DESIGN & ANALYSIS OF ALGORITHMS

Some Time Complexities :

→
```
for(i=1; i<=n; i++){
    for(j=2; j<=n/2; j++){
        x+y;
    }
}
```

$O(n^2)$

→
```
main(){
    i=1;
    while(i<n){
        i = 2*i;
    }
}
```

$i = 1, 2, 4, 8, \ldots, 2^k$

where k is the number of times the loop is executed

$2^k = n$

$\Rightarrow \boxed{k = \log_2 n}$

$O(\log_2 n)$

→
```
main(){
    i=2
    while(i<=n){
        i=i^2;
    }
}
```

$i = 2, 4, 16, 256, \ldots, 2^{2^k}$

$2^{2^k} = n$

$\Rightarrow 2^k = \log_2 n \Rightarrow \boxed{k = \log_2 \log_2 n}$

$O(\log_2 \log_2 n)$

→
```
main(){
    for(i=1; i<=n^2; i=2*i){                    ---- logn
        for(j=n; j>10; j=j^1/5){                ---- loglogn
            for(k=25; k<n^7; k=k^9){            ---- loglogn
                k = y+z;
            }
        }
    }
}
```

$O(\log n * \log\log n * \log\log n)$

→ A(n) {
    if (n <= 2)
        return 1;
    else
        return A(√n);
}

power

$O(\log \log n)$

---

power → loglogn
multiply → logn
add/sub → n

→ main () {
    for (i=1; i<=n; i++) { ——— n
        for (j=1; j<=i^2; j++) { ——— n^2
            for (k=n; k>5; k=k^(1/7)) { ——— loglogn
                x=a+b;
            }
        }
    }

$O(n^2 \cdot \log \log n)$

→ Asymptotic Notations

1. Big Oh Notation (O)
2. Omega Notation (Ω)
3. Theta Notation (Θ)

1. Big Oh Notation (O)

because time cannot be negative

Let $f(n)$ & $g(n)$ be two positive functions and let n be the number of inputs, n cannot be negative and cannot be in fraction.

$f(n) = O(g(n))$ iff $f(n) \leq c \cdot g(n)$ $\forall$ $n \geq n_0$

such that there exists two positive constants $c > 0$ & $n_0 \geq 1$.

Eg: ① $f(n) = n+10$

$g(n) = n$

To prove: $f(n) = O(g(n))$

$\Rightarrow f(n) = O(n)$

$n+10 <= c * n$

for $n_0 = 10$, $n+10 <= 2 * n$

$\therefore f(n) = O(g(n)) = O(n)$

★ $g(n)$ is the upperbound of $f(n)$

★ c is to be chosen as per will

$n_0 = 10$ & $c = 2$

Eg: ② $f(n) = n$

$g(n) = n+10$

To prove: $f(n) = O(g(n))$

$\Rightarrow f(n) = O(n+10)$

$n <= c * (n+10)$

for $n_0 = 1$, $n <= 1 * (n+10)$

$\therefore f(n) = O(g(n)) = O(n+10)$

$n_0 = 1$ & $c = 1$

Eg: ③ $f(n) = n^2+n+10$

$g(n) = n^2$

To prove: $f(n) = O(g(n))$

$\Rightarrow f(n) = O(n^2)$

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

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

for $n_0 = 4$, $n^2+n+10 \leq 2n^2$ $\quad n_0 = 4$ & $c = 2$

$\therefore f(n) = O(g(n)) = O(n^2)$

Eg: ④ $f(n) = n^2$

$g(n) = n$

To prove: $f(n) = O(g(n))$

$\Rightarrow f(n) = O(n)$

$f(n) <= c * g(n)$

$\Rightarrow n^2 <= c * n$

$c = n$

But c can't be a variable as its a constant

$n^2 <= n * n$

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

② Omega Notation (Ω)

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

iff $f(n) >= c \cdot g(n)$ $\forall\ n >= n_0$ $\exists\ c > 0$ & $n_0 \geq 1$

Eg: ① $f(n) = n$

$g(n) = n+10$

To prove: $f(n) = \Omega(g(n))$

$\Rightarrow f(n) = \Omega(n+10)$

$n >= c * (n+10)$

for $n_0 = 10$ & $c = 1/2$;

$n >= \frac{1}{2} * (n+10)$

$\therefore f(n) = \Omega(g(n)) = \Omega(n+10)$

Eg: ② $f(n) = n^2$

$g(n) = n^2 + n + 10$

To prove : $f(n) = \Omega(g(n))$

$\Rightarrow f(n) = \Omega(n^2+n+10)$

$n^2 >= c * (n^2 + n + 10)$

for $n_0 = 4$ & $c = 1/2$;

$n >= \frac{1}{2} * (n^2+n+10)$

$\therefore f(n) = \Omega(g(n)) = \Omega(n^2+n+10)$

③ Theta Notation (Θ)

$f(n) = \Theta(g(n))$

iff $f(n) <= c_1 \cdot g(n)$ && $f(n) >= c_2 \cdot g(n)$

$\forall n \geq n_0 \quad \exists c_1, c_2 > 0$ & $n_0 \geq 1$

Eg: $f(n) = n + 10$

$g(n) = n$

To prove : $f(n) = \Theta(g(n))$

$\Rightarrow f(n) = \Theta(n)$

$n + 10 < c_1 * n$

for $n_0 = 10$ & $c_1 = 2$;

$n + 10 <= 2n$

$n + 10 >= c_2 * n$

for $n_0 = 1$ & $c_2 = 1$;

$n + 10 >= 1n$

$\therefore f(n) = \Theta(g(n)) = \Theta(n)$

Q]: What is the upper bound of $n!$

$$n! = O(?)$$

$$\Rightarrow \quad n! = \underbrace{n \times (n-1) \times \cdots \times 3 \times 2 \times 1}_{\substack{n \text{ times } n \text{ ie } n^n \\ n \; n \; \cdots \; n \; n \; n}}$$

$$\therefore \; n! = O(n^n)$$

Q]: Check whether the following statements are true or false.

① $2^{2n} = O(2^n)$ False

Q: If $f(n) = O(g(n))$ is true, then $2^{f(n)} = O(2^{g(n)})$

Let us take an example

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

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

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

$$2n \leq c \cdot n$$

for $c = 2$ ↑

Properties of Asymptotic Notation

① Reflexive Property

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

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

$f(n) = \Theta(f(n))$

② Symmetric Property

If $f(n) = \Theta(g(n))$, then $g(n) = \Theta(f(n))$

③ Transitive Property

- If $f(n) = O(g(n))$ & $g(n) = O(h(n))$,

  then $f(n) = O(h(n))$

- If $f(n) = \Omega(g(n))$ & $g(n) = \Omega(h(n))$,

  then $f(n) = \Omega(h(n))$

- If $f(n) = \Theta(g(n))$ & $g(n) = \Theta(h(n))$,

  then $f(n) = \Theta(h(n))$

④ If $f(n) = O(g(n))$, then $h(n) \cdot f(n) = O(h(n) \cdot g(n))$

⑤ If $f(n) = O(g(n))$ & $h(n) = O(e(n))$,

then → $f(n) + h(n) = O(\max(g(n), e(n)))$

then →

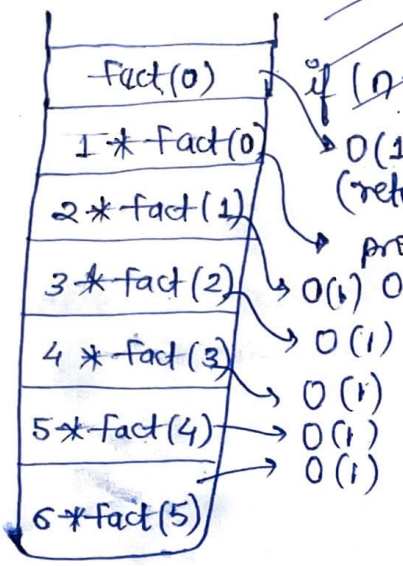
- $f(n) \cdot h(n) = O(g(n) \cdot e(n))$

⑥

Q]: Write a recursive program to find the factorial of a number.

⇒

```
int fact(int n){
    if (n==0){
        return 1;
    }
    else{
        return n * fact(n-1);
    }
}
```



→ O(7)
O(n+1) ie. O(n) Ans

$$T(n) = T(n-1) + O(1)$$ (O(1): constant c)

HW

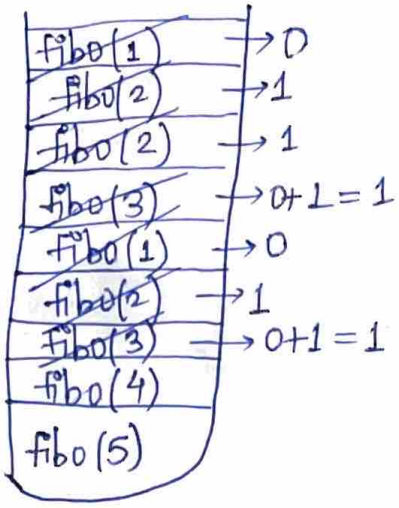
Q]: Write a recursive function for fibonacci series and derive its time complexity.
Also, write its iterative function (without recursion) and derive its time complexity.

