Q What is an Algorithm?

→ Finite set of steps to solve a problem is called algorithm.

→ Step → instn

→ instn → fundamental operators (+, −, *, /, etc...)

→ Characteristics of instructions :-

1) Definitive :-

Every instn must be defined without any ambiguity.

Eg i = i(1)1 ; → invalid

i = i+1; → valid

2) Finiteness :-

Every instn must be terminated within finite amount of time.

Eg Define an instn i(+)1

```
i = 1
while (1) ~
{
    i = i+1;
}
```
← invalid instruction.

3) Every instruction must accept at least 0 input & provide at most 1 output.

'+' a = i + j

→ Steps involved for solving any problem :—

1) Identifying problem statement :—

We have to identify the problem.

→ What problem wants us to do?

For eg Arrange 4 queens (Q_1, Q_2, Q_3, Q_4) on a 4×4 chess board.

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

2) Identifying the constraints :—

To solve a problem what constraints we need to take care of.

...ld be on

need to take care of.

For eg :- No two queens should be on same row or col or diagonal.

3) Design logic :-

Depending on problem statement & Constraints

Following Design strategy can be used :-

1) Divide & Conquer.
2) Greedy method
3) Dynamic programming.
4) Branch & Bound.
5) Backtracking.

4) Validate :-

Validating whether our algorithm works on every test case.

→ Principle of mathematical induction.

→ Proof by contradiction.

5) Analysis :-

This is used to compare two algos w.r.t time & space complexity.

time & space complexity.

No-of registers, B/w etc...

Priory Analysis

1) Analysis is done before executing.

```
x = x+1;
```

Principle: Frequency count of fundamental instructions.

```
2  int n,sum=0;
   scanf("%d",&n);
n  for(int i=0; i<n; i++)
   {  sum += x;
   }
```

2) It provides estimated values

3) It provides uniform values n^2

Posterior Analysis

1) Analysis is done after execution.

```
x = x+1
```

sys1 0.1ns sys2 0.2ns sys3 0.5ns

2) Provides exact value.

3) Dependent on systems input. [non-uniform values]

→ Algo1 $\xrightarrow{1000}$ 0.1ns sys1

4) Independent of sys.

5) Can be used to compare two algos

Notations

→ Algo1 → 0.1ns sys1

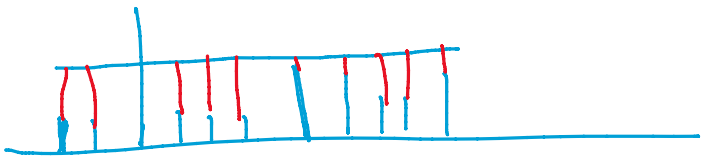
→ Algo2 → 1m → 1ns sys2

4) Dependent on system.

5) Cannot be used to compare two algo.

Asymptotic ⟶ Towards infinity.

These notations are used to tell the complexity of an algorithm when the input is very large ( )



Time Complexity

(n^2)

10 → 100

1000 → 10^6

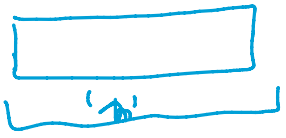
Approx no. of instructions.

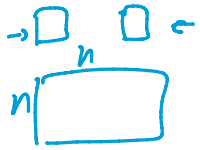
Space Complexity.

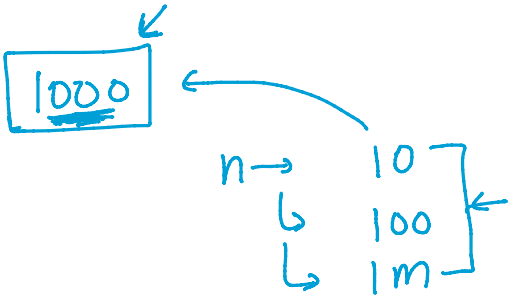
n

extra space that an algorithm takes except input

input 'n'







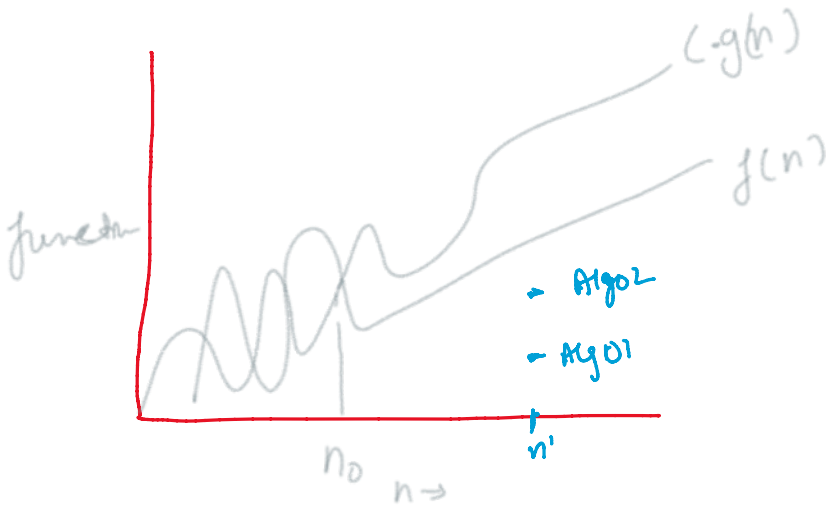
Following notations will be used:-

1) Big-Oh (O):-

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

$g(n)$ is "tight" upper bound of $f(n)$

$g(n)$ is "tight" upper bound of $f(n)$



$f(n) = O(g(n))$
iff
$f(n) \leq c \cdot g(n)$
$\forall n \geq n_0$, some constant $c > 0$

Algo 1 → $O(n^2 + 3n + 1) \rightarrow O(n^2 + n) \rightarrow O(n^2)$

Algo 2 → $O(2n^2 + 1) \rightarrow O(n^2)$

While Calculating complexities :-

→ Constants are ignored if it comes as addition, subtraction, multiplication or division.

→ lower order terms are ignored in addition & subtraction.

$O(n) = O(n * \frac{n}{n})$
$= O(\frac{n^2}{n}) = O(n^2)$

$O(n^3) = O(n^2 \cdot n) = O(n^3)$

Shortcut → Take the highest order term.

$$f(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_n n^n \quad a_n \neq 0$$

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

2) Big Omega (Ω) :-

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

$g(n)$ is "tight" lower bound of $f(n)$.

$f(n) = \Omega\, g(n)$ iff.

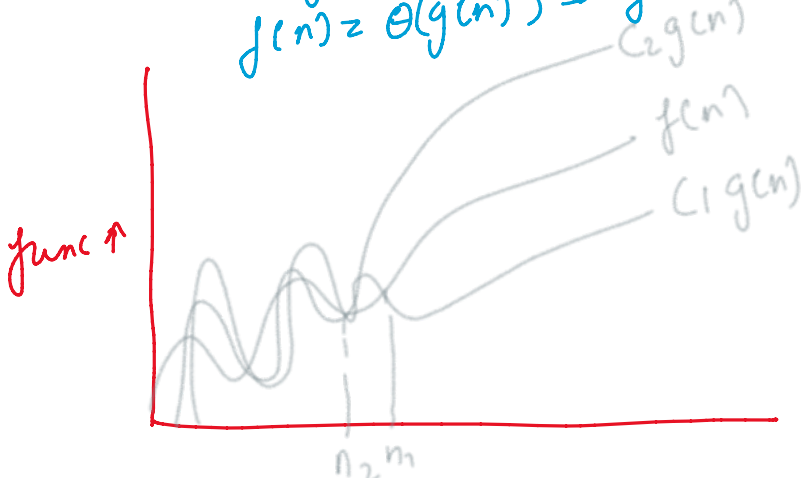
$f(n) \geq c \cdot g(n)$

$\forall\, n \geq n_0$ & some constant $c > 0$

3) Theta (Θ) :-

Theta gives the tight upper & lower bound both.

$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$ & $f(n) = \Omega(g(n))$



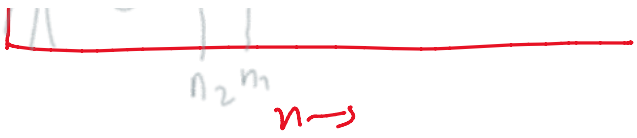
$f(n) = \Theta(g(n))$

iff.

$c_1 g(n) \leq f(n) \leq c_2 g(n)$

$\forall\, n \geq \max(n_1, n_2)$

some constant $c_1 \& c_2 > 0$.



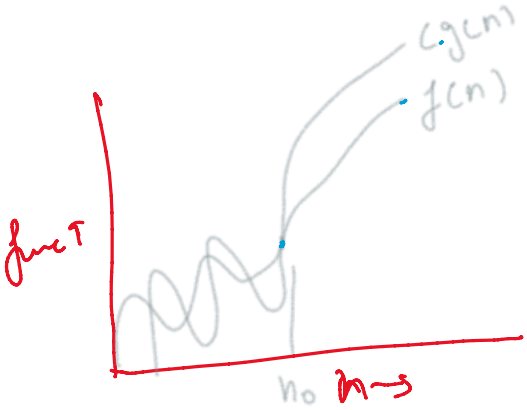
Some cases

4) small - oh (o) :-

o gives us upper bound.

$f(n) = o(g(n))$

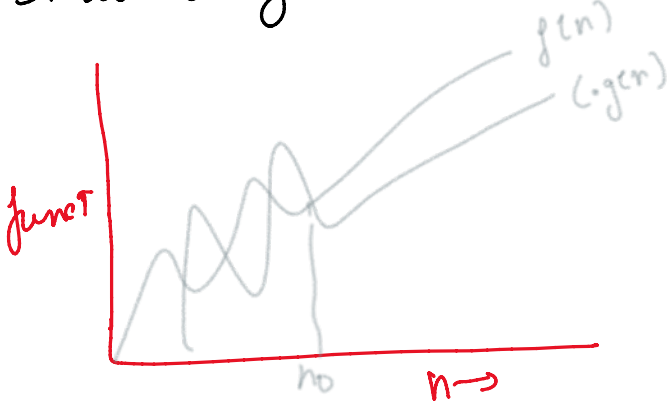
$f(n) < c \cdot g(n) \quad \forall n > n_0 \; \& \; \forall c > 0$



$n = o(n^2)$

$n < 1n^2$ (n_0)
$2n^2$
$0.5n^2$
$n < 0.001n^2$ n_0

5) small - omega (ω) :-



lower bound

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

$f(n) > c \cdot g(n)$
$\forall n > n_0 \; \& \; \forall c > 0$

$n^2 = \omega(n)$

Notes

1) $f(n) = O(g(n)) \rightarrow g(n) = \Omega(f(n))$
2) $f(n) = o(g(n)) \rightarrow g(n) = \omega(f(n))$.
3) $f(n) = \theta(g(n)) \rightarrow f(n) = O(g(n)) \; \& \; f(n) = \Omega g(n)$.

	Reflexive	Symmetric	Transitive
O	✓	X	✓
Ω	✓	X	✓
Θ	✓	✓	✓
o	X	X	✓
ω	X	X	✓

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

$f(n) = O(g(n))$
$g(n) = O(g(n))$
$a R b \rightarrow bRa$

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

$f(n) = \Theta(g(n)) \longrightarrow f(n) = O(g(n))$ & $f(n) = \Omega(g(n))$

$g(n) = \Omega(f(n))$ & $g(n) = O(f(n))$

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

$a R b$ & $b R c \rightarrow a R c$

$f(n) = O(g(n))$ & $g(n) = O(k(n)) \rightarrow f(n) = O(k(n))$

An algo may have :-

1) Simple loop(s)
2) Nested loops
3) if - else
4) Recursive func -

1) Simple for loop(s) :-

To calculate the time complexity we will count the frequency of fundamental instns.

for ex

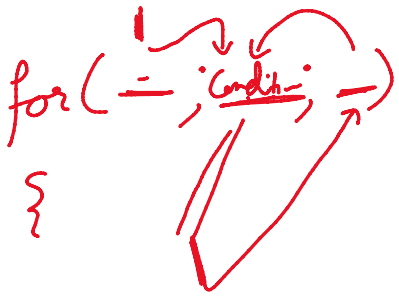
```
int sum = 0;          // 1
for ( int i = 1; i <= n; i++ )   // i=1 : 1 ; i<=n : n+1 ; i++ : n
{
    sum += i;         // n
}
```

1000 ② ✓

$1 + 1 + (n+1) + n + n$

$= (3n+3)$

$= O(n)$.



$n = 5$

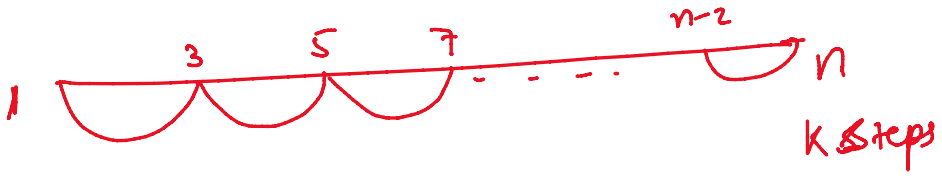
$i = 1, 2, 3, 4, 5, 6$

$1 \quad 1 \quad 1 \quad 1 \quad 1 \quad = 5$

$\Rightarrow O(n) \rightarrow$

Ex 2

```
int sum=0;
for( int i=1; i<=n ; i+=2)
{
    sum+=i;
}
```



1, 3, 5, 7 ... n AP

The number of term which I want

$a + (k-1)d$ ← difference

first term $d = t_2 - t_1$

$3 - 1 = 2$

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

$(n-1) = (k-1)2$

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

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

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

Time complexity $= O\left(\frac{n+1}{2}\right) = O(n)$.

Ex 3

```
int sum=0;
for (int i=1 ; i<=n ; i=i*2)
{
    sum+=i;
}
```

$i = 1, 2, 4, 8, 16 \ldots n$

$2^0, 2^1, 2^2 \ldots 2^k$

$1, 2, 4, 8, 16 \ldots n$

$a = 1, \quad r = \frac{t_2}{t_1} = \frac{2}{1} = 2$

$t_k = a r^{k-1}$

$a^{b-c} = \frac{a^b}{a^c}$

$n = 1 * 2^{k-1}$

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

$\log(a*b) = \log(a) + \log(b)$

$2^k = 2n$

$k = \log_2(2n) = \log_2(n) + \log_2(2)$

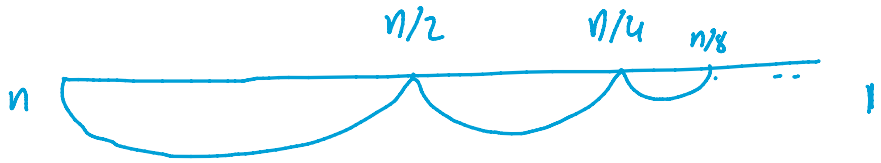
$\log_a a = 1$

$k = \log_2 n + 1$

Time Complexity $= O(\log_2 n + 1) = O(\log n)$.

Ex 4

```
int sum=0;
for (int i=n; i>=1; i=i/2)
{
    sum+=i;
}
```



$n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8} \cdots 1$

K steps

$a = n, \quad r = \frac{t_2}{t_1} = \frac{n/2}{n} = \frac{1}{2}$

$t_k = a r^{k-1}$

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

$2^{k-1} = n$

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

$2^k = 2n$

$k = \log_2 n + 1$

$T.C = O(\log_2 n)$

Nested Loops

Multiply complexity of each loop

→ loop 1 ↻ x
→ loop 2 y
(x*y)

1 → y
2 → y
⋮
x → y
(x*y)

Ex-1

```
for ( int i=1; i<= n ; i++ )
{
   for (int j =1; j<= n ; j++)
   {
      sum += j; //O(1)
   }
}
```

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

i	j	times
1	1→n	n
2	1→n	n
3	1→n	n
⋮	⋮	⋮
n	1→n	n
		n*n

Ex-2

```
sum=0;
for(int i=1;  i<=n ; i++)
{  for( int j=1 ; j<=n ; j+=2)
   {
      sum+= j;
   }
}
```

i	j	times
1	1→n	$\left(\frac{n+1}{2}\right)$
		$\left(\frac{n+1}{2}\right)$

}
}

T.C $O(n*(\frac{n+1}{2}))$

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

1	$1 \to n$	$(\frac{n+1}{2})$
2	$1 \to n$	$(\frac{n+1}{2})$
3	$1 \to n$	$(\frac{n+1}{2})$
⋮	⋮	⋮
n	$1 \to n$	$(\frac{n+1}{2})$
		$n(\frac{n+1}{2})$

Ex-3

```
sum=0;
for(int i=1; i<=n; i*=2)
{
    for(int j=1; j<=n; j+=2)
    {
        sum+=j;
    }
}
```

i	j	times
1	$1 \to n$	$(\frac{n+1}{2})$
2	$1 \to n$	$(\frac{n+1}{2})$
4	$1 \to n$	$(\frac{n+1}{2})$
8	$1 \to n$	$(\frac{n+1}{2})$
⋮	⋮	⋮
n	$1 \to n$	$(\frac{n+1}{2})$
		$\log n\ (\frac{n+1}{2})$

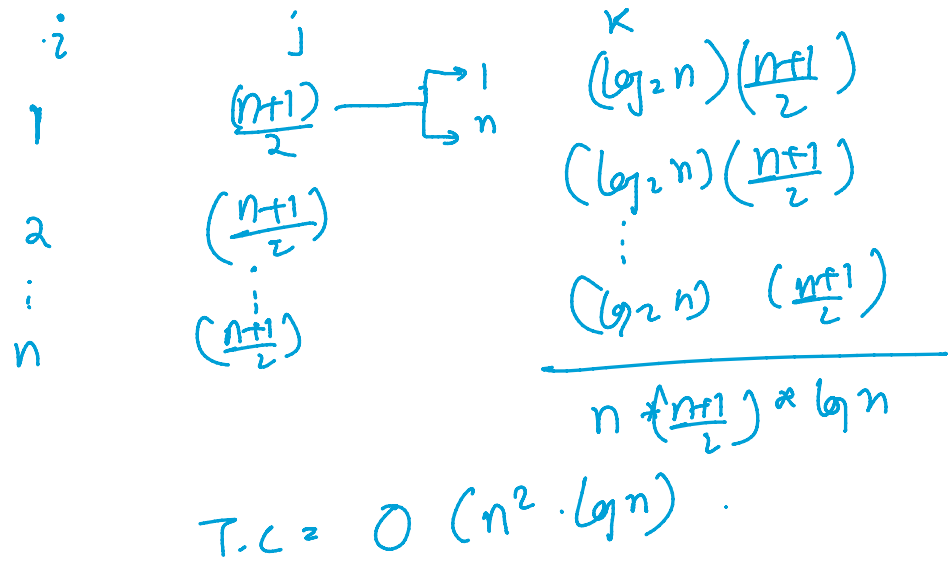
T.C = $O((\frac{n+1}{2})\log n)$

$= O(n \log n)$.

Ex-4

```
sum=0
for(int i=1; i<=n; i++)          → O(n)
  for(int j=1; j<=n; j+=2)       → O(n)
    for(int k=1; k<=n; k*=2)     O(log n)
      sum+=k;   (sqrt(k);)
```

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



T.C = $O(n^2 \cdot \log n)$.

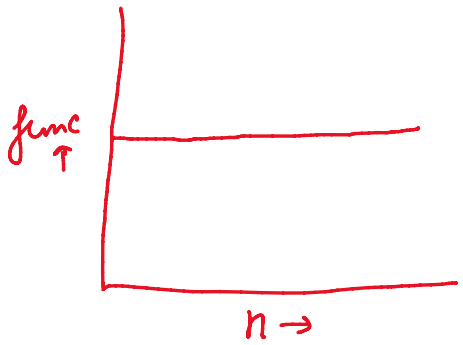
Types of complexities

1) Constant :- O(1)

Constant complexity means the complexity of an algorithm is independent of input size.

