Algorithm is a combination of a sequence of finite steps to solve a problem.

PROPERTIES OF ALGO

- It should terminate in finite time.
- It should produce at least 1 output.
- Every step in the algo should be effective/necessary.
- The algo must be deterministic i.e. for each input the output must remain the same.
- It is independent of programming language i.e. it can be implemented using any programming language.

STEPS REQUIRED TO DESIGN ALGO

1. Problem definition
2. Select designing approach
3. Draw Flowchart
4. Perform verification
5. Implementation using Coding
6. Time and Space complexity analysis

ALGORITHM ANALYSIS

To solve a problem, there can be multiple algorithms for the same. In that case, which one do we choose? To choose an algorithm, we first get the **time complexity** of the algorithms and choose the one which has the lowest time complexity. In cases where 2 algos have the **same time complexity, then check space complexity**.

Suppose a problem has only one possible algo, then we don't need to perform the time and space complexity analysis.

TIME COMPLEXITY

For any program P, the time complexity if the sum of the **compile time** and the **run time.** Compile time depends on the compiler which is a software and run time depends on the processor which is a hardware. Thus, we can write –

$$T(P) = C(P) + R(P)$$

Now, we can analyze the time complexity by first getting the **language of compiler** and the **type of processor** and then combining them to get the final time complexity. This type of analysis is called **Aposteriori (Relative) analysis.** This gives the **exact time complexity** but since it depends on type of processor and language of compiler, the time complexity here will be **system dependent** and different systems will have different complexity.

On the other hand, we can go for **Apriori (Absolute) analysis** which doesn't need to get the Compiler language and Processor type. It just gives a ballpark answer as to what can be the expected time

complexity of the type of program. Thus, it provides an **approximate time complexity** and the answer is **independent of system.**

APRIORI ANALYSIS

As mentioned above, it is a simple determination of the order of the magnitude of the statement. The number of times a statement executes while running is called its **order of magnitude.** This order of magnitude is represented as **Big O.** For example,

```c
#include <stdio.h>
void main()
{
    x = y + z;
}
```

In this case, there is 1 statement and this statement has an order of magnitude as **1** since the statement runs only once. Therefore, the time complexity will be $\boldsymbol{O(1)}$**.** Let us take another example,

```c
#include <stdio.h>
void main()
{
    x = y + z; //Statement 1
    for (i=0 ; i<=n ; i++) {
        x = y + z; //Statement 2
    }
}
```

In this program, Statement 1 will fire 1 time while Statement 2 will fire for n times. Therefore,

$$Order\ of\ magnitude = 1 + n$$

Hence,

$$\boldsymbol{T(P) = O(n)}$$

Additional questions on this are present in the Questions doc 😊

ASYMPTOTIC NOTATION $(\boldsymbol{O, \Omega, \Theta, o, \omega})$

These are mathematical tools to represent the time complexity of a program/algorithm. We have 5 such tools - $O, \Theta, \Omega, o, \omega$

Big – O notation $(\boldsymbol{O})$

For 2 functions $f(n)$ and $g(n)$ we can write $f(n) = O\big(g(n)\big)$ $\boldsymbol{iff}$ $\boldsymbol{f(n) \leq c * g(n)}$ for all integers $n \geq \eta_0$ such that the constants $c > 0$ and $\eta_0 \geq 1$

Big – Omega $(\boldsymbol{\Omega})$

For 2 functions $f(n)$ and $g(n)$ we can write $f(n) = \Omega\big(g(n)\big)$ $\boldsymbol{iff}$ $\boldsymbol{f(n) \geq c * g(n)}$ for all integers $n \geq \eta_0$ such that the constants $c > 0$ and $\eta_0 \geq 1$

Big – Theta $(\boldsymbol{\Theta})$

For 2 functions $f(n)$ and $g(n)$ we can write $f(n) = \Theta\big(g(n)\big)$ $\boldsymbol{iff}$ $\boldsymbol{f(n) = c * g(n)}$ for all integers $n \geq \eta_0$ such that the constants $c > 0$ and $\eta_0 \geq 1$

Small – O notation $(\boldsymbol{o})$

For 2 functions $f(n)$ and $g(n)$ we can write $f(n) = O\big(g(n)\big)$ $\boldsymbol{iff}$ $\boldsymbol{f(n) < c * g(n)}$ for all integers $n \geq \eta_0$ such that the constants $c > 0$ and $\eta_0 \geq 1$

Question

Assume $f(n) = n + 10$ and $g(n) = n$. Prove that $f(n) = O\big(g(n)\big)$

Answer

To prove a Big O relation, we need to satisfy –

$$f(n) \leq c * g(n)$$

$$n + 10 \leq c * n$$

We know that n is an integer that satisfies $n \geq \eta_0 \geq 1$. Thus, to begin with we take $\boldsymbol{n = 1}$. With that, we need to prove –

$$11 \leq c$$

Therefore, we take the value of $\boldsymbol{c = 11, \eta_0 = 1}$. Therefore, for 2 functions $f(n)$ and $g(n)$ we can write $\boldsymbol{f(n) \leq c * g(n)}$ for all integers $n \geq \eta_0$ such that the constants $c = 11$ and $\eta_0 \geq 1$. Thus,

$$\boldsymbol{n + 1 = O(n)}$$

NOTE

- Similar to the Big O, we can also prove the same way for the rest of the asymptotic notations.
- If 2 functions satisfy Small – O condition, then it will definitely satisfy the Big – O condition.
- If 2 functions satisfy both Big – O and Big – Ω, then it will satisfy Big – Θ.

Question

Assume $f(n) = n^2 + n + 10$ and $g(n) = n^2$. Prove that $f(n) = O\big(g(n)\big)$

Answer

To prove a Big O relation, we need to satisfy –

$$f(n) \leq c * g(n)$$

$$n^2 + n + 10 \leq c * n^2$$

Let us take $\boldsymbol{n = 1}$**.** Then,

$$1 + 1 + 10 \leq c * 1$$

$$c \geq 12$$

Therefore, we take the value of $\boldsymbol{c = 12, \eta_0 = 1}$ and for 2 functions $f(n)$ and $g(n)$ we can write $\boldsymbol{f(n) \leq c * g(n)}$ for all integers $n \geq \eta_0$ such that the constants $c = 12$ and $\eta_0 \geq 1$. Thus,

$$\boldsymbol{n^2 + n + 1 = O(n^2)}$$

Question

Assume $f(n) = n^2$ and $g(n) = n$. Prove that $f(n) = O(g(n))$

Answer

To prove a Big O relation, we need to satisfy –

$$f(n) \leq c * g(n)$$

$$n^2 \leq c * n$$

We know that n is a positive integer and for no such value of n that above equation will be satisfied. Therefore, we **can't prove Big – O dependency.** However, we can prove **Big – Omega dependency.**

For Big – Omega, we have –

$$f(n) \geq c * g(n)$$

$$n^2 \geq c * n$$

We can see that this will be true for all values of n when $c = 1$. Therefore,

$$\boldsymbol{n^2 = \Omega(n)}$$

Question

Assume $f(n) = n - 10$ and $g(n) = n + 10$. Prove that $f(n) = O(g(n))$

Answer

To prove a Big O relation, we need to satisfy –

$$f(n) \leq c * g(n)$$

$$n - 10 \leq c * (n + 10)$$

We know from common sense that $n + 10$ will always be greater than $n - 10$ for all integers greater than or equal to 1. Therefore, we can take $\boldsymbol{c = 1}$ and $\boldsymbol{n = 1}$**.**

Therefore, we take the value of $\boldsymbol{c = 1, \eta_0 = 1}$ and for 2 functions $f(n)$ and $g(n)$ we can write $\boldsymbol{f(n) \leq c * g(n)}$ for all integers $n \geq \eta_0$ such that the constants $c = 1$ and $\eta_0 \geq 1$. Thus,

$$\boldsymbol{n - 10 = O(n + 10)}$$

Additionally, if we take $\boldsymbol{c = \frac{1}{2}}$**,** then we can see –

$$n - 10 \geq \frac{1}{2}(n + 10)$$

Therefore,

$$\boldsymbol{n - 10 = \Omega(n + 10)}$$

Since both the Big – O and Big – Ω are being satisfied, Big – Θ will also be satisfied. Therefore,

$$\boldsymbol{n - 10 = \Theta(n + 10)}$$

GENERAL NOTES

Suppose the function $f(n)$ has the max power of n as $\boldsymbol{K}$**.** Then,

$$f(n) = O(n^m)$$

The above equation will only be satisfied if $\boldsymbol{m \geq K}$**.** This condition is called the **Upper Bound** condition of Big – O. Suppose if $\boldsymbol{m = K}$**,** then it is termed as **Tightly Upper Bound** condition. For example, let us take $f(n) = 2n^3 + n$. Then,

$$f(n) \neq O(n)$$

$$f(n) \neq O(n^2)$$

$$f(n) = O(n^3) \qquad Tightly\ Upper\ Bound$$

$$f(n) = O(n^{1000}) \qquad Upper\ Bound$$

We know that Small – O will only be satisfied if $\boldsymbol{m > K}$**.** Therefore, there is **no tightest upper bound** in the Small – O notation.

Suppose the function $f(n)$ has the max power of n as $\boldsymbol{K}$**.** Then,

$$f(n) = \Omega(n^m)$$

The above equation will only be satisfied if $\boldsymbol{m \leq K}$**.** This condition is called the **Lower Bound** condition of Big – O. Suppose if $\boldsymbol{m = K}$**,** then it is termed as **Tightly Lower Bound** condition. For example, let us take $f(n) = 2n^3 + n$. Then,

$$f(n) = O(n)$$

$$f(n) = O(n^2) \qquad Lower\ Bound$$

$$f(n) = O(n^3) \qquad Tightly\ Lower\ Bound$$

$$f(n) \neq O(n^{1000})$$

We know that Small – ω will only be satisfied if $\boldsymbol{m < K}$**.** Therefore, there is **no tightest lower bound** in the Small – ω notation.

When both the Big – O and Big – Ω conditions are being satisfied, then we can say that the Big – Θ condition is also satisfied. Hence, we can say that Big – Θ will only be satisfied if $\boldsymbol{m = K}$ and therefore the condition will always be **both Tightly Upper Bound and Tightly Lower Bound**.

For example, let us take the example below –

```
void main()
{
    x = y + z   //Statement 1
    for (i=0; i<n; i++) //Statement 2
        x = y + z
    for (i=0; i<n**2; i++)  //Statement 3
        x = y + z
}
```

In this case,

$$Order\ of\ mag\ of\ Statement\ 1 = 1$$

$$Order\ of\ mag\ of\ Statement\ 2 = n$$

$$Order\ of\ mag\ of\ Statement\ 3 = n^2$$

Therefore, we get –

$$f(n) = 1 + n + n^2$$

Hence, we can write

$$f(n) = O(n^2) = O(n^{10}) = \Omega(1) = \Omega(n^2) = \Theta(n^2) = o(n^3) = \omega(n)$$

COMPLEXITY CLASSES

- Constant Complexity – $O(1)$
- Logarithmic Complexity – $O(\log(n))$
- Polynomial Complexity – $O(n^c)$ where c is a constant and $c > 0$
    - Linear Complexity – $O(n)$
    - Quadratic Complexity – $O(n^2)$
    - Cubic Complexity – $O(n^3)$
- Decreasing Complexity – $O\left(\frac{1}{n}\right)$
- Exponential Complexity – $O(c^n)$ where c is a constant and $c > 1$

$$Decreasing < Constant < Log < Linear < Quad < Cubic < Exponential$$

DIVIDE AND CONQUER

To understand divided and conquer, we need to explore basic concepts –

- Recursion
- Recurrence Relation
- Recurrence Relation Solving
    - Substitution Method
    - Recursive Tree Method
    - Master Theorem

RECURSION

A function calling itself to solve a problem. The recursion is done using a stack data structure. It does need a termination condition otherwise it will continue indefinitely till we get stack overflow error (runtime error). The number of stack units used to run a recursion program is called the **depth of the stack.**

A recursive program will have a non-recursive counterpart as well. The time complexity of both recursive and non-recursive programs will **be the same** however the stack space complexity of recursive program will be **greater than** non-recursive program as the recursive program will use more stack units.

RECURRENCE RELATION

Recurrence relation is a mathematical notation of the recursive program. Let us take the example of recursive program to calculate the factorial of a number.

```
int f(int n)
{
    if (n <= 1) {
        return (1);
    }
    else {
        return (n * f(n-1));
    }
}
```

To write this as a recurrence relation, we can write as follows –

$$f(n) = \begin{cases} 1, & n \leq 1 \\ n * f(n-1), & n > 1 \end{cases}$$

The time complexity of the program can be also written as follows –

$$T(n) = \begin{cases} O(1) & n \leq 1 \\ T(n-1) + c & n > 1 \end{cases}$$

Here, c is a constant as there is no loop in the function. We can see that $f(n)$ is the recurrence relation for value while $T(n)$ is a recurrence relation for time complexity. Therefore, **one recursive program can have multiple recurrence relations.**

Question

Find the time recurrence relation.

```
int A(int n)
{
    if (n <= 10) {
        return (n**3);
    }
    else {
        return (A(n/2) * A(n/2) * A(n/2));
    }
}
```

Answer

$$T(n) = \begin{cases} O(1) & n \leq 10 \\ 3T\left(\frac{n}{2}\right) + c & n > 10 \end{cases}$$

Question

Find the time recurrence relation for the given program.

```
int A(int n)
{
    if (n <= 10) {
        return (n**10);
    }
    else {
        for (i=1 ; i<=4 ; i++) {
            A(n/2);
        }
        for (i=1 ; i<=n**2 ; i++) {
            x = y + z;
        }
    }
}
```

Answer

$$T(n) = \begin{cases} O(1) & n \leq 10 \\ 4T\left(\frac{n}{2}\right) + n^2 & n > 10 \end{cases}$$

Here, the loop with $A\left(\frac{n}{2}\right)$ is executed 4 times. Thus, we get the time complexity of $4T\left(\frac{n}{2}\right)$. In addition to that, the other for loop is also running for a total of n^2 times. Thus, in the else part we get the total time relation as –

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

RECURRENCE RELATION SOLVING

When we have a time recurrence relation, we can solve the relation to get the time complexity. Additionally, if we solve the value recurrence relation, we will get the value of the function. To solve, we have 3 methods as discussed below.

Substitution Method

Let us assume a recurrence relation as follows –

$$T(n) = \begin{cases} 1 & n = 1 \\ T(n-1) + n & n > 1 \end{cases}$$

Using the above relation, we can substitute n values and get the following series –

$$T(1) = 1$$

$$T(2) = T(1) + 2 = 1 + 2$$

$$T(3) = T(2) + 3 = 1 + 2 + 3$$

Similarly, we can write –

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

Now we have a proper function for $T(n)$. If it was a time recurrence relation, then we can write –

$$\boldsymbol{TC = O(n^2)}$$

Question

Solve the following recurrence relation –

$$T(n) = \begin{cases} 1 & n = 1 \\ T(n-1) * n & n > 1 \end{cases}$$

Answer

We can see that –

$$T(1) = 1$$

$$T(2) = T(1) * 2 = 1 * 2$$

$$T(3) = T(2) * 3 = 1 * 2 * 3$$

Similarly, we can write –

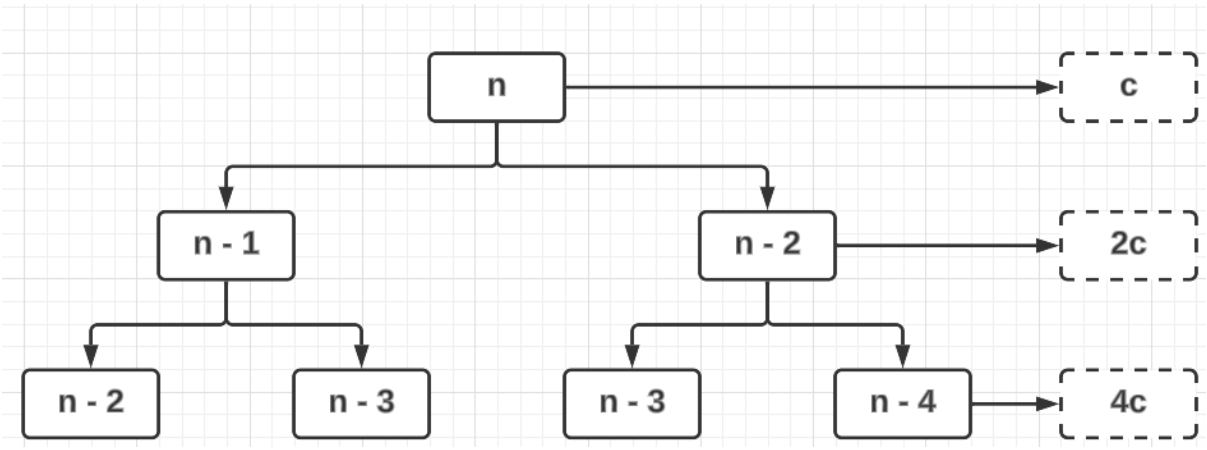
$$T(n) = 1 * 2 * 3 * \ldots * n = \boldsymbol{n!}$$

Recursion Tree Method

We solve the recurrence relation by forming a tree. For example, let us take the recurrence relation –

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + T(n-2) + c & n > 1 \end{cases}$$

We can draw a graph as follows –



This is just for 2 levels. When we extend the graph to $k1\ and\ k2$ levels in LHS and RHS, we get –

$$LHS\ term = n - k1$$

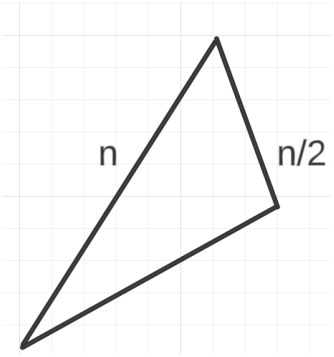
$$RHS\ term = n - 2k2$$

Now, we substitute $k1 = k2 = n$. Thus, we get –

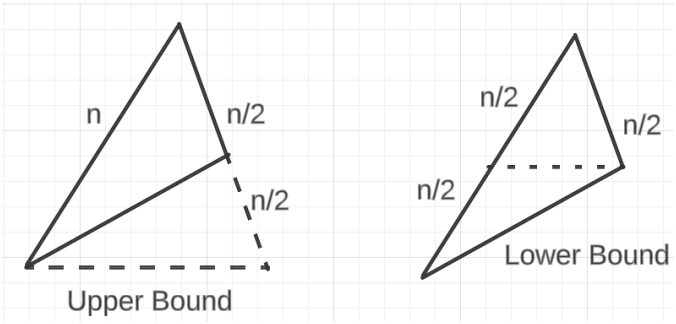
$$Height\ of\ LHS\ tree = n$$

$$Height\ of\ RHS\ tree = \frac{n}{2}$$

Hence, the overall structure of the graph will be something as follows –



This is difficult to get the exact value. So, we can either do **upper bound** or **lower bound** approximation.



For Upper bound, we get –

$$T(n) \leq c + 2c + 4c + 8c + \cdots + 2^n c$$

$$\boldsymbol{T(n) = O(2^n)}$$

For Lower bound, we get –

$$T(n) \geq c + 2c + 4c + 8c + \cdots + 2^{n/2} c$$

$$\boldsymbol{T(n) = \Omega\left(2^{n/2}\right)}$$

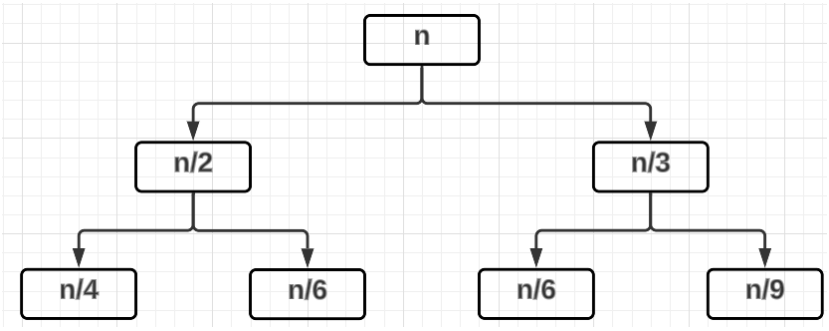
Question

Solve the recurrence relation –

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

Answer

We can draw the graph as follows –



Suppose we take the tree to $k1\ and\ k2$ levels in LHS and RHS, then –

$$LHS\ term = \frac{n}{2^{k1}}$$

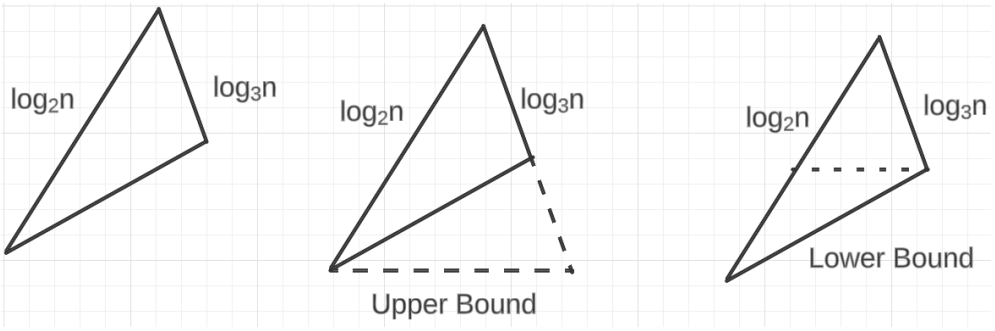
$$RHS\ term = \frac{n}{3^{k2}}$$

Since they haven't explicitly mentioned any termination condition, we can assume a termination condition to solve this. So, let us assume that $T(1)$ is the termination condition. Therefore,

$$k1 = \log_2 n$$

$$k2 = \log_3 n$$

The graph now becomes –



For Upper Bound, we can write –

$$T(n) \leq c + 2c + 4c + 8c + \cdots + 2^{\log_2 n} c$$

$$\boldsymbol{T(n) = O\left(2^{\log_2 n}\right) = O(n)}$$

For Lower Bound, we can write –

$$T(n) \geq c + 2c + 4c + 8c + \cdots + 2^{\log_3 n} c$$

$$\boldsymbol{T(n) = \Omega\left(2^{\log_3 n}\right) = \Omega\left(n^{0.63}\right)}$$

Master Theorem

Assume that there is a recurrence relation in the following format –

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

Such that $a \geq 1, b > 1$ and $f(n)$ is a positive function. In this case, we can write 3 cases –

- If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for any constant $\epsilon > 0$, then $\boldsymbol{T(n) = \Theta\left(n^{\log_b a}\right)}$
- If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $\boldsymbol{T(n) = \Theta\left(n^{\log_b a} * \log n\right)}$
- If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for any constant $\epsilon > 0$ and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for a value $c < 1$, then $\boldsymbol{T(n) = \Theta\left(f(n)\right)}$

NOTE – Always check for Case 1 first. Then go for Case 2. If both fail, then go for case 3.

Question

Solve the following recurrence relation –

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

Answer

Here, $a = 8$; $b = 2$; $f(n) = n^2$. Now,

$$n^{\log_b a} = n^3$$

We can see that –

$$n^2 = O(n^3 - \epsilon)$$

$$f(n) = O\left(n^{\log_b a} - \epsilon\right)$$

Thus, as per the master theorem we can say that –

$$\boldsymbol{T(n) = \Theta\left(n^3\right)}$$

Now that we know how to solve recurrence relations, we can go for the actual **Divide and Conquer** algorithm –

1. First, we divide the bigger problem in sub – problems.
2. Then, we recursively find the solutions of the sub – problems.
3. Combine the sub – problem solutions to get the final solution.

Let us assume that the problem with complexity $T(n)$ is divided into a sub-problems each of complexity $T\left(\frac{n}{b}\right)$. Let us also assume that the divide and combine functions have a time complexity of $f(n)$. Then, we can write –

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

Therefore, we can solve the recurrence relation using the **Master Theorem.**

<u>Find Max and Min elements in an array using DAC</u>

```
#include <stdio.h>
int DACMaxMin(a, i, j)
{
    if (i == j) {
        max = min = a[i];
        return (max, min);
    }

    if (i == j-1) {
        if (a[i] > a[j]) {
            max = a[i];
            min = a[j];
        }
        else {
            max = a[j];
            min = a[i];
        }
        return (max, min);
    }
```

```
    else {
        mid = (i+j)/2;
        (max1, min1) = DACMaxMin(a, i, mid);
        (max2, min2) = DACMaxMin(a, mid+1, j);
        if (max1 > max2)
            max = max1;
        else
            min = min1;
        if (min1 > min2)
            min = min2;
        else
            min = min1;

        return (max, min);
    }
}
```

As we can see, we are recursively dividing the problem with complexity $T(n)$ to 2 problems of complexities $T\left(\frac{n}{2}\right)$ each. Hence, we can write the recurrence relation as follows –

$$T(n) = \begin{cases} 1 & 0 < n \leq 2 \\ 2T\left(\frac{n}{2}\right) + c & n > 2 \end{cases}$$

To solve this recurrence relation, we take –

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

$$T\left(\frac{n}{2}\right) = 2\left[2T\left(\frac{n}{4}\right) + c\right] + c = 2^2T\left(\frac{n}{2^2}\right) + 2^1c + 2^0c$$

$$T\left(\frac{n}{2^2}\right) = 2^3T\left(\frac{n}{2^3}\right) + 2^2c + 2^1c + 2^0c$$

Therefore, if we go for k levels, we can get –

$$T(n) = 2^kT\left(\frac{n}{2^k}\right) + c[1 + 2^1 + 2^2 + \cdots + 2^{k-1}]$$

$$T(n) = 2^kT\left(\frac{n}{2^k}\right) + c\left[\frac{2^k - 1}{2 - 1}\right]$$

Here, we take –

$$\frac{n}{2^k} = 2$$

$$k = \log_2 n - 1$$

Thus, we get –

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

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

$$\boldsymbol{T(n) = O(n)}$$

We would have gotten the same relation with the help of the **Master Theorem** as well. We can write –

- Time complexity = $O(n)$
- Space complexity = $O(\log n)$
- No of comparisons = $1.5n - 2$

If we had not used DAC, we would have the no of comparisons as –

- Best case = $n - 1$
- Worst case = $2(n - 1)$
- Average case = $1.5(n - 1)$

Question

Suppose we have an array with 100 unique elements. If we try to find the max and min elements, what is the difference of the maximum and minimum number of comparisons required?

Answer

Since the algorithm is not mentioned, we need to check the no of comparisons for both DAC and non-DAC programs. For DAC, we have –

$$No\ of\ comparisons\ for\ all\ cases = 1.5n - 2 = 148$$

Without DAC, we have –

- $Best\ case = n - 1 = 99$
- $Worst\ case = 2(n - 1) = 198$
- $Average\ case = 1.5(n - 1) = 148.5$

Therefore, we can have –

$$Max\ comparisons - Min\ comparisons = 198 - 99 = \mathbf{99}$$

Question

Find the minimum time complexity required to find any element that is neither max nor min in an array with all distinct elements.

