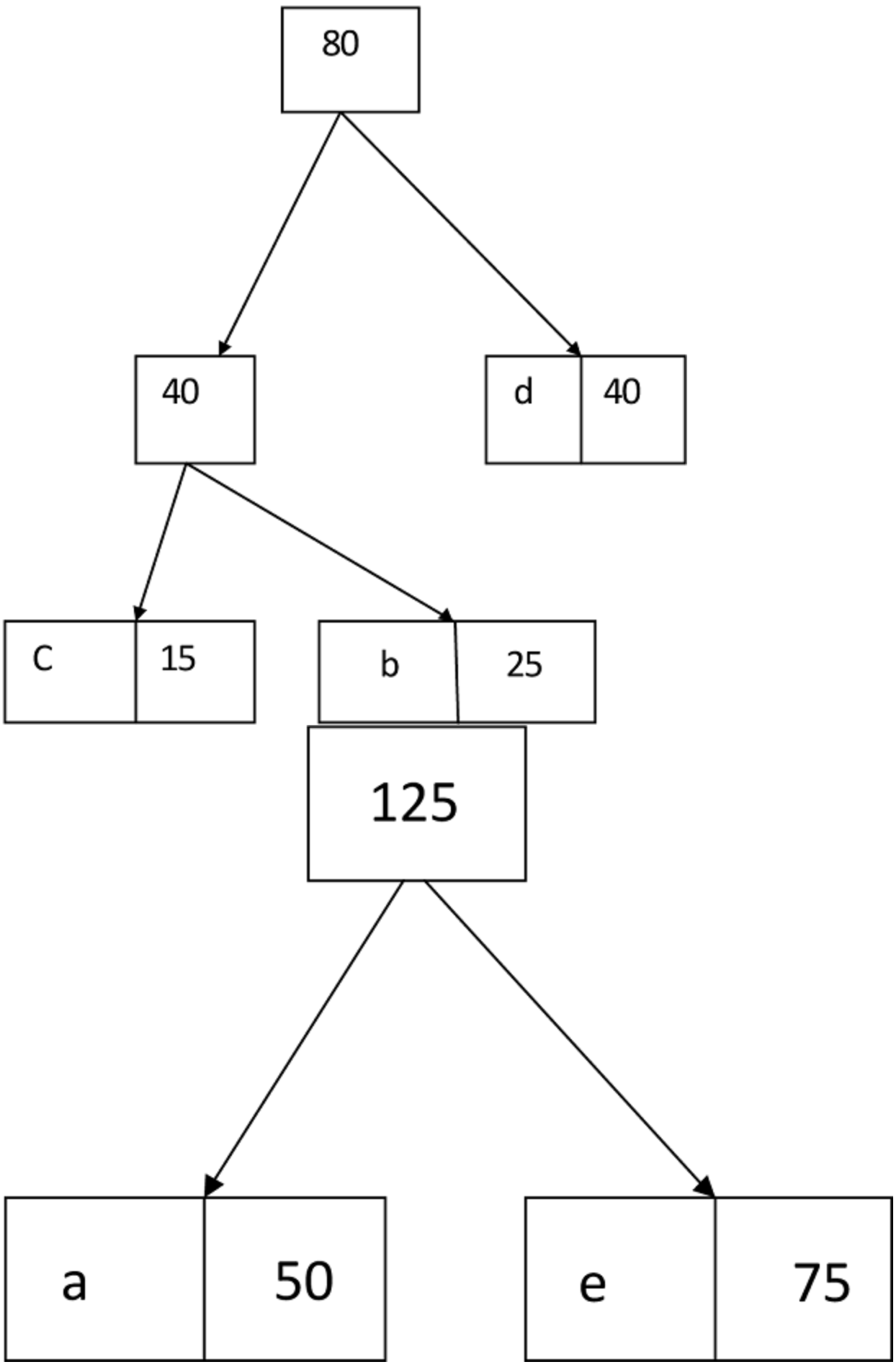
left [z] ⟵ x

right [z] ⟵ y

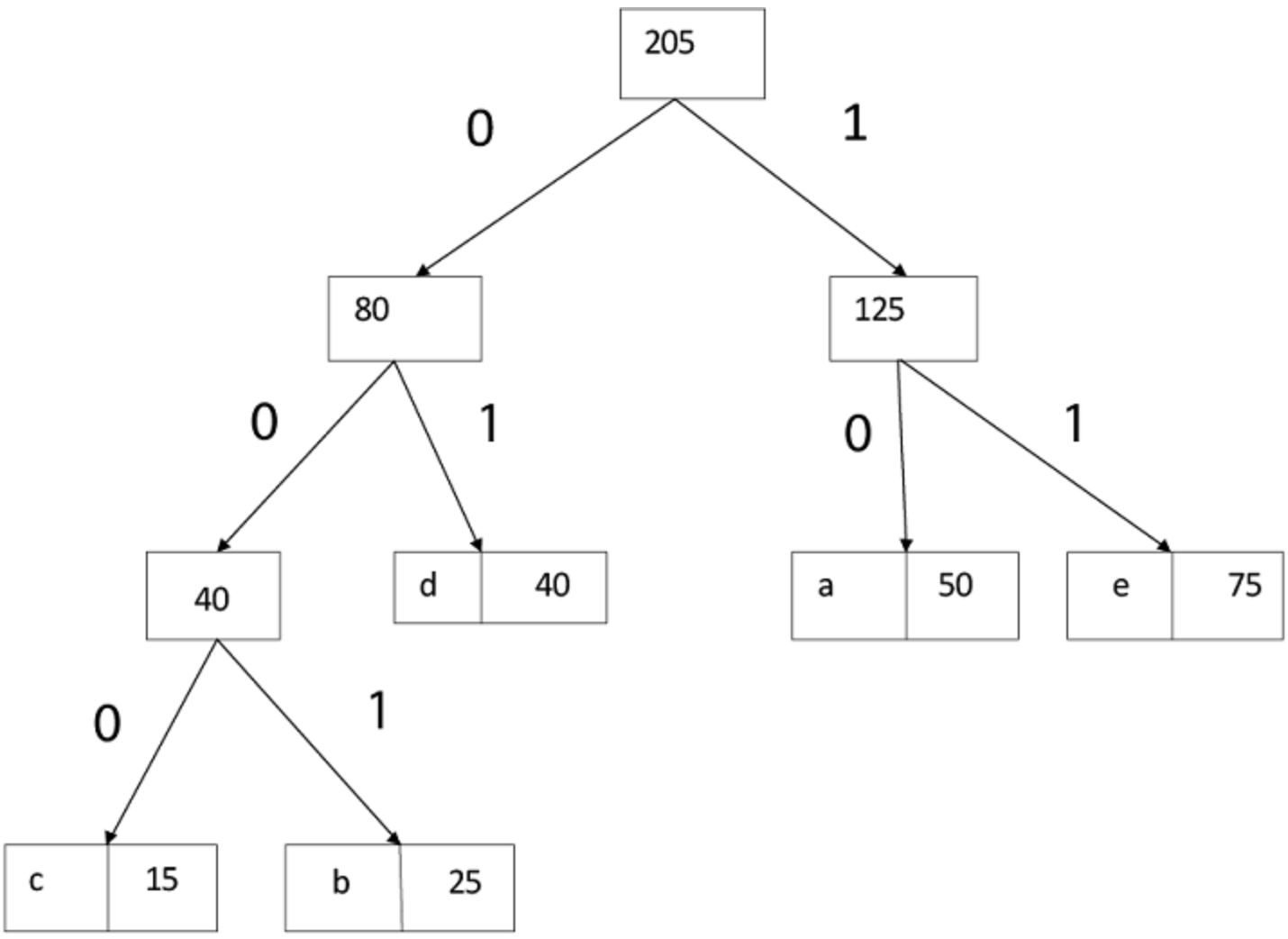
f (z) = 40 + 40 = 80



Similarly, we apply the same process we get



Thus, the final output is:



Activity or Task Scheduling Problem

This is the dispute of optimally scheduling unit-time tasks on a single processor, where each job has a deadline and a penalty that necessary be paid if the deadline is missed.

A unit-time task is a job, such as a program to be rush on a computer that needed precisely one unit of time to complete. Given a finite set S of unit-time tasks, a schedule for S is a permutation of S specifying the order in which to perform these tasks. The first task in the schedule starts at time 0 and ends at time 1; the second task begins at time 1 and finishes at time 2, and so on.

The dispute of scheduling unit-time tasks with deadlines and penalties for each processor has the following inputs:

- a set S = {1, 2, 3.....n} of n unit-time tasks.
- a set of n integer deadlines d_1 d_2 d_3...d_n such that d_i satisfies $1 \leq d_i \leq n$ and task i is supposed to finish by time d_i and
- a set of n non-negative weights or penalties w_1 w_2....w_n such that we incur a penalty of w_i if task i is not finished by time d_i, and we incurred no penalty if a task finishes by its deadline.

Here we find a schedule for S that minimizes the total penalty incurred for missed deadlines.

A task is **late** in this schedule if it finished after its deadline. Otherwise, the task is early in the schedule. An arbitrary schedule can consistently be put into **early-first form**, in which the first tasks precede the late tasks, i.e., if some new task x follows some late task y, then we can switch the position of x and y without affecting x being early or y being late.

An arbitrary schedule can always be put into a **canonical form** in which first tasks precede the late tasks, and first tasks are scheduled in order of nondecreasing deadlines.

A set A of tasks is **independent** if there exists a schedule for the particular tasks such that no tasks are late. So the set of first tasks for a schedule forms an independent set of tasks 'I' denote the set of all independent set of tasks.

For any set of tasks A, A is independent if for t = 0, 1, 2.....n we have $N_t(A) \leq t$ where $N_t(A)$ denotes the number of tasks in A whose deadline is t or prior, i.e. if the tasks in A are expected in order of monotonically growing deadlines, then no task is late.

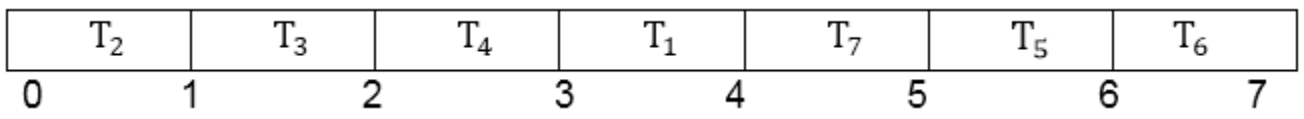
Example: Find the optimal schedule for the following task with given weight (penalties) and deadlines.

	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Solution: According to the Greedy algorithm we sort the jobs in decreasing order of their penalties so that minimum of penalties will be charged.

In this problem, we can see that the maximum time for which uniprocessor machine will run in 6 units because it is the maximum deadline.

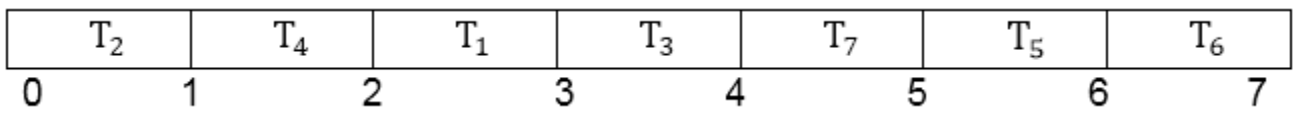
Let T_i represents the tasks where i = 1 to 7



T_5 and T_6 cannot be accepted after T_7 so penalty is

$w_5 + w_6 = 30 + 20 = 50$ (2 3 4 1 7 5 6)

Other schedule is



(2 4 1 3 7 5 6)

There can be many other schedules but (2 4 1 3 7 5 6) is optimal.

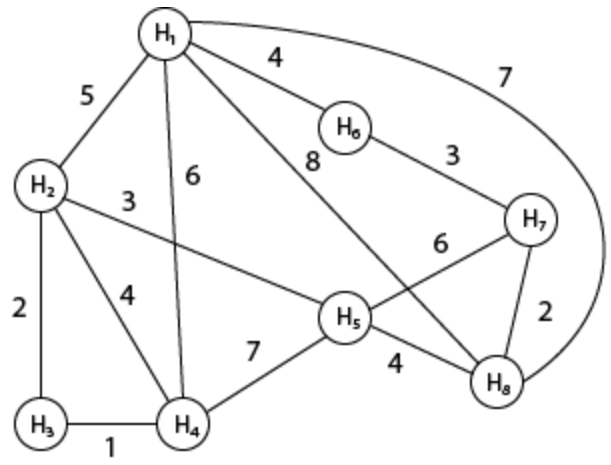
Travelling Sales Person Problem

The traveling salesman problems abide by a salesman and a set of cities. The salesman has to visit every one of the cities starting from a certain one (e.g., the hometown) and to return to the same city. The challenge of the problem is that the traveling salesman needs to minimize the total length of the trip.

Suppose the cities are x_1 x_2..... x_n where cost c_{ij} denotes the cost of travelling from city x_i to x_j. The travelling salesperson problem is to find a route starting and ending at x_1 that will take in all cities with the minimum cost.

Example: A newspaper agent daily drops the newspaper to the area assigned in such a manner that he has to cover all the houses in the respective area with minimum travel cost. Compute the minimum travel cost.

The area assigned to the agent where he has to drop the newspaper is shown in fig:



Solution: The cost- adjacency matrix of graph G is as follows:

$cost_{ij}$ =

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

The tour starts from area H_1 and then select the minimum cost area reachable from H_1.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H_1)	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_6 because it is the minimum cost area reachable from H_1 and then select minimum cost area reachable from H_6.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H_1)	0	5	0	6	0	(4)	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
(H_6)	4	0	0	0	0	0	(3)	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_7 because it is the minimum cost area reachable from H_6 and then select minimum cost area reachable from H_7.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H1)	0	5	0	6	0	(4)	0	7
H2	5	0	2	4	3	0	0	0
H3	0	2	0	1	0	0	0	0
H4	6	4	1	0	7	0	0	0
H5	0	3	0	7	0	0	6	4
(H6)	4	0	0	0	0	0	(3)	0
(H7)	0	0	0	0	6	3	0	(2)
H8	7	0	0	0	4	0	2	0

Mark area H_8 because it is the minimum cost area reachable from H_8.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H1)	0	5	0	6	0	(4)	0	7
H2	5	0	2	4	3	0	0	0
H3	0	2	0	1	0	0	0	0
H4	6	4	1	0	7	0	0	0
H5	0	3	0	7	0	0	6	4
(H6)	4	0	0	0	0	0	(3)	0
(H7)	0	0	0	0	6	3	0	(2)
(H8)	7	0	0	0	(4)	0	2	0

Mark area H_5 because it is the minimum cost area reachable from H_5.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H1)	0	5	0	6	0	(4)	0	7
H2	5	0	2	4	3	0	0	0
H3	0	2	0	1	0	0	0	0
H4	6	4	1	0	7	0	0	0
(H5)	0	(3)	0	7	0	0	6	4
(H6)	4	0	0	0	0	0	(3)	0
(H7)	0	0	0	0	6	3	0	(2)
(H8)	7	0	0	0	(4)	0	2	0

Mark area H_2 because it is the minimum cost area reachable from H_2.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
(H1)	0	5	0	6	0	(4)	0	7
(H2)	5	0	(2)	4	3	0	0	0
H3	0	2	0	1	0	0	0	0
H4	6	4	1	0	7	0	0	0
(H5)	0	(3)	0	7	0	0	6	4
(H6)	4	0	0	0	0	0	(3)	0
(H7)	0	0	0	0	6	3	0	(2)
(H8)	7	0	0	0	(4)	0	2	0

Mark area H_3 because it is the minimum cost area reachable from H_3.

	H_1	H_2	H_3	H_4	H_5	H_6	H_7	H_8
H_1	0	5	0	6	0	4	0	7
H_2	5	0	2	4	3	0	0	0
H_3	0	2	0	1	0	0	0	0
H_4	6	4	1	0	7	0	0	0
H_5	0	3	0	7	0	0	6	4
H_6	4	0	0	0	0	0	3	0
H_7	0	0	0	0	6	3	0	2
H_8	7	0	0	0	4	0	2	0

Mark area H_4 and then select the minimum cost area reachable from H_4 it is H_1.So, using the greedy strategy, we get the following.

$$H_1 \xrightarrow{4} H_6 \xrightarrow{3} H_7 \xrightarrow{2} H_8 \xrightarrow{4} H_5 \xrightarrow{3} H_2 \xrightarrow{2} H_3 \xrightarrow{1} H_4 \xrightarrow{6} H1.$$

Thus the minimum travel cost = 4 + 3 + 2 + 4 + 3 + 2 + 1 + 6 = 25

Matroids:

A matroid is an ordered pair M(S, I) satisfying the following conditions:

1. S is a finite set.
2. I is a nonempty family of subsets of S, called the independent subsets of S, such that if B ∈ I and A ∈ I. We say that I is hereditary if it satisfies this property. Note that the empty set ∅ is necessarily a member of I.
3. If A ∈ I, B ∈ I and |A| <|B|, then there is some element x ∈ B ? A such that A∪{x}∈I. We say that M satisfies the exchange property.

We say that a matroid M (S, I) is weighted if there is an associated weight function w that assigns a strictly positive weight w (x) to each element x ∈ S. The weight function w extends to a subset of S by summation:

$$w(A) = \sum_{x \in A} w(x)$$

for any A ∈ S.

Differentiate between Dynamic Programming and Greedy Method

Dynamic Programming	Greedy Method
1. Dynamic Programming is used to obtain the optimal solution.	1. Greedy Method is also used to get the optimal solution.
2. In Dynamic Programming, we choose at each step, but the choice may depend on the solution to sub-problems.	2. In a greedy Algorithm, we make whatever choice seems best at the moment and then solve the sub-problems arising after the choice is made.
3. Less efficient as compared to a greedy approach	3. More efficient as compared to a greedy approach
4. Example: 0/1 Knapsack	4. Example: Fractional Knapsack
5. It is guaranteed that Dynamic Programming will generate an optimal solution using Principle of Optimality.	5. In Greedy Method, there is no such guarantee of getting Optimal Solution.

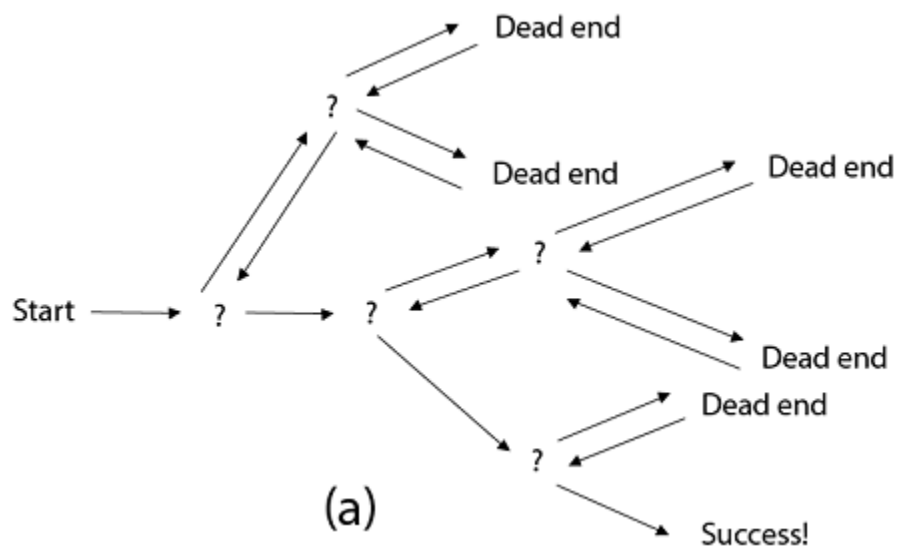
Introduction of Backtracking

The Backtracking is an algorithmic-method to solve a problem with an additional way. It uses a recursive approach to explain the problems. We can say that the backtracking is needed to find all possible combination to solve an optimization problem.

Backtracking is a systematic way of trying out different sequences of decisions until we find one that "works."

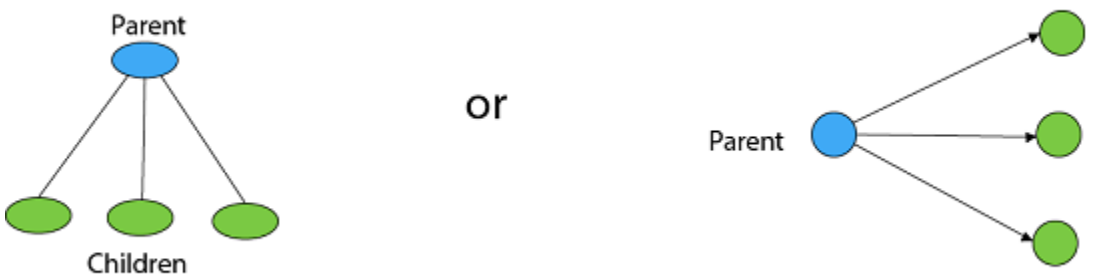
In the following Figure:

- Each non-leaf node in a tree is a parent of one or more other nodes (its children)
- Each node in the tree, other than the root, has exactly one parent



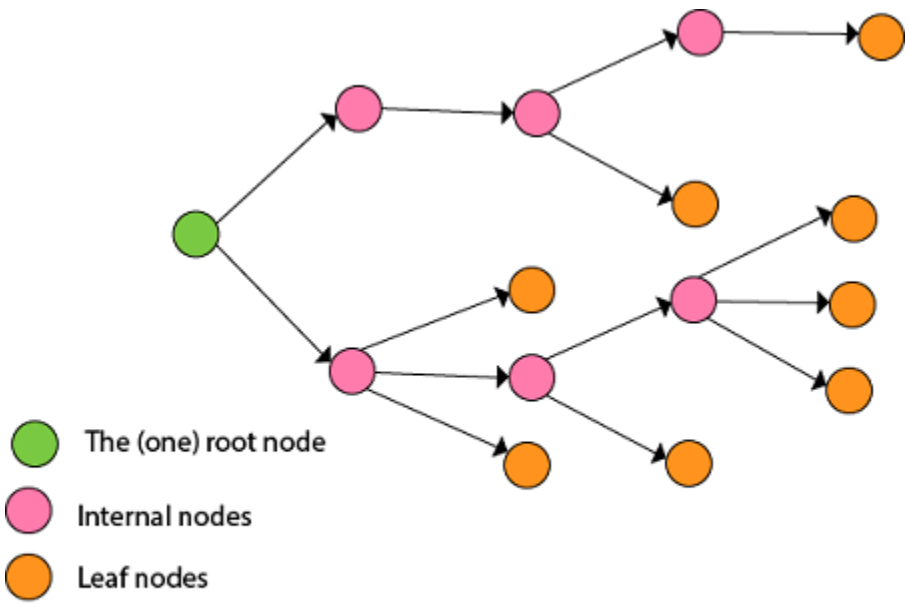
(a)

Generally, however, we draw our trees downward, with the root at the top.



(b)

A tree is composed of nodes.



(c)

Backtracking can understand of as searching a tree for a particular "goal" leaf node.

Backtracking is undoubtedly quite simple - we "explore" each node, as follows:

To "explore" node N:

```
1. If N is a goal node, return "success"
2. If N is a leaf node, return "failure"
```

```
3. For each child C of N,
    Explore C
    If C was successful, return "success"
4. Return "failure"
```

Backtracking algorithm determines the solution by systematically searching the solution space for the given problem. Backtracking is a **depth-first search** with any bounding function. All solution using backtracking is needed to satisfy a complex set of constraints. The constraints may be explicit or implicit.

Explicit Constraint is ruled, which restrict each vector element to be chosen from the given set.

Implicit Constraint is ruled, which determine which each of the tuples in the solution space, actually satisfy the criterion function.

Recursive Maze Algorithm

Recursive Maze Algorithm is one of the best examples for backtracking algorithms. Recursive Maze Algorithm is one of the possible solutions for solving the maze.

Maze

The maze is an area surrounded by walls; in between, we have a path from starting point to ending position. We have to start from the starting point and travel towards from ending point.

Start

End

Principle of Maze

As explained above, in the maze we have to travel from starting point to ending point. The problem is to choose the path. If we find any dead-end before ending point, we have to backtrack and move the direction. The direction for traversing is North, East, West, and South. We have to continue "move and backtrack" until we reach the final point.

Consider that we are having a two-dimensional maze cell [WIDTH] [HEIGHT]. Here cell [x] [y] = 1 denotes wall and cell [x] [y] = 0 denotes free cell in the particular location x, y in the maze. The directions we can change in the array are North, East, West, and South. The first step is to make the boundary of the two - dimensional array as one so that we won't go out of the maze and usually reside inside the maze at any time.

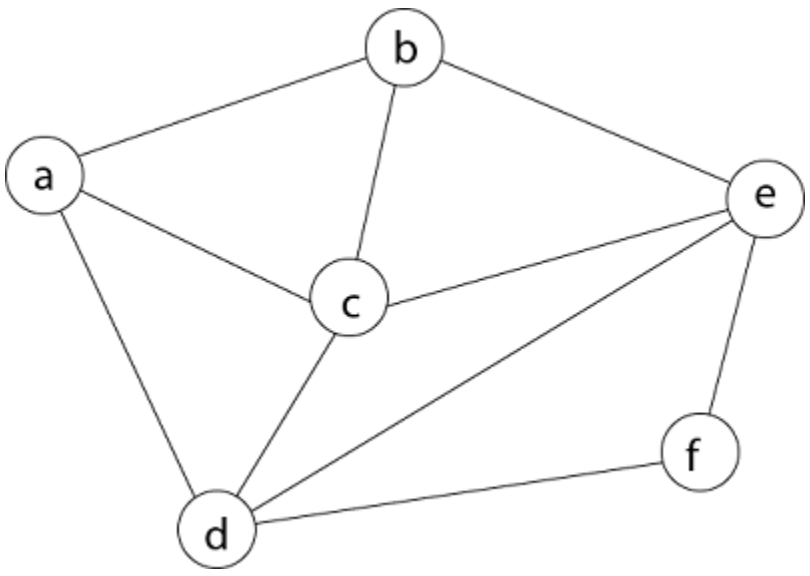
Example Maze						
1	1	1	1	1	1	1
1	0	0	0	1	1	1
1	1	1	0	1	1	1
1	1	1	0	0	0	1
1	1	1	1	1	0	1
1	1	1	1	1	1	1

Now start changing from the starting position (since the boundary is filled by 1) and find the next free cell then turn to the next free cell and so on. If we grasp a dead-end, we have to backtrack and make the cells in the path as 1 (wall). Continue the same process till the final point is reached.

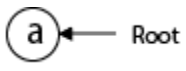
Hamiltonian Circuit Problems

Given a graph G = (V, E) we have to find the Hamiltonian Circuit using Backtracking approach. We start our search from any arbitrary vertex say 'a.' This vertex 'a' becomes the root of our implicit tree. The first element of our partial solution is the first intermediate vertex of the Hamiltonian Cycle that is to be constructed. The next adjacent vertex is selected by alphabetical order. If at any stage any arbitrary vertex makes a cycle with any vertex other than vertex 'a' then we say that **dead end** is reached. In this case, we backtrack one step, and again the search begins by selecting another vertex and backtrack the element from the partial; solution must be removed. The search using backtracking is successful if a Hamiltonian Cycle is obtained.

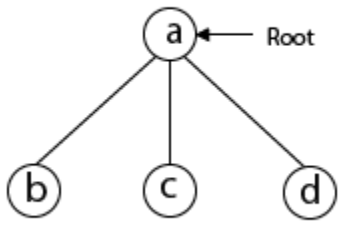
Example: Consider a graph G = (V, E) shown in fig. we have to find a Hamiltonian circuit using Backtracking method.



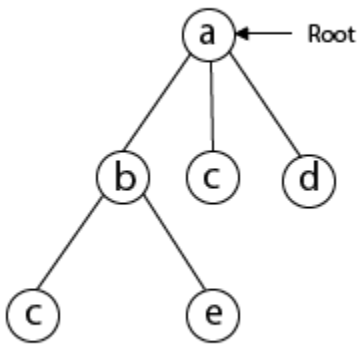
Solution: Firstly, we start our search with vertex 'a.' this vertex 'a' becomes the root of our implicit tree.



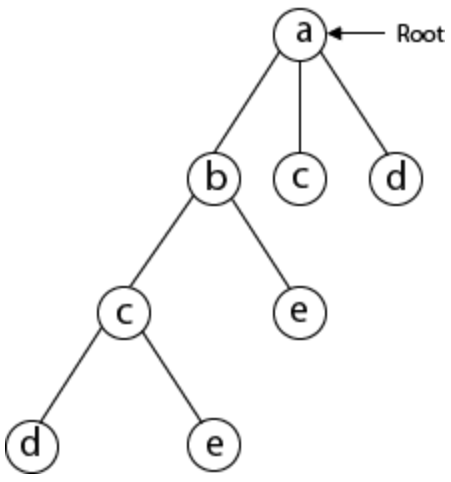
Next, we choose vertex 'b' adjacent to 'a' as it comes first in lexicographical order (b, c, d).



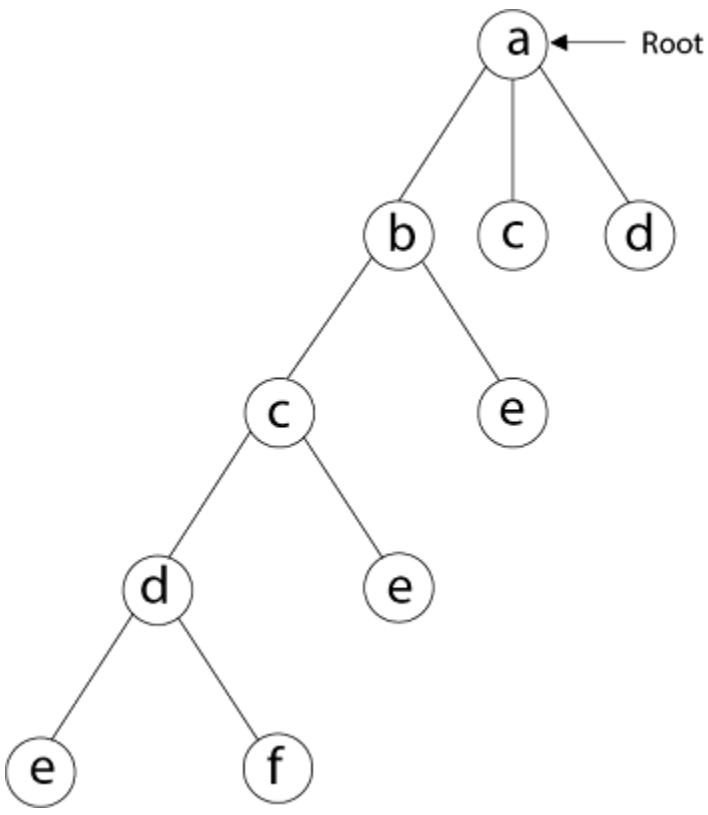
Next, we select 'c' adjacent to 'b.'



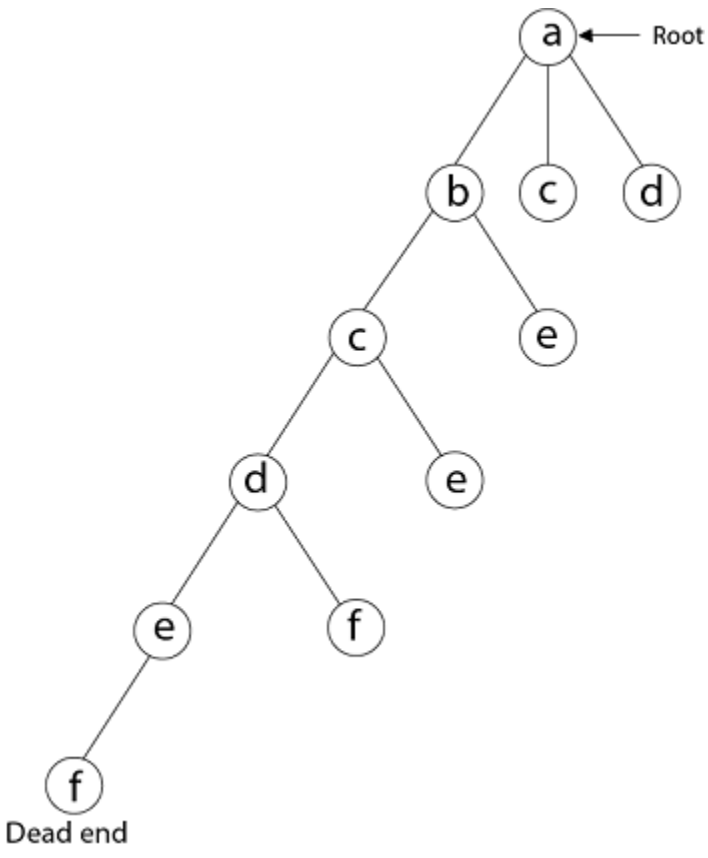
Next, we select 'd' adjacent to 'c.'



Next, we select 'e' adjacent to 'd.'



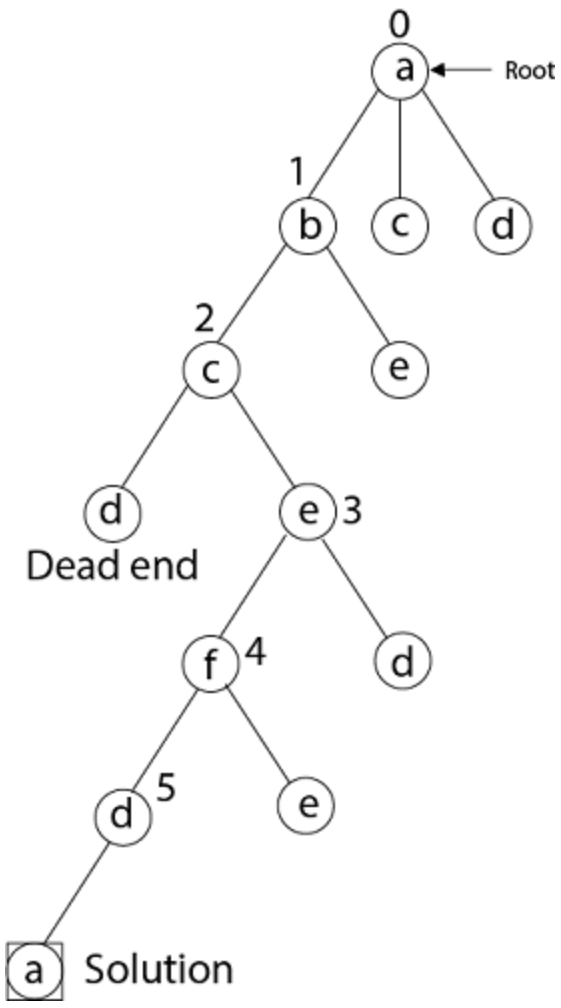
Next, we select vertex 'f' adjacent to 'e.' The vertex adjacent to 'f' is d and e, but they have already visited. Thus, we get the dead end, and we backtrack one step and remove the vertex 'f' from partial solution.