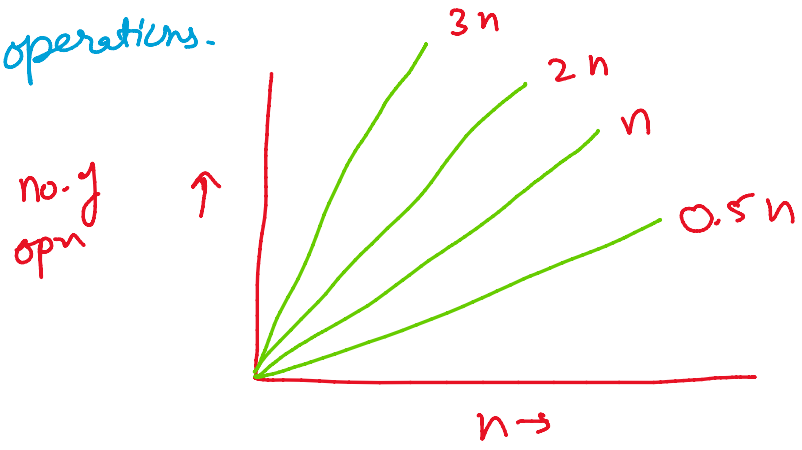
$n \rightarrow 1, 100, 1000, 1000000$

The no. of steps to solve the problem remains same.



2) Linear complexity:- O(n)

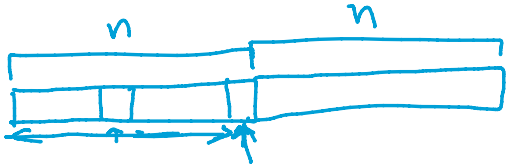
Linear complexity means the complexity of the algo. grows in direct proportion to the size of the input. That means, Algo. will solve a problem with input size 'n' in 'n' no. of operations.



3) Logarithmic Complexity :- $O(\log n)$

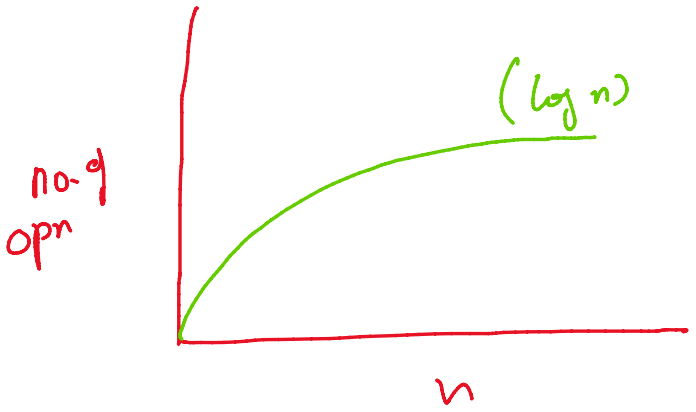
Complexity is in proportion to logarithm of input size.

Complexity of algo. will grow linearly with exponential increase in input size.



n → x

2n → x+1



After every step we are reducing the input by constant factor ($\frac{1}{2}$, $\frac{1}{3}$ ...)

4) Polynomial Complexity :- $O(n^k)$ $(n^2, n^3 .. n^{100})$

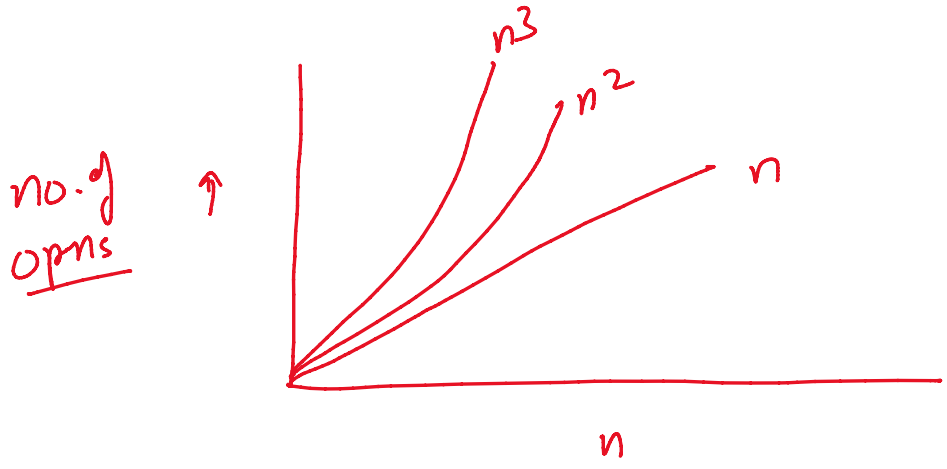
The complexity grows in proportion to k^{th} power of input size.

It takes a lot of opns on large dataset.

$O(n^3)$ $n = 10$, $n^3 = 1000$

$n = 100$, $n^3 = 10^6$.

$n = 1000$, $n^3 = 10^9$.



5) Exponential complexity :- $O(k^n)$

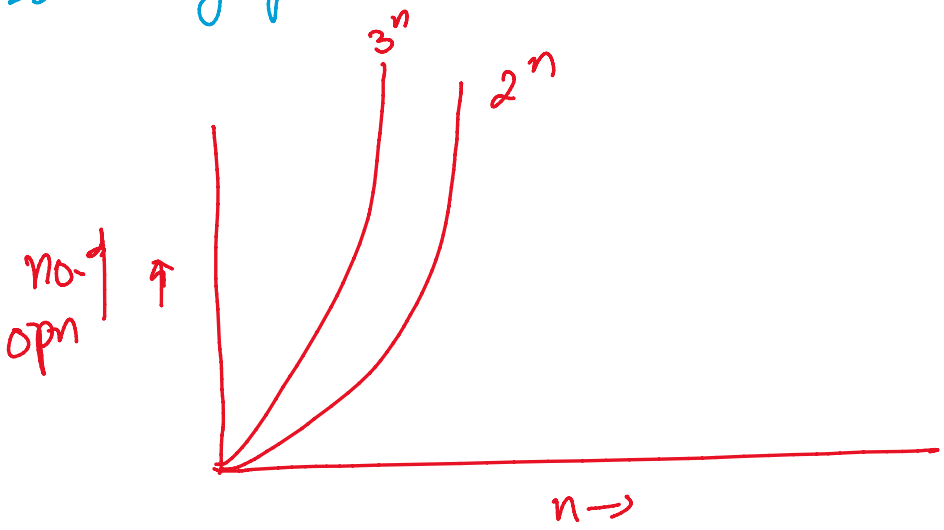
At each step the no. of operations become multiplied by k.

$O(2^n)$ =

$x = n+1$

$2^x \rightarrow 2^{x+1} = 2 \cdot 2^x$

These types of algos are used for exhaustive searching of solution. permutation & combination.



$O(1) < O(\log n) < O(n) < O(n^k) < O(k^n)$

$O(1) < O(\log n) < O(n) < O(n^2) < O(n^3)$
$< O(n^n)$.

Calculating Time complexity of loops using summation

```
for (int i = 1 ; i <= n ; i++)
{
    //O(1)
}
```

$$T.C = \sum_{i=1}^{n} 1 = 1+1+1 \text{ ---- } n \text{ times}$$
$$= n$$
$$= O(n)$$

```
for (int i = 1 ; i <= n ; i+=2)
{
    //O(1)
}
```

n = 5

$\left(\frac{5+1}{2}\right) = \frac{6}{2} = 3$

n = 4

$\left(\frac{4+1}{2}\right) = \frac{5}{2} = 2$

$$\sum_{i=1 \,(i=i+2)}^{n} 1 = 1+1+1\text{---} \frac{(n+1)}{2}$$
$$= O\left(\frac{n+1}{2}\right)$$

```
for (int i = 1 ; i <= n ; i++)
{
    //O(n)
}
```

$$T.C = \sum_{i=1}^{n} n = n+n+n \text{ ---' } n \text{ times}$$
$$= n(1+1+1\text{---} \ n \text{ times})$$
$$= n*n = O(n^2)$$

```
for (int i = 1 ; i <= n ; i++)
{
    for (int j = 1 ; j <= n ; j++)
```

```
{
    for(int j=i+1; j<=n; j++)
    {
        //O(1)
    }
}
```

$$T.C = \sum_{i=1}^{n}\left(\sum_{j=i+1}^{n} 1\right)$$

$$= \sum_{i=1}^{n} (1+1+1 \ldots (n-i) \text{ times})$$

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

$$= \left(\sum_{i=1}^{n} n - \sum_{i=1}^{n} i\right)$$

$$= n^2 - (1+2+3 \cdots + n)$$

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

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

$$= \frac{2n^2 - n^2 - n}{2} = \left(\frac{n^2-n}{2}\right) = O(n^2)$$

$L \to U$

$1 \to n$ = n times

$(U-L+1)$

$(n-1+1) = n$

$\Rightarrow (n-(i+1)+1)$

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

$= (n-i)$

Recurrence Relation

A recurrence relation is used to determine the relation b/w the Time Complexity of problem & Time complexity of subproblem's soln.

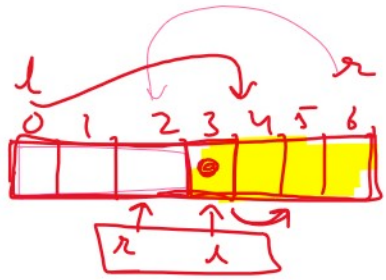
Sorted

T(n)

```
bool binarySearch(int *arr, int l, int r, int key){
    if(l > r) return false;
    int mid = (l+r)/2;
    if(arr[mid] == key) return true;
    else if(arr[mid] < key) return binarySearch(arr, mid+1, r, key);
    else return binarySearch(arr, l, mid-1, key);
}
```

T(n/2)

T(n/2)



l = 0
r = 6

$\frac{l+r}{2}$

$= \frac{0+6}{2} = 3$

$\left(\frac{l+r}{2}\right) = l + \left(\frac{r-l}{2}\right)$

$= \frac{2l+r-l}{2} = \left(\frac{l+r}{2}\right)$

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

$$T(1) = 1$$

Solving Recurrence Relation

```
bool binarySearch(int *arr, int l, int r, int key){

    if(l > r) return false;
    int mid = (l+r)/2;

    if(arr[mid] == key) return true;

    else if(arr[mid] < key) return binarySearch(arr, mid+1, r, key);

    else return binarySearch(arr, l, mid-1, key);
}
```

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

There is no unique method which can solve all the recurrence relation.

1) Forward substitution :-
2) Backward " :-
3) Master's " :-

1) Forward Substitution :-

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

$T(1) = 1$

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

$T(2) = 1 + 2$

$T(3) = T(2) + 3$ $\quad n-1 = 2, \; n = 3$

$= 1+2+3$

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

$= 1+2+3$

$T(4) = T(3) + 4$ $\quad n-1 = 3$, $n = 4$

$1+2+3+4$

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

$= \sum_{i=1}^{n} i = \frac{n*(n+1)}{2}$

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

2) Backward Substitution :-

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

$T(1) = 1$

$T(n) = T(n-1) + n$ —— (1)

$n = n-1$ in eqn (1)

$T(n-1) = T(n-2) + (n-1)$ — (2)

put value of $T(n-1)$ from eqn (2) to eqn (1)

$T(n) = T(n-2) + (n-1) + n$ — (3)

put $n = n-2$ in eqn (1)

$T(n-2) = T(n-3) + (n-2)$ — (4)

put the value of $T(n-2)$ from eqn (4) to eqn (3)

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

$$T(n) = T(n-k)+(n-(k-1))+(n-(k-2))+\cdots+n \quad \text{—— (5)}$$

$$T(1) = 1$$

$$n-k = 1$$

$$k = n-1$$

put value of $k = n-1$ in eqn (5).

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

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

$$= 1+2+3+\cdots+n = \sum_{i=1}^{n} i = \frac{n*(n+1)}{2}$$

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

Example 2

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

$T(1) = 1$.

$T(n) = T(\frac{n}{2}) + 1$ — (1)

put $n = \frac{n}{2}$ in eqn (1)

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

$= T(\frac{n}{4}) + 1$ — (2)

put the value of $T(\frac{n}{2})$ from eqn 2 to eqn (1)

$T(n) = T(\frac{n}{4}) + 1 + 1$

$= T(\frac{n}{4}) + 2$ — (3)

put $n = n/4$ in eqn (1)

$T(\frac{n}{4}) = T(\frac{n}{8}) + 1$ — (4)

put the value of $T(\frac{n}{4})$ from eqn (4) to (3)

$T(n) = T(\frac{n}{8}) + 1 + 2$

$= T(\frac{n}{8}) + 3$ — (5)

⋮

$T(n) = \boxed{T(\frac{n}{2^k})} + k$ — (6)

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

2^k

$2^k = n$

Take $\log_2$ both side

$\log a^b = b \log a$

$\log_a a = 1$

$\log_2 2^k = \log_2 n$

$k \log_2 2 = \log_2 n$

$k = \log_2 n.$

put $k = \log_2 n$ in eqn (6)

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

$a^{\log_a b} = b$

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

$= T(1) + \log_2 n$

$= 1 + \log_2 n$

$= O(\log_2 n)$ —

Example 3

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

$T(1) = 1$

$T(n) = 2T(\frac{n}{2}) + 1$ — (1)

put $n = \frac{n}{2}$ in eqn (1)

$T(\frac{n}{2}) = 2T(\frac{n}{4}) + 1$ — (2)

put value of $T(\frac{n}{2})$ from eqn (2) to eqn (1)

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

$= 4T(\frac{n}{4}) + 2 + 1$ — (3)

put $n = \frac{n}{4}$ in eqn (1)

$T(\frac{n}{4}) = 2T(\frac{n}{8}) + 1$ — (4)

put the value of $T(\frac{n}{4})$ from eqn (4) to (3)

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

$= 8T(\frac{n}{8}) + 4 + 2 + 1$ — (5)

$T(n) = 2^k T(\frac{n}{2^k}) + 2^{k-1} + 2^{k-2} + \cdots + 2 + 1$ — (6)

$T(1) = 1$

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

$2^k = n$

$k = \log_2 n$

put $k = \log_2 n$ in eqn (6)

$a^{b-c} = \frac{a^b}{a^c}$

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

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

$$= n\,T(1) + \frac{2^{\log_2 n}}{2^1} + \frac{2^{\log_2 n}}{2^2} + \cdots + \frac{2^{\log_2 n}}{2^k}$$

$$= \left[n + \frac{n}{2} + \frac{n}{2^2} + \frac{n}{2^3} + \cdots + \frac{n}{2^{\log_2 n}}\right]$$

$0 \to k \to (k+1)$

$$= \left[n\left[1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{n}\right]\right]$$

$a = t_1$ Sum GP $= \frac{a(r^n - 1)}{r - 1}$ if $r > 1$

$r = \frac{t_2}{t_1}$

$r = \frac{t_2}{t_1} = \frac{1}{2}$ → $= \frac{a(1 - r^n)}{1 - r}$ if $r < 1$

$= \frac{a}{1 - r}$ if $r < 1$ & $n \to \infty$

$= \infty$ if $r > 1$ & $n \to \infty$

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

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

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

$$= \cancel{n}\left[\frac{2n-1}{\cancel{n}}\right] = 2n-1 = \underline{\underline{O(n)}}$$

Example 4

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

$T(1) = 1$

$T(n) = T(\frac{n}{2}) + n$ — (1)

put $n = n/2$ in eqn (1)

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

put the value of $T(\frac{n}{2})$ from eqn (2) to eqn (1)

$T(n) = T(\frac{n}{4}) + \frac{n}{2} + n$ — (3)

put $n = \frac{n}{4}$ in eqn (1)

$T(\frac{n}{4}) = T(\frac{n}{8}) + \frac{n}{4}$ — (4)

$T(n) = T(\frac{n}{8}) + \frac{n}{4} + \frac{n}{2} + n$ — (5)

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

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

$2^k = n$

$k = \log_2 n$

$= T(\frac{n}{2^{\log_2 n}}) + \frac{n}{2^{k-1}} + \frac{n}{2^{k-2}} + \cdots + n$

$\cdots + n$

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

$$= \frac{n}{2^k} + \frac{n}{2^{k-1}} + \frac{n}{2^{k-2}} + \cdots + n$$

$$= n + \frac{n}{2} + \frac{n}{2^2} + \cdots + \frac{n}{2^{k-2}} + \frac{n}{2^{k-1}} + \frac{n}{2^k}$$

$$= n\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^k}\right)$$

$$= O(n).$$

$$\left[\begin{array}{l} T(n) = 2T\left(\frac{n}{2}\right) + n \\ T(1) = 1 \end{array}\right] \qquad O(n \log n)$$

Quiz Discussion

1. Consider the following function

```
void DevAnand(int n)
{
    int j=n;

    int i;
    for(i=0;i<n;i++)
    {
            while(j>1)-
            {
                    j>>=1;
            }
    }
}
```

2. What is the complexity of following piece of C code?

```
for(int i = 1; i <=n; i *= 2)
{

    for(int j = 1; j <= n/2; j++)
    {

        //O(j) operation

    }

}
```

3. What is the best case time complexity of below code segment

```
void main( ) {

int j = 0;

for(i=n;i>=1;i=i/2)
{

   for(j=1;j<=i;j=j*2)
   {

       printf("www.csegate.in");

   }
}
```

4. What is the lower bound of f(n) = $n^3 + 12\,nlog(n) + 20\,n$?

5. What is the complexity of the following piece of c code?

```
for (int i = 0; i < n; i++)
{

    for(int j = i; j < n; j++)
    {

        for(int k = j; k < n-i; k++)
        {

            //O(1) operation

        }

    }

}
```

Master's Method

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

$n = 100 \quad b = 20$

$\frac{100}{2} = 50$

$3 \Rightarrow 3, 50$

$c = \log_b a$

n^c & $f(n)$

Case 1 :- $f(n) < n^c$, $T(n) = \theta(n^c)$

Case 2 :- $f(n) = n^c$, $T(n) = \theta(n^c \log n)$.

Case 3 :- $f(n) > n^c$, $T(n) = \theta(f(n))$.

Ex 1

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

$T(n) = a\,T\left(\frac{n}{b}\right) + f(n) \quad [a \geq 1 \text{ & } b > 1]$

$a = 8$

$b = 2$

$c = \log_2 8 = \log_2 2^3 = 3 \log_2 2 = 3$

$n^c = n^3.$

$f(n) = n^2.$

$f(n) = n^2$.

Case 1 :- $f(n) < n^c$ $\quad T(n) = \theta(n^3)$.

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

$T(n) = aT(\frac{n}{b}) + f(n) \quad [a \geq 1 \text{ & } b > 1]$

$a = 2, \; b = 2$

$c = \log_b a = \log_2 2 = 1$

$n^c = n^1 = n$

$f(n) = n$

Case 2 :- $f(n) = n^c \quad T(n) = \theta(n^c \log n)$

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

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

$T(n) = aT(\frac{n}{b}) + f(n) \quad [a \geq 1 \text{ & } b > 1]$

$a = 3, \; b = 2$

$c = \log_b a = \log_2 3 \approx 1.58$

$n^c = n^{1.58}$

$f(n) = n^2$

Case 3 :- $f(n) > n^c$, $T(n) = \underline{\underline{\Theta(n^2)}}$.