Answer

Suppose we have an array as follows –

$$A = [a, b, c, d, e, f, g, h, i, j]$$

Now, let us take the first three elements as follows –

$$A1 = [a, b, c]$$

Now, we sort this array as follows –

$$A1 = [c, b, a]$$

So, we can see that $c < b < a$. Therefore, b is neither max nor min. Therefore, $\boldsymbol{b}$ **is a correct response.** Thus, we perform the following steps –

- Take the 1st three elements
- Sort the three elements
- Return the middle element

Since we are taking only 3 elements, the time complexity will be $\boldsymbol{O(1)}$ **for every case.**

Find the exponent of two numbers

Using DAC, we can write the program as follows –

```
int DACpow(int a, int n)
{
    if (n == 1) {
        return (a);
    }
    else {
        int mid = n/2;
        int b = DACpow(a, mid);
        int c = b * b;
        return (c);
    }
}
```

The recurrence relation can be written as –

$$T(n) = \begin{cases} O(1) & if\ n = 1 \\ O(1) + T\left(\frac{n}{2}\right) + O(1) & if\ n > 1 \end{cases}$$

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

Using Master Theorem, we see that $a = 1$; $b = 2$. Thus,

$$n^{\log_b a} = n^{\log_2 1} = 1$$

Thus, we can get –

$$f(n) = c = 1 * (\log_2 n)^0$$

Therefore, we get –

$$\boldsymbol{T(n) = \Theta(\log n) = O(1) + c \log n}$$

We can also notice that we will have a total of $\boldsymbol{\log n}$ number of multiplications.

Binary Search

Using DAC, we can write the code as –

```
int BinSearch(int a[], int i, int j, int x)
{
    if (i == j) {
        if (a[i] == x) {
            return (i);
        }
        return (-1);
    }
    else {
        int mid = (i+j)/2;
        if (a[mid] == x) {
            return (mid);
        }
        if (a[mid] < x) {
            BinSearch(a, mid+1, j, x);
        }
        else {
            BinSearch(a, i, mid-1, x);
        }
    }
}
```

The following program will have a recurrence relation as follows –

$$T(n) = \begin{cases} O(1) & n = 1 \\ c + T\left(\frac{n}{2}\right) & n > 1 \end{cases}$$

As we can see, here we are dividing the problem into smaller problems and we also conquering as we get the final output. However, there is no combine step. Thus, for DAC a combine step is not necessary all the time. We can solve the recurrence relation as follows –

$$T(n) = T\left(\frac{n}{2}\right) + c = T\left(\frac{n}{4}\right) + c + c = T\left(\frac{n}{8}\right) + c + c + c = \cdots$$

Therefore, we get –

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

Equating $k = \log_2 n$, we get –

$$T(n) = T(1) + c * \log_2 n = 1 + c \log_2 n$$

Therefore, we get –

$$\boldsymbol{T(n) = O(\log_2 n) = \Omega(1)}$$

We can conclude that for best case, the time and space complexity will be $O(1)$ while the average and worst case time and space complexity will be $O(\log_2 n)$.

NOTE

Binary search is an algorithm that can be used to search for an element only in an array which has **distinct values** and are arranged in a **sorted fashion.** In other cases, it is best to use Linear Search.

SUBTRACT AND CONQUER

Suppose we have a recurrence relation as follows –

$$T(n) = \begin{cases} c, & n \leq 1 \\ aT(n-b) + f(n), & n > 1 \end{cases}$$

If we have $f(n) = O(n^k)$, then we can have 3 cases –

- If $a < 1$, then $T(n) = O(n^k)$
- If $a = 1$, then $T(n) = O(n^{k+1})$
- If $a > 1$, then $T(n) = O\left(n^k a^{\frac{n}{b}}\right)$

Question

Q Consider the following recurrence relation

$T(1) = 1$

$T(n + 1) = T(n) + \lfloor \sqrt{n+1} \rfloor$ for all $n \geq 1$

The value of $T(m^2)$ for $m \geq 1$ is **(Gate-2003) (2 Marks)**

(A) $(m/6)\ (21m - 39) + 4$

(B) $(m/6)\ (4m^2 - 3m + 5)$

(C) $(m/2)\ (m^{2.5} - 11m + 20) - 5$

(D) $(m/6)\ (5m^3 - 34m^2 + 137m - 104) + (5/6)$

Answer

From the definition, we can get –

$$T(n) = T(n-1) + floor(\sqrt{n})$$

Using subtract and conquer, we can see that –

$$a = 1\ ;b = 1\ ;f(n) = O(n^{0.5})\ ;k = 0.5$$

Thus, we can see that the Case 2 is satisfied. Therefore,

$$T(n) = O(n^{1.5})$$

Thus, we get –

$$T(m^2) = O(n^3)$$

Hence, **Option B** is the correct answer.

SORTING TECHNIQUES

Sorting techniques are used to sort the elements of a data structure in either ascending or descending order. These are mainly divided into two types –

- **Internal –** The sorting techniques that don't require any additional memory. For example, Heap sort.
- **External –** The sorting techniques that don't require any additional memory. For example, Merge sort.

During sorting, there can be a case where an array contains duplicate values. Thus, there is an order in which these redundant values appear in the original array. If the sorted array is able to preserve this order of duplicate values, then it is termed as **stable.** Else, it is termed as **unstable.** Bubble sort and Insertion sort are Stable and Unstable sorting techniques respectively.

NOTE

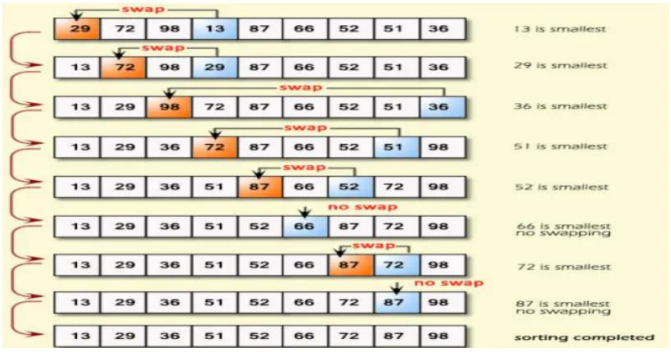
Every sorting technique uses a swapping function. Hence, we can just define a swapping function as follows –

```c
void Swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

SELECTION SORT

1. Traverse the array from Index 0 to Index $(n-1)$.
2. Find the min element for ascending or the max element for descending.
3. Swap the min and $arr[0]$
4. Repeat steps 1-3 but this time from index 1 to $(n-1)$

For example,



There is a **single swap in each iteration.** We can write the code for the same as follows –

```c
int main()
{
    int a[10] = {15, 46, 56, 32, 10, 7, 54, 100, 10, 44};
    int i,j,loc;
    int min;
    for (i=0 ; i<10 ; i++)
    {
        min = a[i];
        loc = i;
        for (j=i+1 ; j<10 ; j++)
        {
            if (a[j] < min) {
                min = a[j];
                loc = j;
            }
        }

        Swap(&a[i], &a[loc]);
    }
}
```

Every array parse will result in 1 swap. Since there is a nested loop, the **Worst Case TC will be $O(n^2)$.** Let us assume that the array being fed as the input is already sorted aka **Best Case.** However, Selection sort will run both the loops but will perform **no swapping.** Hence, in this case also the **TC will be $O(n^2)$.** As there is no additional space required, this is an **internal sorting algorithm.** Finally, we can see that if there are duplicate values, then their order is not preserved and hence the algorithm is **unstable.**

As this algorithm is sorting 1 element at a time, this algorithm approach is an example of **Subtract and Conquer.**

BUBBLE/SHELL/SINKING SORT

In this case, we check between neighbor elements. If the right neighbor is less than the left neighbor, then the two elements are swapped. In Selection sort, the minimum element was swapped to the first place after first iteration, but in Bubble sort the max element will be shifted to the last position after first iteration. We can write the code for the same –

```
int main()
{
    int a[10] = {15, 46, 56, 32, 10, 7, 54, 100, 10, 44};
    int i,j;
    int n = 10;
    for (i=0; i<n; i++) {
        for (j=0 ; j<n-i-1; j++) {
            if (a[j] > a[j+1]) {
                Swap(&a[j], &a[j+1]);
            }
        }
    }
}
```

In worst case, the array being fed will be in descending order and will be asked to be sorted in an ascending order or vice-versa. In this case, the **number of swaps** will be $(n-1)$ **per iteration.** Hence, the **worst case scenario has** $TC = O(n^2)$**.** One thing to note here is that in best case, there will be no swapping. However, the algorithm will traverse the loop again and again. Thus, the **best case scenario has** $TC = O(n^2)$**.** This algorithm also is **stable, internal and uses Subtract and Conquer.**

From what we just observed, the best case scenario will be when the input array is already sorted. In that case as well the algorithm is iterating over and over again even though it is not swapping. Hence, **if in an iteration there is no swapping, it would mean that the array is sorted.** Therefore, we can have a condition as follows –

```
int main()
{
    int a[10] = {15, 46, 56, 32, 10, 7, 54, 100, 10, 44};
    int i,j;
    int n = 10;
    int flag;
    for (i=0; i<n; i++) {
        for (j=0 ; j<n-i-1; j++) {
            if (a[j] > a[j+1]) {
                Swap(&a[j], &a[j+1]);
                flag = 1;
            }
        }

        if (!flag) {
            break;
        }
    }
}
```

In this case, if the array is already sorted, there will be no swaps and after the first iteration, the value of flag will remain 0. Hence, the loop will break and the program will terminate. Thus, in this case the **Best Case TC becomes** $O(n)$**.**

INSERTION SORT

In this case, the algorithm will take each element and insert it (hence the name) in its appropriate location. For example, let us take the following array –

$$A = [6,5,3,1,8,7,2,4]$$

First, it checks 6. As there is no element before it, 6 is in its sorted position. Now, we move on the next element which is 5. We can see that $5 < 6$. Therefore, the algorithm moves 5 to the front of 6.

$$A = [5,6,3,1,8,7,2,4]$$

Now, we reach element 3. We can see that $3 < 5 < 6$. Therefore, the algorithm will push 3 to the front of 5 as well.

$$A = [3,6,5,1,8,7,2,4]$$

Similarly, the next few steps will be as follows –

$$A = [1,3,6,5,8,7,2,4]$$

$$A = [1,3,6,5,8,7,2,4]$$

$$A = [1,3,6,5,7,8,2,4]$$

$$A = [1,2,3,6,5,7,8,4]$$

$$A = [1,2,3,4,6,5,7,8]$$

Therefore, we can see that the entire array is sorted after a **single iteration.** The code can be written as follows –

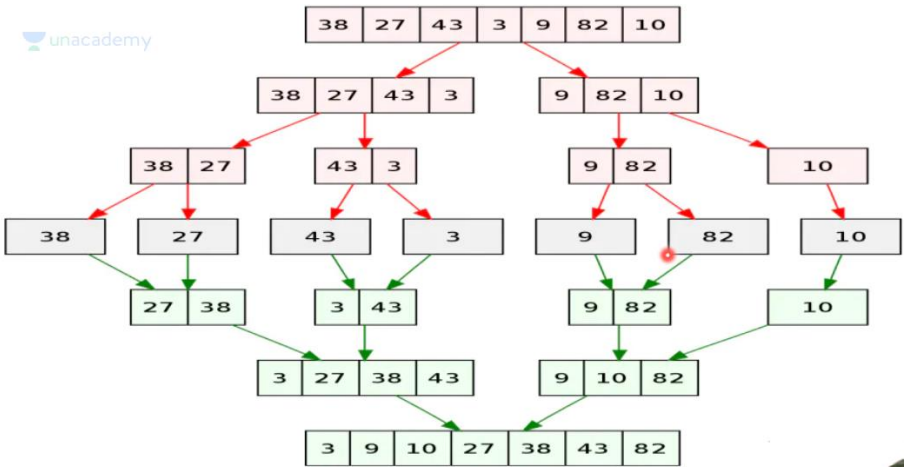
```
int main()
{
    int a[10] = {15, 46, 56, 32, 10, 7, 54, 100, 10, 44};
    int i,j;
    int n = 10;
    for (i=1; i<n; i++) {
        int elt = a[i];
        int j = i-1;
        while (j>=0 && elt<a[j]) {
            a[j+1] = a[j];
            j--;
        }
        a[j+1] = elt;
    }
}
```

This is also the first sorting technique where we haven't used a Swap() function. This algorithm has –

- Internal
- Stable
- Worst case time complexity = $O(n^2)$
- Best case TC = $O(n)$ as the inner while loop will never run.
- Follows a Subtract and Conquer approach

MERGE SORT

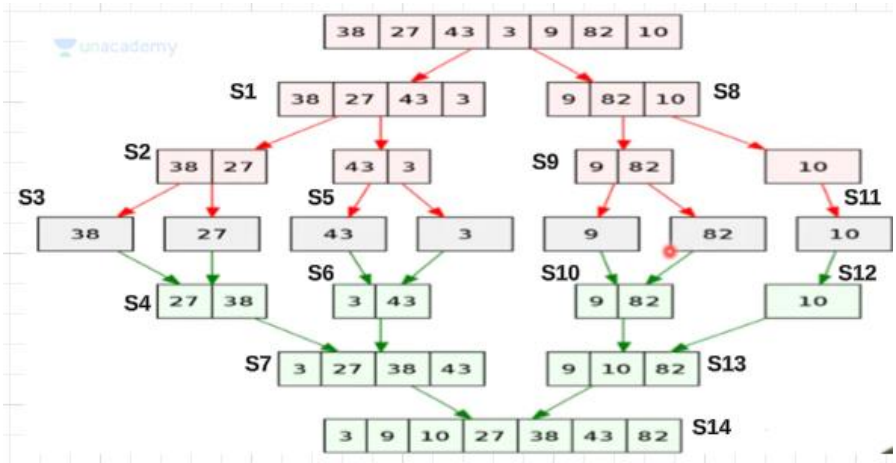
This is the first algorithm we will learn where we are using a **Divide and Conquer** approach. In this case, we first divide an array of length n in half over and over again till we have n arrays of 1 element each. Since these array have 1 element, they will be sorted. Now, we start merging these n array in the correct order to form the final sorted array.



We can write a code for Merge Sort as follows –

```
void MergeSort(int *a, int p, int r) {
    if (p<r) {
        int q = (p+r)/2;
        MergeSort(a, p, q);
        MergeSort(a, q+1, r);
        Merge(a, p, q, r);
    }
}
```

We can also label the order in which these processes will occur –



In the MergeSort function, we have denoted –

- p – First element index
- q – Mid element index
- r – Last element index

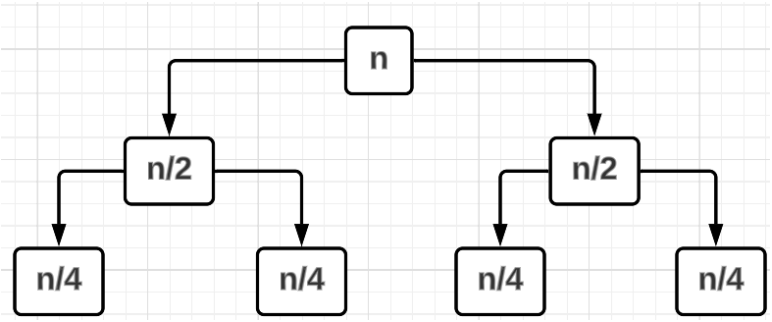
First, we check if $(p < r)$. If this condition is true, it would mean that there are more than 1 elements in the array and hence, we need to split it. Thus, it calculates the middle index aka q and then calls MergeSort recursively twice – one for left child and one for the right child.

Once the splitting is done and the MergeSort functions are complete, we need to perform the Merge function wherein we need to merge the two arrays into a single sorted array. The Merge code can be given as follows –

```
void Merge(int *a, int p, int q, int r) {
    int n1 = q-p+1 //Length of the left child
    int n2 = r-q //Lenght of right child
    //Now we create 2 arrays
    int L[n1+1];
    int R[n2+1];

    //Insert the elements in L[] and R[]
    for (i=0 ; i<n1 ; i++) {
        L[i] = a[p+i-1]
    }

    for (i=0 ; i<n2 ; i++) {
        R[i] = a[i+q]
    }

    //Set the last element as infinity to indicate array end
    L[n1] = INFINITY;
    R[n2] = INFINITY;
    //Now we do the merging
    i = 1;
    int j = 1;
    int k;

    for (k=p ; k<=r ; k++) {
        if (L[i] <= R[j]) {
            a[k] = L[i];
            i++;
        }
        else {
            a[k] = R[j];
            j++;
        }
    }
}
```

There is a reason we put the last element of the L and R arrays as infinity. Suppose the L array is shorter than the R array, then it will reach infinity and as a result, any element in R will be lesser than L and will get pushed into the final array.

Let us now perform Time Complexity analysis. We can see that there are 3 loops in the Merge() function but none of them are nested. Hence, the TC for Merge() will be $\boldsymbol{O(n)}$**.** Now, for MergeSort(), we need to see its functioning. It is basically dividing a n-sized problem to two $\frac{n}{2}$ – sized problems. Thus, we get –



Let us assume that the height of the graph is k levels. Thus, we get –

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

$$k = \log_2 n$$

Thus, the splitting of the array takes $O(\log n)$ time and the merge takes $O(n)$ time. Therefore, the **TC for merge sort becomes $\boldsymbol{O(n \log n)}$.**

In Best case, the input array is already sorted while in the worst case, the input array is sorted in the reverse order. In both these cases, the merge sort will perform the splitting and merging. Thus, the TC for best and worst case is the same.

Since we are using additional arrays to perform the sorting, this is an **external sorting technique.** Since it is taking 2 additional arrays of size related to $n/2$, we can say that **Merge Sort has a Space Complexity of $\boldsymbol{O(n)}$.** Additionally, there is no swapping between elements and hence duplicate elements will preserve their order. Thus, this technique is also **stable.**

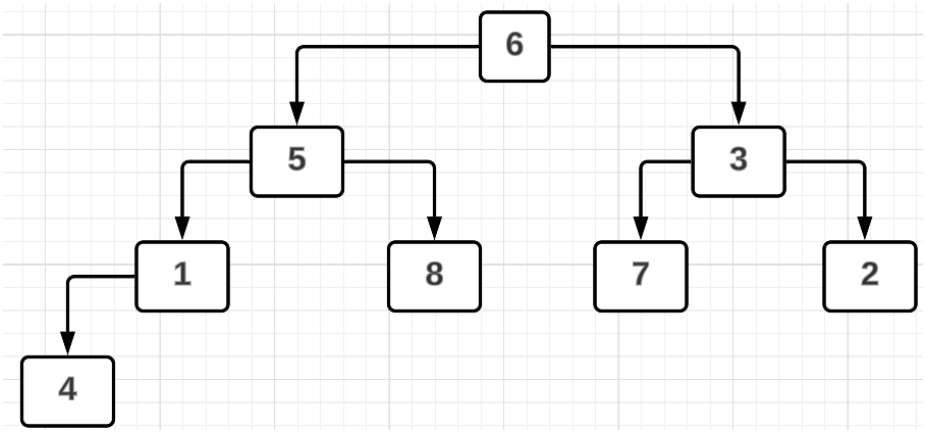
HEAP SORT

To learn about heap sort, we need to refresh our concept of Heaps. Basically, a heap is a **complete binary tree.** A CBT is a binary tree which has all nodes filled at all levels and each level is filled from left – to – right. A CBT may not be filled fully at the last level.

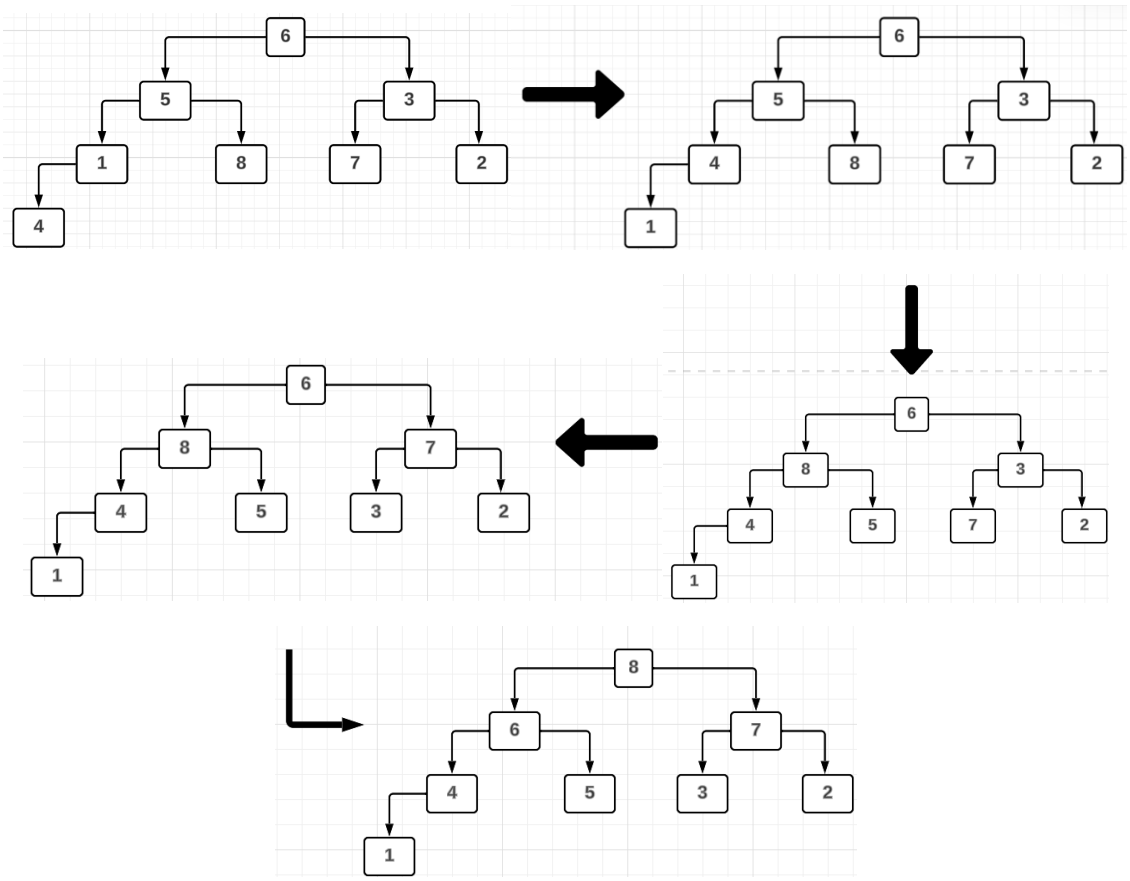
A heap where the parent is greater than the children is called a **Maxheap** and is the one we shall be using in this sorting technique. For example, let us take the array below –

$$A = \{6,5,3,1,8,7,2,4\}$$

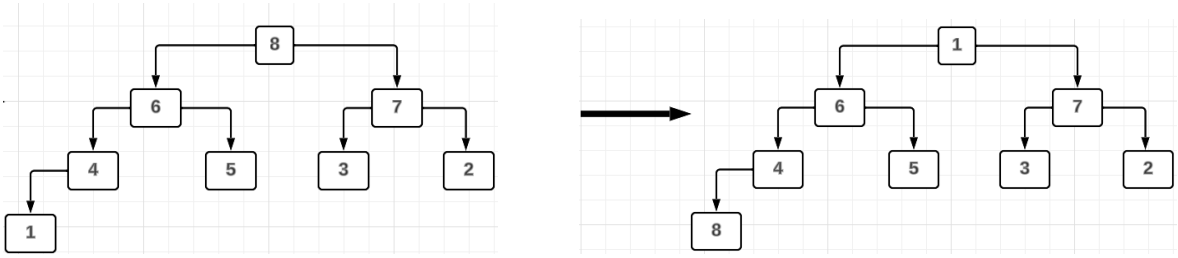
In the first step, we will be building a Heap aka CBT with the elements as follows –



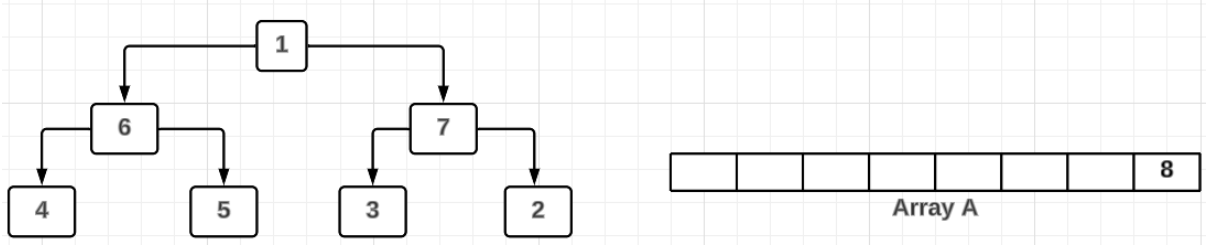
We have created the CBT by filling nodes in a top-down and left-right fashion. Next, we will convert this CBT into a Max heap. To do this, we need to ensure the parent is larger than the children. In case that is not the case, we can swap the position of the children and the parent. For the given CBT, this process looks something like this –



Now, we have a Max heap where every parent is greater than its children. After this, we perform the last step of this iteration wherein we swap the root and the smallest child.



Now, we shall push the new child node to the last element of the array and delete that branch. Thus, in our example, we will have –



This is the end of the 1st iteration and we can see that the largest element has been appended to the last position. For the next iteration, we again will create a max heap, swap the root and child nodes and then append it to the end of the list.

The code for this sorting technique has 3 functions as shown below –

```
Max-Heapify(A, i)
{
        L ← Left[i]
        R ← Right[i]
        if( L <= Heap_size[A] and A[L] > A[i])
                Largest ← L
        Else
                Largest ← i
        if(R <= Heap_size[A] and A[r] > A[Largest])
                Largest ← R
        if(Largest != i)
        {
                Exchange( A[i] ←→A[Largest])
                Max-Heapify(A, Largest)
        }
}
```

```
Build_Max_Heap(A)
{
        Heap-size[A] ← length[A]
        for i ← ⌊ length[A]/2 ⌋ down to 1
        {
                do Max-Heapify (A, i)
        }
}
```

```
Heap_Sort(A)
{
        Build_Max_heap(A)
        for i ← length[A] down to 2
        {
                do exchange (A[1] ←→A[i])
                Heap-size[A] ← Heap-size[A] – 1
                Max-Heapify(A,1)
        }
}
```

As we can see, Max_Heapify() has all statements as O(1) complexity but is recursively calling itself. A CBT of n elements will have $\log n$ levels. Hence, it will recursively be called $\log n$ times. Therefore, the TC becomes $\boldsymbol{O(\log n)}$**.** Now, we can see that Build_Max_Heap() calls Max_Heapify() a total of $n/2$ times. Hence, its TC will be $\boldsymbol{O(n \log n)}$**.** Finally, we can see that the main function Heap_Sort() is calling Build_Max_Heap() and is also calling Max_Heapify() in a loop that runs n times. Thus,

$$TC\ of\ Heap\_Sort() = O(n \log n) + O(n \log n) = \boldsymbol{O(n \log n)}$$

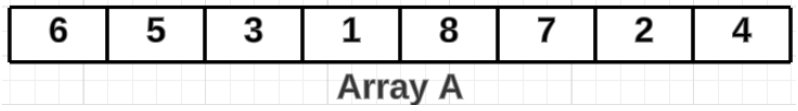
QUICK SORT

This is one of the most popular sorting techniques. Just like merge sort, this algorithm uses a **Divide and Conquer** approach wherein it divides an array into partitions. However, it doesn't always divide the array in half. Instead, Quick sort will decide on a pivot element and then it will divide based on the pivot element. The code for the same is given below –

```
Quick_Sort(A, p, r)
{
        if(p < r)
        {
                q ← partition (A, p, r)
                quick_Sort(A, p, q - 1)
                quick_Sort(A, q + 1, r)
        }
}
```

