INTRODUCTION

* An algorithm is evaluated on the basis of:
(1) Time Complexity
(2) Space Complexity.

* Time Complexity: The order of magnitude of time it takes to run an algorith as a function of its input size.

Space Complexity: The amount of space consumed by the algorithm throughout its execution as a function of its input size.

* For any algorithm, the time complexity can be tested in three possible cases:
(1) Best Case
(2) Worst Case
(3) Average Case

For each case the T.C. can be reported in any of: O, Ω, Θ notations.

T.C. of RECURSIVE PROGRAMS

* Repeated Substitution

Ex. $T(n) = \begin{cases} T(n-1)+1, & n \geq 1 \\ 1, & n=0 \end{cases}$

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

$\Rightarrow (T(n-2)+1)+1$

$\Rightarrow (T(n-3)+1)+2$

$\Rightarrow T(n-3)+3$

$\vdots$

$\Rightarrow T(n-K)+BK.$

when, $n-K=0 \Rightarrow n=K.$

$\Rightarrow T(0)+n$

$\Rightarrow n+1$

Ex. $T(n) = \begin{cases} T(n-1)+n, & n \geq 2. \\ 1, & n=1 \end{cases}$

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

$\Rightarrow (T(n-2)+n-1)+n.$

$\Rightarrow (T(n-3)+n-2)+n+(n-1)$

$\Rightarrow T(n-4)+n+(n-1)+(n-2)+(n-3)$

$\vdots$

$\Rightarrow T(n-K)+n+(n-1)+(n-2)+\ldots+(n-K+1)$

when, $n-K=1 \Rightarrow K=n-1$

$\Rightarrow 1+n+(n-1)+(n-2)+\ldots(n-(n-1)+1)$

$\Rightarrow n+(n-1)+(n-2)\ldots+2+1$

$\Rightarrow \frac{n(n+1)}{2}$

Ex. $T(n) = \begin{cases} 2T(n/2) + n\log n, & n > 1 \\ 1, & n = 1 \end{cases}$

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

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

$\Rightarrow 2^2 T\left(n/2^2\right) + n\log(n/2) + n\log n$

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

$\Rightarrow 2^3 T(n/2^3) + n\log(n/2^2) + n\log(n/2) + n\log(n)$

$\vdots$

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

when $\frac{n}{2^k} = 1 \Rightarrow k = \log n$.

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

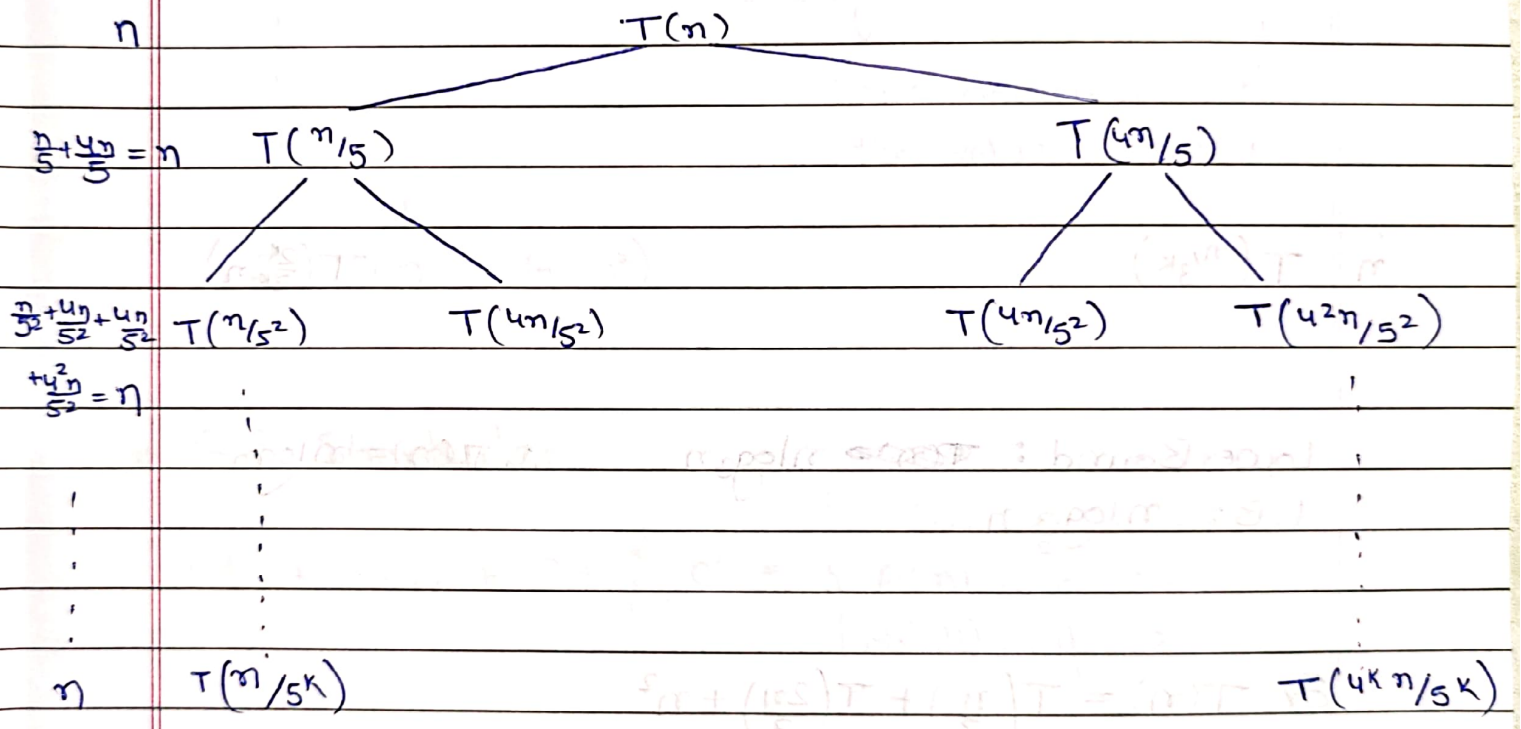
$\Rightarrow n + n\left[\log n \times \log n - \frac{k(k-1)}{2}\right]$

$\Rightarrow n + n\log^2 n - \frac{n\log^2 n}{2} + \frac{n\log n}{2}$

$\Rightarrow n + \frac{n\log^2 n}{2} + \frac{n\log n}{2}$

* Tree Method.

Ex. $T(n) = T(n/5) + T(4n/5) + n$.



when, $\frac{n}{5^K} = 1$

$\Rightarrow K = \log_5 n$

when, $\frac{4^K n}{5^K} = 1$

$\Rightarrow K = \log_{\frac{5}{4}} n$

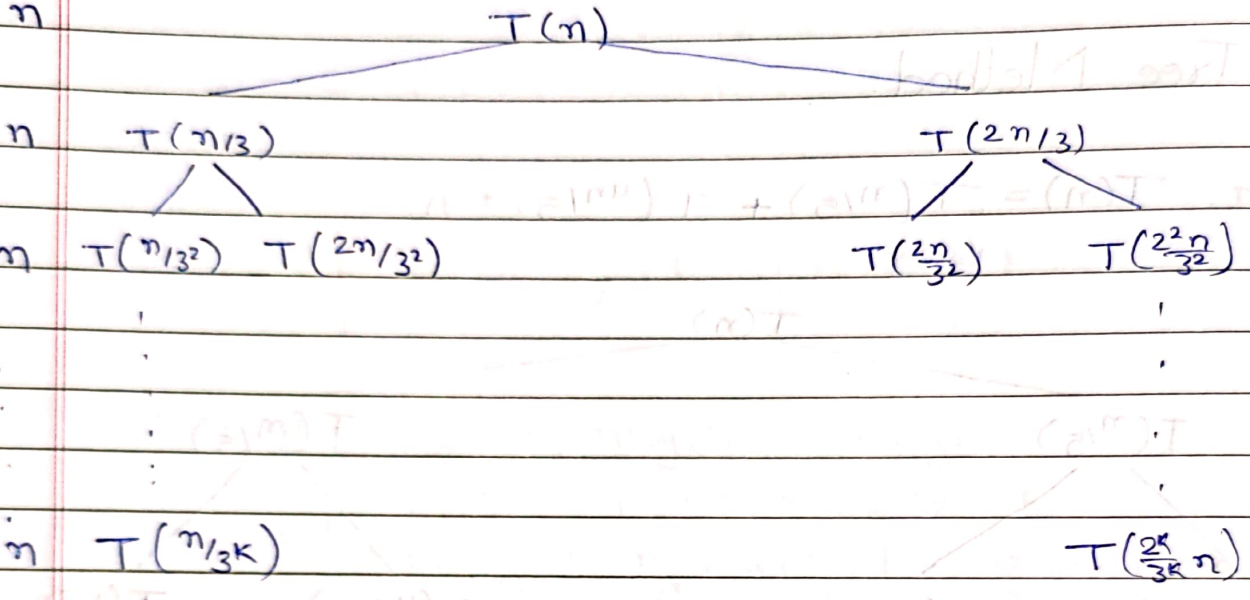
So, $\frac{4^K n}{5^K}$ branch is longer than $n/5^K$ branch.

Lower bound: $n \times \log_5 n$ Upper Bound: $n \times \log_{5/4} n$

$\therefore n\log_5 n \leq T(n) \leq n\log_{\frac{5}{4}} n$.

$\therefore T(n) = \Omega(n\log n)$ & $T(n) = O(n\log n) \Rightarrow T(n) = \Theta(n\log n)$

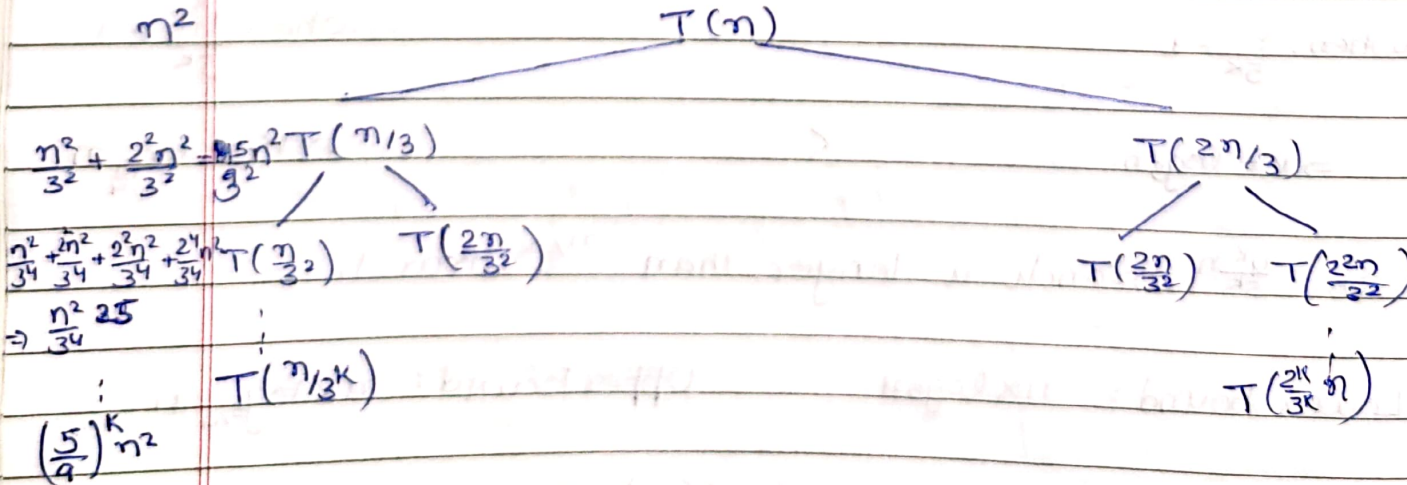
Ex. $T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + n$



Lower Bound : $n\log_3 n$ $\quad \therefore T(n) = n\log n$

UB : $n\log_{\frac{3}{2}} n$.

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



~~Lower Bound~~ $\left(\frac{5}{9}\right)^0 n^2 + \left(\frac{5}{9}\right)^1 n^2 + \left(\frac{5}{9}\right)^2 n^2 + \dots + \left(\frac{5}{9}\right)^K n^2$

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

LB: $(K = \log_3 n) \Rightarrow n^2 \left(\frac{1 - (5/9)^{\log_3 n}}{4/9}\right)$

↓

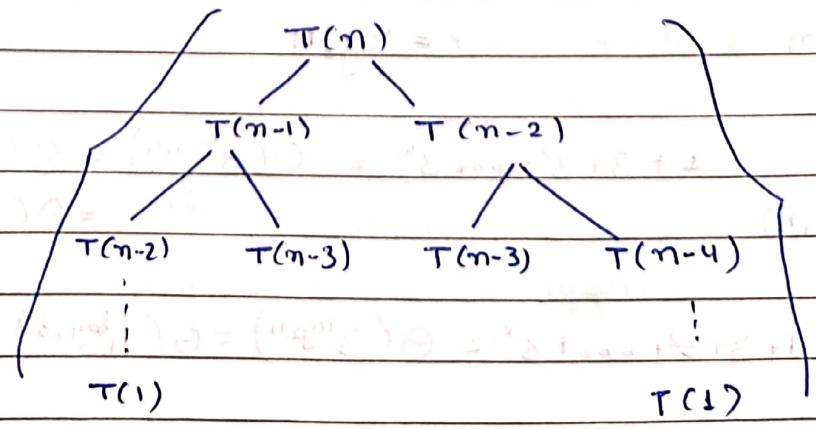
Since decreasing GP, the value of expr. is a constant.

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

* GP Shortcuts:

$$1 + c + c^2 + \dots + c^n = \sum_{i=1}^{n} c^i = \begin{cases} \Theta(1), & \text{if } c < 1 \\ \Theta(n), & \text{if } c = 1 \\ \Theta(c^n), & \text{if } c > 1 \end{cases}$$

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



LB: $2^0 + 2^1 + 2^2 + \dots 2^K = 2^{n/2+1} - 1$

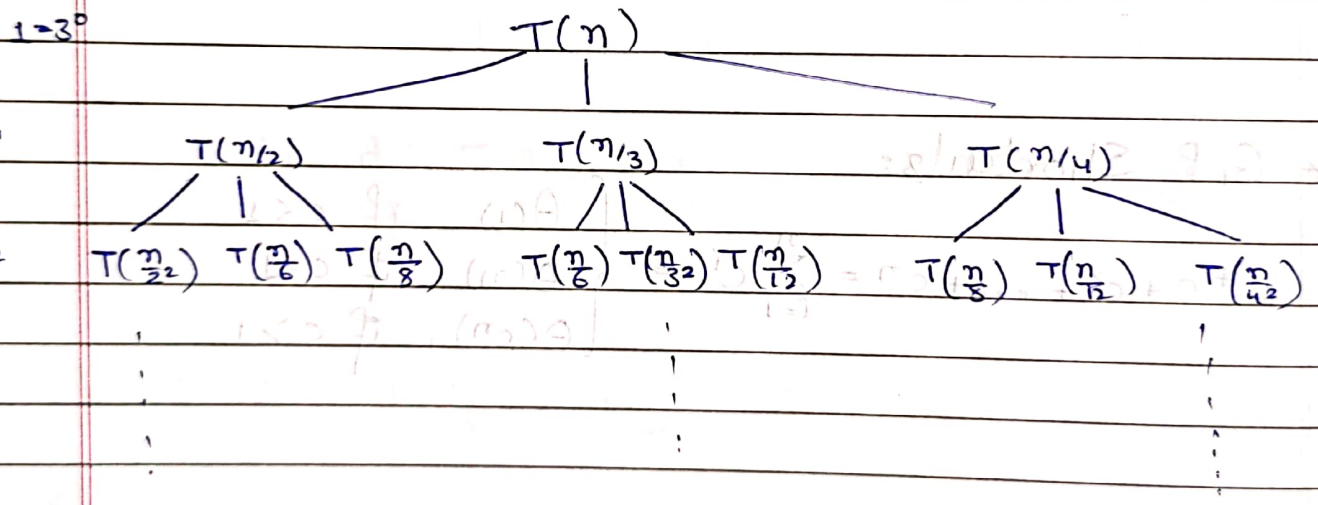
$(K = \frac{n}{2})$

UB: $2^0 + 2^1 + 2^2 + \dots + 2^K = 2^n - 1$

$(K = n)$

$\therefore \Omega(2^{n/2}) \leq T(n) \leq \Theta(2^n)$

Ex. $T(n) = T(n/2) + T(n/3) + T(n/4) + 1$



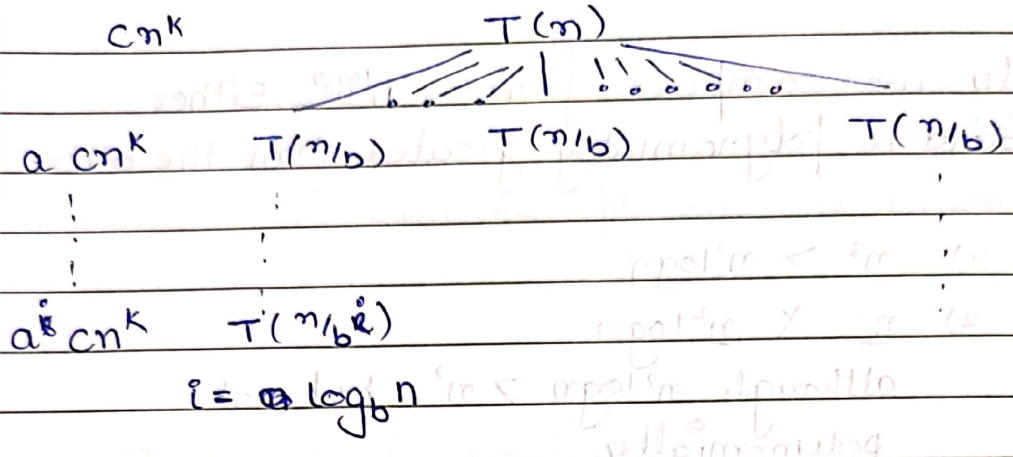
$K = \log_2 n$ $\quad K = \log_3 n$ $\quad K = \log_4 n$

LB: $1 + 3 + 3^2 + \dots 3^K = \Theta(3^{\log_4 n}) = \Theta(n^{\log_4 3})$

$(K = \log_4 n)$ $\quad = \Theta(n^{0.79})$

UB: $1 + 3 + 3^2 + \dots + 3^K = \Theta(3^{\log_2 n}) = \Theta(n^{\log_2 3}) = \Theta(n^{1.5})$

Ex. $T(n) = aT(n/b) + cn^k$.



$i = \log_b n$

$T(n) = cn^k (a^0 + a^1 + \dots + a^i)$

$\Rightarrow cn^k \Theta(a^{\log_b n})$

$\Rightarrow cn^k \Theta(n^{\log_b a})$

* Master Theorem

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

$\{a \geq 1,\ b > 1\}$

C1: $f(n) = O(n^{\log_b a - \epsilon})$, for some constant $\epsilon > 0$, then,

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

C2: $f(n) = \Theta(n^{\log_b a})$, then,

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

CS: $f(n) = \Omega(n^{\log_b a + \epsilon})$, $\epsilon > 0$, then,

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

Basically we compare $f(n)$ & $n^{\log_b a}$. Either one should be polynomially greater than the other one.

Ex: (1) $n^3 > n^2 \log n$

(2) $n^2 \times n^2 \log n$

although $n^2 \log n > n^2$, but not polynomially.

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

$n^{\log_b a} = n^2 \quad < \quad f(n) = n^3$

$\therefore T(n) = \Theta(n^3)$

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

$n^{\log_b a} = n \qquad f(n) = 2^n$

$2^n > n^3$ & $n^3 > n$

$\therefore 2^n > n$

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

Ex. $T(n) = 9T\left(\frac{n}{3}\right) + n^2 \log n$

$n^{\log_b a} = n^2 \qquad f(n) = n^2 \log n$

n^2 $\quad\quad$ $n^2 \log n$

$\Rightarrow 1$ $\quad\quad$ $\Rightarrow \log n$

Not polynomially comparable.

$\therefore$ Can't apply master's theorem.