Master's Method proof

$T(n) = aT(n/b) + f(n)$, $a \geq 1$ & $b > 1$

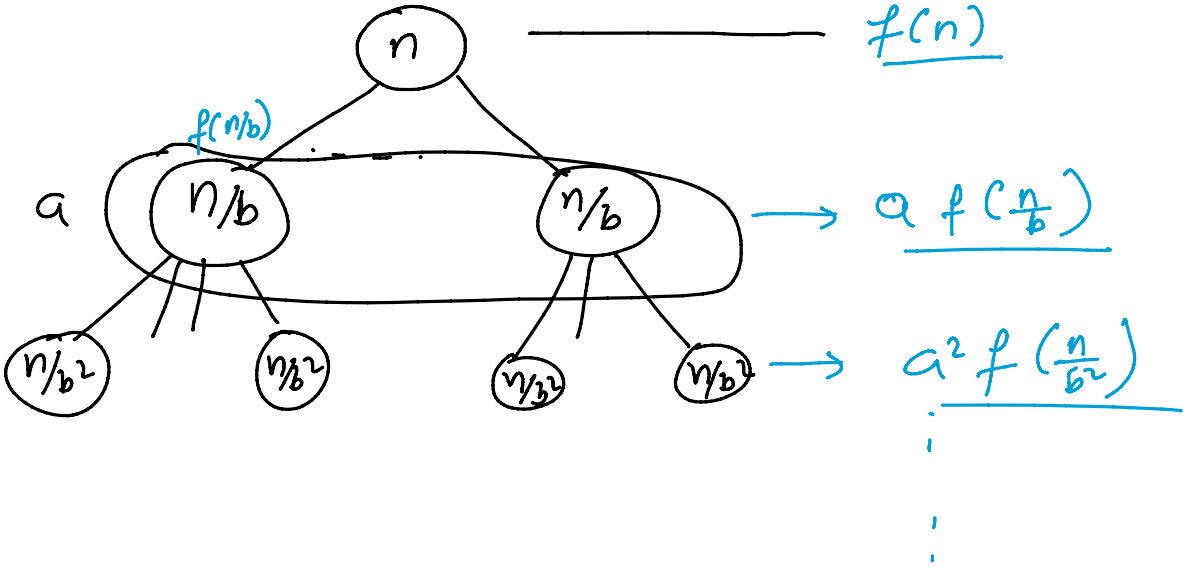
case 1: $f(n) < n^{\log_b a}$, $T(n) = \theta(n^{\log_b a})$

case 2: $f(n) = n^{\log_b a}$, $T(n) = \theta(n^{\log_b a} \log n)$

case 3: $f(n) > n^{\log_b a}$, $T(n) = \theta(f(n))$

regularity condition

$a f(n/b) \leq c f(n)$, for some constant $c < 1$



$T(1) = 1$

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

$b^k = n$

$\log_b b^k = \log_b n$

$k \log_b b = \log_b n$

$k = \log_b n$.

$T(n) = f(n) + a f(\frac{n}{b}) + a^2 f(\frac{n}{b^2}) +$

$a^k f(\frac{n}{\cdot})$

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

$$= \sum_{i=0}^{\log_b n} a^i f\left(\frac{n}{b^i}\right) = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) + n^{\log_b a}$$

(1)

$a^k f\left(\frac{n}{b^k}\right)$

$k = \log_b n$

$= a^{\log_b n} f\left(\frac{n}{b^{\log_b n}}\right)$

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

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

$\log_b n = \frac{\log_a n}{\log_a b}$

$\log_b n = \log_b a \log_a n$

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

$\log_b n \log_a(a) = \log_a(a^{\log_b n})$

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

$\log_k n = \frac{\log_t n}{\log_t k}$

$\log_k s = \frac{1}{\log_s k}$

$k \log x = \log x^k$

$$\Rightarrow \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) + n^{\log_b a} \quad \text{——(2)}$$

Case 1 → $f(n) < n^{\log_b a}$ $\left[f(n) = n^{\log_b a - \epsilon} \right]$

$$= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) + n^{\log_b a}$$

$$= \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon} + n^{\log_b a}$$

$$= n^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n-1} a^i \frac{1}{b^{i(\log_b a - \epsilon)}} + n^{\log_b a}$$

$$= n^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n-1} a^i \frac{1}{b^{\log_b a^i - i\epsilon}} + n^{\log_b a}$$

$$= n^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n-1} a^i \frac{1}{\frac{b^{\log_b a^i}}{b^{i\epsilon}}} + n^{\log_b a}$$

$a^{b-c} = \frac{a^b}{a^c}$

$$= n^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n-1} \cancel{a^i} \frac{b^{i\epsilon}}{\cancel{a^i}} + n^{\log_b a}$$

$a^{\log_a x} = x$

$$= n^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n-1} b^{i\epsilon} + n^{\log_b a}$$

$$= \frac{n^{\log_b a}}{n^{\epsilon}} \sum_{i=0}^{\log_b n-1} b^{i\epsilon} + n^{\log_b a}$$

$$= n^{\log_b a} \left[\frac{1}{n^{\epsilon}} \frac{b^0 \left[\left(b^{\epsilon}\right)^{\log_b n} - 1 \right)}{b^{\epsilon} - 1} \right]$$

$$= n^{\log_b a}\left[\frac{1}{n^{\epsilon}}\cdot\frac{b^0\left[b^{\log_b n^{\epsilon}}-1\right]}{b^{\epsilon}-1}\right]$$

$$= n^{\log_b a}\left[\frac{1}{n^{\epsilon}}\cdot\frac{\left[n^{\epsilon}-1\right]}{b^{\epsilon}-1}\right]$$

$$= \theta\left(n^{\log_b a}\right).$$

Case 2 :- $f(n) = n^{\log_b a}$. $[\epsilon = 0]$

$$= n^{\log_b a - \epsilon}\sum_{i=0}^{\log_b n-1} a^i \frac{b^{i\epsilon}}{a^i} + n^{\log_b a}$$

$$= n^{\log_b a}\sum_{i=0}^{\log_b n-1} 1 + n^{\log_b a}$$

$$\sum_{i=l}^{u} 1 = (u-l+1)$$

$$= n^{\log_b a}\left[\sum_{i=0}^{\log_b n-1} 1 + 1\right]$$

$$= n^{\log_b a}\left[\log_b n - 0 + 1\right]$$

$$= n^{\log_b a}\left[\log_b n + 1\right] = \theta\left(n^{\log_b a}\log_b n\right)$$

$\therefore$ Case 2 :- $f(n) > n^{\log_b a}$ & $af(n/b) \geq cf(n)$

$\Rightarrow$ Case 3 :- $f(n) > n^{\log_b a}$ & $af(n/b) \leq f(n)$

$\hookrightarrow f(n) = n^{\log_b a + \epsilon}$

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

$$= a * \frac{n^{\log_b a + \epsilon}}{b^{\log_b a + \epsilon}}$$

$$= \cancel{a} * \frac{n^{\log_b a + \epsilon}}{\cancel{b^{\log_b a}} * b^{\epsilon}}$$

$$= n^{\log_b a + \epsilon} * b^{-\epsilon}$$

$$= f(n) * b^{-\epsilon}$$

$$a f\left(\frac{n}{b}\right) \leq b^{-\epsilon} * f(n)$$

$$T(n) = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) + n^{\log_b a} \leq$$

$$\sum_{i=0}^{\log_b n - 1} c^i f(n) + n^{\log_b a}$$

$$= f(n) \left(\sum_{i=0}^{\log_b n - 1} \frac{1}{b^{i\epsilon}} \right) + n^{\log_b a}$$

Δ

$$= f(n) \left(\sum_{i=0}^{\log_b n} \frac{1}{b^{i\epsilon}} \right)$$

$$\leq f(n) \sum_{i=0}^{\infty} \frac{1}{b^{i\epsilon}} + n^{\log_b a}$$

$r < 1$, $\frac{1}{b^\epsilon}$

$$= f(n) \frac{1}{1 - b^\epsilon} + n^{\log_b a}$$

$\frac{a}{1-r}$

$$= \frac{n^{\log_b a + \epsilon}}{1 - b^\epsilon} + n^{\log_b a}$$

$$= \frac{n^{\log_b a + \epsilon} + n^{\log_b a} - b^\epsilon n^{\log_b a}}{1 - b^\epsilon}$$

$$\frac{1}{b^0} + \frac{1}{b^\epsilon} + \frac{1}{b^{2\epsilon}} \cdots$$

$a = \frac{1}{b^0} = 1$, $r = \frac{\frac{1}{b^\epsilon}}{\frac{1}{b^0}} = \frac{1}{b^\epsilon}$

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

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

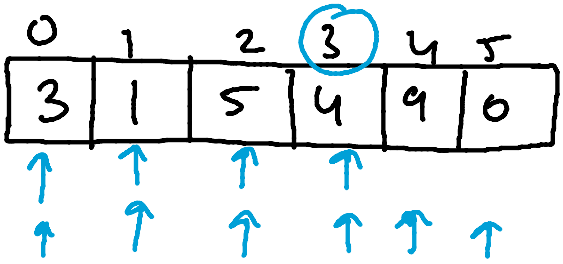
$$= T(n) = 3T\left(\frac{n}{1}\right) + 1$$

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

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

Linear Search

```
int LinearSearch(int *arr, int n, int key)
    for i<- 0 to n - 1
        if arr[i] = key
            return i
    return -1
```



n

Key = 4

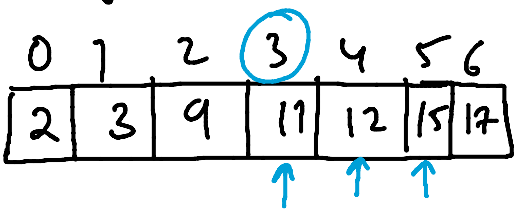
10

- Best Case = O(1)
- Average " = O(n)
- Worst " = O(n)

Iterative Binary Search

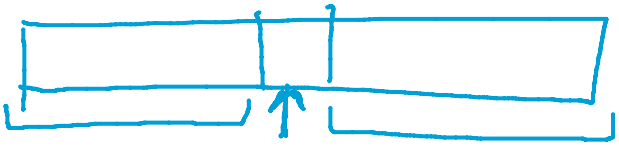
```
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r) {
        int m <- (l + r) / 2;

        if (arr[m] = x)
            return m;

        if (arr[m] < x)
            l <- m + 1;

        else
            r <- m - 1;
    }

    return -1;
}
```

Key = 13



$l = 0$ $\quad \frac{0+6}{2} = 3$

$r = n-1$



- Best case : $O(1)$
- Average " : $O(\log_2 n)$
- Worst " : $O(\log_2 n)$

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

$n/2 \rightarrow 1$

$n/2^2 \rightarrow 1$

$n/2^k \rightarrow 1$

$(k+1)$

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

$2^k = n$

$k = \log_2 n$

$O(k+1)$

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

Recursive Binary Search

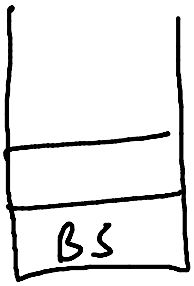
```
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid <- (l + r) / 2;

        if (arr[mid] = x)
            return mid;

        else if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        else
            return binarySearch(arr, mid + 1, r, x);
    }

    return -1;
}
```

$\leftarrow T(n)$

1

$\leftarrow T(n/2)$

$\leftarrow T(n/2)$



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

Best case :- $O(1)$

Average " :- $O(\log_2 n)$

Worst " :- $O(\log_2 n)$

Space complexity

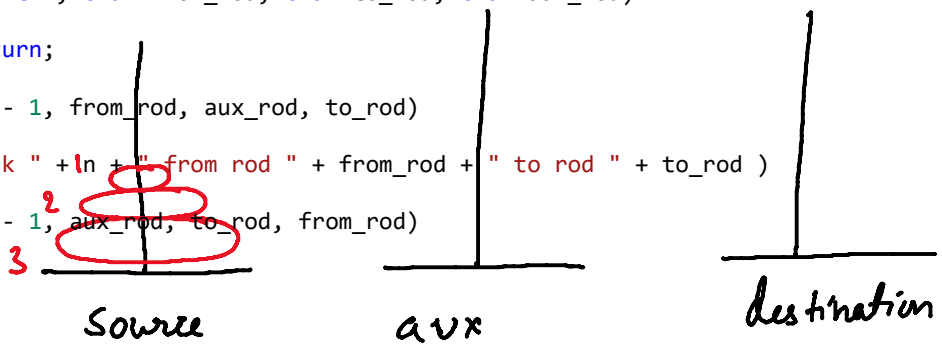
Best case - $O(1)$

Average " - $O(\log_2 n)$

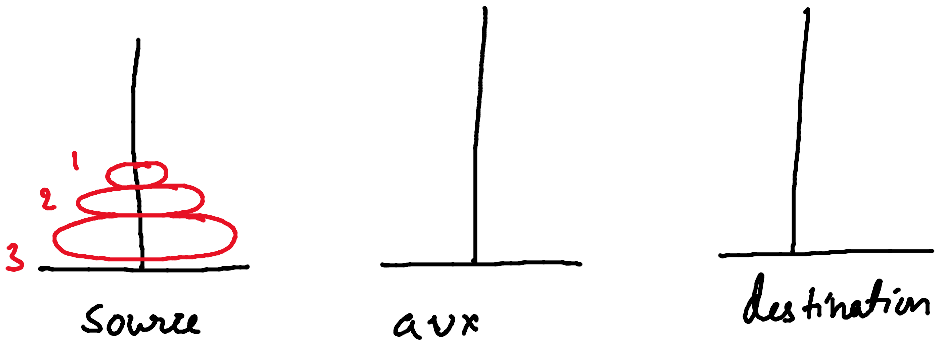
Worst " - $O(\log_2 n)$ $\leftarrow$

Tower Of Hanoi

```
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
    if (n = 0)  return;

    towerOfHanoi(n - 1, from_rod, aux_rod, to_rod)

    Print("Move disk " + n + " from rod " + from_rod + " to rod " + to_rod )

    towerOfHanoi(n - 1, aux_rod, to_rod, from_rod)
}
```



1) Only one disc can move at a time.
2) At any point of time the larger disc should not be above smaller disc.



1. 1 from 's' to 'd'
2. 2 " 's' to 'a'
3. 1 " 'd' to 'a'
4. 3 " 's' to 'd'
5. 1 " 'a' to 's'
6. 2 " 'a' to 'd'
7. 1 " 's' to 'd'

$4 = 2^4 - 1$

$n = 2^n - 1$

towerOfHanoi(3, 's', 'd', 'a')

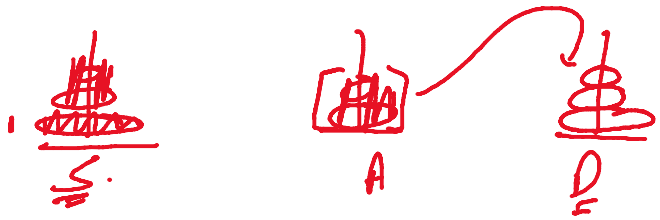
'S' 'D' 'A' T(n)

```
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
    if (n == 0) return;
    towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);
    cout << "Move disk " << n << " from rod " << from_rod <<" to rod " << to_rod << endl;
    towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);
}
```

← T(n-1)

← O(1)

← T(n-1)



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

$$\Theta(2^n).$$

Fibbonacci Series

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

0 1 2 3 4 5

0 1 1 2 3 5 8 ...

Recursive , iterative

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

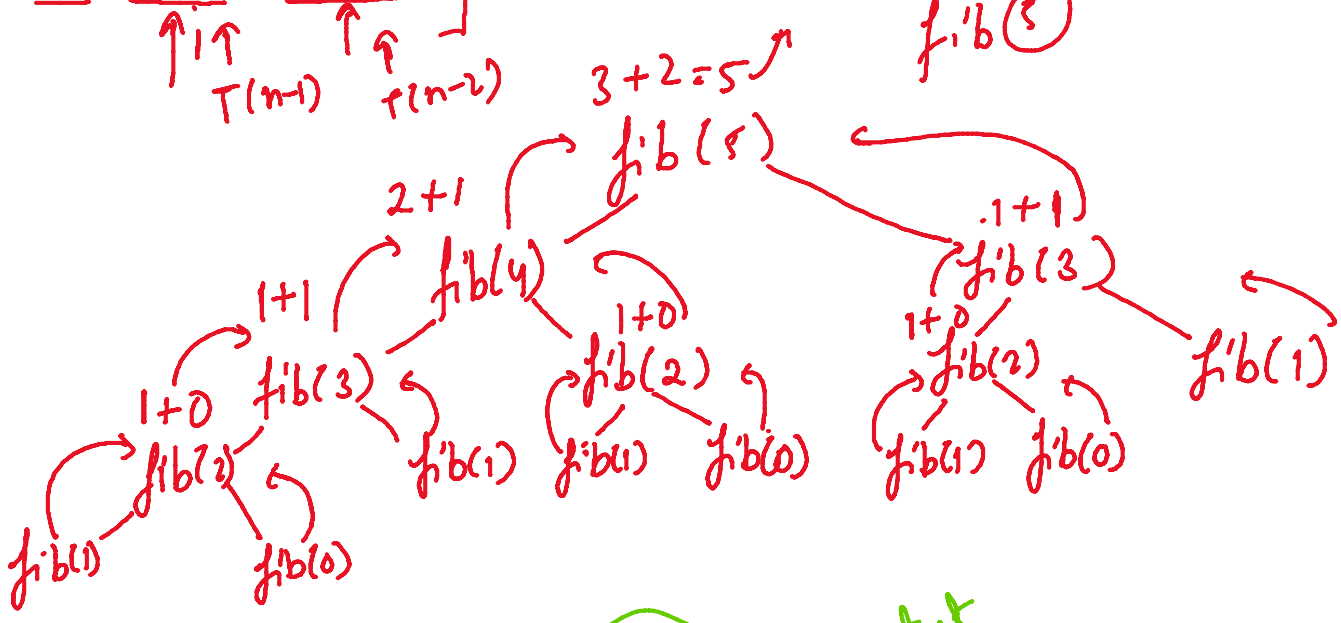
```
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

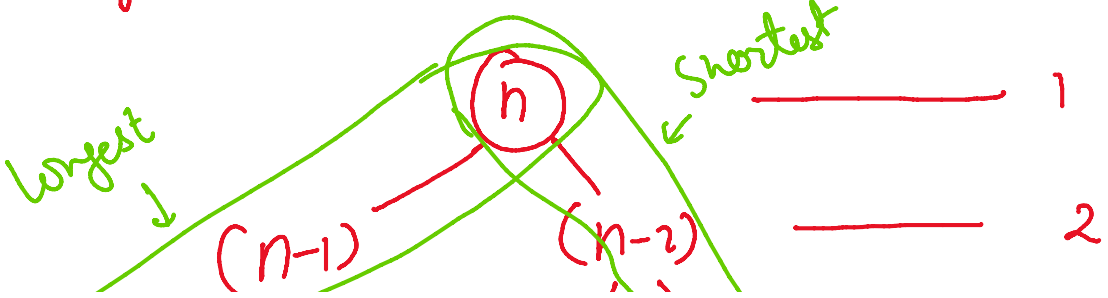
T(n)

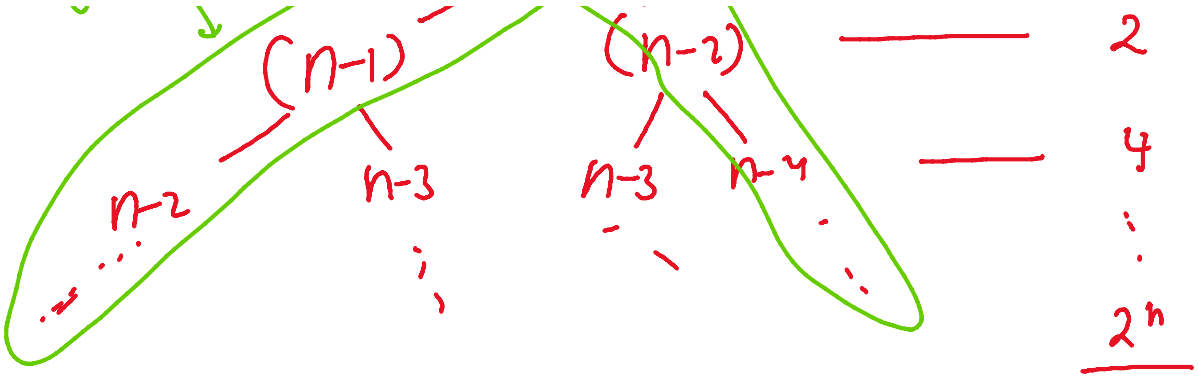
T(n-1) T(n-2)

n = 0 , 0

n = 1 , 1







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

$$a = 1, \quad r = \frac{2}{1} = 2$$

$$\frac{a(r^{term} - 1)}{r - 1} = \frac{1(2^{n+1} - 1)}{2 - 1} = 2^{n+1} - 1$$

$$O(2^{n+1}) = O(2 * 2^n) = O(2^n).$$