```
Partition (A, p, r)
{
        x ← A[r]
        i ← p – 1
        for j ← p to r – 1
        {
                if(A[j] <= x)
                {
                        i ← i + 1
                        Exchange( A[i] ←→A[j])
                }
        }
        Exchange( A[i + 1] ←→A[r])
        return i+1
}
```

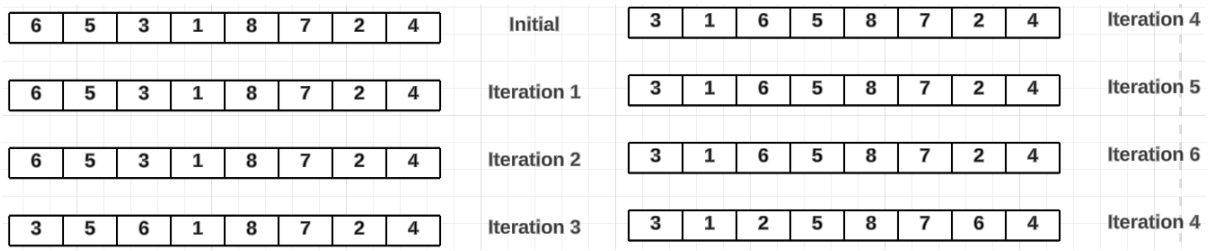
Suppose, let us consider the array as follows –



If we were using Merge Sort, we would have divided the array into two sub-arrays of equal length. However, in Quick sort, we use the Partition() function to get the pivot element. The Partition() function operates as follows –

- We first assume that the last element is the pivot element. Also, we assume $i = 0$
- Then we run a j loop from Index 0 to Index n-2
    - If the j-th element is less than the pivot, then we swap the j-th element and the element at index $i + 1$. Then, we increment i by 1
    - Else, we continue on in the j loop.
- Once the loop breaks, swap the last element and the $i + 1$ element.
- Finally, we return the $i + 1$ index

This is shown below –



Once the j loop breaks, we swap the last element and the $i + 1$ element to get the final array as –

As we can observe, we started with the last element aka **4** as the pivot element and now after 1 call of the Partition() function, we can see that 4 has appeared in the center. Now, we divide the array as –

$$L = \{3,1,2\}$$

$$R = \{4,8,7,6,5\}$$

Then the Quick sort is recursively called just like Merge Sort. In **best case** scenario, the array will always be divided into 2 equal parts and thus the TC becomes $\boldsymbol{O(n \log n)}$. However, in the worst case scenario, we are dividing the array into 2 arrays of size 1 and $(n-1)$. In short, the problem becomes **Subtract and Conquer** rather than Divide and Conquer. Therefore, in the **worst case,** then TC will become $\boldsymbol{O(n^2)}$**.**

CHEAT SHEET

Sorting Algorithm	Best Case	Worst Case
Selection	$O(n^2)$	$O(n^2)$
Bubble	$O(n^2)$ / $O(n)$	$O(n^2)$
Insertion	$O(n)$	$O(n^2)$
Merge	$O(n \log n)$	$O(n \log n)$
Heap	$O(n \log n)$	$O(n \log n)$
Quick	$O(n \log n)$	$O(n^2)$

GREEDY ALGORITHM

Let us take the problem where we have a sorted array and we need to find two elements whose sum is 1000. For example, let us take the array as follows –

$$L = [100, 200, 300, 400, 500, 600, 700, 800]$$

Thus, we can return either (200,800) or (300, 700) or (400, 600). To solve this problem, we can have three approaches.

Approach 1

Let us take the first element as $a = 100$. Now, we use Linear search to parse the rest of the array and check if the sum will equal to a 1000. Hence, for an array of size n, in the worst case we will be applying Linear Search for n times. Hence,

$$T(n) = O(n^2)$$

Approach 2

In this case, we are following the same procedure as Approach 1 but we will use Binary search instead of Linear search. Thus, we will be applying Binary search n times in the worst case scenario. Thus,

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

Approach 3

Let us take the first element as $a = 100$ and the last element as $b = 800$. Now, we add them.

Since $a + b < 1000$, we will shift a to the next element making $a = 200$. Now, (200,800) is a solution and so the program breaks. However, if we had a case where $a + b > 1000$, then we move b down to the next largest number.

For example, let us try to find the elements whose sum equals to 700. For first iteration, $a = 100$; $b = 800$. Since $a + b > 700$, we move b down to get $b = 700$. Still, $a + b = 800 > 700$. Thus, we move b further down to $b = 600$. Now, we get $a + b = 700$ and the program breaks out.

In short, for every case the program will parse the array just once. Thus, for the worst case scenario, we will have –

$$T(n) = O(n)$$

This is called the **Greedy algorithm** and is the best solution to the above problem.

Suppose we modify the above problem so that we need to find 3 elements such that $a + b + c = 1000$. Therefore, in this case we have –

- Approach 1 - $T(n) = O(n^3)$
- Approach 2 - $T(n) = O(n^2 \log n)$
- Approach 3 - $T(n) = O(n^2)$

In general, suppose we need to find $\boldsymbol{m}$ elements such that $a + b + c + \cdots m\ elements = K\ (const.)$, then we get –

- Approach 1 - $T(n) = O(n^m)$
- Approach 2 - $T(n) = O(n^{m-1} \log n)$
- Approach 3 - $T(n) = O(n^{m-1})$

A **greedy algorithm** is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum.

HUFFMAN CODING AND GRAPH

This is one of the problems that are solved using a Greedy Algorithm. In this case, let us take a case where we have 5 symbols that occur in the following frequencies in a string –

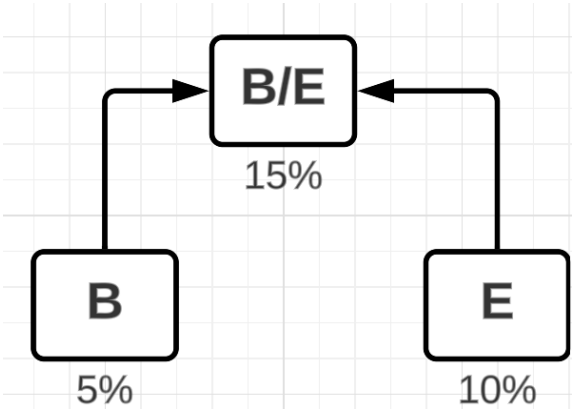
SYMBOL	FREQUENCY
A	35%
B	5%

C	20%
D	30%
E	10%

Let us assume a string of length 200 has arrived with these symbols. What is the minimum size required to store these symbols?

Traditionally, we would say that there are 5 symbols, and hence we need 3 bits to uniquely store them. Thus, the total size required would be $200 * 3 = \mathbf{600}$ **bits.** However, we can optimize this using the Greedy Algorithm.

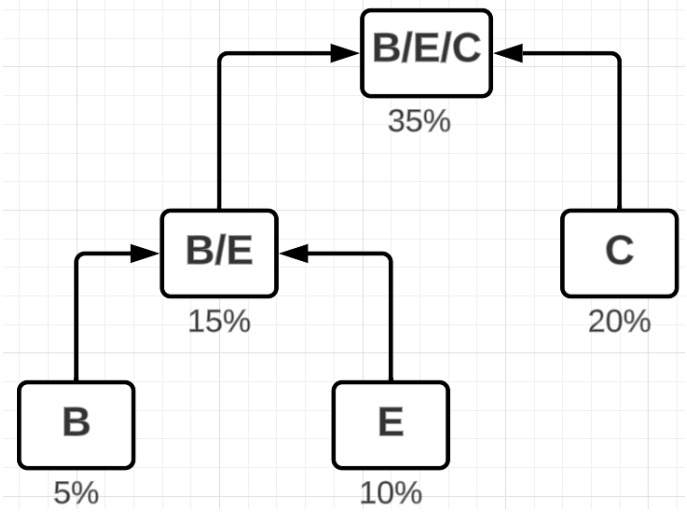
To do this, we create something called the **Huffman Tree.** First, we take the 2 least occurring symbols and make them as a node. Their addition will be the parent.



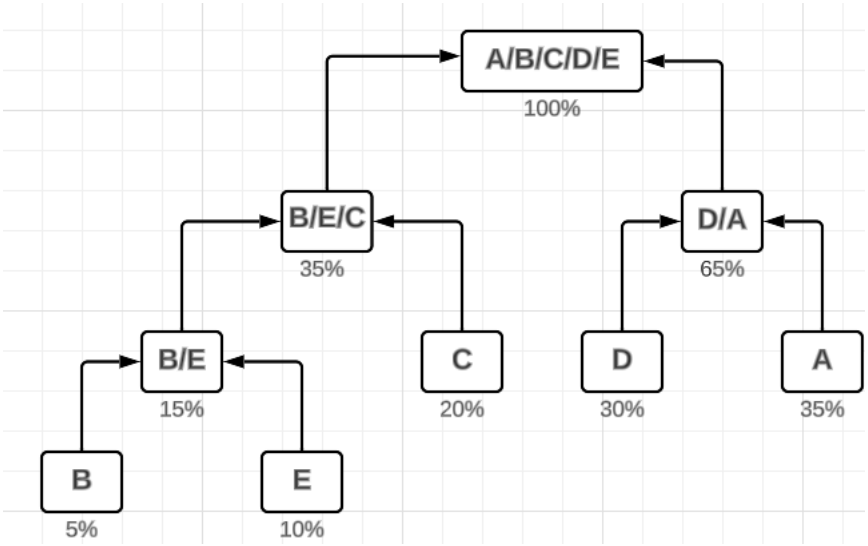
Note that the symbol with the lesser frequency is placed to the left. Now, the table becomes something like this –

SYMBOL	FREQUENCY
A	35%
B/E	15%
C	20%
D	30%

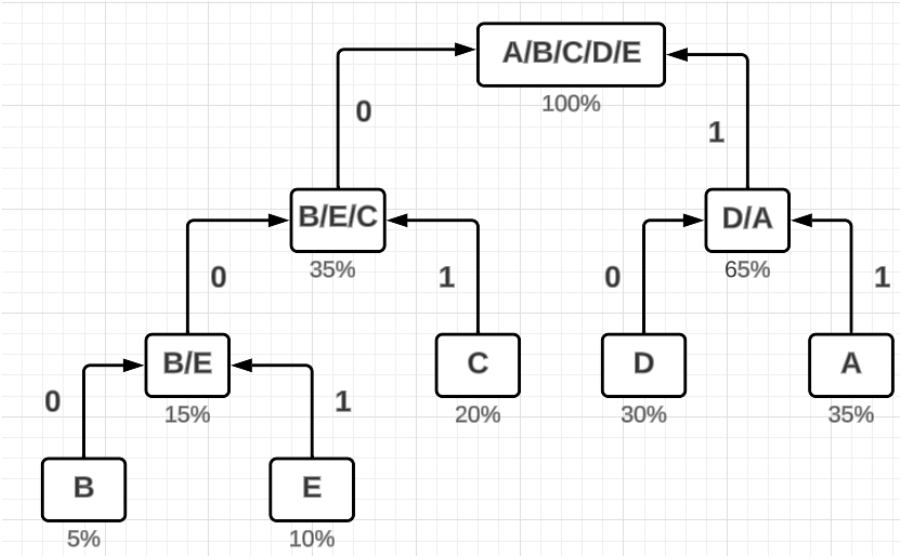
We once again repeat the process as shown below –



Continuing this process, we end up with the final graph as follows –



Finally, we will assign weights to the edges of this Huffman Graph. Basically, all left children get a 0 weight while the right children get a weight of 1. This gives us –



Using the graph, we can now see that the symbols are represented as –

SYMBOL	REPRESENTATION
A	11
B	000
C	01
D	10
E	001

NOTE

What we have done is quite clever here. Basically, we know that some symbols are used more frequently than others. So, we would like to keep them as simple as small as possible. Hence, A,C and D are represented using only 2 bits. The other non-frequent symbols like B and E are represented using 3 bits.

One more beautiful observation about this is that **none of the codes are prefix to other codes.** This means that there will be no confusion when it comes to which code represents which symbol.

Now, let us try to check for a string of 200 characters. We get –

$$Size = (2 * 70) + (3 * 10) + (2 * 40) + (2 * 60) + (3 * 20) = \mathbf{430\ bits}$$

As we can see, we now require just 430 bits to store the string rather than the conventional 600 bits. Since we were greedy with the frequency (Greedy Algorithm), we were able to reduce the storage required using **Huffman Graph and coding.** However, if the frequencies are not mentioned, then we can't use the Greedy Algorithm.

OPTIMAL MERGE PATTERN

This is another application of the Greedy Algorithm. Suppose, we have 2 sorted arrays of length m and n respectively. Then, if we try to merge these two arrays into a single sorted array, then we will have to perform $\boldsymbol{m + n}$ computations/record shifting and $\boldsymbol{m + n - 1}$ comparisons/value checks. With this thought in mind, let us extend the idea to multiple lists.

For example, let us assume we have 4 arrays as follows –

$$A = [1,2]$$

$$B = [10, 11,12]$$

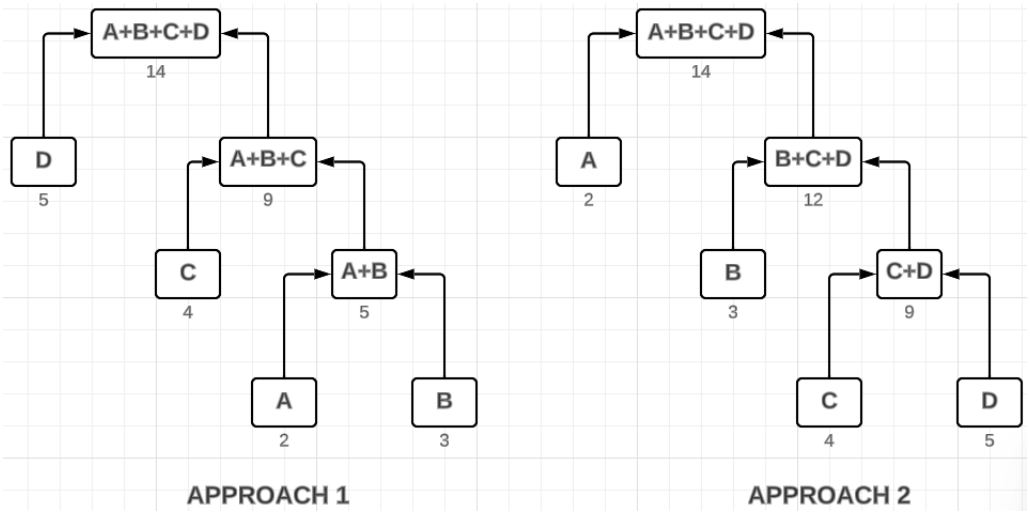
$$C = [20, 21, 22, 23]$$

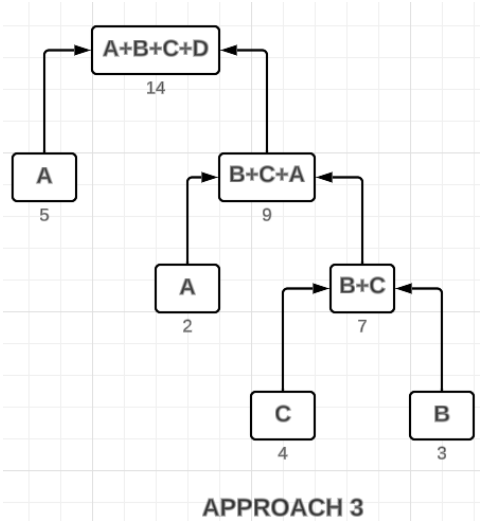
$$D = [30, 31, 32, 33, 34]$$

We merge these arrays in pairs. Thus, we have 3 main approaches –

- Merge the arrays in ascending order of their lengths
- Merge the arrays in descending order of their lengths
- Merge from the center

These approaches are shown below –





APPROACH 3

Using these approaches, we can conclude that –

APPROACH	NO OF COMPUTATIONS	NO OF COMPARISONS
1	28	25
2	35	32
3	30	27

Thus, we can see that Approach 1 is the best option. We are taking the smallest arrays, merging them. Then again taking the smallest and merging them. Hmmm…the tree we just made and this approach sounds EXACTLY LIKE THE HUFFMAN TREE. In short, the **Optimal Merging algorithm is a Greedy Algorithm.**

KNAPSACK PROBLEM

Till now, we have seen Greedy Algorithm being used in basic level problems. In both cases, we were very sure about the variable around which we needed to be Greedy. However, let us now consider a situation of a **Thief.** Assume there is a thief that is trying to steal three objects as listed below –

Object	O_1	O_2	O_3
Profit	25	24	15
Weight	18	15	10

The knapsack (hence the name) of the thief can carry a max of 22.5 kgs of weight. Assuming we can take fractions of the object i.e. we need not steal the entire object but can steal just a fraction of the object, how should the thief steal to increase his profit to the maximum possible level?

In short, here we need to use Greedy Algorithm to maximize profit. We can again use three approaches here –

- Approach 1 – Greedy based on Profit
- Approach 2 – Greedy based on Weight
- Approach 3 – Greedy based on Profit-to-Weight ratio

Approach 1

Since we are greedy for **profit** here, we will take Object O1 first as it has the highest profit. That would mean we have just 4.5 kgs of free space in the knapsack left. Now, we can target the next most profitable object which is Object O2. Since we are allowed to take fractions of the object, we take 4.5 kgs of O2. Therefore, we get –

$$Total\ Profit = (25 * 1) + \left(24 * \frac{4.5}{15}\right) = \mathbf{32.2}$$

Approach 2

Here, we are greedy for **weight.** That means, we need to first carry the lightest and then move on to the heavier objects. Thus, we first carry Object O3 which is the lightest. Then, we are left with 12.5 kgs of space. We would then take 12.5 kgs of next heavier object which is Object O2. Therefore, we get –

$$Total\ Profit = (15 * 1) + \left(24 * \frac{12.5}{15}\right) = \mathbf{35}$$

Approach 3

Here, we are greedy for **profit-to-weight ratio**.

Object	O_1	O_2	O_3
Profit	25	24	15
Weight	18	15	10
Profit/Weight	1.38	1.6	1.5
Solution			

Thus, we first take Object O2 and then take 7.5 kgs of Object O3. Therefore, we get –

$$Total\ Profit = (24 * 1) + \left(15 * \frac{7.5}{10}\right) = \mathbf{35.25}$$

In conclusion, we can see that being Greedy for Profit-to-Weight ratio is the most beneficial for thief. This knapsack problem was designed to indicate that just using Greedy algorithm is not sufficient. Rather, we need to know upon which variable we need to use the algorithm on.

NOTE

In the problem we discussed above, we can have fractions of the object as well. This is called a **Fractional Knapsack problem.** On the other hand, in case we have a problem where we can only take the whole object and not its fraction, then it is called a **0-1 Knapsack Problem.**

In case we consider the same problem as above but consider it to be a 0-1 Knapsack Problem, then –

$$Optimal\ algorithm\ (approach\ 1) = 25$$

$$Greedy\ Algo\ (apporach\ 3) = 24$$

Thus, the Greedy algorithm may not be the best choice here.

JOB SEQUENCING WITH DEADLINE

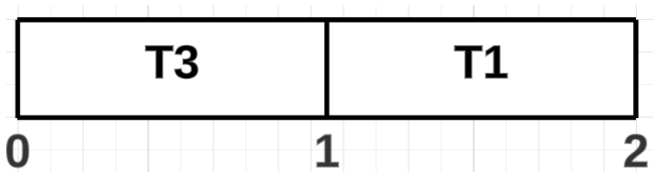
This is another avenue where we can apply the Greedy Algorithm. Suppose we have n processes in a CPU all of whom have Arrival Time as 0, Burst time as 1 and same priority. The CPU performs a form of non-preemptive scheduling. Each process has a certain deadline before which if the process is completed, it will result in a certain profit. How do we select a sub-set of these processes to maximize the profit? The CPU can run only 1 process at a time.

To understand this problem better, let us take an example as follows –

Task	T_1	T_2	T_3	T_4
Profit	100	10	27	15
Deadline	2	1	2	1

Ideally, we should finish these 4 processes in 2 time units. However, we can only run 1 process at a time so max we can run 2 processes here. As we are expected to maximize profit, it is a clear choice that we need to run processes T1 and T3 to get a **max profit of 127.**

One more interesting thing to note here is that we try to run the process with larger profit **as late as possible**. The reason for this is that we can ensure the process will run but at the same time we are providing time before it for other processes to try and run as well. Thus, the final solution becomes –

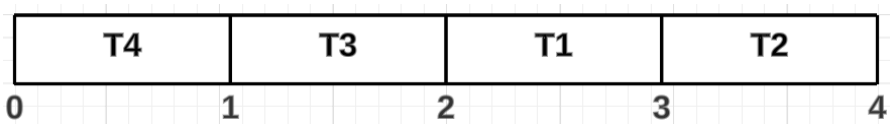


Thus, in this case we are being greedy for both – profit and deadline. We are maximizing profit while delaying the deadline as late as possible 😊

Let us try another example here –

Task	T_1	T_2	T_3	T_4	T_5	T_6	T_7
Profit	35	30	25	20	15	12	5
Deadline	3	4	4	2	3	1	2

In this case, the timeline will be as follows –



The profit here will be **110.**

DYNAMIC PROGRAMMING

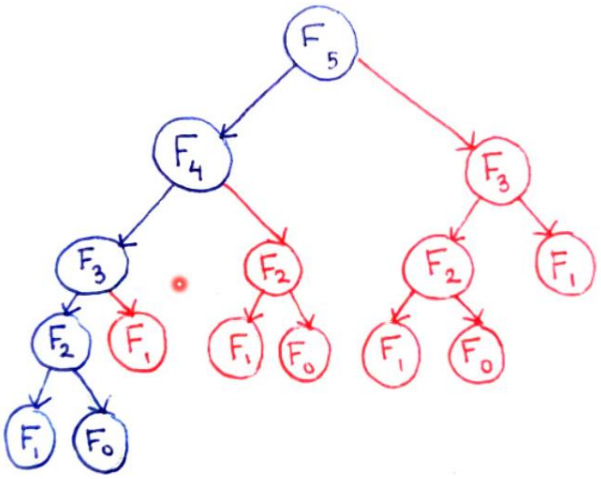
Dynamic programming (DP) is an approach where we are *"Learning from the Past",* so as to speak. In this case, a DAC problem is created such that it has **dependent or overlapping sub-problems.** For example, let us look at the following problem –

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n) &= F(n-1) + F(n-2) \end{aligned}$$

This is the case of the Fibonacci series and the recursive function can be written as –

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

Thus, the TC for this function using regular DAC will be $\boldsymbol{O(2^n)}$**.** The graph for the same can be written as follows –



As we can see, we are calling F0, F1, F2,…multiple times. Instead of calculating them all over again, it would make more sense to store their values in an array as follows –

n	0	1	2	3	4	5
F(n)	0	1	1	2	3	5

Now, instead of calculating the entire F function over again, we can simply take that value from the array itself. This is the concept of **Dynamic Programming** where we have similar sub-problems and we are storing values calculated in the past. Using the array, the complexity of the problem is now reduced to $\boldsymbol{O(n)}$**.**

There are four steps of dynamic programming

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

LONGEST COMMON SUB-SEQUENCE

This is an application of dynamic programming. To proceed, we need to understand what a sub-sequence is. **A sub-sequence** is a part of a string that is **in order but need not be continuous.** For example, we have a string as follows –

$$S = 'abcd'$$

In this case,

$$Substrings = \{a, b, c, d, ab, bc, cd, abc, bcd, abcd\}$$

$$Subsequences = \{a, b, c, d, ab, \boldsymbol{ac}, \boldsymbol{ad}, bc, \boldsymbol{bd}, cd, abc, \boldsymbol{abd}, \boldsymbol{acd}, bcd, abcd\}$$

The sequences in BOLD are part of sub-sequences but not in substrings. Now, we can define the problem statement. Imagine we have 2 strings of sizes m and n as follows –

$$X = \{x1\ x2\ x3\ x4 \dots xm\}$$

$$Y = \{y1\ y2\ y3\ y4 \dots yn\}$$

Now, we need to find a string $Z = \{z1\ z2\ z3 \dots zk\}$ of **max length $\boldsymbol{k}$** such that zi is present in both $X\ and\ Y$.

Since we are trying to find the sub-sequence, we are not bothered by the continuity but only focused on the order of the characters in the strings. Hence, we can start at the back of the strings as follows –

- If $xm = yn$, then
  - Set $zk = xm = yn$
  - $m--$ and $n--$
- If $xm \neq yn$, then
  - $m--$ and repeat the process
  - $n--$ and repeat the process
  - $m--and\ n--$ and repeat the process

Let us take an example to understand this.

$$X = \{A, B, C, B, D, A, B\}$$

$$Y = \{B, D, C, A, B, A\}$$

Here, $m = 7$ and $n = 6$. We can observe that $x7 \neq y6$. So now, we have to –

- Check $x6$ and $y6$

- Check $x7$ and $y5$
- Check $x6$ and $y5$

This can get confusing to just keep on repeating (recursion) over and over again, so we would rather use a **table** to get the same result –

			B	D	C	A	B	A
		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
A	1	0						
B	2	0						
C	3	0						
B	4	0						
D	5	0						
A	6	0						
B	7	0						

The characters of X denotes the row numbers and Y denotes the column numbers. It could be either way…no issues 😊

Now, we need to understand this table. A 4x4 grid in this table is a representation as follows –

X_{M-1} Y_{N-1}	X_{M-1} Y_N
X_M Y_{N-1}	X_M Y_N

In each cell (X_M, Y_N), we do the following set of operations –

- Check if row character and column character match
  - If yes, then $(X_M, Y_N) = 1 + (X_{M-1}, Y_{N-1})$
  - If no, then $(X_M, Y_N) = \max[(X_M, Y_{N-1}), (X_{M-1}, Y_N)]$
    - If $(X_M, Y_{N-1}) = (X_{M-1}, Y_N)$, then take $(X_M, Y_N) = (X_M, Y_{N-1})$

In short, if the row and column values match, then take the upper diagonal value and add 1 to get the cell value. If no match, then take the max of either the vertical or horizontal cell. If these values are also the same, then take the horizontal value. We can take vertical values also, but make sure to take either one…don't mix and match.

Now, let us fill the table as follows –

			B	D	C	A	B	A
	n	**0**	**1**	**2**	**3**	**4**	**5**	**6**
	0	0	0	0	0	0	0	0
A	**1**	0	0	0	0	1D	1H	1D
B	**2**	0	1D	1H	1H	1H	2D	2H
C	**3**	0	1V	1H	2D	2H	2H	2H
B	**4**	0	1D	1H	2V	2H	3D	3H
D	**5**	0	1V	2D	2H	2H	3V	3H
A	**6**	0	1V	2V	2H	3D	3H	4D
B	**7**	0	1D	2V	2H	3V	4D	4H

Here, V denotes vertical, H denotes Horizontal and D denotes Diagonal. Now, we go from the bottom right cell and parse upwards. Every character of the Y with a D cell is the sub-sequence. Hence, in this case, the sub-sequence will be –

$$Z = \{B, D, A, B\}$$

Now that we have the solution, we can proceed to write a code for this –

```
LCS-Length (x, y)
{
        m ← Length[x]
        n ← Length[y]
        for i ←1 to m
        {
                do C[i, 0] ← 0
        }
        for j ←0 to n
        {
                do C[0, j] ← 0
        }
        for i ←1 to m
        {
                for j ←1 to n
                {
                        if (x_i = y_j)
                        {
                                c[i, j] ← c[i-1, j-1] + 1
                                b[i, j] ← 'D_edge'
                        }
                        else if (C[i-1, j] >= C[i, j-1])
                        {
                                C[i, j] ← C[i-1, j]
                                b[i, j] ← 'V_edge'
                        }
                        else
                        {
                                C[i, j] ← C[i, j-1]
                                b[i, j] ← 'H_edge'
                        }
                }
        }
        return b and C
}
```

The above code is used to create the matrix, fill the values and denote which edge is being used (H, V or D) in the cells. Next, we need the code to parse and print the longest sub-sequence