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

$n^{\log_b a} = n^{\log_3 2}$

$\simeq n^{0.6}$

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

$n^{0.6} > n^{0.2}$ & $n^{0.2} > (\log n)^2$

$\therefore n^{0.6} > (\log n)^2$

$\therefore T(n) = \Theta(n^{\log_3 2})$

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

$1, \quad n = 1$

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

$\Rightarrow 2^2 T\left(\frac{n}{2^2}\right) + \frac{n}{\log n/2} + \frac{n}{\log n}$

$\Rightarrow 2^3 T\left(\frac{n}{2^3}\right) + \frac{n}{\log n/2^2} + \frac{n}{\log n/2} + \frac{n}{\log n}$

$\vdots$

$\Rightarrow 2^K T\left(\frac{n}{2^K}\right) + \frac{n}{\log n} + \frac{n}{\log n/2} + \frac{n}{\log n/2^2} + \cdots + \frac{n}{\log n/2^{K-1}}$

when, $\frac{n}{2^K} = 1$ $\quad K = \log n$.

$$\Rightarrow 2 + \frac{n}{\log n} + \frac{n}{\log \frac{n}{2}} + \frac{n}{\log \frac{n}{2^2}} + \cdots + \frac{n}{\log \frac{n}{2^{\log n - 1}}}$$

$$\Rightarrow 2 + n\left(\frac{1}{\log 2^K} + \frac{1}{\log 2^{K-1}} + \frac{1}{\log 2^{K-2}} + \cdots + \frac{1}{\log 2^1}\right)$$

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

{ HP with $d = 1$, $a = 1$ }

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

$$\Rightarrow 2 + n\log(2\log n + 1)$$

$\therefore T(n) = \Theta(n \log\log n)$

- Complete Master Theorem

$$T(n) = aT\left(\frac{n}{b}\right) + n^K \log^p n, \quad a \geq 1, b > 1, K \geq 0, p \in R$$

C1: $a > b^K$ or $\log_b a > K$, then
$T(n) = \Theta(n^{\log_b a})$

C2: $a = b^k$ or $\log_b a = k$

C2.1: $p > -1$, $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$

C2.2: $p = -1$, $T(n) = \Theta(n^{\log_b a} \log\log n)$

C2.3: $p < -1$, $T(n) = \Theta(n^{\log_b a})$

C3: $a < b^k$ or $\log_b a < k$

C3.1: $p \geq 0$, $T(n) = \Theta(n^k \log^p n)$

C3.2: $p < 0$, $T(n) = O(n^k)$

Ex. $T(n) = 2T\left(\frac{n}{2}\right) + \left(\frac{n}{\log n}\right)$, $n > 1$

1, $n = 1$

$n^{\log_b a} = n$ $\quad a=2,\ b=2,\ k=1,\ p=-1.$

$a = b^k$ & $p = -1 \Rightarrow T(n) = \Theta(n \log\log n)$

Ex. $T(n) = 4T(n/2) + n/\log^2 n$

$n^{\log_b a} = n^2$ $\quad a=4,\ b=2,\ k=1,\ p=-2.$

$a > b^k \Rightarrow T(n) = \Theta(n^2)$

Ex. $T(n) = T(\sqrt{n}) + \log n$, $n > 2$

1, $n = 2$

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

$\Rightarrow T(n^{1/2}) + \log n$

$\Rightarrow T(n^{1/2^2}) + \log n^{1/2} + \log n$

⋮

$\Rightarrow T(n^{1/2^k}) + \log n + \log n^{1/2} + \log n^{1/2^2} \ldots + \log n^{1/2^{k-1}}$

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

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

$\Rightarrow \log n = 2^k$

$\Rightarrow k = \log\log n$

$\Rightarrow 1 + \log(n \times n^{1/2} \times n^{1/2^2} \times \ldots \times n^{1/2^{k-1}})$

$\Rightarrow 1 + \log(n^{1 + \frac{1}{2} + 1/2^2 \ldots + 1/2^{k-1}})$

$\Rightarrow 1 + \log(n^{1\left(\frac{1-(1/2)^k}{1-1/2}\right)})$

~~$\Rightarrow 1 + \log(n^{2\log n})$~~

$\Rightarrow 1 + \log(n^{2\left(\frac{\lg n - 1}{\lg n}\right)})$

$\Rightarrow 1 + 2 \times \frac{(\lg n - 1)}{\lg n} \times \lg n$

$\Rightarrow 2\log n - 1$

$\therefore T(n) = \Theta(\log n)$

Ex. $T(n) = T(\sqrt{n}) + \sqrt{n}$.

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

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

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

$\Rightarrow T(n^{1/2^3}) + n^{1/2^3} + n^{1/2^2} + n^{1/2}$

$\vdots$

$\Rightarrow T(n^{1/2^K}) + n^{1/2} + n^{1/2^2} + n^{1/2^3} + \dots + n^{1/2^K}$

when $n^{1/2^K} = 2$

$\Rightarrow K = \log\log n$

$\Rightarrow \Theta(1) + \Theta(n^{1/2})$

$\therefore T(n) = \Theta(\sqrt{n})$

Ex. $T(n) = T(\sqrt{n}) + \log\log n$

$T(n) = T(\sqrt{n}) + \log\log n$.

Let $n = 2^m \Rightarrow m = \log n$. So, $T(2^m) = T(2^{m/2}) + \log\log 2^m$

& $S(m) = T(2^m)$

i.e.

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

$\Rightarrow S(m/2^2) + \log\log 2^{m/2} + \lg\lg 2^m$

$\Rightarrow S(m/2^3) + \lg\lg 2^{m/2^2} + \lg\lg 2^{m/2} + \lg\lg 2^m$

$\vdots$

$\Rightarrow S(m/2^K) + \lg\lg 2^m + \lg\lg 2^{m/2} + \lg\lg 2^{m/2^2} + \dots \lg\lg 2^{m/2^{K-1}}$

when, $\frac{n}{2^k} = 1 \Rightarrow k = \lg n$.

$\Rightarrow \Theta(1) + \lg n + \lg \frac{n}{2} + \lg \frac{n}{2^2} + \cdots + \lg \frac{n}{2^{k-1}}$

$\Rightarrow \Theta(1) + \lg \frac{n^k}{2^{\frac{k(k-1)}{2}}}$

$\Rightarrow \Theta(1) + \lg \frac{n^{\lg n}}{2^{(\lg n - 1)/2 \times \lg n}}$

$\Rightarrow \Theta(1) + (\log n)^2 - \frac{\lg n (\lg n - 1)}{2}$

$\Rightarrow (\log\log n)^2 - \frac{(\log\log n)^2}{2} + \frac{\log\log n}{2}$

$\Rightarrow \frac{(\log\log n)^2}{2} + \log\log n$

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

Ex. $T(n) = T(\sqrt{n}) + n^2$

Let, $n = 2^m$. $\Rightarrow m = \log n$ & $T(2^m) = T(2^{m/2}) + 2^m$

&

$S(m) = T(2^m)$

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

Applying Master's Theorem.

$n^{\log_b a} = n^{\log_2 1} = 1$ $\qquad f(n) = 2^{2n}$

$1 < n < 2^{2^n} n$

$\therefore f(n) > n^{\log_b a}$

$\therefore S(n) = \Theta(2^{2n})$

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

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

$\Rightarrow \Theta(n^2)$

Ex. $T(n) = 12T(n^{1/3}) + (\log n)^2$

let $n = 2^m$, $m = \lg n$, $T(2^m) = 12T(2^{m/3}) + m^2$

$S(m) = T(2^m) = 12S(\frac{m}{3}) + m^2$

$n^{\log_b a} = n^{\lg_3 12} = n^{2.26}$ $\qquad >$ $\qquad f(n) = n^2$

$\therefore S(n) = T(2^n) = \Theta(n^{\lg_3 12})$

$\therefore T(n) = \Theta((\log n)^{\lg_3 12})$

DIVIDE & CONQUER

* DEC solutions comprise of three steps:
  (1) Divide
  (2) Conquer
  (3) Combine.

* Maximum of array:

```
Maximum (Arr[], l, h)
  if (l==h) return A[l];

  m = l + (h-l)/2;
  m1 = Maximum (Arr, l, m);
  m2 = Maximum (Arr, m+1, h);
  return max (m1, m2);
```

T.C. Analysis:

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

By Master's method

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

* Merge Sort

- Divide an array in two halves.
- Sort recursively each half.
- Combine the two halves.

- mergesort (A[], l, h)
  if (l == h)
    m = (l+h)/2
    mergesort (A, l, m)
    mergesort (A, m+1, h)
    merge (A, l, m, h)

- merge (A[], l, m, h)

  LA = A[l...m]
  RA = A[m+1...h]
  i = j = 0
  K = 0
  while (i < LA.length) and (j < RA.length)
    if (LA[i] < RA[j]
      A[K++] = LA[i++]
    else
      A[K++] = RA[j++]

  while (i < LA.length)
    A[K++] = LA[i++]
  while (j < RA.length)
    A[K++] = RA[j++]

T.C. Analysis :

$T(1) = \Theta(1)$

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

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

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

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

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

$\vdots$

$\Rightarrow 2^k T\left(\frac{n}{2^k}\right) + n + n + \ldots \{k-1 \text{ times}\}$

when $\frac{n}{2^k} = 1 \Rightarrow k = \log n.$

$\Rightarrow n \times \Theta(1) + (k-1) \times n.$

$\Rightarrow k \times n$

$\Rightarrow n \log n.$

$\therefore T(n) = \Theta(n \log n)$

Ex. Consider a variant of merge sort: we split the array in three parts, recursively sort then merge them.

(A) what is the total no. of comparisions performed in worst case, while merging three sorted arrays each of size $n/3$?

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

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

$\therefore$ Total $= \frac{3n}{3} - 2 = n - 2$

(B) Write the recurrence relation.

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

Ex. Suppose merge process takes $\Theta(1)$ time. Find complexity of merge sort.

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

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

Ex. Consider MS divides Array such that left part consists of singe element & right of remaining elements.

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

Ex. Consider MS which divides array into K equal parts where K is constant. Find the running time of the algorithm in worst case.

In worst case comparing K sorted arrays of size n/K takes $K(n-1)$ steps. $\therefore \Theta(n)$ {K is const}

$$\therefore T(n) = KT\left(\frac{n}{K}\right) + \Theta(n)$$

$$\therefore T(n) = \Theta(n \log n)$$

Ex. Consider MS which divides array into two parts at random. What is worst case running time of the algo?

In worst case,

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

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

Ex. An array A is called bitonic if $\exists t$ such that $A[1 \ldots t]$ is increasing & $A[t+1 \ldots n]$ is decreasing. Give an efficient algo. to sort A.

Find 't' by traversing A until the next value is smaller than current element in $\Theta(t)$.
Merge the two arrays.

$$T(n) = \Theta(t) + \Theta(n)$$
$$\Rightarrow \Theta(n) + \Theta(n)$$
$$\Rightarrow \Theta(n).$$

Ex. An array A is K-even-mixed if there are exactly K even integers in A & the odd ints appear in sorted order. Given a K-even-mixed array A, containing n distinct integers for $K = n/\log n$, describe an $O(n)$ algo to sort A.

Divide E & O arrays : $\Theta(n)$
Sorting Even array : $\Theta\left(n - n\frac{\log\log n}{\log n}\right) = \Theta(n)$
Merging Two arrays : $\Theta(n)$

$$\therefore T(n) = \Theta(n)$$

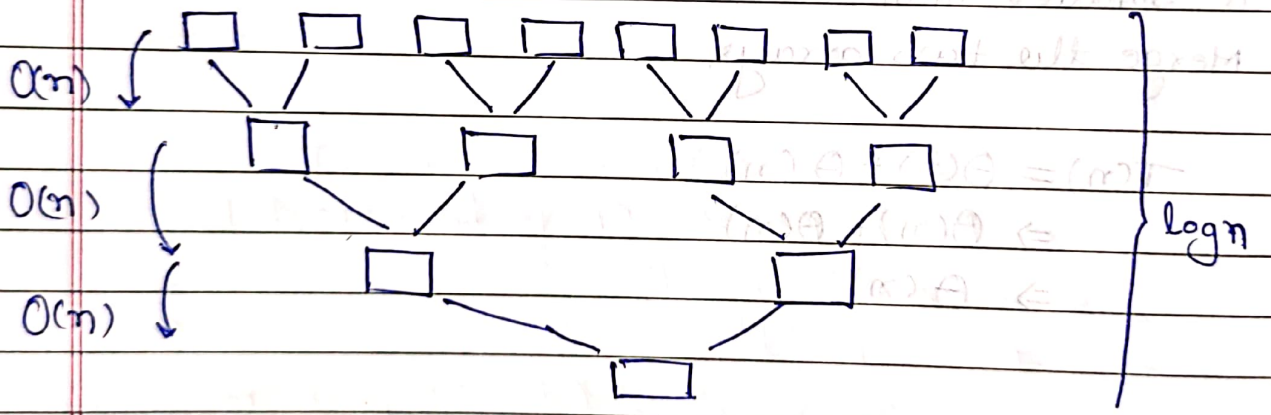
* Iterative Merge Sort

```
merge (A[], l, h)
    sz = 1

    while  sz < (h-l)
        for i=0; i < h-l ; i= i+sz
            md = i + sz/2;
            merge (A, l, md, i+sz-1)

        sz <<= 1
```

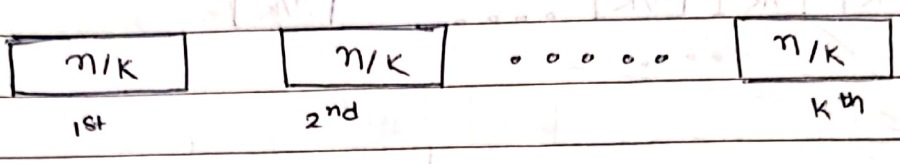
- T.C. Analysis



At each level $O(n)$ time. No. of levels $\log n$.

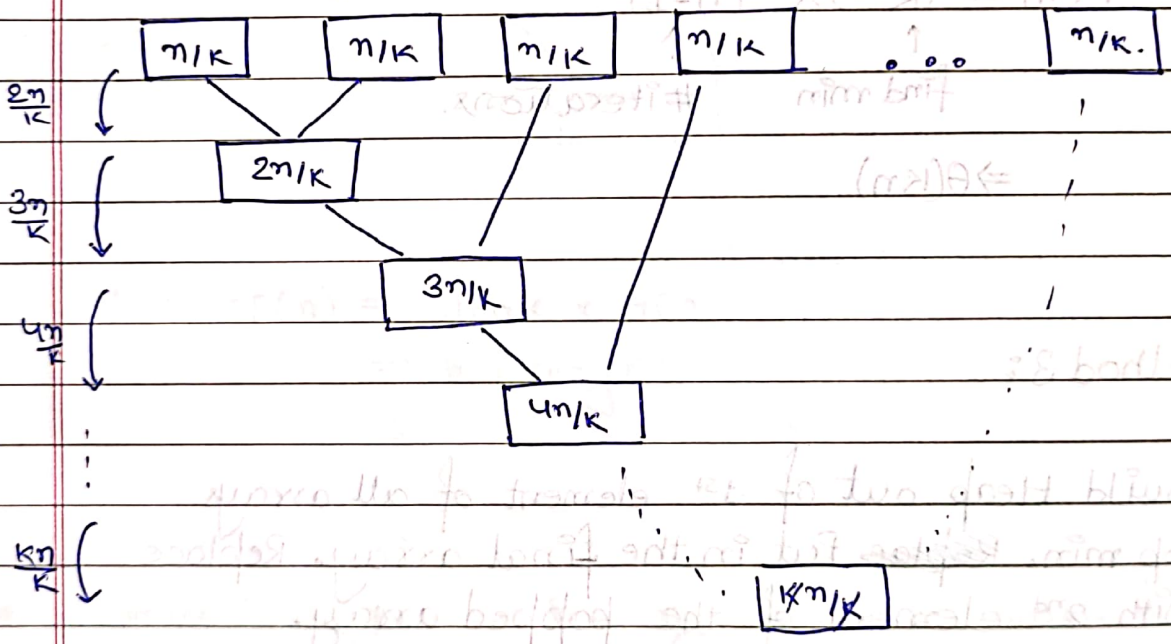
$\therefore T(n) = \Theta(n \log n)$

* Merging K Sorted Arrays



Method 1:

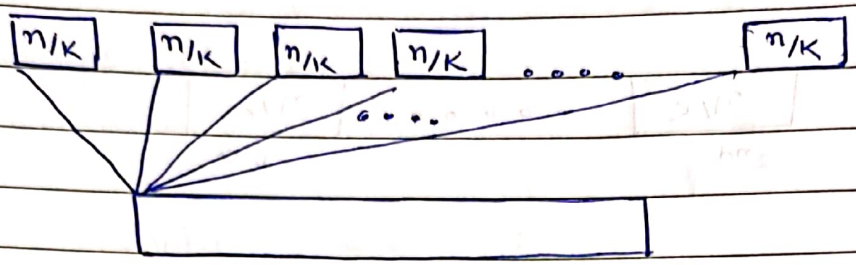
Using merge subroutine:



$$\therefore T(n) = \frac{n}{k}[2+3+4+\ldots K] = \frac{n}{k}\left(\frac{K(K+1)}{2} - 1\right)$$

$$\Rightarrow \Theta(nK)$$

Method 2:



At each step find min. of K arrays' current pointer & put in the sorted array.

$T(n) = (K-1) \times (n-1)$

↑ find min. ↑ #iterations.

$\Rightarrow \Theta(Kn)$

Method 3:

Build Heap out of 1st element of all arrays. Pop min. Put in the final array. Replace with 2nd element of the popped array.

Build Heap: $\Theta(K)$

At each step:

(1) PopHeap $\Theta(1)$

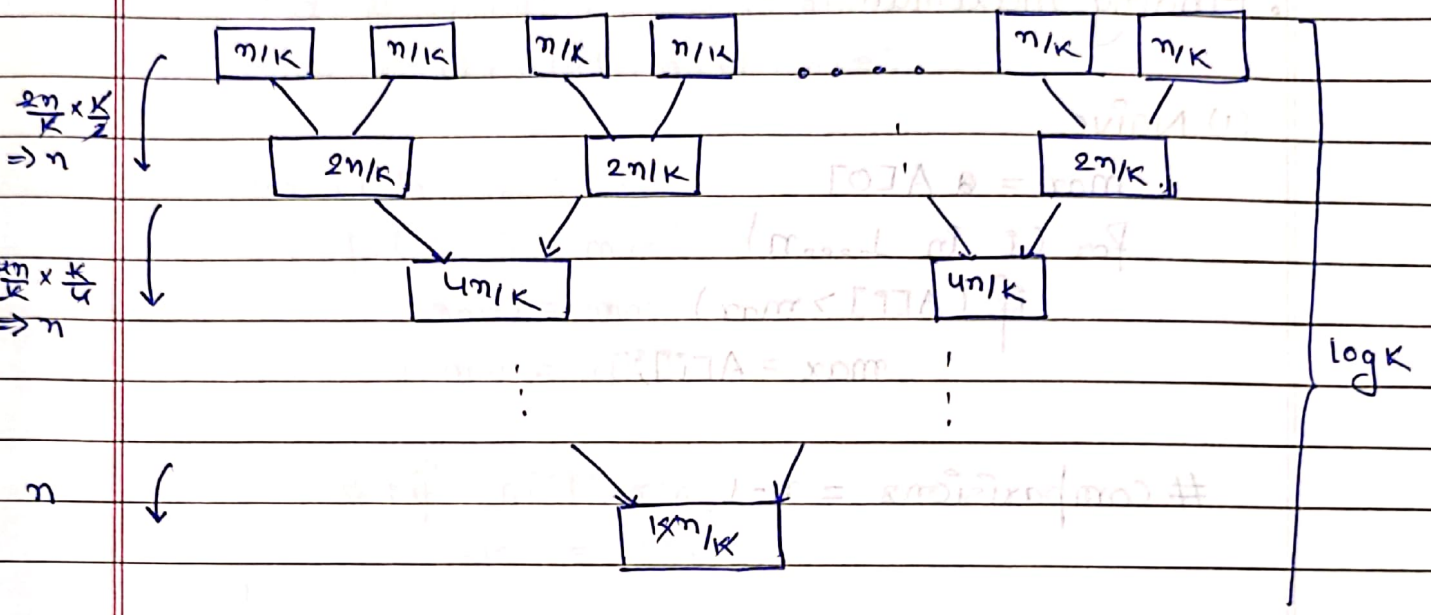
(2) PushHeap $\Theta(\log K)$

steps: $\Theta(n)$

$\therefore T(n) = \Theta(n \log K)$

Method 4:

Pairwise Merge.