From backtracking, the vertex adjacent to 'e' is b, c, d, and f from which vertex 'f' has already been checked, and b, c, d have already visited. So, again we backtrack one step. Now, the vertex adjacent to d are e, f from which e has already been checked, and adjacent of 'f' are d and e. If 'e' vertex, revisited them we get a dead state. So again we backtrack one step.

Now, adjacent to c is 'e' and adjacent to 'e' is 'f' and adjacent to 'f' is 'd' and adjacent to 'd' is 'a.' Here, we get the Hamiltonian Cycle as all the vertex other than the start vertex 'a' is visited only once. (a - b - c - e - f -d - a).

Again Backtrack



Here we have generated one Hamiltonian circuit, but another Hamiltonian circuit can also be obtained by considering another vertex.

Subset-Sum Problem

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:

$$S_1 \leq S_2 \leq S_3 \ldots \leq S_n$$

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.

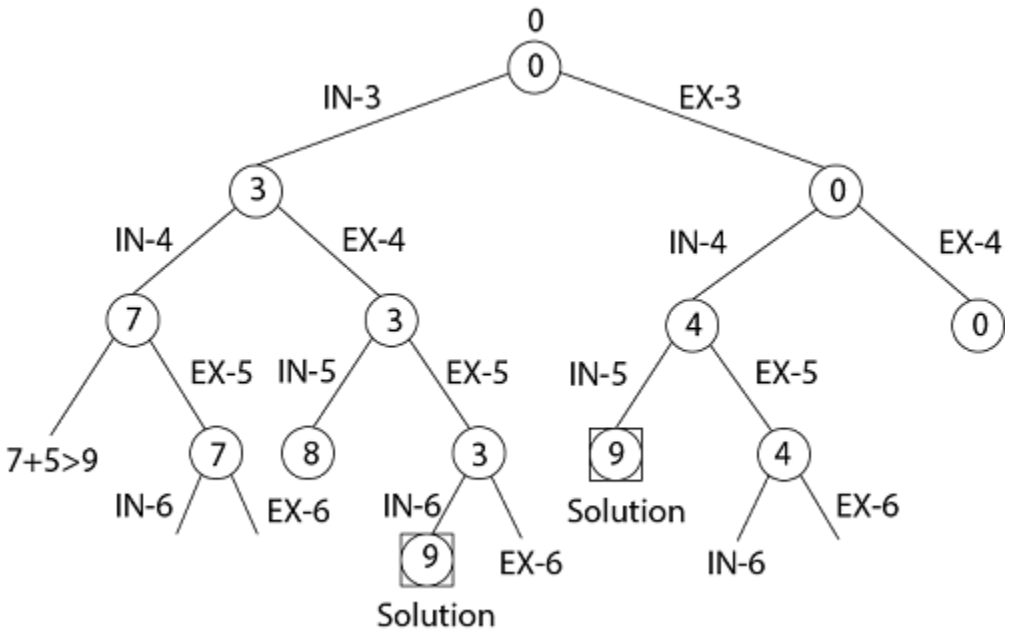
$$s' + \sum_{j=i+1}^{n} S_j < X$$

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'= (Ø)

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.

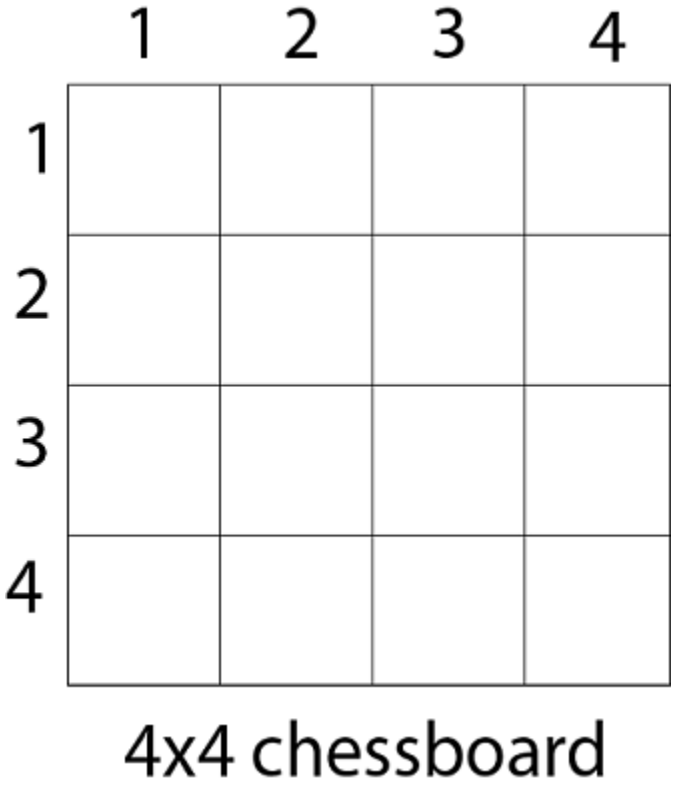
Thus, if our partial solution elements sum is equal to the positive integer 'X' then at that time search will terminate, or it continues if all the possible solution needs to be obtained.

N-Queens Problem

N - Queens problem is to place n - queens in such a manner on an n x n chessboard that no queens attack each other by being in the same row, column or diagonal.

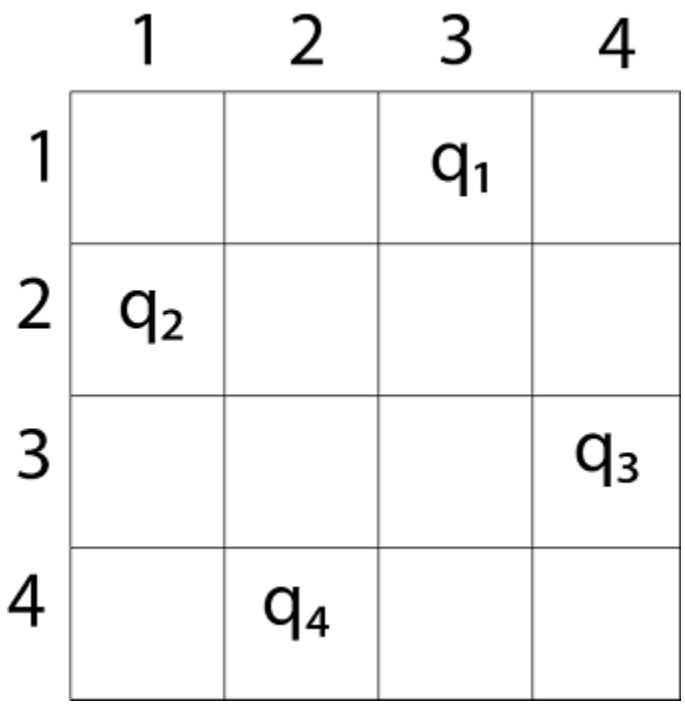
It can be seen that for n =1, the problem has a trivial solution, and no solution exists for n =2 and n =3. So first we will consider the 4 queens problem and then generate it to n - queens problem.

Given a 4 x 4 chessboard and number the rows and column of the chessboard 1 through 4.



Since, we have to place 4 queens such as q_1 q_2 q_3 and q_4 on the chessboard, such that no two queens attack each other. In such a conditional each queen must be placed on a different row, i.e., we put queen "i" on row "i."

Now, we place queen q_1 in the very first acceptable position (1, 1). Next, we put queen q_2 so that both these queens do not attack each other. We find that if we place q_2 in column 1 and 2, then the dead end is encountered. Thus the first acceptable position for q_2 in column 3, i.e. (2, 3) but then no position is left for placing queen 'q_3' safely. So we backtrack one step and place the queen 'q_2' in (2, 4), the next best possible solution. Then we obtain the position for placing 'q_3' which is (3, 2). But later this position also leads to a dead end, and no place is found where 'q_4' can be placed safely. Then we have to backtrack till 'q_1' and place it to (1, 2) and then all other queens are placed safely by moving q_2 to (2, 4), q_3 to (3, 1) and q_4 to (4, 3). That is, we get the solution (2, 4, 1, 3). This is one possible solution for the 4-queens problem. For another possible solution, the whole method is repeated for all partial solutions. The other solutions for 4 - queens problems is (3, 1, 4, 2) i.e.



The implicit tree for 4 - queen problem for a solution (2, 4, 1, 3) is as follows:

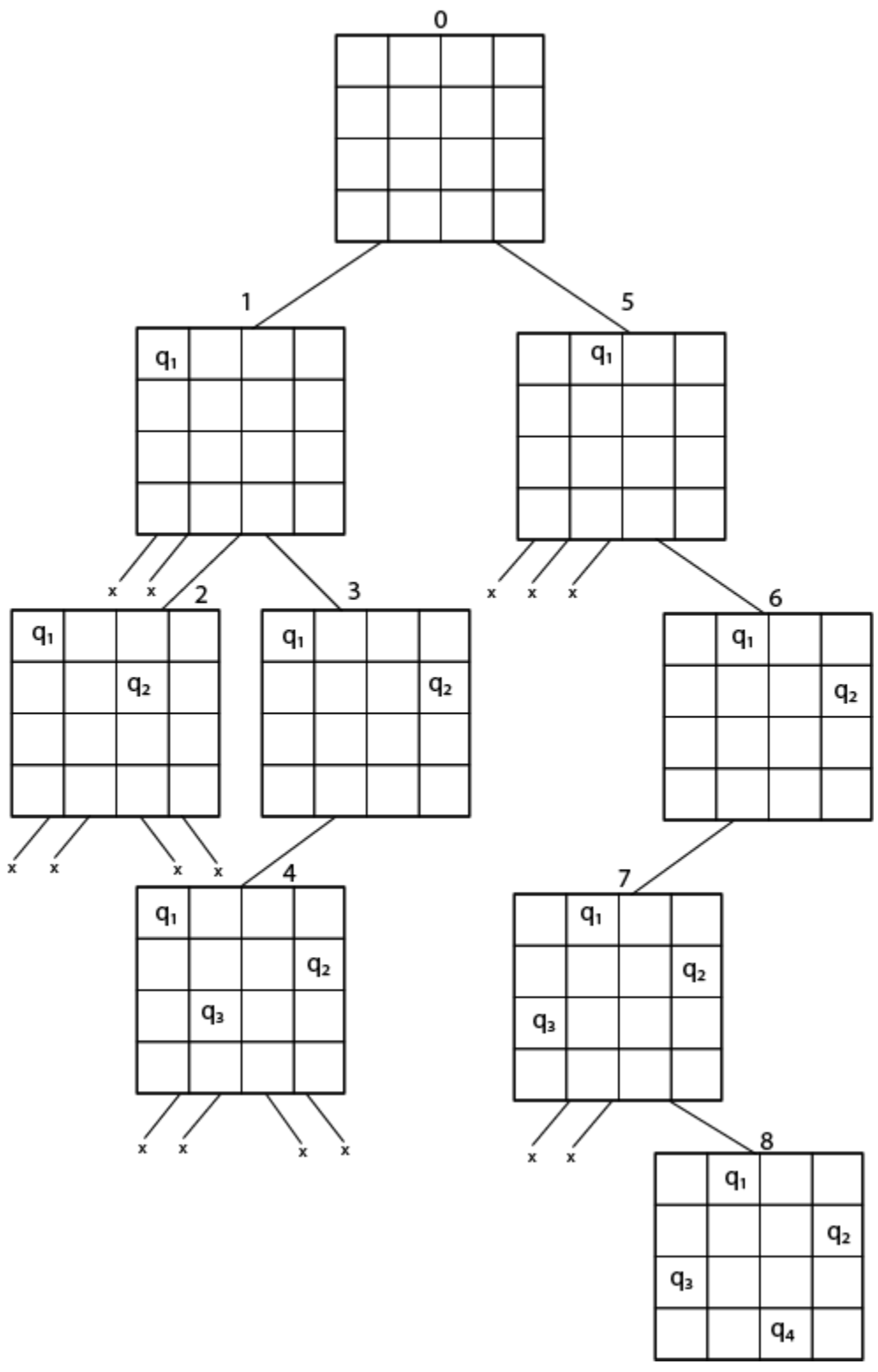
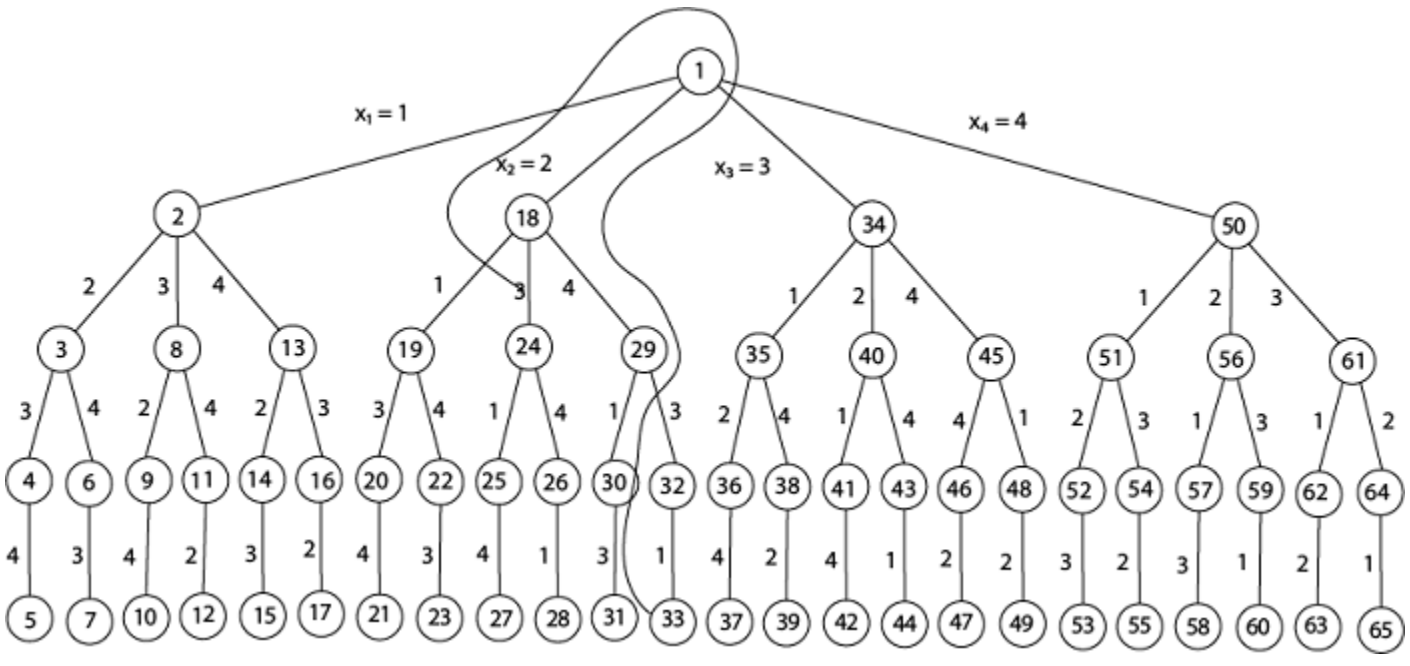


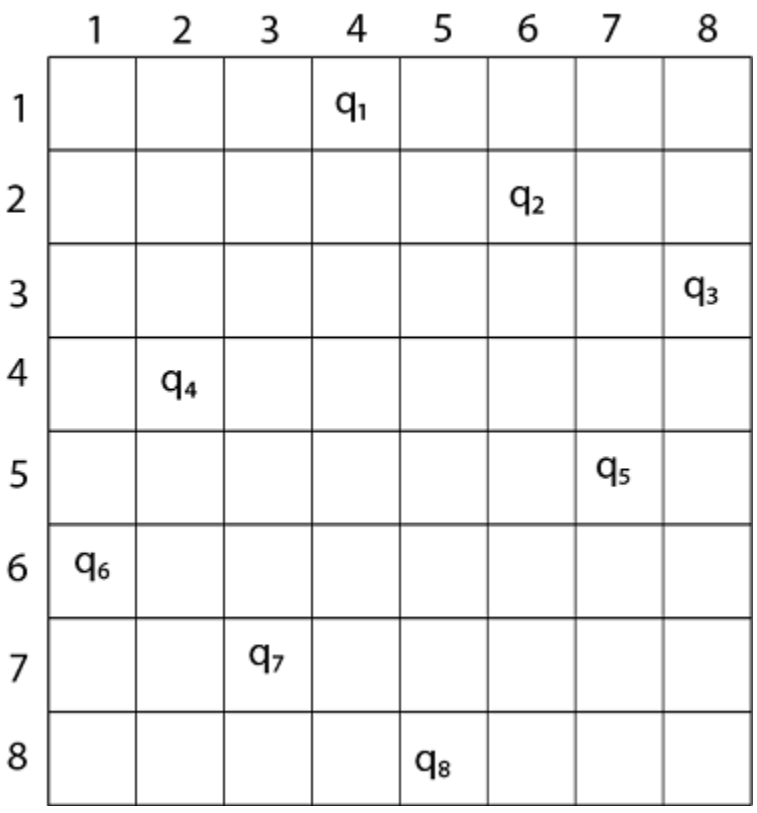
Fig shows the complete state space for 4 - queens problem. But we can use backtracking method to generate the necessary node and stop if the next node violates the rule, i.e., if two queens are attacking.



4 - Queens solution space with nodes numbered in DFS

It can be seen that all the solutions to the 4 queens problem can be represented as 4 - tuples (x_1, x_2, x_3, x_4) where x_i represents the column on which queen "q_i" is placed.

One possible solution for 8 queens problem is shown in fig:



1. Thus, the solution **for** 8 -queen problem **for** (4, 6, 8, 2, 7, 1, 3, 5).
2. If two queens are placed at position (i, j) and (k, l).
3. Then they are on same diagonal only **if** (i - j) = k - l or i + j = k + l.
4. The first equation implies that j - l = i - k.
5. The second equation implies that j - l = k - i.
6. Therefore, two queens lie on the duplicate diagonal **if** and only **if** |j-l|=|i-k|

Place (k, i) returns a Boolean value that is true if the kth queen can be placed in column i. It tests both whether i is distinct from all previous costs x_1, x_2,....x_{k-1} and whether there is no other queen on the same diagonal.

Using place, we give a precise solution to then n- queens problem.

```
1. Place (k, i)
2.   {
3.     For j ← 1 to k - 1
4.      do if (x [j] = i)
5.       or (Abs x [j]) - i) = (Abs (j - k))
6.     then return false;
7.      return true;
```

```
8.  }
```

Place (k, i) return true if a queen can be placed in the kth row and ith column otherwise return is false.

x [] is a global array whose final k - 1 values have been set. Abs (r) returns the absolute value of r.

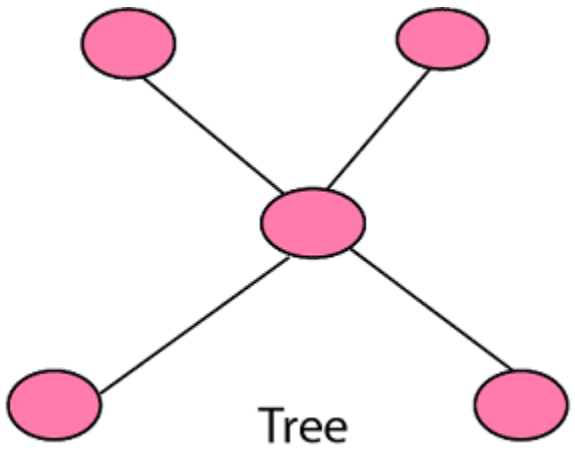
```
1.  N - Queens (k, n)
2.  {
3.     For i  ←  1 to n
4.          do if Place (k, i) then
5.     {
6.        x [k]  ←  i;
7.        if (k ==n) then
8.          write (x [1....n));
9.        else
10.       N - Queens (k + 1, n);
11.    }
12. }
```

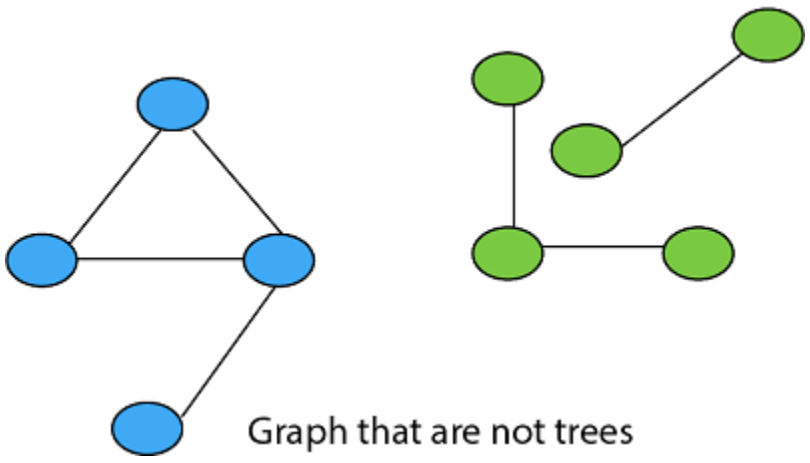
Introduction of Minimum Spanning Tree

Tree:

A tree is a graph with the following properties:

1. The graph is connected (can go from anywhere to anywhere)
2. There are no cyclic (Acyclic)



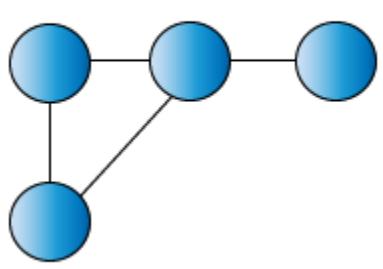


Spanning Tree:

Given a connected undirected graph, a spanning tree of that graph is a subgraph that is a tree and joined all vertices. A single graph can have many spanning trees.

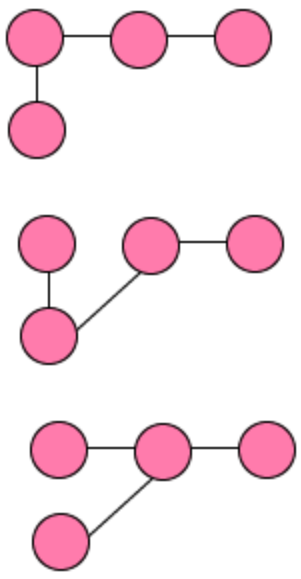
For Example:

Connected Undirected Graph



For the above-connected graph. There can be multiple spanning Trees like

Spanning Trees



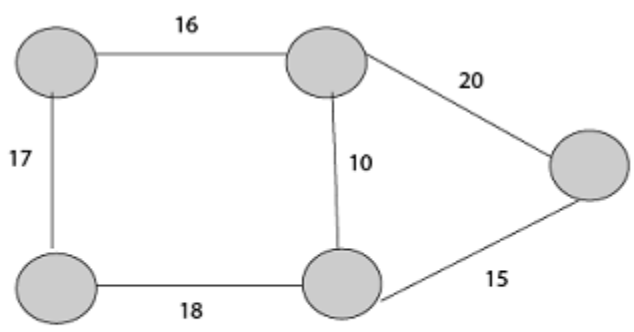
Properties of Spanning Tree:

1. There may be several minimum spanning trees of the same weight having the minimum number of edges.
2. If all the edge weights of a given graph are the same, then every spanning tree of that graph is minimum.
3. If each edge has a distinct weight, then there will be only one, unique minimum spanning tree.
4. A connected graph G can have more than one spanning trees.
5. A disconnected graph can't have to span the tree, or it can't span all the vertices.
6. Spanning Tree doesn't contain cycles.
7. Spanning Tree has **(n-1) edges** where n is the number of vertices.

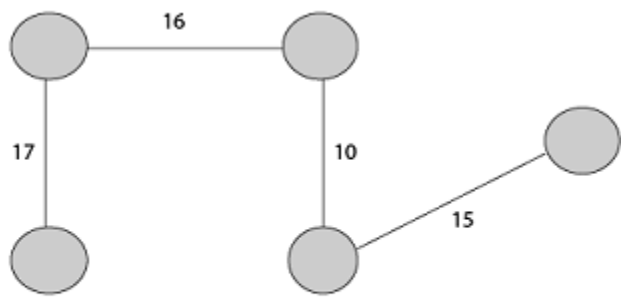
Addition of even one single edge results in the spanning tree losing its property of **Acyclicity** and elimination of one single edge results in its losing the property of connectivity.

Minimum Spanning Tree:

Minimum Spanning Tree is a Spanning Tree which has minimum total cost. If we have a linked undirected graph with a weight (or cost) combine with each edge. Then the cost of spanning tree would be the sum of the cost of its edges.



Connected , Undirected Graph



Minimum Cost Spanning Tree

Total Cost = 17+16+10+15=58

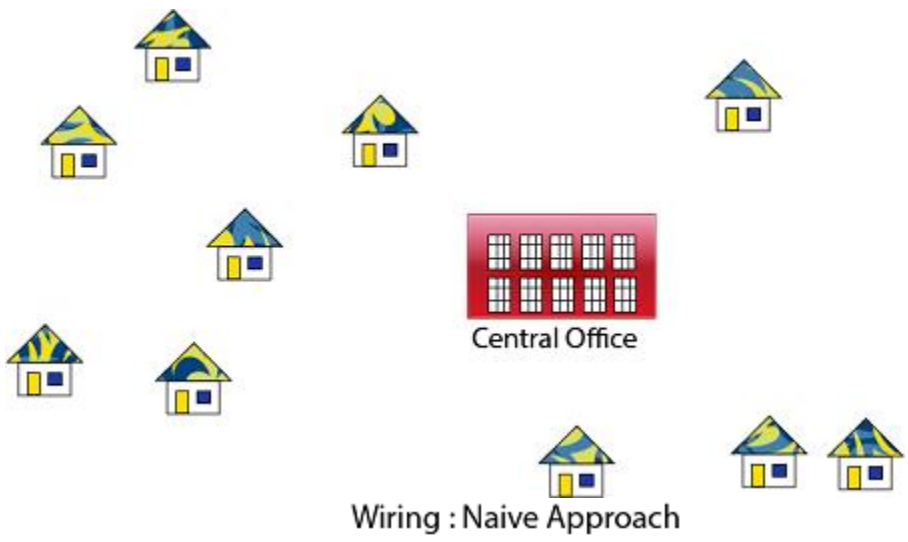
Application of Minimum Spanning Tree

1. Consider n stations are to be linked using a communication network & laying of communication links between any two stations involves a cost.
   The ideal solution would be to extract a subgraph termed as minimum cost spanning tree.

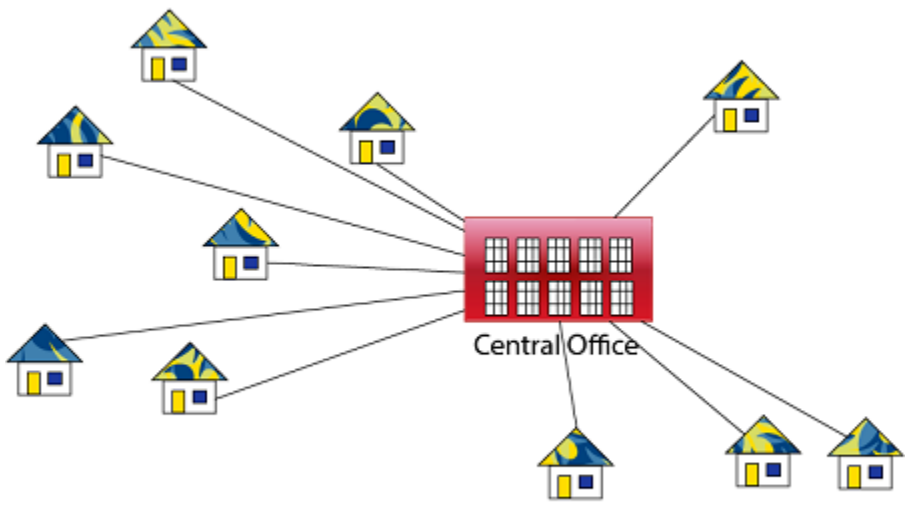
2. Suppose you want to construct highways or railroads spanning several cities then we can use the concept of minimum spanning trees.
3. Designing Local Area Networks.
4. Laying pipelines connecting offshore drilling sites, refineries and consumer markets.
5. Suppose you want to apply a set of houses with
    - Electric Power
    - Water
    - Telephone lines
    - Sewage lines

To reduce cost, you can connect houses with minimum cost spanning trees.

For Example, Problem laying Telephone Wire.



Wiring : Naive Approach



Expensive!
Wiring : Better Approach

Minimize the total length of wire connecting the customers

Methods of Minimum Spanning Tree

There are two methods to find Minimum Spanning Tree

1. Kruskal's Algorithm
2. Prim's Algorithm

Kruskal's Algorithm:

An algorithm to construct a Minimum Spanning Tree for a connected weighted graph. It is a Greedy Algorithm. The Greedy Choice is to put the smallest weight edge that does not because a cycle in the MST constructed so far.

If the graph is not linked, then it finds a Minimum Spanning Tree.

Steps for finding MST using Kruskal's Algorithm:

1. Arrange the edge of G in order of increasing weight.
2. Starting only with the vertices of G and proceeding sequentially add each edge which does not result in a cycle, until (n - 1) edges are used.
3. EXIT.

MST- KRUSKAL (G, w)

```
1. A ← Ø
2. for each vertex v ∈ V [G]
3. do MAKE - SET (v)
4. sort the edges of E into non decreasing order by weight w
5. for each edge (u, v) ∈ E, taken in non decreasing order by weight
6. do if FIND-SET (μ) ≠ if FIND-SET (v)
7. then A  ←  A ∪ {(u, v)}
8. UNION (u, v)
9. return A
```

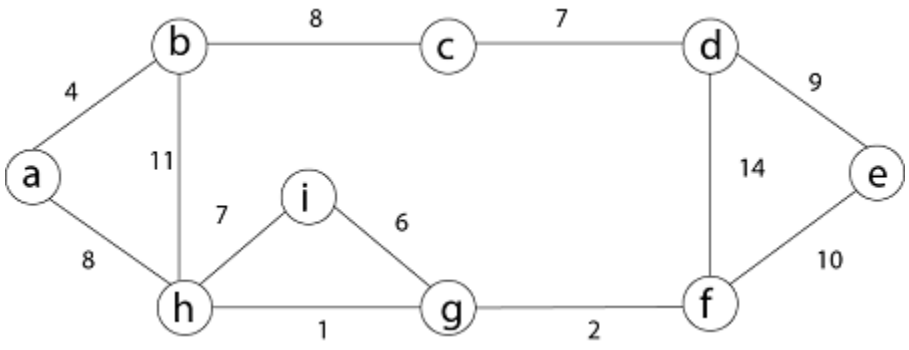
Analysis: Where E is the number of edges in the graph and V is the number of vertices, Kruskal's Algorithm can be shown to run in O (E log E) time, or simply, O (E log V) time, all with simple data structures. These running times are equivalent because:

- E is at most V^2 and log V^2= 2 x log V is O (log V).
- If we ignore isolated vertices, which will each their components of the minimum spanning tree, V ≤ 2 E, so log V is O (log E).

Thus the total time is

1. O (E log E) = O (E log V).

For Example: Find the Minimum Spanning Tree of the following graph using Kruskal's algorithm.



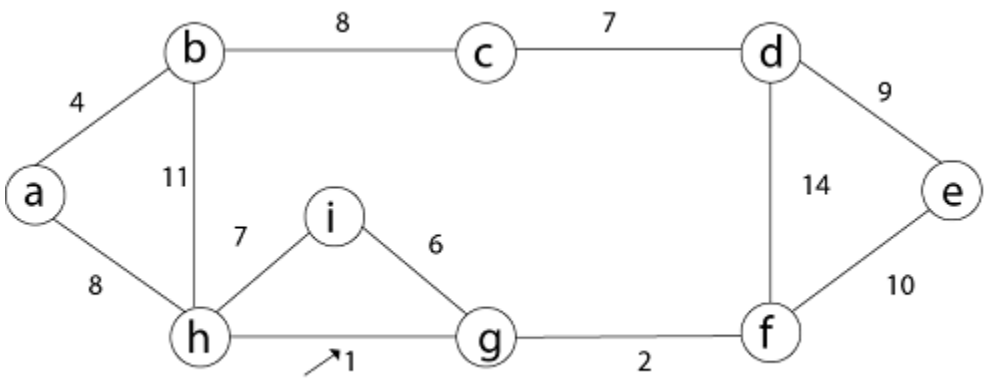
Solution: First we initialize the set A to the empty set and create |v| trees, one containing each vertex with MAKE-SET procedure. Then sort the edges in E into order by non-decreasing weight.

There are 9 vertices and 12 edges. So MST formed (9-1) = 8 edges

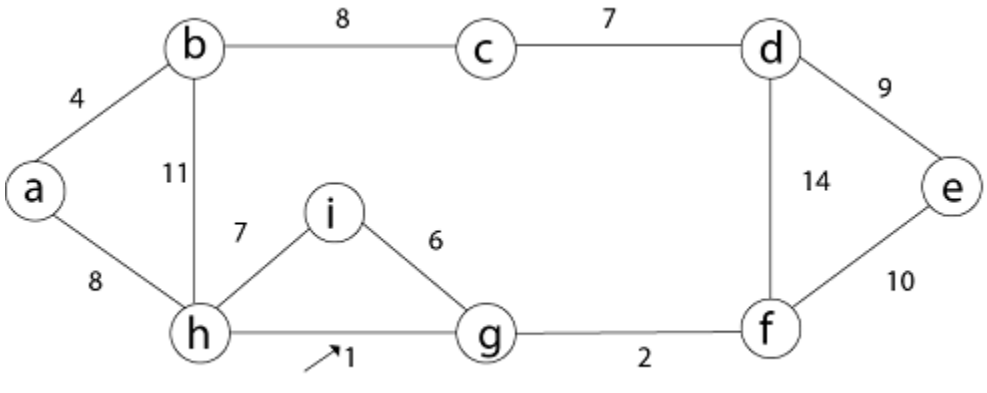
Weight	Source	Destination
1	h	g
2	g	f
4	a	b
6	i	g
7	h	i
7	c	d
8	b	c
8	a	h
9	d	e
10	e	f
11	b	h
14	d	f

Now, check for each edge (u, v) whether the endpoints u and v belong to the same tree. If they do then the edge (u, v) cannot be supplementary. Otherwise, the two vertices belong to different trees, and the edge (u, v) is added to A, and the vertices in two trees are merged in by union procedure.