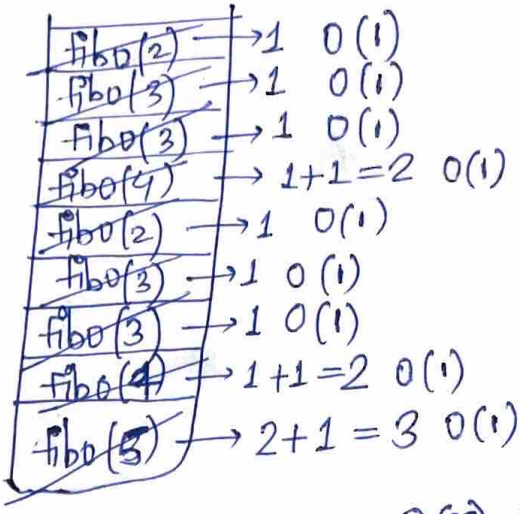
Recursive function

⇒
```
#include <stdio.h>
int fibo(int n){
    int sum=0;
    if (n==0 || n==1){
        return 0;
    }
    else if (n==2 || n==3){
        return 1;
    }
    else{
        sum = fibo(n-1)+ fibo(n-2);
        return sum;
    }
}
```

can directly return without storing in the sum variable

Taking $n=5$





Time complexity = ~~O(n)~~ $O(2^n)$

Iterative function

0, 1, 1, 2, 3, 5, 8, 13, ——

Recursive function

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

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

$O(1)$ → for returning, for comparing

Recurrence relation can be solved using 3 methods:

① substitution method
② Recursive tree method
③ Master Theorem

k → number of repetitions

① Substitution Method

- used when only one (preferred) function call

Eg: factorial $T(n) = \begin{cases} O(1) & , \text{if } n=1; \\ T(n-1)+C & , \text{if } n>1; \end{cases}$

$T(n) = T(n-1) + C$ —— (i)
$T(n-1) = T(n-2) + C$ —— (ii)

$T(n) = T(n-2)+C+C$

⋮

$T(n) = T(n-k) + \underbrace{C+C+\dots+C}_{k \text{ times}}$

k → used to reach till 1 (here)

$\therefore n-k=1$
$\Rightarrow k=(n-1)$

$T(n) = T(n-(n-1)) + (n-1)C$
$\Rightarrow T(n) = T(1) + nC - C$
$\Rightarrow T(n) = O(n)$

Eg: $T(n) = T(n-1) + n$

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

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

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

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

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

$T(n) = T(n-4) + (n-4) + (n-2) + (n-1) + n = T(n-4) + 4n - 7$

$T(n) = T(n-k) + kn - C$

$n-k=1$
$\Rightarrow k = n-1$

$\Rightarrow T(n) = T(1) + n^2 - n - C$

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

Eg: $T(n) = \begin{cases} T(n-1) + \frac{1}{n} & ; n \geq 1 \\ O(1) & ; n = 1 \end{cases}$

$\frac{1}{n-(k-1)}$

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

$n - n+1+1$

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

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

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

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

~~$\Rightarrow T(n) \leq T(n-k) +$~~

$n - k = 1 \Rightarrow k = n-1$

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

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

$$\boxed{T(n) = O(\log n)}$$

Q]: $T(n) = \begin{cases} 1 & \text{if } n=0 \\ T(n-2)+n^2 & \text{if } n>0 \end{cases}$

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

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

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

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

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

$n - 2k = 0$

$2k = n$

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

$\vdots \qquad \vdots$

$T(n) = T(n-2k) + (n-(2k-2))^2 + \ldots + (n-2)^2 + n^2$

$T(n) = T(0) + 4 + 16 + 36 + 64 + \ldots + (n-2)^2 + n^2$

$\Rightarrow T(n) = 1 + 4 + 16 + 36 + 64 + \ldots + (n-2)^2 + n^2$

$= 1 + 4\left\{1 + 4 + 9 + 16 + \ldots + \frac{(n-2)^2}{2} + \left(\frac{n}{2}\right)^2\right\}$

$= 1 + 4\left\{\frac{\frac{n}{2} \times \left(\frac{n}{2}+1\right) \times \left(2 \times \frac{n}{2}+1\right)}{6}\right\}$

$= 1 + 4\left(\frac{n(n+2)(n+1)}{4 \times 6}\right)$

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

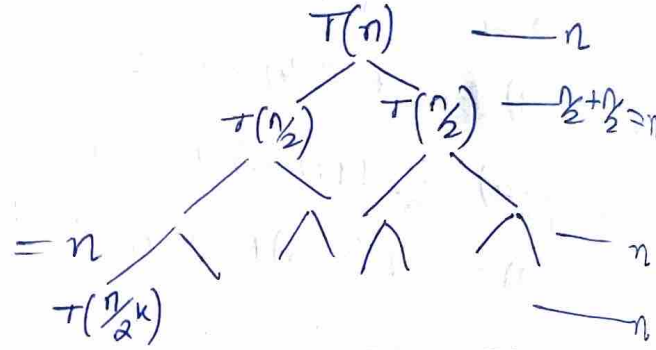
$= \frac{6 + n(n+1)(n+2)}{6}$

$\Rightarrow \boxed{T(n) = O(n^3)}$

② Recursive Tree Method

- Used when there is more than one function call. (preferred)

Eg: $T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n/2) + T(n/2) + n \end{cases}$



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

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

$\Rightarrow 2^k = n$

$\Rightarrow \boxed{k = \log_2 n}$

$\boxed{T(n) = kn = n\log_2 n}$

cost at each level is n

~~$\frac{n}{3^k} + \frac{2n}{3^k} = 1$~~

Eg:

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ T(n/3) + T(2n/3) + n & ; n>1 \end{cases}$$

$\frac{2^k n}{3^k} = 1 \Rightarrow 2^k n = 3^k$

$\Rightarrow \log_3 2^k n = k$

$T(n) = \log_{3/2} n + \log_3 2n = \log_3 n + 2n = \log_3 3n$

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

$\Rightarrow n = \left(\frac{3}{2}\right)^k$

$\Rightarrow \boxed{\log_{3/2} n = k}$

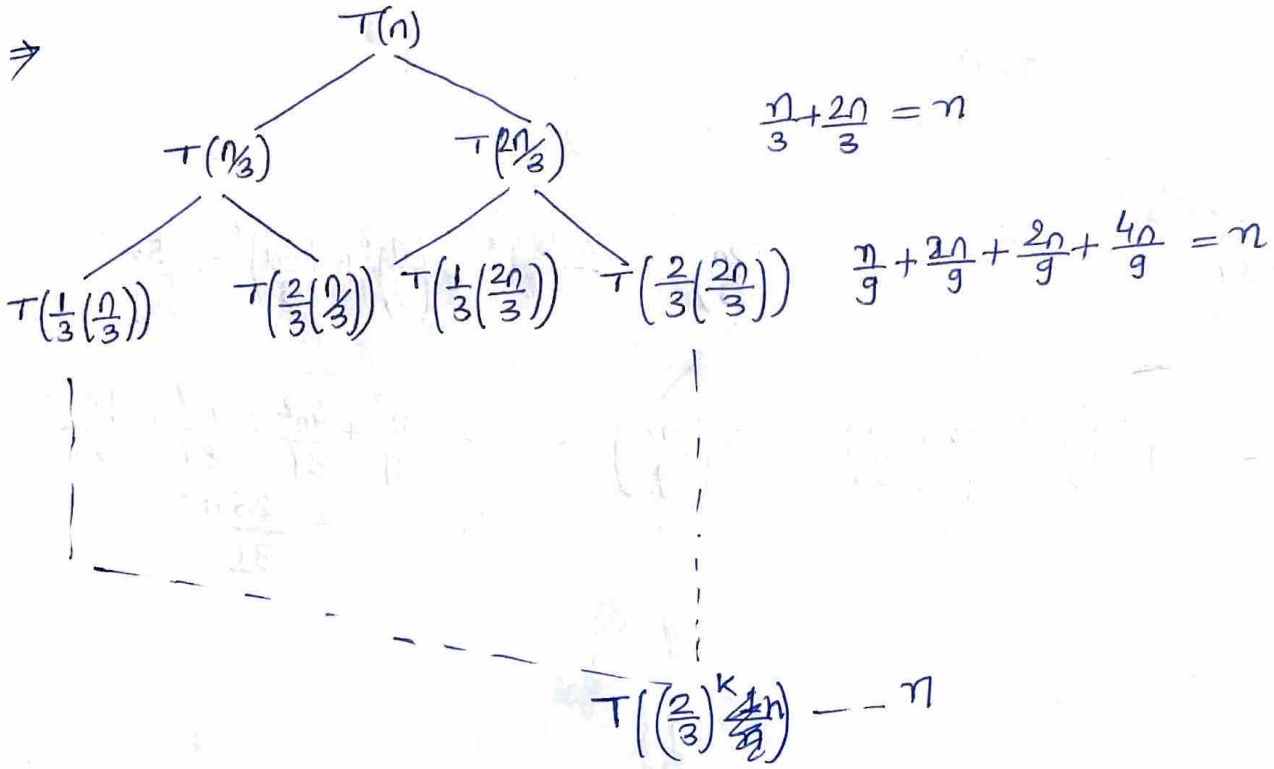
$\left(\frac{1}{3}\right)^k n = 1$

$\Rightarrow n = 3^k$

$\Rightarrow \boxed{\log_3 n = k}$

Eg: $T(n) = \begin{cases} 1 & ; \text{if } n=1 \\ T(n/3) + T(2n/3) + n & ; n>1 \end{cases}$

$\Rightarrow$



$\frac{n}{3} + \frac{2n}{3} = n$

$\frac{n}{9} + \frac{2n}{9} + \frac{2n}{9} + \frac{4n}{9} = n$

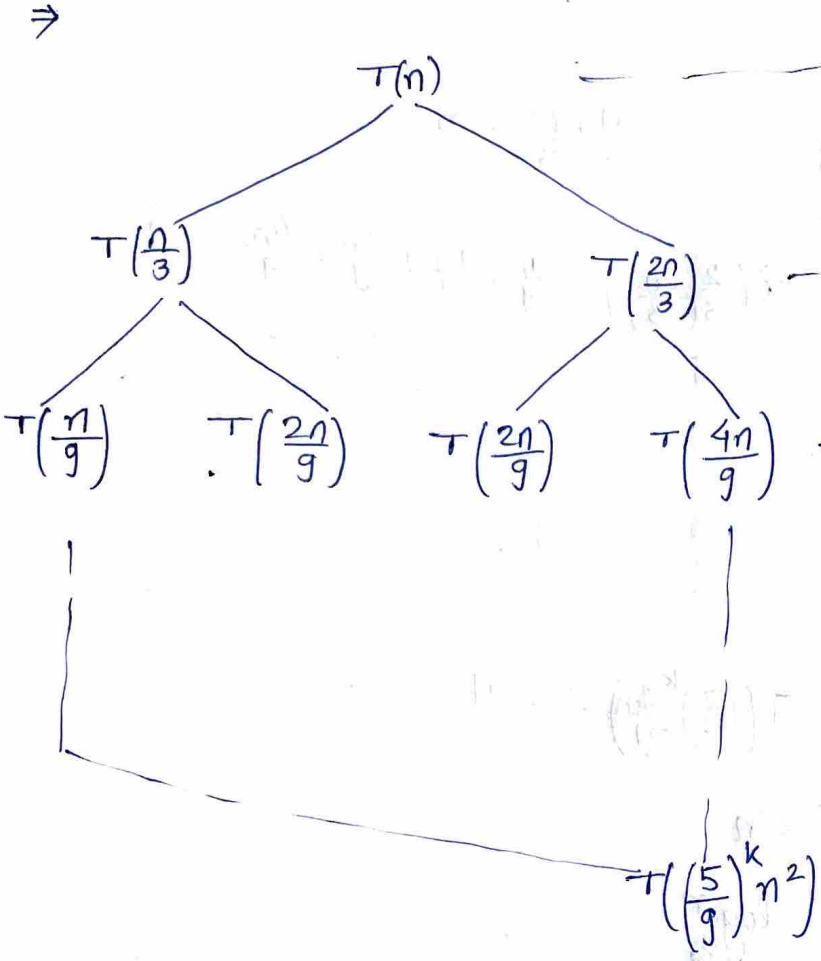
Now, $\left(\frac{2}{3}\right)^k n = 1 \Rightarrow \left(\frac{3}{2}\right)^k = n$