$\therefore \quad T(n) = \log K \times \Theta(n)$

$\Rightarrow \Theta(n \log K)$.

* • Stable Sorting: A sorting algo. is called Stable if elements with same key appear in the same order after sorting as in the original array.

Ex. | 3_a | 1 | 2 | 3_b | 4 | 3_c | $\Rightarrow$ | 1 | 2 | 3_a | 3_b | 3_c | 4 |

• Inplace Algorithm: A sorting algo. is called inplace if it requires $\Theta(1)$ extra space to perform the operation.

* Maximum & Minimum

• Finding maximum

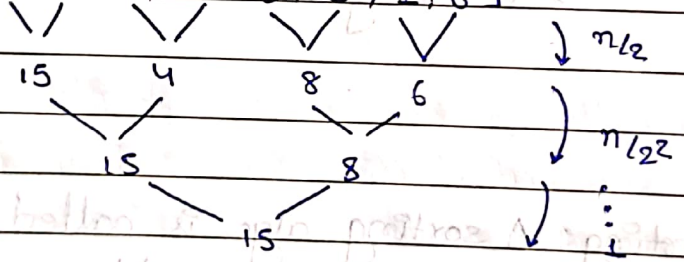
(1) Naive

```
max = A[0]
For (i in 1...n)
    if (A[i] > max)
        max = A[i];
```

comparisions = n-1

(2) Tournament Method

Ex. [10, 15, 2, 4, 8, 3, 1, 6]



$$\# \text{comparisions} = \frac{n}{2} + \frac{n}{2^2} + \cdots + \frac{n}{2^k} \quad (2^k = n)$$

$$\Rightarrow n \times \frac{1}{2}\left(\frac{1-\left(\frac{1}{2}\right)^k}{1/2}\right)$$

$$\Rightarrow n \times \left(1 - \frac{1}{n}\right)$$

$$\Rightarrow n - 1$$

• Finding max & second maximum

(1)

```
max = a[0] > a[1] ? a[0] : a[1];
smax = a[0] > a[1] ? a[1] : a[0];

for (i in 2...n) {
    if (a[i] > max) {
        smax = max
        max = a[i]
    }
    else if (a[i] > smax)
        smax = a[i]
}
```

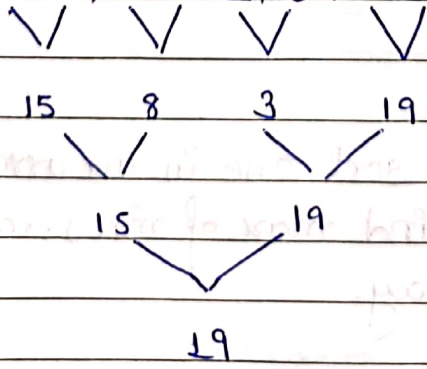
In worst case we do two comparisions.

∴ # comparisions = $(n-2) \times 2 + 1$

$\Rightarrow 2n - 1$

(2) Tournament Method

Ex. [10, 15, 8, 2, 1, 3, 6, 19]



Observationally, the second largest # will always only

get defeated by the largest #.

So, the second largest # will be on the path of largest #.

There are exactly $\log n$ no. of siblings on the path of largest #.

Ex. [6, 3, 15] in the examples.

S1: Find max using tournament method.
S2: Find smax among $\log n$ siblings of max.

$$\#\text{Comparisions} = (n-1) + (\log n - 1)$$
$$\Rightarrow n + \log n - 2.$$

- Finding max & min.

(1) Naive Method.

Assign A[0] as mx & mn both. Loop over all other elements & compare for both mx & mn.

$$\#\text{Comparisions} = 2(n-1) = 2n-2$$

(2)

For every pair sort one in mx arrays other in mn-array. Find max of mx-arrays min of mn-array.

Ex. [10, 5, 21, 23, 48, 32, 21, 6]

mx_arr = [10, 23, 48, 21] mx = 48

mn_arr = [5, 21, 32, 6] mn = 32.

$$\#\text{Comparisions} = \frac{n}{2} + \binom{n/2}{-1} + \binom{n/2}{-1} = \frac{3n}{2} - 2.$$

$$\simeq \left\lceil \frac{3n}{2} \right\rceil - 2.$$

* Counting Inversions

Two indices, i & j, such that $i < j$ form an inversion if $a[i] > a[j]$.

Ex. What is the largest no. of inversions possible for 6 len array?

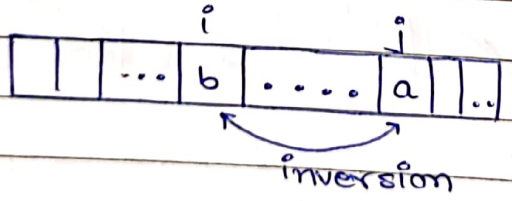
6 5 4 3 2 1

∴ # of Inversions = 5 + 4 + 3 + 2 + 1

⇒ 15

Ex. Let T[0...n-1] be a vector of integers. Show that if there is an inversion (i,j) in T, then T has at least j-i inversions.

Suppose,



i.e. $a < b$.

there can exist three types of values at index k, $i < k < j$:

① $A[k] < A[j]$

② $A[j] < A[k] < A[i]$

③ $A[i] < A[k]$

① a ② b ③

C1: $(A[K] < a \Rightarrow A[K] < b)$ & $i < K$
then (i, K) forms inversion.

C2: $a < A[K] < b$, & $K < j$
So, (K, j) form inversion.

C3: $a < b < A[K]$ & $K < j$
So, (K, j) form inversion.

∴ $\forall K, i < K < j$, K forms inversion either with i or j
So, # inversion is $\geq j - i$

- Method 1:
for each index i, iterate over each index j, $j > i$, if $A[i] > A[j]$, then count inversion.

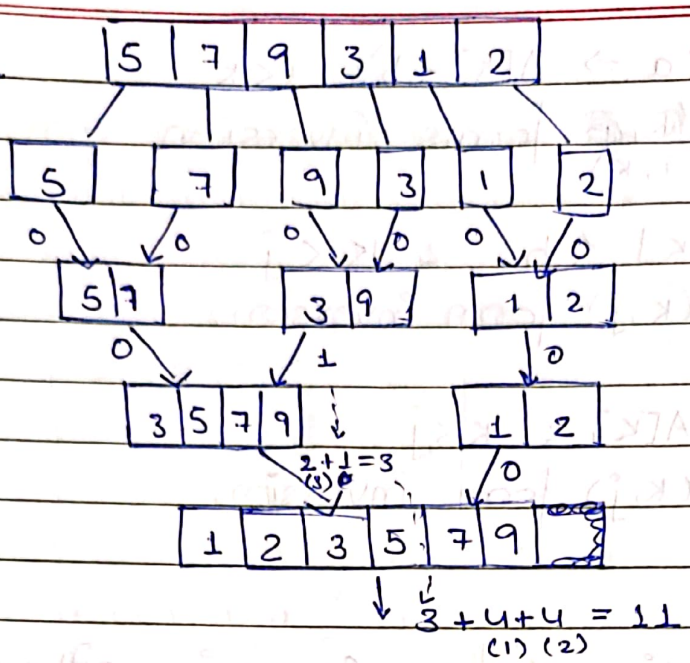
```
for i in 0...n-1
    for j in i+1...n-1
        if A[i] > A[j]
            inversion += 1
```

comparisions in worst case = $\frac{n(n-1)}{2}$

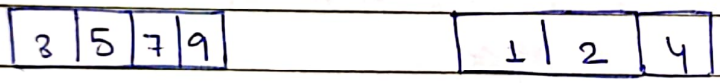
- Method 2:

Use the mergesort & merge procedures, with the assumption that at each level, when the array is divided into two halves, each half will return its inversion count, along with being sorted.

Ex.



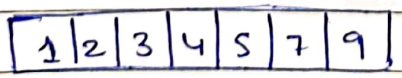
So, say we are merging the sorted arrays:



we ~~count~~ know the # inversions for each half individually. We want to count for the combination.

So, the # inversions for the combination is counted by checking how many places before a element of right array is put in left array.

1 is put 4 places before left arr
2 is put 4 places before in left arr.
4 is put 3 places before in left arr.



Inversions = left-invs +
right-invs +
combine-invs

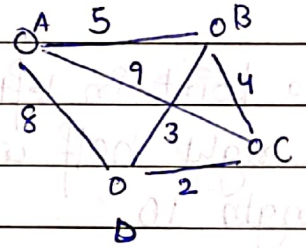
T.C. is still given by $T(n) = 2T(\frac{n}{2}) + \log(n)$

i.e. $T(n) = \Theta(n \log n)$

* Closest Pair in 2D

Given some 2D coordinates, find the pair of 2D coords distance b/w which is the min. of all

Ex.



Distance b/w C & D is minimum of all.

Method 1: Find dist. b/w each pair & compare.

$O(n^2)$.

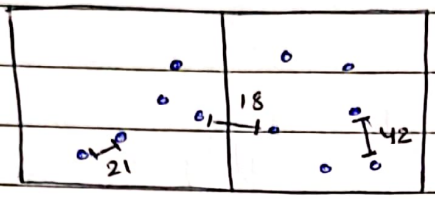
Method 2:

S1: Sort by x coord, store in one array.

S2: Sort by y coord & store in other array.

Say we are able to divide the points in two halves & each part returns its min. distance.

Now we have to see among the two halves which is the min. distance, & also compares points from each half with each other.



So, combination will return:

min (21, 42, 18)

i.e. min (δ_L, δ_R, δ_{LR})

Say, $\delta_L = 10$. So, for a point in left half we only consider points from right half which falls within the radius of length 10.

Within the radius there would only finite no. of points, not of order $\frac{n}{2}$.

So, $T(n) = n\log n \times 2 + S(n)$

$S(n) = 2T(\frac{n}{2}) + \Theta(n)$

$\therefore T(n) = \Theta(n\log n)$

* Exponent of a number

a^b

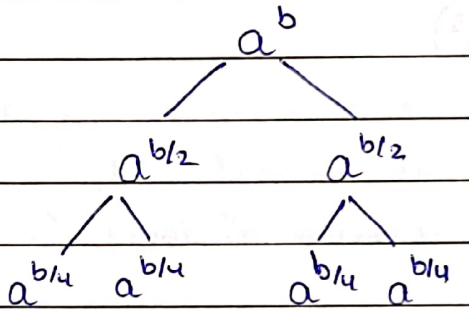
- Method 1

Multiply a, b times. $\Theta(b)$

- Method 2

If we know, $a^{b/2}$, we can get a^b as $a^{b/2} \times a^{b/2}$

So,



We see, at each stage we are calc. $a^{b/2}$ two times. So, we just calc. it one time & multiply.

```
exponent (a,b)
    if (b==0)
        return 1;
    if (b==1)
        return a;
    a_b_2 = exponent(a, b//2);
    if (b is even)
        return a_b_2 x a_b_2;
    else
        return a_b_2 x a_b_2 x a;
```

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

$\therefore T(n) = \Theta(\log n)$

* Matrix Multiplication

Method 1:

Simulate regular multiplication.

$\Theta(n^3)$

Method 2:

Divide $n \times n$ matrix into four $n/2 \times n/2$ matrices.

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} A_{11} \times B_{11} + A_{12} \times B_{21} & A_{11} \times B_{12} + A_{12} \times B_{22} \\ A_{21} \times B_{11} + A_{22} \times B_{21} & A_{21} \times B_{12} + A_{22} \times B_{22} \end{bmatrix}$$

- So, problem of size n, got reduced to eight n/2 problems.
- Combination cost is $\frac{n^2}{4}$ for each pair, i.e. $4 \times \frac{n^2}{4} = n^2$.

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

$a > b^k$

$8 > 2^2$

$\therefore T(n) = \Theta(n^3)$

* Quick Sort

At each step, pick some element, put all elements smaller to it in left subarray, all elems greater to its right subarray & pivot in-between.

```
- QS (A, l, h)
    if (l < h)
Division {  pvtIdx = partition(A,l,h);
Recurrence {  QS (A, l, pvtIdx-1);
              QS (A, pvtIdx+1, h);
```

```
- Partition (A, l, h)
  pvt = A[h]
  i = -1;
  for (j in l....h-1)
    if (A[j] ≤ pvt)
      i++;
      swap (A[i], A[j]);
  i++;
  swap (A[i], A[h]);
  return i;
```

Partition takes $\Theta(n)$.

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

$\therefore T(n) = \Theta(n \log n)$

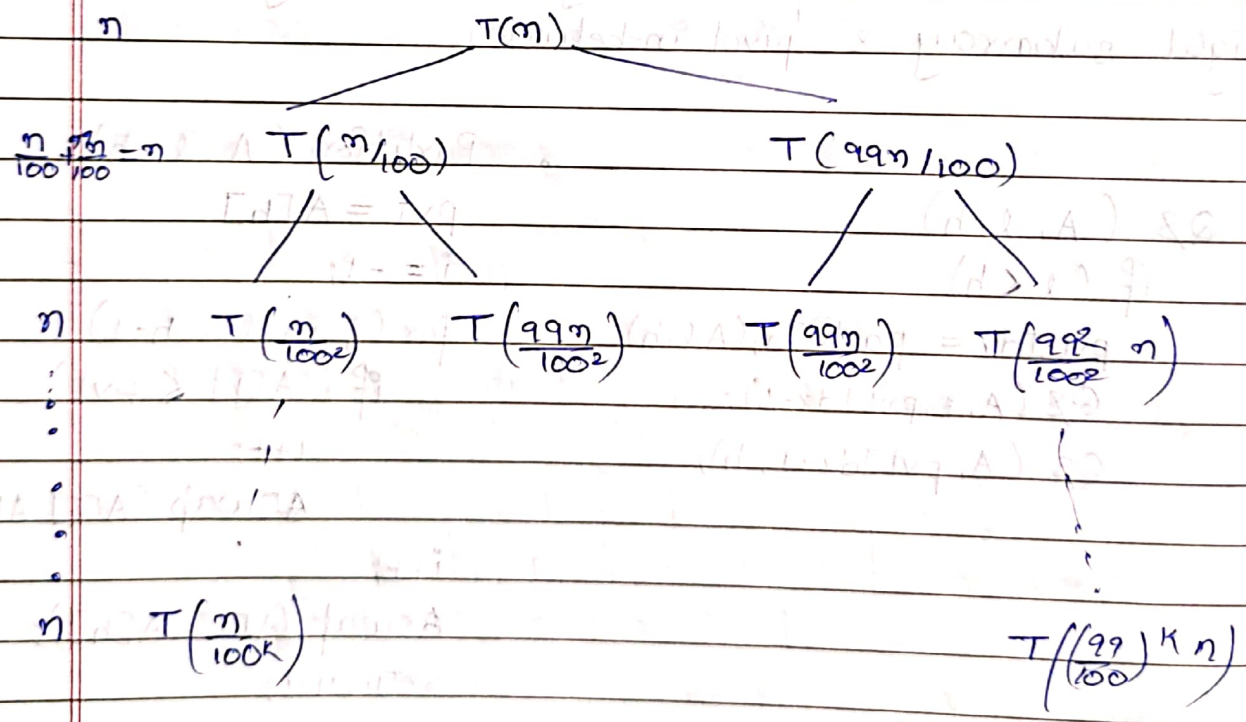
- QuickSort in general is not stable.
- QuickSort is ~~stable~~ inplace.
- Majority of work is done during division.

- Best Case

Each time, the array is divided into $n/2$ subarrays with pivot in middle.

$$T(n) = 2T(n/2) + \Theta(n)$$
$$\Rightarrow T(n) = \Theta(n \log n)$$

Ex. Suppose n is divided into $\frac{n}{100}$ & $\frac{99n}{100}$.



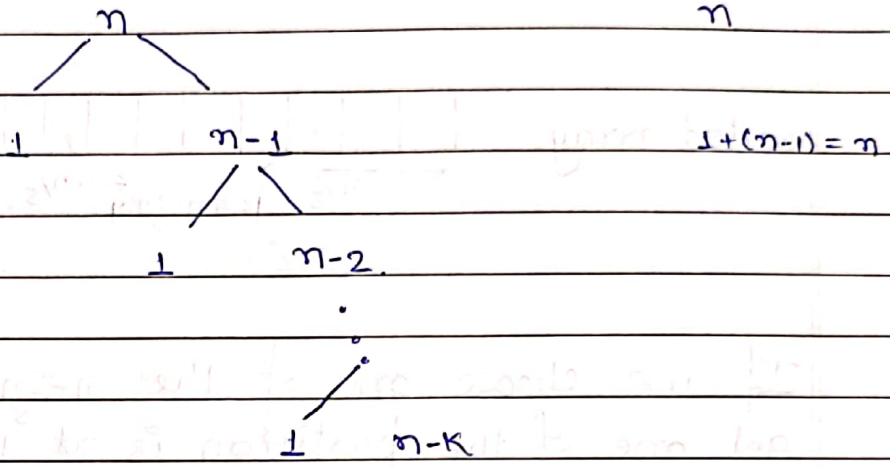
LB: $n = 100^k \Rightarrow k = \log_{100} n$

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

UB: $n = (\frac{100}{99})^k \rightarrow T(n) = n \log n$.

- Worst Case.

  One side of the partition has only a single element.



$n-K=0 \Rightarrow K=n$.

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

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

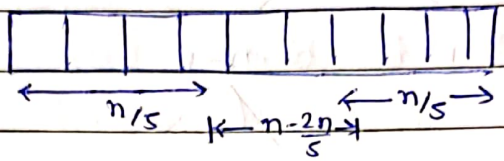
- Avg Case.

  On avg., Quick Sort takes $\Theta(n \log n)$

- If we pick median as pivot element, then even in worst case we get $\Theta(n \log n)$.

Ex. Given an array what is the probability that the Partition algo splits the array such that the smaller split is at least $n/5$ in length.

Sorted array:



If we choose one of the $n - \frac{2n}{5}$ no. we get one of the partition is at least $n/5$ in size.

$$\text{Prob} = \frac{n - \frac{2n}{5}}{n} \Rightarrow 1 - \frac{2}{5}$$

* Select Problem

Select (A, K) gives K^{th} smallest number, from A.

- Use pivot algo. to find index of a pivot.
  - if pivot > K, then K^{th} smallest lies in left subarray.
  - if pivot = K, return A[K].
  - if pivot < K, the K^{th} smallest lies in right subarray.

Ex. A = [1, 7, 5, 2, 8, 3, 4, 6] K = 4 (assuming K > 0)

0 1 2 3 4 5 6 7

S1: [1, 7, 5, 2, 8, 3, 4, 6]
pivot

0 < K.

0 1 2 3 4 5 6 7

S2: [1 | 5, 2, 3, 4, 6, 7, 8]
pivot

6 > K.

S3:

[1 | 2, 3, 4, 5, 6 | 7, 8]
piv

4 == K.

∴ Select (A, K) = 5.