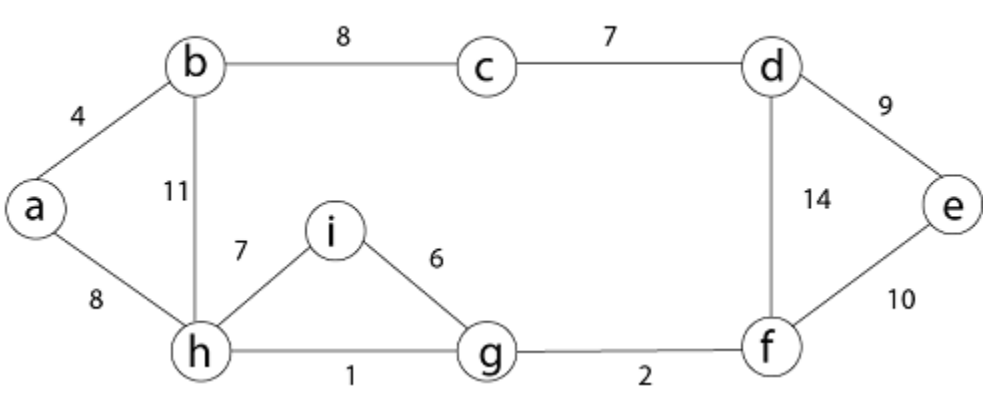
Step1: So, first take (h, g) edge



Step 2: then (g, f) edge.

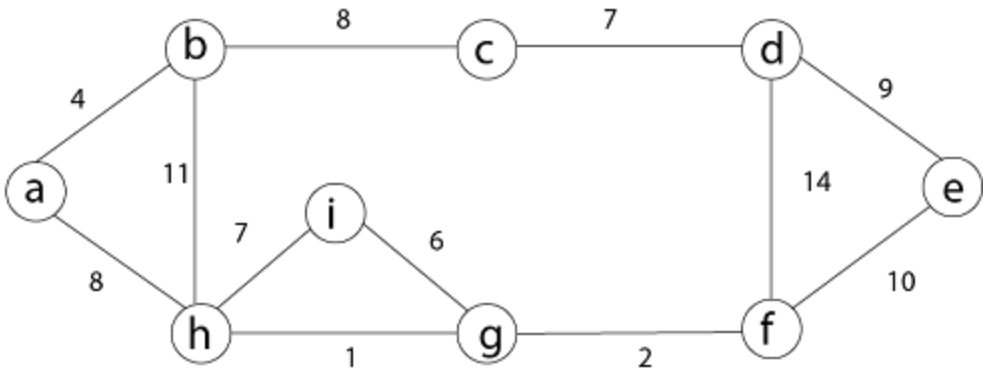


Step 3: then (a, b) and (i, g) edges are considered, and the forest becomes



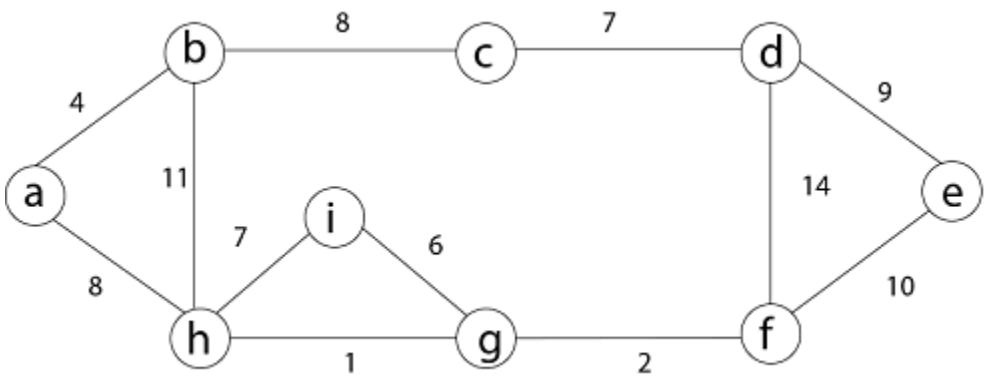
Step 4: Now, edge (h, i). Both h and i vertices are in the same set. Thus it creates a cycle. So this edge is discarded.

Then edge (c, d), (b, c), (a, h), (d, e), (e, f) are considered, and the forest becomes.



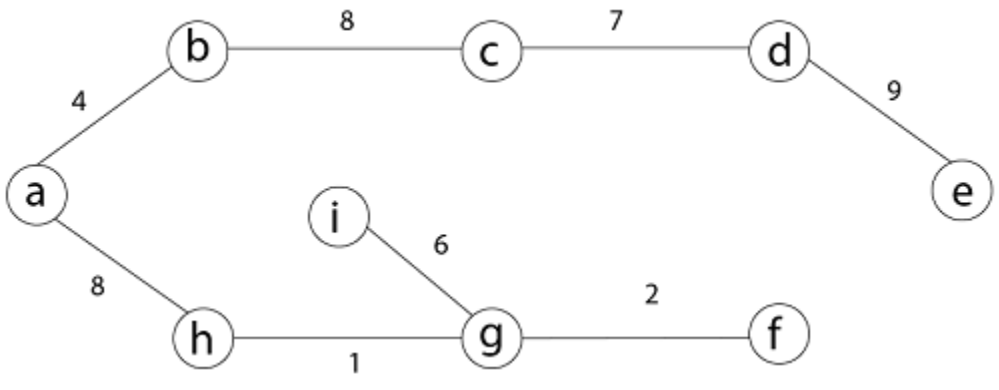
Step 5: In (e, f) edge both endpoints e and f exist in the same tree so discarded this edge. Then (b, h) edge, it also creates a cycle.

Step 6: After that edge (d, f) and the final spanning tree is shown as in dark lines.



Step 7: This step will be required Minimum Spanning Tree because it contains all the 9 vertices and (9 - 1) = 8 edges

1. e → f, b → h, d → f [cycle will be formed]



Minimum Cost MST

Prim's Algorithm

It is a greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices:

- Contain vertices already included in MST.
- Contain vertices not yet included.

At every step, it considers all the edges and picks the minimum weight edge. After picking the edge, it moves the other endpoint of edge to set containing MST.

Steps for finding MST using Prim's Algorithm:

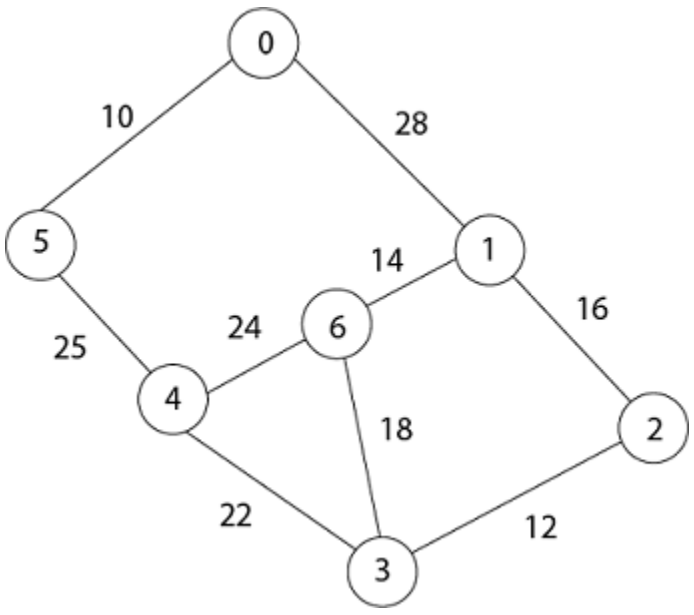
1. Create MST set that keeps track of vertices already included in MST.
2. Assign key values to all vertices in the input graph. Initialize all key values as INFINITE (∞). Assign key values like 0 for the first vertex so that it is picked first.
3. While MST set doesn't include all vertices.
    a. Pick vertex u which is not is MST set and has minimum key value. Include 'u'to MST set.
    b. Update the key value of all adjacent vertices of u. To update, iterate through all adjacent vertices. For every adjacent vertex v, if the weight of edge u.v less than the previous key value of v, update key value as a weight of u.v.

MST-PRIM (G, w, r)

```
1. for each u ∈ V [G]
2. do key [u] ← ∞
```

```
3. π [u] ← NIL
4. key [r] ← 0
5. Q ← V [G]
6. While Q ? Ø
7. do u ← EXTRACT - MIN (Q)
8. for each v ∈ Adj [u]
9. do if v ∈ Q and w (u, v) < key [v]
10. then π [v] ← u
11. key [v] ← w (u, v)
```

Example: Generate minimum cost spanning tree for the following graph using Prim's algorithm.



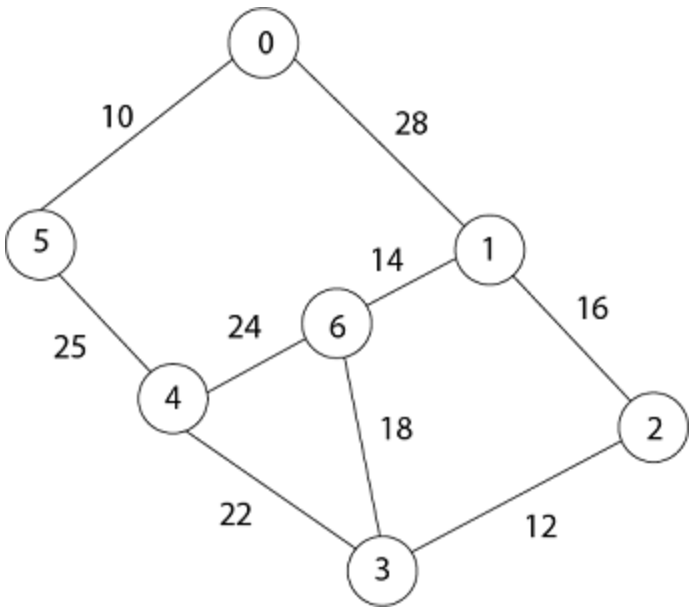
Solution: In Prim's algorithm, first we initialize the priority Queue Q. to contain all the vertices and the key of each vertex to ∞ except for the root, whose key is set to 0. Suppose 0 vertex is the root, i.e., r. By EXTRACT - MIN (Q) procure, now u = r and Adj [u] = {5, 1}.

Removing u from set Q and adds it to set V - Q of vertices in the tree. Now, update the key and π fields of every vertex v adjacent to u but not in a tree.

Vertex	0	1	2	3	4	5	6
Key Value	0	∞	∞	∞	∞	∞	∞
Parent	NIL	NIL	NIL	NIL	NIL	NIL	NIL

1. Taking 0 as starting vertex
2. Root = 0
3. Adj [0] = 5, 1
4. Parent, π [5] = 0 and π [1] = 0
5. Key [5] = ∞ and key [1] = ∞
6. w [0, 5) = 10 and w (0,1) = 28
7. w (u, v) < key [5] , w (u, v) < key [1]
8. Key [5] = 10 and key [1] = 28
9. So update key value of 5 and 1 is:

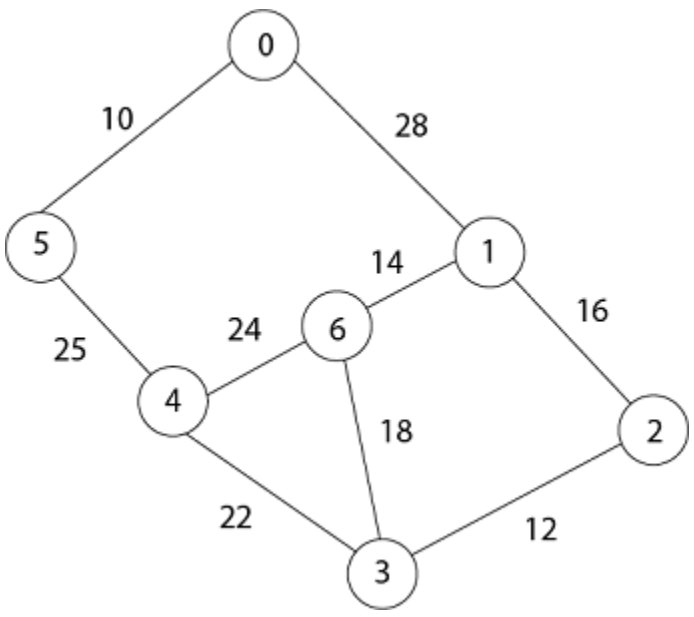
Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	∞	∞	10	∞
Parent	NIL	0	NIL	NIL	NIL	0	NIL



Now by EXTRACT_MIN (Q) Removes 5 because key [5] = 10 which is minimum so u = 5.

1. Adj [5] = {0, 4} and 0 is already in heap
2. Taking 4, key [4] = ∞ π [4] = 5
3. (u, v) < key [v] then key [4] = 25
4. w (5,4) = 25
5. w (5,4) < key [4]
6. date key value and parent of 4.

Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	∞	25	10	∞
Parent	NIL	0	NIL	NIL	5	0	NIL



Now remove 4 because key [4] = 25 which is minimum, so u =4

1. Adj [4] = {6, 3}
2. Key [3] = ∞ key [6] = ∞
3. w (4,3) = 22 w (4,6) = 24
4. w (u, v) < key [v] w (u, v) < key [v]
5. w (4,3) < key [3] w (4,6) < key [6]

Update key value of key [3] as 22 and key [6] as 24.

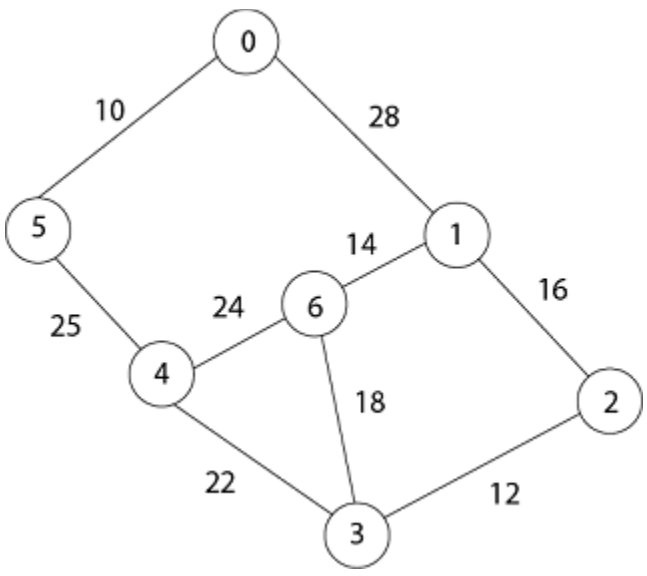
And the parent of 3, 6 as 4.

1. π[3]= 4 π[6]= 4

Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	22	25	10	24
Parent	NIL	0	NIL	4	5	0	4

1. u = EXTRACT_MIN (3, 6) [key [3] < key [6]]
2. u = 3 i.e. 22 < 24

Now remove 3 because key [3] = 22 is minimum so u =3.



1. Adj [3] = {4, 6, 2}
2. 4 is already in heap
3. 4 ≠ Q key [6] = 24 now becomes key [6] = 18
4. Key [2] = ∞ key [6] = 24
5. w (3, 2) = 12 w (3, 6) = 18
6. w (3, 2) < key [2] w (3, 6) < key [6]

Now in Q, key [2] = 12, key [6] = 18, key [1] = 28 and parent of 2 and 6 is 3.

1. π [2] = 3 π[6]=3

Now by EXTRACT_MIN (Q) Removes 2, because key [2] = 12 is minimum.

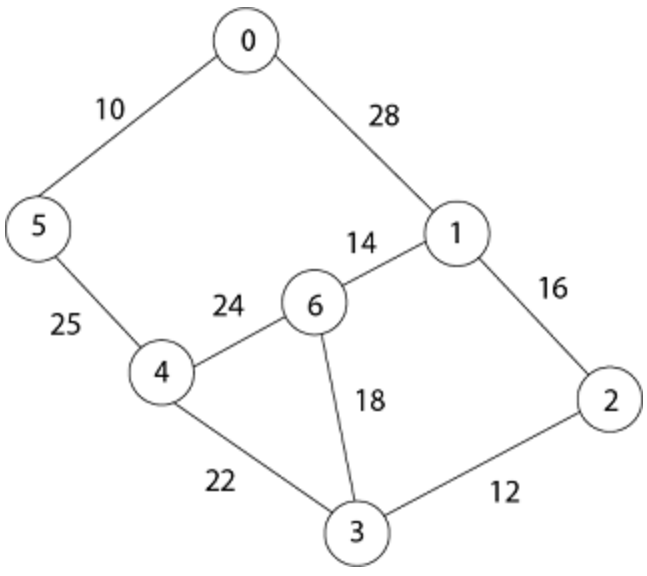
Vertex	0	1	2	3	4	5	6
Key Value	0	28	12	22	25	10	18
Parent	NIL	0	3	4	5	0	3

1. u = EXTRACT_MIN (2, 6)
2. u = 2 [key [2] < key [6]]
3. 12 < 18
4. Now the root is 2
5. Adj [2] = {3, 1}
6. 3 is already in a heap
7. Taking 1, key [1] = 28
8. w (2,1) = 16
9. w (2,1) < key [1]

So update key value of key [1] as 16 and its parent as 2.

1. π[1]= 2

Vertex	0	1	2	3	4	5	6
Key Value	0	16	12	22	25	10	18
Parent	NIL	2	3	4	5	0	3



Now by EXTRACT_MIN (Q) Removes 1 because key [1] = 16 is minimum.

1. Adj [1] = {0, 6, 2}
2. 0 and 2 are already in heap.
3. Taking 6, key [6] = 18
4. w [1, 6] = 14
5. w [1, 6] < key [6]

Update key value of 6 as 14 and its parent as 1.

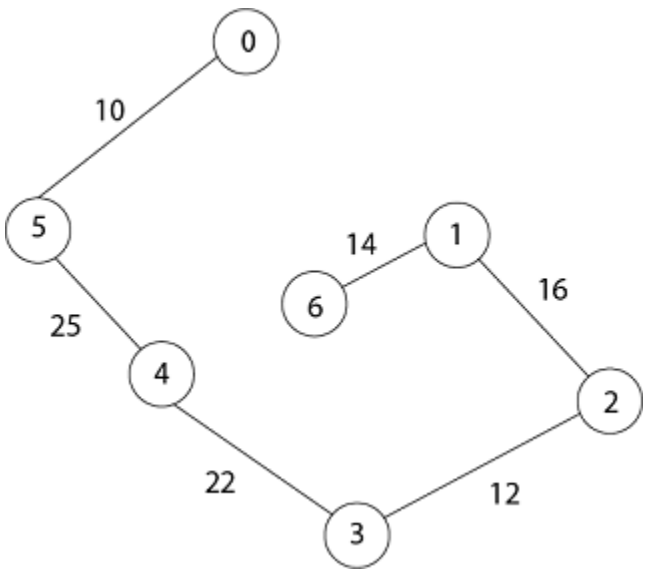
1. Π [6] = 1

Vertex	0	1	2	3	4	5	6
Key Value	0	16	12	22	25	10	14
Parent	NIL	2	3	4	5	0	1

Now all the vertices have been spanned, Using above the table we get Minimum Spanning Tree.

1. 0 → 5 → 4 → 3 → 2 → 1 → 6
2. [Because Π [5] = 0, Π [4] = 5, Π [3] = 4, Π [2] = 3, Π [1] =2, Π [6] =1]

Thus the final spanning Tree is



Total Cost = 10 + 25 + 22 + 12 + 16 + 14 = 99

All-Pairs Shortest Paths

Introduction

It aims to figure out the shortest path from each vertex v to every other u. Storing all the paths explicitly can be very memory expensive indeed, as we need one spanning tree for each vertex. This is often impractical regarding memory consumption, so these are generally considered as all pairs-shortest distance problems, which aim to find just the distance from each to each node to another. We usually want the output in tabular form: the entry in u's row and v's column should be the weight of the shortest path from u to v.

Three approaches for improvement:

Algorithm	**Cost**
Matrix Multiplication	O (V^3 logV)
Floyd-Warshall	O (V^3)
Johnson O	(V^2 log?V+VE)

Unlike the single-source algorithms, which assume an adjacency list representation of the graph, most of the algorithm uses an adjacency matrix representation. (Johnson's Algorithm for sparse graphs uses adjacency lists.) The input is a n x n matrix W representing the edge weights of an n-vertex directed graph G = (V, E). That is, W = (w_{ij}), where

$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ w(i,j) & \text{if } i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases}$$

Floyd-Warshall Algorithm

Let the vertices of G be V = {1, 2.....n} and consider a subset {1, 2.....k} of vertices for some k. For any pair of vertices i, j ∈ V, considered all paths from i to j whose intermediate vertices are all drawn from {1, 2.....k}, and let p be a minimum weight path from amongst them. The Floyd-Warshall algorithm exploits a link between path p and shortest paths from i to j with all intermediate vertices in the set {1, 2.....k-1}. The link depends on whether or not k is an intermediate vertex of path p.

If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set {1, 2.....k-1}. Thus, the shortest path from vertex i to vertex j with all intermediate vertices in the set {1, 2.....k-1} is also the shortest path i to j with all intermediate vertices in the set {1, 2.....k}.

If k is an intermediate vertex of path p, then we break p down into i → k → j.

Let $d_{ij}^{(k)}$ be the weight of the shortest path from vertex i to vertex j with all intermediate vertices in the set {1, 2.....k}.

A recursive definition is given by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

FLOYD - WARSHALL (W)

1. n ← rows [W].
2. D^0 ← W
3. for k ← 1 to n
4. do for i ← 1 to n
5. do for j ← 1 to n
6. do $d_{ij}^{(k)}$ ← min ($d_{ij}^{(k-1)}$, $d_{ik}^{(k-1)}$+$d_{kj}^{(k-1)}$)
7. return $D^{(n)}$

The strategy adopted by the Floyd-Warshall algorithm is **Dynamic Programming**. The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops of lines 3-6. Each execution of line 6 takes O (1) time. The algorithm thus runs in time θ(n^3).

Example: Apply Floyd-Warshall algorithm for constructing the shortest path. Show that matrices $D^{(k)}$ and $\pi^{(k)}$ computed by the Floyd-Warshall algorithm for the graph.

Solution:

$$d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} \text{ if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} \text{ if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

Step (i) When k = 0

$$D^{(0)} = \begin{matrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & -5 & \infty \\ 2 & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{matrix} \qquad \pi^{(0)} = \begin{matrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 3 & \text{NIL} \\ 4 & \text{NIL} & \text{NIL} & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{matrix}$$

Step (ii) When k =1

$d_{ij}^{(k)} = \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

$d_{14}^{(1)} = \min (d_{14}^{(0)}, d_{11}^{(0)} + d_{14}^{(0)})$

$d_{14}^{(1)} = \min (\infty , 0 + \infty) = \infty$

$d_{15}^{(1)} = \min (d_{15}^{(0)}, d_{11}^{(0)} + d_{15}^{(0)})$

$d_{15}^{(1)} = \min (-4, 0 + -4) = -4$

$d_{21}^{(1)} = \min (d_{21}^{(0)}, d_{21}^{(0)} + d_{11}^{(0)})$

$d_{21}^{(1)} = \min (\infty, \infty + 0) = \infty$

$d_{23}^{(1)} = \min (d_{23}^{(0)}, d_{21}^{(0)} + d_{13}^{(0)})$

$d_{23}^{(1)} = \min ((\infty, \infty + 8) = \infty$

$d_{31}^{(1)} = \min (d_{31}^{(0)}, d_{31}^{(0)} + d_{11}^{(0)})$

$d_{31}^{(1)} = \min (\infty, \infty + 0) = \infty$

$d_{35}^{(1)} = \min (d_{35}^{(0)}, d_{31}^{(0)} + d_{15}^{(0)})$

$d_{35}^{(1)} = \min (\infty, \infty + (-4)) = \infty$

$d_{42}^{(1)} = \min (d_{42}^{(0)}, d_{41}^{(0)} + d_{12}^{(0)})$

$d_{42}^{(1)} = \min (\infty, 2 + 3) = 5$

$d_{43}^{(1)} = \min (d_{43}^{(0)}, d_{41}^{(0)} + d_{13}^{(0)})$

$d_{43}^{(1)} = \min (\infty, 2 + 8) = 10$

$d_{45}^{(1)} = \min (d_{45}^{(0)}, d_{41}^{(0)} + d_{15}^{(0)})$

$d_{45}^{(1)} = \min (\infty, 2 + (-4)) = -2$

$d_{51}^{(1)} = \min (d_{51}^{(0)}, d_{51}^{(0)} + d_{11}^{(0)})$

$d_{51}^{(1)} = \min (\infty, \infty + 0) = \infty$

$$D_{ij}^{(1)} = \begin{matrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & -5 & \infty \\ 2 & 5 & 10 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{matrix} \qquad \pi^{(1)} = \begin{matrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 3 & NIL \\ 4 & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{matrix}$$

Step (iii) When k = 2

$d_{ij}^{(k)} = \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

$d_{14}^{(2)} = \min (d_{14}^{(1)}, d_{12}^{(1)} + d_{24}^{(1)})$

$d_{14}^{(2)} = \min (\infty, 3 + 1) = 4$

$d_{21}^{(2)} = \min (d_{21}^{(1)}, d_{22}^{(1)} + d_{21}^{(1)})$

$d_{21}^{(2)} = \min (\infty , 0 + \infty) = \infty$

$d_{34}^{(2)} = \min (d_{34}^{(1)}, d_{32}^{(1)} + d_{24}^{(1)})$

$d_{34}^{(2)} = \min (-5, 4 + 1) = -5$

$d_{35}^{(2)} = \min (d_{35}^{(1)}, d_{32}^{(1)} + d_{25}^{(1)})$

$d_{35}^{(2)} = \min (\infty, 4 + 7) = 11$

$d_{43}^{(2)} = \min (d_{43}^{(1)}, d_{42}^{(1)} + d_{23}^{(1)})$

$d_{43}^{(2)} = \min (10, 5 + \infty) = 10$

$D_{ij}^{(2)} =$

0	3	8	4	-4
∞	0	∞	1	7
∞	4	0	-5	11
2	5	10	0	-2
∞	∞	∞	6	0

$\pi^{(2)} =$

NIL	1	1	2	1
NIL	NIL	NIL	2	2
NIL	3	NIL	3	2
4	1	1	NIL	1
NIL	NIL	NIL	5	NIL

Step (iv) When k = 3

$d_{ij}^{(k)} = \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

$d_{14}^{(3)} = \min (d_{14}^{(2)}, d_{13}^{(2)} + d_{34}^{(2)})$

$d_{14}^{(3)} = \min (4, 8 + (-5)) = 3$

$D_{ij}^{(3)} =$

0	3	8	3	-4
∞	0	∞	1	7
∞	4	0	-5	11
2	5	10	0	-2
∞	∞	∞	6	0

$\pi^{(3)} =$

NIL	1	1	3	1
NIL	NIL	NIL	2	2
NIL	3	NIL	3	2
4	1	1	NIL	1
NIL	NIL	NIL	5	NIL

Step (v) When k = 4

$d_{ij}^{(k)} = \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

$d_{21}^{(4)} = \min (d_{21}^{(3)}, d_{24}^{(3)} + d_{41}^{(3)})$

$d_{21}^{(4)} = \min (\infty, 1 + 2) = 3$

$d_{23}^{(4)} = \min (d_{23}^{(3)}, d_{24}^{(3)} + d_{43}^{(3)})$

$d_{23}^{(4)} = \min (\infty, 1 + 10) = 11$

$d_{25}^{(4)} = \min (d_{25}^{(3)}, d_{24}^{(3)} + d_{45}^{(3)})$

$d_{25}^{(4)} = \min (7, 1 + (-2)) = -1$

$d_{31}^{(4)} = \min (d_{31}^{(3)}, d_{34}^{(3)} + d_{41}^{(3)})$

$d_{31}^{(4)} = \min (\infty, -5 + 2) = -3$

$d_{32}^{(4)} = \min (d_{32}^{(3)}, d_{34}^{(3)} + d_{42}^{(3)})$

$d_{32}^{(4)} = \min (4, -5 + 5) = 0$

$d_{51}^{(4)} = \min (d_{51}^{(3)}, d_{54}^{(3)} + d_{41}^{(3)})$

$d_{51}^{(4)} = \min (\infty, 6 + 2) = 8$

$d_{52}^{(4)} = \min (d_{52}^{(3)}, d_{54}^{(3)} + d_{42}^{(3)})$

$d_{52}^{(4)} = \min (\infty, 6 + 5) = 11$

$d_{53}^{(4)} = \min (d_{53}^{(3)}, d_{54}^{(3)} + d_{43}^{(3)})$

$d_{53}^{(4)} = \min (\infty, 6 + 10) = 16$

$D_{ij}^{(4)} =$

0	3	8	3	-4
3	0	11	1	-1
-3	0	0	-5	-7
2	5	10	0	-2
8	11	16	6	0

$\pi^{(4)} =$

NIL	1	1	3	1
4	NIL	4	2	2
4	4	NIL	3	4
4	1	1	NIL	1
4	4	4	5	NIL

Step (vi) When k = 5

$d_{ij}^{(k)} = \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

$d_{25}^{(5)} = \min (d_{25}^{(4)}, d_{25}^{(4)} + d_{55}^{(3)})$

$d_{25}^{(5)} = \min (-1, -1 + 0) = -1$

$d_{23}^{(5)} = \min (d_{23}^{(4)}, d_{25}^{(4)} + d_{53}^{(3)})$

$d_{23}^{(5)} = \min (11, -1 + 16) = 11$

$d_{35}^{(5)} = \min (d_{35}^{(4)}, d_{35}^{(4)} + d_{55}^{(3)})$

$d_{35}^{(5)} = \min (-7, -7 + 0) = -7$

$D_{ij}^{(5)} =$

0	3	8	3	-4
3	0	11	1	-1
-3	0	0	-5	-7
2	5	10	0	-2
8	11	16	6	0

$\pi^{(5)} =$

NIL	1	1	5	1
4	NIL	4	2	4
4	4	NIL	3	4
4	1	1	NIL	1
4	4	4	5	NIL

TRANSITIVE- CLOSURE (G)

1. n ← |V[G]|
2. for i ← 1 to n
3. do for j ← 1 to n
4. do if i = j or (i, j) ∈ E [G]
5. the $t_{ij}^{(0)} \leftarrow 1$
6. else $t_{ij}^{(0)} \leftarrow 0$
7. for k ← 1 to n
8. do for i ← 1 to n
9. do for j ← 1 to n
10. dod $_{ij}{}^{(k)} \leftarrow t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$
11. Return $T^{(n)}$.

Single Source Shortest Paths

Introduction:

In a **shortest- paths problem**, we are given a weighted, directed graphs G = (V, E), with weight function **w: E → R** mapping edges to real-valued weights. The weight of path p = (v_0,v_1,..... v_k) is the total of the weights of its constituent edges:

$$w(P) = \sum_{i=1}^{k} w(v_{i-1}v_i)$$

We define the shortest - path weight from u to v by δ(u,v) = min (w (p): u→v), if there is a path from u to v, and δ(u,v)= ∞, otherwise.

The **shortest path** from vertex s to vertex t is then defined as any path p with weight w (p) = δ(s,t).

The **breadth-first- search algorithm** is the shortest path algorithm that works on unweighted graphs, that is, graphs in which each edge can be considered to have unit weight.

In a **Single Source Shortest Paths Problem**, we are given a Graph G = (V, E), we want to find the shortest path from a given source vertex s ∈ V to every vertex v ∈ V.

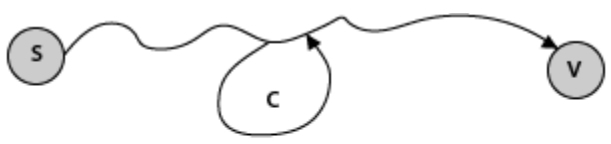
Variants:

There are some variants of the shortest path problem.

- **Single- destination shortest - paths problem:** Find the shortest path to a given destination vertex t from every vertex v. By shift the direction of each edge in the graph, we can shorten this problem to a single - source problem.
- **Single - pair shortest - path problem:** Find the shortest path from u to v for given vertices u and v. If we determine the single - source problem with source vertex u, we clarify this problem also. Furthermore, no algorithms for this problem are known that run asymptotically faster than the best single - source algorithms in the worst case.
- **All - pairs shortest - paths problem:** Find the shortest path from u to v for every pair of vertices u and v. Running a single - source algorithm once from each vertex can clarify this problem; but it can generally be solved faster, and its structure is of interest in the own right.

Shortest Path: Existence:

If some path from s to v contains a negative cost cycle then, there does not exist the shortest path. Otherwise, there exists a shortest s - v that is simple.



Cost of C < 0

Negative Weight Edges

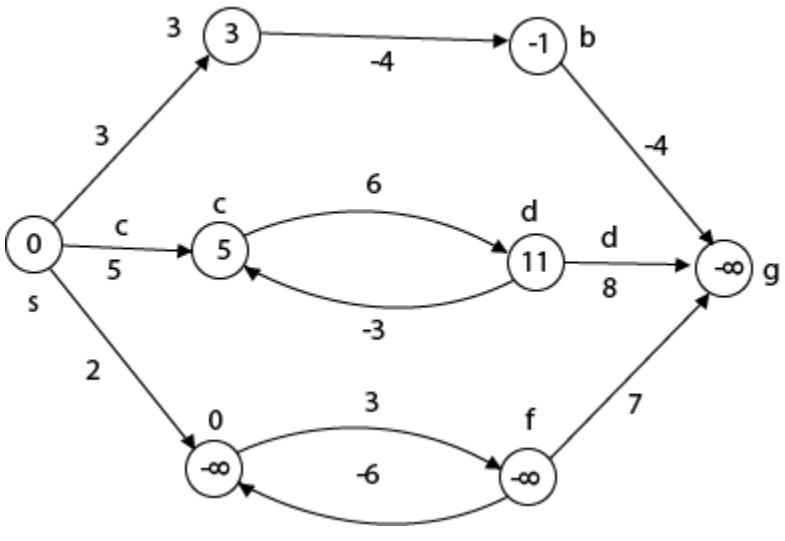
It is a weighted graph in which the total weight of an edge is negative. If a graph has a negative edge, then it produces a chain. After executing the chain if the output is negative then it will give - ∞ weight and condition get discarded. If weight is less than negative and - ∞ then we can't have the shortest path in it.

Briefly, if the output is -ve, then both condition get discarded.

1. - ∞
2. Not less than 0.