```
Print_LCS (b, X, i, j)
{
        if i=0 or j=0
                return
        if (b[i, j] = 'D_edge')
        {
                then Print_LCS (b, X, i-1, j-1)
                Print X_i
        }
        Else if (b[i, j] = 'V_edge')
        {
                Print_LCS (b, X, i-1, j)
        }
        Else
        {
                Print_LCS (b, X, i, j-1)
        }
}
```

The LCS_Length() function has a TC of $\boldsymbol{O(mn)}$ while the TC of Print_LCS() is $\boldsymbol{O(m+n)}$

NOTE

There can be more than 1 longest sub-sequences between 2 strings.

MATRIX CHAIN MULTIPLICATION

We know that a matrix multiplication will be done as follows –

$$A_{a\times b} * B_{b\times c} = C_{a\times c}$$

In this case, we can conclude –

$$No\ of\ scalar\ multiplications = a * b * c$$

For matrix multiplication, we know that the operations is **associative and not commutative.** Let us take an example as follows –

$$(A_{2\times 3} * B_{3\times 5}) * C_{5\times 2} = A_{2\times 3} * (B_{3\times 5} * C_{5\times 2})$$

In this case, we can see –

$$Scalar\ multiplication\ in\ LHS = (2 * 3 * 5) + (2 * 5 * 2) = 50$$

$$Scalar\ multiplication\ in\ RHS = (2 * 3 * 2) + (3 * 5 * 2) = 42$$

Thus, we can see that based on the order of multiplication of the matrices, the number of scalar multiplication changes. Since multiplication is an expensive operation, it is best if we try to minimize the number of scalar multiplications. This is where DP comes into play.

Let us assume that we have a certain number of matrices $A_i, A_{i+1}, A_{i+2}, \dots, A_j$ and we need to multiply them together by performing the minimum number of scalar multiplications. We use DP to solve this problem and hence we need to first use DAC.

Let's say we have –

$$S = A_i * A_{i+1} * A_{i+2} * \dots * A_j = (A_i * A_{i+1} * \dots * A_k) * (A_{k+1} * A_{k+2} * \dots * A_j)$$

As we can see, we have divided the problem into 2 sub-problems where $i \leq k < j$. Like this, we can keep dividing the problem further and further. Let us now define another parameter. Suppose we have 3 matrices as follows – $A_{a\times b}, B_{b\times c}$ and $C_{c\times d}$. In this case, we denote –

$$P_0 = a\ ;\ P_1 = b\ ;\ P_2 = c\ ;\ P_3 = d$$

Basically, P denotes the order of the matrices. Given these parameters, we can say that for a certain value of i and j and with $i \leq k < j$, we say that the **minimum no of scalar multiplications** will be –

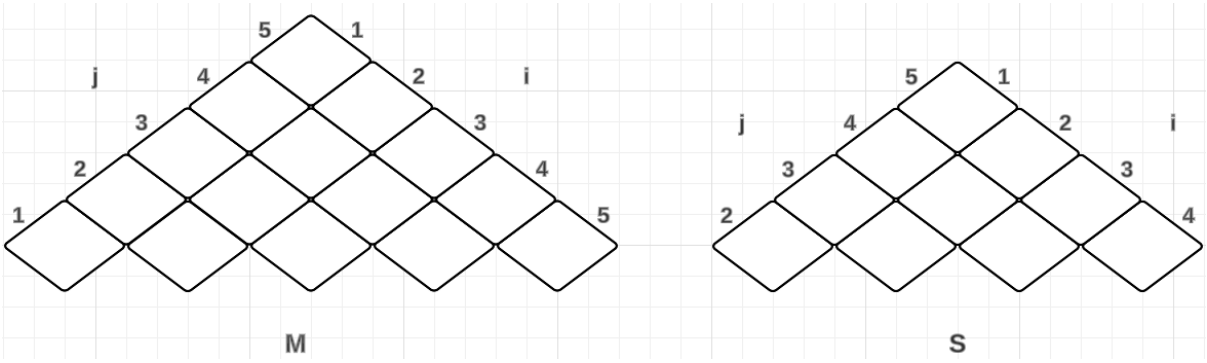
$$\boldsymbol{m[i,j] = \min\left[m[i,k] + m[k+1,j] + P_{i-1}P_kP_j\right]}$$

We can interpret this as that we are taking the value for the no of multiplications taken by the divided (children) processes and the current stage and then getting the minimum value. Let us try to understand this with a problem.

Assume we have the following matrices – A_1, A_2, A_3, A_4, A_5 with the orders $30 \times 35, 35 \times 15, 15 \times 5, 5 \times 10, 10 \times 20$. Thus, we can write –

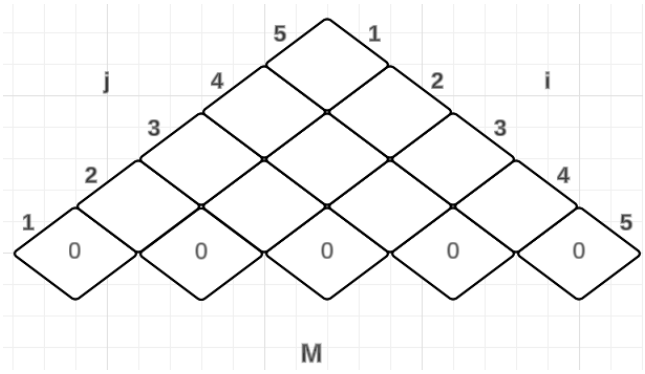
$$P_0 = 30\ ;\ P_1 = 35\ ;\ P_2 = 15\ ;\ P_3 = 5\ ;\ P_4 = 10\ ;\ P_5 = 20$$

Now, we draw 2 structures as follows –



Here, structure $\boldsymbol{M}$ will help us get the minimum no of multiplications while structure $\boldsymbol{S}$ will help us get the sequence of multiplication. Let us first take structure $\boldsymbol{M}$**.**

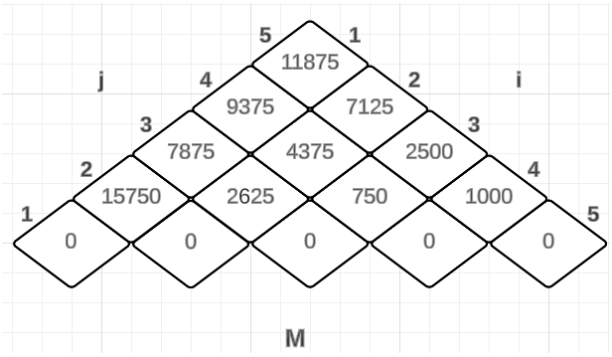
We proceed from left-to-right and bottom-to-top. The first cell will be $(i,j) = (1,1)$. We can see that $(1,1)$ cell basically means we are just referencing matrix A_1 and there is **no multiplication.** Similarly, every case where $i = j$, we can simply put a **zero.**



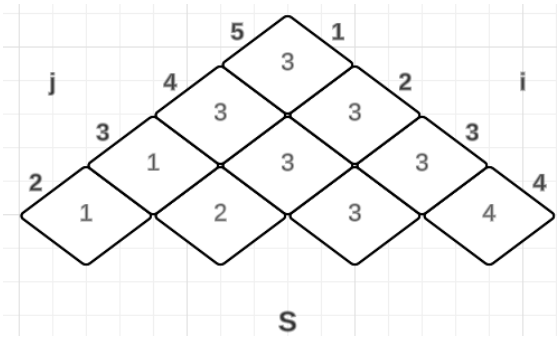
Now, we take the next row's first cell which is $(i, j) = (1,2)$. Thus, we get that $k = 1$. Using the formula we mentioned earlier, we can write –

$$m[1,2] = \min[m[1,1] + m[2,2] + P_0P_1P_2] = 0 + 0 + (30 * 35 * 15) = \mathbf{15750}$$

Similarly, we can fill the rest of the values as follows –



Now, we can start filling structure $\boldsymbol{S}$**.** For every pair of (i, j) in the structure, we will fill the value of k for which we get minimum value. Thus, we get –



Now that we have this, we can re-write our original problem as –

$$A_1 * A_2 * A_3 * A_4 * A_5$$

At this point, we have $(i, j) = (1,5)$. In structure $\boldsymbol{S}$**,** we can see that $(1,5) = 3$. Thus, we need to divide the expression on element 3 as follows –

$$(A_1 * A_2 * A_3) * (A_4 * A_5)$$

Now, we know that $(i, j) = (1,3) = 1$. Therefore, we get the final expression as –

$$\big((\boldsymbol{A_1}) * (\boldsymbol{A_2} * \boldsymbol{A_3})\big) * (\boldsymbol{A_4} * \boldsymbol{A_5})$$

Basically, using structure S we have added parenthesis to the expression so as to get the **minimum no of scalar multiplications.**

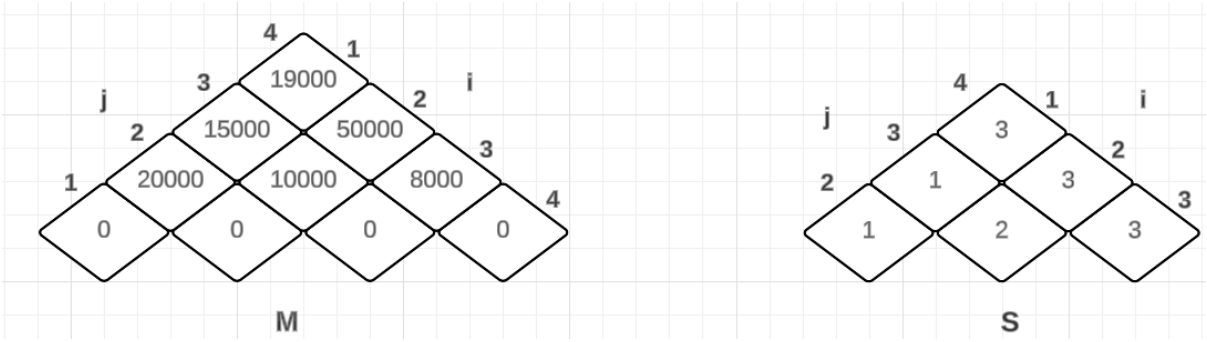
Question

Q Four matrices M_1, M_2, M_3 and M_4 of dimensions pxq, qxr, rxs and sxt respectively can be multiplied is several ways with different number of total scalar multiplications. For example, when multiplied as ((M_1 X M_2) X (M_3 X M_4)), the total number of multiplications is pqr + rst + prt. When multiplied as (((M_1 X M_2) X M_3) X M_4), the total number of scalar multiplications is pqr + prs + pst. If p = 10, q = 100, r = 20, s = 5 and t = 80, then the number of scalar multiplications needed is:

a) 248000 b) 44000 c) 19000 d) 25000

Answer

We can draw the 2 structures as follows –

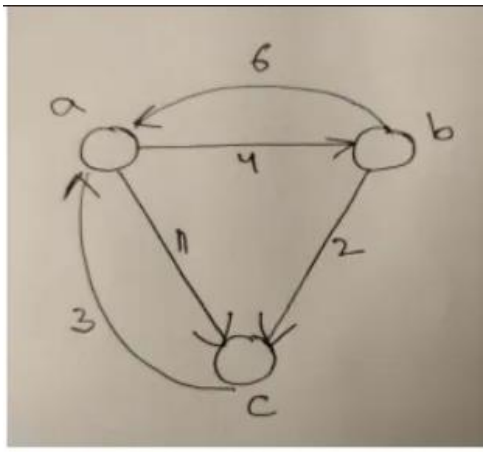


We can see that min no of scalar multiplication is **19000** and the expression will become –

$$(M_1 * M_2) * (M_3 * M_4)$$

FLOYD WARSHALL ALGORITHM (All Pair Shortest Path)

This is another problem that can be solved using DP. In this algorithm, the input provided is a **directed weighted graph.** We use this algorithm to get the **shortest path between all nodes**. For example, let us take the graph below –



This is a basic graph with 3 nodes and edges as shown above. In the first step, we need to create 2 matrices –

- $\boldsymbol{D}$ matrix to note the weight of the **direct path** taken
- $\boldsymbol{P}$ matrix to note the nodes via which the path is taken

In these matrices, the rows denote the **source** and the columns denote the **destination.** Initially, we can write the two matrices as follows –

	a	b	c
a	0	4	11
b	6	0	2
c	3	inf	0

D0

	a	b	c
a	----	a	a
b	b	----	b
c	c	nil	----

P0

Next, we write these matrices by considering the paths via each individual node.

Path via Node a

	a	b	c
a	0	4	11
b	6	0	2
c	3	7	0

D0

	a	b	c
a	----	a	a
b	b	----	a
c	c	a	----

P0

Path via Node b

	a	b	c
a	0	4	6
b	6	0	2
c	3	7	0

Da

	a	b	c
a	----	a	b
b	b	----	a
c	c	a	----

Pa

Path via Node c

	a	b	c
a	0	4	6
b	5	0	2
c	3	7	0

Da

	a	b	c
a	----	a	b
b	c	----	a
c	c	a	----

Pa

This is the final matrix. We can see that D will give the minimal distances between all nodes as expected.

SUM OF SUBSET

The problem is another application of DP. Let us assume we have a set of non-negative numbers and a value $\boldsymbol{S}$**.** Basically, we need to find a subset of the set whose elements add up to $\boldsymbol{S}$**.** We can do this using trial and error, but here is how DP solves this problem. Let us assume –

$$Set\ A = \{2,3,8,10,7\}$$

$$S = 14$$

First, we create a table with columns from $0 - S$ and rows with set elements in ascending order.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
{2}															
{2,3}															
{2,3,7}															
{2,3,7,8}															
{2,3,7,8,10}															

We can see that a set with no elements will result in a sum of 0. Since the null set is a subset of all sets, all sets will be able to result in a sum of 0.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
{2}	T														
{2,3}	T														
{2,3,7}	T														
{2,3,7,8}	T														
{2,3,7,8,10}	T														

Similarly, we can see that none of the subsets will be able to produce a sum of 1. Hence, we can write –

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
{2}	T	F													
{2,3}	T	F													
{2,3,7}	T	F													
{2,3,7,8}	T	F													
{2,3,7,8,10}	T	F													

We can similarly write the same for all the other columns –

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
{2}	T	F	T	F	F	F	F	F	F	F	F	F	F	F	F
{2,3}	T	F	T	T	F	T	F	F	F	F	F	F	F	F	F
{2,3,7}	T	F	T	T	F	T	F	T	F	T	T	F	T	F	F
{2,3,7,8}	T	F	T	T	F	T	F	T	T	T	T	T	T	T	F
{2,3,7,8,10}	T	F	T	T	F	T	F	T	T	T	T	T	T	T	**F**

As we can see, the final cell is a **FALSE.** Hence, we **can't write** 14 as a sum of the elements of any of the subsets of A.

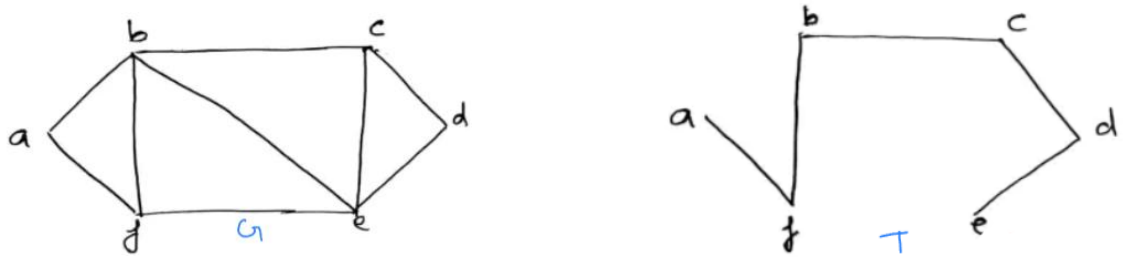
Where does DP come into this? Let us assume we need to find the value of cell in Row $\{2,3,7\}$ and column 10. First, we perform $10 - 7 = 3$. We now move to column 3 and row $\{2,3\}$. We can see that

it is TRUE and hence, the value of cell in Row $\{2,3,7\}$ and column 10 will also be TRUE. As we have referenced a prior value to get the current value, this is where the DP comes into play 😊

MINIMUM SPANNING TREE (MST)

In Programming and DS, we learnt that a tree is a hierarchical method of representing data. That is correct, but here we are going to define a tree as per the Graph Theory concept. A **tree** is a **graph** which is **connected AND acyclic.** This means that each pair of nodes must have at least 1 path connecting them (connected) and there must NOT be any cycle.

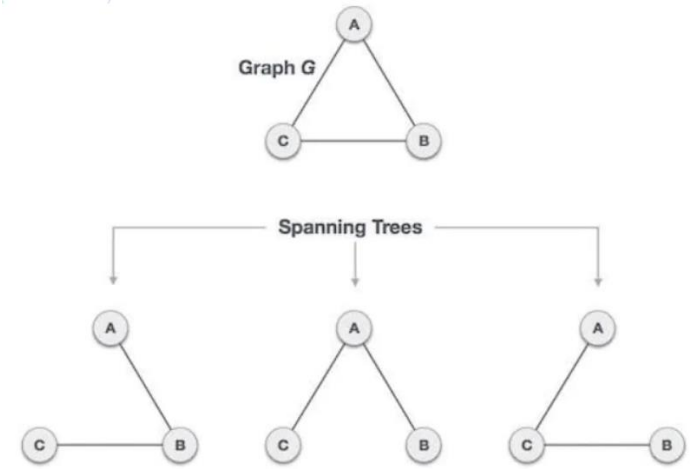
A **spanning tree** is a **subset** of a connected graph such that it **includes ALL vertices.** For example,



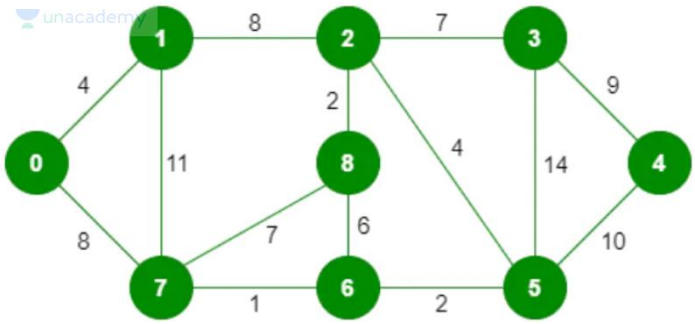
Here, we have the graph G and a subset of the graph T. We can see that T is connected, acyclic and spans all the vertices of G. Hence, $\boldsymbol{T}$ is a **Spanning Tree.**

NOTE

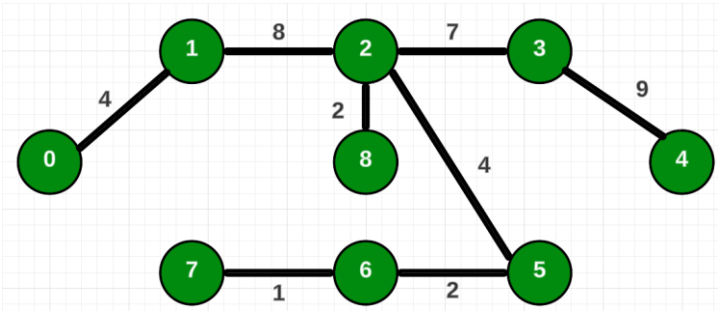
One graph can have multiple spanning trees.



Now, we can proceed with the **Minimum Spanning Tree** problem. Let us consider a graph G which is **weighted and undirected.** Our job is to find a spanning tree $\boldsymbol{T}$ that has the minimum cost. For example, let us assume the graph to be as follows –



Now, this graph can have multiple spanning trees. Let us look at the tree below –



Out of all the spanning trees possible for this graph, the above tree has the minimum cost of **37** and is hence the minimum spanning tree.

NOTE

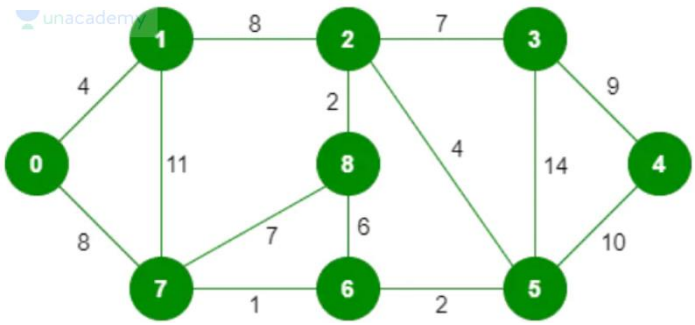
A graph can have **multiple minimum spanning trees.**

KRUSKAL'S ALGORITHM

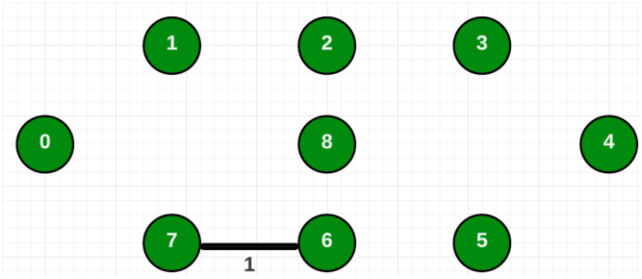
This algorithm is a way to find the MST of a graph. It has 3 steps –

- Find the available edge with the minimum weight
- If that edge doesn't create a cycle, consider it in the tree
- Repeat till we connect all the nodes

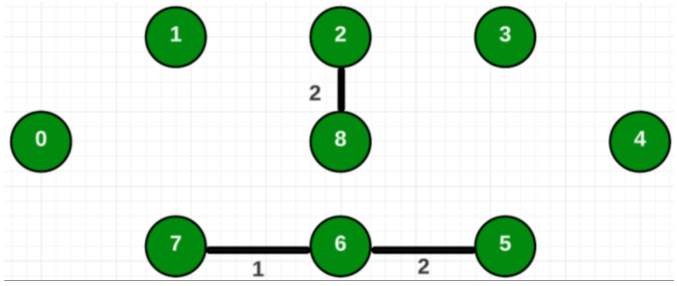
For example, let us take the previous graph itself –



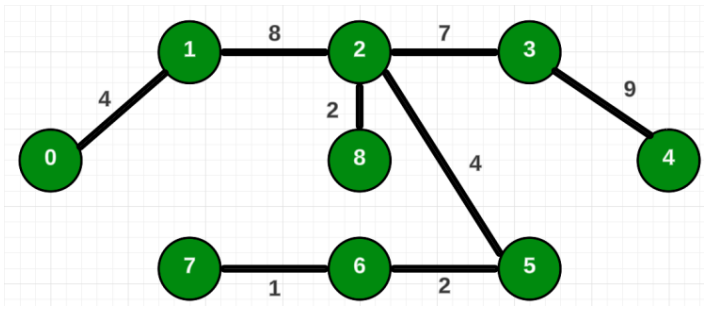
Initially, all the edges are available. We can see that edge with the minimum weight is between 7-6 with weight 1. Hence, we take that edge 1st.



Next, we take the edges with weight 2. Since they don't form a cycle, we can take them.



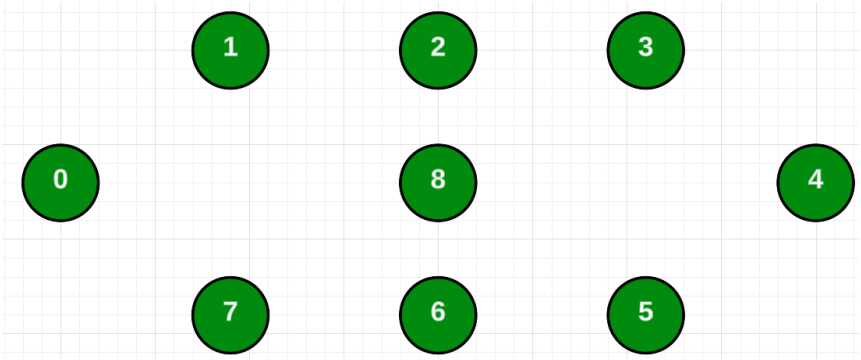
Similarly, we continue this process and end up with the MST as follows –



Kruskal's algorithm can be written as follows –

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

In this case, we first start with an empty tree $A = \phi$. Now, we take every vertex of the graph and apply the **MAKE-SET** function on each vertex. This function basically converts each node into a sort of tree with single node. In short, this function ensures that each node is not connected to any other node. Now we have the initial position set as follows –



Next, we sort the weights by the increasing order of their weights. This is the prime basis of Kruskal's algorithm since we will be checking the edges in increasing order of weights.

Finally, in Line 5 we are executing the main loop. Here, we take each edge between nodes u and v and run the **FIND-SET** function. This function will return a list of nodes we can reach to with from the input node. As we can see, we are checking the condition –

$$FIND - SET(u) \neq FIND - SET(v)$$

The idea is that if node u and node v can be reached by the same nodes, then adding an edge between them will cause a cycle. Hence, we only add an edge if the FIND-SET of the nodes are not equal. Let us run this algorithm step-by-step –

Iteration 1

$$E = \{7 - 6\}\,; U = 7\,; V = 6$$

$$FIND - SET(U) = \{7\}$$

$$FIND - SET(V) = \{6\}$$

Thus, we add an edge between 7-6.

Iteration 2

$$E = \{6 - 5\,, 2 - 8\}\,; U = \{2{,}6\}\,; V = \{5{,}8\}$$

$$FIND - SET(2) = \{2\}$$

$$FIND - SET(5) = \{5\}$$

$$FIND - SET(6) = \{6{,}7\}$$

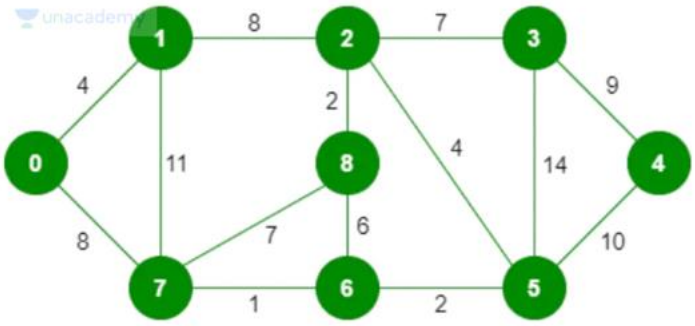
$$FIND - SET(8) = \{8\}$$

Thus, we add an edge between 2-8 and 6-5.

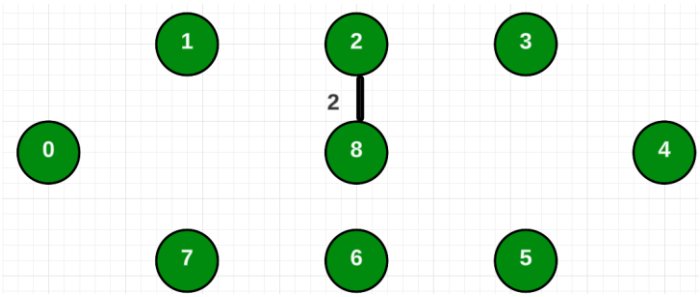
Similarly, we continue and we can get the MST at the end. This is the most effective algorithm with the time complexity of $\boldsymbol{O(E \log V)}$.

PRIM'S ALGORITHM

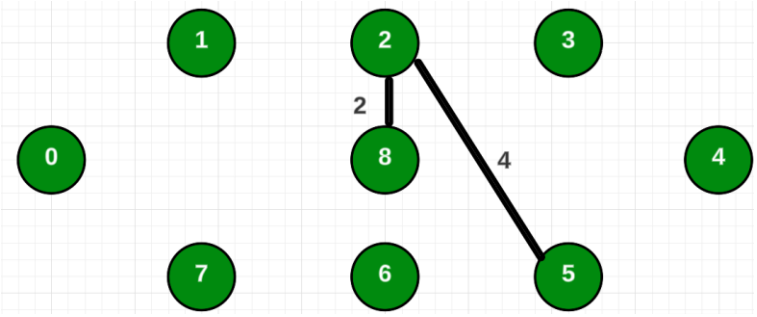
Unlike Kruskal's algorithm, we are not sorting the edges here in ascending order. Instead, we will first start with any node as a root node. Then, we find the shortest path to any of the other nodes. Then, we find the shortest path from these 2 nodes to any other node and so on. For example,



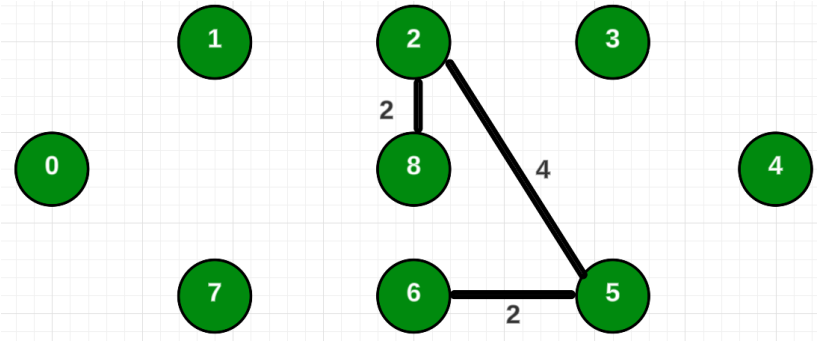
Let us take node 8 as the root node. From Node 8, we can see that 2-8 has the shortest distance. Hence, we get –



Now we check from Nodes 2 and 8. We can see that the shortest length is between 2-5 of weight 4.



Now we check from Nodes 2, 8 and 5. WE can see that the shortest length is between 5-6 of weight 2.



Similarly, we continue and get the MST. We can write the algorithm as follows –