- Best Case
  $T(n) = \Theta(1)$ {first pivot only is equal to K}

- Worst Case
  $T(n) = \Theta(n^2)$ { $T(n) = T(n-1) + \Theta(n)$ }

- Avg. Case.
  $T(n) = \Theta(n)$ { $T(n) = T(n/2) + \Theta(n)$ }

* Binary Search

```
BS (A, l, h, x)
  if (l ≤ h)
    md = (l+h)/2;
    if (A[md] == x) return md;
    else if (x < A[md]) BS(A, l, md-1, x);
    else BS(A, md+1, h, x);
  return -1;
```

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

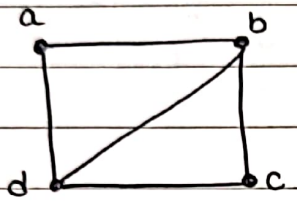
$\therefore T(n) = \Theta(\log n)$

GRAPHS

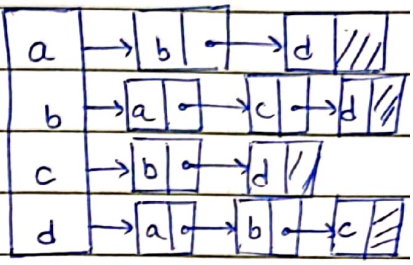
Graphs are generally stored in one of two ways

① Adjacency Matrix ② Adjacency List

Ex.



	a	b	c	d
a	0	1	0	1
b	1	0	1	1
c	0	1	0	1
d	1	1	1	0



Space taken by:

① AM: $\Theta(V^2)$ ② AL: $\Theta(V+E)$

	AM	AL
Test if $u \to v \in E$	$\Theta(1)$	$\Theta(\deg(u))$
List v's neighbour	$\Theta(V)$	$\Theta(\deg(v))$
List all edges	$\Theta(V^2)$	$\Theta(E+V)$
Insert edge $u \to v$	$\Theta(1)$	$\Theta(\deg(u))$
Delete edge $u \to v$	$\Theta(1)$	$\Theta(\deg(u))$

* Complexity of graph algo's depend on $|V|$ & $|E|$.
If graph is dense then $|E| \simeq |V|^2$.
If sparse graph then $|E| \simeq |V|$

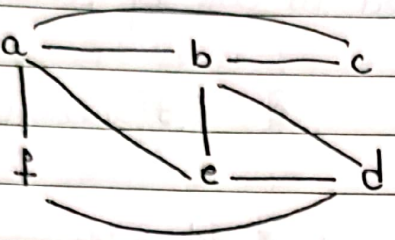
Say $O(V^{3/2})$ vs $O(E)$

Dense: $O(V^{3/2}) < O(V^2)$

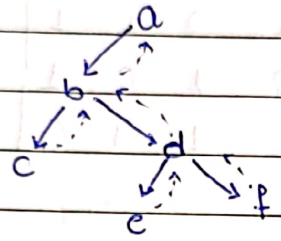
Sparse: $O(V^{3/2}) > O(V)$

* Depth First Search

Ex.

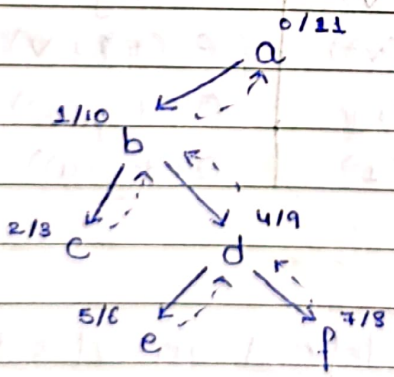


Src: a



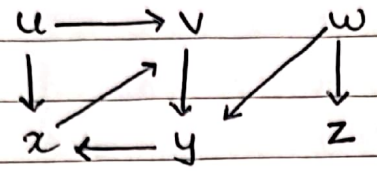
- Start Time: When we visit a vertex very first time.
  Finish Time: When we visit a vertex very last time.

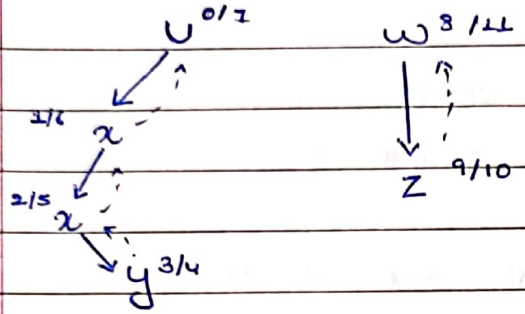
In above example: ST/ET



The start & finish time are useful for various applications of DFS.

Ex.





This is called Forrest

- Implementation

① Recursive

```
DFS (G, src, seen)
    Process (src)
    For (u,v) ∈ G.Edges where u = src
        If  seen[v] = false
            seen[v] = true
            DFS(G, v, seen)
```

Complexity: for each vertex we explore all there edges, i.e

$v_1 \Rightarrow 1 + \deg(v_1)$

$v_2 \Rightarrow 1 + \deg(v_2)$

$\vdots \qquad \vdots$

$v_n \Rightarrow 1 + \deg(v_n)$

$|V| + \{|E| \text{ or } 2|E|\}$

dir. undir.

$\Rightarrow \Theta(|V| + |E|)$ (Adjacency List)

$\Rightarrow \Theta(|V| + |V|^2) \Rightarrow \Theta(|V|^2)$ (AM)

(2) Iterative

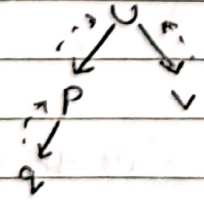
```
DFS (G, src)
    stack = [ ]
    stack.push(src)
    seen = [ ]
    seen[src] = true

    while !stack.empty()
        u = stack.pop()  ~~~> process(u)
        for v ∈ G.Edges[u]
            if seen[v] = false
                seen[v] = true
                stack.push(v)
```

- DFS Paranthses Theorem

The DFS algo maps similar to balance parantheses in terms of start & finish times.

Ex.



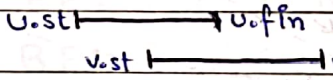
st u
st p
st q
fin q
fin p
st v
fin v
fin u

Theorem: In any DFS of a dir. or undir. graph G for any two vertices u & v, exactly one of the following condition holds:

(1) the intervals [u.start, u.finish] & [v.st, v.fin] are entirely disjoint & neither is descendent of other in DF forest.

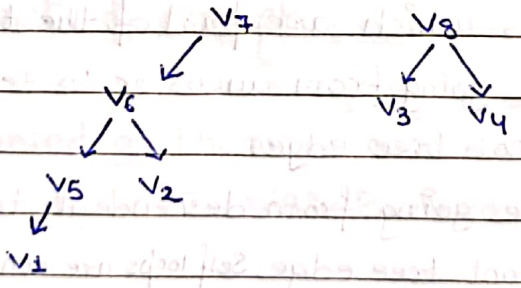
(2) the interval [u.st, u.fin] is entirely contained within [v.st, v.fin] & u is descendent of v.

(3) Vice-versa of (2), i.e. v is descendent of u.

(so intervals like 

are not possible)

Ex. V_1: [4,5] V_3: [12,13] V_5: [3,6] V_7: [1,10]
V_2: [7,8] V_4: [14,15] V_6: [2,9] V_8: [11,16]

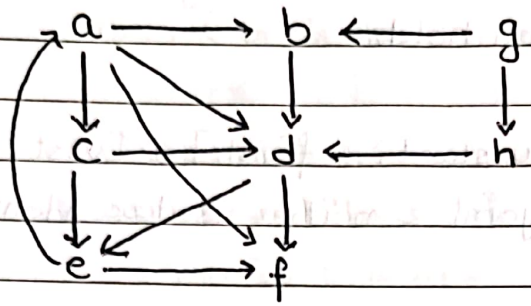


• Edge Classification

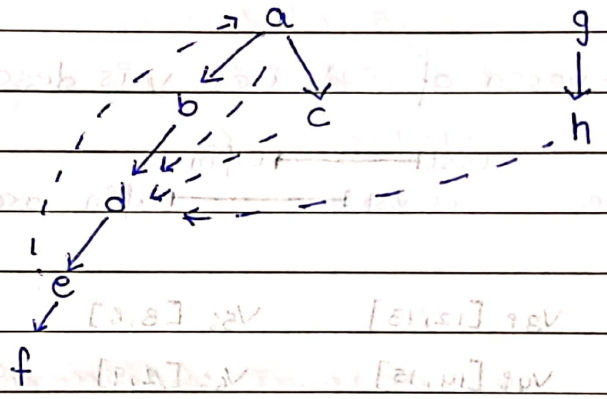
If DFS is run on a graph some edges are part of the DFS forest, some aren't. Based on that they are classified into certain categories.

① Directed Graph

Ex.



lets say src: a.



Tree Edges: Edges which are part of the tree.

Forward Edges: Edges going from ancestor to descendent & is not tree edge.

Backward Edge: Edges going from descendent to ancestor & is not tree edge. Self loops are considered back edges as well.

Cross Edge: Edges b/w two node where neither is descendent nor ancestor of other.

So in above example:

TE: (a,b), (b,d), (d,e), (e,f), (a,c), (g,h)

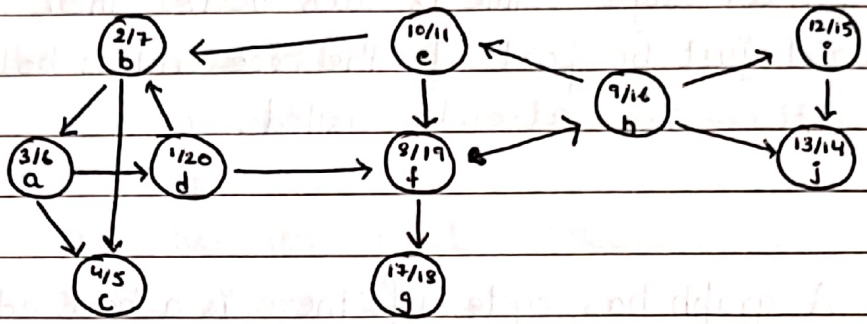
FE: (a,d), (a,f), (d,f)

BE: (e,a)

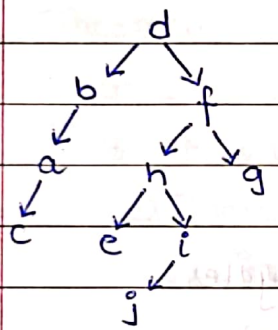
CE: (c,d), (c,e), (g,b), (h,d)

Note: Same graph can have a different traversal & hence different classification of edges.

Ex. St & fin times of nodes given. Classify edges.



(i, j)



TE: (a,c), (b,a), (d,b), (d,f), (f,h), (f,g), (h,e)(h,i)

FE: (b,c), (h,j)

BE: (a,d), (e,f)

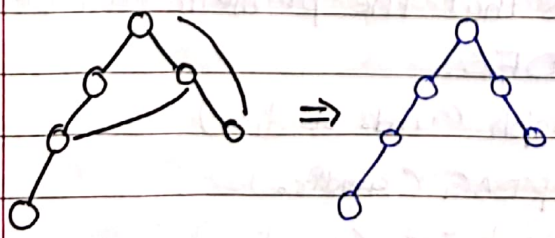
CE: (e,b)

② Undirected Graph

In Undirected graph there are only:

(1) Tree Edges (2) Back Edges

Ex. Let us suppose for the following graph we consider this Depth first forrest



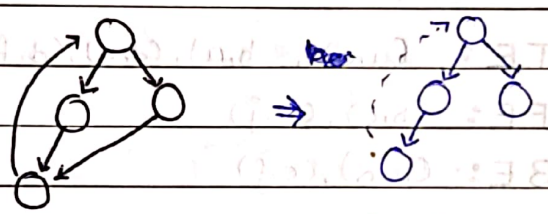
Since it is undirected, we can never have such a graph where an edge can be cross-edge since, if an edge is there we'd just take it as tree edge.

And since there is no direction, every edge which might be classified as forward edge, can also be classified as backward edge.

Crux: If an edge connects two vertex then it might just be part of the tree unless both the vertices are already visited.

Theorem: A graph has cycle iff there is a back edge.

Ex.



So we have one back edge & two cycles in graph.

For $n \geq 1$, if no. of back edges is n, then the no. of cycles in graph is at least n.

• Application of DFS

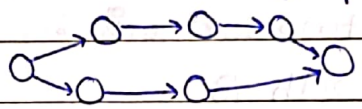
Application of DFS means that the problem can be solved in same T.C. as DFS.

(1) Detecting cycle in a graph (und. or dir.)
(2) Finding topological sort of DAG (dir.)
(3) Cut vertex or articulation point (undir.)
(4) Cut edges or bridges (undir.)

① Cycle Detection (Directed or Undirected)

- We know that, a graph has cycle iff its DFforest yeilds back edge. (for both dir. & undir.)

- In undirected graph for a node, if any of its neighbour (other than its parent) is already visited then there is a cycle.
  This fails for directed graph.



Theorem: Back Edge $\longleftrightarrow$ Cycle

Proof:

① BE → Cycle

① There is always a path from anscestor to descendent using tree edges.

② Since backedge, there is also path from descendent to anscestor.

Hence cycle

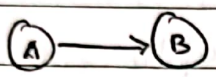
② Cycle → BE

Say while running DFS v_0 is first vertex from cycle that we visit. We know that all nodes in cycle have a path from v_0, hence every node in cycle must be started & finished b/w start & finish of v_0.
We know there is at least one other node v_k in cycle with edge to v_0, but since v_0 is already visited this edge won't be in tree.
Since v_0 is anscestor of v_k, $\{st(v_0) < st(v_k) < fin(v_k) < fin(v_0)\}$
this edge is backedge.

Hence backedge

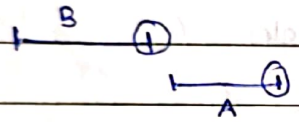
② Topological Sort of DAG

- In a DAG, for any edge $u \to v$, we always have $fin(u) > fin(v)$.
- Sorting order in in reverse of finish values.
- In DAG there is at-least one vertex with no incoming edge.

Ex. 

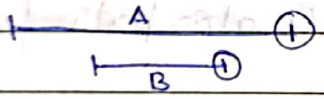
There can be two scenario :

(i) We start with B



$\therefore fin(A) > fin(B)$

(ii) Start with A



$\therefore fin(A) > fin(B)$

$\therefore \forall (u,v) \in E$ in a DAG iff $fin(u) > fin(v)$

- There can be more than one topological order in a DAG.
- Topological sort of a graph exists iff it is a DAG.

③ Articulation Point (Undirected)

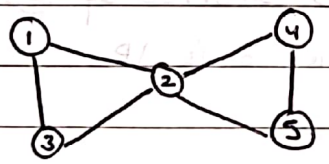
An articulation point of a graph is a vertex which when removed from graph, disconnects it into two or more components.

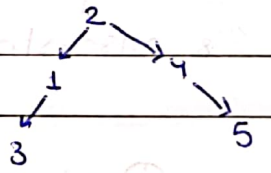
- Brute Force Approach: For every vertex, remove it & check by DFS if singe component or not.

$$O(V.(V+E))$$
$$\Rightarrow O(V^2+VE)$$

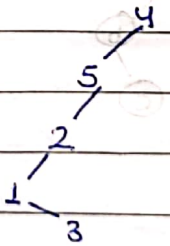
- Root of DFTree is an articulation point iff there are two or more children of it (since each child subtree won't have any edge to the others because no cross edge in undirected graph)

Ex.



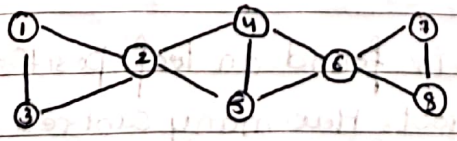


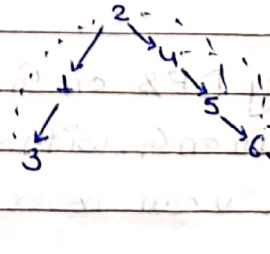
∴ 2 is AP.



∴ 4 is not AP

- A non-root node in DFTree is AP iff there exist no edge b/w any of its descendent & any of its anscestor.

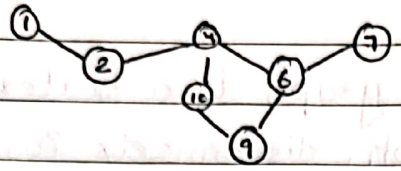


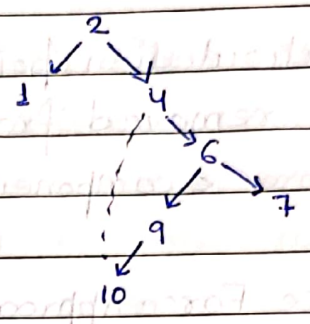


① 2 root with 2 children nodes. ∴ 2 is AP

② 6 has no des. edge with ans. ∴ 6 is AP

There can be cases where this def. fails.

Ex. 

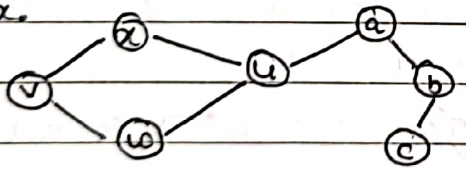


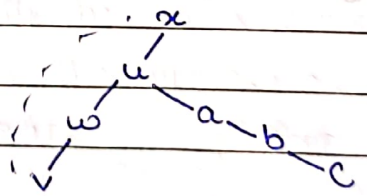
Here 6 clearly is AP, but still has a des-ans. edge b/w 4 & 10.

∴ Defn fails.

A non-root is AP iff it has at least one subtree in which descendents aren't connected by an edge to any of its anscestor.

∴ In above example the (7) subtree of (6) has no edge to 6's anscestor, hence 6 is AP.

Ex. 



∴ u, a & b are all APs.

- A leaf node in DF Tree can never be AP.

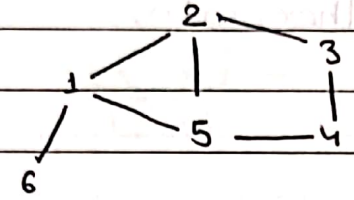
Ex. On running DFS on G, v is found on leaf position. Run DFS again on G, with v as root. How many subtree of v?

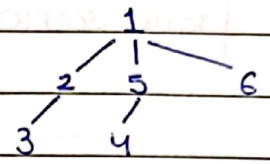
1. ∵ v can never be AP.

* Breadth First Search

For every node explore all of its neighbours first.

Ex.

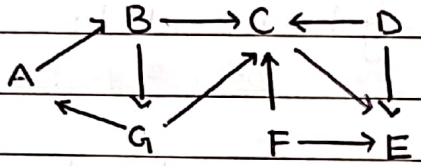




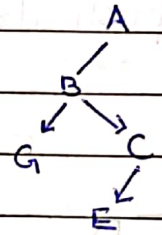
BFS Tree.

Src: 1

Ex.



Src: A



D F

BFS Forrest

(DFS trees are long & skinny, BFS trees are short & fat)

• Implementation

```
BFS (G, src)
    queue = [ ]
    visited = [ ]    { or seen }
    queue.enque(src)
    visited [src] = true

    while !queue.empty()
        node = queue.deque()
        Process(node)
        for (u,v) ∈ G.Edges(node)
            if visited [v] == false
                visited [v] = true
                queue.enque(v)
```

Complexity:

for every vertex explore all its neighbours.

$v_0 \Rightarrow \Theta(1 + \deg(v_0))$

$v_1 \Rightarrow \Theta(1 + \deg(v_1))$

:

$v_n \Rightarrow \Theta(1 + \deg(v_n))$

$\Rightarrow O(V + E)$ (AL)

$\Rightarrow O(V^2 + V^2)$ (AM)

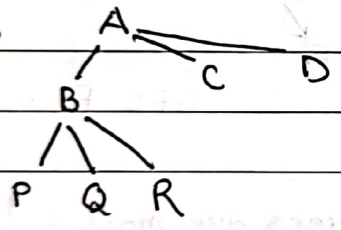
- BFS for shortest path in unweighted graph

In unweighted graph (or graph where every edge has same weight), BFS gives the shortest path from source to all other nodes.

- BFS Queue State.

For every node at the front of queue, we pop it & then push its neighbours at the rear of the queue.

Ex.

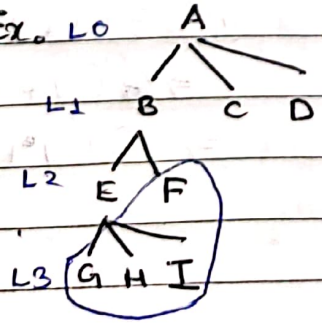


1:	2:	3:
A	~~A~~	~~A~~
	B	~~B~~
	C	C
	D	D
		P
		Q
		R

At any point in time only the nodes currently acting as leaves will be present in the queue, i.e. internal nodes can never be in queue at any time.
These currently acting leaf nodes are called Frontiers.
It's not necessary that all current frontier nodes are in queue.

Next levels of nodes are explored only when all prev. level nodes are already explored.

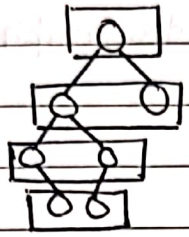
Ex.



C, D must be not in queue, otherwise L3 nodes won't have been explored.
∴ Only potential nodes in queue are:
F, G, H, I.