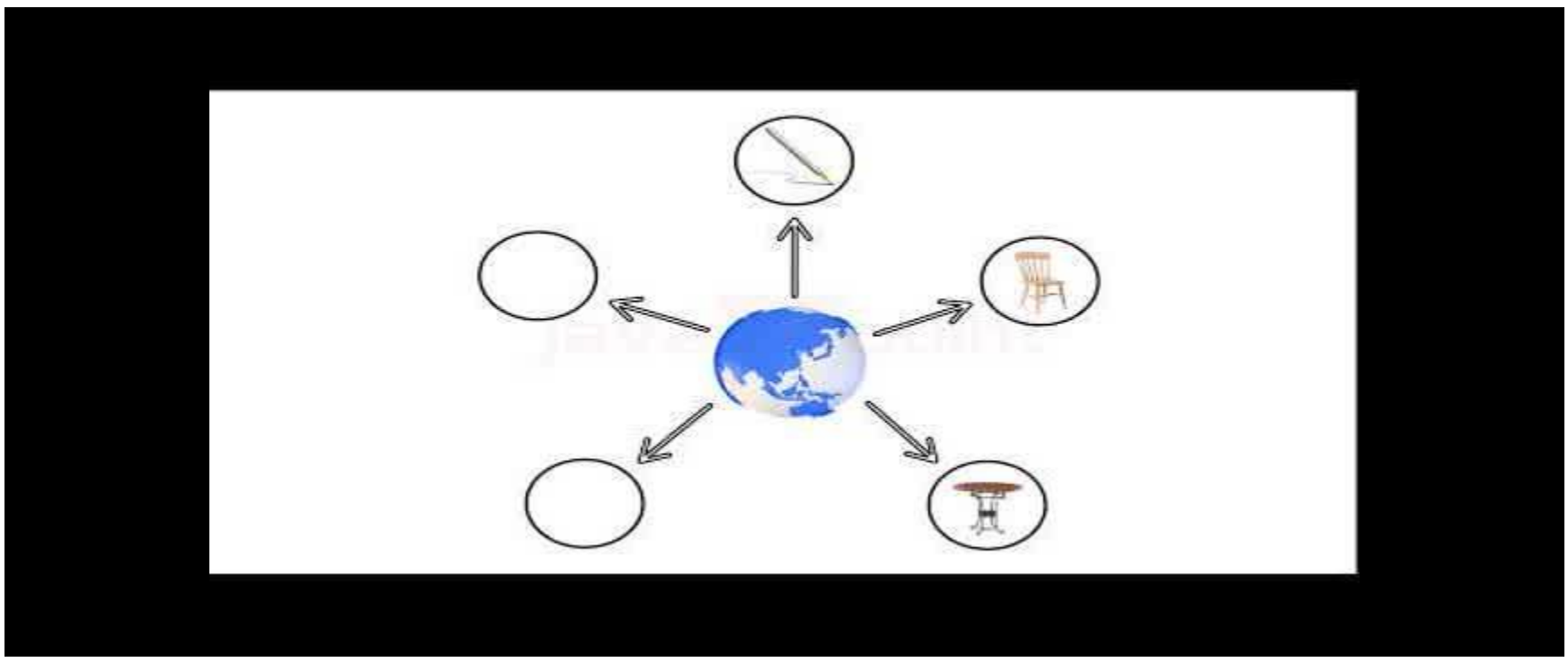
And we cannot have the shortest Path.

Example:



1. Beginning from s
2. Adj [s] = [a, c, e]
3. Weight from s to a is 3

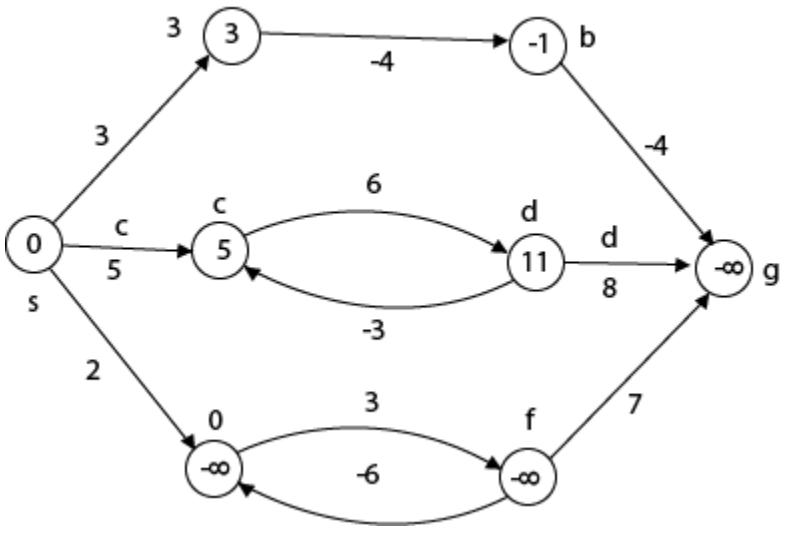
Suppose we want to calculate a path from s→c. So We have 2 paths /weight



1. s to c = 5, s→c→d→c=8
2. But s→c is minimum
3. So s→c = 5

Suppose we want to calculate a path from s→e. So we have two paths again

1. s→e = 2, s→e→f→e=-1
2. As -1 < 0 ∴ Condition gets discarded. If we execute **this** chain, we will get - ∞. So we can't get the shortest path ∴ e = ∞.



This figure illustrates the effects of negative weights and negative weight cycle on the shortest path weights.

Because there is only one path from "s to a" (the path <s, a>), δ (s, a) = w (s, a) = 3.

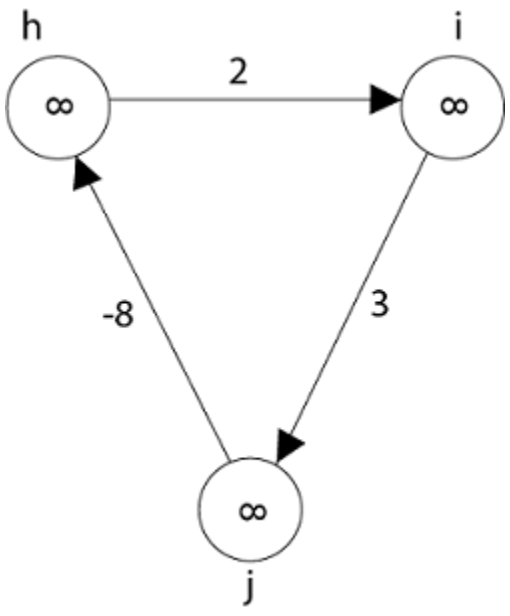
Furthermore, there is only one path from "s to b", so δ (s, b) = w (s, a) + w (a, b) = 3 + (-4) = - 1.

There are infinite many path from "s to c" : <s, c> : <s, c, d, c>, <s, c, d, c, d, c> and so on. Because the cycle <c, d, c> has weight δ (c, d) = w (c, d) + w (d, c) = 6 + (-3) = 3, which is greater than 0, the shortest path from s to c is <s, c> with weight δ (s, c) = 5.

Similarly, the shortest path from "s to d" is <s, c, d> with weight δ (s, d) = w (s, c) + w (s, d) = 11.

Analogously, there are infinite many paths from s to e: <s, e>, <s, e, f, e>, <s, e, f, e, f, e> and so on. Since the cycle <e, f, e> has weight δ (e, f) = w (e, f) + w (f, e) = 3 + (-6) = -3. So - 3 < 0, however there is no shortest path from s to e. B8y traversing the negative weight cycle <e, f, e>. This means path from s to e has arbitrary large negative weights and so δ (s, e) = - ∞.

Similarly δ (s, f) = - ∞ because g is reachable from f, we can also find a path with arbitrary large negative weight from s to g and δ (s, g) = - ∞



Vertices h, i, g also from negative weight cycle. They are also not reachable from the source node, so distance from the source is - ∞ to three of nodes (h, i, j).

Representing: Shortest Path

Given a graph G = (V, E), we maintain for each vertex v ∈ V a **predecessor** π [v] that is either another vertex or NIL. During the execution of shortest paths algorithms, however, the π values need not indicate shortest paths. As in breadth-first search, we shall be interested in the **predecessor subgraph** G_n= (V_n,E_n) induced by the value π. Here again, we define the vertex set V_π, to be the set of vertices of G with non - NIL predecessors, plus the source s:

$$V_\pi = \{v \in V: \pi [v] \neq NIL\} \cup \{s\} \}$$

The directed edge set E_Π is the set of edges induced by the Π values for vertices in V_Π:

$$E_\Pi = \{(\Pi[v], v) \in E: v \in V_\Pi - \{s\}\}$$

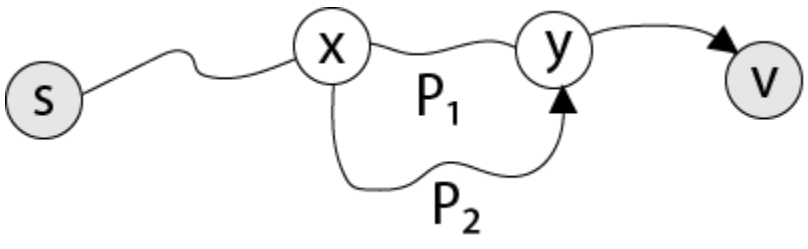
A **shortest - paths tree** rooted at s is a directed subgraph G = (V' E'), where V'∈ V andE'∈E, such that

1. V' is the set of vertices reachable from s in G
2. G' forms a rooted tree with root s, and
3. For all v ∈ V', the unique, simple path from s to v in G' is the shortest path from s to v in G.

Shortest paths are not naturally unique, and neither is shortest - paths trees.

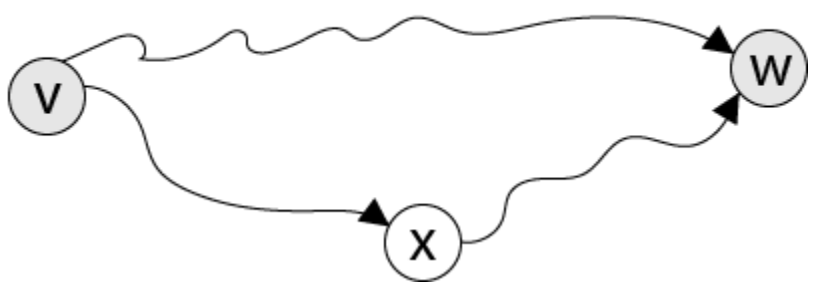
Properties of Shortest Path:

1. Optimal substructure property: All subpaths of shortest paths are shortest paths.



Let P_1 be x - y sub path of shortest s - v path. Let P_2 be any x - y path. Then cost of $P_1 \leq$ cost of P_2,otherwise P not shortest s - v path.

2. Triangle inequality: Let d (v, w) be the length of shortest path from v to w. Then, d (v, w) ≤ d (v, x) + d (x, w)



3. Upper-bound property: We always have d[v] ≥ δ(s, v) for all vertices v ∈ V, and once d[v] conclude the value δ(s, v), it never changes.

4. No-path property: If there is no path from s to v, then we regularly have d[v] = δ(s, v) = ∞.

5. Convergence property: If s->u->v is a shortest path in G for some u, v ∈ V, and if d[u] = δ(s, u) at any time prior to relaxing edge (u, v), then d[v] = δ(s, v) at all times thereafter.

Relaxation

The single - source shortest paths are based on a technique known as **relaxation**, a method that repeatedly decreases an upper bound on the actual shortest path weight of each vertex until the upper bound equivalent the shortest - path weight. For each vertex v ∈ V, we maintain an attribute d [v], which is an upper bound on the weight of the shortest path from source s to v. We call d [v] the **shortest path estimate**.

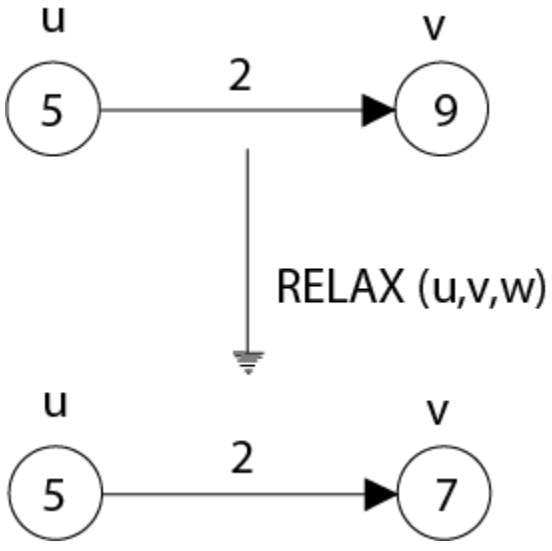
INITIALIZE - SINGLE - SOURCE (G, s)

```
1. for each vertex v ∈ V [G]
2. do d [v] ← ∞
3. π [v] ← NIL
4. d [s] ← 0
```

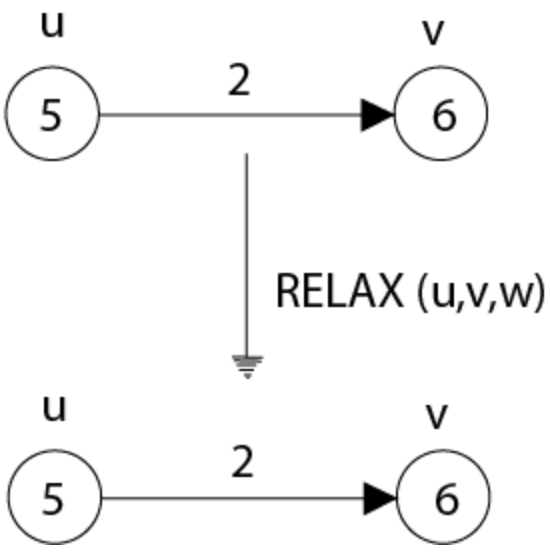
After initialization, π [v] = NIL for all v ∈ V, d [v] = 0 for v = s, and d [v] = ∞ for v ∈ V - {s}.

The development of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and if so, updating d [v] and π [v]. A relaxation step may decrease the value of the shortest - path estimate d [v] and updated v's predecessor field π [v].

Fig: Relaxing an edge (u, v) with weight w (u, v) = 2. The shortest-path estimate of each vertex appears within the vertex.



(a) Because v. d > u. d + w (u, v) prior to relaxation, the value of v. d decreases



(b) Here, v. d < u. d + w (u, v) before relaxing the edge, and so the relaxation step leaves v. d unchanged.

The subsequent code performs a relaxation step on edge (u, v)

RELAX (u, v, w)

```
1. If d [v] > d [u] + w (u, v)
2. then d [v] ← d [u] + w (u, v)
3. π [v] ← u
```

Dijkstra's Algorithm

It is a greedy algorithm that solves the single-source shortest path problem for a directed graph G = (V, E) with nonnegative edge weights, i.e., w (u, v) ≥ 0 for each edge (u, v) ∈ E.

Dijkstra's Algorithm maintains a set S of vertices whose final shortest - path weights from the source s have already been determined. That's for all vertices v ∈ S; we have d [v] = δ (s, v). The algorithm repeatedly selects the vertex u ∈ V - S with the minimum shortest - path estimate, insert u into S and relaxes all edges leaving u.

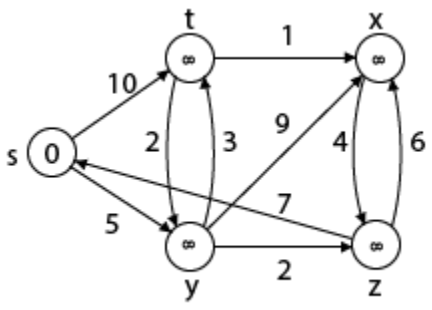
Because it always chooses the "lightest" or "closest" vertex in V - S to insert into set S, it is called as the **greedy strategy**.

Dijkstra's Algorithm (G, w, s)

```
1. INITIALIZE - SINGLE - SOURCE (G, s)
2. S←∅
3. Q←V [G]
4. while Q ≠ ∅
5. do u ← EXTRACT - MIN (Q)
6. S ← S ∪ {u}
7. for each vertex v ∈ Adj [u]
8. do RELAX (u, v, w)
```

Analysis: The running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of |E| and |V| using the Big - O notation. The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and operation Extract - Min (Q) is simply a linear search through all vertices in Q. In this case, the running time is O ($|V^2|+|E|=O(V^2)$).

Example:



Solution:

Step1: Q =[s, t, x, y, z]

We scanned vertices one by one and find out its adjacent. Calculate the distance of each adjacent to the source vertices.

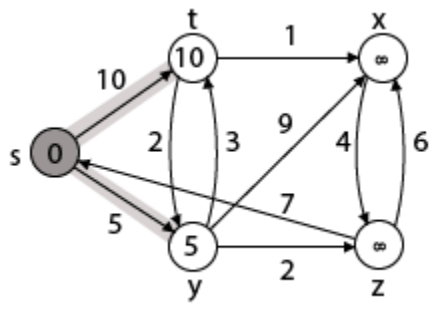
We make a stack, which contains those vertices which are selected after computation of shortest distance.

Firstly we take's' in stack M (which is a source)

1. M = [S] Q = [t, x, y, z]

Step 2: Now find the adjacent of s that are t and y.

1. Adj [s] → t, y [Here s is u and t and y are v]



Case - (i) s → t

d [v] > d [u] + w [u, v]

d [t] > d [s] + w [s, t]

∞ > 0 + 10 [false condition]

Then **d [t] ← 10**

π [t] ← 5

Adj [s] ← t, y

Case - (ii) s→ y

d [v] > d [u] + w [u, v]

d [y] > d [s] + w [s, y]

∞ > 0 + 5 [false condition]

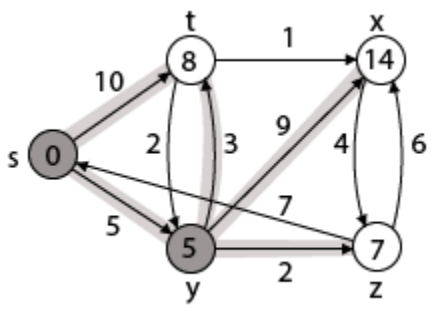
∞ > 5

Then **d [y] ← 5**

π [y] ← 5

By comparing case (i) and case (ii)
Adj [s] → t = 10, y = 5
y is shortest

y is assigned in 5 = [s, y]



Step 3: Now find the adjacent of y that is t, x, z.

1. Adj [y] → t, x, z [Here y is u and t, x, z are v]

Case - (i) y →t

d [v] > d [u] + w [u, v]

d [t] > d [y] + w [y, t]

10 > 5 + 3

10 > 8

Then d [t] ← 8

π [t] ← y

Case - (ii) y → x

d [v] > d [u] + w [u, v]

d [x] > d [y] + w [y, x]

∞ > 5 + 9

∞ > 14

Then d [x] ← 14

π [x] ← 14

Case - (iii) y → z

d [v] > d [u] + w [u, v]

d [z] > d [y] + w [y, z]

∞ > 5 + 2

∞ > 7

Then d [z] ← 7

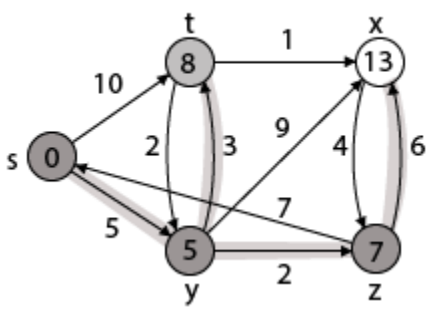
π [z] ← y

By comparing case (i), case (ii) and case (iii)

Adj [y] → x = 14, t = 8, z =7

z is shortest

z is assigned in 7 = [s, z]



Step - 4 Now we will find adj [z] that are s, x

1. Adj [z] → [x, s] [Here z is u and s and x are v]

Case - (i) z → x

d [v] > d [u] + w [u, v]

d [x] > d [z] + w [z, x]

14 > 7 + 6

14 > 13

Then d [x] ← 13

π [x] ← z

Case - (ii) z → s

d [v] > d [u] + w [u, v]

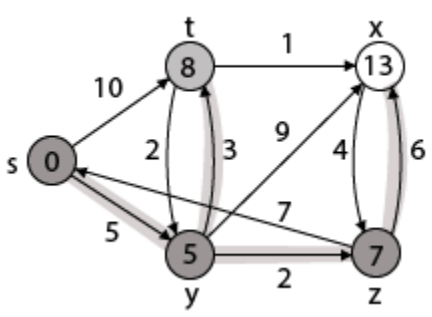
d [s] > d [z] + w [z, s]

0 > 7 + 7

0 > 14

∴ This condition does not satisfy so it will be discarded.

Now we have x = 13.



Step 5: Now we will find Adj [t]

Adj [t] → [x, y] [Here t is u and x and y are v]

Case - (i) t → x

d [v] > d [u] + w [u, v]

d [x] > d [t] + w [t, x]

13 > 8 + 1

13 > 9

Then d [x] ← 9

π [x] ← t

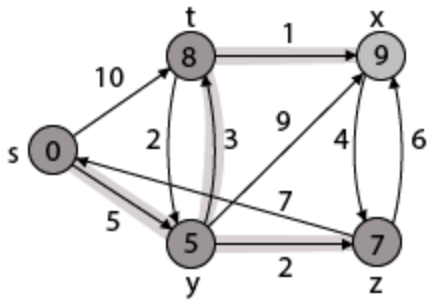
Case - (ii) t → y

d [v] > d [u] + w [u, v]

d [y] > d [t] + w [t, y]

5 > 10

∴ This condition does not satisfy so it will be discarded.

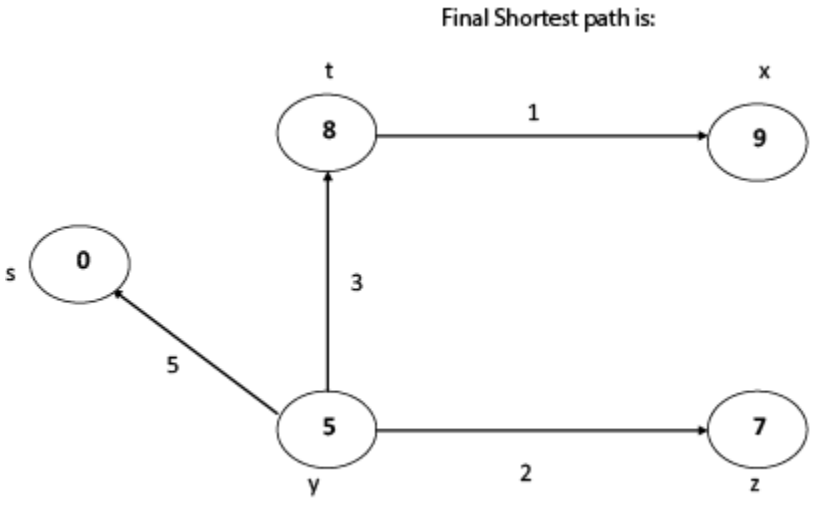


Thus we get all shortest path vertex as

Weight	from	s	to	y	is	5
Weight	from	s	to	z	is	7
Weight	from	s	to	t	is	8

Weight from s to x is 9

These are the shortest distance from the source's' in the given graph.



Disadvantage of Dijkstra's Algorithm:

1. It does a blind search, so wastes a lot of time while processing.
2. It can't handle negative edges.
3. It leads to the acyclic graph and most often cannot obtain the right shortest path.
4. We need to keep track of vertices that have been visited.

Bellman-Ford Algorithm

Solves single shortest path problem in which edge weight may be negative but no negative cycle exists.

This algorithm works correctly when some of the edges of the directed graph G may have negative weight. When there are no cycles of negative weight, then we can find out the shortest path between source and destination.

It is slower than Dijkstra's Algorithm but more versatile, as it capable of handling some of the negative weight edges.

This algorithm detects the negative cycle in a graph and reports their existence.

Based on the "**Principle of Relaxation**" in which more accurate values gradually recovered an approximation to the proper distance by until eventually reaching the optimum solution.

Given a weighted directed graph G = (V, E) with source s and weight function w: E → R, the Bellman-Ford algorithm returns a Boolean value indicating whether or not there is a negative weight cycle that is attainable from the source. If there is such a cycle, the algorithm produces the shortest paths and their weights. The algorithm returns TRUE if and only if a graph contains no negative - weight cycles that are reachable from the source.

Recurrence Relation

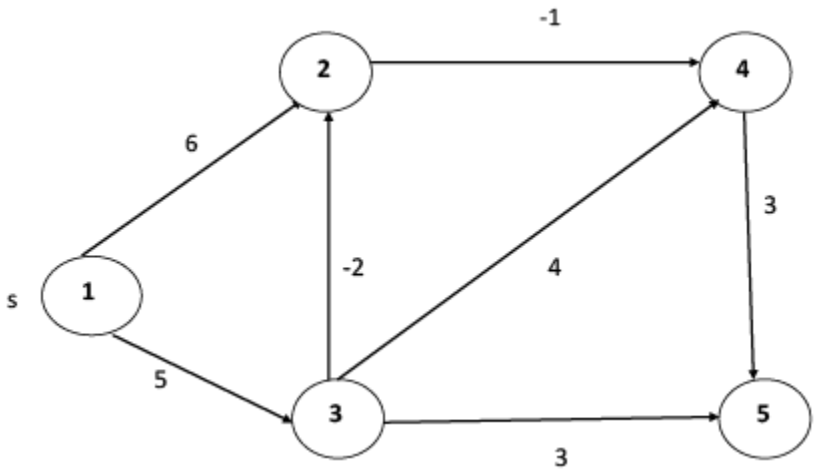
$dist^{k}$ [u] = [min[$dist^{k-1}$ [u],min[$dist^{k-1}$ [i]+cost [i,u]]] as i except u.

k → k is the source vertex
u → u is the destination vertex
i → no of edges to be scanned concerning a vertex.

BELLMAN -FORD (G, w, s)

```
1. INITIALIZE - SINGLE - SOURCE (G, s)
2. for i ← 1 to |V[G]| - 1
3. do for each edge (u, v) ∈ E [G]
4. do RELAX (u, v, w)
5. for each edge (u, v) ∈ E [G]
6. do if d [v] > d [u] + w (u, v)
7. then return FALSE.
8. return TRUE.
```

Example: Here first we list all the edges and their weights.



Solution:

$dist^k$ [u] = [min[$dist^{k-1}$ [u],min[$dist^{k-1}$ [i]+cost [i,u]]] as i ≠ u.

Vertices

No of Edges Traversed

	1	2	3	4	5
1	0	6	5	∞	∞
2	0	3	5	5	8
3	0	3	5	2	8
4	0	3	5	2	5

$dist^2$ [2]=min[$dist^1$ [2],min[$dist^1$ [1]+cost[1,2],$dist^1$ [3]+cost[3,2],$dist^1$ [4]+cost[4,2],$dist^1$ [5]+cost[5,2]]]

Min = [6, 0 + 6, 5 + (-2), ∞ + ∞ , ∞ +∞] = 3

$dist^2$ [3]=min[$dist^1$ [3],min[$dist^1$ [1]+cost[1,3],$dist^1$ [2]+cost[2,3],$dist^1$ [4]+cost[4,3],$dist^1$ [5]+cost[5,3]]]

Min = [5, 0 +∞, 6 +∞, ∞ + ∞ , ∞ + ∞] = 5

$dist^2$ [4]=min[$dist^1$ [4],min[$dist^1$ [1]+cost[1,4],$dist^1$ [2]+cost[2,4],$dist^1$ [3]+cost[3,4],$dist^1$ [5]+cost[5,4]]]

Min = [∞, 0 +∞, 6 + (-1), 5 + 4, ∞ +∞] = 5

$dist^2$ [5]=min[$dist^1$ [5],min[$dist^1$ [1]+cost[1,5],$dist^1$ [2]+cost[2,5],$dist^1$ [3]+cost[3,5],$dist^1$ [4]+cost[4,5]]]

Min = [∞, 0 + ∞,6 + ∞,5 + 3, ∞ + 3] = 8

$dist^3$ [2]=min[$dist^2$ [2],min[$dist^2$ [1]+cost[1,2],$dist^2$ [3]+cost[3,2],$dist^2$ [4]+cost[4,2],$dist^2$ [5]+cost[5,2]]]

Min = [3, 0 + 6, 5 + (-2), 5 + ∞ , 8 + ∞] = 3

$dist^3$ [3]=min[$dist^2$ [3],min[$dist^2$ [1]+cost[1,3],$dist^2$ [2]+cost[2,3],$dist^2$ [4]+cost[4,3],$dist^2$ [5]+cost[5,3]]]

Min = [5, 0 + ∞, 3 + ∞, 5 + ∞,8 + ∞] = 5

$dist^3$ [4]=min[$dist^2$ [4],min[$dist^2$ [1]+cost[1,4],$dist^2$ [2]+cost[2,4],$dist^2$ [3]+cost[3,4],$dist^2$ [5]+cost[5,4]]]

Min = [5, 0 + ∞, 3 + (-1), 5 + 4, 8 + ∞] = 2

$dist^3$ [5]=min[$dist^2$ [5],min[$dist^2$ [1]+cost[1,5],$dist^2$ [2]+cost[2,5],$dist^2$ [3]+cost[3,5],$dist^2$ [4]+cost[4,5]]]

Min = [8, 0 + ∞, 3 + ∞, 5 + 3, 5 + 3] = 8

$dist^4$ [2]=min[$dist^3$ [2],min[$dist^3$ [1]+cost[1,2],$dist^3$ [3]+cost[3,2],$dist^3$ [4]+cost[4,2],$dist^3$ [5]+cost[5,2]]]

Min = [3, 0 + 6, 5 + (-2), 2 + ∞, 8 + ∞] =3

$dist^4$ [3]=min[$dist^3$ [3],min[$dist^3$ [1]+cost[1,3],$dist^3$ [2]+cost[2,3],$dist^3$ [4]+cost[4,3],$dist^3$ [5]+cost[5,3]]]

Min = 5, 0 + ∞, 3 + ∞, 2 + ∞, 8 + ∞] =5

$dist^4$ [4]=min[$dist^3$ [4],min[$dist^3$ [1]+cost[1,4],$dist^3$ [2]+cost[2,4],$dist^3$ [3]+cost[3,4],$dist^3$ [5]+cost[5,4]]]

Min = [2, 0 + ∞, 3 + (-1), 5 + 4, 8 + ∞] = 2

$dist^4$ [5]=min[$dist^3$ [5],min[$dist^3$ [1]+cost[1,5],$dist^3$ [2]+cost[2,5],$dist^3$ [3]+cost[3,5],$dist^3$ [5]+cost[4,5]]]

Min = [8, 0 +∞, 3 + ∞, 8, 5] = 5

Single Source Shortest Path in a directed Acyclic Graphs

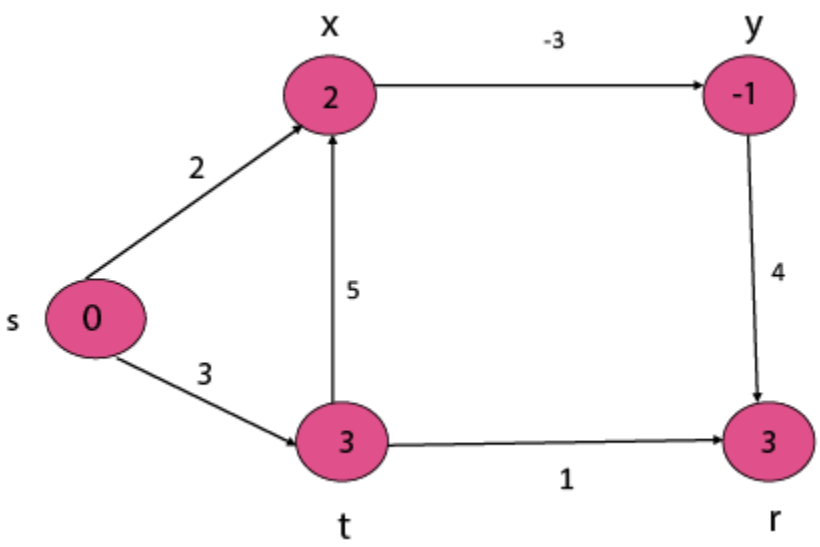
By relaxing the edges of a weighted DAG (Directed Acyclic Graph) G = (V, E) according to a topological sort of its vertices, we can figure out shortest paths from a single source in Ø(V+E) time. Shortest paths are always well described in a dag, since even if there are negative-weight edges, no negative-weight cycles can exist.

DAG - SHORTEST - PATHS (G, w, s)

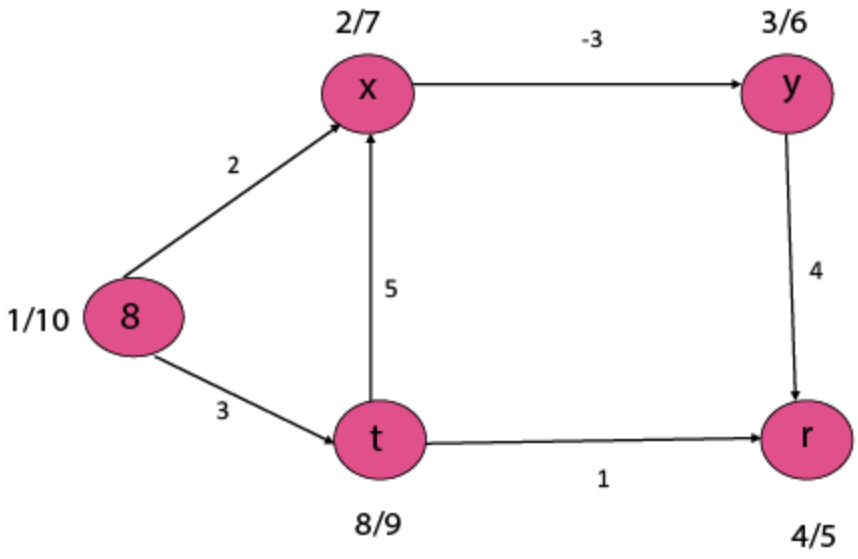
```
1. Topologically sort the vertices of G.
2. INITIALIZE - SINGLE- SOURCE (G, s)
3. for each vertex u taken in topologically sorted order
4. do for each vertex v ∈ Adj [u]
5. do RELAX (u, v, w)
```

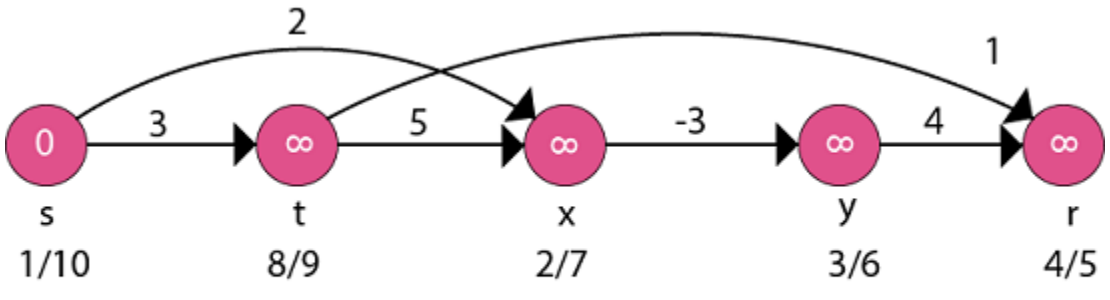
The running time of this data is determined by line 1 and by the for loop of lines 3 - 5. The topological sort can be implemented in Ø (V + E) time. In the for loop of lines 3 - 5, as in Dijkstra's algorithm, there is one repetition per vertex. For each vertex, the edges that leave the vertex are each examined exactly once. Unlike Dijkstra's algorithm, we use only O (1) time per edge. The running time is thus Ø (V + E), which is linear in the size of an adjacency list depiction of the graph.