```
Minimum_Spanning_Tree (G, W, R)
{
        for each u ϵ V(G)
        {
                key[u] ← ∞
                ∏[u] ← NIL
        }
        Key[r] ← 0
        Q ←V[G]
        While (Q != ϕ)
        {
                u ← Extract-Min(Q)
                For each v ϵ adj[u]
                {
                        if (v ϵ Q and w(u, v) < key[v])
                        {
                                ∏[v] ← u
                                key[u] ←  w(u, v)
                        }
                }
        }
}
```

DIJKSTRA'S ALGORITHM

This algorithm is very similar to Prim's algorithm but there are two pivotal differences between the two. First off, **Prim's algorithm** is used to find the **MST** while **Dijkstra's algorithm** is used to get the **shortest path between a root node and all the other nodes**.

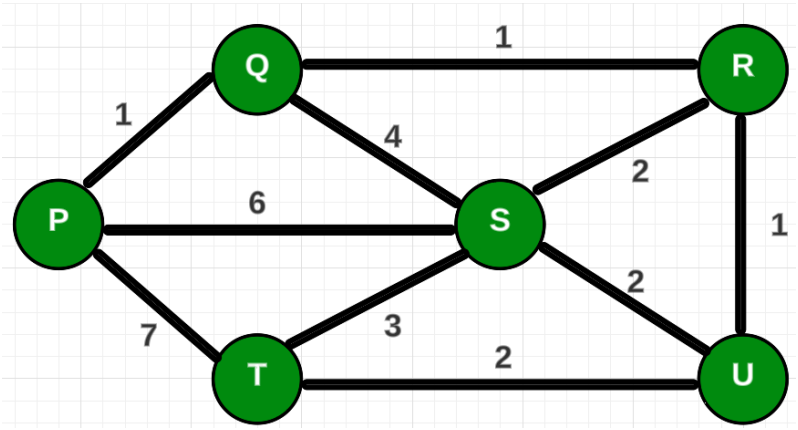
The code for the same is given below –

```
Dijkstra algorithm (G, W, S)
{
        initialize-Single-source (G, S)
        S ← ϕ
        Q ← V[G]
        While (Q != ϕ)
        {
                u ← extract-min (Q)
                S ← S U {u}
                for each vertex v ϵ adj(u)
                {
                        relax (u, v, w)
                }
        }
}
```

```
Initialize_Single_Source (G, S)
{
        for each vertex v ϵ V[G]
        {
                d[v] ← ∞
                ∏[v] ← NIL
        }
        d[S] ← 0
}
```

```
Relax (u, v, w)
{
        if(d[v] > d[u] + w (u, v))
        {
                d[v] ← d[u] + w (u, v)
                ∏[v] ← u
        }
}
```

Let us take an example to see how the algorithm is working. Consider the graph below –



First, we run the Initialize-Single-Source function. This will create a structure as follows –

	P	Q	R	S	T	U
d	0	inf	inf	inf	inf	inf
Pi	-	-	-	-	-	-

Here, we have taken P as the source node. Since P is the minimum, we extract that.

	P	Q	R	S	T	U
d	0	inf	inf	inf	inf	inf
Pi	-	-	-	-	-	-

Now, we are going to look at the adjacent nodes of node P which are Q, S, T. For each adjacent node, we find the path from P and fill the table.

	P	Q	R	S	T	U
d	0	1	inf	6	7	inf
Pi	-	P	-	P	P	-

Again, we extract Q as that is the minimum distance node.

	P	Q	R	S	T	U
d	0	1	inf	6	7	inf
Pi	-	P	-	P	P	-

Again, we take the adjacent nodes of Q which are R, S. Now, here is where things get interesting.

In the Prim's algorithm, we calculate the distance from the parent node. However, here we calculate the distance from the **source node.** So, in Prim's algorithm $d[R] = 1$ but in Dijkstra's algorithm we have $d[R] = 2$.

Thus, if we continue, we will get –

	P	Q	R	S	T	U
d	0	1	2	4	5	3
Pi	-	P	Q	R	U	R

Thus, we can get the shortest paths between source and the nodes as follows –

$$P \to Q(1)$$

$$P \to Q \to R(2)$$

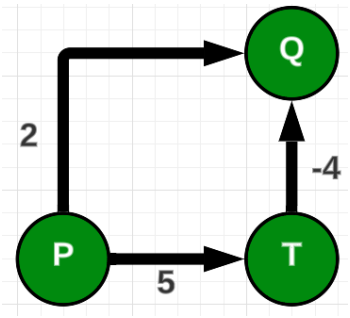
$$P \to Q \to R \to S(4)$$

$$P \to Q \to R \to U(3)$$

$$P \to Q \to R \to U \to T(5)$$

NOTE

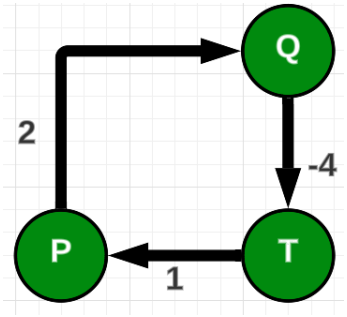
Dijkstra's algorithm can work on both directed and un-directed graphs while Prim's works only on un-directed graphs. Additionally, Dijkstra's **can fail for negative weights.** Consider the following graph –



Let us assume we are using P as the source. In this case, Dijkstra will first clear P. Then, it will see that Q is the closest and clear Q with path length 2. However, we can see that $P - T - Q$ has a path length of 1 and is clearly the shorter path. Hence, this is a simple example of why Dijkstra's algorithm fails with negative weights.

BELLMAN – FORD ALGORITHM

This algorithm is developed to resolve the negative weight situation. It is slower than Dijkstra but is more versatile. There is one case however where Shortest Path can't be found. Let us take the case below –



In this case, since P is the source we assign $d[P] = 0$. This gives us $d[Q] = 2$ and $d[T] = -2$. However, we can see that $P - Q - T - P$ will provide a path weight of -1. Thus, we get $d[P] = -1$ and hence we get $d[Q] = 1$ and $d[T] = -3$. But now, we get the path weight of $P - Q - T - P$ will become -2.

In short, in the above case we can't find the shortest path since the **cycle weight is negative.** In short, the sum of the weights in the cycle is negative. In such a case, we can't find the shortest path.

Bellman – Ford algorithm finds the shortest path and if the cycle weight is negative, then it returns FALSE. The algorithm is as follows –

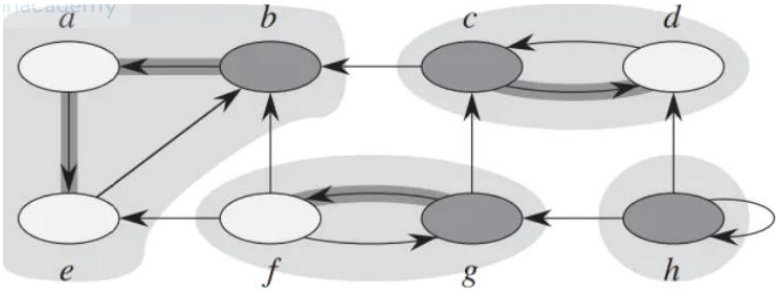
```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

The basic point is that for a graph with n nodes, the shortest path between 2 nodes can either be direct or can involve a maximum of $n - 1$ edges. Hence, the algorithm loops from $1\ to\ (n - 1)$ and for each iteration, it runs a loop on all edges and performs the RELAX function same as Dijkstra.

After this loop is completed, it can be guaranteed that we would have found the shortest path. However, we run another RELAX loop. In this case, if there is relaxation i.e. we were able to find a shorter path, then it means that there is a **cycle with negative weight** and the algorithm returns FALSE and breaks out.

STRONGLY CONNECTED COMPONENTS

In a directed graph, if any node can be reached by any other node, then it is called **a Strongly Connected** graph. Suppose, we take the graph as follows –



In the graph below, if we start at node H, then we can reach every other node. However, if we start at A, then it can only be reached by $\{B, E\}$. Hence, this graph is **NOT a strongly connected graph.**

In this graph, suppose we take the sub-graph $\{A, B, E\}$. That sub-graph is a strongly connected graph as every node can be connected to any other node. This is called a **Strongly Connected Component.**

Basically, a **strongly connected component (SCC)** is a sub-graph of a non-strongly connected graph which is strongly connected. Hence, in the above graph we have 4 SCCs – $\{A, B, E\}$, $\{F, G\}$, $\{C, D\}$, $\{H\}$.

Now, we can treat every SCC as a node as follows –

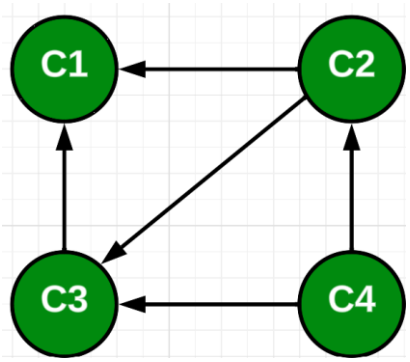
$$C1 = \{A, B, E\}$$

$$C2 = \{F, G\}$$

$$C3 = \{C, D\}$$

$$C4 = \{H\}$$

Then the graph will be as follows –



This is a typical example of a **Directed Acyclic Graph (DAG).** A graph with every node as a SCC will **NEVER have a cycle.** Since this is a DAG, we can have a **topological order** for the same. This is the same as the topological order discussed in DBMS. For the above graph, we have –

$$Topological\ Order = C4 \rightarrow C2 \rightarrow C3 \rightarrow C1$$

Question

Find TC of the following code –

```c
#include <stdio.h>
void main()
{
    x = y + z; //Statement 1
    for (i=0 ; i<n ; i++) {
        x = y + z; //Statement 2
    }
    for (i=0 ; i<n ; i++) {
        for (j=0 ; j<n ; j++) {
            x = y + z; //Statement 3
        }
    }
}
```

Answer

$$Order\ of\ Magnitude(Statement\ 1) \rightarrow 1$$

$$Order\ of\ Magnitude(Statement\ 2) \rightarrow n$$

$$Order\ of\ Magnitude(Statement\ 3) \rightarrow n * n = n^2$$

Therefore,

$$\boldsymbol{T(P) = O(n^2)}$$

Question

```c
#include <stdio.h>
void main()
{
    int i = 1;
    while (i <= n) {
        i++;
    }
}
```

Answer

$$Order\ of\ magnitude = 1 + n$$

$$\boldsymbol{T(P) = O(n)}$$

Question

```c
#include <stdio.h>
void main()
{
    int i = 1;
    while (i <= n) {
        i = i + 5;
        i = i + 7;
    }
}
```

Answer

Here, every time the loop runs the value of i gets incremented by 12. Therefore,

$$Order\ of\ mag = 1 + \frac{n}{12}$$

$$\boldsymbol{T(P) = O(n)}$$

Question

```c
#include <stdio.h>
void main()
{
    int i = 1;
    while (i <= n) {
        i = i * 2;
    }
}
```

Answer

In this case, after each iteration the value of i is being doubled. Suppose, the above program loops for some $\boldsymbol{k}$ iterations. In that case, we can say –

$$i_{final} = 2^k = n$$

$$\boldsymbol{k = \log_2 n}$$

Since the loop iterates k times, we can write –

$$Order\ of\ mag = 1 + k = 1 + \log_2 n$$

$$\boldsymbol{T(P) = O(\log_2 n)}$$

Question

```c
#include <stdio.h>
void main()
{
    int i = 10;
    while (i <= n) {
        i = 12 * i;
    }
}
```

Answer

Initially $i = 10$. With every iteration, $i^+ = 12 * i$. Therefore, after K iterations the value of i becomes $12K * 10$. Therefore,

$$12K * 10 = n$$

$$K = \log_{12}\left(\frac{n}{10}\right) = \log_{12} n - \log_{12} 10$$

Therefore, we get –

$$Order\ of\ mag = 1 + K = 1 + \log_{12} n - \log_{12} 10$$

$$\boldsymbol{T(P) = O(\log_{12} n)}$$

Question

```
#include <stdio.h>
void main()
{
    while (n >= 1) {
        n = n/2;
    }
}
```

Answer

After K iterations, we can see that n becomes $^{n}/_{2^K}$. Therefore,

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

$$K = \log_2 n$$

Therefore,

$$\boldsymbol{T(P) = O(\log_2 n)}$$

Question

```
#include <stdio.h>
void main()
{
    i = 2;
    while (i <= n) {
        i = i**2;
    }
}
```

Answer

After K iterations, we get $i^+ = i^{2^K}$. Therefore,

$$2^{2^K} = n$$

$$K = \log_2(\log_2 n)$$

Therefore,

$$\boldsymbol{T(P) = O(\log_2(\log_2 n))}$$

Question

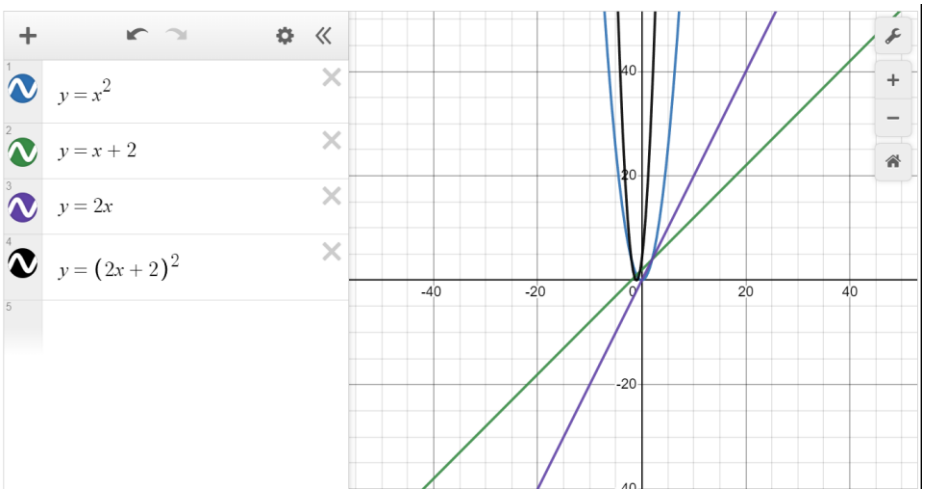
```
#include <stdio.h>
void main()
{
    i = 1;
    while (i <= n) {
        i = i + 2;
        i = i * 2;
        i = i ** 2;
    }
}
```

Answer

We can see that –

$$Order\ of\ mag = 1 + n + \log_2 n + \log_2(\log_2 n)$$

We can also see that if we plot the graph of the statements, we can see –



We can see that $\log_2 n$ and $\log_2(\log_2 n)$ are very similar. Therefore, we can say –

$$\boldsymbol{T(P) = O(\log_2(\log_2 n))}$$

Question

Find the TC and the final value of q

```
#include <stdio.h>
int main()
{
    int p = 0;
    int q = 0;
    for (i=1 ; i<=n ; i=2*i)
        p++;
    for (i=1 ; i<=p ; i=2*i)
        q++;
}
```

Answer

We can see that the 1st loop will run $\boldsymbol{\log_2 n}$ times while the second loop will run for $\boldsymbol{\log_2 p}$ times. Therefore,

$$Order\ of\ mag = 1 + \log_2 n + \log_2 p$$

Since p is getting incremented every time the first loop runs, we can write –

$$p = \log_2 n$$

$$Order\ of\ mag = 1 + \log_2 n + \log_2(\log_2 n)$$

Since the 2nd loop runs $\log_2(\log_2 n)$ times, we can write –

$$\boldsymbol{q = \log_2(\log_2 n)}$$

In addition to that, we get –

$$\boldsymbol{T(P) = O(\log_2 n)}$$

Question

```c
#include <stdio.h>
int main()
{
    for (i=1 ; i<=n ; i=2*i) {
        int p = 0;
        int q = 0;

        for (j=10 ; j<=n**2 ; j=7*j)
            p++;
        for (k=n**3 ; k>=7 ; k=k/9)
            q++;
    }
}
```

Answer

The outer loop (i) will loop for $\boldsymbol{\log_2 n}$ times. Every time it loops, we also loop j and k. For j loop, let's assume it executes A times. Then –

$$7A * 10 = n^2$$

$$7A = \frac{n^2}{10}$$

$$\boldsymbol{A = 2\log_7 n - \log_7 10}$$

Similarly, if the k loop iterates for B times, then –

$$\boldsymbol{B = 3\log_9 n - \log_9 7}$$

We can see that we have multiple log functions with different bases. Hence, in such cases we can write –

$$\log_2 n \approx \log_7 n \approx \log_9 n \approx \log n$$

The proof of the above statement will be mentioned in the notes. Therefore, the outer loop executes for $\log n$ times and with every iteration, the inner loop also executes for $\log n$ times. Therefore,

$$\boldsymbol{T(P) = O\left((\log n)^2\right)}$$

Question

```
void main()
{
    int s = 1;
    int i = 1;
    while(s <= n)
    {
        s = s + i;
        i++;
    }
}
```

Answer

If we assume that the loop iterates k times, then –

$$1 + 2 + 3 + \cdots + k + 1(S) = n$$

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

Thus, we can see –

$$\boldsymbol{k \approx \sqrt{n}}$$

Therefore,

$$\boldsymbol{T(P) = O\left(\sqrt{n}\right)}$$

Question

```
void main()
{
    for(i=1 ; i<=n**7 ; i++)
    {
        for(j=1 ; j<=n**93 ; j++)
        {
            for(k=1 ; k<=64 ; k++)
            {
                x = y + z;
            }
        }
    }
}
```

Answer

Let us assume that the statement $x = y + z$ executes k times. Then,

$$k = 64 * n^{93} * n^{7} = 64n^{100}$$

Therefore,

$$\boldsymbol{T(P) = O(n^{100})}$$

Question

```
void main()
{
    for(i=1 ; i**4<=n**64 ; i++)
    {
        for(j=1 ; j**9<=n**53 ; j++)
        {
            x = y + z;
        }
    }
}
```

Answer

Let the i loop execute A times, then –

$$A^4 = n^{64}$$

$$A = n^{16}$$

Similarly, if the j loop executes B times, then –

$$B = n^{\frac{53}{9}}$$

Therefore,

$$\boldsymbol{T(P) = O\left(n^{\frac{197}{9}}\right)}$$

Question

```
void main()
{
    for(i=1 ; i<=n ; i++)
    {
        for(j=1 ; j<=i ; j++)
        {
            x = y + z;
        }
    }
}
```

Answer

If the statement iterates k times, then –

$$k = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

Hence,

$$\boldsymbol{T(P) = O(n^2)}$$

Question

```
void main()
{
    for(i=1 ; i<=log(n) ; i++)
    {
        for(j=1 ; j<=2**i ; j++)
        {
            x = y + z;
        }
    }
}
```

Answer

We know that –

$$2^{\log_2 x} = x$$

Let the entire loops be iterated K times, then –

$$2^1 + 2^2 + 2^3 + \cdots + 2^{\log_2 n} = K$$

$$K = 2\left(2^{\log_2 n} - 1\right)$$

$$K = 2n - 2$$

Therefore,

$$\boldsymbol{T(P) = O(n)}$$

Question

```
void main()
{
    for(i=2 ; i<=n/2 ; i++)
    {
        if(n % i == 0)
        {
            for(j=1 ; j<=n ; j++)
            {
                x = y + z;
            }
        }
    }
}
```

Find the time complexity assuming that n is a prime number

Answer

Since n is a prime number, then the **if condition** will only be satisfied when $i = \{1, n\}$. We can see that i doesn't take either 1 or n so the **if condition will never be satisfied.** Therefore,

$$\boldsymbol{T(P) = O(n)}$$

Question

Consider the following segment of C-code:

```
int j, n;
j = 1;
while (j <= n)
    j = j * 2;
```

The number of comparisons made in the execution of the loop for any $n > 0$ is:

A. $\lceil \log_2 n \rceil + 1$
B. n
C. $\lceil \log_2 n \rceil$
D. $\lfloor \log_2 n \rfloor + 1$

Answer

Option D

Question

Consider the following C function

```
int fun(int n) {
    int i, j;
    for(i=1; i<=n; i++) {
        for (j=1; j<n; j+=i) {
            printf("%d %d", i, j);
        }
    }
}
```

Time complexity of fun in terms of Θ notation is

A. $\Theta(n\sqrt{n})$
B. $\Theta(n^2)$
C. $\Theta(n \log n)$
D. $\Theta(n^2 \log n)$

Answer

In this case, the Θ and O complexity will be the same which will become apparent later on. Thus, we need to find the Big O complexity itself.

First, let us take the j loop. If the j loop executes **A** times, then –

$$A = (n-1) + \frac{(n-1)}{2} + \frac{(n-1)}{3} + \frac{(n-1)}{4} + \cdots + 1$$

$$A = (n-1)\left[1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n-1}\right]$$

$$A \approx n \ln n$$

We know that i loop will execute for n times. Therefore,

$$\boldsymbol{T(P) = O(n^2 \ln n)}$$

Therefore, **Option D** is the correct answer.

Question

Find out which of the following statements are true.

1. $1000 = O(1)$
2. $\frac{1}{n} = O(1)$
3. $\frac{1}{n} = \Theta(1)$
4. $n^x = O(n^y)$ if $0 < x < y$

Answer

For Statement 1, we need to satisfy the condition –

$$1000 \leq c * 1$$

Thus, with $c = 1000$, we can satisfy the condition thus making **Statement 1 as TRUE.** Similarly, for Statement 2, we need to satisfy the condition –

$$\frac{1}{n} \leq c * 1$$

We know that $n \geq 1$. Therefore, the above equation will be satisfied with $c = 1$ and hence **Statement 2 is TRUE.** Suppose we check the Big – Omega check for the same statement, we get –

$$\frac{1}{n} \geq c * 1$$

We can see that there is no constant value of c for which the above statement would return true. Thus, we get –

$$\frac{1}{n} \neq \Omega(1)$$

As a result, **Statement 3 is FALSE.** Finally, we know that for a function $f(n) = n^2$, we get –

$$f(n) = O(n^2) = O(n^{10}) = O(n^y) \qquad y \geq 2$$

Therefore, **Statement 4 is TRUE.** Finally, we can conclude that Statements **1,2 & 4** are **TRUE.**

Question

Find out which of the following statements are true.

1. $n^4 * 64^{\log_2 n} = O(n^{10})$
2. $n^8 * 16^{\log_4 n} = O(n^{10})$
3. $f(n) = O\left[\left(f(n)\right)^2\right]$
4. If $f(n) = O\big(g(n)\big)$, then we also have $2^{f(n)} = O\big(2^{g(n)}\big)$

Answer

For statements 1 and 2, the LHS can be written as –

$$n^4 * 64^{\log_2 n} = n^4 * 2^{6\log_2 n} = n^4 * 2^{\log_2(n^6)} = n^4 * n^6 = \boldsymbol{n^{10}}$$

$$n^8 * 16^{\log_4 n} = n^8 * 4^{\log_4 n^2} = n^8 * n^2 = \boldsymbol{n^{10}}$$

Therefore, both Statements 1 and 2 will boil down to –

$$n^{10} = O(n^{10})$$

Thus, **Statements 1 & 2 are TRUE.** For statement 3, let us assume –

$$f(n) = \frac{1}{n}$$

Then, the statement becomes –

$$\frac{1}{n} = O\left[\frac{1}{n^2}\right]$$

Therefore, we can say that **Statement 3 is FALSE.** Now, for Statement 4, let us assume –

$$f(n) = 2n \qquad g(n) = n$$

Thus, we get –

$$2^{2n} = O(2^n)$$

This is a clear case wherein **Statement 4 is FALSE.** Finally, we can conclude that **Statements 1 & 2 are TRUE.**

Question

Prove that $\log_2(n!) = \Theta(n \log_2 n)$

Answer

To prove Big – Theta, we need to prove both Big – O and Big – Omega. To prove Big – O, we get –

$$\log_2(n!) \leq c * n \log_2 n$$

$$\log_2 n + \log_2(n-1) + \log_2(n-2) + \cdots + 1 + 0 \leq c * (\log_2 n + \log_2 n + \cdots + \log_2 n)$$

Since the above statement is true, we can say that –

$$\log_2(n!) = O(n \log_2 n)$$

To prove Big – Omega, we get –

$$\log_2(n!) \geq c * n \log_2 n$$

We can see that –

$$n! = n * (n-1) * (n-2) * \ldots * 1$$

Therefore,

$$n! > n * (n-1) * (n-2) * \ldots * \left(\frac{n}{2}\right)$$

$$n! > \left(\frac{n}{2}\right)^{\left(\frac{n}{2}\right)}$$

Therefore, we get –

$$\log_2(n!) > \frac{n}{2} \log_2 \left(\frac{n}{2}\right)$$

Finally, we get –

$$\log_2(n!) = \Omega(n \log_2 n)$$

Therefore, since both Big – O and Big – Omega are being satisfied, we can say that –

$$\log_2(n!) = \Theta(n \log_2 n)$$

Question

Check for Big – O relation if $f(n) = n^{\sin n}$ and $g(n) = n^{2+\cos n}$

Answer

For Big – O relation to be satisfied, we need –

$$n^{\sin n} \leq c * n^{2+\cos n}$$

Since the base is the same, we can simply compare the exponents. We can see that –

$$\max(\sin n) = 1$$

$$\min(2 + \cos n) = 2$$

Therefore, we can conclude that for all values of n,

$$2 + \cos n > \sin n$$

Thus, we can conclude that –

$$n^{\sin n} = O(2 + \cos n)$$

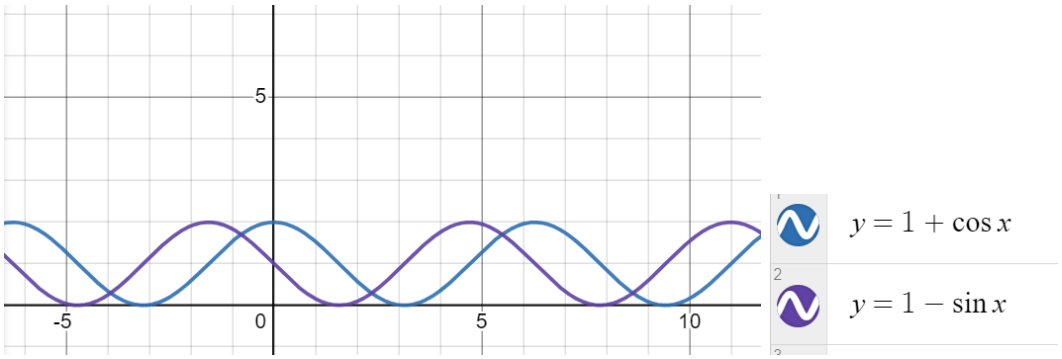
There is no value where $g(n) = f(n)$. Therefore, we can have Big – O and Small – O relations, but we can't have Big – Omega or Small – Omega relations. Additionally, we also can't have Big – Theta relation.