- BFS as layers

Consider each level of BFS as layer.



① Is it possible that there is an edge b/w two non-adjacent layers in undirected graph? (Skip-level edges)

No. If edge was there, then it would have been explored already. {Fwd Edge}

② Is it possible in directed graph?

No. Same reason. {Fwd Edge}

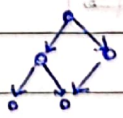
Yes. {Back Edge}

③ Can you have edge ~~b/w~~ within same level (dir, undir)?

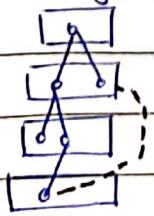
Yes.

Ex.  {Cross Edge}

④ Can you have such edges

Yes. Ex.  {Cross Edge}

⑤ Can you have such edges (undir.)



No. Its Skip-level edge. Hence not possible.

- Cross edges on same level always make an odd length cycle, in undir. graph
- Cross edges across levels in undir. graph always makes even length cycle.

BFS Edges

Directed	Undirected
① Tree edge	① Tree edge
② Back edge	② Cross Edge
③ Cross Edge	2.1. Across same level
3.1. Same Level	2.2. Across two consecutive levels.
3.2. Across two consecutive levels	

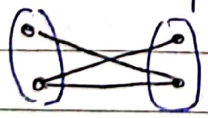
- Application

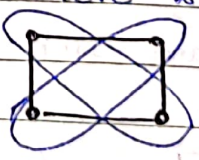
Can be solved in same TC as BFS.

① Bipartite Graph (Undir)

- A graph is Bipartite iff there is no odd length cycles.

Ex.





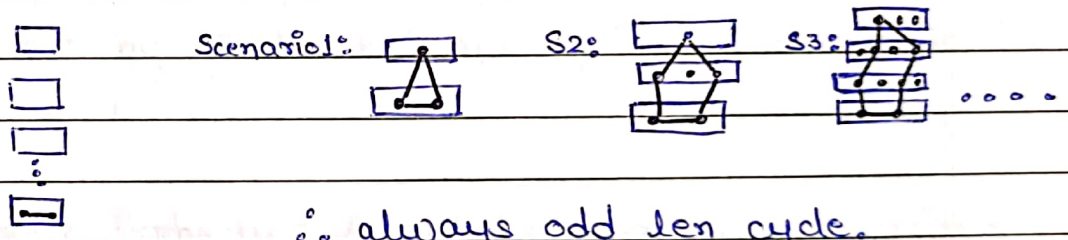
- A tree is always bipartite, since every level has no edge b/w itself.

- In undirected graph:
  ① No cross edge → bipartite
  {∵ No CE means graph is tree}

  ② If there is cross-edge b/w across same level, then odd length cycle.
  ∴ Not bipartite.

  ③ If cross-edge b/w two consecutive level then even length cycle.

Proof: ● Across Same level.



∴ always odd len cycle.

∴ ①Run BFS
② Bipartite iff
① No Cross edges or
② Only cross edges accross two consecutive levels
i.e. even length cycles.

② Number of Connected Components (Undir.)
Can be checked using both BFS or DFS.
- Run BFS covering all nodes. The no. of trees in the BF forrest is the no. of components.

GREEDY ALGOS.

A Greedy approach to a problem is characterized by making the greediest choice at the current moment in time.

A greedy approach is not always guarenteed to give the most optimal solution, but it gives a feasible solution always.

* Property of Greedy Algorithms

If a problem can be characterized to have the following properties then a Greedy algorithm can be devised for it:

(1) Optimal Substructure : Optimal solution can be achieved for a problem by first acheiving optimal solution to its subproblem.

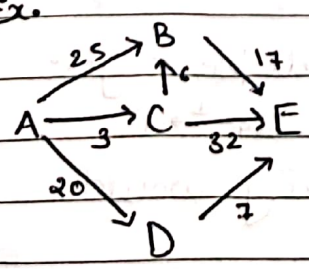
(2) Greedy Choice Property : A global optimal can be arrived at by selecting local optimal.

* Single Source Shortest Path

Given a graph G and a source u, the task is to find the shortest possible path from the source u to every other node in G.

Approach 1: Find all possible paths to a node & choose shortest.

Ex.



Say A→E

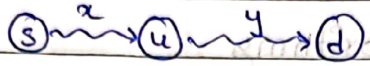
A→B→E : 42 A→C→E : 35 A→C→B→E : 26

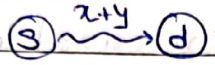
A→D→E : 27.

∴ A→C→B→E

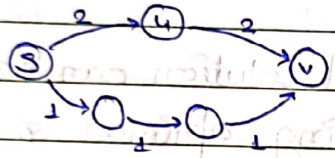
This is Brute Force approach & could take exponential time.

Approach 2: Find two shortest path & join.



$\therefore$ 

Ex.



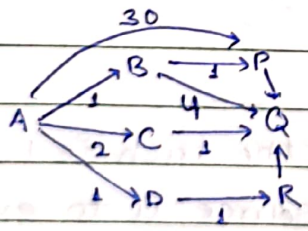
$s \rightsquigarrow u : 2 \quad u \rightsquigarrow v : 2$

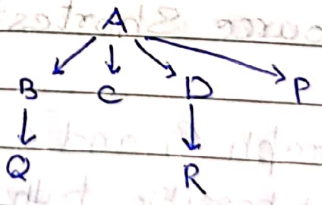
$\Rightarrow s \rightsquigarrow v : 4$

This approach doesn't work.

Approach 3: Using BFS

Ex.





If the graph would be unweighted, this would work.

We can convert a n-weight edge into multiple edges:

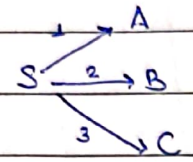
$A \xrightarrow{30} P \quad \Rightarrow \quad A \xrightarrow{1} x_1 \xrightarrow{1} x_2 \longrightarrow \ldots \ldots x_{29} \xrightarrow{1} P$

BFS would work right by the complexity would depend on edge weights.

Lemma: On the shortest path p from u to v $u \rightsquigarrow v$, any subpath of p, say from x to y ($u \rightsquigarrow x \rightsquigarrow y \rightsquigarrow v$) is also the shortest path from x to y.

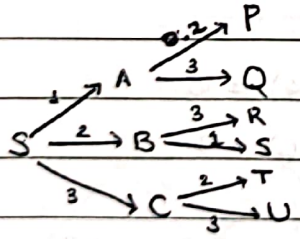
Approach 4: (Considering only +ve weights)

Ex. Say we have partial graph with all outgoing edges of S:



- Can you go S→B in less than 2?
  Maybe. There can be a path $S \xrightarrow{1} A \overset{0.5}{\rightsquigarrow} B$
- Can you go S→C in less than 3?
  Maybe. There can be paths like: $S \xrightarrow{1} A \overset{1}{\rightsquigarrow} C$
  $S \xrightarrow{2} B \overset{0.5}{\rightsquigarrow} C$
- Can yo go S→A in less than 1?
  No. Any other path would have to be of type $S \xrightarrow{2} B \overset{\alpha}{\rightsquigarrow} A$ or $S \xrightarrow{3} C \overset{\beta}{\rightsquigarrow} A$
  $2 + \alpha > 1$ & $3 + \beta > 1$
  ∴ S→A is shortest.

We can repeatedly explore all the nodes in this way.



Starting at S

S1: Known dist: {1, 2, 3}

Winner: 1 i.e. S→A is shortest possible.

S2: Known dist: {2, 3, ~~1.2~~, ~~4~~ B}
S→B S→C ~~S→P~~ ~~S→Q~~

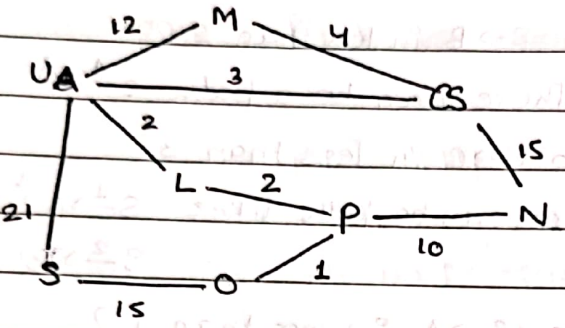
Winner: ~~1.2~~ S→A→P

∴ S→P is shortest possible.

So at every step we explore outgoing edges of finalled node, such as at Step1 S was already final so we explored its edges, at S2 A was final so we explored its edges along with other ~~a~~ edges of S.

This esentially is Dijkstra's Algorithm.

Ex.



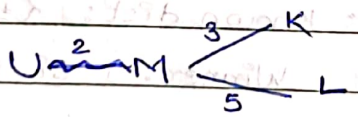
Source : U

Initialize a Priority Queue with dist to all nodes to ∞, except source.

At each step choose the min. dist node from queue. This node is finalled. Also relax all the vertices the finalled node is outgoing toward.

Relaxing means to update dist of a node from source if lesser is found through finalled node.

Say source : U finalled M.



$w(K) = 10$

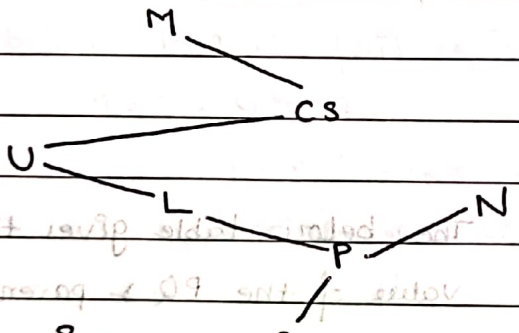
$w(L) = 3$

$$\text{weight}(K) = \min[\text{weight}(K), w(M) + M \to K]$$
$$\Rightarrow \min[10, 5]$$
$$\Rightarrow 5$$

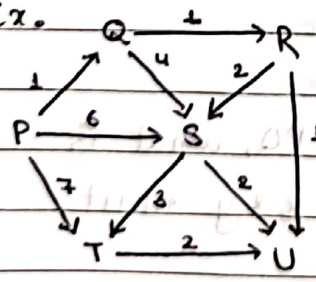
$$w(L) = \min[w(L), w(M) + M \to L]$$
$$\Rightarrow \min[3, 2+5]$$
$$\Rightarrow 3.$$

Finalled	U	M	CS	L	P	N	S	O
—	0	∞	∞	∞	∞	∞	∞	∞
U (0)	-	12	3	2	∞	∞	21	∞
L (2)	-	12	3	-	4	∞	21	∞
CS (3)	-	7	-	-	4	18	21	∞
P (4)	-	7	-	-	-	14	21	5.
O (5)	-	7	-	-	-	14	20	-
M (7)	-	-	-	-	-	14	20	-
N (14)	-	-	-	-	-	-	20	-
S (20)	-	-	-	-	-	-	-	-

∴



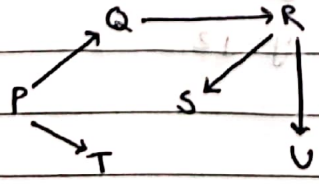
Ex.



Source: P

	P	Q	R	S	T	U
-	0	∞	∞	∞	∞	∞
P (0)	-	1	∞	6	7	∞
Q (1)	-	-	2	5	7	∞
R (2)	-	-	-	4	7	3
U (3)	-	-	-	4	7	-
S (4)	-	-	-	-	7	-
T (7)	-	-	-	-	-	-

∴



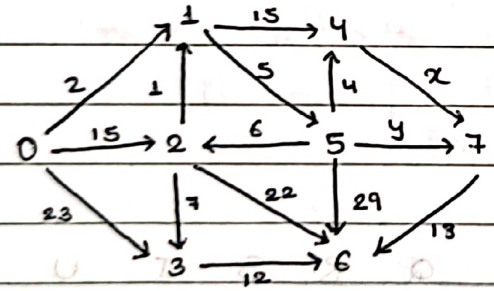
Implementation.

```
DIJKSTRA (G, src)
    seen = [ ]
    pq = [ ]
    initialise pq to ∞ for every node.
    pq[src] = 0

    while !pq.empty()
        curr = pq.dequeue()
        seen[curr] = true
        for v ∈ G.Edges(curr)
            if seen[v] == false
                RELAX (curr, v, G)
```

```
RELAX (u, v, G)
    if G.w(u, v) + d[u]
    < d[v]
        d[v] = G.w(u,v)
               + d[u]
```

Ex.



Source: 0

V	PQ	Parent
0	0	null
1	2	0
2	13	5
3	23	0
4	11	5
5	7	1
6	36	5
7	19	4

The below table gives the value of the PQ & parent values immediately after vertex 4 has been deleted from PQ & relaxed.

Based on the PQ, what is the condition x & y must satisfy:

(A) $x = 8$ & $y \geq 12$

(B) $x > 8$ & $y = 11$

(C) $x = 7$ & $y = 11$

(D) $x > 8$ & $y = 12$.

Tracing steps from start uptill Vertex 4 is popped & relaxed.

		0	1	2	3	4	5	6	7
S1	–	0	∞	∞	∞	∞	∞	∞	∞
S2	0 (0)	–	2	15	23	∞	∞	∞	∞
S3	1 (2)	–	–	15	23	17	7	∞	∞
S4	5 (7)	–	–	13	23	11	–	36	7+y
S5	4 (11)	–	–	13	23	–	–	36	min(11+x, 7+y)

Since 4 was selected over 7 in S5,

$11 < 7+y$

& $\min(11+x, 7+y) = 19$.

& Parent (7) = 4

Option B & C fail upfront.

A: $x = 8$ & $y \geq 12$.

$11 < (7+12)$ ✓

& $\min(11+8, \geq(7+12))$ ✓

& Parent (7) = 4 ✓

D: $x > 8$ & $y = 12$

$11 < 19$ ✓

& $\min(\geq(19), 19)$ ✓

& Parent (7) = 4 ✗

∴ A.

• Dijkstra on -ve Edges

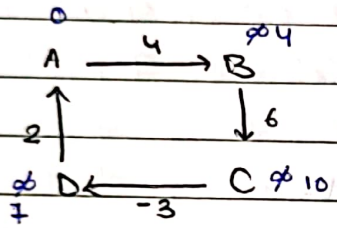
A graph has -ve edges if any of its edge weight is negative.

Ex. A $\xrightarrow{20}$ B $\xrightarrow{-10}$ C $\xrightarrow{3}$ D

20 10 13

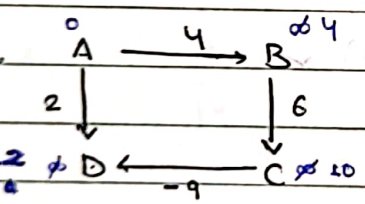
⇒ Dijkstra works here.

Ex:



So Dijkstra works again.

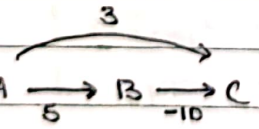
Ex.



Since D was shortest during S2 it was removed from consideration with dist. 2, but we can see path A→B→C→D with dist. 1.

So, Dijkstra didn't work here.

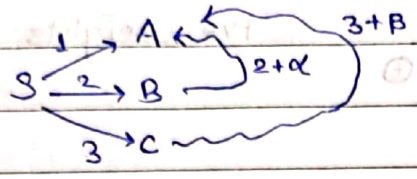
Ex.



	A	B	C
–	0	∞	∞
A(0)	–	5	3
C(3)	–	5	–
B(5)	–	–	–

Again A→C we get 3, but path A→B→C gives -5.
Dijkstra fails.

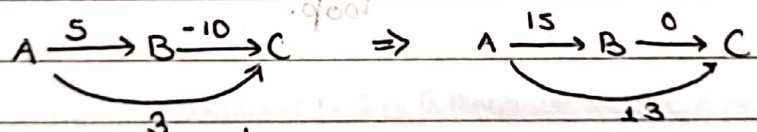
The assumption we make each time we finalize a node is that the path through any other node would only add to current dist. of finalized node



But with -ve weights the other paths can maybe decrease from current dist. to the finalized node.

∴ Dijkstra doesn't work on graphs with -ve weight edges.

We can add some common weight to all edges to make -ve weights go away.

Ex. 

	A	B	C
	0	∞	∞
A(0)	0	15	13
C(13)	0	15	13
B(15)	0	15	13

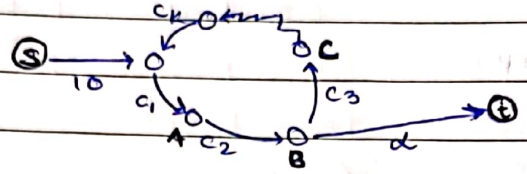
This changed shortest path from A→B→C to A→C.

∴ This approach of tackling -ve edges doesn't work.

There's no strategy to make Dijkstra work on -ve edges.

- Can a shortest path contain cycle?
  (pos or neg)

Ex.



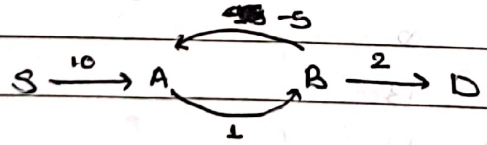
+ve weights

If from source a node say c is reachable in dist. $10 + c_1 + c_2 + c_3$, why would you loop around again to get to C?

Or to get to t you can get $10 + c_1 + c_2 + \alpha$, why would you go in loop?

$$10 + c_1 + c_2 + \underbrace{(c_3 + \dots c_K + c_1 + c_2)}_{\text{loop.}} n + \alpha$$

Ex. -ve weight



$S \to A \to B$: $10 + 1 = 11$

$S \to A \to B \to A \to B$: $10 + 1 - 5 + 1 = 7$

:

There is no end to it. It would keep reducing.

∴ There can't be a cycle in shortest path w/ neither +ve nor -ve weights.

With -ve cycle, we at least expect the Shortest Path algorithm to detect the cycle.

Dijkstra doesn't even care about -ve weights, let alone detecting -ve cycle.
It is not guarentted to give correct answer in case of -ve weights.

- Time Complexity

In the loop, for each vertex, we extract the min-dist vertex & relax for all its edges.
Overall for every edge in graph we relax exactly one time.
For every vertex, extract-min runs exactly once.

$$TC = |V| \times TC(\text{Extract-Min}) + |E| \times TC(\text{Relax})$$

$$\Rightarrow \sum_{u \in V} \left(TC(\text{Extract-Min}) + \deg(u) \times TC(\text{Relax}) \right)$$

$$\Rightarrow V \times TC(\text{Extract-Min}) + \sum_{u \in V} \deg(u) \times TC(\text{Relax})$$

① Adjacency List + Heap

$$\Rightarrow V \times \log(V) + TC(\text{Relax}) \times E$$
$$\Rightarrow V \log V + E \log V$$
$$\Rightarrow (V+E) \log V$$

② AM + Heap.

for every vertex v : [: ; 1 0 0 1 0 0 1 . . . ; : :]

if entry is 1, we relax,
if entry is 0, we ignore

$$\Rightarrow V \times \log V + \sum_{u \in V} \left(\underbrace{\deg(u)}_{\text{1 entry}} \times TC(Relax) + \underbrace{(V - \deg(u))}_{\text{0 entry}} \times TC(1) \right)$$

$$\Rightarrow V \times \log V + E \times TC(Relax) + (V^2 - E)$$

$$\Rightarrow V \log V + \underbrace{V^2}_{\text{This is extra for AM}} + E[TC(Relax) - 1]$$

$$\Rightarrow (V+E) \log V + V^2$$

∴ AL: $V \times TC(Extract\,min) + E \times TC(Relax)$

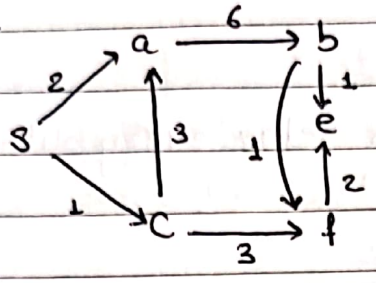
AM: $V \times TC(Extract\,min) + E \times TC(Relax) + V^2$

Graph Repr.	PQ Data Structure	Dijkstra T.C.
AL	Heap	$V \log V + E \log V$
AM	Heap	$(V+E) \log V + V^2$
AL	Unsorted Arr.	$V^2 + E = O(V^2)$
AM	Unsorted Array	$V^2 + E + V^2 = V^2 + E = O(V^2)$
AL	Sorted Array	$V + EV = O(EV)$
AM	Sorted Array.	$V + EV + V^2 = O(V^2 + EV)$
AL	Fibonacci Heap	$V \log V + E$
AM	Fibonacci Heap	$V \log V + E + V^2$

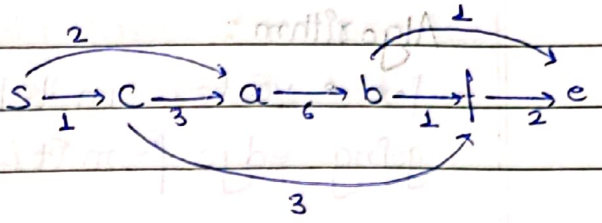
∴ Dijkstra's TC = $O(V \log V + E)$

• Shortest Path in DAG

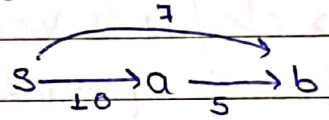
Ex.