$\Rightarrow k = \log_{3/2} n$

$\therefore T(n) = kn = n \log_{3/2} n$

Q]: $T(n) = \begin{cases} 1 & ; \text{if } n=1 \\ T(n/3) + T(2n/3) + n^2 \end{cases}$

$\Rightarrow$



Now, $k\left(\frac{5}{9}\right)^k n^2 = 1$

$\Rightarrow$ ~~$k = \neq$~~

$\Rightarrow \left(\frac{5}{9}\right)^k n^2 = 1$

$\Rightarrow \left(\frac{5}{9}\right)^k = \frac{1}{n^2}$

$\Rightarrow \left(\frac{9}{5}\right)^k = n^2$

$\Rightarrow k = \log_{9/5} n^2$

~~$T(n) = k \cdot \left\{ \frac{5n^2}{9} + \frac{25n^2}{81} + \cdots + \left(\frac{5}{9}\right)^k n^2 \right\}$~~

~~$= 1$~~

~~$\Rightarrow k \cdot n^2 \sum \left(\frac{5}{9}\right)^k = 1$~~

Now, $T(n) = k \cdot n^2 \left\{ 1 + \frac{5}{9} + \frac{5^2}{9^2} + \cdots + \frac{5^k}{9^k} \right\}$

$= k \cdot n^2 \times \left(\frac{1 - (5/9)^k}{1 - 5/9} \right) 1$

$= \frac{\log n^2}{\log 9/5} n^2 \frac{1 - (5/9)^{\log_{9/5} n^2}}{4/9}$

$= \frac{\log n^2}{\log 9/5} n^2 \frac{1 - (5/9)^{\log_{5/9} 1/n^2}}{5/9}$

$= \frac{\log n^2}{\log 9/5} n^2 \left(1 - \frac{1}{n^2}\right) \frac{}{4/9}$

$T(n) \simeq n^2$

Master's Theorem

Master's Theorem can be applied to the following type of

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

where, a, b are constants & $a \geq 1$, $b > 1$

$f(n)$ is a positive function.

$\epsilon \rightarrow$ constant

Case ① If $f(n) = O\left(n^{\log_b^a - \epsilon}\right)$,
then $T(n) = \theta\left(n^{\log_b^a}\right)$

$\rightsquigarrow n^{\log_b^a} > f(n)$
so $T(n) = \theta\left(n^{\log_b^a}\right)$

Case ② If $f(n) = \Omega\left(n^{\log_b^a + \epsilon}\right)$,
then $T(n) = \theta(f(n))$

★★ This only works for the specific type of $T(n)$

Case ③ If $f(n) = \theta\left(n^{\log_b^a}\right)$,
then $T(n) = \theta\left(n^{\log_b^a} * \log n\right) = \theta(f(n) * \log n)$

Eg: $T(n) = 4T(n/2) + n$

Here, $a = 4$, $b = 2$ & $f(n) = n$

$n^{\log_2^4} = n^2$

★ Calculate $n^{\log_b^a}$ first

Now, $n^{2-1} = f(n) = n$

$\Rightarrow T(n) = \theta(n^2)$

Eg: $T(n) = 2T(n/2) + n^2$

Here, $a = 2$ $b = 2$ & $f(n) = n^2$

$n^{\log_2^2} = n^1 = n$

$\because n^{1+1} = f(n) = n^2$

$\Rightarrow T(n) = \theta(n^2)$

Q:

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

$a = 4$
$b = 2$
$f(n) = n^2$

$n^{\log_2 4} = n^2$

$f(n) = n^2$ → equal

$\Rightarrow T(n) = \theta(n^2 \log n)$

Special case

If $f(n) = \theta(n^{\log_b a} \cdot (\log n)^k)$, where $k \geq 0$ (constant)

then

$$T(n) = \theta(n^{\log_b a} (\log n)^{k+1})$$

$T\left(\frac{n}{9}\right)$

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

$a = 2$
$b = 2$
$f(n) = n \log n$

Binary Search

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

$\Rightarrow a = 8$
$\Rightarrow b = 2$
$\Rightarrow f(n) = n^2$

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

$f(n) = n^2$

$n^{2+1} = n^3$

$\Rightarrow T(n) = \theta(n^3)$

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

$a = 1$
$b = 2$
$f(n) = c$

$n^{\log_b a} = n^0 = 1$

$T(n) = \theta(1) \cdot \log n)$

$\Rightarrow T(n) = \theta(\log n)$

Q: Write the function for insertion sort and derived its time complexity.

⇒
```
#include <stdio.h>
void insertionSort(int arr[]){
    for (i=1; i<n; i++){
        int temp = a[i];
        int j = i-1;
        while (j>=0 && a[j] > temp){
            a[j+1] = a[j];
            j--;
        }
        a[j+1] = temp;
    }
}
```

Here, the outer loop iterates through each element in the array (except the first one), which runs in $O(n)$ time.

& the inner loop; for each element, it compares and shifts elements of the sorted part of the array to insert the current element into its correct position. This can take upto $O(n)$ time for each element.

∴ The total time complexity = $O(n) * O(n)$

⇒ $TC = O(n^2)$

Q]: Q]: If you have a already sorted array, which sorting technique will give the worst time complexity?

⇒ Bubble sort will give the worst time complexity.

⇒ The worst case time complexity of bubble sort is $O(n^2)$, but for an already sorted array, its time complexity remains $O(n^2)$.

- This is because the algorithm of bubble sort does not detect that the array is sorted or not and continues to do unnecessary passes.

check Yourself

Q: $T(n) = T(\sqrt{n}) + c$ if $n=2$ $T(n)=1$

$T(\sqrt{n}) = T(\sqrt{\sqrt{n}}) + c$

$\Rightarrow T(n) = T(n^{1/4}) + c$

$\Rightarrow T(\sqrt{n^{1/2}}) = T(n^{1/8}) + c$

$\Rightarrow T(n^{1/8}) = T(n^{1/16}) + c$

|

$\Rightarrow T(n^{1/2^k}) = T(n^{1/2^{k+1}}) + c$

$n^{1/2^k} = 2$

$\Rightarrow \frac{1}{2^k}\log_2 n = 1$

$\Rightarrow 2^k = \log_2 n$

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

Now $T(n) = T(n^{1/2^k}) + c + c + \ldots$ k times

$\Rightarrow T(n) = \underbrace{T(n^{1/2^k})}_{\text{constant}} + kc$

$\Rightarrow T(n) = O(\log_2 \log_2 n)$

Using Master's Theorem,

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

$$T(n) = T(\sqrt{n}) + c$$

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

Let $n = 2^m \Rightarrow m = \log_2 n$

$$T(2^m) = T(2^{m/2}) + c$$

Let $T(2^m) = S(m)$ → some function of m

Then, $S(m) = S\left(\frac{m}{2}\right) + c$

$a = 1$, $b = 2$, $f(n) = c$

$n^{\log_b a} = n^{\log_2 1} = n^0 = 1$ (constant)

$f(n) = c$ (constant) $\quad \because f(n) = \Theta(n^{\log_b a})$ i.e. constant = constant

$\therefore S(m) = \Theta(\log m)$

$\Rightarrow T(2^m) = \Theta(\log m)$

$\Rightarrow T(n) = \Theta(\log_2 \cdot \log_2 n)$

Q: $T(n) = 2T(\sqrt{n}) + n$

Let $n = 2^m \Rightarrow m = \log_2 n$

$T(2^m) = 2T(2^{m/2}) + 2^m$

$S(m) = 2S(m/2) + 2^m$

Here, $a = 2$, $b = 2$, $f(m) = 2^m$

$m^{\log_b a} = m^{\log_2 2} = m$ ✓

$2^m > m$

$f(m) = 2^m$

$\because f(m) = m^{\log_2 2} = m$ $\Rightarrow S(m) = \Theta(m \cdot \log m)$

So $T(2^m) = \Theta(m \cdot \log m) = \Theta(2^m)$

$\Rightarrow T(n) = \Theta(\log_2 n \cdot \log\log_2 n)$ $\Theta(n)$ Ans

Q: $T(n) = 2T(\sqrt{n}) + \log n$

$S(m) = \theta(m)$

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

$\Rightarrow T(n) = \theta(\log_2 n \log n)$

$2^m = n$

$\Rightarrow m = \log_2 n$

Q: $T(n) = 2T(\sqrt{n}) + \log n$

Let $n = 2^m \Rightarrow m = \log_2 n$

$T(2^m) = 2T(2^{m/2}) + \log 2^m$

$\Rightarrow S(m) = 2S(m/2) + \log 2^m$ (some function of m)