Example:



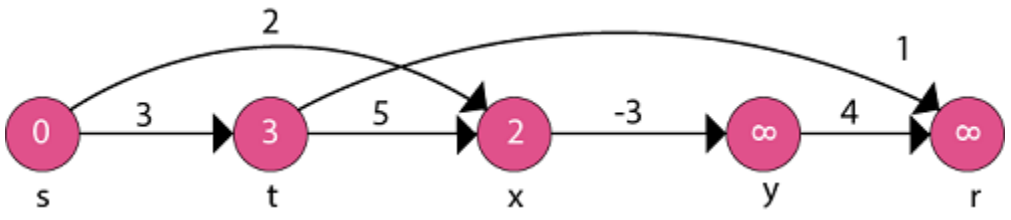
Step1: To topologically sort vertices apply **DFS (Depth First Search)** and then arrange vertices in linear order by decreasing order of finish time.



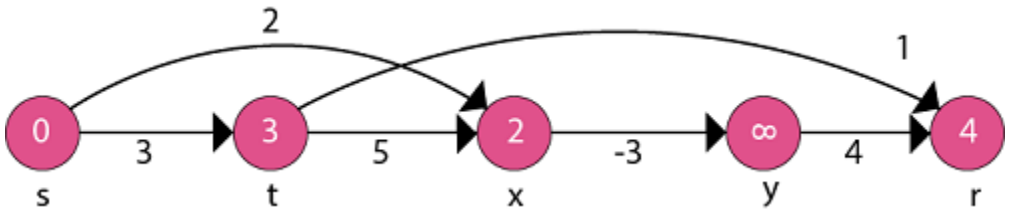


Initialize Single Source

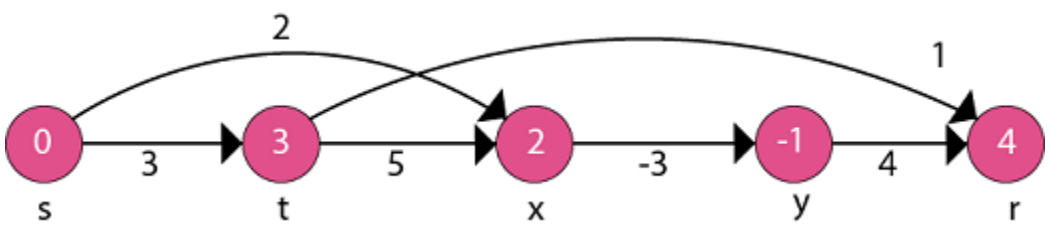
Now, take each vertex in topologically sorted order and relax each edge.



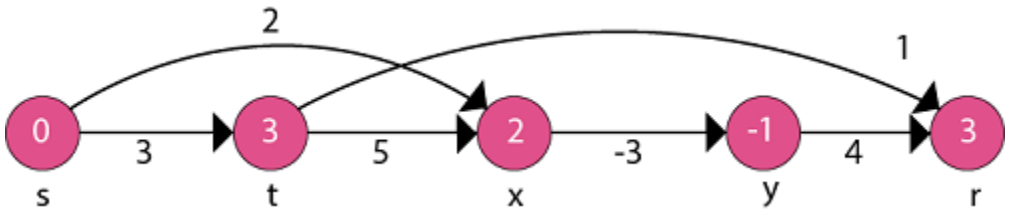
1. adj [s] →t, x
2. 0 + 3 < ∞
3. d [t] ← 3
4. 0 + 2 < ∞
5. d [x] ← 2



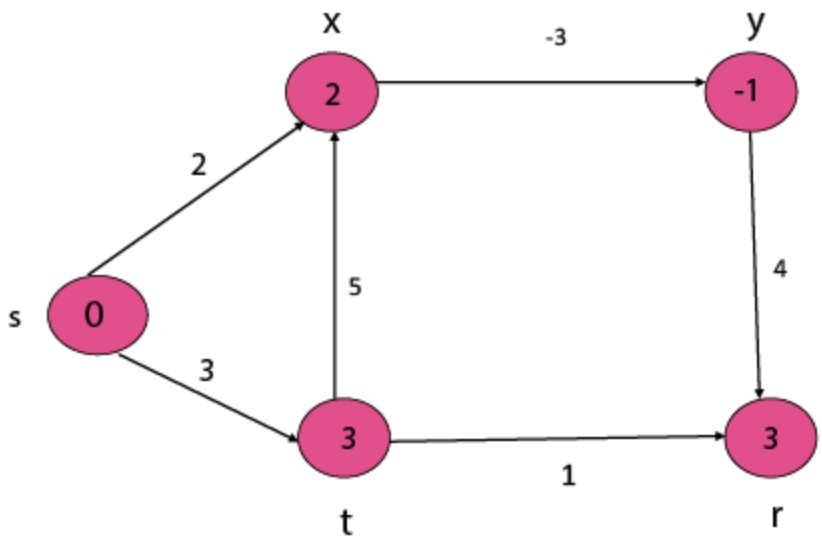
1. adj [t] → r, x
2. 3 + 1 < ∞
3. d [r] ← 4
4. 3 + 5 ≤ 2



1. adj [x] → y
2. 2 - 3 < ∞
3. d [y] ← -1



1. adj [y] → r
2. -1 + 4 < 4
3. 3 <4
4. d [r] ← 3



Thus the Shortest Path is:

1. s to x is 2
2. s to y is -1
3. s to t is 3
4. s to r is 3

Flow Networks and Flows

Flow Network is a directed graph that is used for modeling material Flow. There are two different vertices; one is a **source** which produces material at some steady rate, and another one is sink which consumes the content at the same constant speed. The flow of the material at any mark in the system is the rate at which the element moves.

Some real-life problems like the flow of liquids through pipes, the current through wires and delivery of goods can be modeled using flow networks.

Definition: A Flow Network is a directed graph G = (V, E) such that

1. For each edge (u, v) ∈ E, we associate a nonnegative weight capacity c (u, v) ≥ 0.If (u, v) ∉ E, we assume that c (u, v) = 0.
2. There are two distinguishing points, the source s, and the sink t;
3. For every vertex v ∈ V, there is a path from s to t containing v.

Let G = (V, E) be a flow network. Let s be the source of the network, and let t be the sink. A flow in G is a real-valued function f: V x V→R such that the following properties hold:

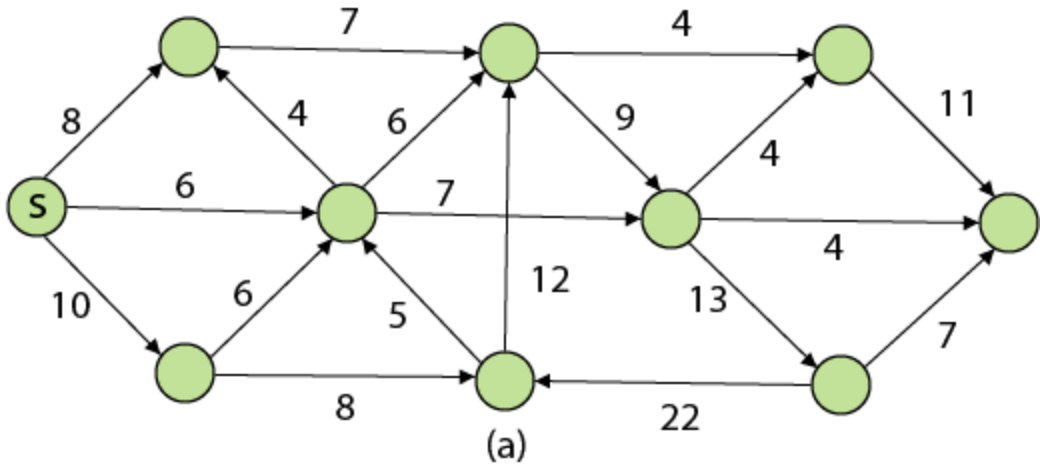
- **Capacity Constraint:** For all u, v ∈ V, we need f (u, v) ≤ c (u, v).
- **Skew Symmetry:** For all u, v ∈ V, we need f (u, v) = - f (u, v).
- **Flow Conservation:** For all u ∈ V-{s, t}, we need

$$\sum_{v \in V} f(u,v) = \sum_{u \in V} f(u,v) = 0$$

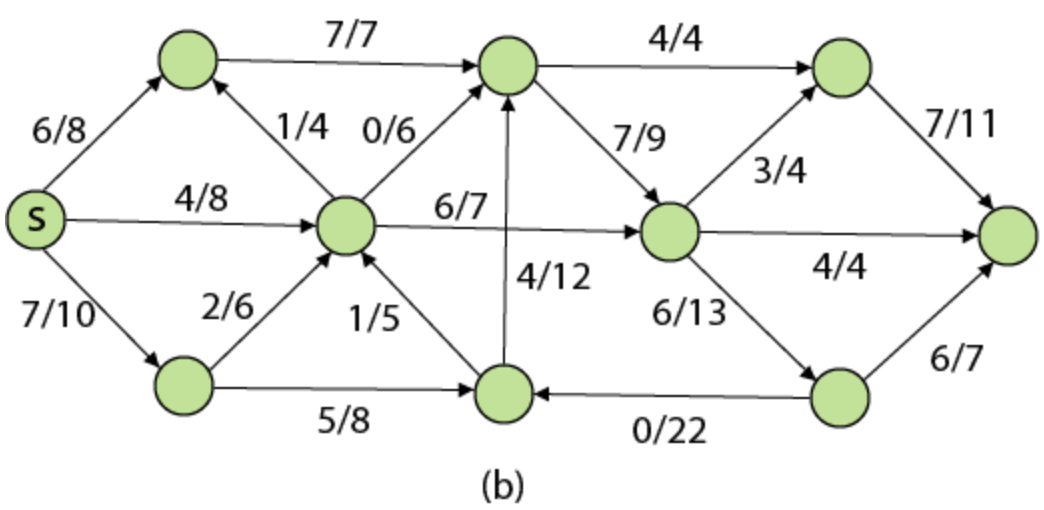
The quantity f (u, v), which can be positive or negative, is known as the net flow from vertex u to vertex v. In the **maximum-flow problem**, we are given a flow network G with source s and sink t, and we wish to find a flow of maximum value from s to t.

The three properties can be described as follows:

1. **Capacity Constraint** makes sure that the flow through each edge is not greater than the capacity.
2. **Skew Symmetry** means that the flow from u to v is the negative of the flow from v to u.
3. The flow-conservation property says that the total net flow out of a vertex other than the source or sink is 0. In other words, the amount of flow into a v is the same as the amount of flow out of v for every vertex v ∈ V - {s, t}



(a)



(b)

The value of the flow is the net flow from the source,

$$|f| = \sum_{v \in V} f(s, v)$$

The **positive net flow entering** a vertex v is described by

$$\sum_{\{u \in V : f(u,v) > 0\}} f(u, v)$$

The **positive net flow** leaving a vertex is described symmetrically. One interpretation of the Flow-Conservation Property is that the positive net flow entering a vertex other than the source or sink must equal the positive net flow leaving the vertex.

A flow f is said to be **integer-valued** if f (u, v) is an integer for all (u, v) ∈ E. Clearly, the value of the flow is an integer is an integer-valued flow.

Network Flow Problems

The most obvious flow network problem is the following:

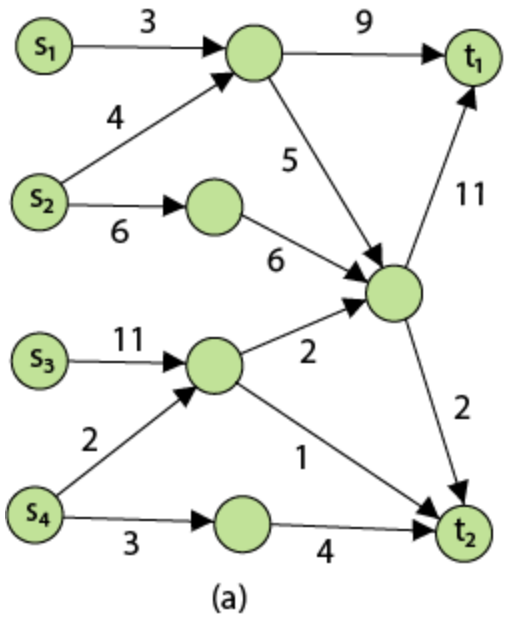
Problem1: Given a flow network G = (V, E), the maximum flow problem is to find a flow with maximum value.

Problem 2: The multiple source and sink maximum flow problem is similar to the maximum flow problem, except there is a set $\{s_1, s_2, s_3 \ldots\ldots s_n\}$ of sources and a set $\{t_1, t_2, t_3 \ldots\ldots\ldots t_n\}$ of sinks.

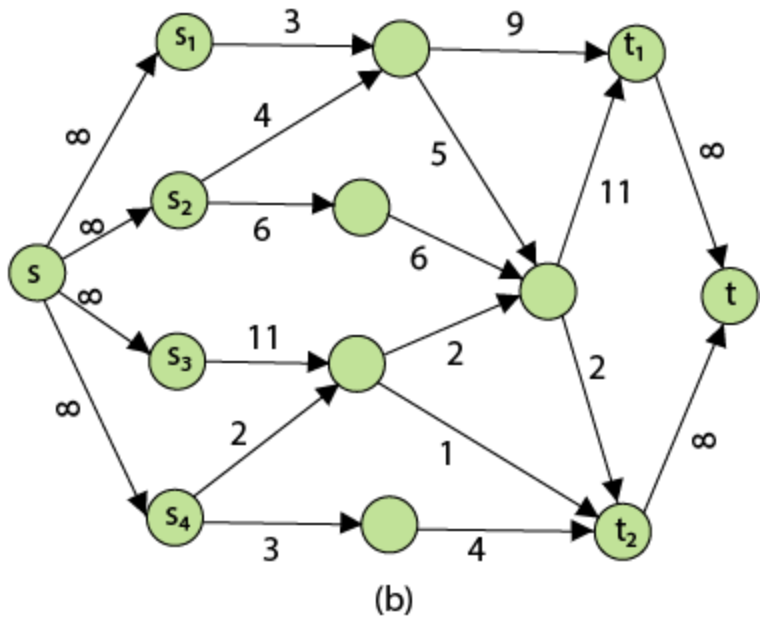
Fortunately, this problem is no solid than regular maximum flow. Given multiple sources and sink flow network G, we define a new flow network G' by adding

- A super source s,
- A super sink t,
- For each s_i, add edge (s, s_i) with capacity ∞, and
- For each t_i,add edge (t_i, t) with capacity ∞

Figure shows a multiple sources and sinks flow network and an equivalent single source and sink flow network



(a)



(b)

Residual Networks: The Residual Network consists of an edge that can admit more net flow. Suppose we have a flow network G = (V, E) with source s and sink t. Let f be a flow in G, and examine a pair of vertices u, v ∈ V. The sum of additional net flow we can push from u to v before exceeding the capacity c (u, v) is the residual capacity of (u, v) given by

$$c_f(u, v) = c\,(u, v) - f\,(u, v).$$

When the net flow f (u, v) is negative, the residual capacity c_f (u,v) is greater than the capacity c (u, v).

For Example: if c (u, v) = 16 and f (u, v) =16 and f (u, v) = -4, then the residual capacity c_f (u,v) is 20.

Given a flow network G = (V, E) and a flow f, the residual network of G induced by f is G_f = (V, E_f), where

$$E_f = \{(u, v) \in V \times V : C_f(u, v) \geq 0\}$$

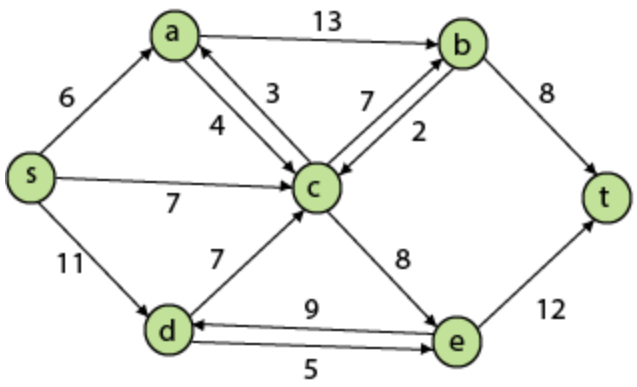
That is, each edge of the residual network, or residual edge, can admit a strictly positive net flow.

Augmenting Path: Given a flow network G = (V, E) and a flow f, an **augmenting path** p is a simple path from s to t in the residual networkG_f. By the solution of the residual network, each edge (u, v) on an augmenting path admits some additional positive net flow from u to v without violating the capacity constraint on the edge.

Let G = (V, E) be a flow network with flow f. The **residual capacity** of an augmenting path p is

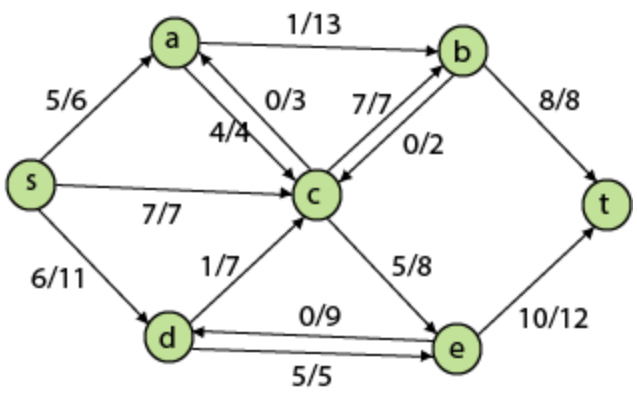
$$C_f(p) = \min \{C_f(u, v) : (u, v) \text{is on } p\}$$

The residual capacity is the maximal amount of flow that can be pushed through the augmenting path. If there is an augmenting path, then each edge on it has a positive capacity. We will use this fact to compute a maximum flow in a flow network.



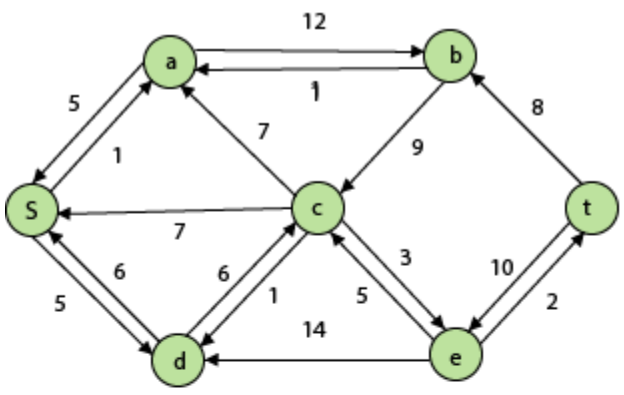
(a)

A flow network G= (V,E)

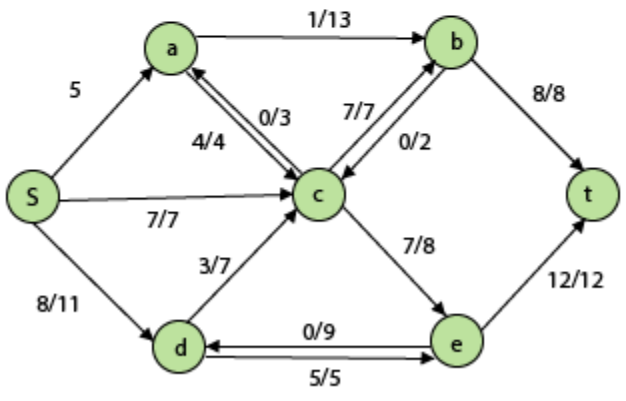


(b)

A flow in f in G



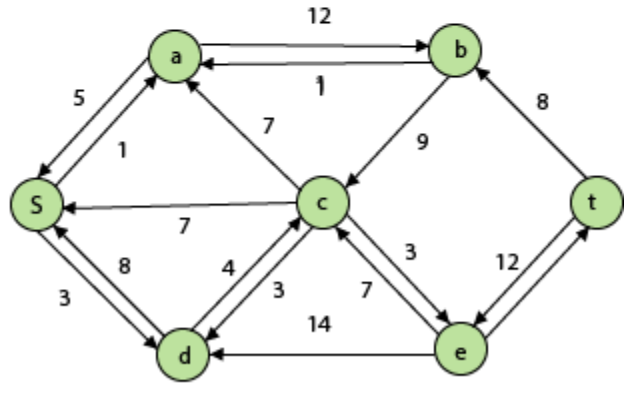
(c)
The residual network G_f



(e)

A new flow $f' = f + f_p$

(d)
The grey edges form an augmenting path with capacity z.



(f)

The residual network G_f

Ford-Fulkerson Algorithm

Initially, the flow of value is 0. Find some augmenting Path p and increase flow f on each edge of p by residual Capacity c_f (p). When no augmenting path exists, flow f is a maximum flow.

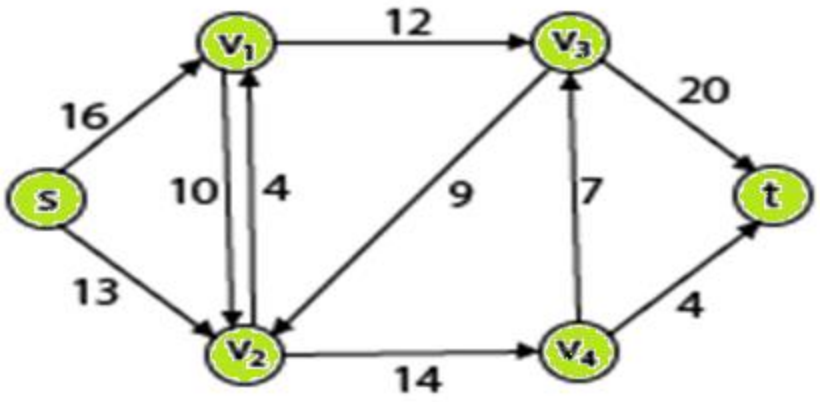
FORD-FULKERSON METHOD (G, s, t)

```
1. Initialize flow f to 0
2. while there exists an augmenting path p
3. do argument flow f along p
4. Return f
```

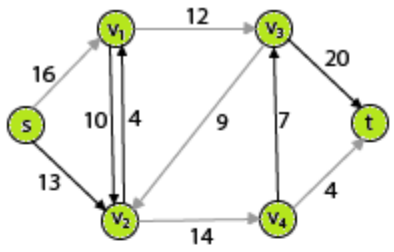
FORD-FULKERSON (G, s, t)

```
1. for each edge (u, v) ∈ E [G]
2. do f [u, v] ← 0
3. f [u, v] ← 0
4. while there exists a path p from s to t in the residual network Gf.
5. do cf (p)←min?{ Cf (u,v):(u,v)is on p}
6. for each edge (u, v) in p
7. do f [u, v] ← f [u, v] + cf  (p)
8. f [u, v] ←-f[u,v]
```

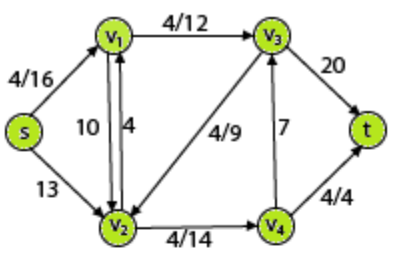
Example: Each Directed Edge is labeled with capacity. Use the Ford-Fulkerson algorithm to find the maximum flow.

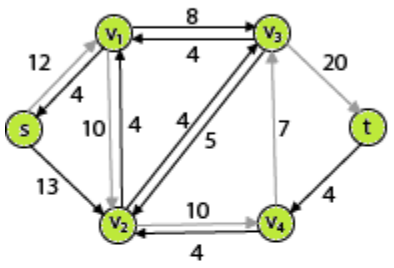


Solution: The left side of each part shows the residual network G_f with a shaded augmenting path p,and the right side of each part shows the net flow f.

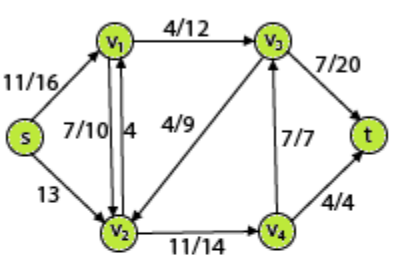


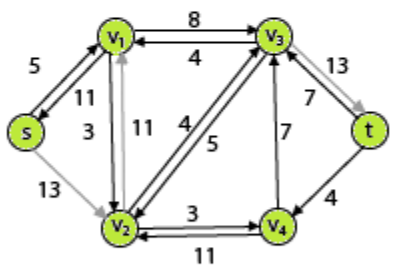
(a)



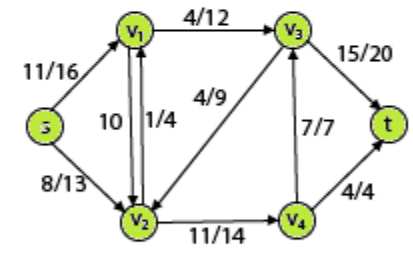


(b)

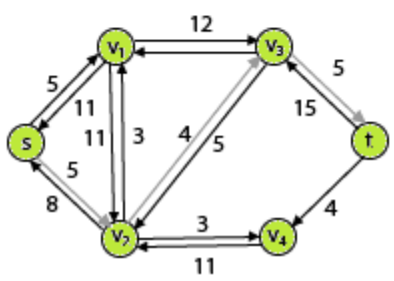




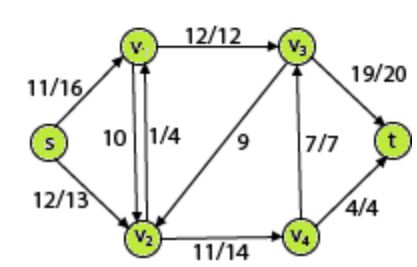
(c)

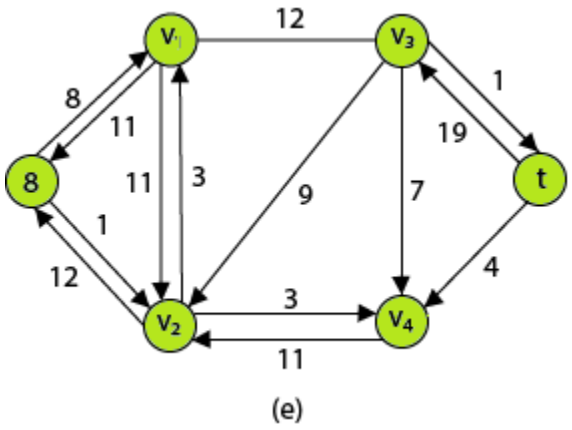


(In this , 8 is break into 7 and 1 and 7 is canclled by $v_1v_2$flow)



(d)



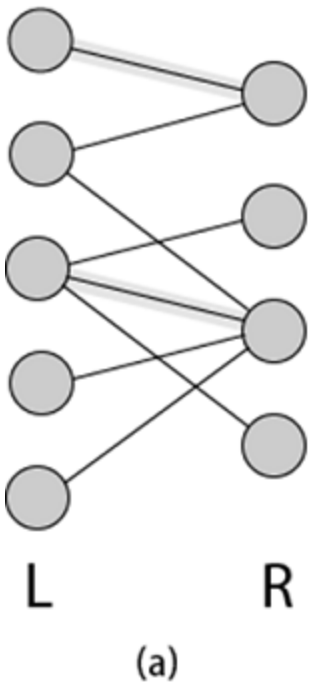


(e)

Now , it has no augmenting paths .So, the maximum flow shown in (d) is 23 is a maximum flow .

Maximum Bipartite Matching

A Bipartite Graph is a graph whose vertices can be divided into two independent sets L and R such that every edge (u, v) either connect a vertex from L to R or a vertex from R to L. In other words, for every edge (u, v) either $u \in L$ and $v \in L$. We can also say that no edge exists that connect vertices of the same set.



(a)

Matching is a Bipartite Graph is a set of edges chosen in such a way that no two edges share an endpoint. Given an undirected Graph G = (V, E), a Matching is a subset of edge M ⊆ E such that for all vertices v ∈ V, at most one edge of M is incident on v.

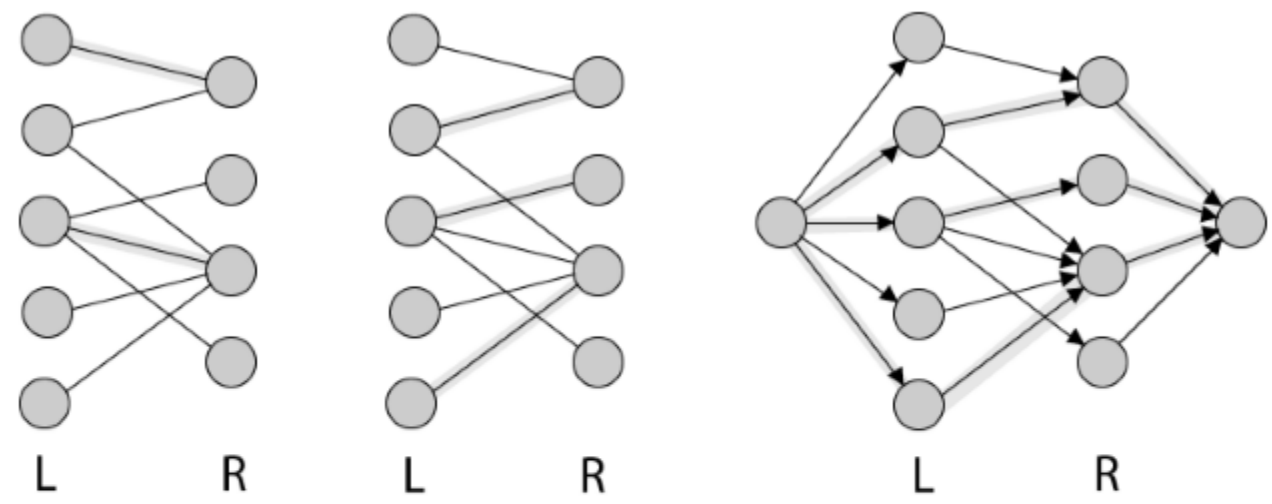
A Maximum matching is a matching of maximum cardinality, that is, a matching M such that for any matching M', we have|M|>|M' |.

Finding a maximum bipartite matching

We can use the Ford-Fulkerson method to find a maximum matching in an undirected bipartite graph G= (V, E) in time polynomial in |V| and |E|. The trick is to construct a flow network G= (V',E') for the bipartite graph G as follows. We let the source s and sink t be new vertices not in V, and we let V'=V ∪{s,t}.If the vertex partition of G is V = L∪R, the directed edges of G' are the edges of E, directed from L to R, along with |V| new directed edges:

$$E' = \{(s,u): u \in L\} \cup \{(u,v): (u,v) \in E\} \cup \{(v,t): v \in R\}$$

Fig: A Bipartite Graph G = (V, E) with vertex partition V = L ∪ R.



(a) Matching with Cardinality 2.

(b) Matching with Cardinality 3.

(c) The corresponding flow network G′ with a maximum flow shown. Each edge has unit Capacity. The shaded edges from L to R correspond to those in the maximum matching from (b).

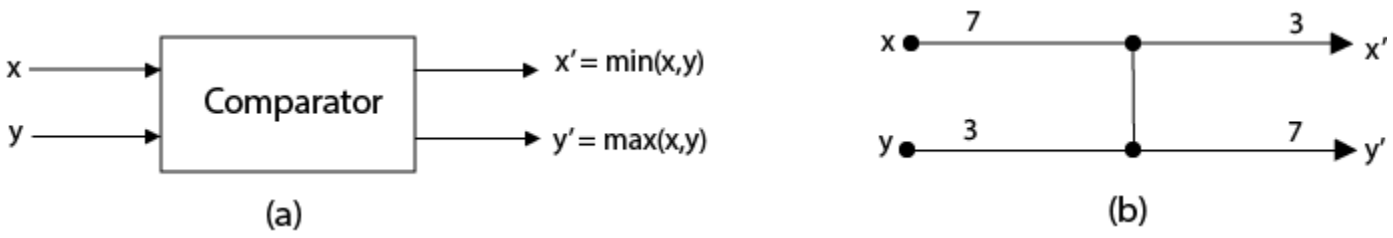
Comparison Networks

A comparison network is made of wires and comparators. A **comparator** is a device with two inputs, x and y, and two outputs, x' and y,' where

x'= min (x, y)
y' = max (x, y)

In Comparison Networks input appear on the left and outputs on the right, with the smallest input value appearing on the top output and the largest input value appearing on the bottom output. Each comparator operates in O (1) time. In other words, we consider that the time between the appearance of the input values x and y and the production of the output values x' and y' is a constant.

A wire transmits a value from place to place. A comparison network contains n input wires $a_1,a_2,.....a_n$ through which the benefits to be sorted enter the network, and n output wires $b_1,b_2,.....b_n$ which produce the results computed by the network.

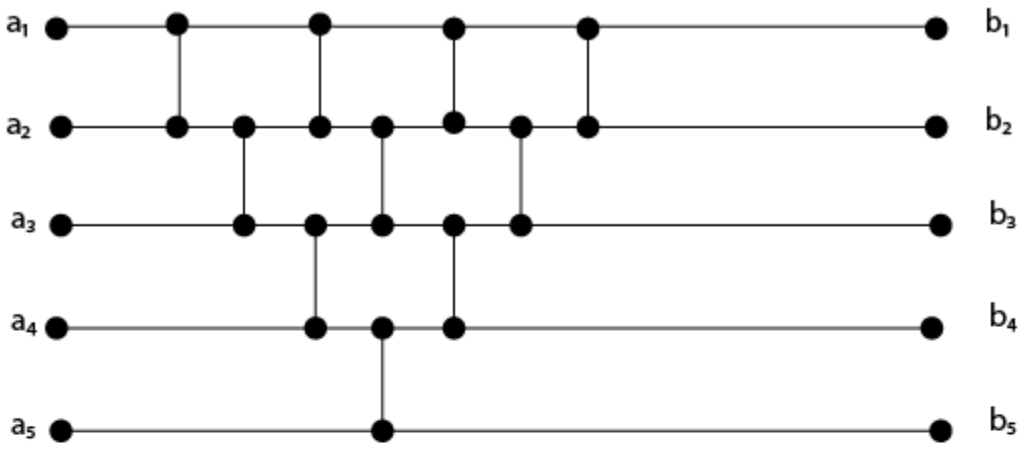


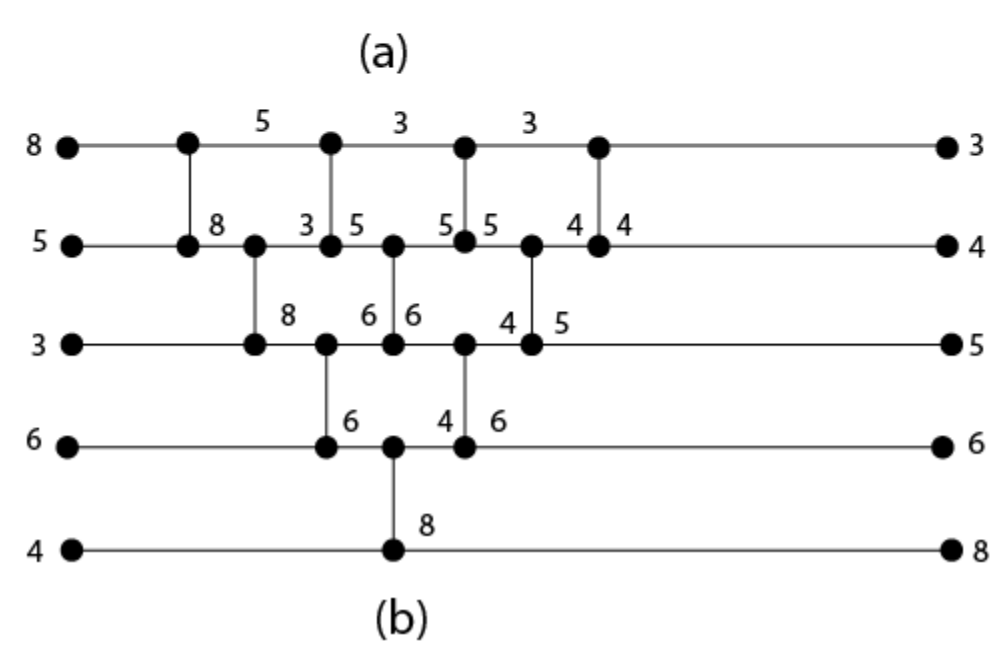
Comparison Network is a set of comparators interconnected by wires. Running time of comparator can define regarding **depth**.