⇒



If we relax all the vertices in topological order starting from given source, everytime we go to the next node we can be sure that there are no other incoming edge to that node.



Source: s

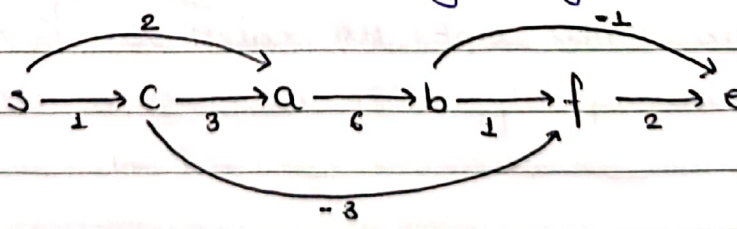
S1: s=0 a=10 b=7.

S2: Since we are at node 'a', we are sure that no other edge is incoming to 'a', hence 10 is min dist.

s=0 a=10 b = min(7, 10+5) = 7.

S3: Since we are at 'b' we can be sure that there's no other incoming edge to it.

Ex.



Source : c

	s	c	a	b	f	e
c (0)	∞	-	3	∞	-3	∞
a (3)	∞	-	-	9	-3	∞
b (9)	∞	-	-	-	-3	8
f (-3)	∞	-	-	-	-	-1
e (-1)	∞	-	-	-	-	-

This algo doesn't care about -ve edges. Works wells with both +ve & -ve weight.

Algorithm:

for • vertex in topological order relax every outgoing edge from it.

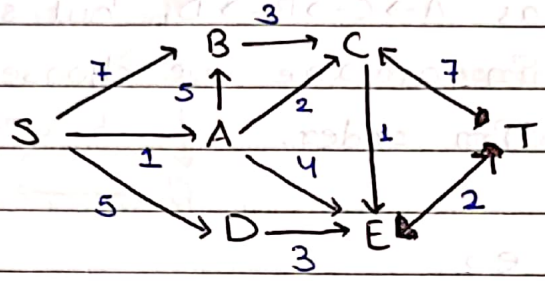
TC (Topological Order) + TC (Relax every edge)

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

$\Rightarrow O(V+E)$

* Bellman - Ford

Ex. Suppose that we already know the shortest path b/w S & T, S→A→C→E→T, but we don't know the path cost.



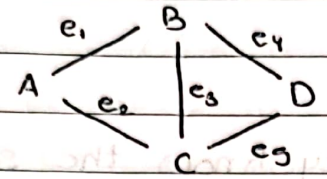
Now to know the path cost, we can just traverse from S→T through the path & relax the edges on the path in order : S→A, A→C, C→E, E→T.

We can maintain this relaxation order while also relaxing other edges.

S→B, S→D, S→A, A→B, A→E, A→C, D→E, B→C, C→E, E→T, T→C

So, even if we relax all edges while still maintaining path order, we still get right path cost.

Ex.



Source: A

Suppose we ~~know~~ can observe the shortest path from A to D as A→C→B→D, but since we don't know this in advance we choose some random relaxation order.

e_4 e_5 e_1 e_3 e_2

To get the shortest path from A→D we must relax the following edges in order.

e_2, e_3, e_4.

But due to random order that's not happening. Lets relax twice.

e_4, e_5, e_1, e_3, e_2 | e_4, e_5, e_1, e_3, e_2.

Lets do it thrice:

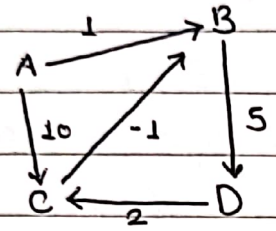
e_4, e_5, e_1, e_3, e_2 | e_4, e_5, e_1, e_3, e_2 | e_4, e_5, e_1, e_3, e_2

we get the order we require. whatever is relaxed in b/w doesn't matter as long as the order is maintained.

Bellman-Ford says that if we relax all the edges (in any order) V-1 times, then the shortest path order to all the vertices is guarenteed to appear.

B-F works with -ve edges as well.

Ex.

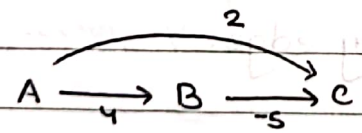


Source : A

Say we choose order : A→C, A→B, D→C, C→B & B→D

	A	B	C	D
—	0	∞	∞	∞
S1	0	1	10	6
S2	0	1	~~10~~ 8	6
S3	0	1	8	6

Ex.



(Dijkstra will fail here)

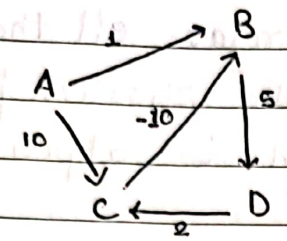
Source : A

Order : B→C, A→C, A→B. V-1 = 3-1 = 2.

	A	B	C.
Initial	0	∞	∞
1	0	4	~~-2~~ 2
2	0	4	-1

∴ Bellman-ford works here.

Ex.



Order: C→B, B→D, D→C, A→B, A→C

V-1 = 4-1 = 3.

Source: A.

	A	B	C	D.
Initial	0	∞	∞	∞
1	0	~~∞~~ 1	~~∞~~ 10	∞
2	0	0	7	5
3	0	-3	4	2
Extra	0	-6	1	-1

Bellman-Ford runs an extra cycle to check for -ve cycle. If any value changes in the extra cycle, that means there is -ve cycle.

Algorithm:

BF

for V-1 times relax every edge.

for 1 time relax every edge

if any dist value changes, then -ve cycle.

If while running BF, say from i^{th} to $(i+1)^{th}$ step nothing changes, then afterwards nothing will change, so we can stop after $(i+1)^{th}$ step only.

$$1 \leq i < V-1$$

Time Complexity:

$$(V-1) \times TC(Relax) + 1 \times E \times TC(Relax)$$
$$\Rightarrow VE\ TC(Relax)$$
$$\Rightarrow O(VE)$$

Comparision with Dijkstra

① Complexity.

	Dj		BF
	$(V+E)\log V$		VE
Dense:	$(V+V^2)\log V$		$V \times V^2$
	$\Rightarrow V^2 \log V$	$<$	$\Rightarrow V^3$
Sparse:	$(V+V)\log V$		$V \times V$
	$\Rightarrow V \log V$	$<$	$\Rightarrow V^2$

$\therefore$ Complexity of Dijkstra is better.

② -ve Edge & -ve Cycle.

Dj may or maynot give correct result with -ve edge & cycles. It can't detect -ve cycles either.

Bellman-Ford after K times relaxation gives the shortest dist. to all the vertices which have at most K-length shortest path.

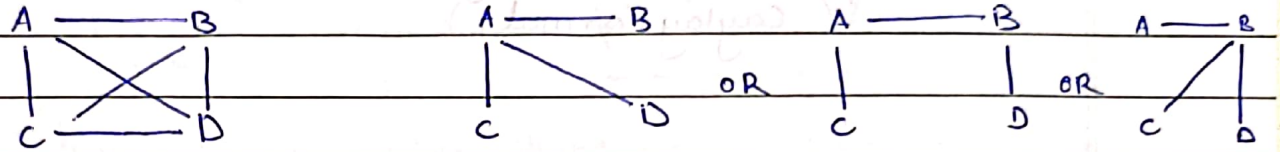
So if shortest path from A ↠ T is 3 len. say, A → L → M → T, then after 3 cycles of relaxation we'll get final cost of A ↠ T

Although the algo. doesn't know it since the path & hence path lengths are unknown.

* Minimum Spanning Tree

Spanning Tree (defined for undirected graph) is a connected subgraph which contains all the vertices & has no cycle.

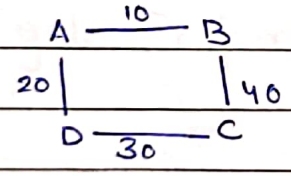
Ex.



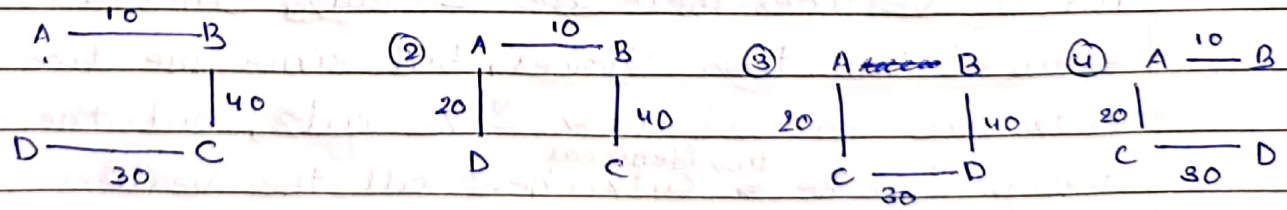
So, for this graph we have ${}^{6}C_{3}$ STs.

When there is weight associated w/ edges then the Spanning Tree which gives the minimum cost, is called the Minimum Spanning Tree.

Ex.



① ② ③ ④



So, 4 is the Minimum Spanning Tree for the graph.

- **Approach 1:**

Find out all spanning trees & find the minimum.
Not feasible.
No. of possible spanning trees = n^{n-2}
(Caylay Formula)

- **Cut Property:**

A cut in a graph is a partition of its vertices into two (nonempty) sets.

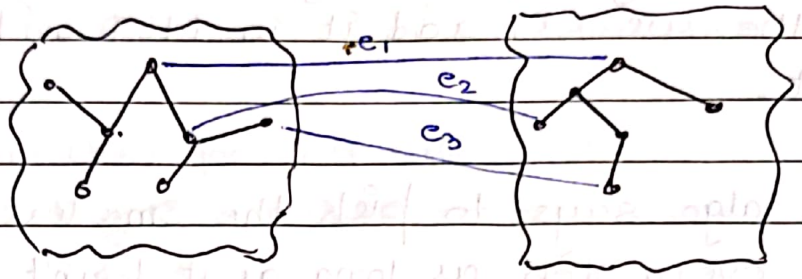
Ex. $V = \{a, b, c, d, e\}$
$\{\{a, b\}, \{c, d, e\}\}$
$\{\{a, e\}, \{b, c, d\}\}$
$\{\{b, c\}, \{a, d, e\}\}$
⋮

For n vertices there are 2^n cuts, since each element has two choices, but since the two cuts are identical, $\Rightarrow 2^n/2$ cuts, but there would also be two identical cuts where all the vertices are on one side & other is empty.

$$\therefore \quad \frac{2^n}{2} - 2 = 2^{n-1} - 1$$

Cut Property says, the smallest cost edge crossing a cut must be part of all possible MSTs of the graph.

Ex.



Say for this cut we've three edges crossing the cut then $\min(\{e_1, e_2, e_3\})$ is part of all possible MSTs of the graph.

If there is more than one edge crossing a cut with their cost equal to min. cost, then those edge would be part of some MST, not all.

- Cycle Property

The largest cost edge in a cycle can never be part of any MST of a graph.

If there is a cycle such that there are more than one edge with largest cost then at least one of them will not be in MST.

An edge e is not in any MST iff it is the unique heaviest edge in some cycle.

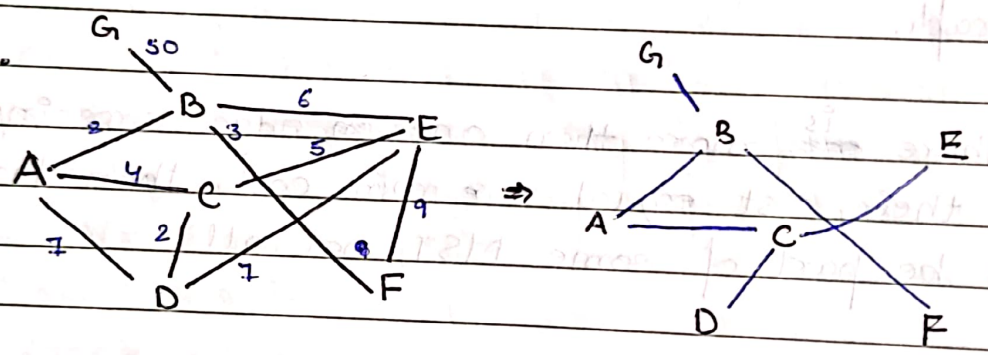
- If an edge is a bridge, it will always be in MST.

- Kruskal Algorithm

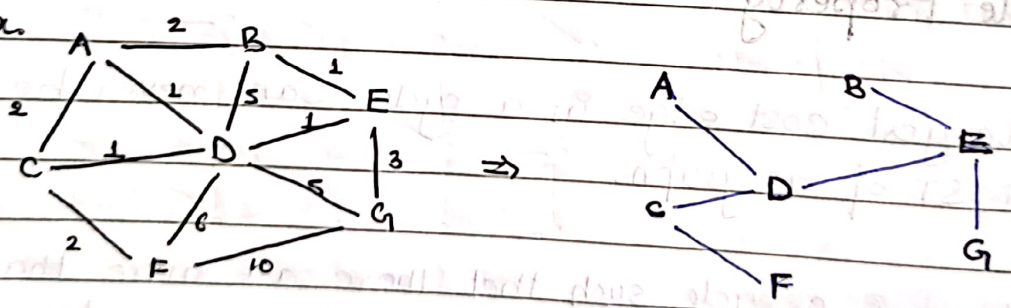
For every edge if the edge must be in the MST, the we can add it in MST without a doubt.

Kruskal algo. says to pick the smallest edge at every step, as long as it doesn't form a cycle.

Ex.



Ex.



Kruskal (G)

T = { }

Sort edges by weight in ascending order

for each edge e_i

if $\{e_i\} \cup T$ doesn't form cycle

$T = T \cup \{e_i\}$

TC(Kruskal) = Sorting edges + For each edge check cycle

$\Rightarrow$ $E \lg E + E \times O(\text{check cycle})$

Say we check for cycle using DFS

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

$\Rightarrow O(E^2)$

If we use Union-Find data structure, then we can ~~check~~ find cycle in $O(1)$

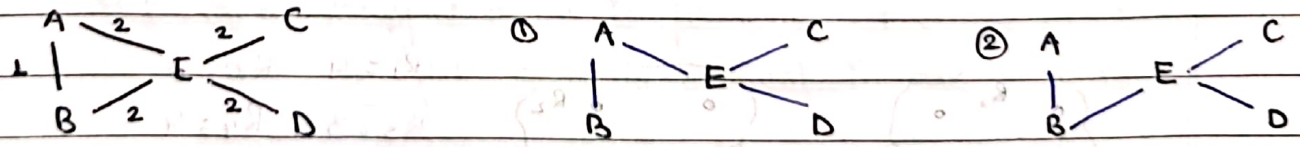
$\Rightarrow E \log E + E \times O(1)$

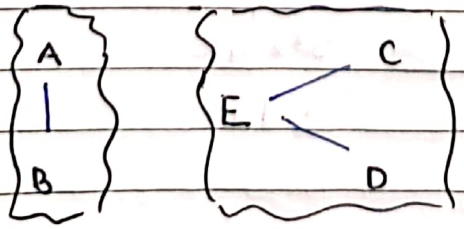
$\Rightarrow O(E \log E)$

① Kruskal only cares about the order of edges (ascending) so it can easily work with -ve weights.

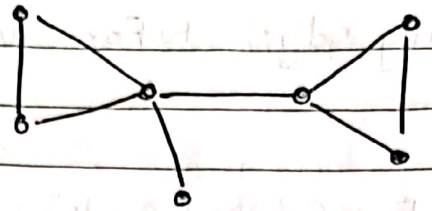
② If a const. value is added/subtracted from all edges, the order of sorting doesn't change, so MST doesn't change.

Ex. How many MSTs for following Graph possible?



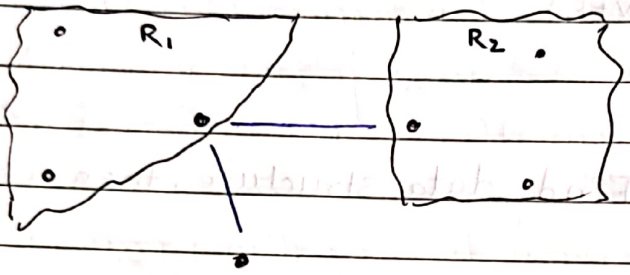


We have two options to connect the two parts.

Ex. 

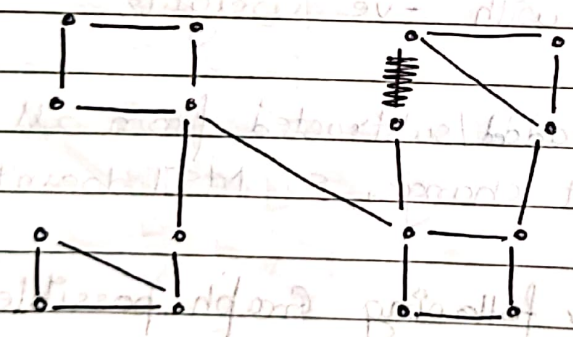
No. of MSTs if all edge weights are same.

Bridges will have to be in MST.

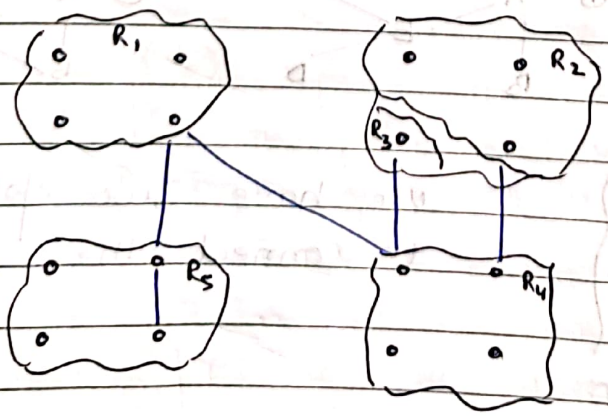


In R_1 we can choose any 2 of the 3 edges.
Similarly in R_2, choose any 2 of 3 edges.

$${}^3C_2 \times {}^3C_2 = 9.$$

Ex. 

All weights same.
No. of MST?

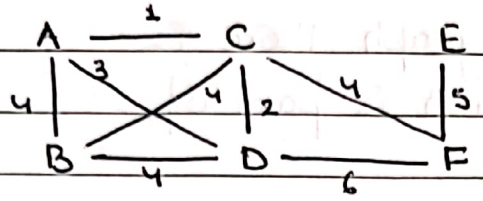


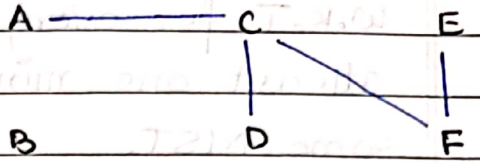
$R_1 : 4 \quad R_2 : 3$
$R_3 : 3 \quad R_4 : 4$

$3^2 \times 4^2$

$\Rightarrow 149.$

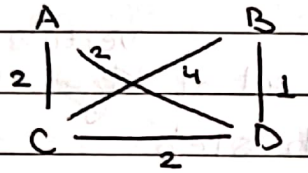
Ex. No. of MSTs.

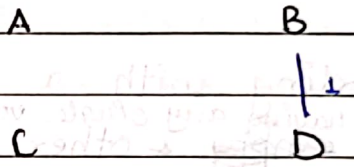




∴ 3 MSTs

Ex.





Any 2 of 3 edges with weight 2.

$^3C_2 = 3$.