Here, $a = 2$ & $b = 2$

$m^{\log_b^a} = m^{\log_2^2} = m$

$f(m) = \log_2 2^m = m$ (assuming base to be 2) ✓

Here, $f(m) \neq m^{\log_b^a}$

So, ~~second~~ third case will be applied

~~$S(m) = \theta(f(m))$~~

$S(m) = \theta(m^{\log_b^a} \cdot \log m)$

$S(m) = \theta(m \cdot \log m)$

$\Rightarrow T(2^m) = \theta(m \cdot \log m)$

$\Rightarrow \boxed{T(n) = \theta(\log_2 n \cdot \log\log_2 n)}$

assuming base to be e;

$f(m) = \log_e 2^m = m \log 2 = m \times 0.693 < m$

so, first case will be applied

$S(m) = \theta(m)$

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

$\Rightarrow \boxed{T(n) = \theta(\log_2 n)}$

DIVIDE AND CONQUER

(03/08/24)

- Binary search

Let sorted array A

```
void binarysearch ( int A[], int i, int j, int x) {
    if (i==j) {
        if (A[i] ==x) {                    } constant
            return i;                      }
        }
        else {
            return -1; // not in array A
        }
    }int mid = (i+j)/2;                    } constant
    if (A[mid] ==x) {                      } constant
        return mid;
    }
    else if ( A[mid] > x){                 } constant
        binarysearch (A, i, mid-1, x);     } T(n/2)
    }
    else {                                 } constant
        binarysearch (A, mid+1, j, x);     } T(n/2)
    }
}
```

* only one condition

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

Using substitution method,

$T(n) = T(n/2) + c$

$T(n/2) = T(n/4) + c_1$

$\Rightarrow T(n) = T(n/4) +$

Using Master's Theorem,

$T(n) = T(n/2) + c$

$a = 1 \quad b = 2 \quad f(n) = c$

$n^{\log_2 1} = n^0 = 1 = \text{constant}$

$T(n) = \text{constant}$

so $\boxed{T(n) = O(\log n)}$

lower bound → upper bound

```
min_max ( a, l, b) {
    max = a[0];
    min = a[0];
    for (i=1 ; i<n ; i++){
        if ( a[i] > max){
            max = a[i];
        }
        else if ( a[i] < min){
            min = a[i];
        }
    }
}
```

Worst Case Here will be when there will be most number of comparisions. ie. both the conditions if and else if will run.

- This will happen for a decreasing array. 2(n-1) comparisions

Best Case Here will be when there will be least number of comparisions. ie only the first conditions will run.

- This will happen for an increasing array. (n-1) comparisions

Q]. Write the divide and conquer method for the min_max.

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

// write code for returning both the variables

```
        else {
            min = a[j];
            max = a[i];
        }
        return (max, min);
    }
    else {
        mid = (i+j)/2;
(max1, min1) = DAC_max_min(a, i, mid);        } T(n/2)
(max2, min2) = DAC_max_min(a, mid+1, j);      } T(n/2)
```

dividing part

Conquering part

```
if (max1 > max2) {
    max = max1;
} else {
    max = max2;
}
if (min1 < min2) {
    min = min1;
} else {
    min = min2;
}
return (max, min);
}
```

→ In-place Algorithm & Out-place Algorithm

- In-Place Algorithm
  - → An in-place algorithm is one that transforms the input using a constant amount of extra space.
  - → It modifies the input directly without needing to allocate additional space proportional to the input size.

Eg: Quick sort
Bubble sort

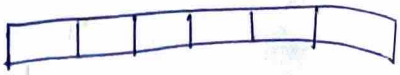
- Out-of-Place Algorithm
  - → An out-of-place algorithm requires extra space proportional to the size of the input.
  - → It does not modify the input directly but creates new structures to store the result.

Eg: Merge sort
Array Copying

→ Merge Sort

nlogn

```
void merge ( int arr[], int l, int m, int r)
    n1 = m-l+1;
    n2 = r-m;
    int L[] = new int [n1];
    int R [] = new int [n2];
    for (int i=0 ; i<n1 ; ++i)
        L[i] = arr[l+i];
    for (int j=0 ; j<n2 ; ++j)
        R[j] = arr[m+1+j];
    i=0 ; j=0;
    int k=l
    while (i<n1 && j<n2) {
        if ( L[i] <= R[j]){
            arr [k] = L[i];
            i++;
            k++;
        }
    }
    while (j<n2){
        arr [k] = R[j];
            j++;
            k++;
        }
    }
```



m=2

```
void sort ( int arr[], int l, int r) {
        if ( l<r){
            m = l+(r-l)/2 ;
            sort (arr, l,m);
            sort (arr, m+1, r);
            merge ( arr, l, m, r) ;
        }
}
```

- Time complexity for divide & conquer method for min-max

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

(Number of comparisions)

this is less than previously calculated $(2n-2)$ number of comparisions without using divide and conquer.

Q]: Show that $\left(\frac{3n}{2}-2\right)$ number of comparisions are necessary in the worst case to find both minimum & maximum elements of n numbers.

Q]: finding power of an element (a^n) (a → constant, n → variable) without and with using divide & conquer.

Using substitution method,

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

$$\Rightarrow T(n) = 2\{2T(n/4)+2\}$$

$$\Rightarrow T(n) = 4T(n/4) + 4 \quad \text{---(i)}$$
$$= 4T(n/4) + c$$

$$T(n/4) = 2T(n/8) + 2$$

$$\Rightarrow T(n/2) = \cancel{2\{}(2T(n/8)+2) + 2$$
$$= 4\,2T(n/4) + 2$$
$$= 2\left(2T(n/8)+2\right) + 2$$
$$= 4T(n/8) + 6$$

$$\Rightarrow T(n) = 2T(n/2) + 2$$
$$= 2(4T(n/8)+6) + 2$$
$$= 8T(n/8) + 14 \quad \text{---[ii]}$$
$$= 8T(n/8) + c$$

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

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

$\Rightarrow n = 2^k$

$\Rightarrow \boxed{k = \log_2 n}$

Quick sort (06/08/2024)

Avg $O(n \log n)$

worst $O(n^2)$

- Divide takes Time

```
QS(A, i, j) {
    if(i==j){
        return (a[i]);
    }
    else{
        m=partition (a,i,j);
        QS(a,i, m-1);
        QS(a, m+1, j);
        return a;
    }
}

partition (A, i, j) {
    p = A[i];
    k = i;
    for (q=i+1; q<=j; q++){
        if (p>= A[q]){
            k = k+1;
            swap (A[k] , A[q]);
        }
    }
    swap(A[k] , A[i]);
    return (k);
}
```

4 6 7 8 9

p ↑

i

Eg: 45, 90, 54, 88, 30, 25, 17, 75, 69, 12

* Best case and worst case will all be same ie $O(n)$.

$$T(n) = \begin{cases} O(1) & \text{if } n=1 \\ T(n-1) + T(j-n) + n \end{cases}$$

Recurrence Relation for Quick Sort

Best case

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

~~Best case~~, when array is divided into two equal parts (when median is selected), it will give best case time complexity.

Worst case, when array is divided unevenly (eg left side has 0 elements and right side has n elements)
so, n elements will have n time complexity (for partition function).
so, it will take $O(n^2)$ time complexity.

~~Worst Case~~

→ When the pivot element is median of array, the array gets divided equally. (Best Case)

→ (Worst Case) $O(n^2)$ sorted array
$T(n-1) + n$

Q]: Suppose in quick sort, the $n/5^{th}$ smallest element are selected as pivot element using $O(\log n)$ time complexity. Then what will be worst case time complexity of quick sort?

→ ~~For selecting $n/5^{th}$ element, it takes $\log n$ time.~~

It will split the array in $n/5$ and $4n/5$ element.
So, time complexity is

$$T(n) = T(4n/5) + T(n/5) + O(\log n)$$

$T(n/5)$ is much smaller than $T(4n/5)$

So, $T(n) = T(4n/5) + O(\log n)$

Using Master's theorem,

$a = 1$, $b = 5/4$ $\quad f(n) = \log n$

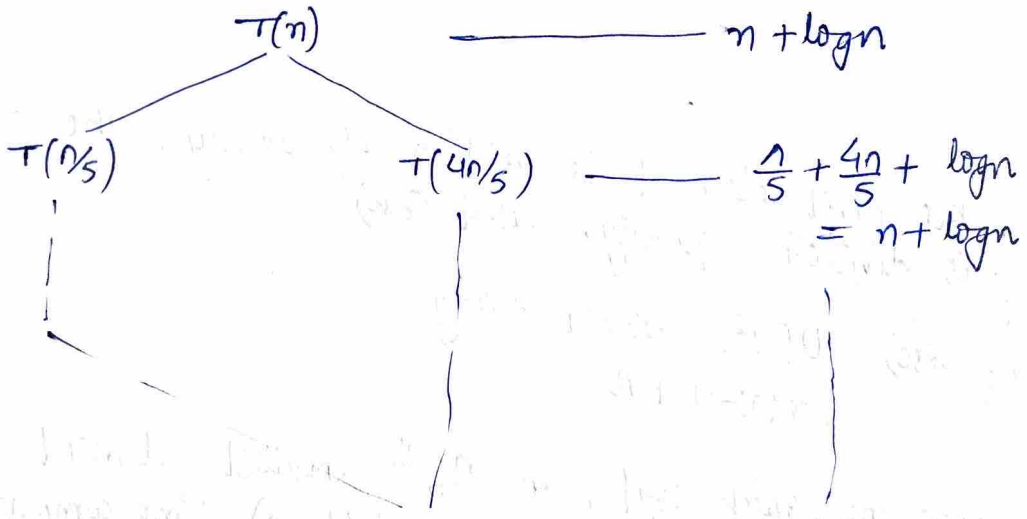
$$n^{\log_b^a} = n^{\log_{5/4}^1} = n^0 = 1$$

$\because f(n) = \log n > 1$; for $n > 2$

Therefore, second case of master's theorem will be applied

→ $T(n) = T(n/5) + T(4n/5) + n + \log n$

Using recursive tree method,



$T(n) = kn + k\log n$

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

$= n\log n + (\log n)^2$

$\because n\log n > (\log n)^2$

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

$T\left(\frac{n}{(5/4)^k}\right) = 1$

$\Rightarrow \frac{n}{(5/4)^k} = 1$

$\Rightarrow n = (5/4)^k \Rightarrow \boxed{k = \log_{5/4}^n}$

Heap (10/08)

- Complete Binary Tree
- Heap Property

→ Applications of Min Heap & Max Heap

→ time complexity to find the maximum element in a min-heap tree is $O(n)$.

because we will just be checking in the leaf nodes whose number is $\lceil n/2 \rceil$ for n nodes in the heap.