```
int fib(int n)
{
    int a <- 0, b <- 1, c

    if( n = 0)
        return a

    for i <- 2 to n
    [   c <- a + b
        a <- b
        b <- c
    return b
}
```

if(n=1) return b;

$\theta(n)$.

$c = 0 + 1 = 1$
$a = b = 1$

$c = 1 + 1 = 2$
$a = b = 1$
$b = c = 2$

```
}
```

$a = b = 1$

$b = c = 1$

$a = b = 1$

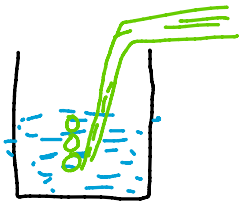
$b = c = 2$

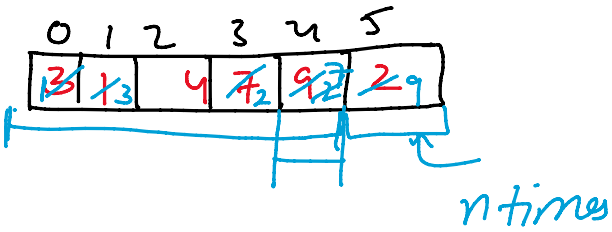
Sorting is a technique to arrange elements of an array in certain order.

- Ascending
- descending

1. Bubble sort
2. Selection ,,
3. Insertion ,,
4. Merge ,,
5. Quick ,,
6. Heap ,,

Bubble Sort

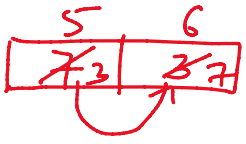





```
void bubbleSort(int arr[], int n)
{
    int i, j;
    for i <- 0 to n-1
        for j <- 0 to n-i-1
            if (arr[j] > arr[j+1])
                swap(arr[j], arr[j+1]);
}
```

$i = 0$ $\quad n-1 \quad (n-i-1) = (n-0-1)$

$i = 1$ $\quad n-2 \quad (n-1-1) = (n-2)$



$$\sum_{i=l}^{u} 1 = u - l + 1$$

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

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

$$T.C = \sum_{i=0}^{n-1} \left(\sum_{j=0}^{n-i-1} 1 \right)$$

$$= \sum_{i=0}^{n-1} (n-i-1-0+1)$$

$$= \sum_{i=0}^{n-1} (n-i)$$

$$= (n-0) + (n-1) + (n-2) + \cdots + (n-(n-1))$$

$$= n + (n-1) + (n-2) + \cdots + 1$$

$$= \frac{n*(n+1)}{2} = O(n^2).$$

- worst

$= \frac{n*(n+1)}{2} = O(n^2)$.

Time Complexity

- Best case :- $O(n^2)$
- Average " :- $O(n^2)$
- Worst " :- $O(n^2)$

Number of Swaps :-

- Best case = 0
- Average " = $\frac{n^2}{}$
- Worst " = $\frac{n*(n-1)}{2}$

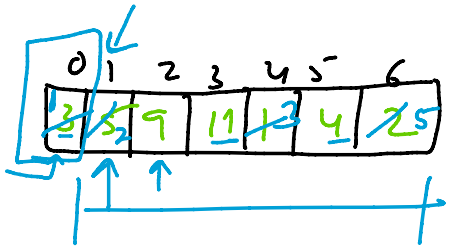
Number of Comparisons

- Best case :- $O(n^2)$
- Average case :- $O(n^2)$
- Worst case :- $O(n^2)$

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

Space Complexity → $O(1)$.

Selection Sort



min_idx = 4 6

```
void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    for i <- 0 to n-1
    {
        min_idx <- i;

        for j <- i+1 to n
            if (arr[j] < arr[min_idx])
                min_idx <- j;

        swap(arr[min_idx], arr[i]);
    }
}
```

Time Complexity.

Best case = $O(n^2)$

Average = $O(n^2)$

Worst = $O(n^2)$

Number of swaps

Best case :- n times

Average :- n

$$T.C = \sum_{i=0}^{n-1} \left(\sum_{j=i+1}^{n} 1 \right)$$

$$= \sum_{i=0}^{n-1} \left(n-(i+1)+1 \right)$$

$$= \sum_{i=0}^{n-1} \left(n-i \right)$$

$$= n + (n-1) + (n-2) + \dots + n-(n-1)$$

$$= n + (n-1) + (n-2) + \dots + 1$$

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

Average :- n

Worst :- n

Number of Comparisons

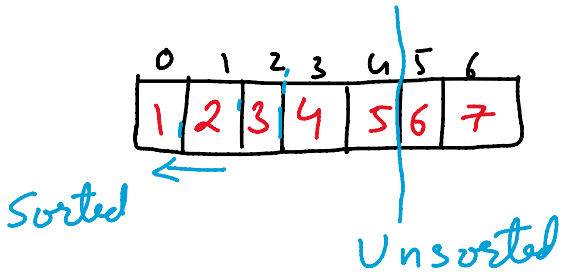
Best Case - $\frac{n*(n+1)}{2}$

Average - $\frac{n*(n+1)}{2}$

Worst - $\frac{n*(n+1)}{2}$

Space Complexity - $O(1)$.

Insertion sort



temp = 4

```
void insertionSort(int arr[], int n)
{
    int i, temp, j;
    for i <- 1 to n
    {
        temp <- arr[i];
        j <- i - 1;
        while (j >= 0 AND arr[j] > temp)
        {
            arr[j + 1] <- arr[j];
            j <- j - 1;
        }
        arr[j + 1] <- temp;
    }
}
```

S.C = O(1)

Time Complexity

Best case = O(n)

Average " = $O(n^2)$

Worst " = $O(n^2)$

1, 2, 3, 4, 5, 6, 7

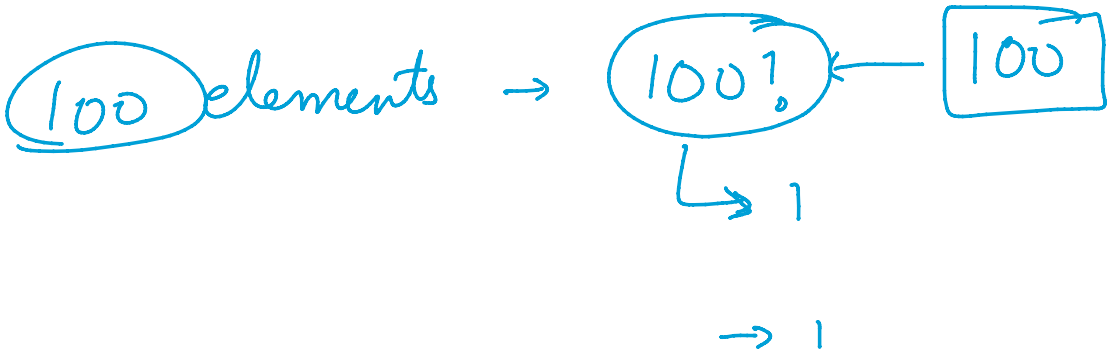
5 | 4, 3, 2, 1

4, 5, 3, 2, 1

3, 4, 5, 2, 1

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

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



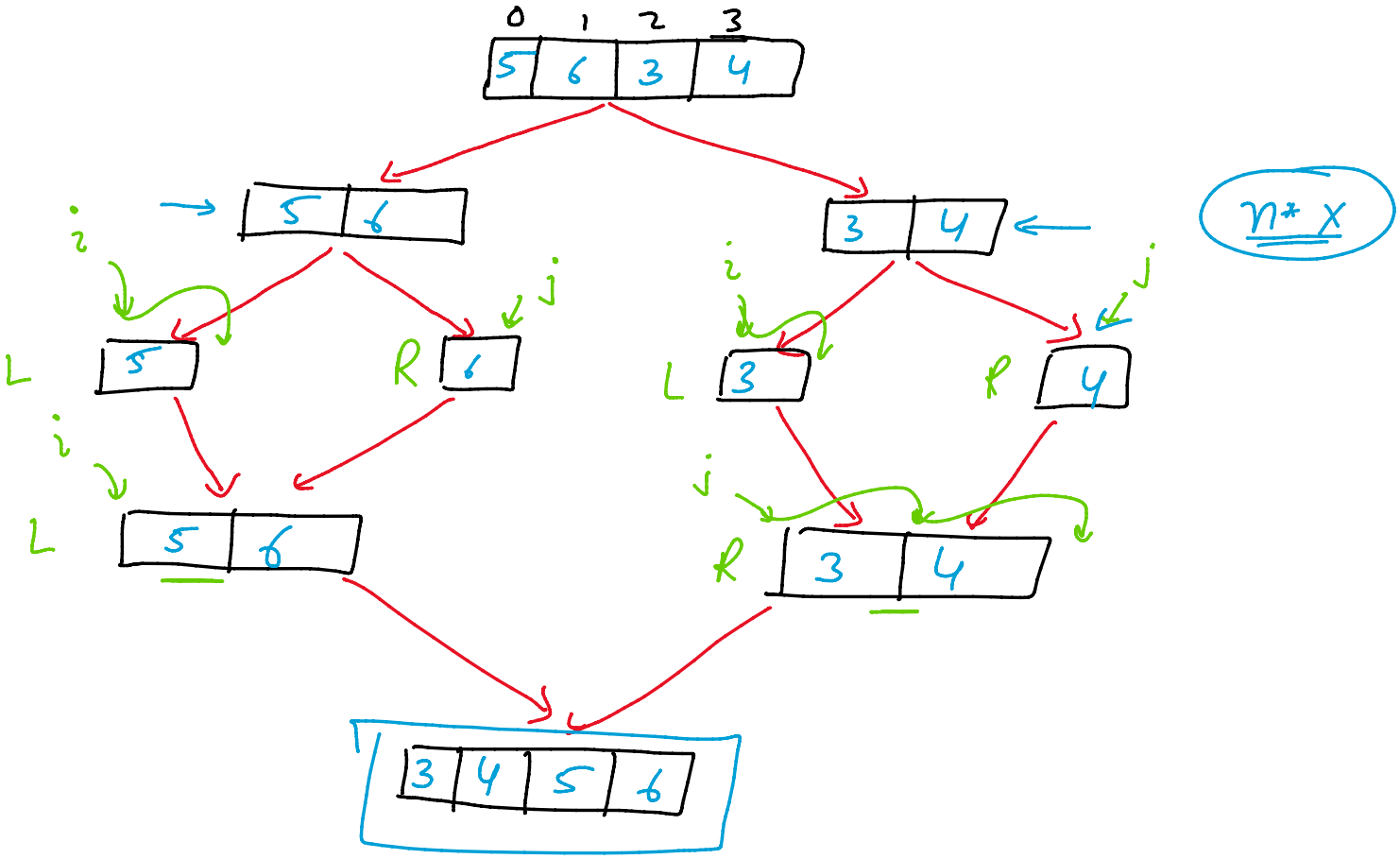
Which sorting algorithm would you consider if array is almost sorted?

1000 → 1→2

A Insertion sort

Divide & Conquer

1. Keep dividing the problem into subproblems until the solution of subproblems is trivial
2. Solve the subproblems.
3. Merge the subsolutions to make the solution of main problem.



T(n)

```
void mergeSort(int arr[],int l,int r)
```

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

T(n)

T(n/2)

T(n/2)

O(n)

```
void merge(int arr[], int l, int m, int r)
{
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];

    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;

    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
            arr[k++] = L[i++];

        else
            arr[k++] = R[j++];
    }

    while (i < n1)
        arr[k++] = L[i++];

    while (j < n2)
        arr[k++] = R[j++];
}
```

n/2

n/2

n

l = 0 , r = n-1

l = r

log n

ms(2,3)

ms(0,3)

l = 0, r = 3

m = (0+1)/2 = 0

(0, 0, 1)
(l, m, r)

0 → 0

1 → 1

l → m

| 1 | 3 | 4 | 7 |

i

j

L: | 3 | 4 |

R: | 1 | 7 |

l → u ⇒ u - l + 1

0 → 10 = 10 - 0 + 1 = 11

m+1 → r

⇒ r - (m+1) + 1

= r - m - 1 + 1 = r - m

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

$$T(1) = 1$$

$a = 2, b = 2$, $\log_b a = \log_2 2 = 1$

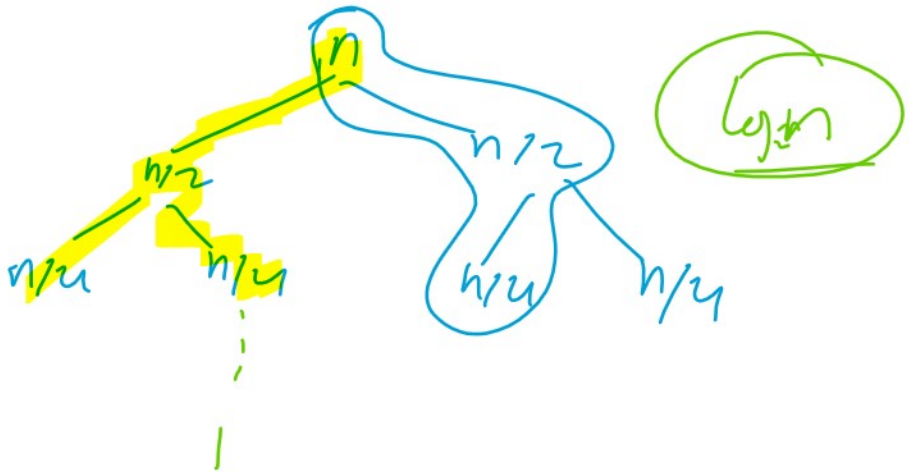
$f(n) = n$

$n^{\log_a a} = n^1 = n$

$T.C = \Theta(n^1 \log_2 n) = \Theta(n \log n)$

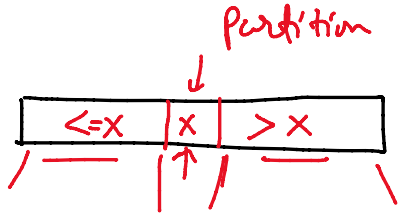
Best case = Average case = worst case

Space complexity = $(\log_2 n + n) = O(n)$

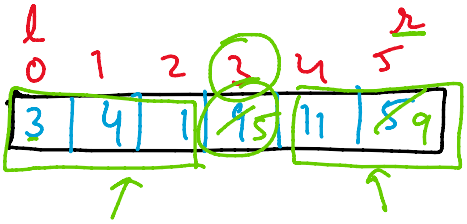


Quick Sort

Divide & Conquer sorting method



Pivot



pivot = 5

i = -1 0 1 2 3

j = 0 1 2 3 4

```
void quickSort(int arr[], int l, int r)
{
    if (l < r)
    {
        int p <- partition(arr, l, r);
        quickSort(arr, l, p - 1);
        quickSort(arr, p + 1, r);
    }
}
```

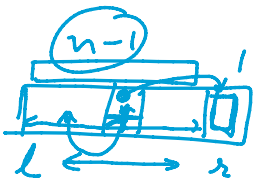
$T(n)$ ← $T(n)$

n ← n

$T(n/2)$ ← $T(n/2)$

$T(n/2)$ ← $T(n/2)$

Sorted
Reverse sorted



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

$$T(1) = 1$$

```
int partition (int arr[], int l, int r)
{
    int x = rand()%(r-l+1) + l;
    swap(&arr[x], &arr[r]);
    int pivot <- arr[r];
    int i <- (l - 1);

    for (int j = l; j <= r - 1; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[r]);
    return (i + 1);
}
```

randomized quick sort

T.C = $O(n \log_2 n)$

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

$T(1) = 1$

$O(n^2)$

T.C

Best case → $O(n \log_2 n)$

Average " → $O(n \log_2 n)$

Worst case → $O(n^2)$

100 → 100!

l ↔ r

$(X \% 100) + 10$ → 0 − 99

10 → 109

rand() % (r - l + 1) + l

(0 → r - l)

l → r

$T(n) = T\left(\frac{99}{100}n\right) + T\left(\frac{n}{100}\right) + n$

$T(1) = 1$

$O(n \log_{\frac{100}{99}} n)$

$O\left(\frac{n \log_2 n}{\log_2 \frac{100}{99}}\right) \leftarrow O(n \log_2 n)$

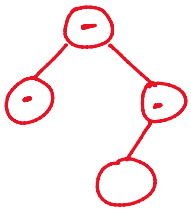
$n/2$

Space complexity $= O(\log_2 n)$

$\hookrightarrow O(n)$

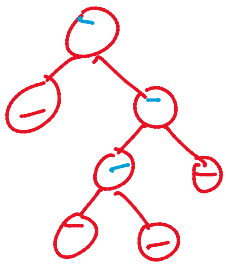
Heap

1) Binary tree :- 0, 1, 2



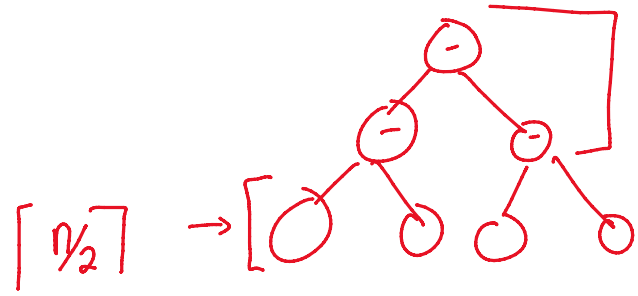
a) Strict B-T :-

0, 2



b) Perfect B-T :-

A perfect B-T is a tree in which all levels are completely filled



$\lceil n/2 \rceil = \lceil \frac{7}{2} \rceil = \lceil 3.5 \rceil = 3$

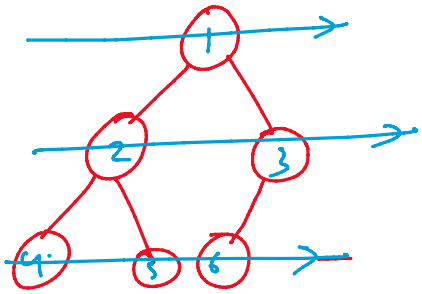
$\lceil n/2 \rceil \rightarrow$

$\lceil \frac{7}{2} \rceil = \lceil 3.5 \rceil = 4$

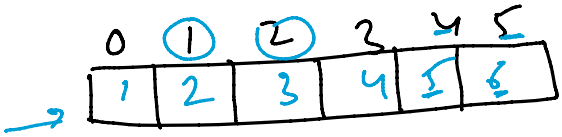
c) Complete Binary tree :-

A CBT is a tree in which all levels are completely filled except probably the last level and the nodes in the last level are as left as possible.

last level are as left as possible.



LC = 2*1+1 = 3
RC = 2*1+2 = 4



A node at index i

Left child = 2*i+1
Right " = 2*i+2

$\left\lfloor \frac{5-1}{2} \right\rfloor = \left\lfloor \frac{4}{2} \right\rfloor = [2] = 2$

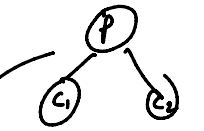
$\left\lfloor \frac{4-1}{2} \right\rfloor = \left\lfloor \frac{3}{2} \right\rfloor = [1.5] = 1$

Child at index i

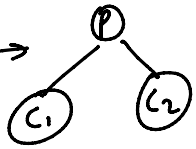
Parent = $\left\lfloor \frac{i-1}{2} \right\rfloor$

Heaps are C.B-T

↳ Binary heaps
↳ Min heap
↳ Max "



P < C1 AND P < C2

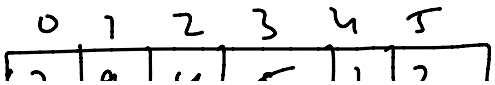


P > C1 AND P > C2

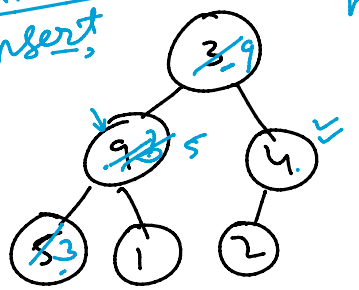
Any array to heap (Min, Max) in O(n)

extract-max

Max heap



extract-max
insert



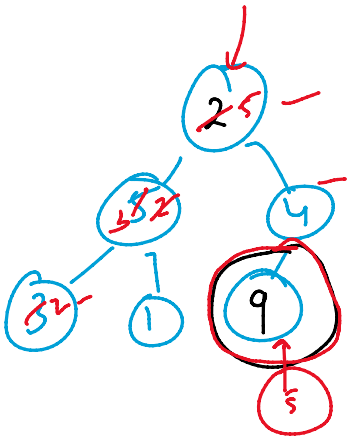
MAX heap

0	1	2	3	4	5
3	9	4	5	1	2

9	5	4	3	1	2