Depth of a Wire: An input wire of a comparison network has depth 0. Now, if a comparator has two input wires with depths d_x and d_y' then its output wires have depth max (d_x,d_y) + 1.

A sorting network is a comparison network for which the output sequence is monotonically increasing (that is $b_1 \leq b_2 \leq ....b_n$) for every input sequence.

Fig: A Sorting network based on Insertion Sort





Bitonic Sorting Network

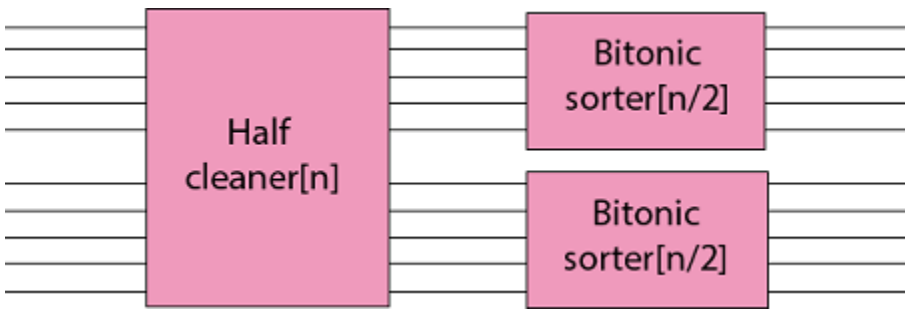
A sequence that monotonically increases and then monotonically decreases, or else monotonically decreases and then monotonically increases is called a bitonic sequence. For example: the sequence (2, 5, 6, 9, 3, 1) and (8, 7, 5, 2, 4, 6) are both bitonic. The bitonic sorter is a comparison network that sorts bitonic sequence of 0's and 1's.

Half-Cleaner: A bitonic sorter is containing several stages, each of which is called a half-cleaner. Each half-cleaner is a comparison network of depth 1 in which input line i is compared with line $1+\frac{n}{2}$ for i = 1, 2..... $\frac{n}{2}$.

When a bitonic sequence of 0's and 1's is practiced as input to a half-cleaner, the half-cleaner produces an output sequence in which smaller values are in the top half, larger values are in the bottom half, and both halves are bitonic, and at least one of the halves is clean.

Bitonic Sorter: By recursively connecting half-cleaners, we can build a bitonic sorter, which is a network that sorts bitonic sequences. The first stage of BITONIC-SORTER [n] consists of HALF-CLEANER [n], which produces two bitonic sequences of half the size such that every element in the top half is at least as small as each element in the bottom half. Thus, we can complete the sort by utilizing two copies of BITONIC-SORTER [n/2] to sort the two halves recursively.

Fig: The depth D (n) of BITONIC-SORTER [n] is given by recurrence whose solution is D (n) = log n.



Merging Network

Merging Network is the network that can join two sorted input sequences into one sorted output sequence. We adapt BITONIC-SORTER [n] to create the merging network MERGER [n].

The merging network is based on the following assumption:

Given two sorted sequences, if we reverse the order of the second sequence and then connect the two sequences, the resulting sequence is bitonic.

For Example: Given two sorted zero-one sequences X = 00000111 and Y =00001111, we reverse Y to get Y^R = 11110000. Concatenating X and Y^R yield 0000011111110000, which is bitonic.

The sorting network SORTER [n] need the merging network to implement a parallel version of merge sort. The first stage of SORTER [n] consists of n/2 copies of MERGER [2] that work in parallel to merge pairs of a 1-element sequence to produce a sorted sequence of length 2. The second stage subsists of n/4 copies of MERGER [4] that merge pairs of these 2-element sorted sequences to generate sorted sequences of length 4. In general, for k = 1, 2..... log n, stage k consists of $n/2^k$ copies of MERGER [2^k] that merge pairs of the 2^k-1 element sorted sequence to produce a sorted sequence of length2^k. At the last stage, one sorted sequence consisting of all the input values is produced. This sorting network can be shown by induction to sort zero-one sequences, and therefore by the zero-one principle, it can sort arbitrary values.

The recurrence given the depth of SORTER [n]

$$D(n) = \begin{cases} 0 & \text{if } n = 1 \\ D\left(\frac{n}{2}\right) + \log n & \text{if } n = 2^k \text{ and } k \geq 1 \end{cases}$$

Whose solution is D (n) = θ ($\log^2 n$). Thus, we can sort n numbers in parallel in ₒ ($\log^2 n$) time.

Complexity Classes

Definition of NP class Problem: - The set of all decision-based problems came into the division of NP Problems who can't be solved or produced an output within polynomial time but verified in the **polynomial time**. NP class contains P class as a subset. NP problems being hard to solve.

> ***Note: - The term "NP" does not mean "not polynomial." Originally, the term meant "non-deterministic polynomial. It means according to the one input number of output will be produced.***

Definition of P class Problem: - The set of decision-based problems come into the division of P Problems who can be solved or produced an output within polynomial time. P problems being easy to solve

Definition of Polynomial time: - If we produce an output according to the given input within a specific amount of time such as within a minute, hours. This is known as Polynomial time.

Definition of Non-Polynomial time: - If we produce an output according to the given input but there are no time constraints is known as Non-Polynomial time. But yes output will produce but time is not fixed yet.

Definition of Decision Based Problem: - A problem is called a decision problem if its output is a simple "yes" or "no" (or you may need this of this as true/false, 0/1, accept/reject.) We will phrase many optimization problems as decision problems. For example, Greedy method, D.P., given a graph G= (V, E) if there exists any Hamiltonian cycle.

Definition of NP-hard class: - Here you to satisfy the following points to come into the division of NP-hard

1. If we can solve this problem in polynomial time, then we can solve all NP problems in polynomial time
2. If you convert the issue into one form to another form within the polynomial time

Definition of NP-complete class: - A problem is in NP-complete, if

1. It is in NP
2. It is NP-hard

Pictorial representation of all NP classes which includes NP, NP-hard, and NP-complete

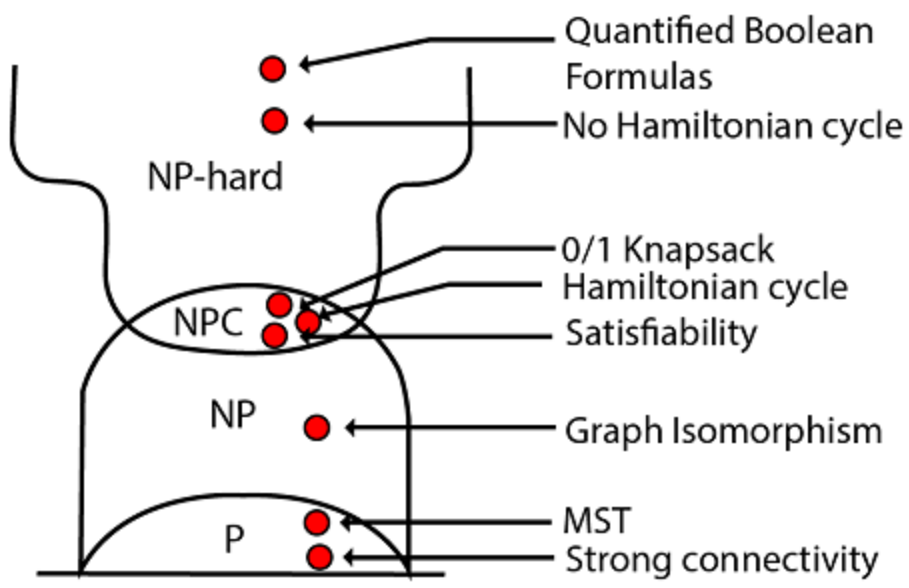


Fig: Complexity Classes

Polynomial Time Verification

Before talking about the class of NP-complete problems, it is essential to introduce the notion of a verification algorithm.

Many problems are hard to solve, but they have the property that it easy to authenticate the solution if one is provided.

Hamiltonian cycle problem:-

Consider the Hamiltonian cycle problem. Given an undirected graph G, does G have a cycle that visits each vertex exactly once? There is no known polynomial time algorithm for this dispute.

> ***Note: - It means you can't build a Hamiltonian cycle in a graph with a polynomial time even if there is no specific path is given for the Hamiltonian cycle with the particular vertex, yet you can't verify the Hamiltonian cycle within the polynomial time***

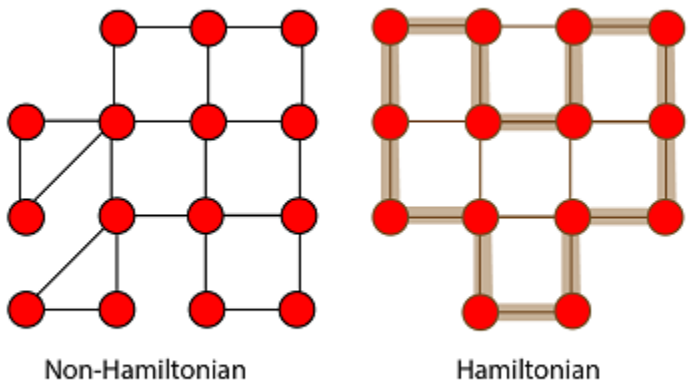


Fig: Hamiltonian Cycle

Let us understand that a graph did have a Hamiltonian cycle. It would be easy for someone to convince of this. They would similarly say: "the period is hv3, v7, v1....v13i.

We could then inspect the graph and check that this is indeed a legal cycle and that it visits all of the vertices of the graph exactly once. Thus, even though we know of no efficient way to solve the Hamiltonian cycle problem, there is a beneficial way to verify that a given cycle is indeed a Hamiltonian cycle.

> ***Note:-For the verification in the Polynomial-time of an undirected Hamiltonian cycle graph G. There must be exact/specific/definite path must be given of Hamiltonian cycle then you can verify in the polynomial time.***

Definition of Certificate: - A piece of information which contains in the given path of a vertex is known as certificate

Relation of P and NP classes

1. P contains in NP
2. P=NP

1. Observe that P contains in NP. In other words, if we can solve a problem in polynomial time, we can indeed verify the solution in polynomial time. More formally, we do not need to see a certificate (there is no need to specify the vertex/intermediate of the specific path) to solve the problem; we can explain it in polynomial time anyway.
2. However, it is not known whether P = NP. It seems you can verify and produce an output of the set of decision-based problems in NP classes in a polynomial time which is impossible because according to the definition of NP classes you can verify the solution within the polynomial time. So this relation can never be held.

Reductions:

The class NP-complete (NPC) problems consist of a set of decision problems (a subset of class NP) that no one knows how to solve efficiently. But if there were a polynomial solution for even a single NP-complete problem, then every problem in NPC will be solvable in polynomial time. For this, we need the concept of reductions.

Suppose there are two problems, A and B. You know that it is impossible to solve problem A in polynomial time. You want to prove that B cannot be explained in polynomial time. We want to show that ($A \notin P$) => ($B \notin P$)

Consider an example to illustrate reduction: The following problem is well-known to be NPC:

3-color: Given a graph G, can each of its vertices be labeled with one of 3 different colors such that two adjacent vertices do not have the same label (color).

Coloring arises in various partitioning issues where there is a constraint that two objects cannot be assigned to the same set of partitions. The phrase "coloring" comes from the original application which was in map drawing. Two countries that contribute a common border should be colored with different colors.

It is well known that planar graphs can be colored (maps) with four colors. There exists a polynomial time algorithm for this. But deciding whether this can be done with 3 colors is hard, and there is no polynomial time algorithm for it.

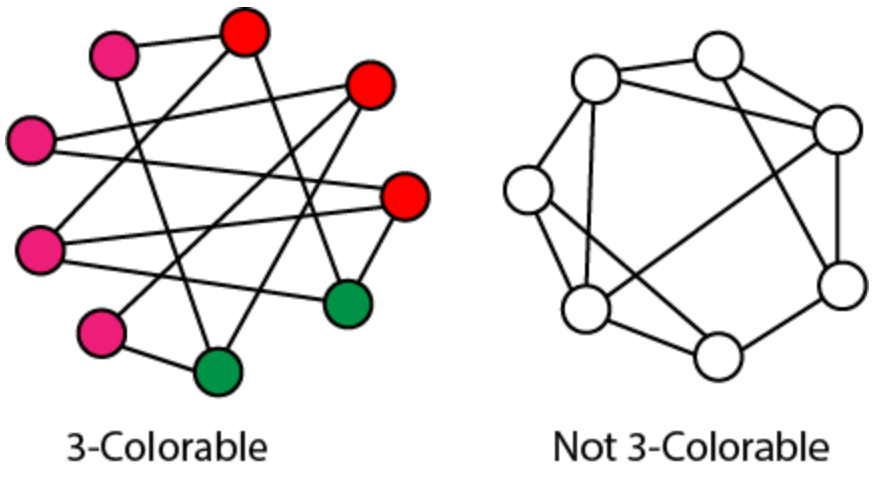


Fig: Example of 3-colorable and non-3-colorable graphs.

Polynomial Time Reduction:

We say that Decision Problem L_1 is Polynomial time Reducible to decision Problem L_2 ($L_1 \leq_p L_2$) if there is a polynomial time computation function f such that of all x, $x \epsilon L_1$ if and only if $x \epsilon L_2$.

NP-Completeness

A decision problem L is NP-Hard if

L' ≤p L for all L' ∈ NP.

Definition: L is NP-complete if

1. L ∈ NP and
2. L' ≤ p L for some known NP-complete problem L.' Given this formal definition, the complexity classes are:

P: is the set of decision problems that are solvable in polynomial time.

NP: is the set of decision problems that can be verified in polynomial time.

NP-Hard: L is NP-hard if for all L' ∈ NP, L' ≤p L. Thus if we can solve L in polynomial time, we can solve all NP problems in polynomial time.

NP-Complete L is NP-complete if

1. L ∈ NP and
2. L is NP-hard

If any NP-complete problem is solvable in polynomial time, then every NP-Complete problem is also solvable in polynomial time. Conversely, if we can prove that any NP-Complete problem cannot be solved in polynomial time, every NP-Complete problem cannot be solvable in polynomial time.

Reductions

Concept: - If the solution of NPC problem does not exist then the conversion from one NPC problem to another NPC problem within the polynomial time. For this, you need the concept of reduction. If a solution of the one NPC problem exists within the polynomial time, then the rest of the problem can also give the solution in polynomial time (but it's hard to believe). For this, you need the concept of reduction.

Example: - Suppose there are two problems, **A** and **B**. You know that it is impossible to solve problem **A** in polynomial time. You want to prove that B cannot be solved in polynomial time. So you can convert the problem **A** into problem **B** in polynomial time.

Example of NP-Complete problem

NP problem: - Suppose a DECISION-BASED problem is provided in which a set of inputs/high inputs you can get high output.

Criteria to come either in NP-hard or NP-complete.

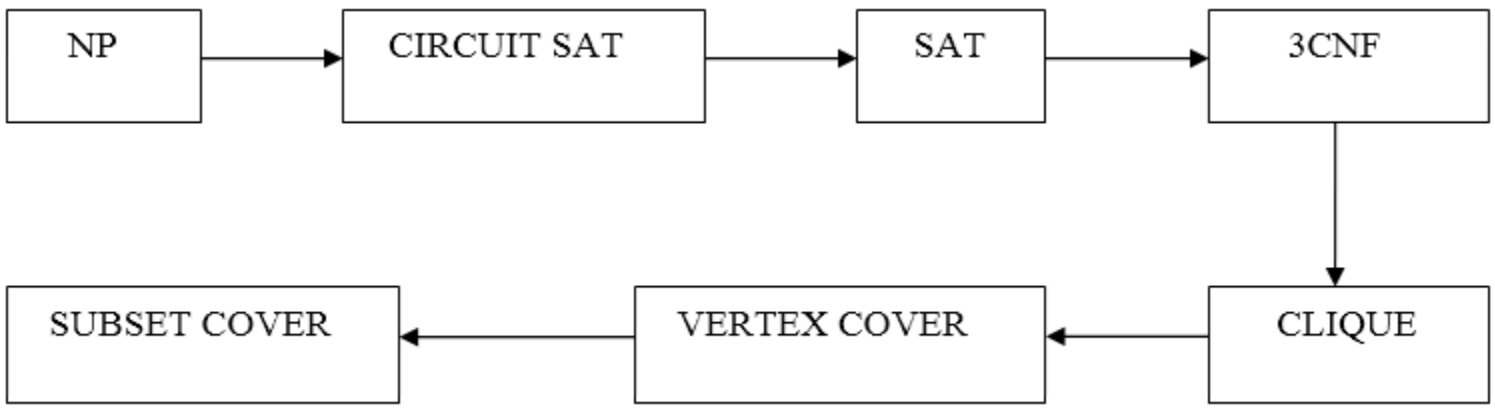
1. The point to be noted here, the output is already given, and you can verify the output/solution within the polynomial time but can't produce an output/solution in polynomial time.

2. Here we need the concept of reduction because when you can't produce an output of the problem according to the given input then in case you have to use an emphasis on the concept of reduction in which you can convert one problem into another problem.

Note1:- If you satisfy both points then your problem comes into the category of NP-complete class
Note2:- If you satisfy the only 2nd points then your problem comes into the category of NP-hard class

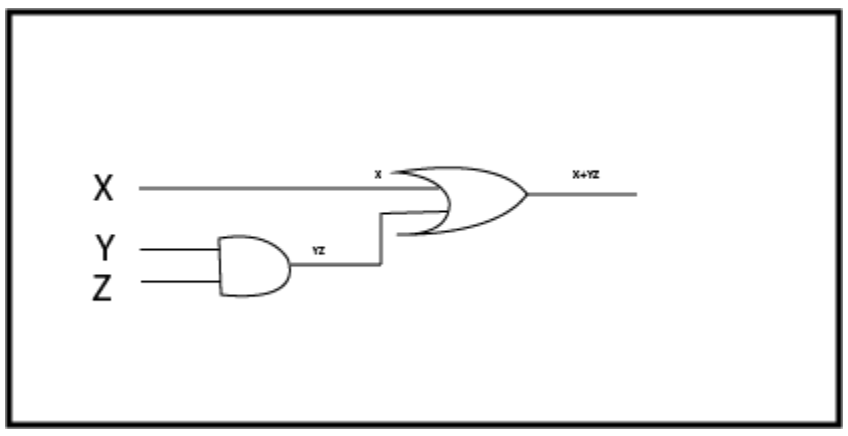
So according to the given decision-based NP problem, you can decide in the form of yes or no. If, yes then you have to do verify and convert into another problem via reduction concept. If you are being performed, both then decision-based NP problems are in NP compete.

Here we will emphasize NPC.



CIRCUIT SAT

According to given decision-based NP problem, you can design the CIRCUIT and verify a given mentioned output also within the P time. The CIRCUIT is provided below:-



Note:- You can design a circuit and verified the mentioned output within Polynomial time but remember you can never predict the number of gates which produces the high output against the set of inputs/high inputs within a polynomial time. So you verified the production and conversion had been done within polynomial time. So you are in NPC.

SAT (Satisfiability):-

A Boolean function is said to be SAT if the output for the given value of the input is true/high/1

F=X+YZ (Created a Boolean function by CIRCUIT SAT)

These points you have to be performed for NPC

1. CONCEPTS OF SAT
2. CIRCUIT SAT≤ρ SAT
3. SAT≤ρ CIRCUIT SAT
4. SAT ϵ NPC

1. **CONCEPT:** - A Boolean function is said to be SAT if the output for the given value of the input is true/high/1.
2. **CIRCUIT SAT≤ρ SAT:** - In this conversion, you have to convert CIRCUIT SAT into SAT within the polynomial time as we did it
3. **SAT≤ρ CIRCUIT SAT:** - For the sake of verification of an output you have to convert SAT into CIRCUIT SAT within the polynomial time, and through the CIRCUIT SAT you can get the verification of an output successfully
4. **SAT ϵ NPC:** - As you know very well, you can get the SAT through CIRCUIT SAT that comes from NP.

Proof of NPC: - Reduction has been successfully made within the polynomial time from CIRCUIT SAT TO SAT. Output has also been verified within the polynomial time as you did in the above conversation.

So concluded that SAT ϵ NPC.

3CNF SAT

Concept: - In 3CNF SAT, you have at least 3 clauses, and in clauses, you will have almost 3 literals or constants

Such as (X+Y+Z) (X+Y+Z) (X+Y+Z)
You can define as (XvYvZ) ^ (XvYvZ) ^ (XvYvZ)
V=OR
^ =AND operator

These all the following points need to be considered in 3CNF SAT.

To prove: -

1. Concept of 3CNF SAT
2. SAT≤ρ 3CNF SAT
3. 3CNF≤ρ SAT
4. 3CNF ϵ NPC

1. **CONCEPT:** - In 3CNF SAT, you have at least 3 clauses, and in clauses, you will have almost 3 literals or constants.
2. **SAT ≤ρ 3CNF SAT:**- In which firstly you need to convert a Boolean function created in SAT into 3CNF either in POS or SOP form within the polynomial time

   F=X+YZ
   = (X+Y) (X+Z)
   = (X+Y+ZZ') (X+YY'+Z)
   = (X+Y+Z) (X+Y+Z') (X+Y+Z) (X+Y'+Z)
   = (X+Y+Z) (X+Y+Z') (X+Y'+Z)
3. **3CNF ≤p SAT:** - From the Boolean Function having three literals we can reduce the whole function into a shorter one.

   F= (X+Y+Z) (X+Y+Z') (X+Y'+Z)
   = (X+Y+Z) (X+Y+Z') (X+Y+Z) (X+Y'+Z)
   = (X+Y+ZZ') (X+YY'+Z)
   = (X+Y) (X+Z)
   = X+YZ
4. **3CNF ϵ NPC:** - As you know very well, you can get the 3CNF through SAT and SAT through CIRCUIT SAT that comes from NP.

Proof of NPC:-

1. It shows that you can easily convert a Boolean function of SAT into 3CNF SAT and satisfied the concept of 3CNF SAT also within polynomial time through Reduction concept.
2. If you want to verify the output in 3CNF SAT then perform the Reduction and convert into SAT and CIRCUIT also to check the output

If you can achieve these two points that means 3CNF SAT also in NPC

Clique

To Prove: - Clique is an NPC or not?

For this you have to satisfy the following below-mentioned points: -

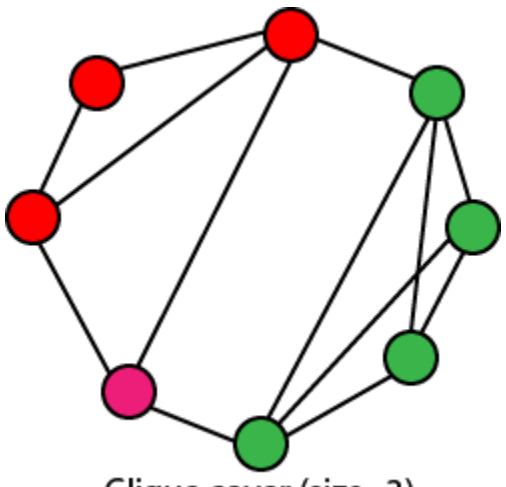
1. Clique
2. 3CNF ≤ρ Clique
3. Clique ≤ρ 3CNF≤SAT
4. Clique ϵ NP

1) Clique

Definition: - In Clique, every vertex is directly connected to another vertex, and the number of vertices in the Clique represents the Size of Clique.

CLIQUE COVER: - Given a graph G and an integer k, can we find k subsets of verticesV_1, V_2...V_K, such that UiVi = V, and that each Vi is a clique of G.

The following figure shows a graph that has a clique cover of size 3.



Clique cover (size=3)

2)3CNF ≤ρ Clique

Proof:-For the successful conversion from 3CNF to Clique, you have to follow the two steps:-

Draw the clause in the form of vertices, and each vertex represents the literals of the clauses.

1. They do not complement each other
2. They don't belong to the same clause
   In the conversion, the size of the Clique and size of 3CNF must be the same, and you successfully converted 3CNF into Clique within the polynomial time

Clique ≤ρ 3CNF

Proof: - As you know that a function of K clause, there must exist a Clique of size k. It means that P variables which are from the different clauses can assign the same value (say it is 1). By using these values of all the variables of the CLIQUES, you can make the value of each clause in the function is equal to 1

Example: - You have a Boolean function in 3CNF:-

(X+Y+Z) (X+Y+Z') (X+Y'+Z)

After Reduction/Conversion from 3CNF to CLIQUE, you will get P variables such as: - x +y=1, x +z=1 and x=1

Put the value of P variables in equation (i)

(1+1+0)(1+0+0)(1+0+1)

(1)(1)(1)=1 output verified

4) Clique ϵ NP:-

Proof: - As you know very well, you can get the Clique through 3CNF and to convert the decision-based NP problem into 3CNF you have to first convert into SAT and SAT comes from NP.

So, concluded that CLIQUE belongs to NP.

Proof of NPC:-

1. Reduction achieved within the polynomial time from 3CNF to Clique
2. And verified the output after Reduction from Clique To 3CNF above
   So, concluded that, if both Reduction and verification can be done within the polynomial time that means **Clique also in NPC**.

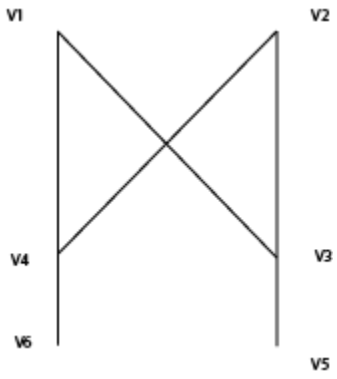
Vertex Cover

1. Vertex Cover Definition
2. Vertex Cover ≤ρ Clique
3. Clique ≤ρ Vertex Cover
4. Vertex Cover ϵ NP

1) Vertex Cover:

Definition: - It represents a set of vertex or node in a graph G (V, E), which gives the connectivity of a complete graph

According to the graph G of vertex cover which you have created, **the size of Vertex Cover =2**



2) Vertex Cover ≤ρ Clique

In a graph G of Vertex Cover, you have N vertices which contain a Vertex Cover K. There must exist of Clique Size of size N-K in its complement.

According to the graph G, you have Number of vertices=6
Size of Clique=N-K=4

You can also create the Clique by complimenting the graph G of Vertex Cover means in simpler form connect the vertices in Vertex Cover graph G through edges where edges don?t exist and remove all the existed edges

You will get the graph G with Clique Size=4

3) Clique ≤ρ Vertex Cover

Here through the Reduction process, you can get the Vertex Cover form Clique by just complimenting the Clique graph G within the polynomial time.

4) Vertex Cover ϵ NP

As you know very well, you can get the Vertex Cover through Clique and to convert the decision-based NP problem into Clique firstly you have to convert into 3CNF and 3CNF into SAT and SAT into CIRCUIT SAT that comes from NP.

Proof of NPC:-

1. Reduction from Clique to Vertex Cover has been made within the polynomial time. In the simpler form, you can convert into Vertex Cover from Clique within the polynomial time
2. And verification has also been done when you convert Vertex Cover to Clique and Clique to 3CNF and satisfy/verified the output within a polynomial time also, so it concluded that Reduction and Verification had been done in the polynomial time that means **Vertex Cover also comes in NPC**

Subset Cover

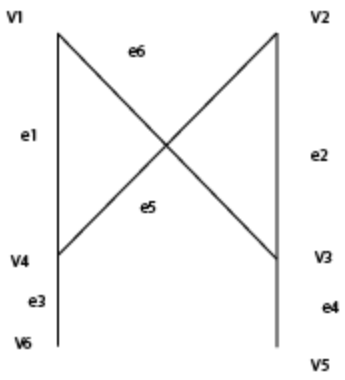
To Prove:-

1. Subset Cover
2. Vertex Cover ≤ρ Subset Cover
3. Subset Cover≤ρ Vertex Cover
4. Subset Cover ϵ NP

1) Subset Cover

Definition: - Number of a subset of edges after making the union for a get all the edges of the complete graph G, and that is called Subset Cover.

According to the graph G, which you have created the size of Subset Cover=2



1. v1{e1,e6} v2{e5,e2} v3{e2,e4,e6} v4{e1,e3,e5} v5{e4} v6{e3}
2. v3Uv4= {e1, e2, e3, e4, e5, e6} complete set of edges after the union of vertices.

2) Vertex Cover ≤ρ Subset Cover

In a graph G of vertices N, if there exists a Vertex Cover of size k, then there must also exist a Subset Cover of size k even. If you can achieve after the Reduction from Vertex Cover to Subset Cover within a polynomial time, which means you did right.

3) Subset Cover ≤ρ Vertex Cover

Just for verification of the output perform the Reduction and create Clique and via an equation, N-K verifies the Clique also and through Clique you can quickly generate 3CNF and after solving the Boolean function of 3CNF in the polynomial time. You will get output. It means the output has been verified.

4) Subset Cover ϵ NP:-

Proof: - As you know very well, you can get the Subset-Cover through Vertex Cover and Vertex Cover through Clique and to convert the decision-based NP problem into Clique firstly you have to convert into3CNF and 3CNF into SAT and SAT into CIRCUIT SAT that comes from NP.

Proof of NPC:-

The Reduction has been successfully made within the polynomial time form Vertex Cover to Subset Cover

Output has also been verified within the polynomial time as you did in the above conversation so, concluded that **SUBSET COVER also comes in NPC**.

Independent Set:

An independent set of a graph G = (V, E) is a subset V'⊆V of vertices such that every edge in E is incident on at most one vertex in V.' The independent-set problem is to find a largest-size independent set in G. It is not hard to find small independent sets, e.g., a small independent set is an individual node, but it is hard to find large independent sets.

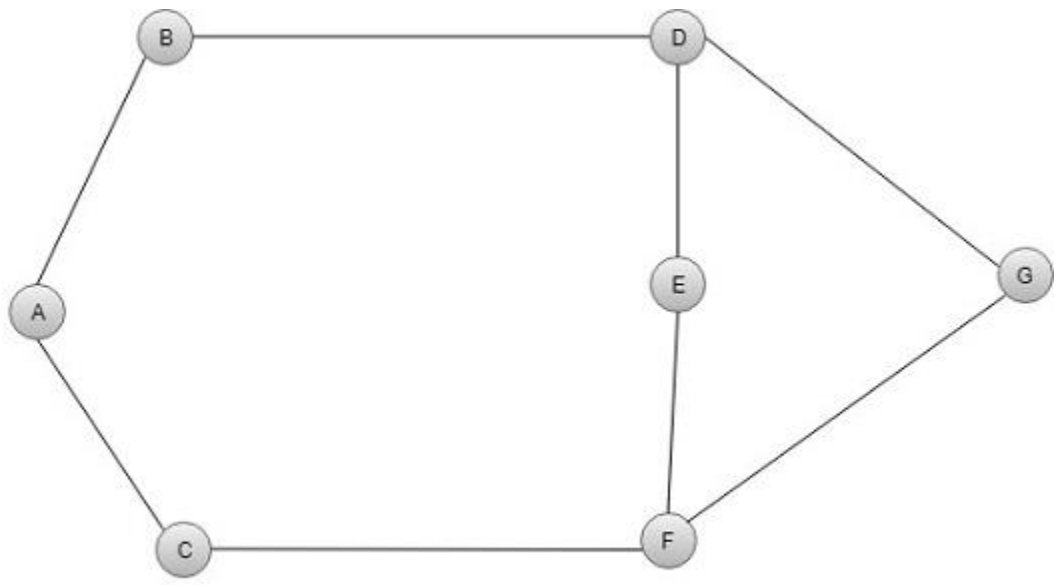


Fig:A graph with large independent sets of size 4 and smallest vertex cover of size 3.

Approximate Algorithms

Introduction:

An Approximate Algorithm is a way of approach **NP-COMPLETENESS** for the optimization problem. This technique does not guarantee the best solution. The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at the most polynomial time. Such algorithms are called approximation algorithm or heuristic algorithm.

- For the traveling salesperson problem, the optimization problem is to find the shortest cycle, and the approximation problem is to find a short cycle.
- For the vertex cover problem, the optimization problem is to find the vertex cover with fewest vertices, and the approximation problem is to find the vertex cover with few vertices.

Performance Ratios

Suppose we work on an optimization problem where every solution carries a cost. An Approximate Algorithm returns a legal solution, but the cost of that legal solution may not be optimal.

For Example, suppose we are considering for a **minimum size vertex-cover (VC)**. An approximate algorithm returns a VC for us, but the size (cost) may not be minimized.

Another Example is we are considering for a **maximum size Independent set (IS)**. An approximate Algorithm returns an IS for us, but the size (cost) may not be maximum. Let C be the cost of the solution returned by an approximate algorithm, and C* is the cost of the optimal solution.

We say the approximate algorithm has an approximate ratio P (n) for an input size n, where

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq P(n)$$

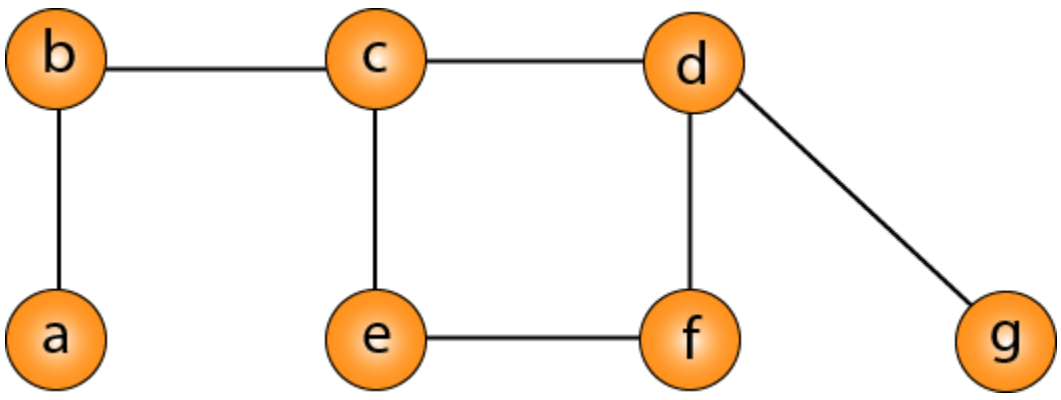
Intuitively, the approximation ratio measures how bad the approximate solution is distinguished with the optimal solution. A large (small) approximation ratio measures the solution is much worse than (more or less the same as) an optimal solution.