Max-Heapify

```
BUILD_Max-heap(A){
        heap_size(A) ← length(A)
        for i ← ⌊length[A]/2⌋ to 1        → to go to the first parent node of the tree
            do Max_heapify(A, i)
}

Max_heapify(A, i){
        l ← LEFT(i)
        r ← RIGHT(i)
        if l <= heapsize(A) and A[l] > A[i]
            then largest = l
            else largest = i
        if r <= heapsize(A) and A[r] > A[largest]
            then largest ← r
        if (largest ≠ i)
            swap(A[i], A[largest])
            MAX_heapify(A, largest)
}
```

GREEDY (12/08)

A greedy algorithm is a problem-solving approach used in dsa that builds up a solution piece by piece, always choosing the next piece that offers the most immediate benefit or seems to be the best at the moment.

- The choice at the particular moment will be the one that looks best at the moment.

→ solution space
The entire set of possible solutions that can be considered for a given problem (over the given 'n' number of inputs).

→ Feasible solution
It is a potential solution to an optimization or decision problem that satisfies all the constraints or conditions imposed by the problem.

→ Optimal solution
It is one of the feasible solution which is giving optimal value.

- Application of Greedy method
  1. Fractional Knapsack
  2. Job sequencing with deadline
  3. Huffman Coding
  4. Minimum cost Spanning Tree
  5. Single source shortest Path

1. Fractional Knapsack

We are given n objects and a knapsack.
Object i has weight w_i and the knapsack has the capacity m.
If a fraction x_i, where $0 \leq x_i \leq 1$ of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned.
So, our objective is to obtain a filling of the knapsack that maximizes the total profit earned.

Constraint

Total weight of all the chosen objects to be atmost m.

ie $\max \sum_{i=1}^{n} p_i x_i$ subject to $\sum_{i=1}^{n} w_i x_i \leq m$

Q: $m = 20$ kg $\quad n = 4$

obj	obj 1	obj 2	obj 3	obj 4
weight	10 kg	8 kg	12 kg	5 kg
profit	200	250	300	180

Algo 1 Greedy about profit

obj 3 $\quad X = < \underset{obj1}{0}, \underset{obj2}{1}, \underset{obj3}{1}, \underset{obj4}{0} >$

Capacity is filled by obj 3 + obj 2

Profit $= (300 + 250) = 550$

After obj 3, Capa

After obj 2, capacity = 0 kg left

Algo 2 Greedy about weight

obj 4 $\quad X = < \underset{obj1}{7/10}, \underset{obj2}{1}, \underset{obj3}{0}, \underset{obj4}{1} >$

$\text{profit} = (180 + 250 + (\frac{7}{10} \times 200))$

$= 570$

Algo 3 Greedy about Profit/weight

	P/W
obj 1	20
obj 2	31.25
obj 3	25
obj 4	36

$X = < \underset{obj1}{0}, \underset{obj2}{1}, \underset{obj3}{7/12}, \underset{obj4}{1} >$

$\text{Profit} = (250 + (\frac{7}{12} \times 300) + 180)$

$= 250 + 175 + 180$

$= 605$

So the best suited here is profit/weight

Q: $n=7 \quad m=33$

obj	obj1	obj2	obj3	obj4	obj5	obj6	obj7
profit	50	70	30	60	20	55	45
weight	5	9	8	5	4	5	8
p/w	10	7.77	3.75	12	5	11	5.625

$X = < \underline{1}, \underline{1}, \underline{0}, \underline{1}, \underline{\frac{1}{4}}, \underline{1}, \underline{1} >$

Capacity $5+5+5+9+8+1 = 33$

$$\text{Profit} = 50+70+0+60+\left(20\times\frac{1}{4}\right)+55+45$$
$$= 285 \quad \underline{\text{Ans}}$$

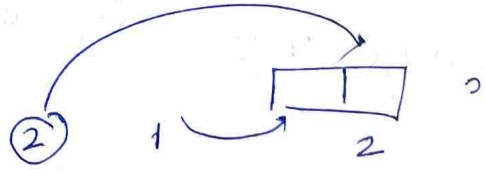
② Job sequencing with deadline

We are given a set of n jobs, each job i is associated with a deadline $d_i \geq 1$ and a profit $P_i > 0$.
For any job i, the profit P_i is earned iff the job i is completed by its deadline
To complete a job, one has to process, the job on a machine for one unit of time, and only one unit is available for the processing; a feasible solution for this problem is a subset J of jobs, such that each job in this subset can be completed by its deadline.
Objective is to maximize the profit.

Q: $n=4$

Jobs:	J_1	J_2	J_3	J_4
profit:	200	250	150	350
deadline:	2	1	2	1

J_4	J_2	J_1	J_3
1	1	2	2



Sets of solutions (possible)

(J_1, J_2)	J_2, J_1	$P = 250+200 = 450$
(J_2, J_3)	J_2, J_3	$P = (250+150) = 400$
(J_1, J_3) X	J_4, J_1	$P = (350+200) = 550$
(J_1, J_4)	J_4, J_3	$P = (350+150) = 500$
(J_2, J_4) X		
(J_3, J_4)		

This method is very costly (in terms of time).
Therefore, we cannot proceed with these kind of algorithms

```
if ( deadline equal)          ——— (1)
    return

else{
  for(i=0; i<n-1; i++) {   for(j=0; j<n; j++) {
  if(di ≪ dj){                  (n²)
    do dj;
  }  do dj;
  else {
       do dj;
  }    do di;
}
```

Sort all the jobs in decreasing order of their profits.
For each job, do the following:

→ Find the time slot i such that slot is empty and $i < \text{deadline}_{\text{maxm}}$ and i is greatest.

→ Put the job in this slot and mark this slot filled.

→ If no such i exists, then ignore the job.

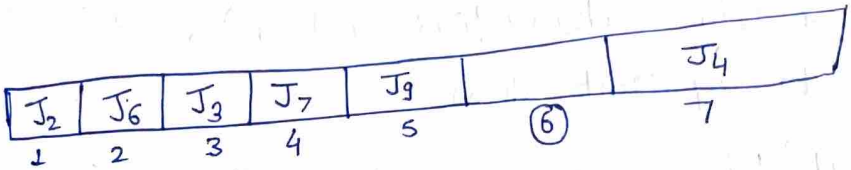
• Use merge sort for sorting here — nlogn —

Q1: n=9

job:	J_1	J_2	J_3	J_4	J_5	J_6	J_7	J_8	J_9
deadline:	3	5	4	7	5	2	4	1	5
profit:	25	45	55	20	40	50	80	15	60

decreasing order of profit

J_7	J_9	J_3	J_6	J_2	J_5	J_1	J_4	J_8
4	5	4	2	5	5	3	7	1

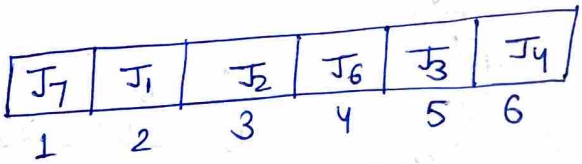


Q1: n=7

Jobs	J_1	J_2	J_3	J_4	J_5	J_6	J_7
deadline	3	3	5	6	4	5	1
Profit	70	90	80	60	50	75	95

J_7	J_2	J_3	J_6	J_1	J_4	J_5
1	3	5	5	3	6	4

Max^m deadline = 6



Max^m profit = (95 + 70 + 90 + 75 + 80 + 60)

= 470

③ Huffman Coding

Eg: A = 45 B = 2 C = 15 D = 7 E = 30 F = 1

A – 65 – 01000001
B – 66 – 01000010
C – 67 – 01000011
D – 68 – 01000100
E – 69 – 01000101
F – 70 – 01000110

Total characters = 45 + 2 + 15 + 7 + 30 + 1
= 100

∴ Total bits = (8 × 100) = 800 bits

This will take a lot of time to get sent.

Uniform Coding (Fixed length coding)

Therefore, we need to compress the data

∵ we have only 6 characters (A–F), we need maxm 3 bits to represent them

∴ Total bits here = (3 × 100) = 300 bits

Also we need to send the chart

000 – A
001 – B
010 – C
011 – D
100 – E
101 – F

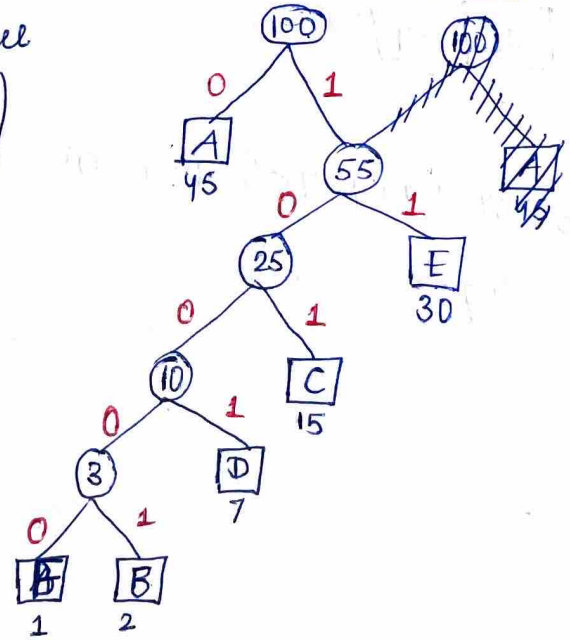
6 (characters) × 8 (ASCII size) = 48 + 3 (bits) × 6 (characters) = 18

∴ (48 + 18) = 66 bits for the chart

So, the total bits used here are (300 + 66) = 366 bits

Non-Uniform Coding (Variable length coding)

Huffman Tree (Binary tree Gadha!)