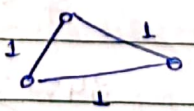
But one of the forms cycle.

∴ 2.

Ex. S1: Min weight edge is present in some MST
S2: Min weight edge is present in all MSTs.

S1: True

S2: False



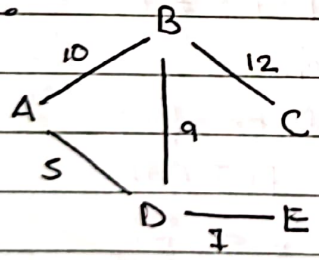
• Prim's Algorithm

W.K.T for every cut of a graph there is atleast one min. edge which is part of some MST.

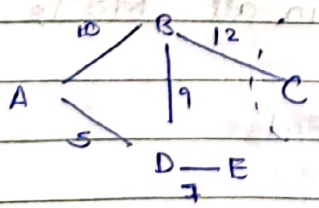
Prim's algorithm uses this property.
Take any set of vertices, one of the least weight edge in the cut is always part of MST.

Starting with a cut where one set of vertices having any single vertex & other has rest all, at each step choose the edge crossing the cut with min weight & move adjoining vertex to the first set.

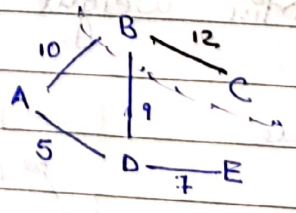
Ex.



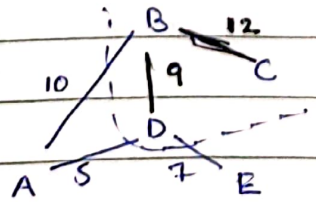
S1: {C} {A,B,D,E}



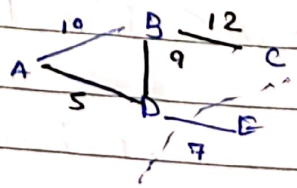
S2: {B,C} {A,D,E}



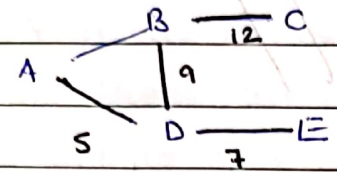
S3: {B,C,D} {A,E}



S4: {A,B,C,D} {E}



SS: {A, B, C, D, E}



Since we grow a single tree in Prim's algo. by bringing a vertex which isn't in the left set of the cut, i.e. no existing edge exists to this vertex yet. Adding the vertex adds one edge to the tree, & single edge can't create cycle.

Implementation

① Prim (G)

```
pq = [ ]
visited = [ ]
v = random vertex from G.V
visited [v] = true
push all edges adjacent to v in pq
tree = { }
while ! pq.empty ( ) ∧ tree doesn't consist of all vertices
    s, d = pq.Extract_Min ( )
    visited [d] = true
    for n ∈ G.Edges(d)
        if visited [n] = false
            pq.enque (d→n)
    tree ∪ {s→d}
```

Time Complexity (Prim)

⇒ $1 \times Adj(u) + (V) \times (T.C.(Extract\ Min)) + (E - deg(u))$

⇒ $deg(u) \times T.C.(Push\ in\ PQ) + V \times T.C.(Extract\text{-}Min) + (E - deg(u)) \times T.C.(Push\ to\ PQ)$

⇒ $V \times T.C.(Extract\ Min) + E \times T.C\ (Push\ to\ PQ)$

① Using Binary Heap

⇒ $V \times \log E + E \times \log E$

⇒ $(V+E) \log E$

② Prim (G)

```
dist = [ ]
dist [...] = ∞
pick random node from G ⇒ v
dist [v] = 0
tree = ∅
pq = [ ]
pq.enque (v)

while ! pq.empty()
    u = pq.Extract_Min()
    tree = tree ∪ {u}
    for all v adjacent u
        if v in pq and d[v] => Wuv
            d[v] = u
            parent[v] = u.
```

Prim vs Dijkstra

Prim at every step finds the next node closest to the ~~tree~~ current tree.
Dijksta at every step finds the next node closest to the source.

Time Complexity:

$V \times T(\text{Extract Min}) + E \times T(\text{Relax})$ (AL)

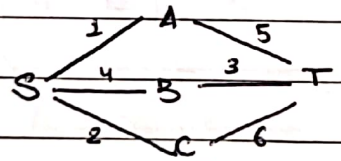
$V \times T(\text{Extract Min}) + E \times T(\text{Relax}) + V^2$ (AM)

① Binary Heap: $O(V \log V + E \log V)$

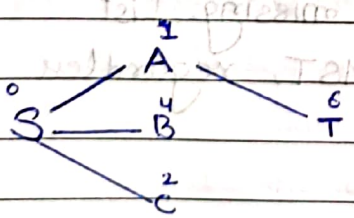
② Fibonacci Heap: $O(V \log V + E)$

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

Ex.



Dj:

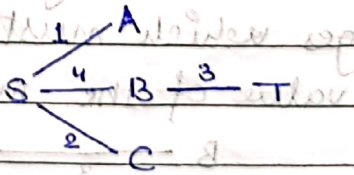


Cost = 12

S→A = 1 S→C = 2

S→B = 4 S→T = 6

Prim:



Cost = 10

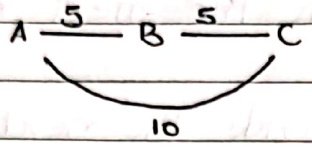
S→A = 1 S→C = 2

S→B = 4 S→T = 7

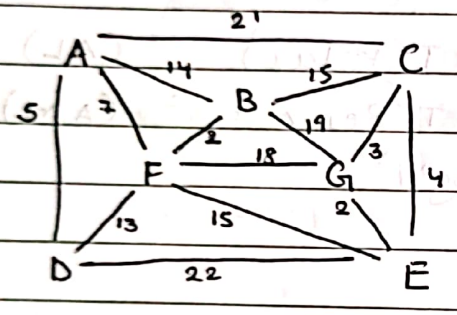
$\therefore$ Dj aims to minimize dist from source to individual nodes & Prim aims to minimize tree traversal cost.

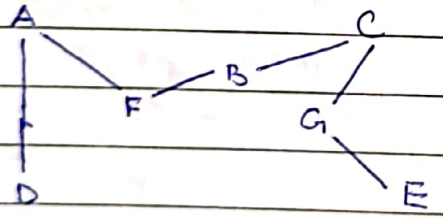
- MST ⇔ Shortest Path.

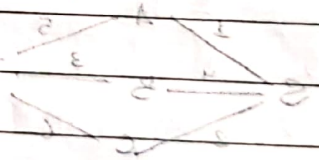
Ex.



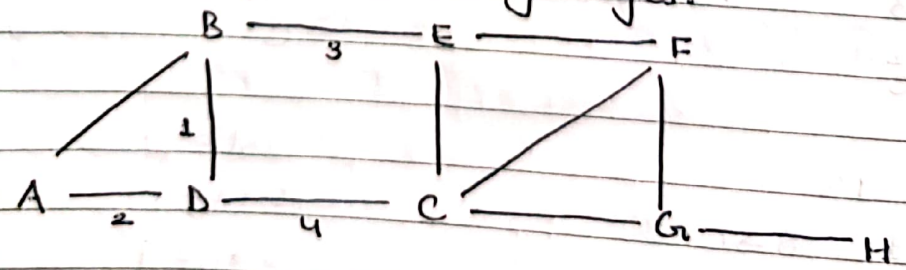
Ex. Find MST.







Ex. In G, some edge weights are missing. List the edges which must be in the MST, regardless of the value of the missing edges.



- G-H is a bridge.
- For the cut $\{D\}$ $\{A,B,C,E,F,G,H\}$, all the edge weights are known. Since for every cut min. edge crossing it must be in every MST if unique. ∴ B–D is in MST.

- For cut $\{A,B,D\}$ $\{C,E,F,G,H\}$, B–E is is min.

Ex. Consider an undirected graph with distinct +ve weights & MST T. If we square every edge weight & compute the MST again will we still get T?

with original weights, if

$e_1 < e_2 < e_3 \ldots$

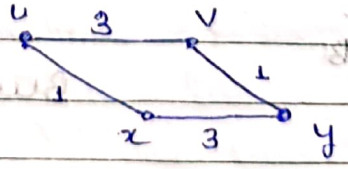
then

$e_1^2 < e_2^2 < e_3^2 < \ldots$

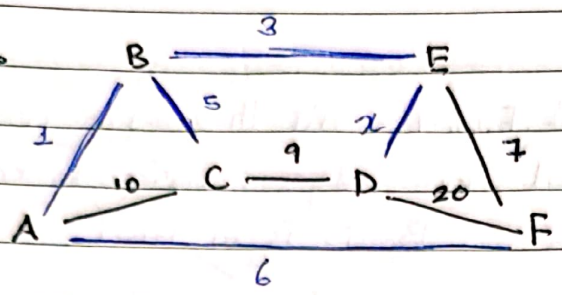
Since all edge weights are +ve & unique.

Ex. Edge $e=(u,v)$ doesn't belong to ~~any~~ an MST of G iff u & v can be joined by a path consistingly entirely of ~~path~~ edges with weight less than e.

False.

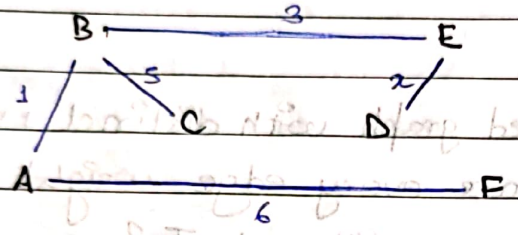


The path uxy consists of edges with weight equal to uv.

Ex. 

Comment about range of value of x.

Blue is MST.

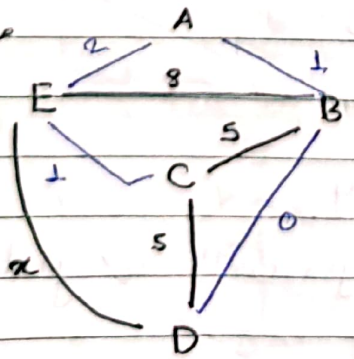


Say we add E-F. But this doesn't form cycle with E-D.

Adding C-D would form cycle with DE.
So, $x \leq 9$.

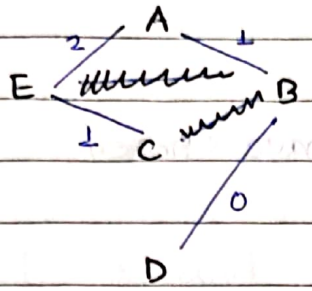
Similarly adding DF forms cycle.
So, $x \leq 20$

$\therefore x \leq 9$.

Ex. 

Comment on value of x.

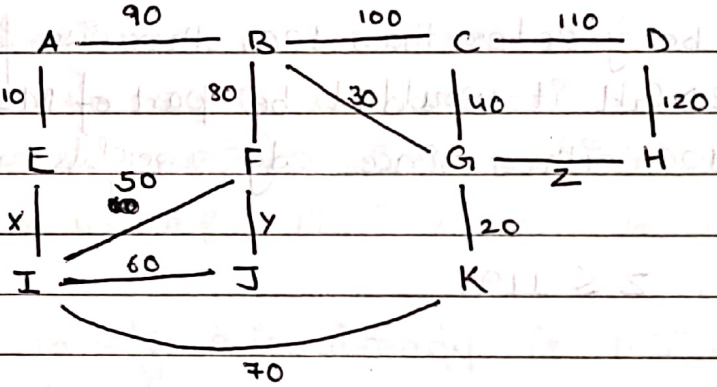
Blue is MST.



Adding DE, forms cycle, ABDE.
∴ value of α must be ≥ 2, that's why it isn't in MST.

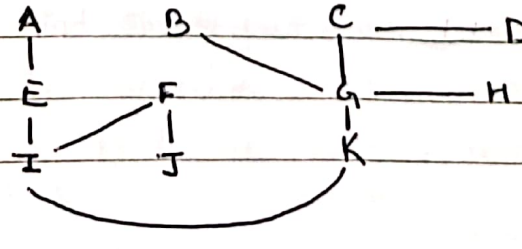
Ex. In graph G a certain MST comprises the edges with weight X, Y & Z, as well as seven other edges.

① Identify the other seven edges.



Since Z is in MST, it must be the heaviest in CDHG.
∴ Cut {D} {Rest} must join through CD.
Cut {A} {Rest} ⇒ AE.
Cut {AEI} {Rest} ⇒ IF
∵ Y is in MST, ∴ IJ isn't.
Cut {AEIJF} {Rest} ⇒ IK
{AEIJFK} {Rest} ⇒ GK
{ —"— G} {—"—} ⇒ BG
{ —"— BH} {Rest} ⇒ GC.

∴ {AE, EI, IF, FJ, IK, GK, BG, GH, GC, CD}



② Could x be 120

No. For cut {AE} {Rest} we'd have chosen 90 otherwise.

③ Could Y be 55?

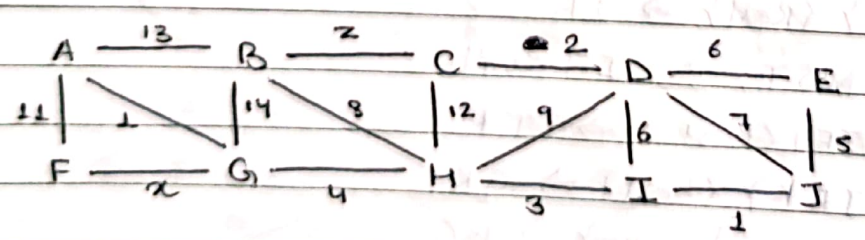
Yes. We chose Y over 80.

④ What is largest pos. value of z if all weights are distinct?

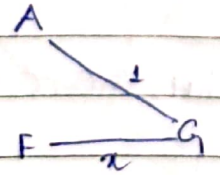
z can't be greater than 120, otherwise for the cycle CDGH it wouldn't be part of MST. z can't be 120 either since edge weights are distinct.

$\therefore z \leq 119$

Ex. Suppose MST of follo graph consist edge FG & BC with weight x & z resp. What are the maximal values of x & z.

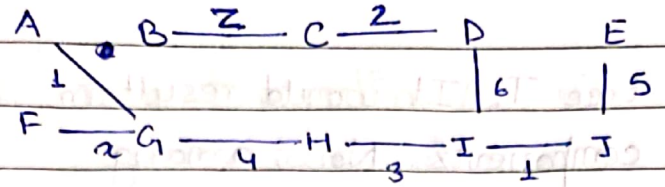


Say we start with A.



since FG & AG are in MST, AF can't.

$\Rightarrow x \leq 11.$



If we add BH to MST then cycle BCDIH would form.

$\therefore Z \leq 8$.

- Describe an efficient algorithm which finds the MST when: (T is current MST)
  1. An edge, $e \in T$ is decreased in weight.
  2. An edge, $e \notin T$ is decreased in weight.
  3. An edge, $e \in T$ is increased in weight.
  4. An edge, $e \notin T$ is increased in weight.

(1) If an edge e is already in MST, and then its cost is reduced, the MST won't change since if it was earlier taken with cost c_1 then with cost $c_2 (< c_1)$ it must also be taken.

(2) Add the new decreased edge e to T, it obviously forms cycle. Whichever is the heaviest in the cycle, remove that edge.

1. Detect Cycle : $O(V+E)$ {Only in Tree edges, i.e $E = V-1$}
2. Find max edge weight : $O(E)$
3. Remove from T : $O(1)$

$\therefore O(V+E)$

Alt: In G, say (u,v) is decr. [(u,v) not in T].

1. Find simple path from u to v in T. $O(V)$
2. If any edge on this path has weight greater than new weight of (u,v), remove the max weight edge on path & add (u,v) to T. $O(V)$

③ Remove e from tree T. It would result in two disconnected components. Now, among all the edges crossing the cut, find the min edge.

① Remove edge from T : O(1)

② Run DFS to find the components O(V+E)

③ Find edges passing from one component to another : O(E)

④ Find the min. among the edges : O(E)

∴ O(V+E)

④ If non-tree edge cost increases, then the tree won't change, since it would only increase cost.

* Huffman Encoding

- Optimal Codes
  Inp: Some distribution on characters.
  Out: A way of encoding the characters as efficiently as possible.

- Fixed len codes: All codes for each character is of same size.
  Say, A=000 B=001 C=010 D=011 ... H=111
  & freq: A=45 B=100 C=10 D=50 E=20
  then 675 bits.

- Variable len codes: Codes for different characters have different bit length.
  Ex. A=01 B=011 C=1 D=10 E=110
  & freq is same as above,
  then 560 bits used.
  Hence less bits used.

  But there can be ambiguity while decoding.
  Say Code: 01101
  Decode 1: ACA Decode 2: ADC
  Decode 3: BA
  ∴ Ambiguity.

So we want coding strategy with following properties:
① Uses least possible bit size.
② Non-ambiguous.

- Prefix ~~Code~~ Free Code: A coding scheme is called prefix free if encoding of no character is prefix of any other encoding of character. (a.k.a Prefix Code)

Ex. [T/F] If all codes in a PFC are reversed then the result is also prefix free?

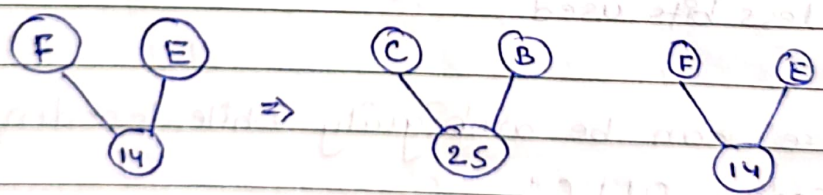
No. Counterexample:

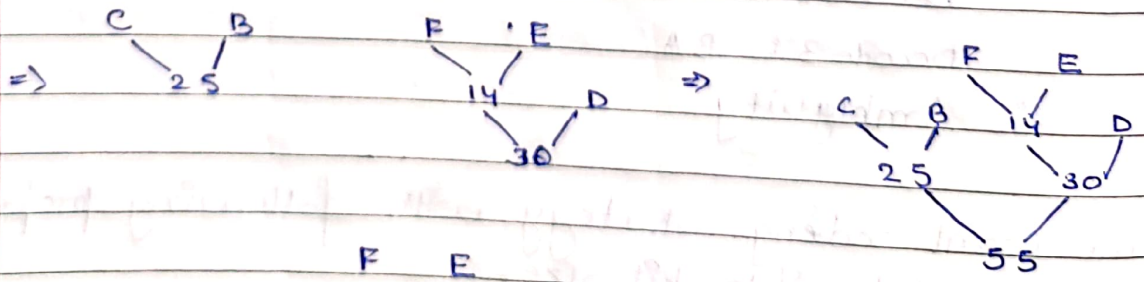
$\begin{bmatrix} 01 \\ 1 \end{bmatrix}$ $\begin{bmatrix} 10 \\ 1 \end{bmatrix}$

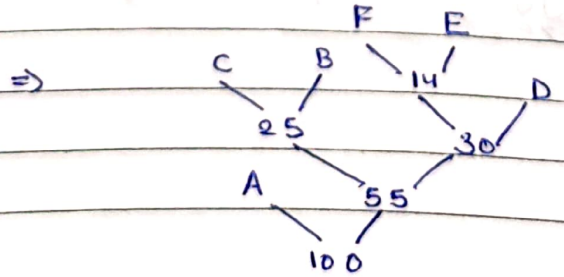
PFC Non PFC

- Huffman Encoding Approach: Greedily build subtrees by merging, taking 2 most infrequent chars at each step.

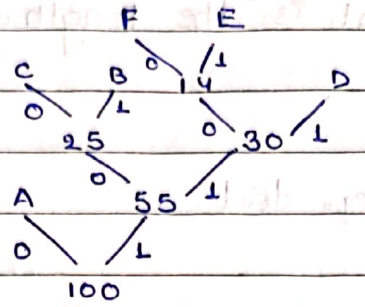
Ex. freq: A=45 B=13 C=12 D=16 E=9 F=5







Now we assign codes:



∴ A=0 B=101 C=100 D=111 E=1101 F=1100

So, bit length = 224

(w/ fixed len encoding => 300).

- Huffman codes are always prefix free, since all chars are on leaf, so there can't be any other char below another.