Question

Check for Big – O relation if $f(n) = n^{1-\sin n}$ and $g(n) = n^{1+\cos n}$

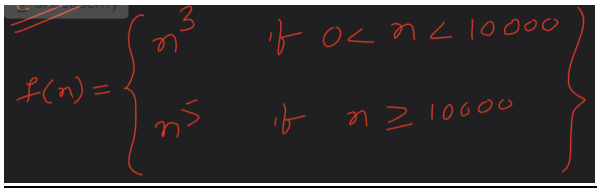
Answer

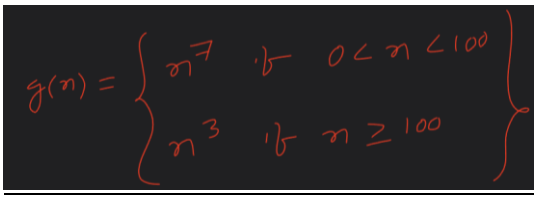
Let us draw the graph of $1 - \sin n$ and $1 + \cos n$.



Hence, we can see that there are cases where $g(n) > f(n)$ while there are also cases where $g(n) < f(n)$ and this is repeated over and over again. Hence, we can conclude that the functions are **not comparable.**

Question

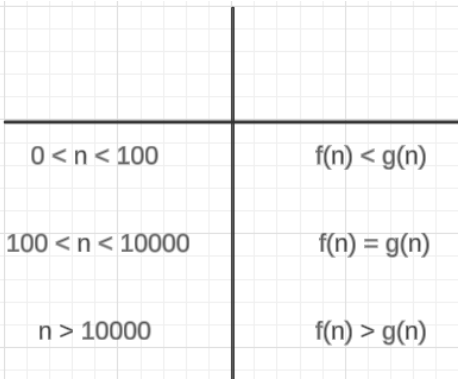




Find the relation between $f(n)$ and $g(n)$.

Answer

We can find the relation between the two functions as follows –



Therefore, we can write –

$$f(n) = \Omega\big(g(n)\big) \qquad n \geq 100$$

Question

Find the relation between the functions –

$$f(n) = log_2^*(\log_2 n)$$

$$g(n) = \log_2(log_2^* n)$$

Answer

To solve this, we first need to understand $log_2^* n$. This returns the number of times we need to apply $\log_2 n$ to get the final value as 1. For example, let us take –

$$n = 2^{2^2}$$

Then, we can get the value as 1 if we apply $\log_2 n$ a total of **3 times.** Therefore, we write –

$$log_2^* n = 3$$

Now, let us take the case of the following value of n –

$$n = 2^{2^{2^2}}$$

For this large value, we have –

$$f(n) = log_2^*(\log_2 n) = 4$$

$$g(n) = \log_2(log_2^* n) = 2.32$$

Therefore, we can see that $f(n) > g(n)$. Thus, we can conclude that –

$$f(n) = \Omega\big(g(n)\big)$$

Question

Find the true statements –

1. $\log_5 n = O(\log_{10} n)$
2. $n^2 \log_2 n = O(n^{2.01})$

Answer

For statement 1, we can write the LHS and RHS as follows –

$$\log_5 n = \frac{\log_2 n}{\log_2 5} = c_1 * \log_2 n$$

$$\log_{10} n = \frac{\log_2 n}{\log_2 10} = c_2 * \log_2 n$$

Therefore, we can see that –

$$\log_5 n = O(\log_{10} n) = \Omega(\log_{10} n) = \Theta(\log_{10} n)$$

For statement 2, we can compare as follows –

$$n^2 \log_2 n = n^{2.01}$$

$$\log_2 n = n^{0.01}$$

$$\log_2(\log_2 n) = 0.01 \log_2 n$$

Therefore, we can see that –

$$n^2 \log_2 n = O(n^{2.01})$$

Finally, we can see that **both statements are TRUE.**

Question

Solve the following recurrence relation –

$$T(n) = \begin{cases} 1 & n = 1 \\ T(n-1) + \log n & n > 1 \end{cases}$$

Answer

We can see that –

$$T(1) = 1$$

$$T(2) = T(1) + \log_2 2 = 1 + \log_2 2$$

$$T(3) = T(2) + \log_2 3 = 1 + \log_2 2 + \log_2 3$$

In short, we can write –

$$T(n) = 1 + \log_2 2 + \log_2 3 + \log_2 4 + \cdots + \log_2 n$$

$$T(n) = 1 + \log_2(2 * 3 * 4 * \ldots * n)$$

Therefore,

$$\boldsymbol{T(n) = 1 + \log_2(n!)}$$

We also know that as per **Stirling's approximation,**

$$\log_2(n!) = n \log_2 n$$

Therefore,

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

Also, we can write in general that –

$$2^n < n! < n^n$$

$$n < \log_2(n!) < n \log_2(n)$$

Thus, if this was a time recurrence relation, we can write –

$$TC = \Theta(n \log_2 n) = O(n \log_2 n) = \Omega(n)$$

Question

Solve the following recurrence relation –

$$T(n) = \begin{cases} 1 & n = 1 \\ T\left(\frac{n}{2}\right) + n & n > 1 \end{cases}$$

Answer

We can see that –

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

Therefore, we can see that –

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

The above is a GP series with $a = n$ and $r = \frac{1}{2}$. We can also see –

$$a + ar + ar^2 + \cdots + ar^{n-1} = \frac{a}{1-r}$$

Therefore, we get –

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

If we substitute $k = \log_2 n$, then we can write –

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

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

Question

Solve the following recurrence relation –

$$T(n) = \begin{cases} 10 & n = 0 \\ T(n-2) + n^2 & n > 0 \end{cases}$$

Answer

We can see that –

$$T(n) = T(n-2) + n^2 = T(n-4) + (n-2)^2 + n^2$$

Therefore,

$$T(n) = T(n-k) + (n-k+2)^2 + (n-k+4)^2 + (n-k+6)^2 + \cdots + n^2$$

If we substitute $k = n$, we get –

$$T(n) = T(0) + 2^2 + 4^2 + 6^2 + \cdots + n^2$$

$$T(n) = T(0) + \sum_{i=1}^{n/2} (2n)^2$$

$$\boldsymbol{T(n) = 10 + \frac{n(n+1)(n+2)}{6}}$$

Question

Solve the following recurrence relation –

$$T(n) = \begin{cases} 1 & n = 1 \\ T\left(\frac{n}{2}\right) + c & n > 1 \end{cases}$$

Answer

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

Therefore, we can say that –

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

Taking $k = \log_2 n$, we get –

$$T(n) = T(1) + c * \log_2 n$$

$$\boldsymbol{T(n) = 1 + c \log_2 n}$$

Question

Solve the following recurrence relation –

$$T(n) = 2T\left(\frac{n}{2}\right) + n \log_2 n$$

Answer

$$T(n) = 2T\left(\frac{n}{2}\right) + n \log_2 n = 4T\left(\frac{n}{2^2}\right) + n \log_2 \left(\frac{n}{2}\right) + n \log_2 n$$

Thus, we can write –

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + n\left[\log_2\left(\frac{n}{2^{k-1}}\right) + \log_2\left(\frac{n}{2^{k-2}}\right) + \cdots + \log_2\left(\frac{n}{2^0}\right)\right]$$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + n \log_2 \left(\frac{n^k}{1 * 2 * 2^2 \ldots * 2^{k-1}}\right)$$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + n\left[k \log_2 n - \frac{k(k-1)}{2} \log_2 2\right]$$

Substituting $k = \log_2 n$, we get –

$$T(n) = nT(1) + n\left[(\log_2 n)^2 - \frac{\log_2 n\,(\log_2 n - 1)}{2}\right]$$

$$\boldsymbol{T(n) = nT(1) + \frac{n \log_2 n}{2}[1 + \log_2 n]}$$

Question

Solve the following recurrence relation –

$$T(n) = \begin{cases} 2 & n = 2 \\ nT(\sqrt{n}) + n & n > 2 \end{cases}$$

Answer

$$T(n) = n\left[T(\sqrt{n}) + 1\right] = n\left[n^{\frac{1}{2}}\left[T\left(n^{\frac{1}{4}}\right) + 1\right]\right] = n\left[n^{\frac{1}{2}}\left[n^{\frac{1}{4}}\left[T\left(n^{\frac{1}{8}}\right) + 1\right]\right]\right]$$

Thus, we can write –

$$T(n) = n^{\left(1+\frac{1}{2}+\frac{1}{2^2}+\frac{1}{2^3}+\cdots+\frac{1}{2^{k-1}}\right)}\left[T\left(n^{\frac{1}{2^k}}\right) + 1\right]$$

We can see that –

$$1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{k-1}} = \frac{1\left(1 - \frac{1}{2^k}\right)}{1 - \frac{1}{2}} = 2 - \frac{1}{2^{k-1}}$$

Thus,

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

Substituting,

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

$$k = \log_2(\log_2 n)$$

Thus, we get –

$$\boldsymbol{T(n) = n^{2-\frac{1}{\log_2 n}} * 3}$$

Question

Solve the following recurrence relation –

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n-1) + n & n > 1 \end{cases}$$

Answer

We have –

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

Thus, we get –

$$T(n) = 2^kT(n-k) + nk - [1 + 2 + 3 + \cdots + (k-1)]$$

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

Substituting $k = n - 1$, we get –

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

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

$$\boldsymbol{T(n) = \frac{2^n + n^2 - 5n + 2}{2}}$$

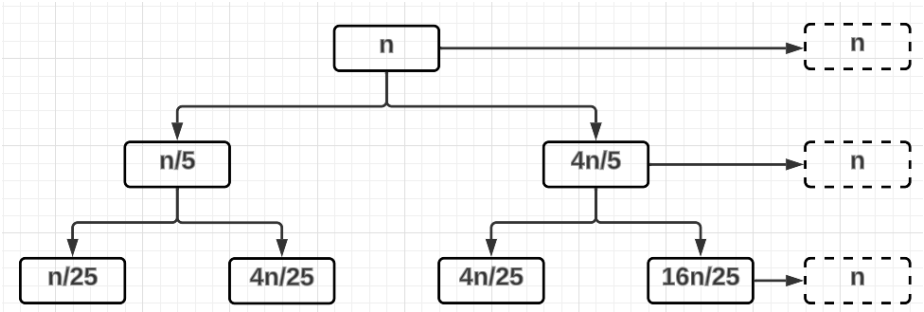
Question

Solve the recurrence relation –

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{4n}{5}\right) + n$$

Answer

We can draw the graph as follows –



Suppose we take the tree to $k1\ and\ k2$ levels in LHS and RHS, then –

$$LHS\ term = \frac{n}{5^{k1}}$$

$$RHS\ term = \frac{4^{k2} n}{5^{k2}}$$

Since they haven't explicitly mentioned any termination condition, we can assume a termination condition to solve this. So, let us assume that $T(1)$ is the termination condition. Therefore,

$$k1 = \log_5 n$$

$$k2 = \frac{\log_5 n}{1 - \log_5 4}$$

$$k2 > k1$$

For Upper Bound, we get –

$$T(n) \leq \frac{n \log_5 n}{1 - \log_5 4}$$

$$\boldsymbol{T(n) = O(n \log_5 n)}$$

For Lower Bound, we get –

$$T(n) \geq n \log_5 n$$

$$\boldsymbol{T(n) = \Omega(n \log_5 n)}$$

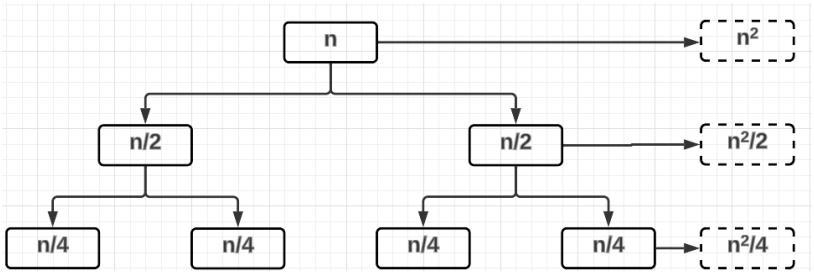
Question

Solve the recurrence relation –

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

Answer

We can draw the graph as follows –



We can see that for this case, the LHS and RHS height will be the same. So, for k levels, we get –

$$LHS\ term = RHS\ term = \frac{n}{2^k}$$

Since they haven't explicitly mentioned any termination condition, we can assume a termination condition to solve this. So, let us assume that $T(1)$ is the termination condition. Therefore,

$$k = \log_2 n$$

We can get –

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

$$T(n) = 2n^2 - n$$

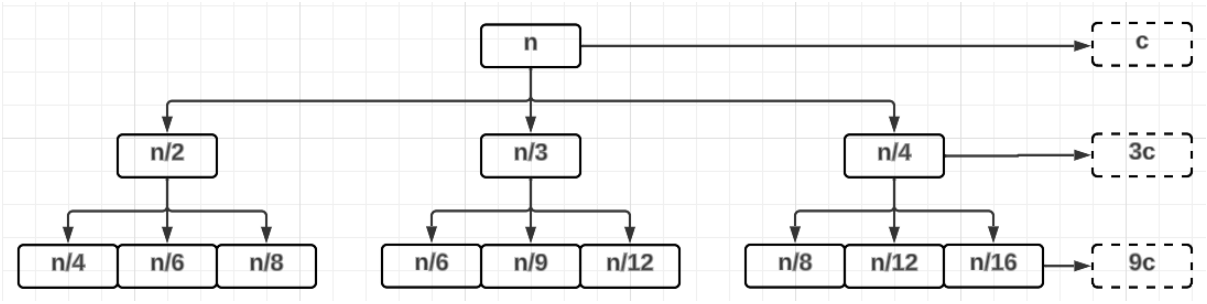
$$\boldsymbol{T(n) = \Theta(n^2)}$$

Question

Solve the recurrence relation –

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{3}\right) + T\left(\frac{n}{4}\right) + c$$

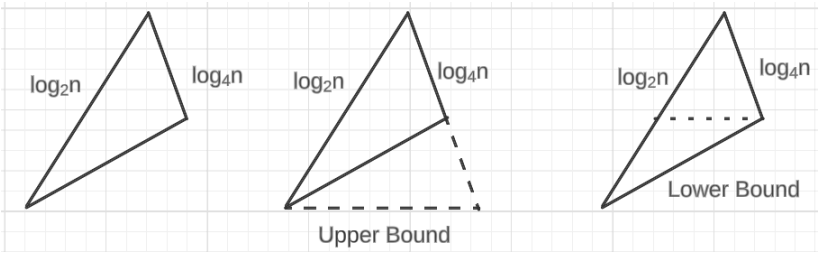
Answer



From the above tree, we can calculate the LHS and RHS height of the tree as follows –

$$LHS\ height = \log_2 n$$

$$RHS\ height = \log_4 n$$



For Upper Bound,

$$T(n) \leq c + 3c + 3^2c + 3^3c + \cdots + 3^{\log_2 n}c$$

$$\boldsymbol{T(n) = O(n^{1.585})}$$

For Lower Bound,

$$T(n) \geq c + 3c + 3^2c + 3^3c + \cdots + 3^{\log_4 n}c$$

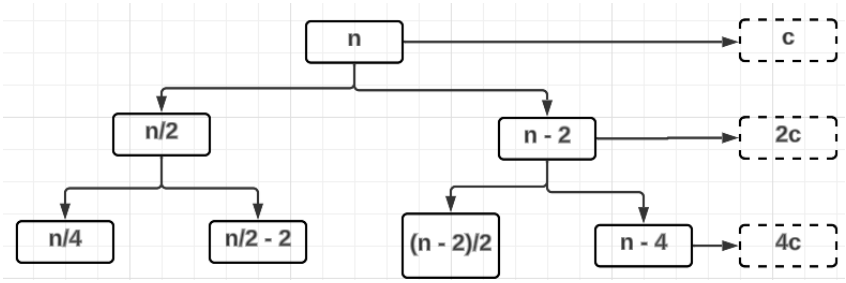
$$\boldsymbol{T(n) = \Omega(n^{0.79})}$$

Question

Solve the recurrence relation –

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

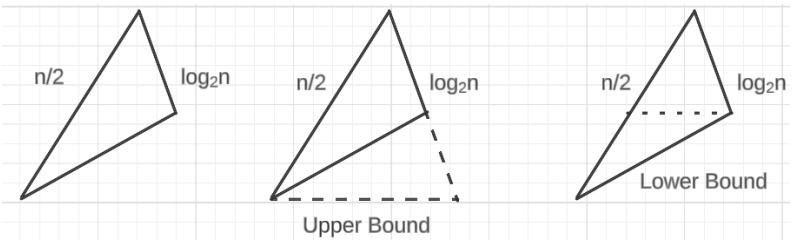
Answer



From the above tree, we can calculate the LHS and RHS height of the tree as follows –

$$LHS\ height = \log_2 n$$

$$RHS\ height = \frac{n}{2}$$



For Upper Bound,

$$T(n) \leq c + 2c + 2^2c + 2^3c + \cdots + 2^{\frac{n}{2}}c$$

$$\boldsymbol{T(n) = O\left(\left(\sqrt{2}\right)^n\right)}$$

For Lower Bound,

$$T(n) \geq c + 2c + 2^2c + 2^3c + \cdots + 2^{\log_2 n}c$$

$$\boldsymbol{T(n) = \Omega(n)}$$

Question

Solve the following recurrence relation –

$$T(n) = 16T\left(\frac{n}{4}\right) + n^{25}$$

Answer

Here, $a = 16$; $b = 4$; $f(n) = n^{25}$. Now,

$$n^{\log_b a} = n^2$$

We can see that –

$$n^{25} = \Omega(n^2 + \epsilon)$$

$$f(n) = \Omega\left(n^{\log_b a} + \epsilon\right)$$

Thus, as per the master theorem we can say that –

$$\boldsymbol{T(n) = \Theta\left(n^{25}\right)}$$

Question

Solve the following recurrence relation –

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

Answer

Here, $a = 2$; $b = 2$; $f(n) = n$. Now,

$$n^{\log_b a} = n$$

We can see that –

$$n = \Theta(n * (\log n)^0)$$

$$f(n) = \Theta\left(n^{\log_b a} * (\log n)^{\epsilon}\right)$$

Thus, as per the master theorem we can say that –

$$\boldsymbol{T(n) = \Theta(n \log n)}$$

Question

Solve the following recurrence relation –

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

Answer

Here, $a = 2$; $b = 2$; $f(n) = n \log n$. Now,

$$n^{\log_b a} = n$$

We can see that –

$$n \log n = \Theta(n * (\log n)^1)$$

$$f(n) = \Theta\left(n^{\log_b a} * (\log n)^{\epsilon}\right)$$

Thus, as per the master theorem we can say that –

$$\boldsymbol{T(n) = \Theta\left(n(\log n)^2\right)}$$

Question

Solve the recurrence relation –

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

Answer

We can see that –

$$n^{\log_2 2} = n$$

Thus, we have –

$$f(n) = \Omega(n + \epsilon)$$

Therefore, we have –

$$\boldsymbol{T(n) = \Theta(n^2)}$$

Question

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

Answer

$$n^{\log_a b} = n^{\log_2 2^n} = n^n$$

We can see that –

$$f(n) = \Theta(n^n * (\log n)^0)$$

Therefore,

$$\boldsymbol{T(n) = \Theta(n^n \log n)}$$

Question

$$T(n) = T(\sqrt{n}) + c$$

Answer

Let us assume the following –

$$n = 2^k$$

$$T(2^k) = S(k)$$

Thus, we can write –

$$S(k) = S\left(\frac{k}{2}\right) + c$$

Here, we have $a = 1$; $b = 2$. Thus,

$$k^{\log_2 1} = 1$$

Thus, we get –

$$f(k) = \Theta(1 * (\log k)^0)$$

Therefore, we get –

$$S(k) = \Theta(\log k)$$

$$T(2^{\log_2 n}) = \Theta(\log(\log_2 n))$$

$$\boldsymbol{T(n) = \Theta(\log(\log_2 n))}$$

Question

$$T(n) = 2T\left(\sqrt{n}\right) + \log_2 n$$

Answer

Let us assume that –

$$n = 2^k$$

Thus, we get –

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

Also, let us assume that

$$T\left(2^k\right) = S(k)$$

Therefore,

$$S(k) = 2S\left(\frac{k}{2}\right) + k$$

Here, $a = 2$; $b = 2$. Thus, we get –

$$k^{\log_b a} = k$$

Thus, we get –

$$f(k) = \Theta(k * (\log k)^0)$$

Therefore, we get –

$$S(k) = \Theta(k \log k)$$

$$\boldsymbol{T(n) = \Theta(\log_2 n * \log_2(\log_2 n))}$$

QUESTION BANK

Question 1

Q Let w(n) and A(n) denote respectively, the worst case and average case running time of an algorithm executed on an input of size n. which of the following is ALWAYS TRUE? **(Gate-2012) (1- Marks)**

(A) A(n) = Ω(W(n)) **(B)** A(n) = Θ(W(n))

(C) A(n) = O(W(n)) **(D)** A(n) = o(W(n))

Question 2

Q Let f(n), g(n) and h(n) be functions defined for positive inter such that f(n) = O(g(n)), g(n) ≠ O(f(n)), g(n) = O(h(n)), and h(n) = O(g(n)).
Which one of the following statements is FALSE? **(Gate-2004) (2 Marks)**

(A) f(n) + g(n) = O(h(n)) + h(n)) **(C)** h(n) ≠ O(f(n))

(B) f(n) = O(h(n)) **(D)** f(n)h(n) ≠ O(g(n)h(n))

Question 3

Consider the following two functions:

$$g_1(n) = \begin{cases} n^3 \text{ for } 0 \le n \le 10,000 \\ n^2 \text{ for } n \ge 10,000 \end{cases}$$

$$g_2(n) = \begin{cases} n \text{ for } 0 \le n \le 100 \\ n^3 \text{ for } n > 100 \end{cases}$$

Which of the following is true?

A. $g_1(n)$ is $O(g_2(n))$

B. $g_1(n)$ is $O(n^3)$

C. $g_2(n)$ is $O(g_1(n))$

D. $g_2(n)$ is $O(n)$

Question 4

Q Consider the following three claims
I (n + k)^m = (n^m), where k and m are constants
II 2^(n + 1) = 0(2^n)
III 2^(2n + 1) = 0(2^n)
Which of these claims are correct? (GATE-2003) (1 Marks)
(A) I and II **(B)** I and III **(C)** II and III **(D)** I, II and III

Question 5

Q Two alternative packages A and B are available for processing a database having 10^k records. Package A requires $0.0001n^2$ time units and package B requires $10n\log_{10}n$ time units to process n records. What is the smallest value of k for which package B will be preferred over A? **(Gate-2010) (1 Marks)**

a) 12

c) 6

b) 10

d) 5

Question 6

Q Let $f(n) = n$ and $g(n) = n^{(1+\sin n)}$, where n is a positive integer. Which of the following statements is/are correct? **(Gate-2015) (2 Marks)**

I. $f(n) = O(g(n))$

II. $f(n) = \Omega(g(n))$

(A) Only I

(C) Both I and II

(B) Only II

(D) Neither I nor II

Question 7

Q Consider the following three functions.

$f_1 = 10^n$

$f_2 = n^{\log n}$

$f_3 = n^{\sqrt{n}}$

Which one of the following options arranges the functions in the increasing order of asymptotic growth rate? **(Gate 2021)**

(a) f_3, f_2, f_1

(c) f_1, f_2, f_3

(b) f_2, f_1, f_3

(d) f_2, f_3, f_1

Question 8

Q Consider the following functions from positives integers to real numbers $10, \sqrt{n}, n, \log_2 n, 100/n$. The CORRECT arrangement of the above functions in increasing order of asymptotic complexity is: **(GATE-2017) (1 Marks)**

(A) $\log_2 n, 100/n, 10, \sqrt{n}, n$

(C) $10, 100/n, \sqrt{n}, \log_2 n, n$

(B) $100/n, 10, \log_2 n, \sqrt{n}, n$

(D) $100/n, \log_2 n, 10, \sqrt{n}, n$

Question 9

Q Let f(n) = n^2Logn and g(n) = n $(logn)^{10}$ be two positive functions of n. Which of the following statements is correct? **(Gate-2001) (1 Marks)**

(A) f(n) = O(g(n)) and g(n) != O(f(n))

(B) f(n) != O(g(n)) and g(n) = O(f(n))

(C) f(n) = O(g(n)) but g(n) = O(f(n))

(D) f(n) != O(g(n)) but g(n) != O(f(n))

Question 10

Q The increasing order of following functions in terms of asymptotic complexity is **(Gate-2015) (2 Marks)**

$$f_1(n) = n^{0.999999}\log n$$
$$f_2(n) = 10000000n$$
$$f_3(n) = 1.000001^n$$
$$f_4(n) = n^2$$

(A) $f_1(n)$; $f_4(n)$; $f_2(n)$; $f_3(n)$
(B) $f_1(n)$; $f_2(n)$; $f_3(n)$; $f_4(n)$;
(C) $f_2(n)$; $f_1(n)$; $f_4(n)$; $f_3(n)$
(D) $f_1(n)$; $f_2(n)$; $f_4(n)$; $f_3(n)$

Question 11

Q Consider the following functions:
f(n) = 2^n g(n) = n! h(n) = n^{logn}
Which of the following statements about the asymptotic behaviour of f(n), g(n), and h(n) is true? **(Gate-2008) (2 Marks)**

(A) f(n) = O(g(n)); g(n) = O(h(n))
(B) f(n) = Ω(g(n)); g(n) = O(h(n))
(C) g(n) = O(f(n)); h(n) = O(f(n))
(D) h(n) = O(f(n)); g(n) = Ω(f(n))

Question 12

Q Arrange the following functions in increasing asymptotic order: **(GATE-2008) (1 Marks)**
a) $n^{1/3}$ **b)** e^n **c)** $n^{7/4}$ **d)** $n\log^9 n$ **e)** 1.0000001^n

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

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

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

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

Question 13

Q Which of the given options provides the increasing order of asymptotic complexity of functions f1, f2, f3 and f4? **(Gate-2011) (2 Marks)**

$f_1(n) = 2^n$ $f_2(n) = n^{(3/2)}$ $f_3(n) = nLogn$ $f_4(n) = n^{(Logn)}$

(A) f_3, f_2, f_4, f_1

(C) f_2, f_3, f_1, f_4

(B) f_3, f_2, f_1, f_4

(D) f_2, f_3, f_4, f_1

Question 14

Q The equality above remains correct if X is replaced by **(Gate-2015) (1 Marks)**

Consider the equality $\sum_{i=0}^{n} i^3 = X$ and the following choices for X

I. $\Theta(n^4)$

II. $\Theta(n^5)$

III. $O(n^5)$

IV. $\Omega(n^3)$

(A) Only I
(B) Only II
(C) I or III or IV but not II
(D) II or III or IV but not I

Question 15

Q For constants a≥1 and b>1, consider the following recurrence defined on the non-negative integers:
$T(n) = a \cdot T(n/b) + f(n)$
Which one of the following options is correct about the recurrence T(n)? **(GATE 2021) (2 MARKS)**

A. If $f(n)$ is $n\log_2(n)$, then $T(n)$ is $\Theta(n\log_2(n))$

B. If $f(n)$ is $\frac{n}{\log_2(n)}$, then $T(n)$ is $\Theta(\log_2(n))$

C. If $f(n)$ is $O(n^{\log_b(a)-\epsilon})$ for some $\epsilon > 0$, then $T(n)$ is $\Theta(n^{\log_b(a)})$

D. If $f(n)$ is $\Theta(n^{\log_b(a)})$, then $T(n)$ is $\Theta(n^{\log_b(a)})$

Question 16

Q The running time of an algorithm is represented by the following recurrence relation:

if n <= 3 then T(n) = n

else T(n) = T(n/3) + cn

Which one of the following represents the time complexity of the algorithm? **(Gate-2009) (2 Marks)**

(A)Θ(n) **(B)**Θ(n log n) **(C)**Θ(n^2) **(D)**Θ(n^2log n)

Question 17

Q Suppose T(n) = 2T (n/2) + n, T(0) = T(1) = 1

Which one of the following is FALSE? **(Gate-2005) (2 Marks)**

(A) T(n) = $O(n^2)$ **(C)** T(n) = $\Omega(n^2)$

(B) T(n) = θ(n log n) **(D)** T(n) = O(n log n)

Question 18

Q Let T(n) be a function defined by the recurrence

T(n) = 2T(n/2) + √n for n ≥ 2 and T(1) = 1

Which of the following statements is TRUE? **(Gate-2005) (2 Marks)**

(A) T(n) = θ(log n) **(C)** T(n) = θ(n)

(B) T(n) = θ(√n) **(D)** T(n) = θ(n log n)

Question 19

Q Which one of the following correctly determines the solution of the recurrence relation with T(1) = 1? **(Gate-2014) (1 Marks)**

T(n) = 2T(n/2) + Logn

a) Θ(n) **c)** Θ(n*n)

b) Θ(nLogn) **d)** Θ(log n)

Question 20

Q The solution to the recurrence equation $T(2^k) = 3\,T(2^{k-1}) + 1$, $T(1) = 1$, is: **(Gate-2002) (2 Marks)**

(A) 2^k

(C) $3^{\log_2 k}$

(B) $(3^{k+1} - 1)/2$

(D) $2^{\log_3 k}$

Question 21

Q The recurrence equation

$T(1) = 1$

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

evaluates to **(Gate-2004) (2 Marks)**

(A) $2^{n+1} - n - 2$

(C) $2^{n+1} - 2n - 2$

(B) $2^n - n$

(D) $2^n + n$

Question 22

Q The recurrence relation capturing the optimal execution time of the Towers of Hanoi problem with n discs is **(Gate-2012) (1 Marks)**

(A) $T(n) = 2T(n - 2) + 2$

(C) $T(n) = 2T(n/2) + 1$

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

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

Question 23

Q Consider the following recurrence:

$T(n)=2T(\sqrt{n})+1,\ T(1)=1$

Which one of the following is true? **(Gate-2006) (2 Marks)**

a) $T(n)=\Theta(\log\log n)$

c) $T(n)=\Theta(\sqrt{n})$

b) $T(n)=\Theta(\log n)$

d) $T(n)=\Theta(n)$

Question 24

For parameters a and b, both of which are $\omega(1)$, $T(n) = T(n^{1/a}) + 1$, and $T(b) = 1$.