A = 0
B = 10001
C = 101
D = 1001
E = 11
F = 10000

So cool

Total number of bits

$= (1 \times 45) + (5 \times 2) + (3 \times 15) + (4 \times 7) + (2 \times 30) + (5 \times 1)$

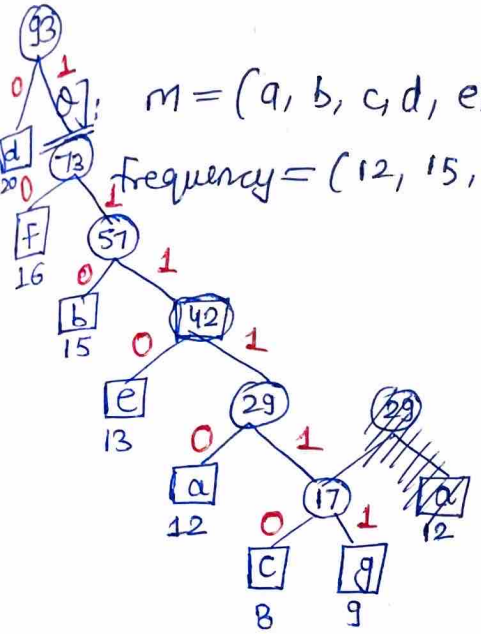
$= 45 + 10 + 45 + 28 + 60 + 5$

$= 193$

For chart, (6×8) (ASCII) $+ (1+5+3+4+2+5)$ (for bits)

$= 48 + 20$

$= 68$



$m = (a, b, c, d, e, f, g)$

Frequency $= (12, 15, 8, 20, 13, 16, 9)$

$a = 11110$

$b = 110$

$c = 111110$

$d = 0$

$e = 1110$

$f = 10$

$g = 111111$

Total number of bits

$= (12 \times 5) + (15 \times 3) + (8 \times 6) + (20 \times 1) + (13 \times 4) + (16 \times 2) + (9 \times 6)$

$= 60 + 45 + 48 + 20 + 42 + 32 + 54$

$= 301$

For chart $= (7 \times 8) + (5+3+6+1+4+2+6)$

$= 56 + 27$

$= 83$

Average $= \left(\frac{83}{7}\right) =$

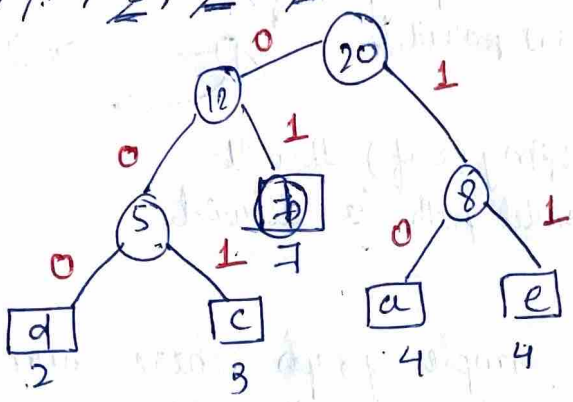
Q1: abbcd bccd aabbeeebeab

a — 4
b — 7
c — 3
d — 2
e — 4

Total characters = 20

a	b	c	d	e
4	7	3	2	4

8, 8, 12



a — 10
b — 01
c — 001
d — 000
e — 11

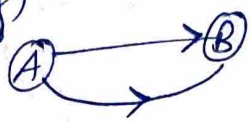
Total number of bits
$= (4\times2) + (7\times2) + (3\times3) + (2\times3) + (4\times2)$
$= 8 + 14 + 9 + 6 + 8$
$= 45$

For chart $= (5\times8) + (2+2+3+3+2)$
$= 40 + 12$
$= 52$

∴ Total number of bits used $= 45 + 52$
$= 97$

GRAPHS

- Simple graph — No looping (self)
  No parallel



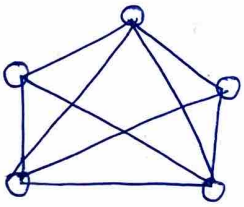
- Multigraph — Looping (self) allowed
  Parallel path is allowed

* Null graph is a simple graph where number of edges is zero
  (We can have multiple nodes though)
  ∴ Minimum ~~vertex~~ degree = 0

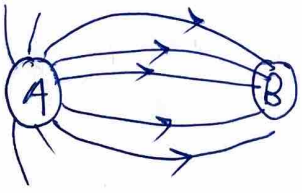
* Complete graph is a simple graph in which each vertex is connected to the other remaining nodes.
  ∴ Maximum degree = (n-1) if n is the total number of vertices

- K_n (Complete ~~Connected~~ graph of n vertices)

Eg: K_5



- Minimum degree in a multi-graph is 0 (null graph)
- Maximum degree in a multi-graph is infinite.



- No. of edges in $K_n = {}^nC_2 = \frac{n(n-1)}{2}$
- No. of subgraphs $= 2^{\text{edges}} = 2^{\frac{n(n-1)}{2}}$

~~④ Minimum Co~~

• Spanning Tree

→ A sub-graph S of the given graph G is said to be spanning tree iff i) S should contain all the vertices of the given graph G.

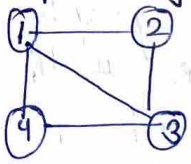
~~S~~

ii) S should contain V-1 edges where V is the number of vertices and S must not contain any cycle

(iii)

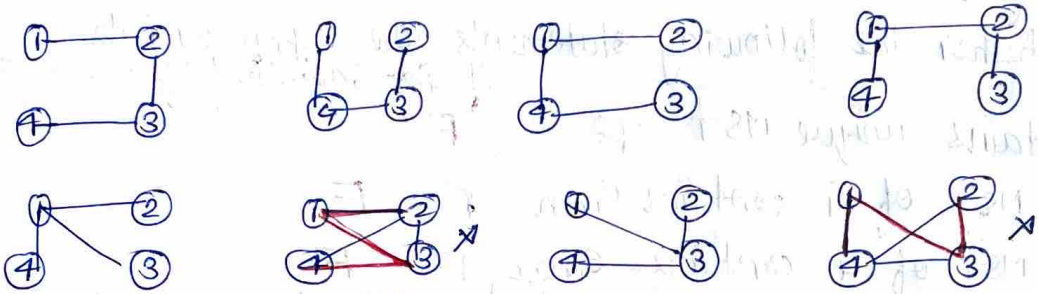
* No. of spanning trees for a **complete graph** K_n is n^{n-2}.

Eg:



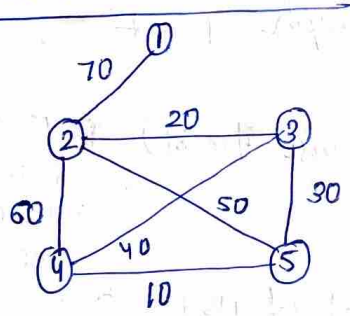
G

8 spanning trees will be formed



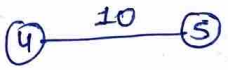
~~④~~ Minimum cost Spanning Tree

Q:

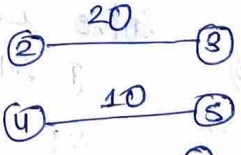


Find MCST using **Kruskal's Algorithm**

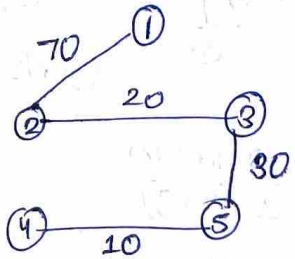
① Pick the minimum edge



② Take the next minimum edge (it should not form a cycle)



similarly

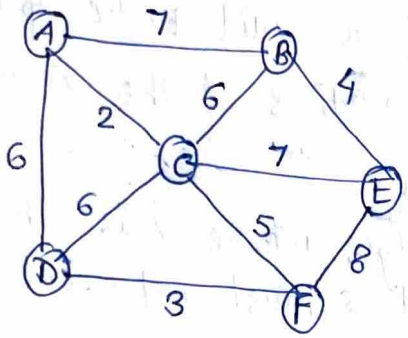


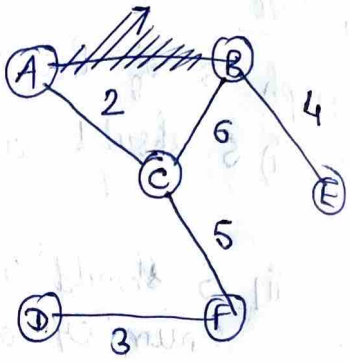
5 vertices
4 edges

∴ This is the MCST

cost of MCST is (70+20+30+10) = 130

Q]: Consider the following graph
Find MST using Kruskal's Algorithm





Cost of MST = (3+5+2+6+4)
= 20

Q]: Let G be an undirected connected weighted graph with distinct edge weights and e_{min} be the edge ~~width~~ with minimum weight and e_{max} be the edge with maximum edge weight.

check whether the following statements are True or False:
if ~~for~~ indistinct edges are present

① G contains unique MST ~~T~~ F

② Every MST of G contains e_{min} ~~T~~ F

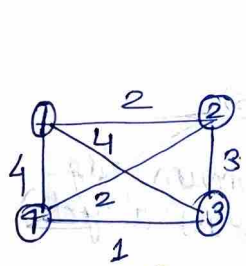
③ Every MST of G contains e_{max} F F

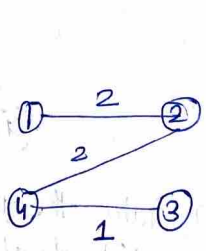
④ If e_{max} is in the MST, then removal of e_{max} from the graph G must disconnect the graph ~~T~~ F

⑤ MST of G may contain e_{max} ~~T~~ T

⑥ MST of G may contain e_{min} F (because it must) ~~F~~ T (because it might not include all the e_{min}'s)

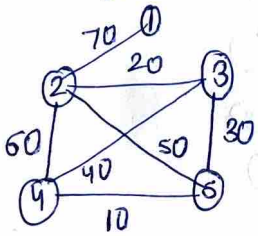
Q]:





cost of MST = 5

Q]:

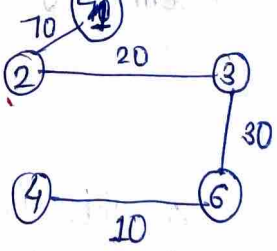


Solve using **Prim's Algorithm**

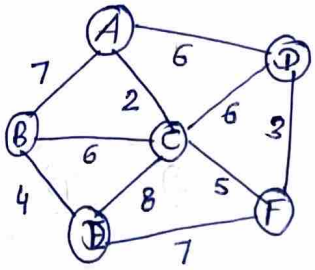
① Select a source
Let us take a vertex ②

② No we will take the minimum edge from

③ Take the minimum from ② & ③ both now.
Whatever edge is the minimum, chose it.