```
Void buildheap (int *arr, int n)
{
    for( int i = n/2 ; i >= 0 ; i--)
    {
        Heapify ( arr, n, i) ;
    }
}

Void Heapify ( int *arr, int n, int i)
{
    int c1 = 2*i+1, c2 = 2*i+2;
    int maximum = i;
    if ( (c1 < n && arr[c1] > arr[maximum])
    {
        maximum = c1;
    }
    if( c2 < n && arr[c2] > arr[maximum])
        maximum = c2;

    if ( maximum != i)
```

```
if ( maximum != i)
{
    swap ( arr [i], arr[maximum]);
    hapify ( arr , n , maximum);
}
}
```

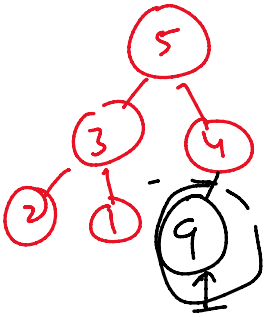


Extract max

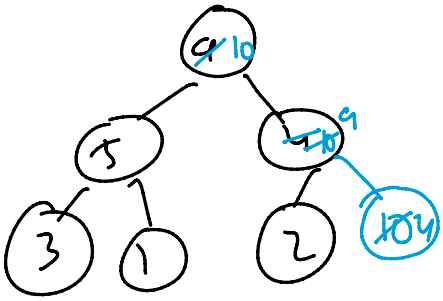
n--

n = 6

n = 5 0 → 4



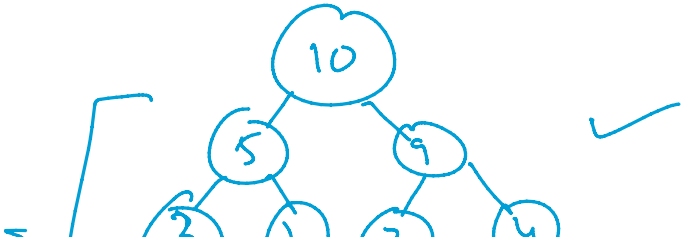
arr[0]

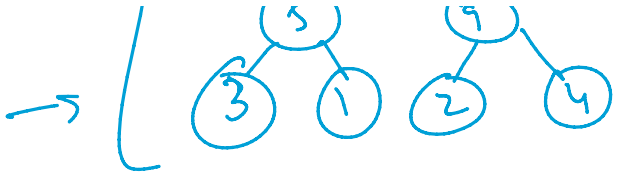


10

percolate - up

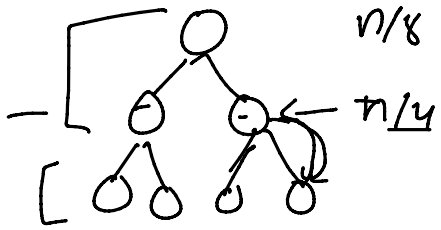
" - down





Time complexity of building a heap from array

$a^{\log_a b} = b$



A.G.P $\quad r = 1/2$

$$S = \left[\frac{n}{4} * 1 + \frac{n}{8} * 2 + \frac{n}{16} * 3 + \cdots + \frac{n}{2^{\log_2 n}}(\log_2 n - 1)\right]$$

$$\frac{S}{2} = \quad + \frac{n}{8} + \frac{n}{16} \times 2 + \frac{n}{32} \times 3 + \cdots + \frac{n}{2^{\log_2 n}}(\log_2 n - 2) + \frac{n}{2^{\log_2 n}}(\log_2 n - 1)$$

$$\frac{S}{2} = \left(\frac{n}{4} + \frac{n}{8} + \frac{n}{16} + \frac{n}{32} \cdots \frac{n}{2^{\log_2 n}}\right) - \frac{(\log_2 n - 1)}{2}$$

$$S = \left(\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + \frac{n}{2^{\log_2 n - 1}}\right) - \log_2 n + 1$$

$$= n\left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^{\log_2 n - 1}}\right) - \log_2 n + 1 \quad \text{— (1)}$$

$$\left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots \frac{1}{2^{\log_2 n - 1}}\right)$$

$$a = \frac{1}{2}, \quad r = \frac{1}{2}$$

$$\frac{a(1 - r^{\#\#})}{1 - r}$$

$\cdot \quad \cdot \wedge \log_2 n - 1)$

$$\frac{1 \cdot \left(1 - \left(\frac{1}{2}\right)^{\log_2 n - 1}\right)}{1 - 1/2}$$

$$1 - \frac{1}{2^{\log_2 n - 1}}$$

$$1 - \frac{2}{2^{\log_2 n}}$$

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

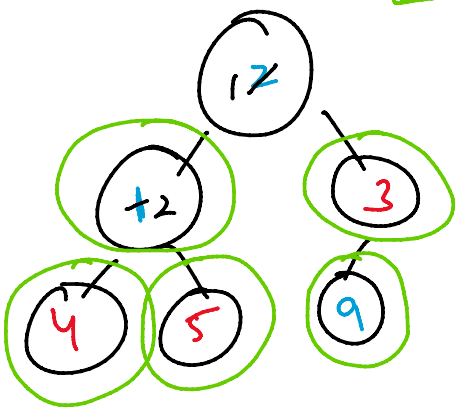
$$= n\left(\frac{n-2}{n}\right) - \log_2 n + 1$$

$$= (n - 2 - \log_2 n + 1) = (n - \log_2 n - 1)$$

$$= O(n) \leftarrow$$

Heap Sort

Cap = n = 6



0	1	2	3	4	5
9	5	4	3	1	2

0	1	2	3	4	5
1	2	3	4	5	9

n = ~~6~~ ~~5~~ ~~4~~ ~~3~~ ~~2~~ 1

n = Cap

1, 2, 3, 4, 5, 9

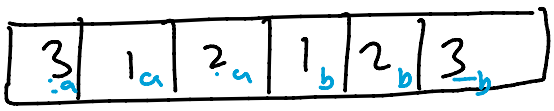
$O(n + n \log n)$

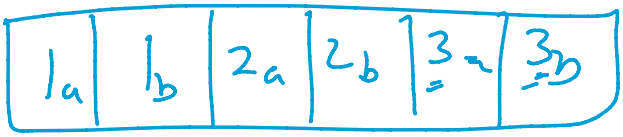
$= O(n \log n)$

1) in place sorting :-

Space Complexity (O(1))

2) Stable Sorting :-





The relative order of equal element will not change after sorting.

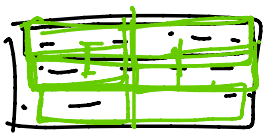
3) Online Sorting :-

insertion sort

4) External Sorting

RAM = 2 GB

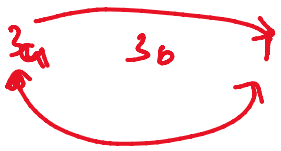
File = 10 GB



Comparison of Sorting algorithm

T.C

	Best Case	Average Case	Worst Case	Space complexity Worst Case	Stable	Inplace
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	✓	✓
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	✗	✓
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	✓	✓
Merge Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	✓	✗
Quick Sort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(n)$	✗	✗
Heap Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	✗	✓



Topological Sorting

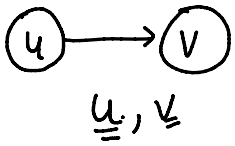
Q. What is topological sort?

A. It is linear ordering of vertices such that for every directed edge(u,v), vertex `u` comes before vertex `v` in the ordering.

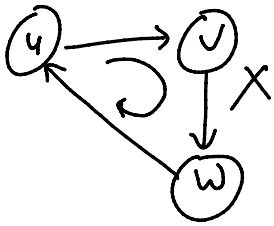
no cycle

It can only be applied on DAG (Directed acyclic Graph).

edges are directed



u, v

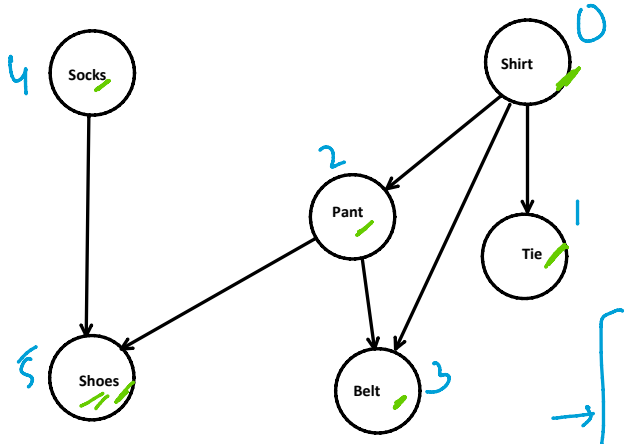


$u \rightarrow v$

$v \not\rightarrow u$

4, 0, 3, 5, 2, 1 ✓

Example



Possible ordering

1. Socks, shirt, tie, pant, shoes, Belt.
2. Shirt, Pant, tie, Belt, socks, shoes.
3. Socks, shirt, Pant, shoes, Belt, tie

etc...

Socks, shoes, shirt, pant, belt, tie.

In DFS we print a vertex and then recursively call dfs on its adjacent vertices.

In topological sorting we print vertex before it's adjacent vertex

Code

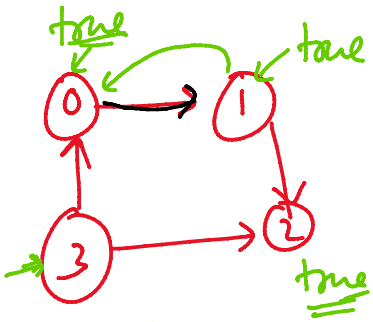
```cpp
#include <iostream>
#include <vector>
#include <stack>
using namespace std;
void topologicalSortUtils(vector<int> graph[], bool visited[], stack<int> &st, int i)
{
    visited[i] = true;
    int n = graph[i].size();
    for (int j = 0; j < n; j++)
    {
        if(!visited[graph[i][j]])
            topologicalSortUtils(graph, visited, st, graph[i][j]);
    }
    st.push(i);

}
void topologicalSort(vector<int> graph[], int V){
    bool visited [V];
    stack <int> st;
    for (int i = 0; i < V; i++)
        visited[i] = false;

    for (int i = 0; i < V; i++)
    {
        if(!visited[i])
            topologicalSortUtils(graph, visited, st, i);
    }
    while(!st.empty())
    {
        cout<<st.top()<<" ";
        st.pop();
    }
}
int main(){
    int V, E, s, d;
    cin>>V>>E;
    vector<int> graph[V];
    for (int i = 0; i < E; i++){
        cin>>s>>d;
        graph[s].push_back(d);
    }
    topologicalSort(graph, V);
}
```

E

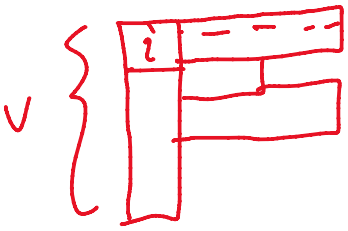
V

T.C = O(V+E)

S.C = O(V)



3, 0, 1, 2



=> What are disjoint sets & operations on them

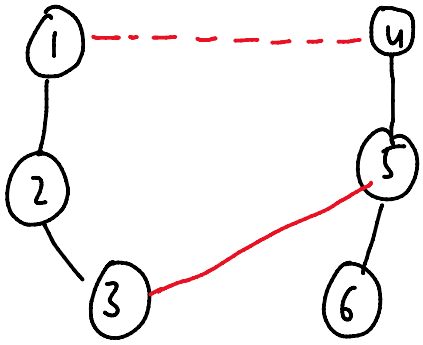
Similar to sets in mathematics.

X $S_1 = \{1, 2, 3\}$
$S_2 = \{4, 5, 6\}$

$S_1 \cup S_2$

$S_1 \cap S_2 = \phi$

$S_3 = \{1, 2, 3, 4, 5, 6\}$

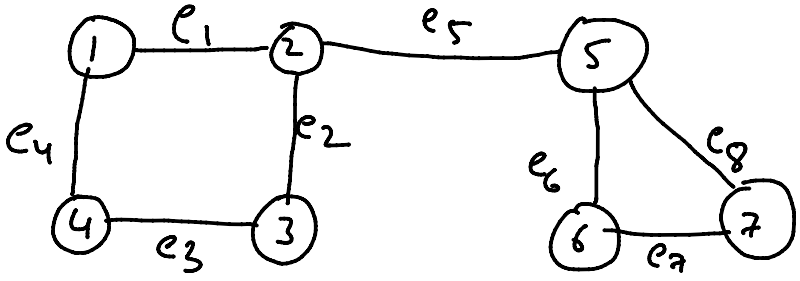


Find → Tells the set to which an element belong

Find (1) = S1
Find (5) = S2

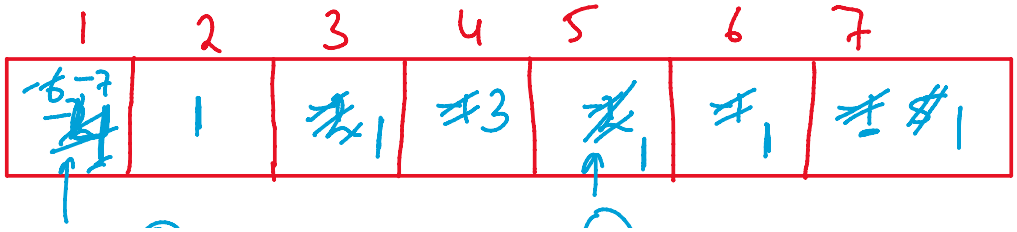
Union → Merge two sets when an edge is added.

Ex

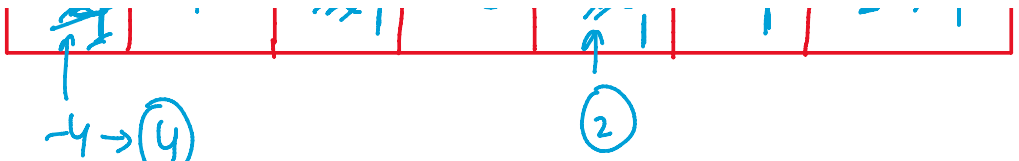


$-n \rightarrow n$

Parent.

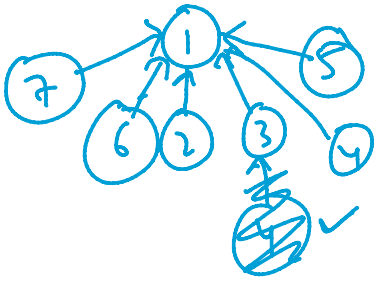


parent



-4 → 4

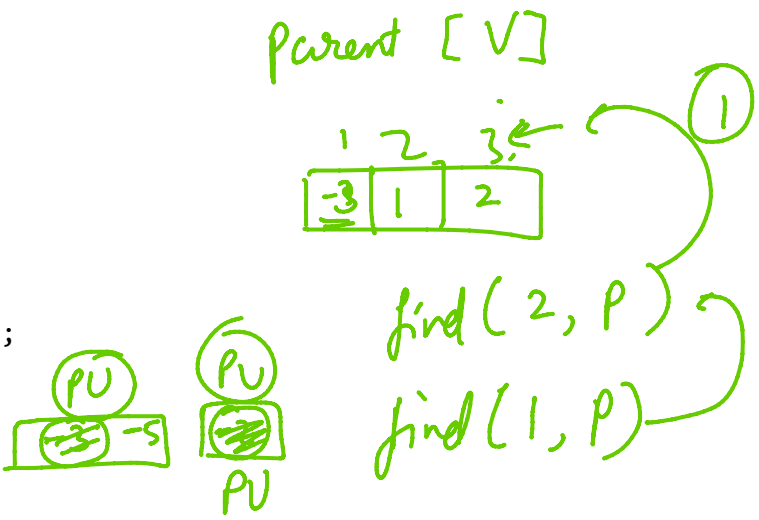
2



Collapsing find

Code

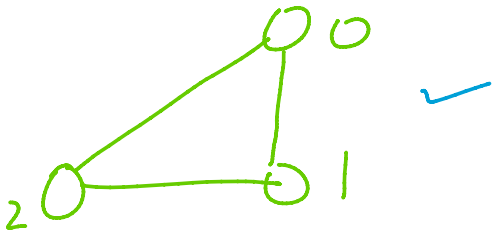
```
int find(int u, int *parent)
{
    if(parent[u] < 0) return u;
    return find(parent[u], parent);
}


void unionByWeight(int u, int v, int *parent)
{
    int pu = find(u, parent), pv = find(v, parent);