Then $T(n)$ is **(Gate-2020) (1 Marks)**

$\Theta(\log_a \log_b n)$

$\Theta(\log_{ab} n)$

$\Theta(\log_b \log_a n)$

$\Theta(\log_2 \log_2 n)$

Question 25

Q Which one of the following is the tightest upper bound that represents the number of swaps required to sort n numbers using selection sort? **(Gate-2013) (1 Marks)**

A) O(log n)

C) O(n log n)

B) O(n)

D) $O(n^2)$

Question 26

Q The number of swappings needed to sort the numbers 8, 22, 7, 9, 31, 5, 13 in ascending order, using bubble sort is **[Asked in Amcat 2018]**

a) 11

c) 13

b) 12

d) 10

Question 27

Q The given array is arr = {1,2,3,4,5}. (bubble sort is implemented with a flag variable)The number of iterations in selection sort and bubble sort respectively are _____ **[Asked in Infosys 2021]**

a) 5 and 4

c) 0 and 4

b) 1 and 4

d) 4 and 1

Question 28

Q) The number of swapping needed to sort numbers 8,22,7,9,31,19,5,13 in ascending order using bubble sort is ? **[Asked in Accenture]**

a) 12

c) 14

b) 11

d) 13

Question 29

Q) If the array A contains the items 10, 4, 7, 23, 67, 12 and 5 in that order, what will be the resultant array A after third pass of insertion sort? **[Asked in L&T Infotech (LTI) 2021]**

a) 67, 12, 10, 5, 4, 7, 23

b) 4, 7, 10, 23, 67, 12, 5

c) 4, 5, 7, 67, 10, 12, 23

d) 10, 7, 4, 67, 23, 12, 5

Question 30

Q) Consider an array of length 5, arr[5] = {9,7,4,2,1}. What are the steps of insertions done while running insertion sort on the array? **[Asked in Amcat 2020]**

a) 7 9 4 2 1 4 7 9 2 1 2 4 7 9 1 1 2 4 7 9

b) 9 7 4 1 2 9 7 1 2 4 9 1 2 4 7 1 2 4 7 9

c) 7 4 2 1 9 4 2 1 9 7 2 1 9 7 4 1 9 7 4 2

d) 7 9 4 2 1 2 4 7 9 1 4 7 9 2 1 1 2 4 7 9

Question 31

Q) An array contains the following elements in order: 7 6 12 30 18. Insertion sort is used to sort the array in ascending order. How many times will an insertion be made? **[AMCAT]**

a) 2 **c)** 4

b) 3 **d)** 5

Question 32

Q What is the running time of an insertion sort algorithm if the input is pre-sorted? **[Asked in IBM 2020]**

a) $O(N^2)$ **c)** O(N)

b) O(N log N) **d)** O(M log N)

Question 33

Q Assume that a Merge_Sort algorithm in the worst case takes 30 seconds for an input of size 64. Which of the following most closely approximates the maximum input size of a problem that can be solved in 6 minutes? **(Gate-2015) (1 Marks)** **[Asked in Capgemini 2021]**

a) 256 **c)** 1024

b) 512 **d)** 2048

Question 34

Q) Given two sorted lists of size m and n respectively . The number of comparisons needed in the worst case by the merge sort algorithm will be? **[Asked in Accenture]**

a) min(m , n) **c)** max(m , n)

b) m n **d)** m+n-1

Question 35

Q Suppose P, Q, R, S, T are sorted sequences having lengths 20, 24, 30, 35, 50 respectively. They are to be merged into a single sequence by merging together two sequences at a time. The number of comparisons that will be needed in the worst case by the optimal algorithm for doing this is ____. **[Asked in Wipro NLTH 2020]**

a) 500

c) 450

b) 358

d) 259

Question 36

Q What is the auxiliary space complexity of merge sort? **[Asked in Hexaware 2020]**

a) O(1)

b) O(log n)

c) O(n)

d) O(n log n)

Question 37

Q An operator delete(i) for a binary heap data structure is to be designed to delete the item in the i-th node. Assume that the heap is implemented in an array and i refers to the i-th index of the array. If the heap tree has depth d (number of edges on the path from the root to the farthest leaf), then what is the time complexity to re-fix the heap efficiently after the removal of the element? **(Gate-2016) (1 Marks)**

a) O(1)

c) $O(2^d)$ but not O(d)

b) O(d) but not O(1)

d) $O(d2^d)$ but not $O(2^d)$

Question 38

Q Consider a complete binary tree where the left and the right subtrees of the root are max-heaps. The lower bound for the number of operations to convert the tree to a heap is **(GATE-2015) (1 Marks)**

a) Ω(logn)

c) Ω(nlogn)

b) Ω(n)

d) $\Omega(n^2)$

Question 39

Q Consider the process of inserting an element into a Max Heap, where the Max Heap is represented by an array. Suppose we perform a binary search on the path from the new leaf to the root to find the position for the newly inserted element, the number of comparisons performed is: **(Gate-2007) (1 Marks)**

a) $\theta(\log_2 n)$

c) $\theta(n)$

b) $\theta(\log_2 \log_2 n)$

d) $\theta(n \log_2 n)$

Question 40

Q Sorting compares two adjoining values and exchange them if they are not in proper order. **[Asked in TCS NQT 2020]**

a) Heap

c) Insertion

b) Selection

d) Bubble

Question 41

Q) Which of the following sorting algorithms yield approximately the same worst-case? and average-case Running time behavior in O (n log n)? **[AMCAT]**

a) Bubble sort and Selection sort

b) Heap sort and Merge sort

c) Quick sort and Radix sort

d) Tree sort and Median-of-3 Quick sort

Question 42

Q Suppose we are sorting an array of eight integers using quicksort, and we have just finished the first partitioning with the array looking like this:

2 5 1 7 9 12 11 10

Which statement is correct?

(A) The pivot could be either the 7 or the 9
(B) The pivot could be the 7, but it is not the 9
(C) The pivot is not the 7, but it could be the 9
(D) Neither the 7 nor the 9 is the pivot.

Question 43

Q Consider the Quicksort algorithm. Suppose there is a procedure for finding a pivot element which splits the list into two sub-lists each of which contains at least one-fifth of the elements. Let T(n) be the number of comparisons required to sort n elements. Then **(Gate-2008) (2 Marks)**

(A) $T(n) <= 2T(n/5) + n$

(B) $T(n) <= T(n/5) + T(4n/5) + n$

(C) $T(n) <= 2T(4n/5) + n$

(D) $T(n) <= 2T(n/2) + n$

Question 44

Q You have an array of n elements. Suppose you implement quicksort by always choosing the central element of the array as the pivot. Then the tightest upper bound for the worst-case performance is **(Gate-2014) (1 Marks)**

a) $O(n^2)$

b) O(nLogn)

c) Theta(nLogn)

d) $O(n^3)$

Question 45

Q An array of 25 distinct elements is to be sorted using quicksort. Assume that the pivot element is chosen uniformly at random. The probability that the pivot element gets placed in the worst possible location in the first round of partitioning (rounded off to 2 decimal places) is _____. **(Gate-2019) (1 Marks) [Asked in CoCubes 2020]**

Question 46

Q The worst-case running times of Insertion sort, Merge sort and Quick sort, respectively, are: **(Gate-2016) (1 Marks)**

a) $\Theta(n \log n)$, $\Theta(n \log n)$ and $\Theta(n^2)$

b) $\Theta(n^2)$, $\Theta(n^2)$ and $\Theta(n \text{ Log } n)$

c) $\Theta(n^2)$, $\Theta(n \log n)$ and $\Theta(n \log n)$

d) $\Theta(n^2)$, $\Theta(n \log n)$ and $\Theta(n^2)$

Question 47

Q Assume that the algorithms considered here sort the input sequences in ascending order. If the input is already in ascending order, which of the following are TRUE? **(Gate-2016) (1 Marks)**

I. Quicksort runs in $\Theta(n^2)$ time
II. Bubble sort runs in $\Theta(n^2)$ time
III. Merge_Sort runs in $\Theta(n)$ time
IV. Insertion sort runs in $\Theta(n)$ time

a) I and II only
b) I and III only
c) II and IV only
d) I and IV only

Question 48

Q An unordered list contains n distinct elements. The number of comparisons to find an element in this list that is neither maximum nor minimum is **(Gate-2015) (1 Marks)**

(A) $\Theta(n \log n)$

(C) $\Theta(\log n)$

(B) $\Theta(n)$

(D) $\Theta(1)$

Question 49

Q Match the algorithms with their time complexities: **(Gate-2017) (1 Marks)**

Algorithm	Time complexity
(P) Towers of Hanoi with n disks	(i) $\Theta(n^2)$
(Q) Binary search given n sorted numbers	(ii) $\Theta(n \log n)$
(R) Heap sort given n numbers at the worst case	(iii) $\Theta(2^n)$
(S) Addition of two $n \times n$ matrices	(iv) $\Theta(\log n)$

a) P-> (iii), Q -> (iv), R -> (i), S -> (ii)

b) P-> (iv), Q -> (iii), R -> (i), S -> (ii)

c) P-> (iii), Q -> (iv), R -> (ii), S -> (i)

d) P-> (iv), Q -> (iii), R -> (ii), S -> (i)

Question 50

Q Which of the following sorting algorithms has the lowest worst-case complexity?

(A) Merge Sort

(C) Quick Sort

(B) Bubble Sort

(D) Selection Sort

Question 51

Q Which of the following is not a stable sorting algorithm in its typical implementation.

(A) Insertion Sort

(C) Quick Sort

(B) Merge Sort

(D) Bubble Sort

Question 52

Q Which of the following sorting algorithms in its typical implementation gives best performance when applied on an array which is sorted or almost sorted (maximum 1 or two elements are misplaced).

(A) Quick Sort

(C) Merge Sort

(B) Heap Sort

(D) Insertion Sort

Question 53

Q Consider a situation where swap operation is very costly. Which of the following sorting algorithms should be preferred so that the number of swap operations are minimized in general?

(A) Heap Sort (C) Insertion Sort

(B) Selection Sort (D) Merge Sort

Question 54

Q You have to sort 1 GB of data with only 100 MB of available main memory. Which sorting technique will be most appropriate?

(A) Heap sort (C) Quick sort

(B) Merge sort (D) Insertion sort

Question 55

Q Which sorting algorithm will take least time when all elements of input array are identical? Consider typical implementations of sorting algorithms.

(A) Insertion Sort (C) Merge Sort

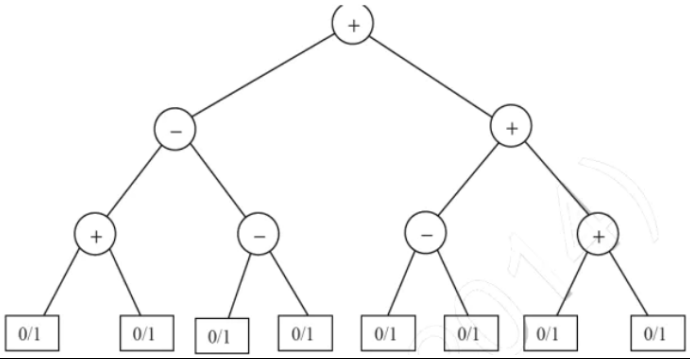
(B) Heap Sort (D) Selection Sort

Question 56

Q The minimum number of comparisons required to find the minimum and the maximum of 100 numbers is _____. **(Gate-2014) (1 Marks)**

Question 57

Q Consider the expression tree shown. Each leaf represents a numerical value, which can either be 0 or 1. Over all possible choices of the values at the leaves, the maximum possible value of the expression represented by the tree is ___. **(Gate-2014) (2 Marks)**



Question 58

Q A message is made up entirely of characters from the set $X = \{P, Q, R, S, T\}$. The table of probabilities for each of the characters is shown below: If a message of 100 characters over X is encoded using Huffman coding, then the expected length of the encoded message in bits is _____. **(Gate-2017) (2 Marks)**

Character	Probability
P	0.22
Q	0.34
R	0.17
S	0.19
T	0.08
Total	1.00

Question 59

Q Consider the string abbccddeee. Each letter in the string must be assigned a binary code satisfying the following properties:

- For any two letters, the code assigned to one letter must not be a prefix of the code assigned to the other letter.
- For any two letters of the same frequency, the letter which occurs earlier in the dictionary order is assigned a code whose length is at most the length of the code assigned to the other letter.

Among the set of all binary code assignments which satisfy the above two properties, what is the minimum length of the encoded string? **(GATE 2021) (2 MARKS)**

(A) 21 **(C)** 25

(B) 23 **(D)** 30

Question 60

Q Suppose the letters a, b, c, d, e, f has probabilities 1/2, 1/4, 1/8, 1/16, 1/32, 1/32 respectively. Which of the following is the Huffman code for the letter a, b, c, d, e, f? **(Gate - 2007) (2 Marks)**

(A) 0, 10, 110, 1110, 11110, 11111

(B) 11, 10, 011, 010, 001, 000

(C) 11, 10, 01, 001, 0001, 0000

(D) 110, 100, 010, 000, 001, 111

Question 61

Q Suppose the letters a, b, c, d, e, f has probabilities 1/2, 1/4, 1/8, 1/16, 1/32, 1/32 respectively. What is the average length of Huffman codes? **(Gate - 2007) (2 Marks)**

(A) 3 **(B)** 2.1875 **(C)** 2.25 **(D)** 1.9375

Question 62

Q The characters a to h have the set of frequencies based on the first 8 Fibonacci numbers as follows
a : 1, b : 1, c : 2, d : 3, e : 5, f : 8, g : 13, h : 21
A Huffman code is used to represent the characters. What is the sequence of characters corresponding to the following code? **(Gate-2006) (2 Marks)**

110111100111010

(A) fdheg
(C) dchfg
(B) ecgdf
(D) fehdg

Question 63

Q Suppose P, Q, R, S, T are sorted sequences having lengths 20, 24, 30, 35, 50 respectively. They are to be merged into a single sequence by merging together two sequences at a time. The number of comparisons that will be needed in the worst case by the optimal algorithm for doing this is ____. **(Gate-2014) (2 Marks)**

Question 64

Q Consider the weights and values of items listed below. Note that there is only one unit of each item. The task is to pick a subset of these items such that their total weight is no more than 11 Kgs and their total value is maximized. Moreover, no item may be split. The total value of items picked by an optimal algorithm is denoted by V_{opt}. A greedy algorithm sorts the items by their value-to-weight ratios in descending order and packs them greedily, starting from the first item in the ordered list. The total value of items picked by the greedy algorithm is denoted by V_{greedy}. The value of $V_{opt} - V_{greedy}$ is _____ . **(Gate-2018) (2 Marks)**

Item number	Weight (in Kgs)	Value (in Rupees)
1	10	60
2	7	28
3	4	20
4	2	24

Question 65

Q We are given 9 tasks T1, T2.... T9. The execution of each task requires one unit of time. We can execute one task at a time. Each task Ti has a profit Pi and a deadline di Profit Pi is earned if the task is completed before the end of the dith unit of time.

Task	T1	T2	T3	T4	T5	T6	T7	T8	T9
Profit	15	20	30	18	18	10	23	16	25
Deadline	7	2	5	3	4	5	2	7	3

Are all tasks completed in the schedule that gives maximum profit? (Gate-2005) (2 Marks)
(A) All tasks are completed
(B) T1 and T6 are left out
(C) T1 and T8 are left out
(D) T4 and T6 are left out

Question 66

Q The following are the starting and ending times of activities A, B, C, D, E, F, G and H respectively in chronological order: "a_s b_s c_s a_e d_s c_e e_s f_s b_e d_e g_s e_e f_e h_s g_e h_e" Here, x_s denotes the starting time and x_e denotes the ending time of activity X. W need to schedule the activities in a set of rooms available to us. An activity can be scheduled in a room only if the room is reserved for the activity for its entire duration. What is the minimum number of rooms required? **(Gate-2003) (2 Marks)**

(A) 3 **(B)** 4 **(C)** 5 **(D)** 6

Question 67

Q Consider two strings A = "qpqrr" and B = "pqprqrp". Let x be the length of the longest common subsequence (not necessarily contiguous) between A and B and let y be the number of such longest common subsequences between A and B. Then x + 10y = ___ **(Gate-2014) (2 Marks)**

Question 68

Q A sub-sequence of a given sequence is just the given sequence with some elements (possibly none or all) left out. We are given two sequences X[m] and Y[n] of lengths m and n respectively, with indexes of X and Y starting from 0. We wish to find the length of the longest common sub-sequence(LCS) of X[m] and Y[n] as l(m, n), where an incomplete recursive definition for the function l(i, j) to compute the length of The LCS of X[m] and Y[n] is given below **(Gate-2009) (2 Marks)**

l(i,j) = 0, if either i=0 or j=0

= $expr_1$, if i, j > 0 and X[i-1] = Y[j-1]

= $expr_2$, if i, j > 0 and X[i-1] != Y[j-1]

			B	D	C	A	B	A
	0	1	2	3	4	5	6
0							
A 1							
B 2							
C 3							
B 4							
D 5							
A 6							
B 7							

(A) $expr_1 \equiv l(i-1, j) + 1$

(B) $expr_1 \equiv l(i, j-1)$

(C) $expr_2 \equiv \max(l(i-1, j), l(i, j-1))$

(D) $expr_2 \equiv \max(l(i-1,j-1),l(i, j))$

Question 69

Q Consider the data given in the previous question. The values of l(i, j) could be obtained by dynamic programming based on the correct recursive definition of l(i, j) of the form given above, using an array L[M, N], where M = m+1 and N =n+1, such that L[i, j] = l(i, j). Which one of the following statements would be TRUE regarding the dynamic programming solution for the recursive definition of l(i, j)? **(Gate-2009) (2 Marks)**

(A) All elements L should be initialized to 0 for the values of l(i, j) to be properly computed

(B) The values of l(i, j) may be computed in a row major order or column major order of L(M,N)

(C) The values of l(i, j) cannot be computed in either row major order or column major order of L(M,N)

(D) L[p, q] needs to be computed before L[r, s] if either p < r or q < s.

Question 70

Q Consider the following steps:

S_1: Characterize the structure of an optimal solution

S_2: Compute the value of an optimal solution in bottom-up fashion

Which of the following step(s) is/are common to both dynamic programming and greedy algorithms?**(NET 2019 June)**

(A) Only S_1

(B) Only S_2

(C) Both S_1 and S_2

(D) Neither S_1 nor S_2

Question 71

Q Assume that multiplying a matrix G_1 of dimension p × q with another matrix G_2 of dimension q × r requires pqr scalar multiplications. Computing the product of n matrices $G_1G_2G_3$ ..... G_n can be done by parenthesizing in different ways. Define G_iG_{i+1} as an **explicitly computed pair** for a given paranthesization if they are directly multiplied. For example, in the matrix multiplication chain $G_1G_2G_3G_4G_5G_6$ using paranthesization $(G_1(G_2G_3))(G_4(G_5G_6))$, G_2G_3 and G_5G_6 are only explicitly computed pairs. Consider a matrix multiplication chain $F_1F_2F_3F_4F_5$, where matrices F_1, F_2, F_3, F_4 and F_5 are of dimensions 2×25,25×3,3×16,16×1 and 1×1000, respectively. In the paranthesization of $F_1F_2F_3F_4F_5$ that minimizes the total number of scalar multiplications, the explicitly computed pairs is/are **(Gate-2018) (2 Marks)**

(A) F_1F_2 and F_3F_4 only

(B) F_2F_3 only

(C) F_3F_4 only

(D) F_1F_2 and F_4F_5 only

Question 72

Q Let A_1, A_2, A_3, and A_4 be four matrices of dimensions 10 x 5, 5 x 20, 20 x 10, and 10 x 5, respectively. The minimum number of scalar multiplications required to find the product $A_1A_2A_3A_4$ using the basic matrix multiplication method is **(Gate-2016) (2 Marks)**

Question 73

Q The Floyd-Warshall algorithm for all-pair shortest paths computation is based on: **(Gate-2016) (1 Marks)**

(A) Greedy paradigm.

(B) Divide-and-Conquer paradigm.

(C) Dynamic Programming paradigm.

(D) neither Greedy nor Divide-and-Conquer nor Dynamic Programming paradigm.

Question 74

Q Consider the weighted undirected graph with 4 vertices, where the weight of edge {i, j} g is given by the entry W_{ij} in the matrix W. The largest possible integer value of x, for which at least one shortest path between some pair of vertices will contain the edge with weight x is _____ **(Gate-2016) (2 Marks)**

$$W = \begin{bmatrix} 0 & 2 & 8 & 5 \\ 2 & 0 & 5 & 8 \\ 8 & 5 & 0 & x \\ 5 & 8 & x & 0 \end{bmatrix}$$

Question 75

Q Let G (V, E) be a directed graph with n vertices. A path from vi to vj in G is sequence of vertices $(v_i, v_{i+1}, \ldots\ldots, v_j)$ such that $(v_k, v_{k+1}) \in E$ for all k in i through j – 1. A simple path is a path in which no vertex appears more than once.
Let A be an n*n array initialized as follow

$$A[j,k] = \begin{cases} 1 \text{ if } (j,k) \epsilon E \\ 0 \text{ otherwise} \end{cases}$$

Consider the following algorithm.

```
for i = 1 to n
  for j = 1 to n
    for k = 1 to n
      A [j, k] = max (A[j, k] (A[j, i] + A [i, k]);
```

Which of the following statements is necessarily true for all j and k after terminal of the above algorithm? (Gate-2003) (2 Marks)
(A) A[j, k] ≤ n
(B) If A[j, k] ≥ n – 1, then G has a Hamiltonian cycle
(C) If there exists a path from j to k, A[j, k] contains the longest path lens from j to k
(D) If there exists a path from j to k, every simple path from j to k contain most A[j, k] edges

---

Question 76

Q The subset-sum problem is defined as follows. Given a set of n positive integers, S = $\{a_1, a_2, a_3, \ldots, a_n\}$ and positive integer W, is there a subset of S whose elements sum to W? A dynamic program for solving this problem uses a 2-dimensional Boolean array X, with n rows and W+1 column. X[i, j],1 <= i <= n, 0 <= j <= W, is TRUE if and only if there is a subset of $\{a_1, a_2, \ldots, a_i\}$ whose elements sum to j. Which of the following is valid for 2 <= i <= n and ai <= j <= W? **(Gate-2008) (2 Marks)**

(A) $X[i, j] = X[i-1, j] \vee X[i, j-a_i]$

(B) $X[i, j] = X[i-1, j] \vee X[i-1, j-a_i]$

(C) $X[i, j] = X[i-1, j] \wedge X[i, j-a_i]$

(D) $X[i, j] = X[i-1, j] \wedge X[i-1, j-a_i]$

Question 77

Q In the above question, which entry of the array X, if TRUE, implies that there is a subset whose elements sum to W?