Q1:

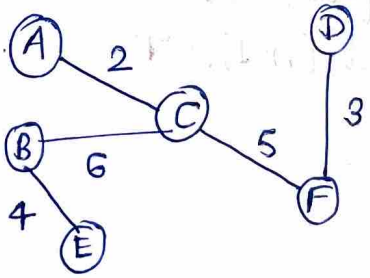


Find MST using Prim's Algo using min priority Queue / min Heap Tree

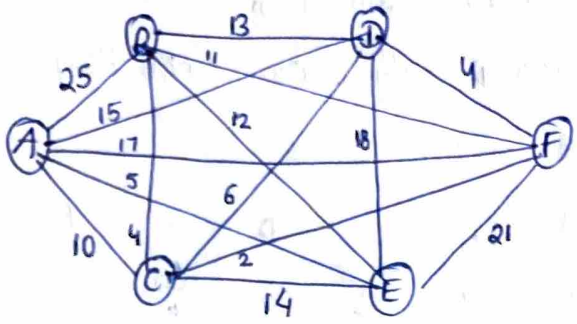
	A	B	C	D	E	F
A	0	7	2	6	∞	
B		7				
C		2				
D		6				
E						
F						

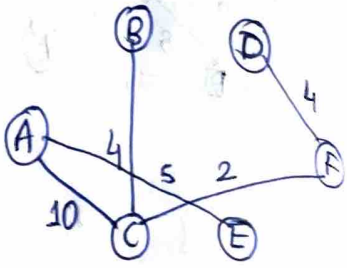
	A	B	C	D	E	F
Key	0	∞	∞	∞	∞	∞
Parent	N	N	N	N	N	N
A		7 A	2 A	6 A	∞ N	∞ N
C		6 C		6 A	8 C	5 C
F		6 C		3 F	7 F	
D		6 C			7 F	
B					4 B	
E						

* extracting from min heap = Taking the min value (constant time) + deleting that node (logn time)

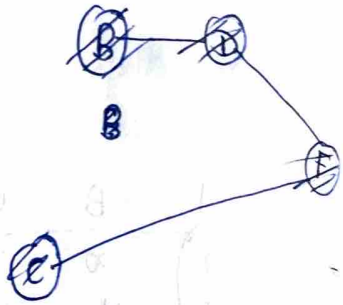


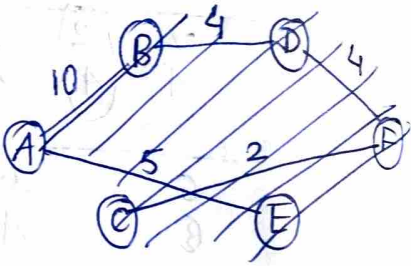
Q1:





	A	B	C	D	E	F
Key	∞	∞	0	∞	∞	∞
Parent	N	N	N	N	N	N
C	10 C	4 C		6 C	14 C	2 C
F	10 C	4 C		4 F	14 C	
D	10 C	4 C			14 C	
B	10 C				12 B	
A					5 A	
E						





Q1: Let G be an undirected, connected unweighted graph with n vertices whose adjacency matrix is given below

All diagonal elements are zero

All non-diagonal elements are one.

So check whether the following are true or false:

1. Graph G contains unique MST. F
2. G contains multiple MST each of different cost. F
3. G contains multiple MST each of cost (n-1). T
4. G does not have any MST. F

Q: Prove that $\log E = O(\log V)$ where E is no. of edges and V is no. of vertices in a ~~complete~~ simple graph.

⇒ $\because E \leq \frac{V(V-1)}{2}$ for a simple graph

⇒ $E = c \cdot \frac{V(V-1)}{2}$

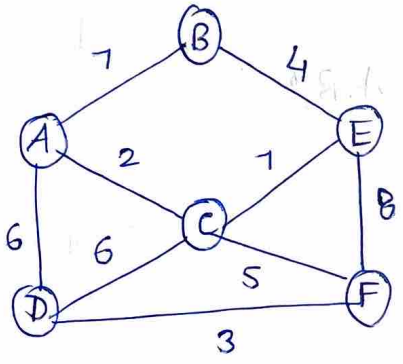
⇒ $E = O(cV^2)$

⇒ $E = O(V^2)$

Taking log on both sides,

$\log E = 2 \log V \Rightarrow \boxed{\log E = O(\log V)}$

- $O(V \log E) = O(V \log V)$
- $O(E \log E) = O(V^2 \log E)$ $\left\{ \begin{array}{l} O(E) \neq O(V) \\ O(E) = O(V^2) \end{array} \right\}$



$S_1 = \{A\}$
$S_2 = \{B\}$
$S_3 = \{C\}$
$S_4 = \{D\}$
$S_5 = \{E\}$
$S_6 = \{F\}$

For each vertex we need to call Makeset function.

Sort ~~them~~ edges in non-decreasing order by weight.

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

Take ~~one~~ the first edge from the above list and find the representative sets of A & C.

If Find(A) ≠ Find(C) $S_1 = \{A, C\}$
Union(A,C)

Now, perform the same for the other edges as well.

(D,F) If Find(D) ≠ Find(F) $S_4 = \{D, F\}$
Union(D,F)

(B,E) If Find(B) ≠ Find(E)
Union (B,E) $S_2 = \{B, E\}$

(C,F) If Find (C) ≠ Find (F)
Union (C,F) $S_1 = \{A, C, D, F\}$
$\{S_1 \leftarrow S_1 \cup S_4\}$

(C,D) X If Find (C) ≠ Find (D)
(But this condition isn't true)
So we will not take union

(A,D) X

(A,B) If Find (A) ≠ Find (B) $S_1 = \{A, B, C, D, E, F\}$
Union (A, B)
$\{S_1 \leftarrow S_1 \cup S_2\}$

(C,E) X

(E,F) X

So we are getting 5 edges.
ie $\{(A,C), (D,F), (B,E), (C,F), \cancel{(C,D)}, A,B\}$

Kruskal's Algorithm

```
Kruskals_Algo {
    for each V∈V[G]                    } O(V)
        do Make_set[V]
    sort the edges in non-decreasing order } O(ElogE)
    for each (u,v) from the list
        if (Find(u) ≠ Find(v))         } O(ElogV)
            Union(u,v)
```

Assignment
Find how this will be the time complexity for this case.

Sorting edge takes ElogE time ~~which is equal to~~

$= E = V^2$

$\log E = 2 \log V$

$\log E = O(\log V)$

$\Rightarrow E \log V$

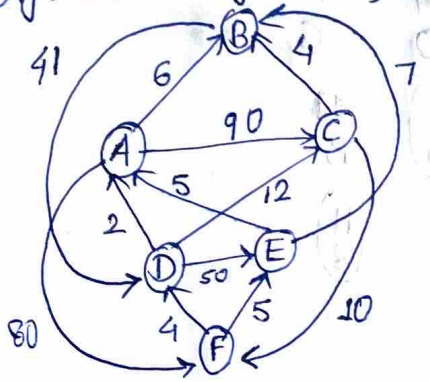
Find union will take $O(\alpha(V))$ which is practically constant.

As find (U,V) is very small and will take near constant time.

So, time complexity is ElogV.

⑤ SINGLE SOURCE SHORTEST PATH (31/08/24)

(Dijkstra's Algorithm)



if $(d[u] + C(u,v) < d[v])$

$d[v] = d[u] + C(u,v)$

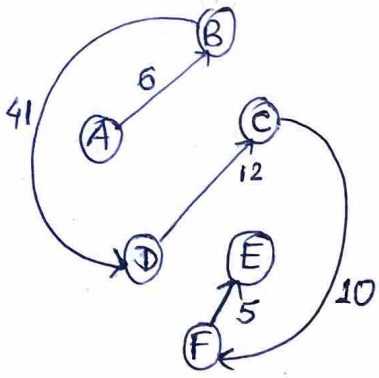
$d[v] \rightarrow$ ~~direct path~~ distance of vertex from A

$d[u] + C(u,v) < d[v]$

intermediate bet'n V & A

cost of edge (u,v)

	A	B	C	D	E	F
	0 N	∞ N	∞ N	∞ N	∞ N	∞ N
		6 A	90 A	∞ N	∞ N	~~80~~ A
A			90 A	47 B	∞ N	~~80~~ A
B			59 D	~~59~~ D	97 D	80 A
D					91 D	69 C
C						74 F
F						
E						



Q]: Minimum cost for A → F

A → B → D → C → F

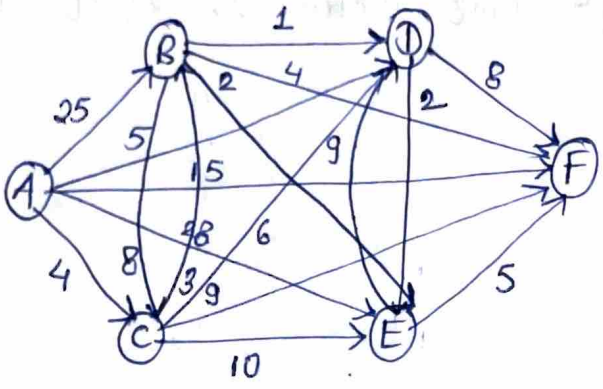
(69)

Q]: Minimum cost for A → E

A → B → D → C → F → E

(74)

Q):



(i) Write the output sequences of the vertices identified by the Dijkstra's Algorithm when the algo started from vertex A.

(ii) What will be the cost of the shortest path A → F?

(iii) What will be the shortest path from A → F?