Observe that P (n) is always ≥ 1, if the ratio does not depend on n, we may write P. Therefore, a 1-approximation algorithm gives an optimal solution. Some problems have polynomial-time approximation algorithm with small constant approximate ratios, while others have best-known polynomial time approximation algorithms whose approximate ratios grow with n.

Vertex Cover

A Vertex Cover of a graph G is a set of vertices such that each edge in G is incident to at least one of these vertices.

The decision vertex-cover problem was proven NPC. Now, we want to solve the optimal version of the vertex cover problem, i.e., we want to find a minimum size vertex cover of a given graph. We call such vertex cover an optimal vertex cover C*.

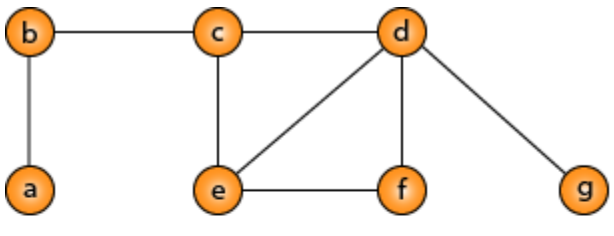


An approximate algorithm for vertex cover:

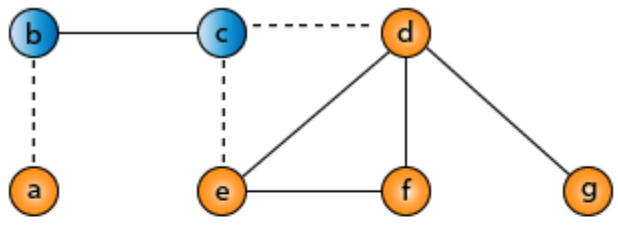
```
1. Approx-Vertex-Cover (G = (V, E))
2. {
3.         C = empty-set;
4.       E'= E;
5.       While E' is not empty do
6.         {
```

7. Let (u, v) be any edge in E': (*)
8. Add u and v to C;
9. Remove from E' all edges incident to
10. u or v;
11. }
12. Return C;
13. }

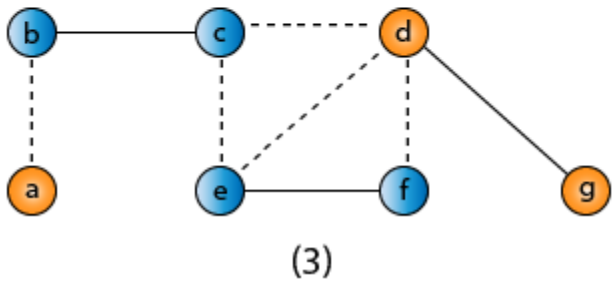
The idea is to take an edge (u, v) one by one, put both vertices to C, and remove all the edges incident to u or v. We carry on until all edges have been removed. C is a VC. But how good is C?



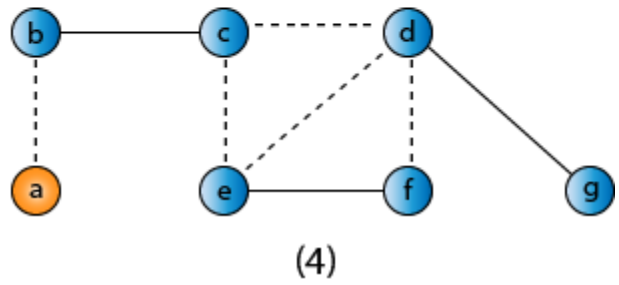
(1)



(2)



(3)



(4)

VC = {b, c, d, e, f, g}

Traveling-salesman Problem

In the traveling salesman Problem, a salesman must visits n cities. We can say that salesman wishes to make a tour or Hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. There is a non-negative cost c (i, j) to travel from the city i to city j. The goal is to find a tour of minimum cost. We assume that every two cities are connected. Such problems are called Traveling-salesman problem (TSP).

We can model the cities as a complete graph of n vertices, where each vertex represents a city.

It can be shown that TSP is NPC.

If we assume the cost function c satisfies the triangle inequality, then we can use the following approximate algorithm.

Triangle inequality

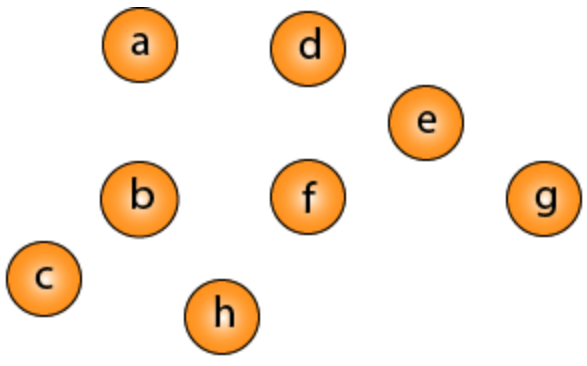
Let u, v, w be any three vertices, we have

$$c(u, w) \leq c(u, v) + c(v, w)$$

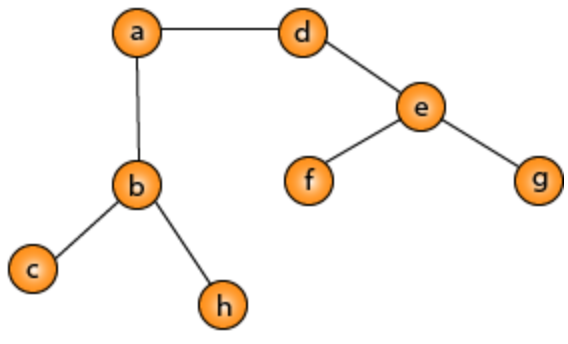
One important observation to develop an approximate solution is if we remove an edge from H*, the tour becomes a spanning tree.

1. Approx-TSP (G= (V, E))
2. {
3. 1. Compute a MST T of G;
4. 2. Select any vertex r is the root of the tree;
5. 3. Let L be the list of vertices visited in a preorder tree walk of T;
6. 4. Return the Hamiltonian cycle H that visits the vertices in the order L;
7. }

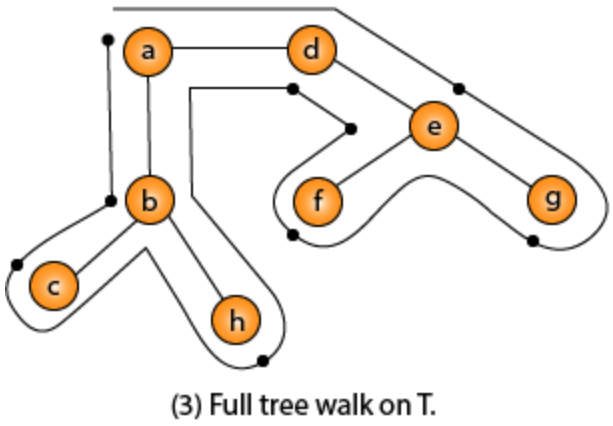
Traveling-salesman Problem



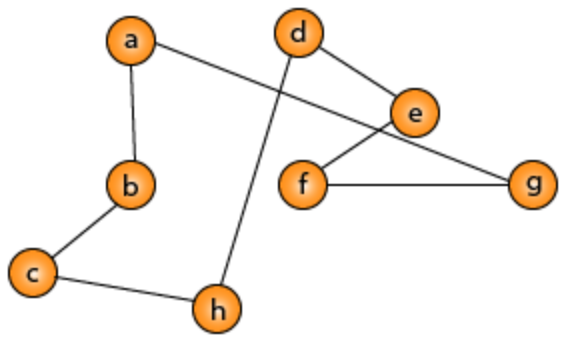
(1) A given set of points



(2) MST T



(3) Full tree walk on T.



(4) A preorder sequence gives a tour H.

Intuitively, Approx-TSP first makes a full walk of MST T, which visits each edge exactly two times. To create a Hamiltonian cycle from the full walk, it bypasses some vertices (which corresponds to making a shortcut)

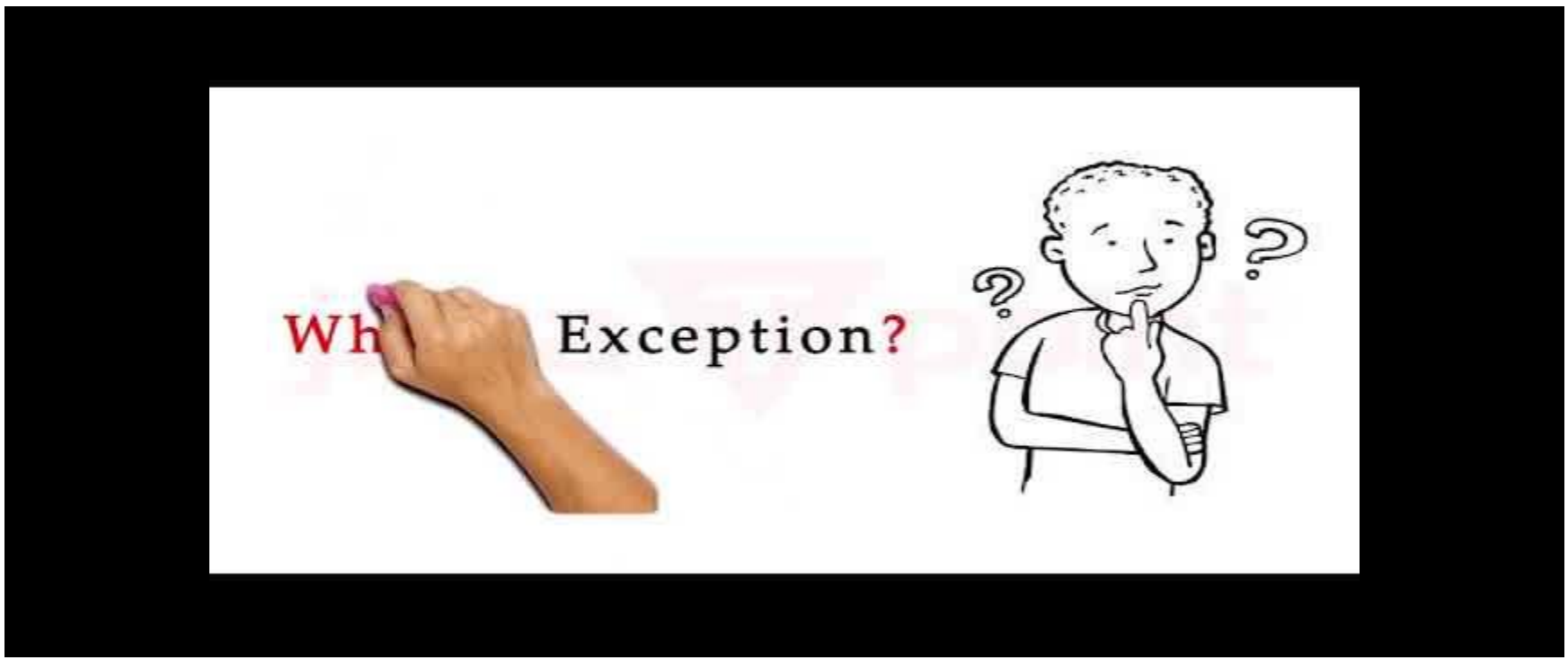
String Matching Introduction

String Matching Algorithm is also called "String Searching Algorithm." This is a vital class of string algorithm is declared as "this is the method to find a place where one is several strings are found within the larger string."

Given a text array, T [1.....n], of n character and a pattern array, P [1.....m], of m characters. The problems are to find an integer s, called **valid shift** where 0 ≤ s < n-m and T [s+1.....s+m] = P [1.....m]. In other words, to find even if P in T, i.e., where P is a substring of T. The item of P and T are character drawn from some finite alphabet such as {0, 1} or {A, B .....Z, a, b..... z}.Given a string T [1.....n], the **substrings** are represented as T [i.....j] for some 0≤i ≤ j≤n-1, the string formed by the characters in T from index i to index j, inclusive. This process that a string is a substring of itself (take i = 0 and j =m).

The **proper substring** of string T [1.....n] is T [1.....j] for some 0<i ≤ j≤n-1. That is, we must have either i>0 or j < m-1.

Using these descriptions, we can say given any string T [1.....n], the substrings are



1. T [i.....j] = T [i] T [i +1] T [i+2].....T [j] **for** some 0≤i ≤ j≤n-1.

And proper substrings are

1. T [i.....j] = T [i] T [i +1] T [i+2].....T [j] **for** some 0≤i ≤ j≤n-1.

Note: If i>j, then T [i.....j] is equal to the empty string or null, which has length zero.

Algorithms used for String Matching:

There are different types of method is used to finding the string

1. The Naive String Matching Algorithm
2. The Rabin-Karp-Algorithm
3. Finite Automata
4. The Knuth-Morris-Pratt Algorithm
5. The Boyer-Moore Algorithm

The Naive String Matching Algorithm

The naïve approach tests all the possible placement of Pattern P [1.....m] relative to text T [1.....n]. We try shift s = 0, 1.....n-m, successively and for each shift s. Compare T [s+1.....s+m] to P [1.....m].

The naïve algorithm finds all valid shifts using a loop that checks the condition P [1.....m] = T [s+1.....s+m] for each of the n - m +1 possible value of s.

NAIVE-STRING-MATCHER (T, P)

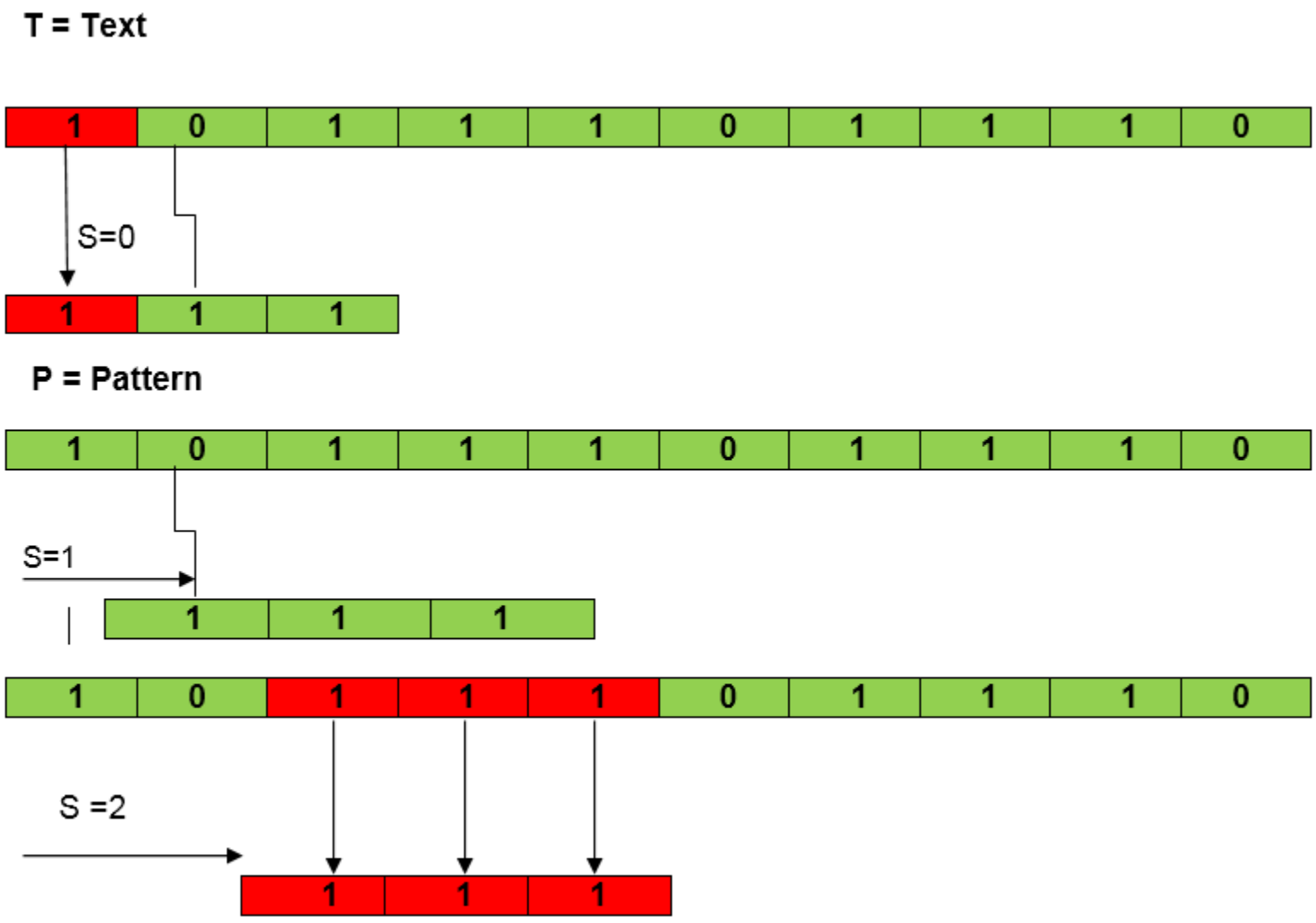
```
1. n ← length [T]
2. m ← length [P]
3. for s ← 0 to n -m
4. do if P [1.....m] = T [s + 1....s + m]
5. then print "Pattern occurs with shift" s
```

Analysis: This for loop from 3 to 5 executes for n-m + 1(we need at least m characters at the end) times and in iteration we are doing m comparisons. So the total complexity is O (n-m+1).

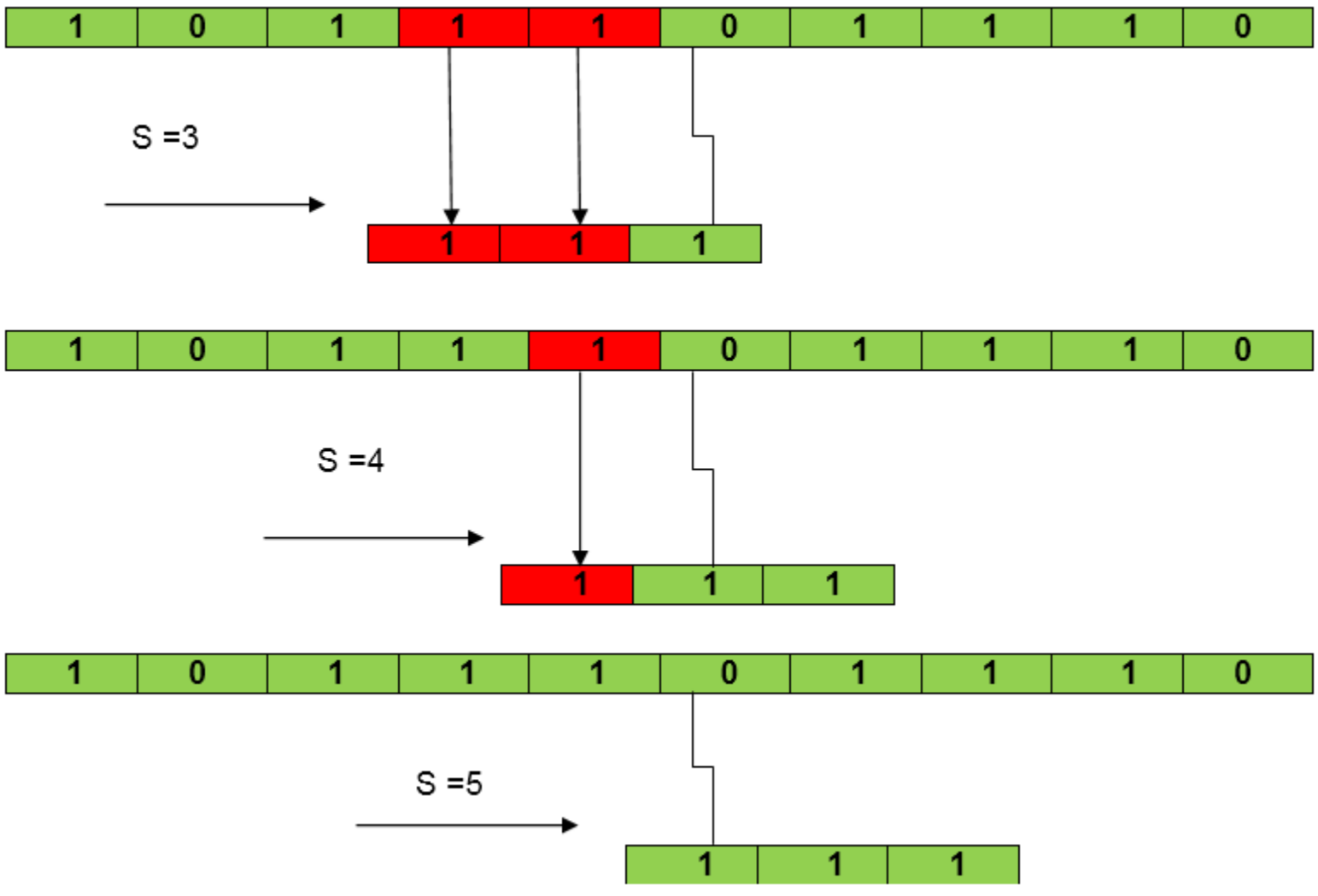
Example:

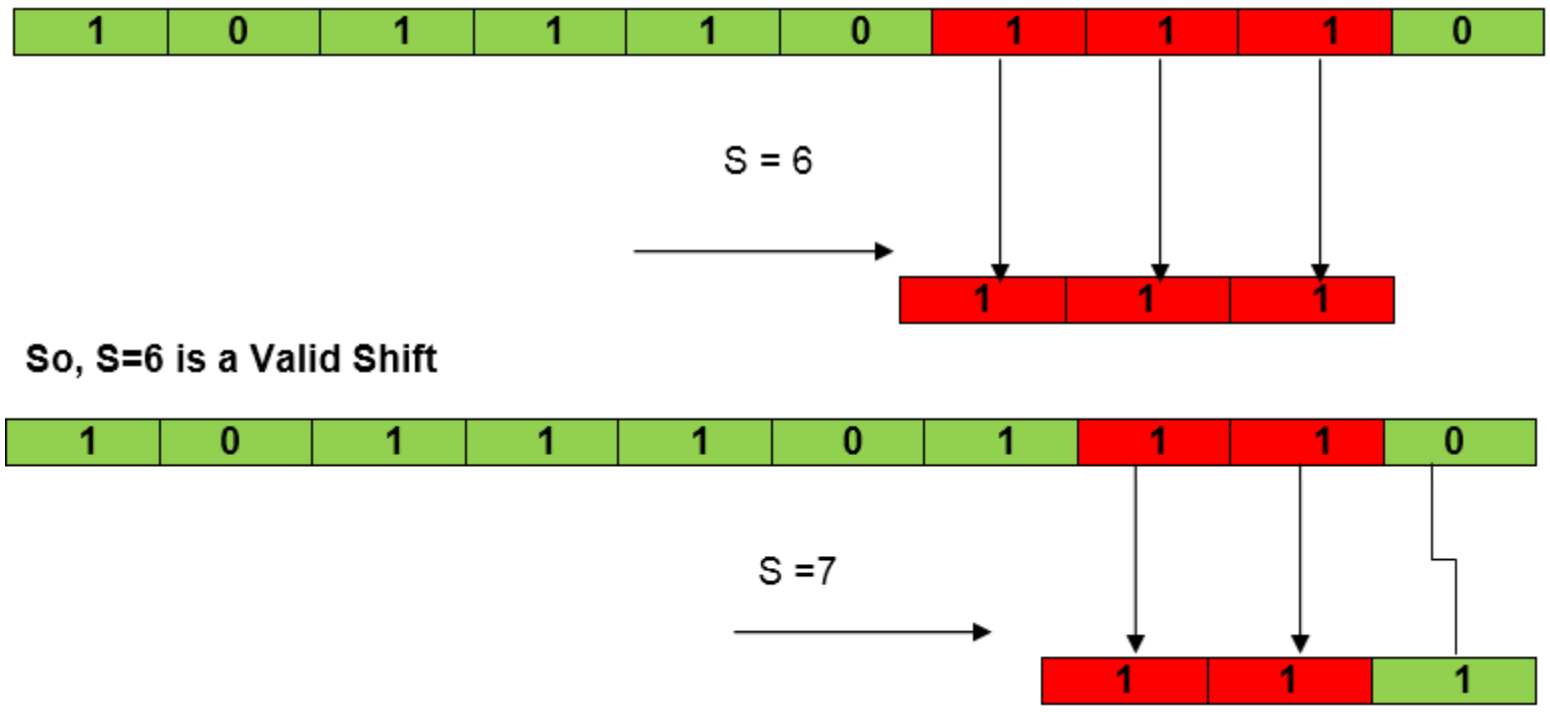
1. Suppose T = 1011101110
2. P = 111
3. Find all the Valid Shift

Solution:



So, S=2 is a Valid Shift





The Rabin-Karp-Algorithm

The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character subsequences of text to be compared. If the hash values are unequal, the algorithm will determine the hash value for next M-character sequence. If the hash values are equal, the algorithm will analyze the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and character matching is only required when the hash values match.

RABIN-KARP-MATCHER (T, P, d, q)

```
1. n ← length [T]
2. m  ← length [P]
3. h  ←  d^(m-1) mod q
4. p ←  0
5. t0 ←  0
6. for i ← 1 to m
7. do p ←  (dp + P[i]) mod q
8. t0 ← (dt0+T [i]) mod q
9. for s  ←  0 to n-m
10. do if p = ts
11. then if P [1.....m] = T [s+1.....s + m]
12. then "Pattern occurs with shift" s
13. If s < n-m
14. then ts+1 ←  (d (ts-T [s+1]h)+T [s+m+1])mod q
```

Example: For string matching, working module q = 11, how many spurious hits does the Rabin-Karp matcher encounters in Text T = 31415926535.....

1. T = 31415926535.....
2. P = 26
3. Here T.Length =11 so Q = 11
4. And P mod Q = 26 mod 11 = 4
5. Now find the exact match of P mod Q...

Solution:

T = 3 1 4 1 5 9 2 6 5 3 5

P = 2 6

S = 0

3 1 4 1 5 9 2 6 5 3 5

31 mod 11 = 9 not equal to 4

S =1

3 1 4 1 5 9 2 6 5 3 5

14 mod 11 = 3 not equal to 4

S =2

3 1 4 1 5 9 2 6 5 3 5

41 mod 11 = 8 not equal to 4

S = 3

3 1 4 1 5 9 2 6 5 3 5

15 mod 11 = 4 equal to 4 SPURIOUS HIT

S = 4

3 1 4 1 5 9 2 6 5 3 5

59 mod 11 = 4 equal to 4 SPURIOUS HIT

S = 5

3 1 4 1 5 9 2 6 5 3 5

92 mod 11 = 4 equal to 4 SPURIOUS HIT

S = 6

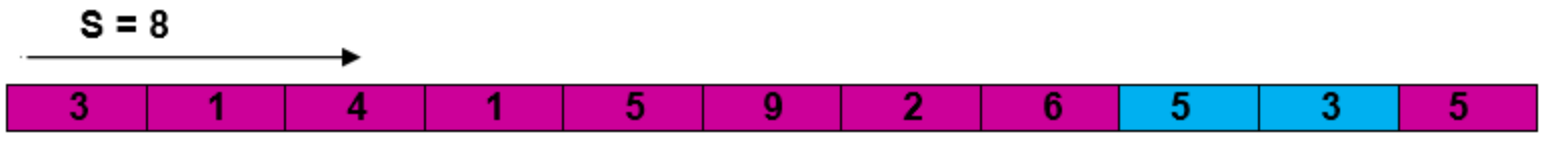
3 1 4 1 5 9 2 6 5 3 5

26 mod 11 = 4 EXACT MATCH

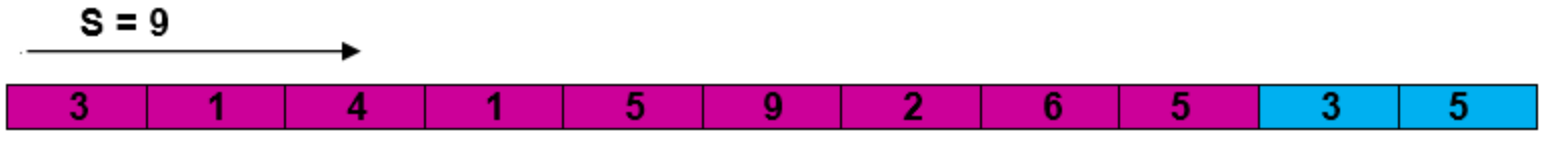
S = 7

3 1 4 1 5 9 2 6 5 3 5

65 mod 11 = 10 not equal to 4



53 mod 11 = 9 not equal to 4



35 mod 11 = 2 not equal to 4

The Pattern occurs with shift 6.

Complexity:

The running time of **RABIN-KARP-MATCHER** in the worst case scenario **O ((n-m+1) m** but it has a good average case running time. If the expected number of strong shifts is small **O (1)** and prime q is chosen to be quite large, then the Rabin-Karp algorithm can be expected to run in time **O (n+m)** plus the time to require to process spurious hits.

String Matching with Finite Automata

The string-matching automaton is a very useful tool which is used in string matching algorithm. It examines every character in the text exactly once and reports all the valid shifts in O (n) time. The goal of string matching is to find the location of specific text pattern within the larger body of text (a sentence, a paragraph, a book, etc.)

Finite Automata:

A finite automaton **M** is a 5-tuple **$(Q, q_0, A, \sum \delta)$**, where

- Q is a finite set of **states**,
- $q_0 \in Q$ is the **start state**,
- $A \subseteq Q$ is a notable set of **accepting states**,
- $\sum$ is a **finite input alphabet**,
- δ is a function from **Q x $\sum$** into **Q** called the **transition function** of **M**.

The finite automaton starts in state **q_0** and reads the characters of its input string one at a time. If the automaton is in state q and reads input character a, it moves from state q to state δ (q, a). Whenever its current state q is a member of A, the machine M has accepted the string read so far. An input that is not allowed is **rejected**.

A finite automaton M induces a function $\emptyset$ called the called the **final-state function**, from $\sum^*$ to Q such that $\emptyset$(w) is the state M ends up in after scanning the string w. Thus, M accepts a string w if and only if $\emptyset(w) \in A$.

The function f is defined as

$\emptyset(\in) = q_0$

$\emptyset(wa) = \delta((\emptyset(w), a)$ for $w \in \sum^*, a \in \sum$)

FINITE- AUTOMATON-MATCHER (T,δ, m),

```
1. n ← length [T]
2. q ← 0
3. for i ← 1 to n
4. do q ← δ (q, T[i])
5. If q =m
6. then s←i-m
7. print "Pattern occurs with shift s" s
```

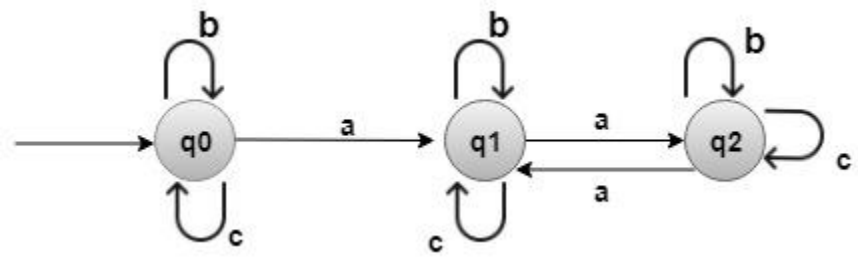
The primary loop structure of FINITE- AUTOMATON-MATCHER implies that its running time on a text string of length n is O (n).

Computing the Transition Function: The following procedure computes the transition function δ from given pattern P [1.....m]

COMPUTE-TRANSITION-FUNCTION (P, ∑)

```
1. m ← length [P]
2. for q ← 0 to m
3. do for each character a ∈ ∑*
4. do k ← min (m+1, q+2)
5. repeat k←k-1
6. Until
7. δ(q,a)←k
8. Return δ
```

Example: Suppose a finite automaton which accepts even number of a's where ∑ = {a, b, c}



Solution:

q_0 is the initial state.

q_2 is the accepting or final state, and transition function δ is defined as

$$\delta(q_0, a) = q_1 \qquad \delta(q_1, c) = q_1$$

$$\delta(q_0, b) = q_0 \qquad \delta(q_2, a) = q_1$$

$$\delta(q_0, c) = q_0 \qquad \delta(q_2, b) = q_2$$

$$\delta(q_1, a) = q_2 \qquad \delta(q_2, c) = q_2$$

$$\delta(q_1, b) = q_1 \qquad \delta(q_1, c) = q_1$$

Suppose w is a string such as

w = b c a a b c a a a b a c, then $\delta(q_0,$ b c a a a b c a a a b a c)

$= \delta(q_0,$ c a a b c a a a b a c)

$= \delta(q_0,$ a a b c a a a b a c) $= \delta(q_1,$ a b c a a a b a c)

$= \delta(q_2,$ b c a a a b a c) $= \delta(q_2,$ c a a a b a c)

$= \delta(q_2,$ a a a b a c) $= \delta(q_1,$ a a b a c)

$= \delta(q_2, a, b, a, c) = \delta(q_1,$ b a c)

$= \delta(q_1, b, c) = \delta(q_2, c) = q_2$(Accepting State)

Thus, given finite automation accepts w.

The Knuth-Morris-Pratt (KMP)Algorithm

Knuth-Morris and Pratt introduce a linear time algorithm for the string matching problem. A matching time of O (n) is achieved by avoiding comparison with an element of 'S' that have previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs

Components of KMP Algorithm:

1. The Prefix Function (Π): The Prefix Function, Π for a pattern encapsulates knowledge about how the pattern matches against the shift of itself. This information can be used to avoid a useless shift of the pattern 'p.' In other words, this enables avoiding backtracking of the string 'S.'

2. The KMP Matcher: With string 'S,' pattern 'p' and prefix function 'Π' as inputs, find the occurrence of 'p' in 'S' and returns the number of shifts of 'p' after which occurrences are found.