(A) X[1, W]

(C) X[n, W]

(B) X[n ,0]

(D) X[n -1, n]

Question 78

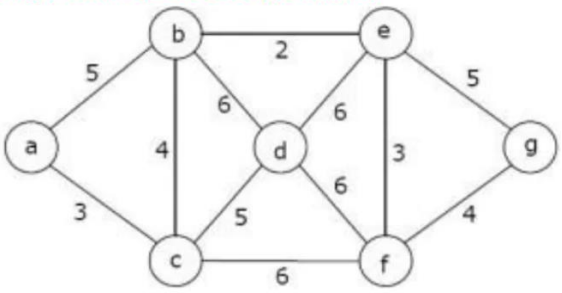
Q Consider the following graph: Which one of the following is NOT the sequence of edges added to the minimum spanning tree using Kruskal's algorithm? **(Gate-2009)(2 Marks)**

(A) (b, e),(e, f),(a, c),(b, c),(f, g),(c, d)

(B) (b, e),(e, f),(a, c),(f, g),(b, c),(c, d)

(C) (b, e),(a, c),(e, f),(b, c),(f, g),(c, d)

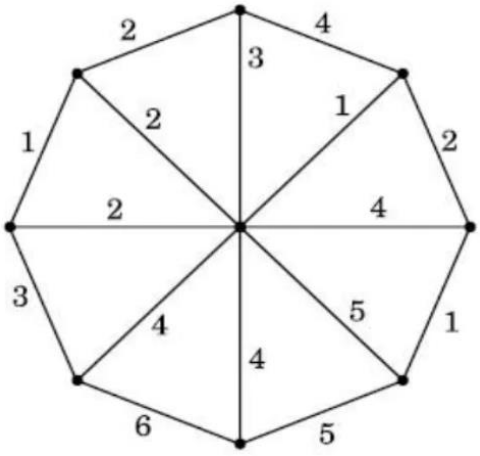
(D) (b, e),(e, f),(b, c),(a, c),(f, g),(c, d)



Question 79

Q Consider the graph shown below **(NET-DEC-2018)**

Use Kruskal's algorithm to find the minimum spanning tree of the graph. The weight of this minimum spanning tree is



a) 17

b) 14

c) 16

d) 13

Question 80

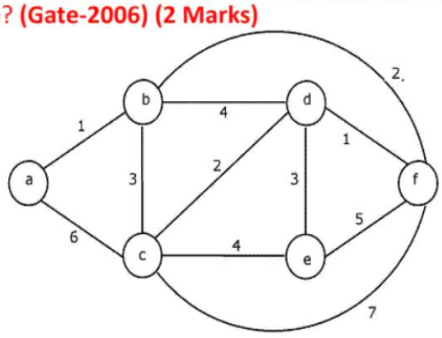
Q Consider the following graph:
Which one of the following cannot be the sequence of edges added, in that order, to a minimum spanning tree using Kruskal's algorithm? **(Gate-2006) (2 Marks)**

(A) (a—b),(d—f),(b—f),(d—c),(d—e)

(B) (a—b),(d—f),(d—c),(b—f),(d—e)

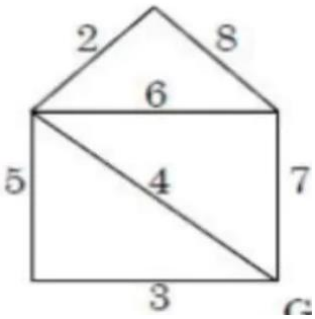
(C) (d—f),(a—b),(d—c),(b—f),(d—e)

(D) (d—f),(a—b),(b—f),(d—e),(d—c)



Question 81

Q The weight of minimum spanning tree in graph G, calculated using Kruskal's algorithm is: **(NET 2019 DEC)**



(A) 14

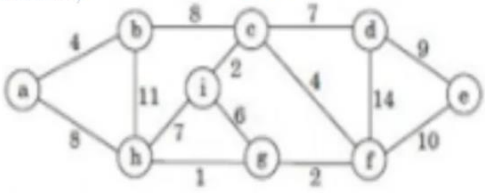
(B) 15

(C) 17

(D) 18

Question 82

Q Consider the undirected graph below:



Using Prim's algorithm to construct a minimum spanning tree starting with node a, which one of the following sequences of edges represents a possible order in which the edges would be added to construct the minimum spanning tree?**(NET 2020 OCT)**

(A) (a, b),(a, h),(g, h),(f, g),(c, f),(c, i), (c, d),(d, e)

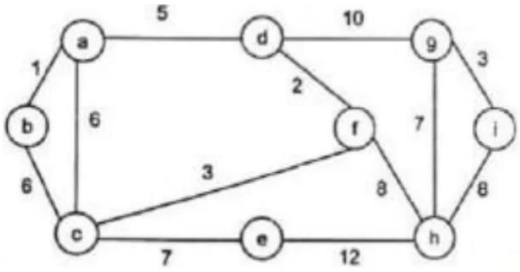
(B) (a, b),(b, h),(g, h),(g, i),(c, i),(c, f), (c, d),(d, e)

(C) (a, b),(b, c),(c, i),(c, f),(f, g),(g, h), (c, d),(d, e)

(D) (a, b),(g, h),(g, f),(c, f),(c, i),(f, e), (b, c),(d, e)

Question 83

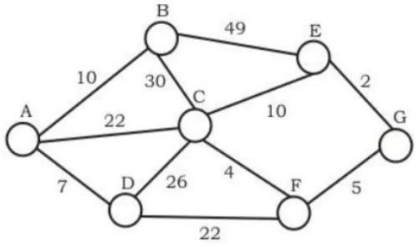
Q For the undirected, weighted graph given below, which of the following sequences of edges represents a correct execution of Prim's algorithm to construct a Minimum Spanning Tree? **(Gate-2008) (2 Marks)**



(A) (a, b), (d, f), (f, c), (g, i), (d, a), (g, h), (c, e), (f, h)
(B) (c, e), (c, f), (f, d), (d, a), (a, b), (g, h), (h, f), (g, i)
(C) (d, f), (f, c), (d, a), (a, b), (c, e), (f, h), (g, h), (g, i)
(D) (h, g), (g, i), (h, f), (f, c), (f, d), (d, a), (a, b), (c, e)

Question 84

Q Consider the undirected graph below:



Using Prim's algorithm to construct a minimum spanning tree starting with node A, which one of the following sequences of edges represents a possible order in which the edges would be added to construct the minimum spanning tree? **(Gate-2004) (2 Marks)**

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

Question 85

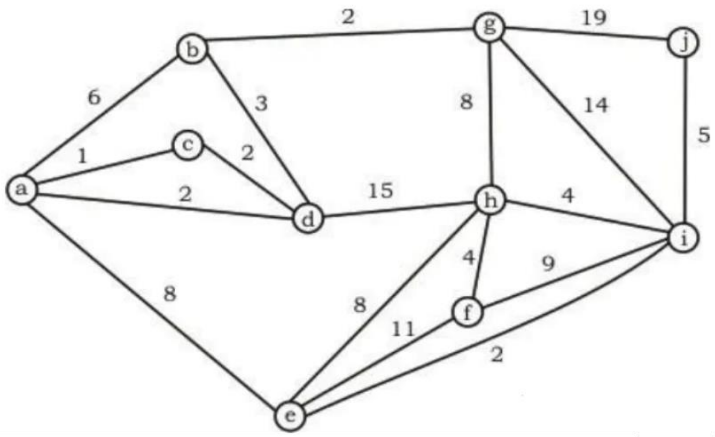
Q What is the weight of a minimum spanning tree of the following graph? **(Gate-2003) (2 Marks)**

(A) 29

(B) 31

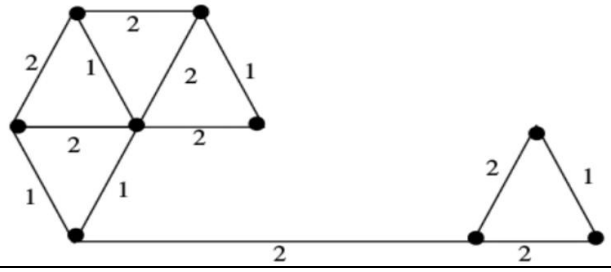
(C) 38

(D) 41



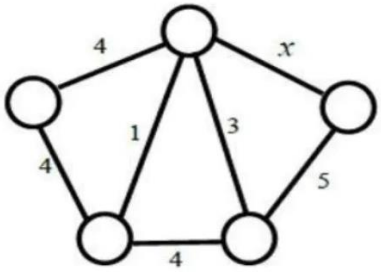
Question 86

Q The number of distinct minimum spanning trees for the weighted graph below is _____ **(GATE-2017) (2 Marks)**



Question 87

Q Consider the following undirected graph, Choose a value for x that will maximize the number of minimum weights spanning trees (MWSTs) of G. The number of MWSTs of G for this value of x is _____. **(Gate-2018) (2 Marks)**



Question 88

Q Let G be an undirected connected graph with distinct edge weight. Let e_{max} be the edge with maximum weight and e_{min} the edge with minimum weight. Which of the following statements is false? **(Gate-2000) (2 Marks)** (NET-NOV-2017)

(A) Every minimum spanning tree of G must contain e_{min}

(B) If e_{max} is in a minimum spanning tree, then its removal must disconnect G

(C) No minimum spanning tree contains e_{max}

(D) G has a unique minimum spanning tree

Question 89

Q Let G = (V, E) be any connected undirected edge-weighted graph. The weights of the edges in E are positive any distinct. Consider the following statements: **(Gate-2017) (2 Marks)**

I. Minimum Spanning Tree of G is always unique.
II. Shortest path between any two vertices of G is always unique.
Which of the above statements is/are necessarily true?

a) I only

c) both I and II

b) II only

d) neither I and II

Question 90

Q G = (V, E) is an undirected simple graph in which each edge has a distinct weight, and e is a particular edge of G. Which of the following statements about the minimum spanning trees (MSTs) of G is/are TRUE **(Gate-2016) (2 Marks)**

I. If e is the lightest edge of some cycle in G, then every MST of G includes e

II. If e is the heaviest edge of some cycle in G, then every MST of G excludes e

(A) I only

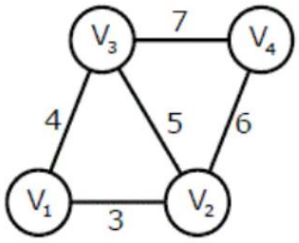
(C) both I and II

(B) II only

(D) neither I nor II

Question 91

Q An undirected graph G(V, E) contains n (n > 2) nodes named v_1 , v_2 ,....v_n. Two nodes v_i , v_j are connected if and only if 0 < |i – j| <= 2. Each edge (v_i, v_j) is assigned a weight i + j. A sample graph with n = 4 is shown below. What will be the cost of the minimum spanning tree (MST) of such a graph with n nodes? **(GATE - 2011) (2 Marks)**



(A) $1/12(11n^2 – 5n)$

(C) $6n – 11$

(B) $n^2 – n + 1$

(D) $2n + 1$

Question 92

Q An undirected graph G (V, E) contains n (n > 2) nodes named v_1, v_2..., v_n. Two nodes v_i and V_j are connected if and only if 0 < | i – j| ≤ 2. Each edge (v_i, v_j) is assigned a weight i + j. The cost of the minimum spanning tree of such a graph with 10 nodes is: **(NET-NOV-2017)**

(a) 88 **(c)** 49

(b) 91 **(d)** 21

Question 93

Q Consider a weighted complete graph G on the vertex set $\{v_1, v_2, ... v_n\}$ such that the weight of the edge (v_i, v_j) is $4 \mid i - j \mid$. The weight of minimum cost spanning tree of G is: **(NET-JULY-2016)**

(a) $4n^2$ **(c)** $4n - 4$

(b) n **(d)** $2n - 2$

Question 94

Q Let G be connected undirected graph of 100 vertices and 300 edges. The weight of a minimum spanning tree of G is 500. When the weight of each edge of G is increased by five, the weight of a minimum spanning tree becomes _____. **(GATE-2015) (2 Marks)**

Question 95

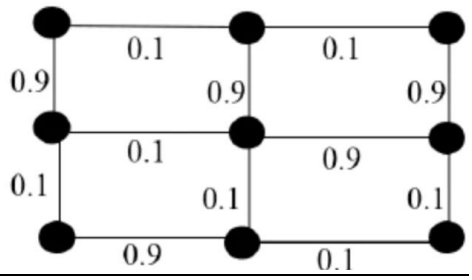
Q Consider the following undirected graph with edge weights as shown:
The number of minimum-weight spanning trees of the graph is _____.
(GATE 2021)

(a) 3

(b) 4

(c) 5

(d) 2



Question 96

Q Let G be a connected undirected weighted graph. Consider the following two statements.

- **S_1:** There exists a minimum weight edge in G which is present in every minimum spanning tree of G.
- **S_2:** If every edge in G has distinct weight, then G has a unique minimum spanning tree.

Which one of the following options is correct? **(GATE 2021) (1 MARKS)**

(A) Both S_1 and S_2 are true

(B) S_1 is true and S_2 is false

(C) S_1 is false and S_2 is true

(D) Both S_1 and S_2 are false

Question 97

Q Consider a complete undirected graph with vertex set {0, 1, 2, 3, 4}. Entry Wij in the matrix W below is the weight of the edge {i, j}. What is the minimum possible weight of a spanning tree T in this graph such that vertex 0 is a leaf node in the tree T? **(GATE - 2010) (2 Marks)**

$$W = \begin{pmatrix} 0 & 1 & 8 & 1 & 4 \\ 1 & 0 & 12 & 4 & 9 \\ 8 & 12 & 0 & 7 & 3 \\ 1 & 4 & 7 & 0 & 2 \\ 4 & 9 & 3 & 2 & 0 \end{pmatrix}$$

(A) 7

(B) 8

(C) 9

(D) 10

Question 98

Q Let w be the minimum weight among all edge weights in an undirected connected graph. Let e be a specific edge of weight w . Which of the following is FALSE? **(Gate-2007) (2 Marks)**

(A) There is a minimum spanning tree containing e.

(B) If e is not in a minimum spanning tree T, then in the cycle formed by adding e to T, all edges have the same weight.

(C) Every minimum spanning tree has an edge of weight w .

(D) e is present in every minimum spanning tree.

Question 99

Q Consider a weighted complete graph G on the vertex set $\{v_1, v_2, ...v_n\}$ such that the weight of the edge (v_i, v_j) is $2|i-j|$. The weight of a minimum spanning tree of G is: **(GATE - 2006)**

(A) n — 1

(B) 2n — 2

(C) ${}^nC_2$

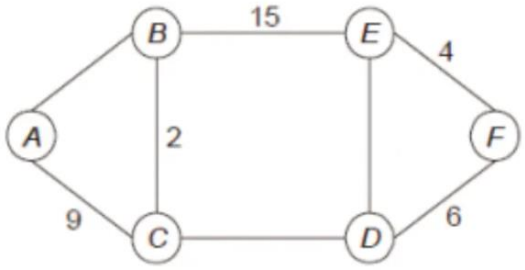
(D) 2

Question 100

Q Let G be a complete undirected graph on 4 vertices, having 6 edges with weights being 1, 2, 3, 4, 5, and 6. The maximum possible weight that a minimum weight spanning tree of G can have is. **(Gate-2016) (2 Marks)**

Question 101

Q The graph shown below 8 edges with distinct integer edge weights. The minimum spanning tree (MST) is of weight 36 and contains the edges: {(A, C), (B, C), (B, E), (E, F), (D, F)}. The edge weights of only those edges which are in the MST are given in the figure shown below. The minimum possible sum of weights of all 8 edges of this graph is _____. **(Gate - 2015) (2 Marks)**



Question 102

Q An undirected graph G has n nodes. It's adjacency matrix is given by an n × n square matrix whose (i) diagonal elements are 0's and (ii) non-diagonal elements are 1's. which one of the following is TRUE? **(Gate-2005) (2 Marks)**

(A) Graph G has no minimum spanning tree (MST)

(B) Graph G has a unique MST of cost n-1

(C) Graph G has multiple distinct MSTs, each of cost n-1

(D) Graph G has multiple spanning trees of different costs

Question 103

Q Let s and t be two vertices in a undirected graph G(V, E) having distinct positive edge weights. Let [X, Y] be a partition of V such that s ∈ X and t ∈ Y. Consider the edge e having the minimum weight amongst all those edges that have one vertex in X and one vertex in Y, The edge e must definitely belong to: **(Gate-2005) (2 Marks)**

(A) the minimum weighted spanning tree of G

(B) the weighted shortest path from s to t

(C) each path from s to t

(D) the weighted longest path from s to t

Question 104

Q Let G be a weighted undirected graph and e be an edge with maximum weight in G. Suppose there is a minimum weight spanning tree in G containing the edge e. Which of the following statements is always TRUE? **(Gate-2005)(2 Marks)**

(A) There exists a cut set in G having all edges of maximum weight.

(B) There exists a cycle in G having all edges of maximum weight

(C) Edge e cannot be contained in a cycle.

(D) All edges in G have the same weight

Question 105

Q Consider a simple undirected weighted graph G, all of whose edge weights are distinct. Which of the following statements about the minimum spanning trees of G is/are TRUE? **(GATE 2022) (2 MARKS)**

(A) The edge with the second smallest weight is always part of any minimum spanning tree of G.

(B) One or both of the edges with the third smallest and the fourth smallest weights are part of any minimum spanning tree of G .

(C) Suppose $S \subseteq V$ be such that $S \neq \Phi$ and $S \neq V$. Consider the edge with the minimum weight such that one of its vertices is in S and the other in $V \setminus S$. Such an edge will always be part of any minimum spanning tree of G .

(D) G can have multiple minimum spanning trees.

Question 106

Q Let G(V , E) be a directed graph, where V ={1, 2,3, 4,5} is the set of vertices and E is the set of directed edges, as defined by the following adjacency matrix A.

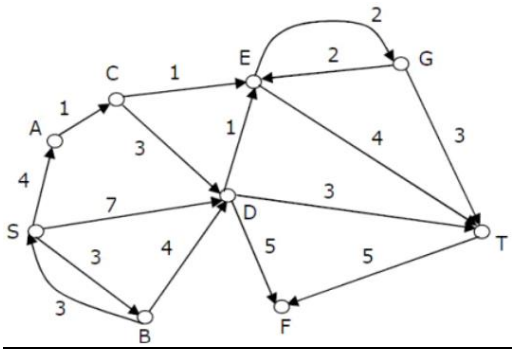
$$A[i][j] = \begin{cases} 1, & 1 \leq j \leq i \leq 5 \\ 0, & otherwise \end{cases}$$

$A[i][j]$ = 1 indicates a directed edge from node i to node j. A directed spanning tree of G, rooted at r ∈V , is defined as a sub graph T of G such that the undirected version of T is a tree, and T contains a directed path from r to every other vertex in V. The number of such directed spanning trees rooted at vertex 5 is_____. **(GATE 2022) (2 MARKS)**

Question 107

Q Consider the directed graph shown in the figure below. There are multiple shortest paths between vertices S and T. Which one will be reported by Dijkstra's shortest path algorithm? Assume that, in any iteration, the shortest path to a **(Gate-2012) (2 Marks)**

a) SDT **b)** SBDT **c)** SACDT **d)** SACET



Question 108

Q Which of the following algorithms solves the single-source shortest paths? **(NET-JULY-2018)**

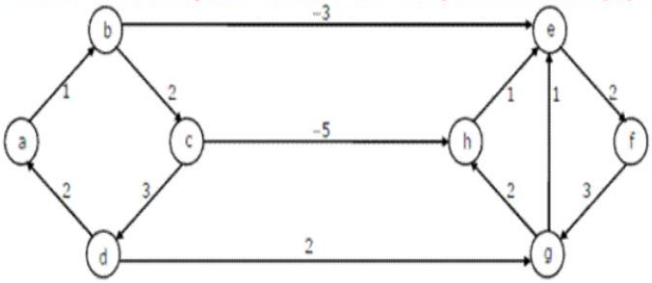
(a) Prim's algorithm **(c)** Johnson's algorithm

(b) Floyd - Warshall algorithm **(d)** Dijkstra's algorithm

Question 109

Q Dijkstra's single source shortest path algorithm when run from vertex a in the below graph, computes the correct shortest path distance to **(Gate-2008) (2 Marks)**

(A) only vertex a
(B) only vertices a, e, f, g, h
(C) only vertices a, b, c, d
(D) all the vertices



Question 110

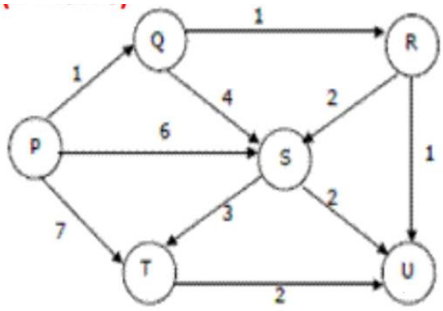
Q Suppose we run Dijkstra's single source shortest-path algorithm on the following edge weighted directed graph with vertex P as the source. In what order do the nodes get included into the set of vertices for which the shortest path distances are finalized? **(GATE - 2004) (2 Marks)**

(A) P, Q, R, S, T, U

(B) P, Q, R, U, S, T

(C) P, Q, R, U, T, S

(D) P, Q, T, R, U, S



Question 111

Q To implement Dijkstra's shortest path algorithm on unweighted graphs so that it runs in linear time, the data structure to be used is: **(Gate-2006) (1 Marks)**

(A) Queue

(C) Heap

(B) Stack

(D) B-Tree

Question 112

Q Consider a weighted directed graph. The current shortest distance from source S to node x is represented by d[x]. Let d[v]=29, d[u]=15 , w[u,v]=12 . What is the updated value of d[v] based on current information? **(NET 2019 DEC)**

(A) 29

(C) 25

(B) 27

(D) 17

Question 113

Q When using Dijkstra's algorithm to find shortest path in a graph, which of the following statement is not true? **(NET 2019 DEC)**

(A) It can find shortest path within the same graph data structure

(B) Every time a new node is visited, we choose the node with smallest known distance/ cost (weight) to visit first

(C) Shortest path always passes through least number of vertices

(D) The graph needs to have a non-negative weight on every edge

---

Question 114

Q An all-pairs shortest-paths problem is efficiently solved using **(NET-JUNE-2018)**

a) Dijkstra' algorithm

c) Kruskal algorithm

b) Bellman-Ford algorithm

d) Floyd-Warshall algorithm

Question 115

Q Match the following **(Gate-2015) (2 Marks)**

List-I	List-II
A. Prim's algorithm for minimum spanning tree	1. Backtracking
B. Floyd-Warshall algorithm for all pairs shortest paths	2. Greed method
C. Merge sort	3. Dynamic programming
D. Hamiltonian circuit	4. Divide and conquer

	A	B	C	D
a)	3	2	4	1
b)	1	2	4	3
c)	2	3	4	1
d)	2	1	3	4

Question 116

Q Consider the following table

Algorithms	Design Paradigms
(P) Kruskal	(i) Divide and Conquer
(Q) Quicksort	(ii) Greedy
(R) Floyd-Warshall	(iii) Dynamic Programming

Match the algorithm to design paradigms they are based on: **(Gate-2017) (2 Marks)**

a) P-(ii), Q-(iii), R-(i)

c) P-(ii), Q-(i), R-(iii)

b) P-(iii), Q-(i), R-(ii)

d) P-(i), Q-(ii), R-(iii)

Question 117

Q Which of the following statement(s) is/are correct regarding Bellman-Ford shortest path algorithm? **(Gate-2009) (1 Marks)**

P: Always finds a negative weighted cycle, if one exists.
Q: Finds whether any negative weighted cycle is reachable from the source.

a) P only

c) Both P and Q

b) Q only

d) Neither P nor Q

Question 118

Q Consider a weighted undirected graph with positive edge weights and let (u, v) be an edge in the graph. It is known that the shortest path from source vertex s to u has weight 53 and shortest path from s to v has weight 65. Which statement is always true? **(NET-JUNE-2012)**

(A) Weight (u, v) < 12

(C) Weight (u, v) > 12

(B) Weight (u, v) = 12

(D) Weight (u, v) >= 12

Question 119

Q Is the following statement valid?
Given a weighted graph where weights of all edges are unique (no two edge have same weights), there is always a unique shortest path from a source to destination in such a graph.

(A) True

(B) False

Question 120

Q Is the following statement valid?
Given a graph where all edges have positive weights, the shortest paths produced by Dijkstra and Bellman Ford algorithm may be different but path weight would always be same.

(A) True

(B) False

Question 121

Q Is the following statement valid about shortest paths?
Given a graph, suppose we have calculated shortest path from a source to all other vertices. If we modify the graph such that weights of all edges is becomes double of the original weight, then the shortest path remains same only the total weight of path changes.

(A) True

(B) False

Question 122

Q In a weighted graph, assume that the shortest path from a source 's' to a destination 't' is correctly calculated using a shortest path algorithm. Is the following statement true?
If we increase weight of every edge by 1, the shortest path always remains same.

(A) Yes **(B)** No

Question 123

Q Which of the following standard algorithms is not a Greedy algorithm?

(A) Dijkstra's shortest path algorithm **(B)** Prim's algorithm
(C) Kruskal algorithm **(D)** Huffman Coding
(E) Bellmen Ford Shortest path algorithm

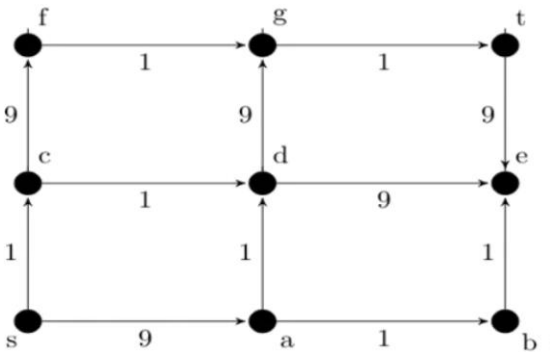
Question 124

Q Which of the following standard algorithms is not Dynamic Programming based.

(A) Bellman–Ford Algorithm for single source shortest path
(B) Floyd Warshall Algorithm for all pairs shortest paths
(C) 0-1 Knapsack problem
(D) Prim's Minimum Spanning Tree

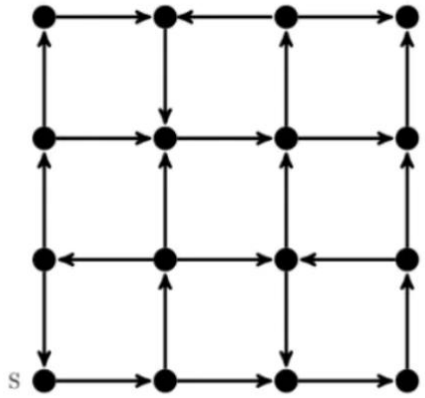
Question 125

Q In a directed acyclic graph with a source vertex s, the quality-score of a directed path is defined to be the product of the weights of the edges on the path. Further, for a vertex v other than s, the quality-score of v is defined to be the maximum among the quality-scores of all the paths from s to v. The quality-score of s is assumed to be 1. The sum of the quality-scores of all vertices on the graph shown above is _____ .
(GATE 2021) (2 MARKS)



Question 126

Consider the following directed graph:



Which of the following is/are correct about the graph?

A. The graph does not have a topological order
B. A depth-first traversal starting at vertex S classifies three directed edges as back edges
C. The graph does not have a strongly connected component
D. For each pair of vertices u and v, there is a directed path from u to v

Question 127

Q Let G be a simple undirected graph. Let T_D be a depth first search tree of G. Let T_B be a breadth first search tree of G. Consider the following statements.

(I) No edge of G is a cross edge with respect to T_D. (A cross edge in G is between two nodes neither of which is an ancestor of the other in T_D).

(II) For every edge (u, v) of G, if u is at depth i and v is at depth j in T_B, then $|i - j| = 1$.

Which of the statements above must necessarily be true? **(Gate-2018) (2 Marks)**

a) I only
c) Both I and II
b) II only
d) Neither I nor II

ANSWER KEY

1	C	**33**	B	**65**	D	**97**	D
2	D	**34**	D	**66**	B	**98**	D
3	A,B	**35**	B	**67**	34	**99**	B
4	A	**36**	C	**68**	C	**100**	7
5	C	**37**	B	**69**	B	**101**	69
6	D	**38**	A	**70**	A	**102**	C
7	D	**39**	B	**71**	C	**103**	A
8	B	**40**	D	**72**	1500	**104**	A
9	B	**41**	B	**73**	C	**105**	A,B,C
10	D	**42**	A	**74**	11	**106**	24
11	D	**43**	B	**75**	D	**107**	D

12	A	**44**	A	**76**	B	**108**	D
13	A	**45**	0.08	**77**	C	**109**	D
14	C	**46**	D	**78**	D	**110**	B
15	C	**47**	D	**79**	C	**111**	A
16	A	**48**	D	**80**	D	**112**	B
17	C	**49**	C	**81**	B	**113**	C
18	C	**50**	A	**82**	A,C	**114**	D
19	A	**51**	C	**83**	C	**115**	C
20	B	**52**	D	**84**	D	**116**	C
21	A	**53**	B	**85**	B	**117**	B
22	D	**54**	B	**86**	6	**118**	D
23	B	**55**	A	**87**	4	**119**	B
24	A	**56**	148	**88**	C	**120**	A
25	B	**57**	6	**89**	A	**121**	A
26	D	**58**	225	**90**	B	**122**	B
27	D	**59**	B	**91**	B	**123**	E
28	C	**60**	A	**92**	B	**124**	D
29	B	**61**	D	**93**	C	**125**	929
30	A	**62**	A	**94**	995	**126**	A,B
31	A	**63**	358	**95**	A	**127**	A
32	C	**64**	16	**96**	C		