    if(pu != pv)
    {
        if(parent[pu] < parent[pv])
        {
            parent[pu] += parent[pv];
            parent[pv] = pu;
        }
        else {
            parent[pv] += parent[pu];
            parent[pu] = pv;
        }
    }
}
```

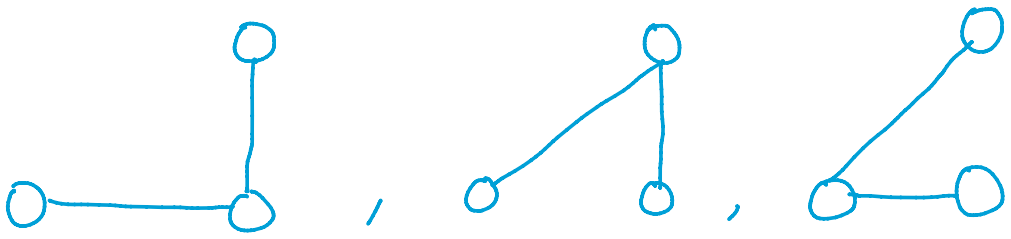


1. Spanning Tree :-

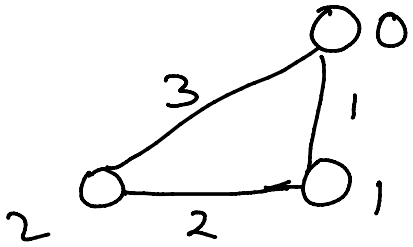
A spanning tree is a tree which spans all the vertices of a graph.

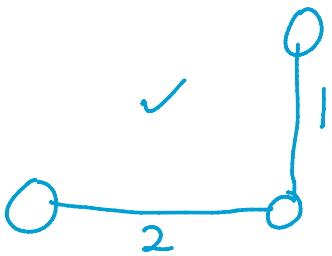
$|V| \Rightarrow (|V|-1)$ edges



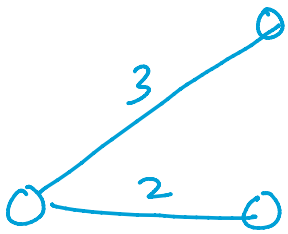


2. Minimum spanning tree :-

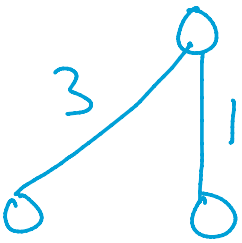




3



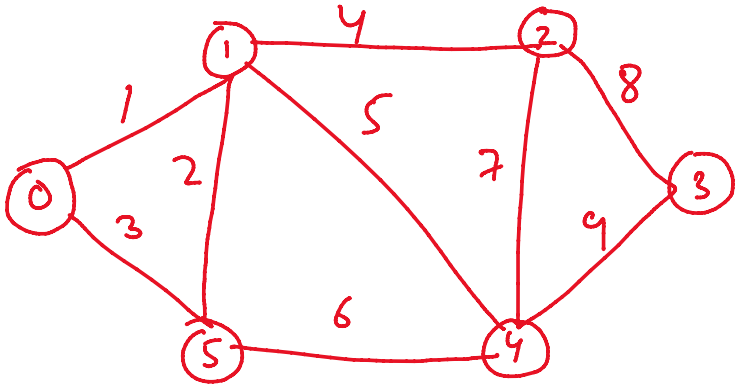
5



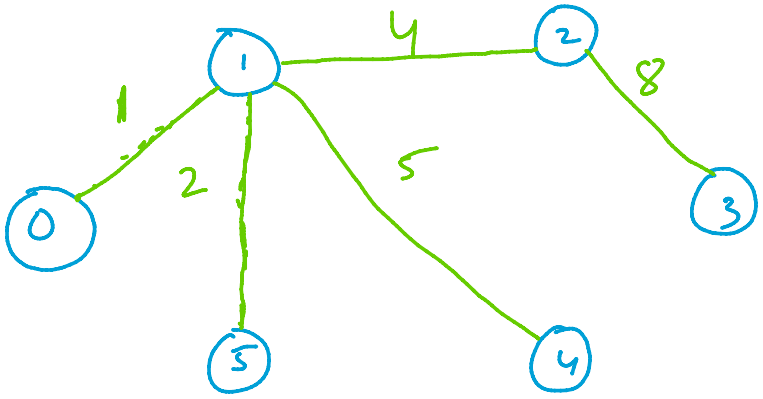
4

Kruskal's → prim's

Greedy approach.



U	V	W	
0	1	1	✓
1	5	2	✓
0	5	3	X
1	2	4	✓
1	4	5	✓
4	5	6	X
2	4	7	X
2	3	8	✓
3	4	9	X



1+2+4+5+8 = 20

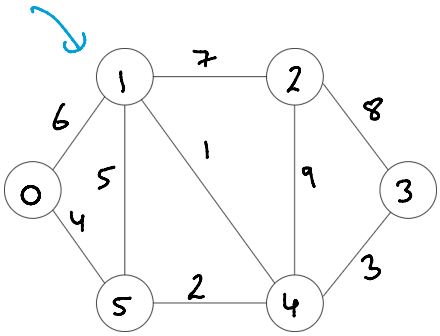
Disjoint set data structure.

Code

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
typedef vector<pair<int, pair<int, int> > > viii;

viii kruskals(viii g, int V)
{
    viii res;
    int s, d, w;
    int parent[V];
    for (int  i = 0; i < V; i++)
        parent[i] = -1;

    sort(g.begin(), g.end());

    int E = g.size();
    for (int i = 0; i < E; i++)
    {
        s = g[i].second.first;
        d = g[i].second.second;
        w = g[i].first;
        if(find(s, parent) != find(d, parent))
        {
            res.push_back(g[i]);
            unionByWeight(s, d, parent);
        }
    }
    return res;
}

int main()
{
    viii g, res;
    int V, E, s, d, w;
    cin>>V>>E;
    for(int i = 0; i < E; i++)
    {
        cin>>s>>d>>w;
        g.push_back(make_pair(w, make_pair(s,d)));
    }
    res = kruskals(g, V);
    int sum = 0 ;
    for (int i = 0; i < res.size(); i++)
    {
        int w = res[i].first;
        cout<<res[i].second.first<<" "<<res[i].second.second<<endl;
        sum += w;
    }
    cout<<"sum of weights = "<<sum<<endl;

}
```

Prim's Algorithm

Greedy Algo → MST

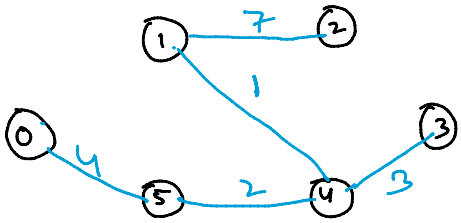


Weight

0	1	2	3	4	5
~~∞~~ 0	~~∞~~ 6	∞	∞	∞	~~∞~~ 4
	5	∞	∞	2	
	1	9	3		
		7	3		
		7			

Parent

0	1	2	3	4	5
-1	~~1~~ 0	~~-1~~ 4	~~-1~~ 4	~~-1~~ 5	~~-1~~ 0



1+2+3+4+7 = 17

0	1	2	3	4	5
-1	4	1	4	5	0

```
MST_PRIM(G, src)
{
    for each vertex u bt g.vertices
    {
        u.key = inf
        u.parent = NIL
    }
    src.key = 0
    Q = minheap(g.vertices)
    while(Q != phi)
    {
        u = extract_min(Q)
        for each vertex 'v' adjacent to u
        {
            if v bt Q and w(u,v) < v.key
            {
                v.parent = u
                v.key = w(u,v)
            }
        }

    }
}
```

|V|

O(|V|)

|V|

O(lg|V|)

2E

O(lg|V|)

u ⟶ v

|E| >= |V| - 1

|E| = |V|*|V|

= O(|V| lg|V| + 2|E| lg|V| + |V|)

T.C = O(|E| lg|V|) ⇒

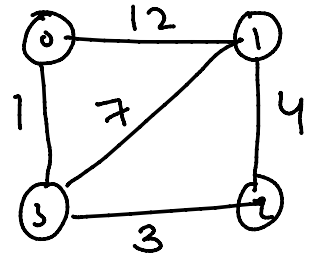
S.C = O(|V|).

Code

Floyd Warshal's

All pair shortest path

Dijkstra's



$0 \to 1$ $\quad 2 \to 0$

$0 \to 2$ $\quad 2 \to 1$

$0 \to 3$ $\quad 2 \to 3$

$1 \to 0$ $\quad 3 \to 0$

$1 \to 2$ $\quad 3 \to 1$

$1 \to 3$ $\quad 3 \to 2$

$$D = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 12 & \infty & 1 \\ 12 & 0 & 4 & 7 \\ \infty & 4 & 0 & 3 \\ 1 & 7 & 3 & 0 \end{bmatrix} \end{array}$$

i j [k]

d[i][j] > d[i][k] + d[k][j]

$$D_0 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 12 & \infty & 1 \\ 12 & 0 & 4 & 7 \\ \infty & 4 & 0 & 3 \\ 1 & 7 & 3 & 0 \end{bmatrix} \end{array}$$

(0 3) → 1

(0,1) + (1,3)

12 + 7 = 19

$$D_1 = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 12 & 16 & 1 \\ 12 & 0 & 4 & 7 \\ 16 & 4 & 0 & 3 \\ 1 & 7 & 3 & 0 \end{bmatrix} \end{array}$$

0 2 → ∞

(0,1) + (1,2)

12 + 4 = 16

(2,3) = 3

(2,1) + (1,3)

4 + 7 = 11

D_2

D_3

Code

```
for (int i = 0; i < V; i++)
{
    for (int j = 0; j < V; j++)
    {
        for (int k = 0; k < V; k++)
        {
            if(dist[i][j] > dist[i][k] + dist[k][j])
            {
                dist[i][j] = dist[i][k] + dist[k][j];
            }
        }

    }

}
```

T.C $O(|V|^3)$ ← dijkstra's $O(|V|^3 \log |V|)$

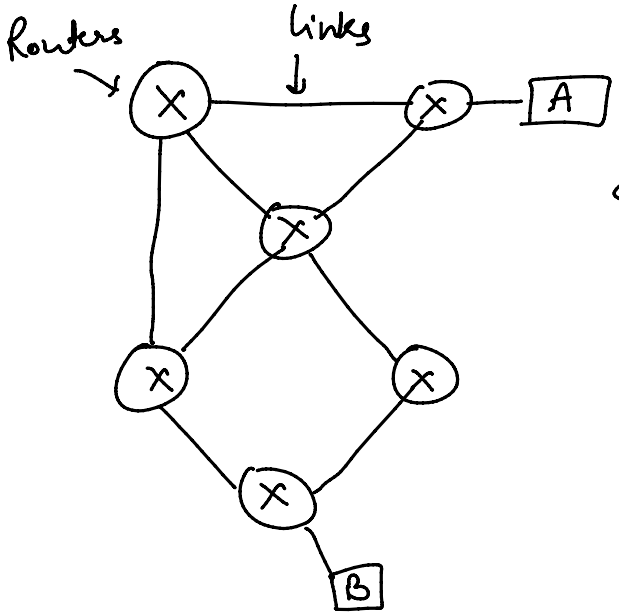
S.C = $O(|V|^2)$

Greedy method

Optimization problems

Minimize

Maximize



← Minimum Spanning tree

min. time

" cost

max. "

Approach to solve using Greedy method

A problem should be solved in stages, as we get input we see if that can be fit in result.

Company hires.

→ Written test, GD, series of interview

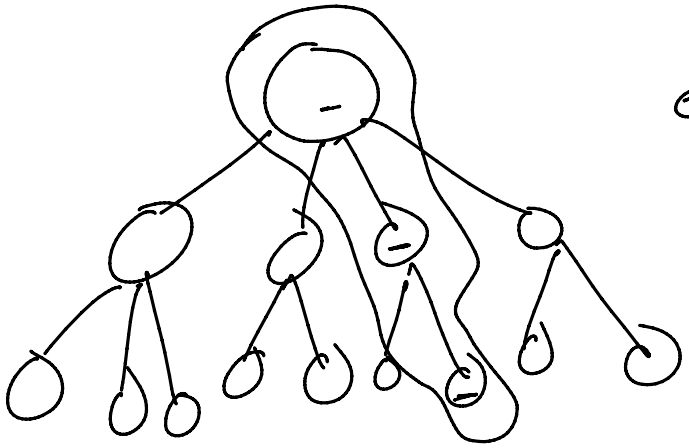
Which problems?

Optimal substructure property

Optimal solution of the problem is constituted of optimal solutions of it's subproblem.

optimal solution ...

of optimal solutions of it's subproblem.



eg Dijkstra's
Kruskal,
prims etc...

Activity selection, fractional knapsack,
Huffman encoding, Job sequencing etc...

Activity Selection

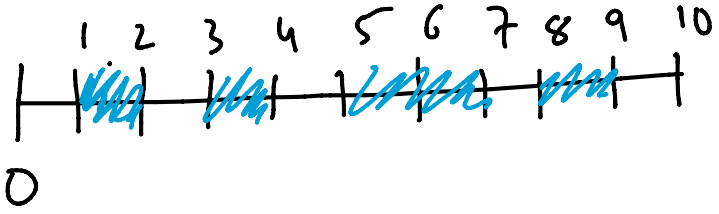
Uniprocessor → 1 activity at a time.

Start time end time

maxm no. of activities that you can run.

	0	1	2	3	4	5	6
Start Time	1	3	0	5	8	5	7
End time	2	4	6	7	9	9	9

4



Code

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;
int main()
{
    vector<pair<int, int> > activity;
    int n, s, e;
    cin>>n;

    for(int i = 0 ; i < n; i++){
        cin>>s>>e;
        activity.push_back(make_pair(e,s));
    }
    sort(activity.begin(), activity.end());

    int count = 0;
    int currentEnd = -1;

    for (int i = 0; i < n; i++){
        if(activity[i].second > currentEnd){
            count++;
            currentEnd = activity[i].first;
        }
    }

    cout<<count<<endl;
```

```
    cout<<count<<endl;
}
```

Job sequencing

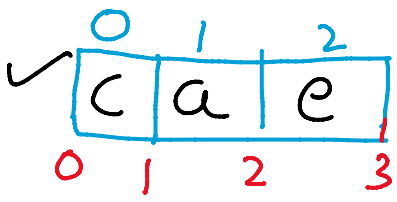
1 unit time

deadline & profit

↳ Maximize

JobID	Profit	Deadline
a	100	2
b	19	1
c	27	2
d	25	1
e	15	3

a	100	2
c	27	2
d	25	1 X
b	19	1 X
e	15	3



100 + 27 + 15

= 142

Code

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
bool compare (pair<int, int> a, pair<int, int> b)
{
    return a.first > b.first;
}
int main()
{
    vector<pair<int, int> > job;
    int n, profit, deadline;
    cin>>n;
    for(int i = 0 ; i < n; i++){
        cin>>profit>>deadline;
        job.push_back(make_pair(profit,deadline));
    }
    sort(job.begin(), job.end(), compare);
    int maxEndTime = 0;
    for (int i = 0; i < n; i++)
    {
        if(job[i].second > maxEndTime)
            maxEndTime = job[i].second;
    }

    int fill[maxEndTime];
    int count = 0, maxProfit = 0;
    for(int i = 0; i < maxEndTime; i++)
    {
        fill[i] = -1;
    }
    for (int i = 0; i < n; i++)
    {
        int j = job[i].second - 1;
        while(j>= 0 && fill[j] != -1)
        {
            j--;
        }

        if(j>=0 && fill[j] == -1)
        {
            fill[j] = i;
            count ++;
```

profit deadline

O(nlogn)

O(n)

O(maxEndtime) → n

n

O(n)

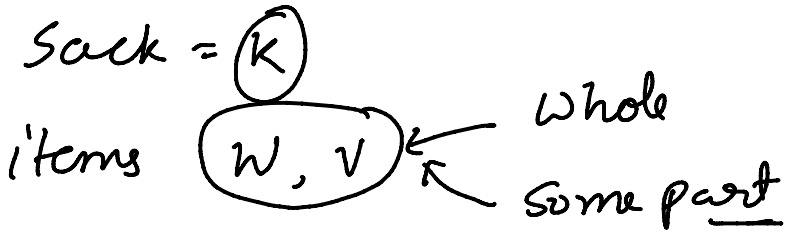
$O(n^2)$

T.C $O(n^2)$

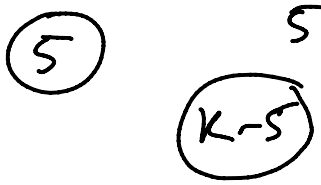
```
        {
            fill[j] = i;
            count ++;
            maxProfit += job[i].first;
        }
    }

    cout<<count<<" "<<maxProfit<<endl;
}
```

SC = O(n)

Fractional knapsack



V = 100
W = 10
V/W = 10



maximize the profit

index	0	1	2	3	4
w	5	10	20	30	40
v	30	20	100	90	160
v/w	6	2	5	3	4

index	0	2	4	3	1
W	5	20	40	30	10
V	30	100	160	90	20
V/W	6	5	4	3	2

K = 50 - 5 - 20 - 25

profit = 30 + 100 + 100 = 230

T.C = O(n log n) + O(n)

$= O(n \log n)$

S-C = $O(n)$.

Huffman encoding

a → 8
b → 8
c → 8
d → :
e → :
. :
: :
z → 8

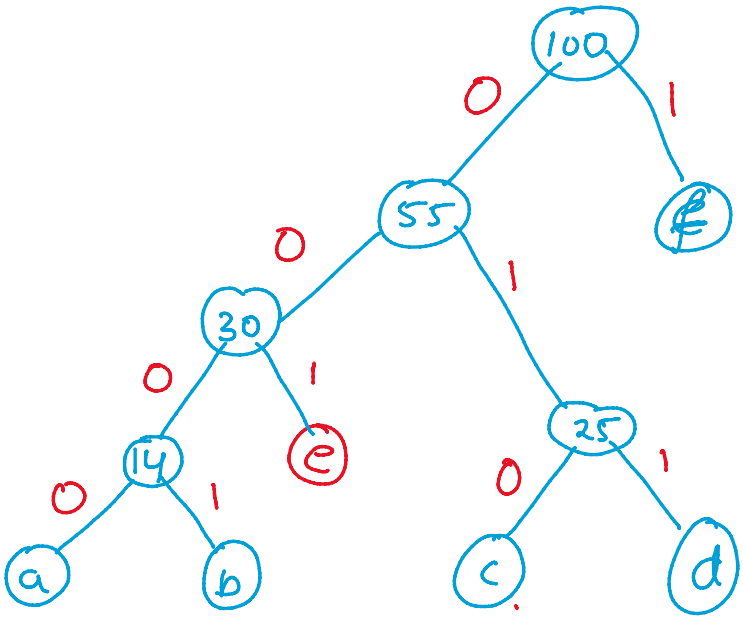
Varicble length encoding

a → 01
. 01
:

001
000
101

character	a	b	c	d	e	f
freq	5	9	12	13	16	45

abcdef = 100
a → 5
b → 9
c → 12
d → 13
e → 16
f → 45
ab → 14
cd → 25
abe → 30
abcde → 55



a = 0000
b = 0001
c → 010
d → 011
e → 001
f → 1

001 010 011 0000
e c d a

a → 5 → 8 bits
b → 9 → 8 bits
c → 12 → 8 "
d → 13 → 8 "
e → 16 → " "
f → 45 → " "