The Prefix Function (Π)

Following pseudo code compute the prefix function, Π:

```
COMPUTE- PREFIX- FUNCTION (P)
 1. m ←length [P]            //'p' pattern to be matched
 2. Π [1] ← 0
 3. k ← 0
 4. for q ← 2 to m
 5. do while k > 0 and P [k + 1] ≠ P [q]
 6. do k ← Π [k]
 7. If P [k + 1] = P [q]
 8. then k← k + 1
 9. Π [q] ← k
 10. Return Π
```

Running Time Analysis:

In the above pseudo code for calculating the prefix function, the for loop from step 4 to step 10 runs 'm' times. Step1 to Step3 take constant time. Hence the running time of computing prefix function is O (m).

Example: Compute Π for the pattern 'p' below:

Solution:

```
Initially: m = length [p] = 7
          Π [1] = 0
          k = 0
```

Step 1: q = 2, k = 0

Π [2] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0					

Step 2: q = 3, k = 0

Π [3] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1				

Step3: q =4, k =1

Π [4] = 2

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	A
π	0	0	1	2			

Step4: q = 5, k =2

Π [5] = 3

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3		

Step5: q = 6, k = 3

Π [6] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	

Step6: q = 7, k = 1

Π [7] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
π	0	0	1	2	3	0	1

After iteration 6 times, the prefix function computation is complete:

q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

The KMP Matcher:

The KMP Matcher with the pattern 'p,' the string 'S' and prefix function 'Π' as input, finds a match of p in S. Following pseudo code compute the matching component of KMP algorithm:

KMP-MATCHER (T, P)

```
1. n ← length [T]
2. m ← length [P]
```

```
3. Π← COMPUTE-PREFIX-FUNCTION (P)
4. q ← 0              // numbers of characters matched
5. for i ← 1 to n     // scan S from left to right
6. do while q > 0 and P [q + 1] ≠ T [i]
7. do q ← Π [q]              // next character does not match
8. If P [q + 1] = T [i]
9. then q ← q + 1            // next character matches
10. If q = m                            // is all of p matched?
11. then print "Pattern occurs with shift" i - m
12. q ← Π [q]                       // look for the next match
```

Running Time Analysis:

The for loop beginning in step 5 runs 'n' times, i.e., as long as the length of the string 'S.' Since step 1 to step 4 take constant times, the running time is dominated by this for the loop. Thus running time of the matching function is O (n).

Example: Given a string 'T' and pattern 'P' as follows:

T:	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a

P:	a	b	a	b	a	c	a

Let us execute the KMP Algorithm to find whether 'P' occurs in 'T.'

For 'p' the prefix function, ? was computed previously and is as follows:

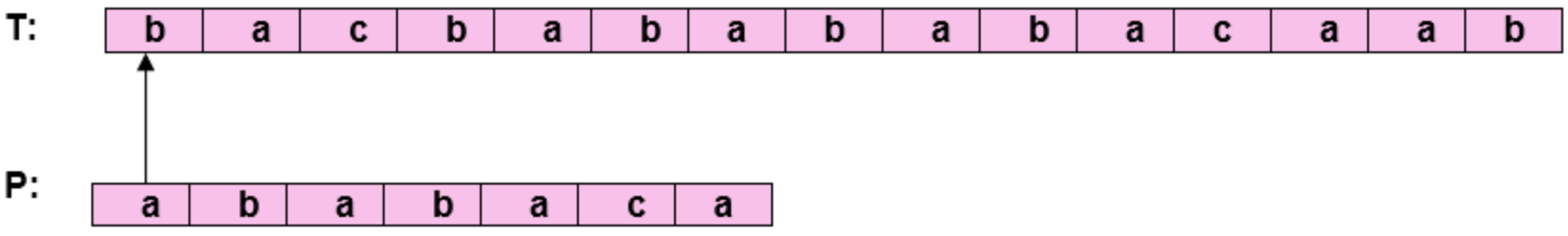
q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
π	0	0	1	2	3	0	1

Solution:

```
Initially: n = size of T = 15
m = size of P = 7
```

Step1: i=1, q=0

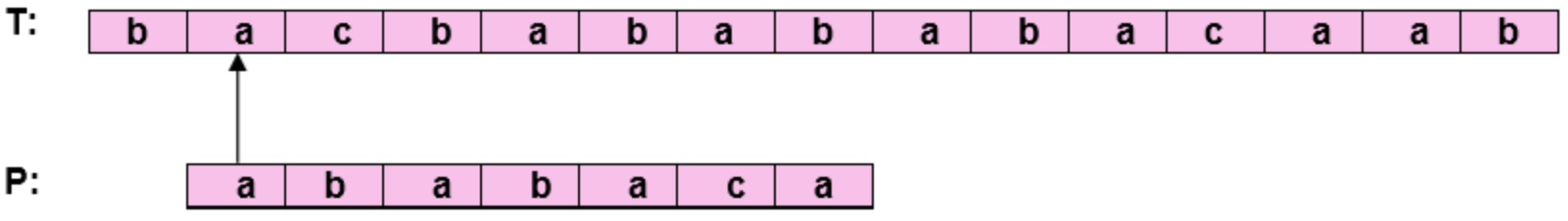
Comparing P [1] with T [1]



P [1] does not match with T [1]. 'p' will be shifted one position to the right.

Step2: i = 2, q = 0

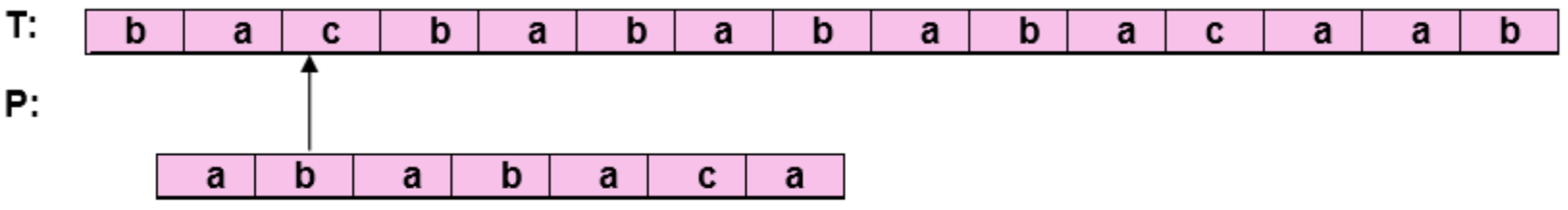
Comparing P [1] with T [2]



P [1] matches T [2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1

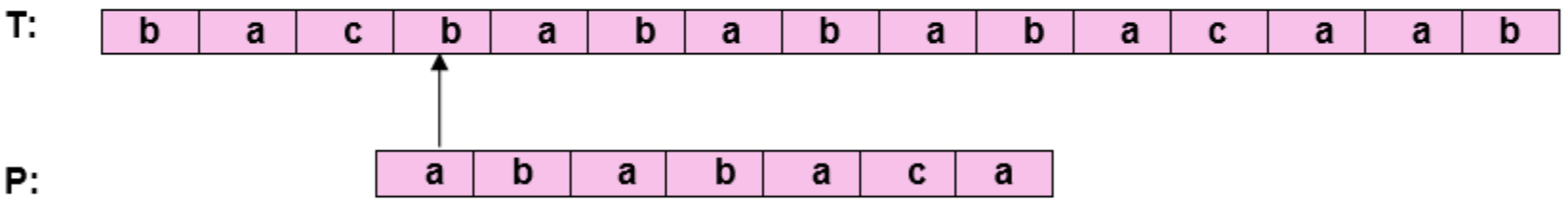
Comparing P [2] with T [3] P [2] doesn't match with T [3]



Backtracking on p, Comparing P [1] and T [3]

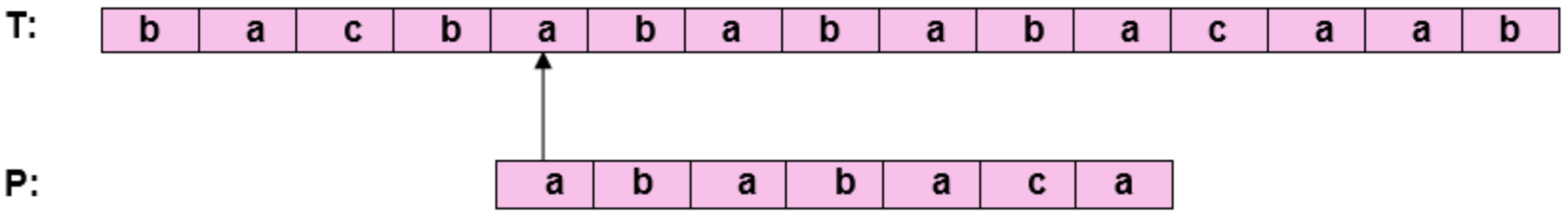
Step4: i = 4, q = 0

Comparing P [1] with T [4] P [1] doesn't match with T [4]



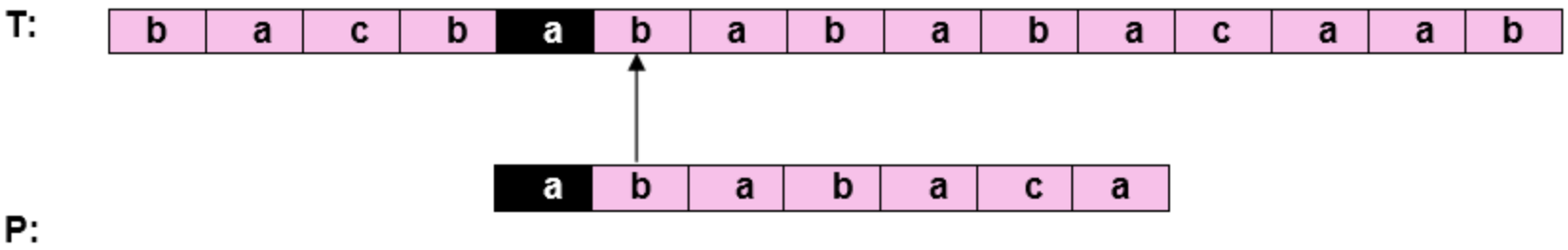
Step5: i = 5, q = 0

Comparing P [1] with T [5] P [1] match with T [5]



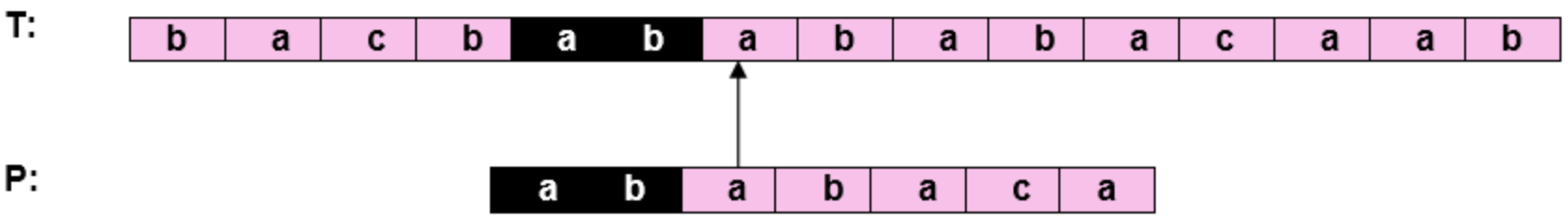
Step6: i = 6, q = 1

Comparing P [2] with T [6] P [2] matches with T [6]



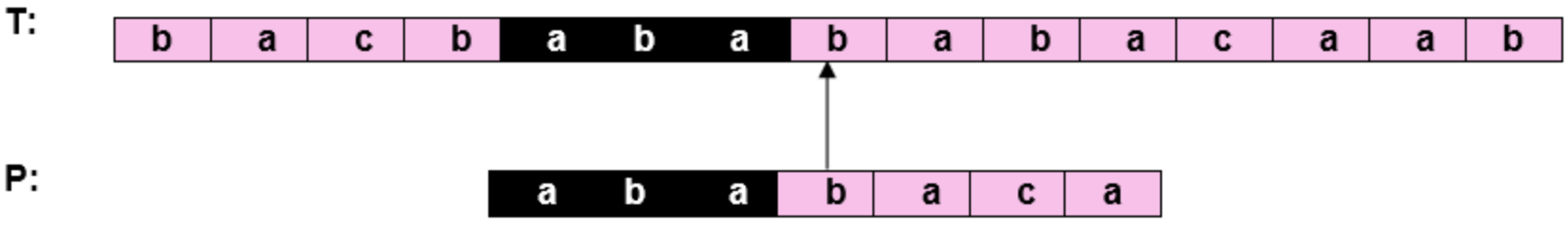
Step7: i = 7, q = 2

Comparing P [3] with T [7] P [3] matches with T [7]



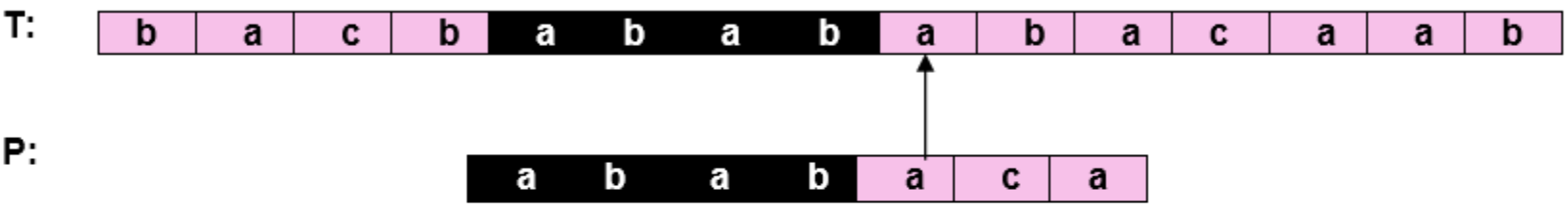
Step8: i = 8, q =3

Comparing P [4] with T [8] P [4] matches with T [8]



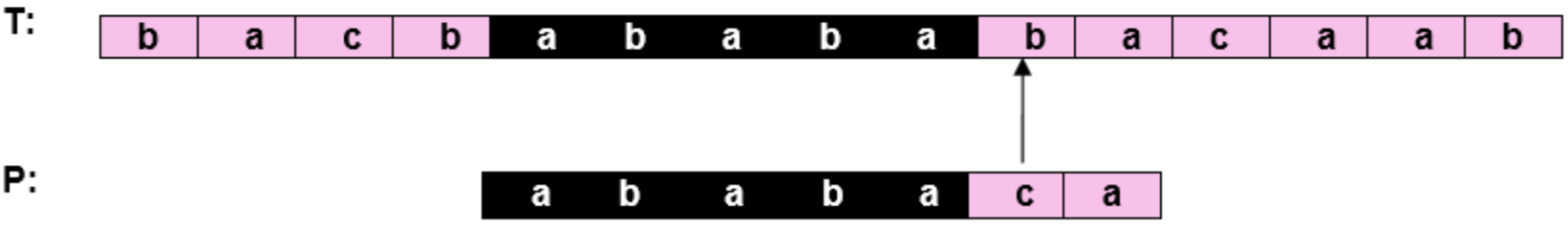
Step9: i = 9, q = 4

Comparing P [5] with T [9] P [5] matches with T [9]



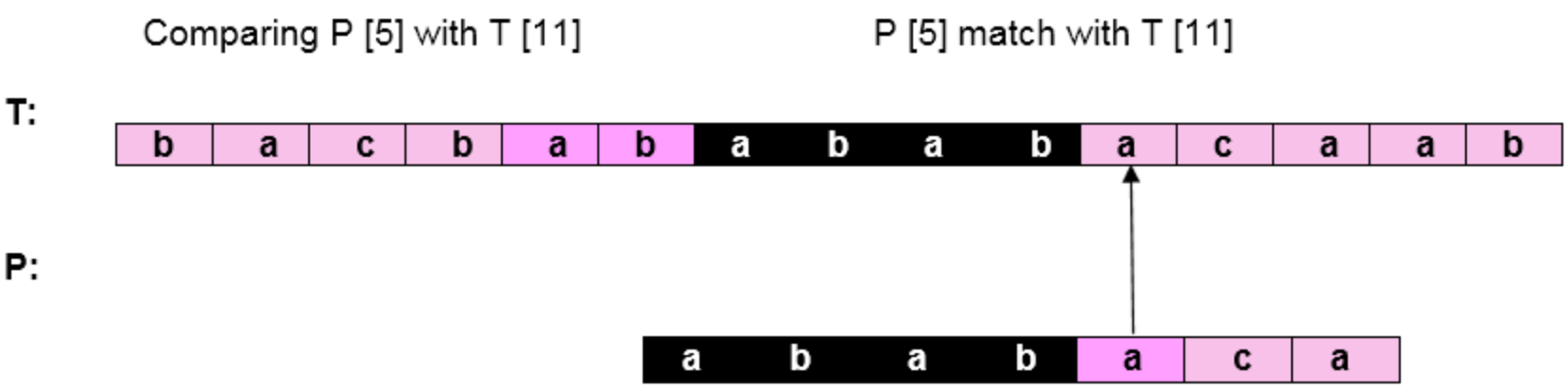
Step10: i = 10, q = 5

Comparing P [6] with T [10] P [6] doesn't match with T [10]

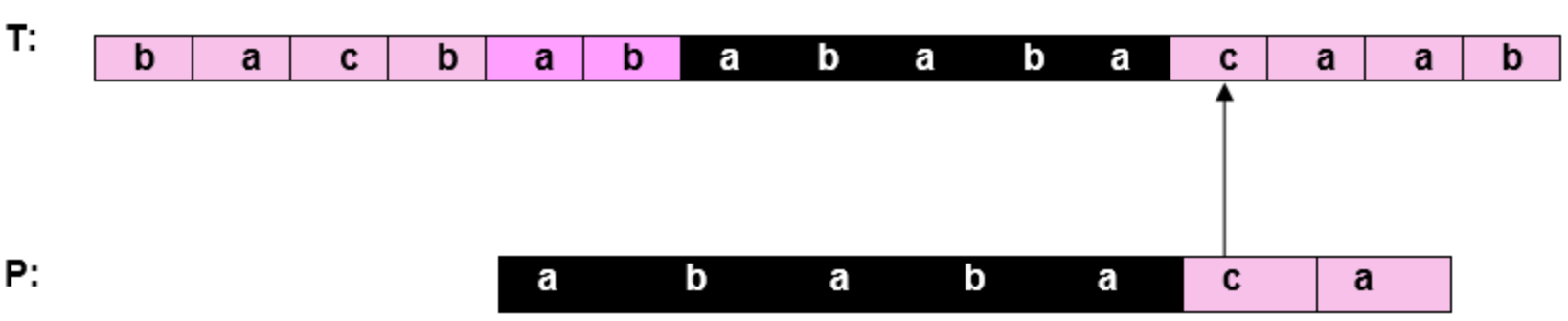


Backtracking on p, Comparing P [4] with T [10] because after mismatch q = π [5] = 3

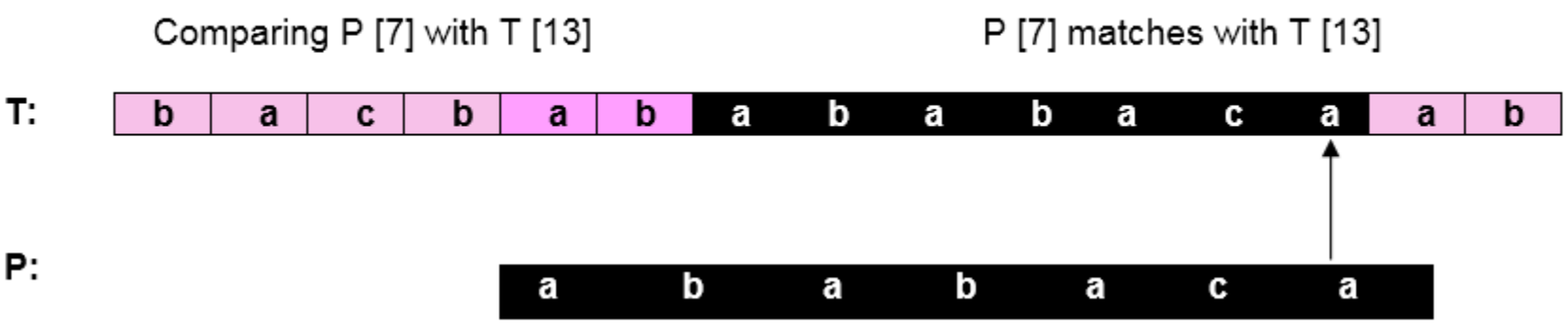
Step11: i = 11, q =4



Step12: i = 12, q = 5



Step13: i = 3, q = 6



Pattern 'P' has been found to complexity occur in a string 'T.' The total number of shifts that took place for the match to be found is i-m = 13 - 7 = 6 shifts.

The Boyer-Moore Algorithm

Robert Boyer and J Strother Moore established it in 1977. The B-M String search algorithm is a particularly efficient algorithm and has served as a standard benchmark for string search algorithm ever since.

The B-M algorithm takes a 'backward' approach: the pattern string (P) is aligned with the start of the text string (T), and then compares the characters of a pattern from right to left, beginning with rightmost character.

If a character is compared that is not within the pattern, no match can be found by analyzing any further aspects at this position so the pattern can be changed entirely past the mismatching character.

For deciding the possible shifts, B-M algorithm uses two preprocessing strategies simultaneously. Whenever a mismatch occurs, the algorithm calculates a variation using both approaches and selects the more significant shift thus, if make use of the most effective strategy for each case.

The two strategies are called heuristics of B - M as they are used to reduce the search. They are:

1. Bad Character Heuristics
2. Good Suffix Heuristics

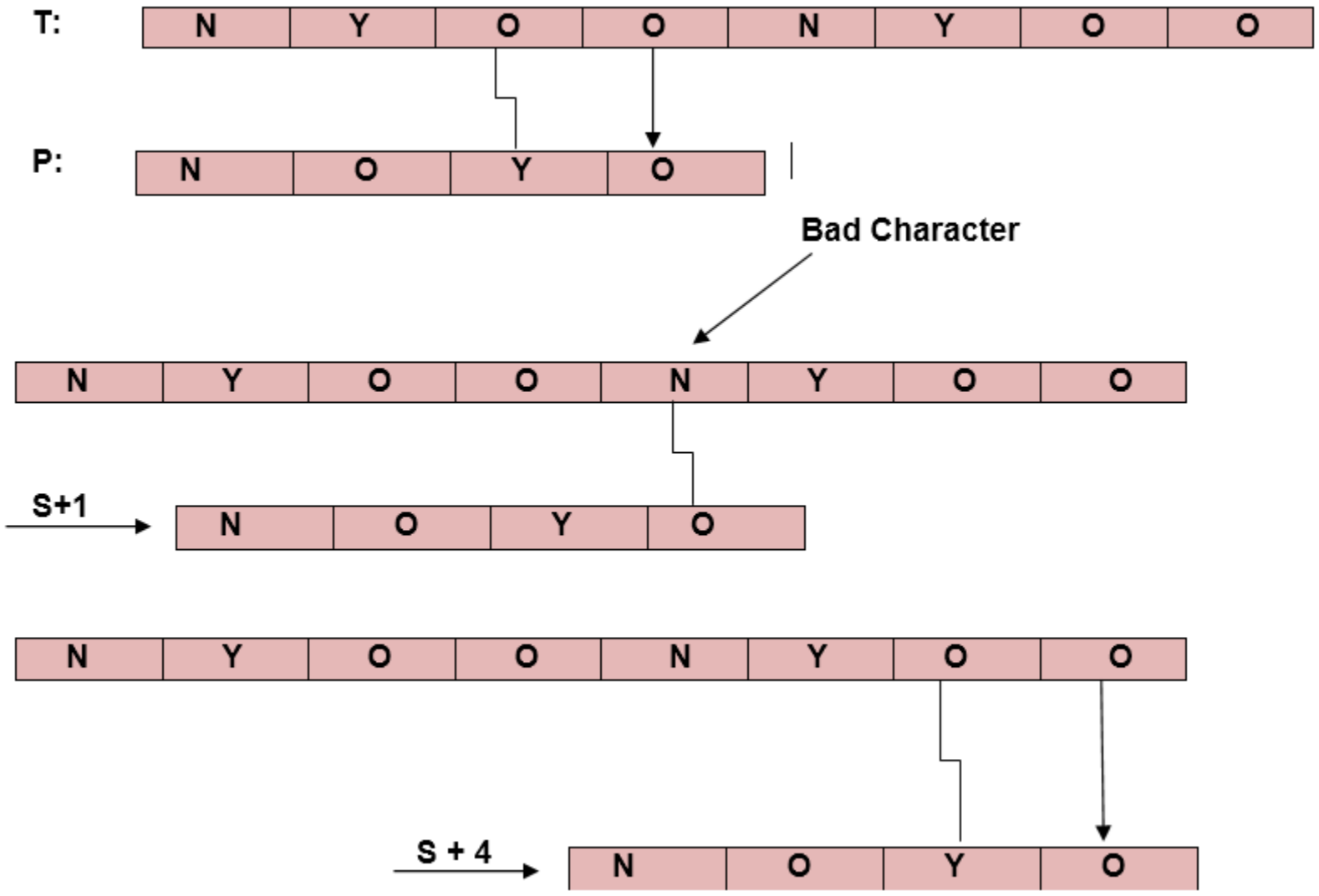
1. Bad Character Heuristics

This Heuristics has two implications:

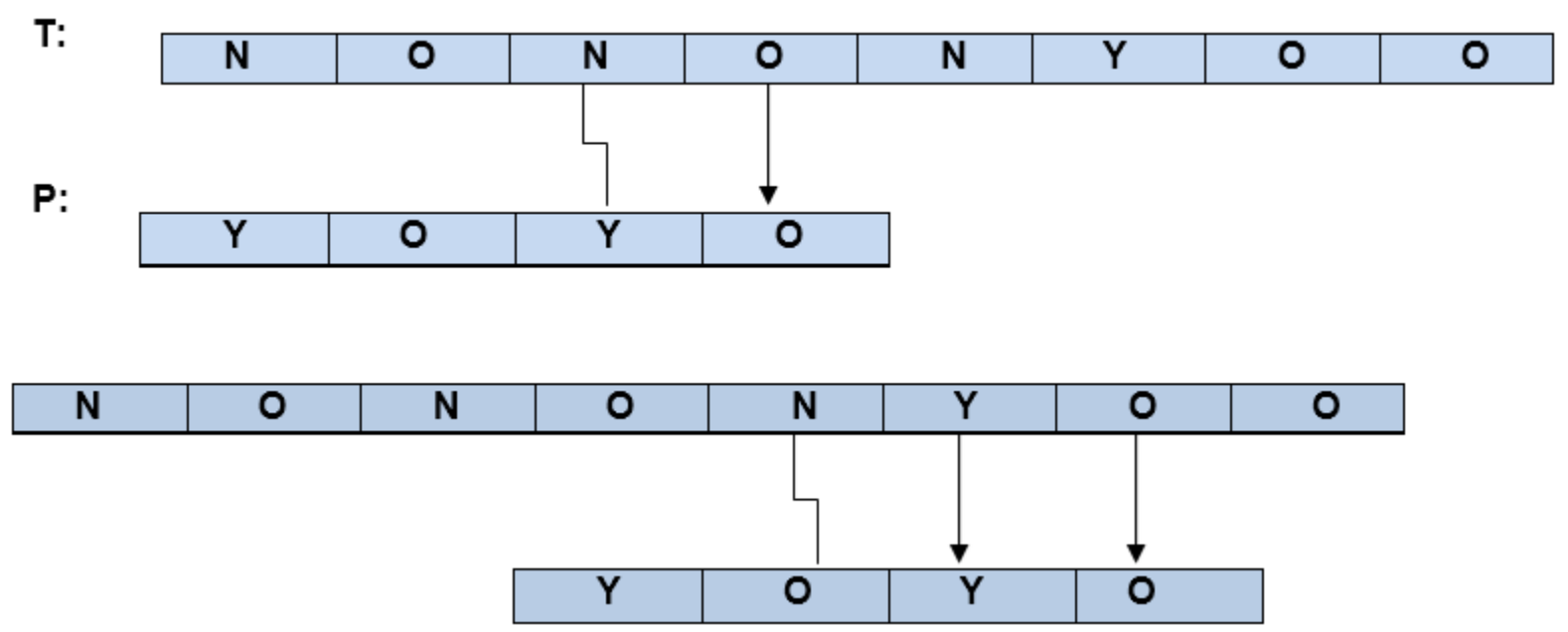
- Suppose there is a character in a text in which does not occur in a pattern at all. When a mismatch happens at this character (called as bad character), the whole pattern can be changed, begin matching form substring next to this 'bad character.'
- On the other hand, it might be that a bad character is present in the pattern, in this case, align the nature of the pattern with a bad character in the text.

Thus in any case shift may be higher than one.

Example1: Let Text T = <nyoo nyoo> and pattern P = <noyo>



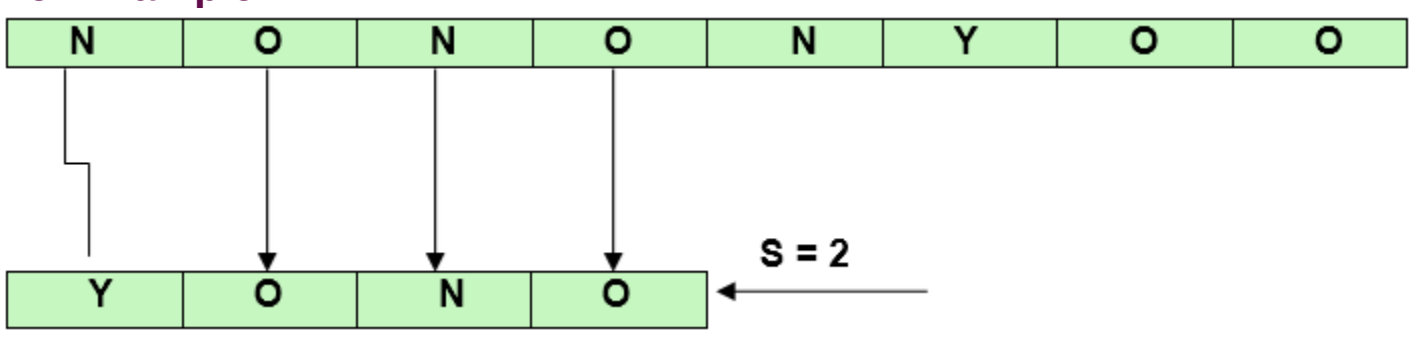
Example2: If a bad character doesn't exist the pattern then.



Problem in Bad-Character Heuristics:

In some cases, Bad-Character Heuristics produces some negative shifts.

For Example:



This means that we need some extra information to produce a shift on encountering a bad character. This information is about the last position of every aspect in the pattern and also the set of characters used in a pattern (often called the alphabet ∑of a pattern).

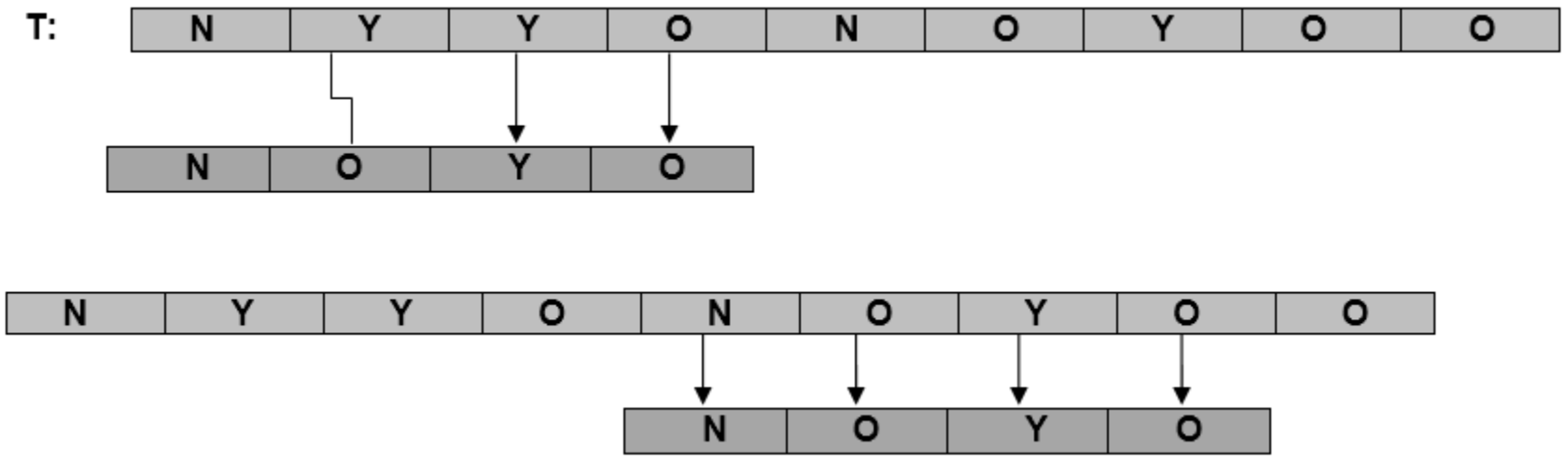
COMPUTE-LAST-OCCURRENCE-FUNCTION (P, m, ∑)

```
1. for each character a ∈ ∑
2. do λ [a] = 0
3. for j ← 1 to m
4. do λ [P [j]] ← j
5. Return λ
```

2. Good Suffix Heuristics:

A good suffix is a suffix that has matched successfully. After a mismatch which has a negative shift in bad character heuristics, look if a substring of pattern matched till bad character has a good suffix in it, if it is so then we have an onward jump equal to the length of suffix found.

Example:



COMPUTE-GOOD-SUFFIX-FUNCTION (P, m)

```
1. Π ← COMPUTE-PREFIX-FUNCTION (P)
2. P'← reverse (P)
3. Π'← COMPUTE-PREFIX-FUNCTION (P')
4. for j ← 0 to m
5. do ɣ [j] ← m - Π [m]
6. for l ← 1 to m
7. do j ← m - Π' [L]
8. If ɣ [j] > l - Π' [L]
9. then ɣ [j] ← 1 - Π'[L]
10. Return ɣ
```

BOYER-MOORE-MATCHER (T, P, ∑)

```
1. n ←length [T]
2. m ←length [P]
3. λ← COMPUTE-LAST-OCCURRENCE-FUNCTION (P, m, ∑ )
4. ɣ← COMPUTE-GOOD-SUFFIX-FUNCTION (P, m)
5. s ←0
6. While s ≤ n - m
7. do j ← m
8. While j > 0 and P [j] = T [s + j]
9. do j ←j-1
10. If j = 0
11. then print "Pattern occurs at shift" s
12. s ← s + ɣ[0]
13. else s ← s + max (ɣ [j], j -  λ[T[s+j]])
```

Complexity Comparison of String Matching Algorithm:

Algorithm	Preprocessing Time	Matching Time
Naive	O	(O (n - m + 1)m)
Rabin-Karp	O(m)	(O (n - m + 1)m)
Finite Automata	O(m|∑|)	O (n)
Knuth-Morris-Pratt	O(m)	O (n)
Boyer-Moore	O(|∑|)	(O ((n - m + 1) + |∑|))