(I)

	A	B	C	D	E	F
	0 A	∞ N	∞ N	∞ N	∞ N	∞ N
A		25 A	4 A	5 A	28 A	15 A
C		7 C		5 A	14 C	13 C
D		7 C			~~14~~ D	13 C
B					7 D	~~11~~ B
E						11 B
F						

(II) 11

(III) A → C → D → B → E → F

Prims Algorithm

```
MST_Prims (G, w, r) {
    for each u ∈ G.V {
        key[u] = INFINITY
        P[u] = NULL
    }                                   } O(V)
    key[r] = 0
    Q = G.V                  Q is a min-heap tree } using Build heap method } O(V)
    while Q ≠ ϕ {            (V times)
        s ← Extract_min(Q)   taking the value and deleting the node  } O(log V)
        for each v ∈ Adj[s] {        (E times)
            if v∈Q and w(s,v) < key[v] {
                P[v] = s             } O(log V)
                key[v] = w(s,v)      } O(log V)
            }
        }                                   } O(E log V)
    }
```

do not multiply by V because it only runs E time in total which we already addressed

$O(V \log V) + O(E \log V)$

$$TC = O(V) + O(V) + O(V \log V) + O(E \log V)$$
$$= O((V+E) \log V)$$
$$= O(E \log V)$$

Dijkstra's Algorithm

```
Dijkstra( G, sourcenode){
    for each u ∈ V[G] {                          } O(V)
        distance[u] = INANITY
        Previous[u] = NULL
    }
    distance[source_node] = 0
    Q = G.V                                      } O(V)
    while Q is not empty{
        N ← Extract-min(Q)                       } O(log V)
        Mark N visited
        for each unvisited Node M of N {          (adjacent unvisited)
            temp_dist = distance[N] + C(N,M)
            if temp_dist < distance[M]{          } O(E log V)
                distance[M] = temp_dist          } O(log V)
                previous[M] = N
            }
        }
    }
}
```

$$TC = O(V) + O(V) + O(V \log V) + O(E \log V)$$
$$= O((V+E) \log V)$$

Q.① Describe the partition function of Quick Sort and also write the step-by-step method to get the pass 1 result by taking last elements as pivot on the following data.

2, 5, 7, 5, 9, 3, 8, 6

Q.② State and explain Master's Theorem and use this method to give tight asymptotic bounds for the following recurrence relations:

i) $T(n) = 4T(n/5) + n^2$

ii] $T(n) = T(\sqrt{n}) + 1$

Q.③ Write a C function to arrange the numbers stored in an array as follows
Odd numbers followed by even numbers.
The complexity (time) should be order of n $O(n)$.

Sol: ①

Sol: ② Master's theorem gives us a method to solve recurrence relation.

→ Master's theorem can be applied to the following type of recurrence relation

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

where, a, b are constants
$a \geq 1$ & $b > 1$
$f(n)$ is a positive function

Case ① If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$
Then, $T(n) = \theta\left(n^{\log_b a}\right)$

Case ② If $f(n) = \theta\left(n^{\log_b a + \epsilon}\right)$
Then, $T(n) = \theta(f(n))$

Case ③ If $f(n) = \theta\left(n^{\log_b a}\right)$
Then, $T(n) = \theta\left(n^{\log_b a} * \log n\right) = \theta(f(n) * \log n)$

(i) $T(n) = 4T(n/5) + n^2$

Here, $a=4$ $b=5$ $f(n)=n^2$

$$n^{\log_b a} = n^{\log_5 4} < n^2$$

$\therefore T(n) = \theta(n^2)$

[ii] $T(n) = T(\sqrt{n}) + 1$

~~Here, $a=1$ / $b=$~~

Let $\boxed{n = 2^m} \Rightarrow m = \log_2 n$

$$T(2^m) = T\left(2^{m/2}\right) + 1$$

Let $T(2^m)$ be $S(m)$

$$S(m) = S(m/2) + 1$$

Here, $a=1$ & $b=2$ $f(m)=1$

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

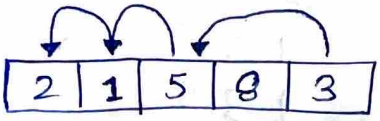
So, $S(m) = \theta(1 \cdot \log m)$

$1 = 1$

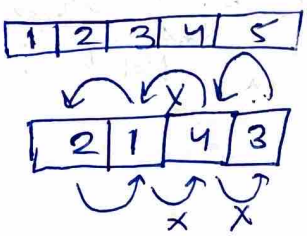
$\Rightarrow T(2^m) = \theta(m \log m)$

$\Rightarrow T(n) = \theta(\log(\log_2 n))$

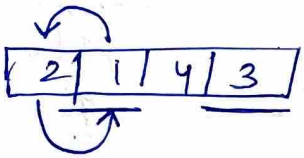
Sol ③ Odd followed by even numbers
Time complexity should be $O(n)$



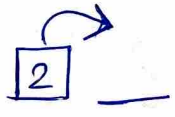

```
for (i=0; i<n; i++) {
    k++;
    if ( arr[i] % 2 != 0) {   //checking for odd
        int temp = arr[i];
        arr[i-k] = arr
```



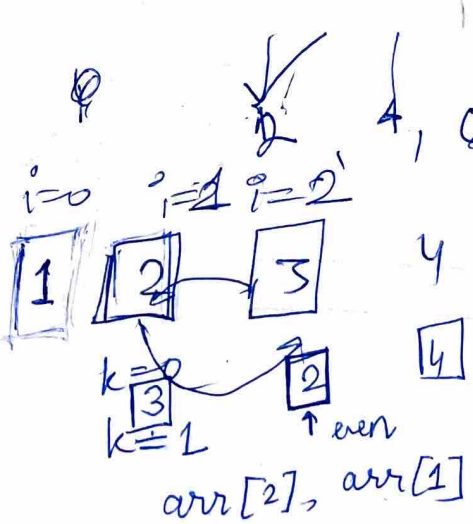
k=1 k=3 k=0 k=2
0 2 3



~~k=i else(k++~~



arr[1-1=0] = arr[i];
~~k=0~~;
arr[i-k] = arr[i]



Algorithms of 4 Sorting Techniques

- → Selection sort
- → Bubble sort
- → Insertion sort
- → Merge sort
- → Quick sort
- → Heap sort

→ Selection sort $O(n^2)$

- Select the smallest element from the unsorted portion and swap it with the current element.

```
int min;
for (int i=0; i<n-1; i++) {
    min = i;
    for (int j=i+1; j<n; j++) {
        if (arr[j] < arr[min]) {
            min = j;
        }
    }
    if (min != i) {
        swap(arr, i, min);
    }
}
```

★ Call By Value & Call by Reference

Actual Parameters – The parameters passed to a function

Formal Parameters – The parameters recieved by a function

```
add (m, n);
```
actual parameters

```
int add (int a, int b) {
    return (a+b);
}
```
formal parameters

★ Call By Value

Here, the values of actual parameters are copied to formal parameters

& these two (actual & formal) store values in different memory locations.

```
int x=10, y=20;
fun (x,y);
```
stored at a diff place

```
int fun (int x, int y) {
    x=20;
    y=10;
}
```
these are local to these function

stored at a diff place

★ Call By Reference

Here, both actual and formal parameters refer to the same memory location.

Therefore, any change to the formal parameters will get reflected to actual parameters.

→ Instead of passing value, we pass addresses.

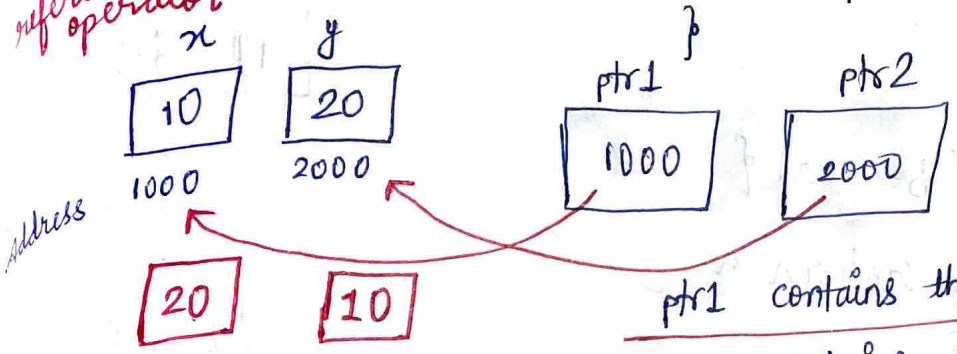
- **Pointers** are those variables that can store addresses.

```
int x=10, y=20;
fun(&x, &y);
```

reference operator

```
int fun(int *ptr1, int *ptr2){
    *ptr1 = 20;
    *ptr2 = 10;
}
```

dereference operator



ptr1 contains the address of x

*ptr1 contains the value at the address of x.

Bubble Sort $O(n^2)$

- works by repeatedly swapping the adjacent elements if they are in the wrong order.

→ Firstly, the largest element gets sorted and is stored at the last position. Then, we do the same for the remaining $n-1$ elements.

```
int k=n;
while (k>0) {
    int i=0;
    while (i<k-1) {
        if (arr[i] > arr[i+1]) {
            swap(arr, i, i+1);
        }
        i++;    // to keep on comparing and swapping adjacent elem
    }
    k--;        // moving the remaining unsorted array
}
```

```
int i, j, temp;
boolean swapped;
for (i=0; i<n-1; i++) {
    swapped = false;
    for (j=0; j<n-i-1; j++) {
        if (arr[j] > arr[j+1]) {
            // swap arr[j] and arr[j+1]
            swapped = true;
        }
    }
    if (!swapped) break;
}
```

reducing by i because after 'i' passes the last 'i' elements are already sorted.

→ Insertion sort $O(n^2)$ Best $O(n)$

- works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list.

→ It is a stable sorting algorithm, meaning that elements with equal values maintain their relative order in the sorted output.

```
int n = arr.length;
for (int i=1; i<n; i++) {
    int key = arr[i];          // the element to be inserted into the sorted array
    int j = i-1;
    while (j>=0 && arr[j] > key) {          // move elements that are greater than key to one position right
        arr[j+1] = arr[j];
        j--;
    }
    arr[j+1] = key;          // inserting the key at the right position
}
```