100K 800000 bits

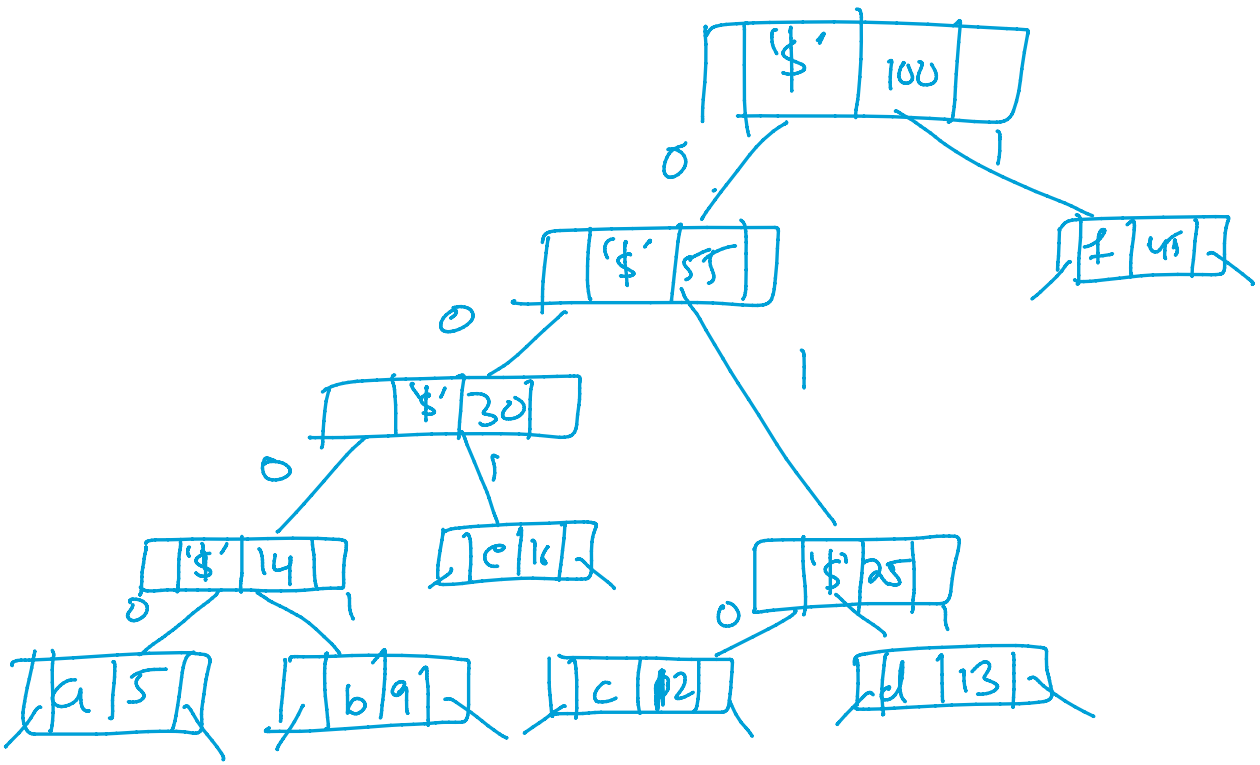
100KB

Save 72KB

4 = 20
4 = 36
3 = 36
3 = 39
3 = 48
1 = 45

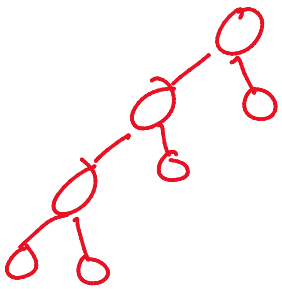
224000 / 8

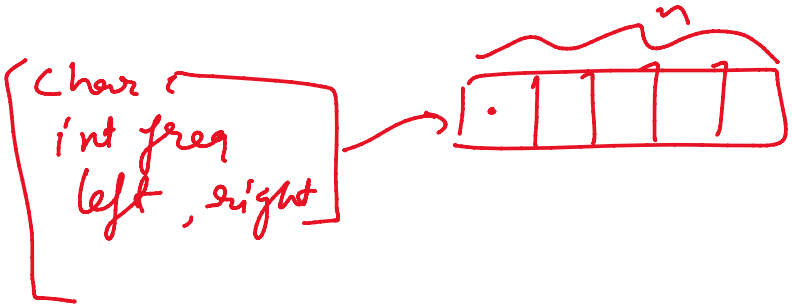
28000 = 28KB



Code

```
void print(minHeapElement *root, string str)
{
    if(root->c != '$')
    {
        cout<<root->c<<" "<<str<<endl;
        return;
    }
    print(root->left, str+'0');
    print(root->right, str+'1');
}

int main()
{
    int n;
    cin>>n;
    minHeapElement* arr[n];
    char c;
    int freq;
    for (int i = 0; i < n; i++)
    {
        cin>>c>>freq;
        arr[i] = new minHeapElement();
        arr[i]->c = c;
        arr[i]->freq = freq;
        arr[i]->left = arr[i]->right = NULL;
    }
    MinHeapify(arr, n);
    while(n != 1)
    {
        minHeapElement *he1 = extract_min(arr, n), *he2 = extract_min(arr, n), *he3;
        he3 = new minHeapElement();
        he3->c = '$';
        he3->freq = he1->freq + he2->freq;
        he3->left = he1;
        he3->right = he2;
        insert(arr, n, he3);
    }
    print(arr[0], "");
}
```





O(n)

O(log n)

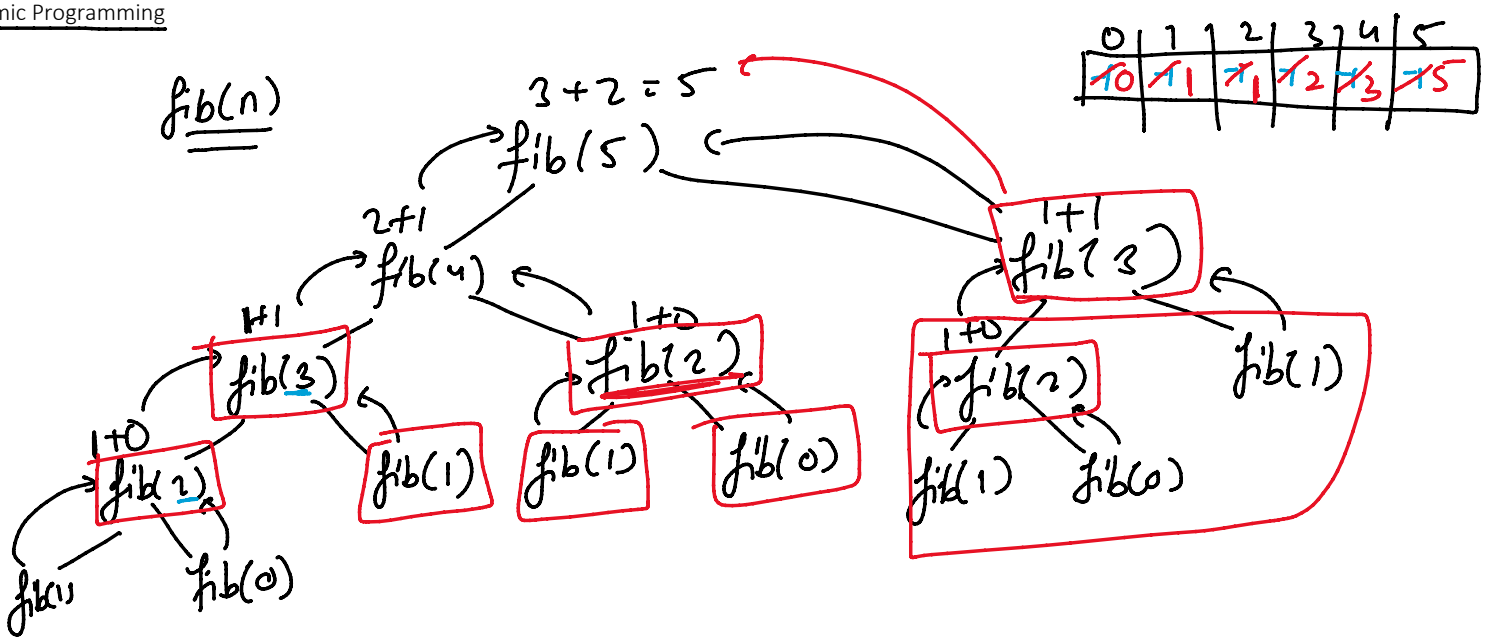
O(log n)

O(log(n))

① O(n log n)

T.C = O(n log n)

S.C → O(n)



→ Overlapping subproblems

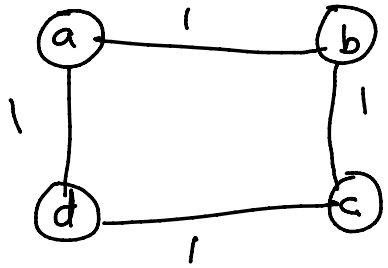
→ optimal substructure.

↳ optimal solution of the problem depends on optimal solution of the subproblem.

a → c → a → b, b → c
→ a → d, d → c

a b c

a d c



a → c → a b c
→ a d c

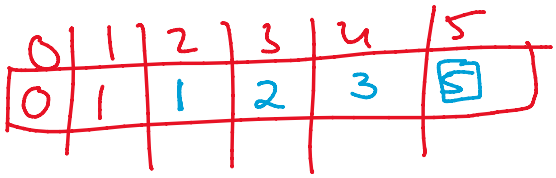
Shortest distance
Bellman ford

[longest distance without cycle]

Tabulation → bottom up ✓

memorization → Top down

memoization → Top down



Fibonacci

$$fib(n) = \begin{cases} fib(n-1) + fib(n-2) & , \text{if } n > 1 \\ n & , \text{otherwise} \end{cases}$$

$O(2^n)$

0, 1, 1, 2, 3, 5, 8, 13, 21 ----

$\theta(1.6^n)$

$O(n)$ dp

Code

Recursive

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

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

Top Down

memoization.

```
int fib(int n)
{
    if (lookup[n] == -1)
    {
        if (n <= 1)
            lookup[n] = n;
        else
            lookup[n] = fib(n - 1) + fib(n - 2);
    }
    return lookup[n];
}
```

lookup

0	1	2	3	4	5
-1	-1	-1	-1	-1	-1

Bottom up

```
int fib(int n)
{
  int f[n+1];
  int i;

  f[0] = 0;   f[1] = 1;

  for (i = 2; i <= n; i++)
      f[i] = f[i-1] + f[i-2];

  return f[n];
}
```

0	1	2	3	4	5
0	1	1	2	3	5

$f[n+1] =$

Fractional Knapsack → whole item / fraction of it

0/1 Knapsack → take / ignore

W	20	60	50	2
V	100	300	500	40

K = 50

max

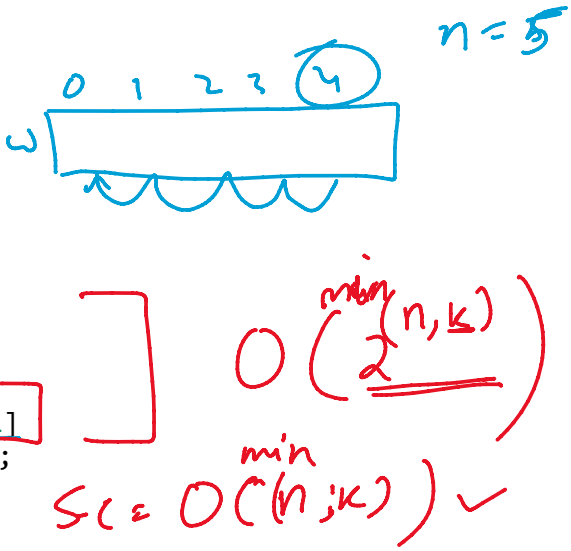
take: next item, K - W, P ↑ V

ignore: next item, K < W

Code

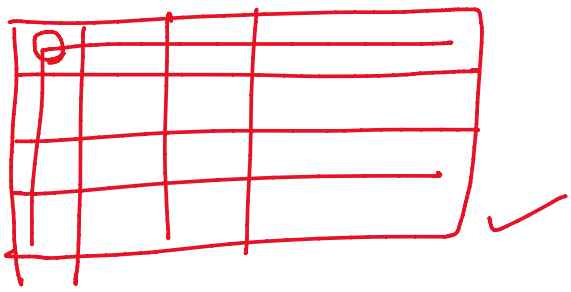
Recursive

```
int knapSack(int wt[], int val[], int n, int K)
{
    if (n == 0 || K == 0)
        return 0;

    if (wt[n - 1] > K)
        return knapSack(wt, val, n - 1, K);

    else
        return max (knapSack(wt, val, n - 1, K), val[n - 1]
                + knapSack(wt, val, n - 1, K - wt[n - 1]));
}
```



Bottom UP

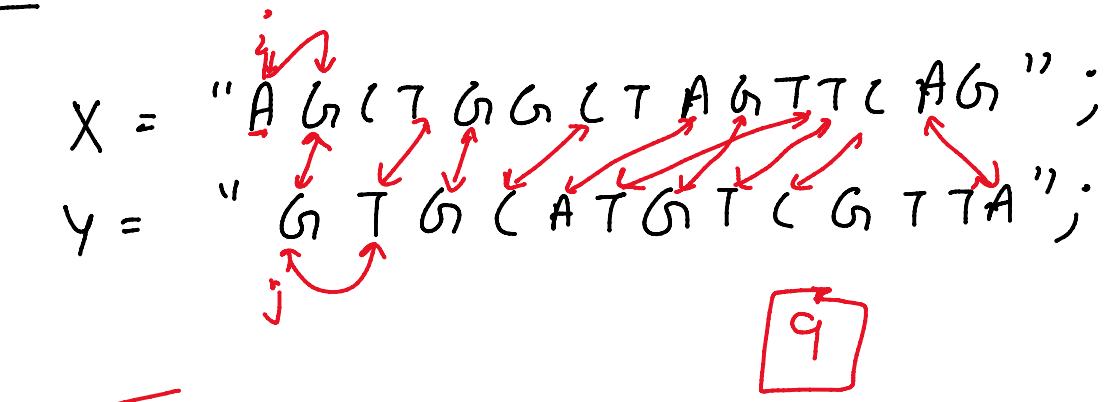
```
int knapSack(int wt[], int val[], int n, int K)
{
    int i, w;
    int sol[n + 1][K + 1];

    for(i = 0; i < n + 1; i++)
    {
        for(j = 0; j < K + 1; j++)
        {
            if (i == 0 || j == 0)
                sol[i][j] = 0;

            else if (wt[i-1] > j)
                sol[i][j] = sol[i - 1][j];

            else
                sol[i][j] = max(sol[i - 1][j],
                                val[i - 1] + sol[i - 1][j - wt[i - 1]]);
        }
    }
    return sol[n][K];
}
```



T.C O(n*k)

S.C ⇒ O(n*K)

Longest Common Subsequence



$$X[i] == Y[j] \quad (i{+}{+};\ j{+}{+};)$$

else $\max(LCS(i+1, j),\ LCS(i, j+1))$

Code

Recursive

```
int lcs(string X, string Y, int m, int n )
{
    if (m == 0 || n == 0)
        return 0;
    if (X[m-1] == Y[n-1])
        return 1 + lcs(X, Y, m-1, n-1);
    else
        return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
}
```

i=0 j=0

m → len(X) → 0 → m-1
n → len(Y) → 0 → n-1

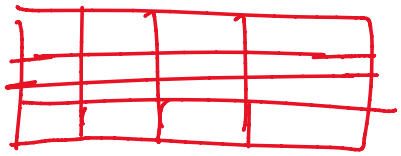
T.C $O(2^{\min(m,n)})$

S.C = O(m+n)

DNA

DP

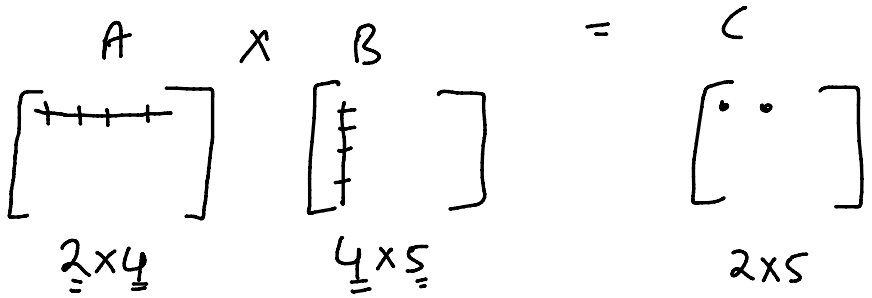
```
int lcs(string X, string Y, int m, int n )
{
    int sol[m + 1][n + 1];
    int i, j;
    for (i = 0; i <= m; i++)
    {
        for (j = 0; j <= n; j++)
        {
            if (i == 0 || j == 0)
                sol[i][j] = 0;

            else if (X[i - 1] == Y[j - 1])
                sol[i][j] = sol[i - 1][j - 1] + 1;

            else
                sol[i][j] = max(sol[i - 1][j], sol[i][j - 1]);
        }
    }
    return sol[m][n];
}
```



T.C = O(m*n)

S.C = O(m*n)

n matrices



Total opn : $2 \times 4 \times 5 = 40$

$A \underset{2\times3}{} B \underset{3\times5}{} C \underset{5\times4}{} = \underset{2\times4}{D}$

- $(AB)C$
- $A(BC)$

$\underset{2\times3}{A} \underset{3\times5}{B} \underset{5\times4}{C}$

$(AB)C$

$AB \rightarrow 2 \times 3 \times 5 = 30$ $\quad \underset{2\times5}{A'}$

$A'C = 2 \times 5 \times 4 = 40$

70 ✓

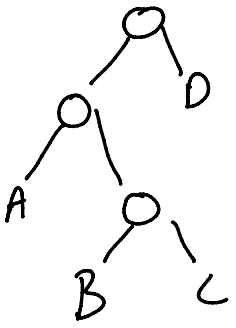
$A(BC)$

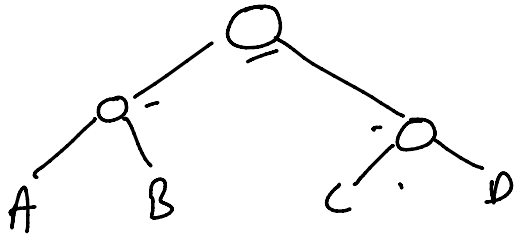
$BC \rightarrow 3 \times 5 \times 4 = 60$ $\quad \underset{3\times4}{B'}$

$AB' \rightarrow 2 \times 3 \times 4 = 24$

84 ✓

⇒ A B C D





n internal nodes

$\Rightarrow$ # of trees = $\frac{2n C_n}{n+1}$ ← Catalan number.

$$\frac{2*3 C_3}{(3+1)} = \frac{6C_3}{4} = \frac{6!}{3!3! \times 4}$$

$$= \frac{6 \times 5 \times 4 \times 3!}{6 \times 3! \times 4} = 5$$

1) (A (B (C D)))

2) ((A B) (C D))

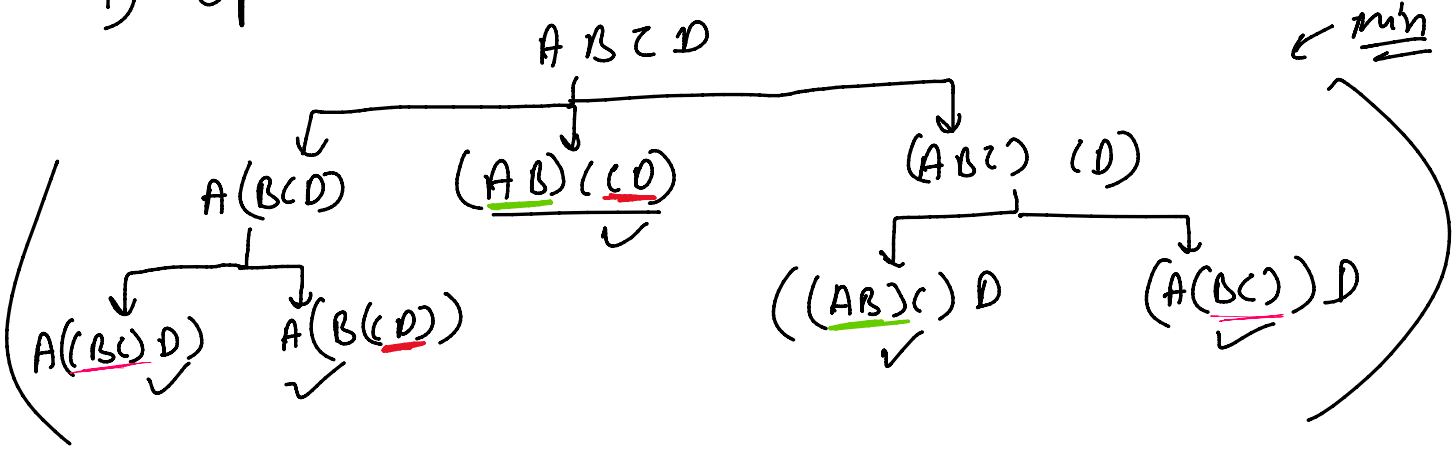
3) (A ((B C) D))

4) ((A (B C)) D)

5) (((A B) C) D)

If we have $n+1$ matrices to multiply total ways are $\frac{2n C_n}{n+1}$.

1) Optimal substructure →



2) Overlapping subproblems

↳ Recursion.