- Implementation

```
Huffman (c)
    n = |c|
    PQ = C
    for all chars in C
        x = Extract-Min (PQ)
        y = Extract-Min (PQ)
        z = NewNode (Left: x, Right: y)
        z.freq = x.freq + y.freq
        Insert (PQ, z)
```

TC = $n \times (\log(n) + \log(n))$

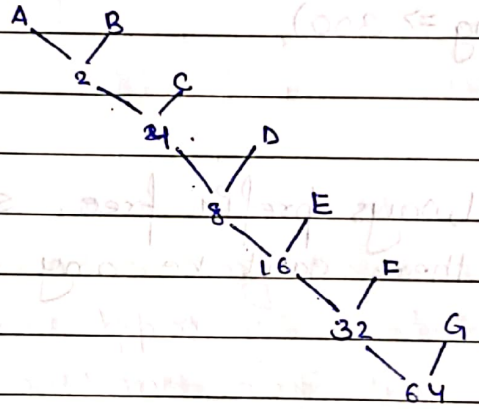
=> $n \times \log(n)$

∴ $O(n \log n)$

Ex. For Huffman encoding of n chars with freq. $f_1, f_2, \ldots f_n$ what is the length of longest codeword possible.

Say there's a freq. dist as

A	B	C	D	E	F	G
1	1	2	4	8	16	32

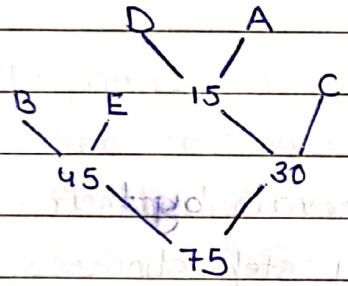


∴ max length can be (n-1) since can't have any more levels.

* Optimal Merge Pattern

Inp: A set of files of different length.
Out: An optimal sequence of two-way merges to obtain a single file.

Ex. The min. no. of record movements required to merge 5 files: A (10 records) B(20 records) C (15 records) D (5 records) E (25 records)



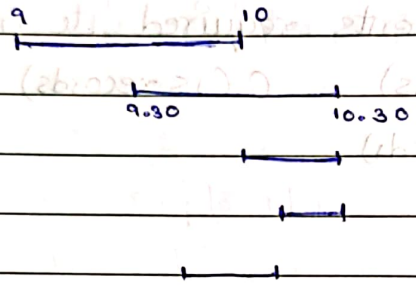
∴ 15 + 30 + 45 + 75 = 165

Number of record comparisons: 14 + 29 + 44 + 74 = 161

* Interval Scheduling

Given a set of intervals, find the subset of interval of largest size possible such that no two intervals are possible.

Ex. [9, 10], [9.30, 10.30], [10, 10.30], [10.15, 10.30]
[9.45, 10.15]



- Greedy Approach: Sort the intervals by their finish time, and then in every step choose the interval with start time ≥ the latest finish time.

Ex. [9,10], [9.30, 10.30], [10, 10.30], [10.15, 10.30], [9.45, 10.15]

Sorted: [9,10], [9.45,10.15], [9.30, 10.30], [10, 10.30] [10.15, 10.30]

Interv.	Latest Finish Time	Selected
S1: [9,10]	0	✓
S2: [9.45, 10.15]	10	x
S3: [9.30, 10.30]	10	x
S4: [10, 10.30]	10	✓
S5: [10.15, 10.30]	10.30	x

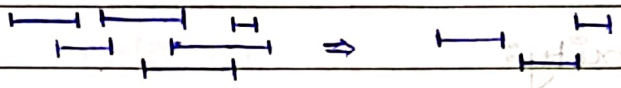
∴ At max 2 intervals can be selected.

Time Complexity = $T(\text{Sort intervals}) + n \times \Theta(1)$

$\Rightarrow n\log n + n$

$\therefore O(n\log n)$

Ex. Does following algo work:
Choose the interval x that starts last, discard all classes that conflict with x & recurse.

Say: 

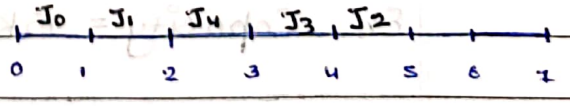
It would.
When starting from earliest finish time, we picked the next interval with $ST_{i+k} \geq FT_i$.
This just flips the ~~ones~~ intervals & chooses the latest start time, & pick next interval with $FT_{i-k} \leq ST_i$

* Job Scheduling with Deadline

Given some jobs, their deadline & their profit schedule the jobs such that the profit is maximised.

Ex.

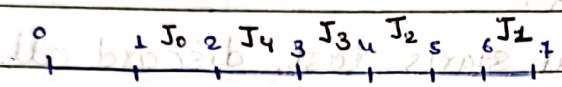
JId	Profit	Deadline
0	10	2
1	30	7
2	15	5
3	20	4
4	40	3



↑ this is one possible schedule.

- Greedy Approach: Sort the jobs in reverse order of profit (i.e. descending). Loop over & pick the ~~[illegible]~~ latest interval before the job's deadline.

Ex. [Previous Example]



Time Complexity:

T(Sort Descending)
+ n x T(Put in right slot)

$\Rightarrow n \log n + n^2$

$\Rightarrow O(n^2)$

* Fractional Knapsack

Given some items with their weight & profit. You have a fixed size (in weight) knapsack you have to pick items to fit in your knapsack such that the profit is maximum. Picking fraction of item allowed

Ex. Capacity = 15 kg

	I_1	I_2	I_3	I_4	I_5
Weight:	12	1	2	1	10
Profit:	4	2	2	1	4

- Greedy Approach: Sort items by their Profit/Weight ratio. Pick items in descending order.

Ex. [Previous Example].

	I_2	I_3	I_4	I_5	I_1
Weight	1	2	1	10	12
Profit	2	2	1	4	4
P/W	2	1	1	0.4	0.3

Cap: 15 Kg. Profit = 0

S1: Cap $\geq W_{I_2}$ $\Rightarrow$ Pick Whole

Cap = 15 - 1 = 14

Prof. = 0 + 2 = 2.

S2: Cap $\geq W_{I_3}$ $\Rightarrow$ Pick Whole.

Cap = 14 - 2 = 12

Prof. = 2 + 2 = 4

S3: Cap $\geq W_{I_4}$ $\Rightarrow$ Pick Whole.

Cap = 12 - 1 = 11

Prof. = 4 + 1 = 5

S4: Cap $\geq W_{I_5}$ $\Rightarrow$ Pick Whole

Cap = 11 - 10 = 1

Prof. = 5 + 4 = 9.

S5: Cap $< W_{I_1}$ $\Rightarrow$ Pick Partial.

Part to pick = Cap / W_{I_1} = 1/12.

Part Prof. = $4 \times \frac{1}{12} = 0.3$

Cap = 1 - 1 = 0

Prof. = 9 + 0.3 = 9.3

$\therefore$ Prof = 9.3

- Time Complexity =

  T (Sort by P/ω)

  + n × O(1)

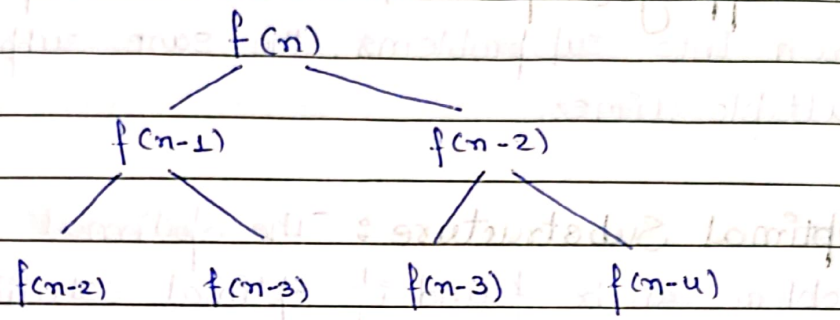
  ⇒ n log n + n

  ∴ O(n log n)

Dynamic Programming

There are problems which can be broken down into smaller problems.

Say calculating Fibonacci Series. For n^{th} Fibonacci series



$\therefore 2^0 + 2^2 + 2^3 + \dots$

$LB = 2 + 2^2 + 2^3 + \dots = 2^{n/2} = 2^{n/2+1} - 2$

$UB = 2 + 2^2 + \dots + 2^n = 2^{n+1} - 2.$

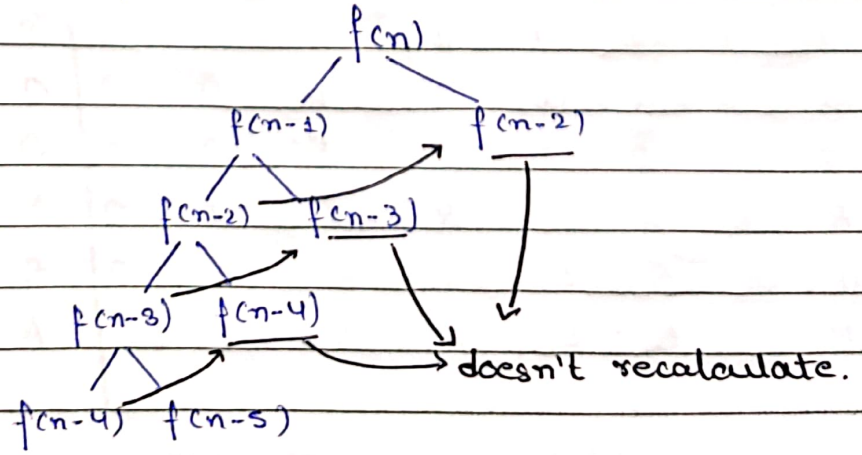
$\therefore \quad TC = O(2^n)$

Two noticable things:

(1) The problem takes exponential time.

(2) There are subproblems which appear again & again.

Dynamic Programming is an approach to try to solve a problem by efficiently calc. result of subproblem the least no. of times possible.



* A Dynamic Programming approach can be devised for problems with the following properties:

(1) Overlapping Subproblems: When the problem is broken down into subproblems, the same subproblems appear multiple times.

(2) Optimal Substructure: The optimal solution to the problem can be found if optimal solutions to the subproblems are found.

* Longest Common Subsequence.

Given a string, a subsequence of it can be found by deleting 0 or more chars without modifying the order of chars.

Given two strings, A and B, find the longest possible subsequence which belongs to both A and B.

- Recursive Formula

$|A| = m \quad |B| = n$.

LCS(i, j) describes the length of longest common subsequence of A[0..i] and B[0..j].

$$LCS(i,j) = \begin{cases} 0, & i \leq 0 \text{ or } j \leq 0 \\ 1 + LCS(i-1, j-1), & A[i] == B[j] \\ \max(LCS(i-1, j), LCS(i, j-1)), & A[i] \neq B[j] \end{cases}$$

- Time Complexity

(1) Recursive Approach: $O(2^{\min(m,n)}) \leq TC \leq O(2^{m+n})$

(2) DP Approach: $O(m \times n)$

Ex. X = <B, D, C, A, B, A> Y = <A, B, C, B, D, A, B>

	0	A	B	C	B	D	A	B
0	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

∴ max len = 4 BCAB.

* Matrix Chain Multiplication

Given a sequence of matrices, find the order of matrix multiplication such that the no. of computations is minimum.

Ex. A_1 A_2 A_3 A_4

10×100 100×200 200×20 20×10

① $((A_1 A_2) A_3) A_4$: $10 \times 100 \times 200 + 10 \times 200 \times 20 + 10 \times 20 \times 10 = 24,2,000$

② $(A_1 A_2)(A_3 A_4)$: $10 \times 100 \times 200 + 200 \times 20 \times 10 + 10 \times 200 \times 10 = 28,0,000$

③ $A_1((A_2 A_3) A_4)$: $100 \times 200 \times 20 + 100 \times 20 \times 10 + 10 \times 100 \times 10 = 430,000$

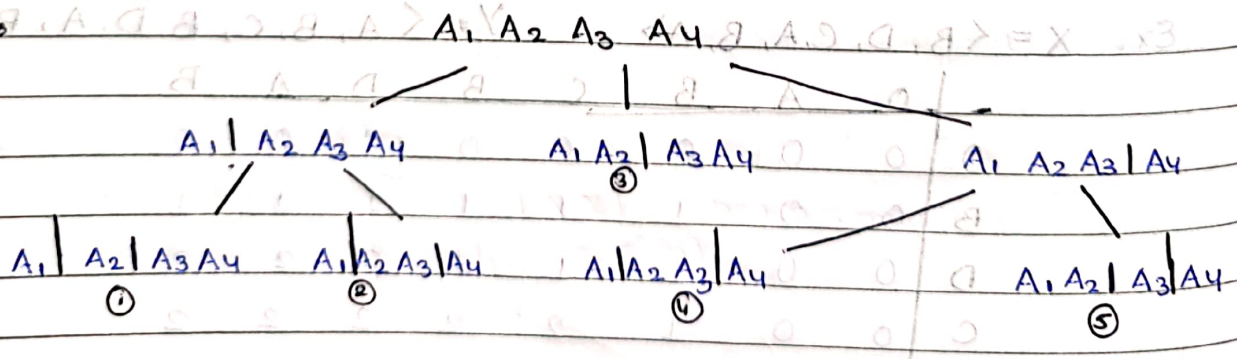
④ $(A_1(A_2 A_3))A_4$: $100 \times 200 \times 20 + 10 \times 100 \times 20 + 10 \times 20 \times 10 = 422,000$

So the costs differ drastically.

We want to find the least no. of operations.

- Given a seq. of n matrices total no. of multiplication orders is given by Catalan Number of $(n-1)$.

  i.e. $\frac{^{2(n-1)}C_{(n-1)}}{n}$

- 

also, $\frac{^{2\times 3}C_3}{4} = 5$

- We can notice same multiplication operations being performed again & again.
Also the optimal sol. of problem can be found as the min. of optimal solution of subproblems.

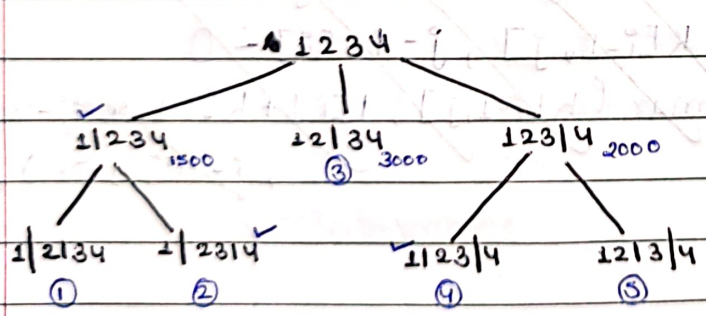
Also, note none of the "ordering of mult." (leaves) are same.

- Let the min. cost of Mult. $A_i \dots A_j$ be $M[i,j]$. then

$$M[i,j] = \begin{cases} \min_{k=i}^{j-1} \left(M[i,k] + M[k+1,j] + \text{row}_i \times \text{col}_k \times \text{col}_j \right) & , \ i \neq j \ \& \ i \leq j \\ 0 & , \ i = j \text{ or } i > j \end{cases}$$

Ex. A_1 A_2 A_3 A_4

10×5 5×20 20×10 10×5



	A_1	A_2	A_3	A_4
A_1	0	1000	1500	1500
A_2	0	0	1000	1250
A_3	0	0	0	1000
A_4	0	0	0	0

$\therefore \ 1(23)4)$

- Time Complexity:
  (1) DP Approach: $O(n^3)$
  (2) Recursive Approach: $\geq O(2^n)$

* 0-1 Knapsack.

Given some items with their profit & weight, and a knapsack with capacity W, choose some or all items & put in the knapsack, such that the cumilative weight of taken items is $\leq W$ & the profit (cumilative) is maximal.

- For each item, we have a choice to either take it or leave it. An item can be taken if its weight along with already taken items is less than or equal to W.

- Let α be capacity. P[i] & W[i] be profit & weight of i^{th} item. Let K[i, j] be the max. possible profit of choosing from Item$_1$...$_i$ & capacity j.

$$K[i,j] = \begin{cases} 0 & , i=0 \text{ or } j=0 \\ K[i-1, j] & , j - W[i] < 0 \\ \max(K[i-1,j], P[i] + K[i-1, j-W[i]]) & , j - W[i] \geq 0. \end{cases}$$

- Time Complexity
  (1) DP Approach : $O(n \times \alpha)$ when n is num. items & α is cap.
  (2) Recursive Approach: $O(2^n)$

$$K[i,j] = \begin{cases} 0 & , i=0 \text{ or } j=0 \\ K[i-1, j] & , j - W[i] < 0 \\ \max(K[i-1,j], P[i] + K[i-1, j-W[i]]) & , j - W[i] \geq 0 \end{cases}$$

Ex.

Item	P	W
1	2	1
2	2	2
3	3	4
4	5	5

Cap: 6.

	0	1	2	3	4	5	6
I_0	0	0	0	0	0	0	0
I_1	0	2	2	2	2	2	2
I_2	0	2	2	4	4	4	4
I_3	0	2	2	4	4	5	5
I_4	0	2	2	4	4	5	7

∴ Max. Profit = 7

* Floyd Warshall : All pair shortest path

- TC: $O(|V|^3)$
- Can work on -ve edges.
- If -ve cycle then at least one of the diagonal is -ve.

* Travelling Salesman:

- TC: $O((n-1)!)$
- No polynomial time solution exists.

Sorting Algorithms.

* Quick Sort, Quick Select, Median Heuristic.

- For Quick Sort we generally want to find pivot element as close to the middle as possible.
  Using Median as the pivot is a good approach for that.

- The partition method can easily be employed to find (select) k^{th} element from the array {k^{th} element in sorted order}. This is called "Quick Select".
  In average case Quick Select gives k^{th} element in $O(n)$ & in worst case $O(n^2)$.

- Further 'Median of Median" algo. uses this: "Given an array of n elements divide it in chunks of size K (constant), & for each of n/k chunk, find median. Then create array of (n/k) medians & find median of them."

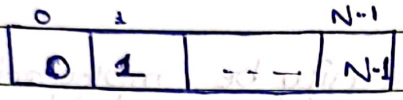
  - $O(1) \times \frac{n}{k} + O(\frac{n}{k}) = O(n)$ {Avg Case}
  - In worst case (also) $O(n^2)$.

- Suppose we use median as pivot for Quick Sort, we'll have
  $T(n) = 2T(\frac{n}{2}) + \Theta(n) + O(n)$ {$n/2$ since we divide in middle}
  ($\Theta(n)$: median finding; $O(n)$: pivot algo.)

* Bubble Sort

- Compare pair of adjacent items. Swap if items out of order.

- Best Case: Array is already sorted.



#swaps = 0

#comparisions $\Rightarrow n-1$ { if we stop after first pass }

$\Rightarrow \frac{n(n-1)}{2}$

- Worst Case: Array is reverse sorted.

#swaps = #comparisions = $\frac{n(n-1)}{2}$

- In avg & worst case : $O(n^2)$

best case : $O(n)$ { w/ early termination }

- Inplace, Stable.

* Insertion Sort

- At each iteration i, assume $A[0 \ldots i]$ is sorted already. Put $A[i+1]$ at it's place in $A[0 \ldots (i+1)]$ for i in range 0 to $(n-1)$.
- Can also be used to "Count Inversions".
- Best Case: Already Sorted.
  $\#swaps = 0$
  $\#comparisons = (n-1)$
- Worst Case: Reverse Sorted.
  $\#swaps = \#comparisons = 0+1+2+\ldots(n-1)$
  $\Rightarrow \frac{n(n-1)}{2}$
- In worst & average case $= O(n^2)$
  best case $= O(n)$
- Inplace, Stable sort.

* Selection Sort

- Find largest of n elements & put in last. Repeat for $(n-1)$ & so on.
- Best Case: Already Sorted.
  $\#swaps = 0$ $\#comparisons = \frac{n(n-1)}{2}$
- Worst Case: Reverse sorted. as N, 1, N-1, 2, ...
  $\#swaps = (n-1)$ $\#comparisons = (n(n-1))/2$
- In all cases $= O(n^2)$.
- Inplace, Unstable.

* Theorem: Any comparision based sorting algorithm, must take atleast $n \log n$ in worst case.
i.e. for worst case, the time complexity is $\Omega(n \log n)$.