↳ DP

	A	B	C	D
	2×3	3×5	5×4	4×7

		2×3 A	3×5 B	5×4 C	4×7 D
		0	1	2	3
2×3	A 0	0	30	70	126
3×5	B 1		0	60	148
5×4	C 2			0	140
4×7	D 3				0

	0	1	2	3
0		2×5 (AB)	2×4 (AB)C	((AB)C)D 2×7
1			3×4 BC	3×7 (BC)D
2				5×7 CD
3				

L = 2, 3, 4 ↑

A B C

(A B) C = 30 + 2 × 5 × 4 = 70

A B C

↳ (A B) C = 30 + 2 × 5 × 4 = 70

A (B C) = 60 + 2 × 3 × 4 = 84

B C D

↳ (B C) D = 60 + 3 × 4 × 7 = 148

↳ B (C D) = 140 + 3 × 5 × 7 = 245

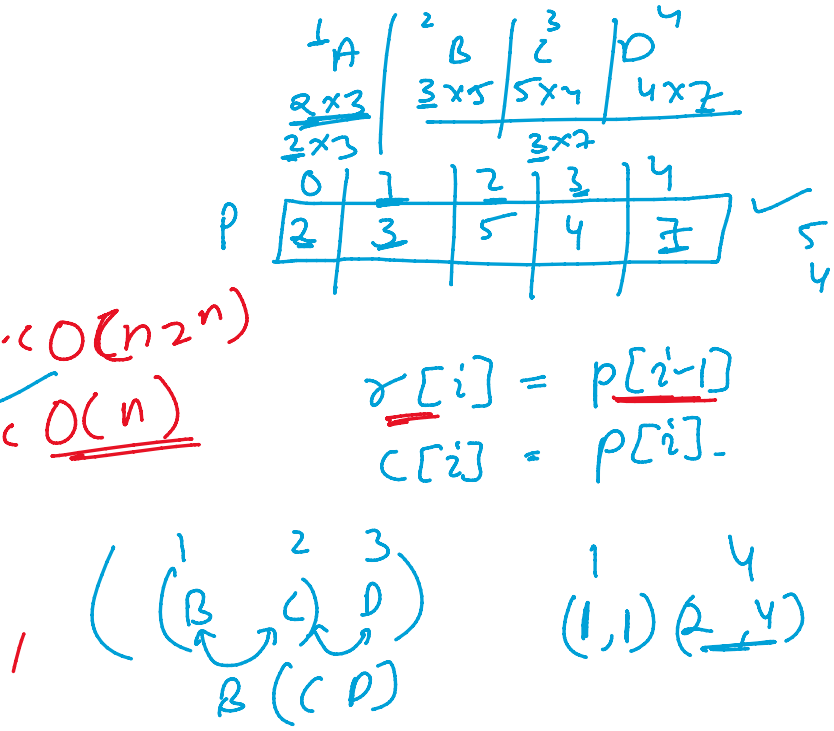
A B C D

↳ ((A B) C) D → 70 + 2 × 4 × 7 = 126

↳ A ((B C) D) ⇒ 148 + 2 × 3 × 7 = 190

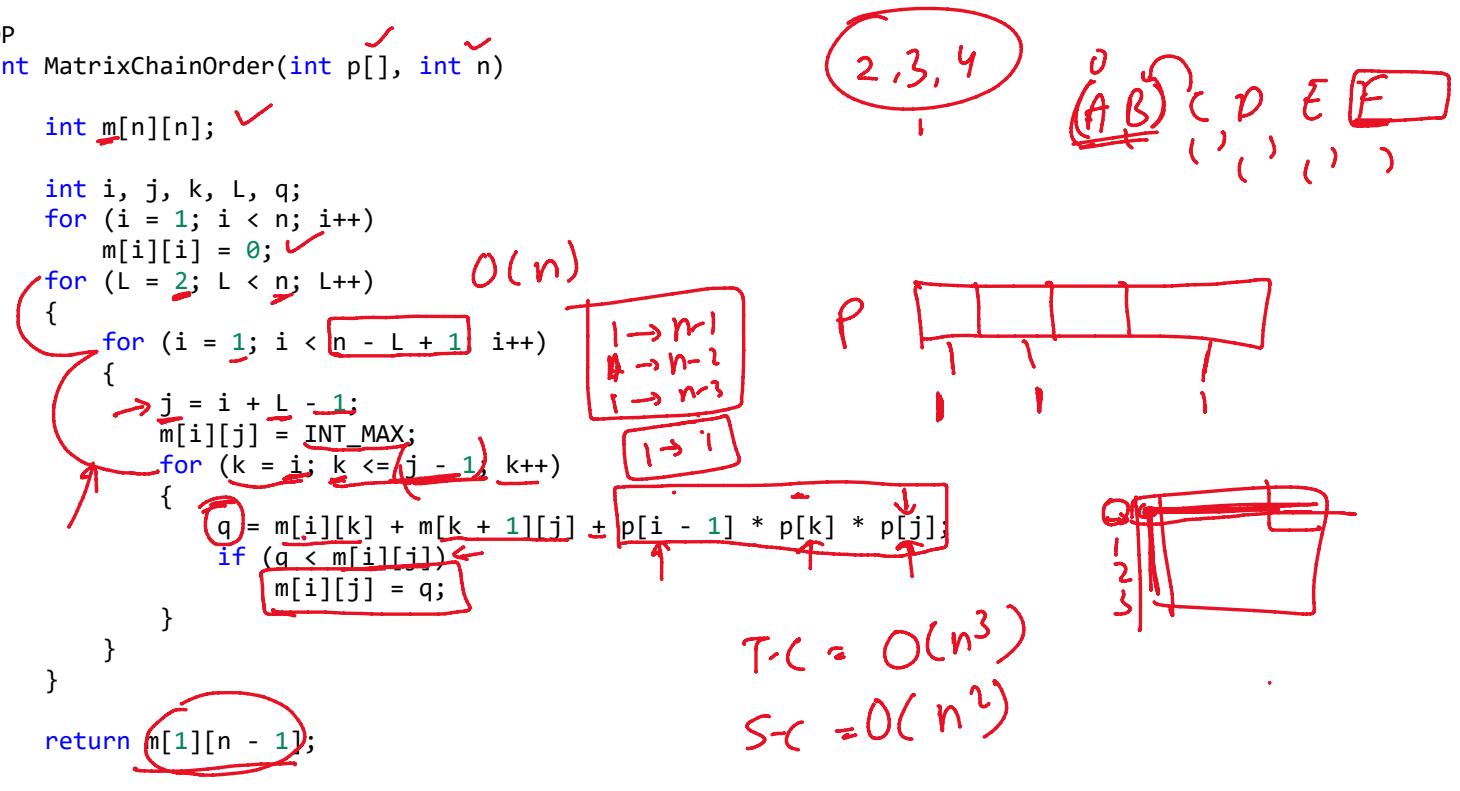
Code

```
int MatrixChainOrder(int p[], int i, int j)
{
    if (i == j)
        return 0;
    int k;
    int min = INT_MAX;
    int count;
    for (k = i; k < j; k++)
    {
        count = MatrixChainOrder(p, i, k)
                + MatrixChainOrder(p, k + 1, j)
                + p[i - 1] * p[k] * p[j];

        if (count < min)
            min = count;
    }

    return min;
}
```



DP

```
int MatrixChainOrder(int p[], int n)
{
    int m[n][n];

    int i, j, k, L, q;
    for (i = 1; i < n; i++)
        m[i][i] = 0;
    for (L = 2; L < n; L++)
    {
        for (i = 1; i < n - L + 1; i++)
        {
            j = i + L - 1;
            m[i][j] = INT_MAX;
            for (k = i; k <= j - 1; k++)
            {
                q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }

    return m[1][n - 1];
}
```



Naive String Matching

P → All problems that can be solved & verified in polynomial time.

NP → All problems that can be verified in polynomial time.

P

G,

∃ MST $w \leq 10$

→ fractional Knapsack

$p > 10$ 50

$p = 25$

$x^2 + y^2 = 10$

$x = \sqrt{5}$

$y = \sqrt{5}$

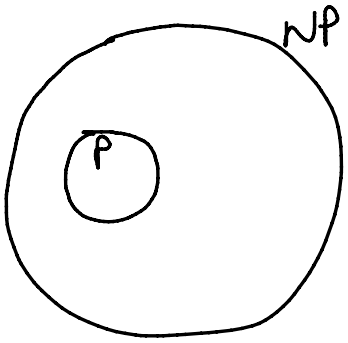
NP

→ Verifiable in polynomial time.

G T.S.P

Subset sum problem

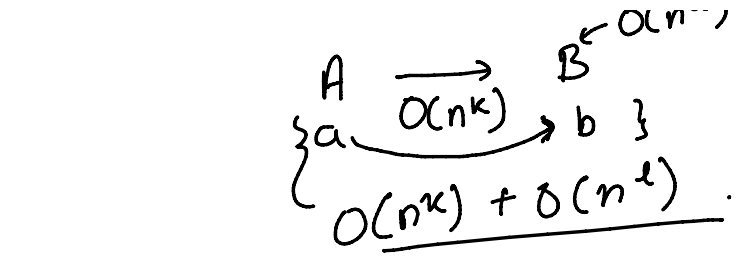
0/1 Knapsack.



→ A ⟶ B ← Polynomial time soln of B

polynomial time

A ⟶ B ← $O(n^2)$

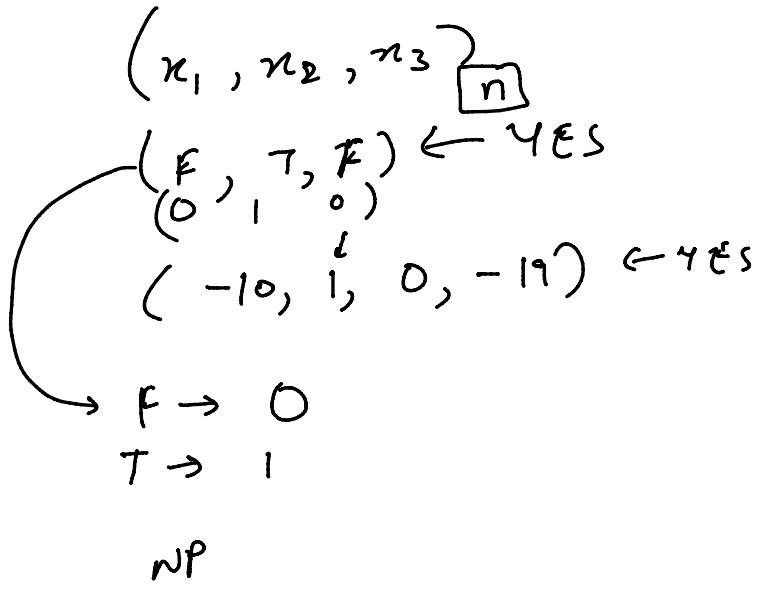


1) Every instance a of A can be transformed to some instance b of B in polynomial time

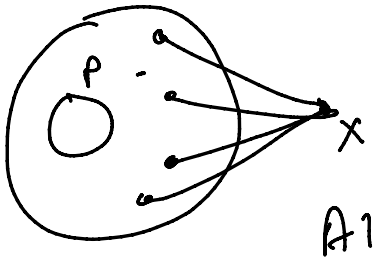
2) If a's answer is YES then b's answer should be YES

Eg A → boolean variables
↳ $x_1, x_2 \text{ ---- } x_n$

B → Given n ints.
Whether there is a true integer

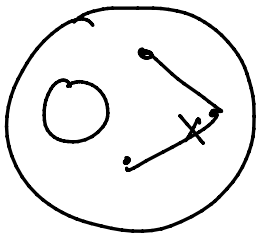


NP



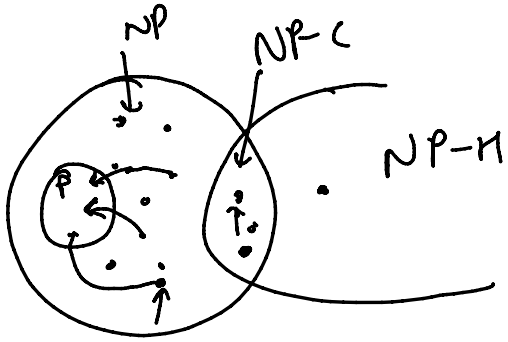
All NP problems can be transformed to 'X' in polynomial time

X → NP-Hard.



& X is NP.

X → NP-Complete.



P = NP

P ≠ NP

ABC → MDS

NP = P

NP ≠ P

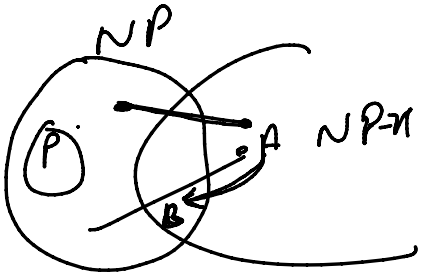
→ NP-H or NP-C is solved in polynomial time then P = NP

→ NP or NP-C no polynomial time solution exists then P ≠ NP.

NP-H → A → B

NP

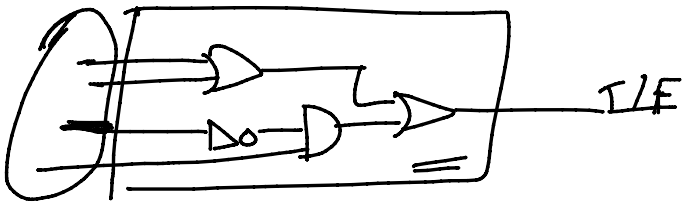
NP-H → A $\xrightarrow{P}$ B ↑ NP-H.



NP-H ⇒ A $\xrightarrow{P}$ B ← NP

then B → NP-C

1) Circuit Satisfiability (SAT)

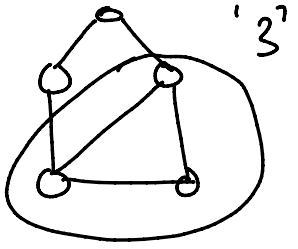


2) SAT
↳ $(x^2+y+3)(x+y) = 2$

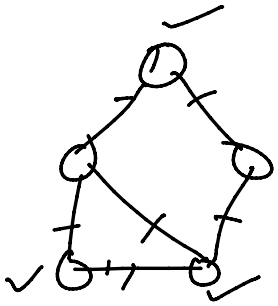
3) 3-SAT 3 CNF SAT
↳ $(x+y+\bar{z})(\bar{x}\,y+z)$

4) subset sum & T-SP.

5) Clique ←



6) Vertex cover



Subset sum problem

$S = \{ x_1, x_2 \ldots\ldots x_n \}$

W -

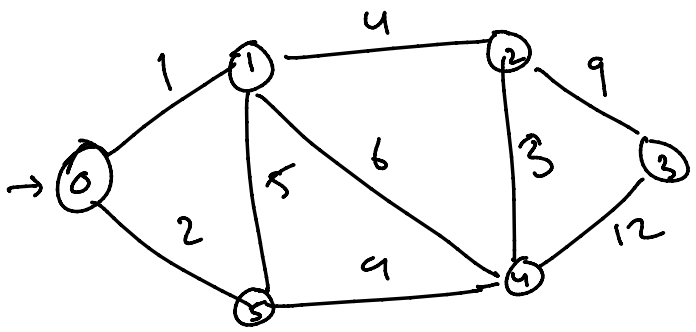
included excluded

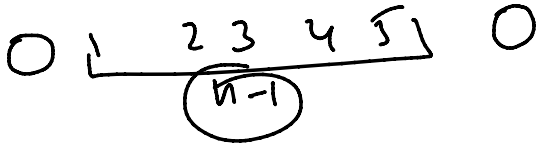
```
bool SS (int *s, int n, int W)
{
    if (W == 0) return true;
    if (n == 0) return false;
    if ( W < S[n-1] )
        return SS (S, n-1, W);
    else
        return SS(S, n-1, W) || SS(S, n-1, W-S[n-1]);
}
```

O(2^n).

$S = \{ 5, 4, 3, 1 \}$

$W = 7 \rightarrow 2$



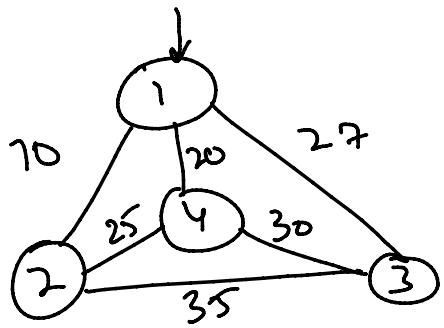


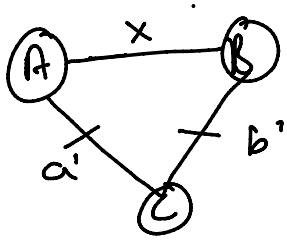
$(n-1)!$

$O(n!) \approx O(n^n) >>> O(2^n)$

$\hookrightarrow O(n\,2^n)$

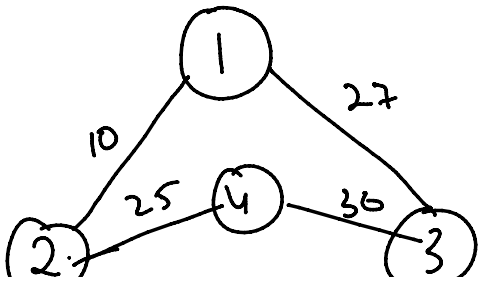
Eg





$a' + b' > x$ ✓

1. **Triangle inequality**



1 2 4 3

1 2 4 3 1

1 2 4 3 1

X = "a b b a a b a b b"

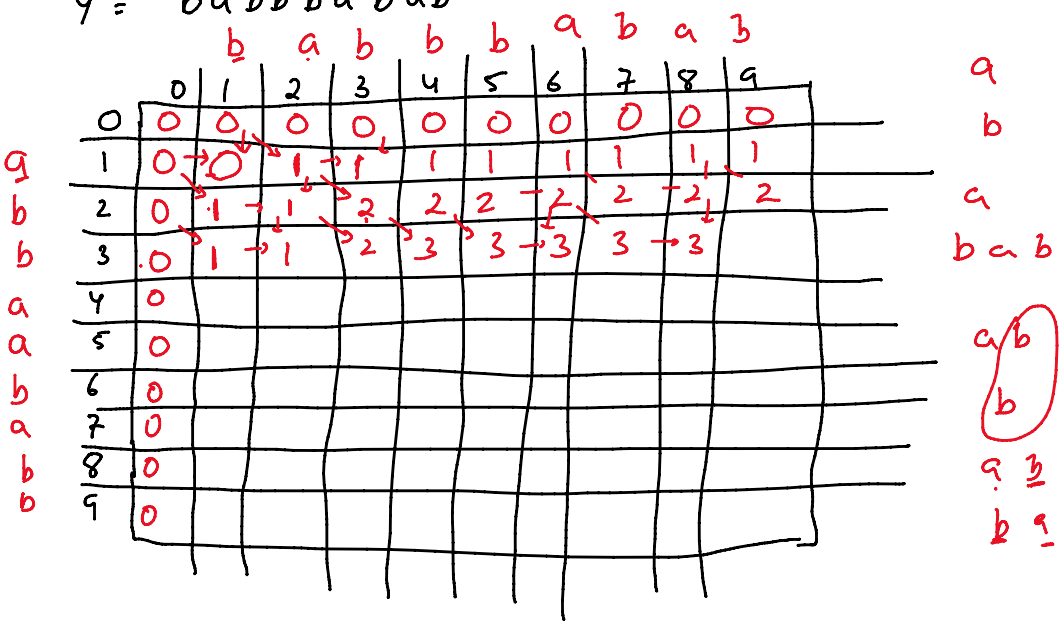
Y = "b a b b b a b a b"

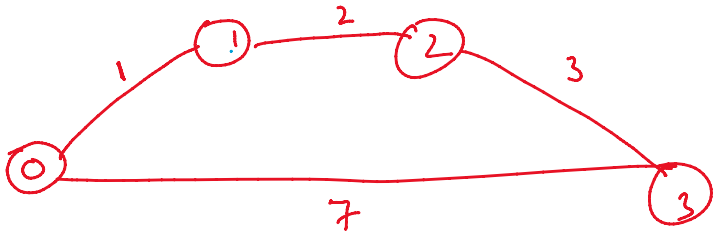
a b

b a b

b a b b

a b b





7 + 3 = 10

$$D(0,2) = \min \begin{cases} D(0,2) \\ D(0,1) + D(1,2) \\ D(0,3) + D(3,2) \end{cases}$$

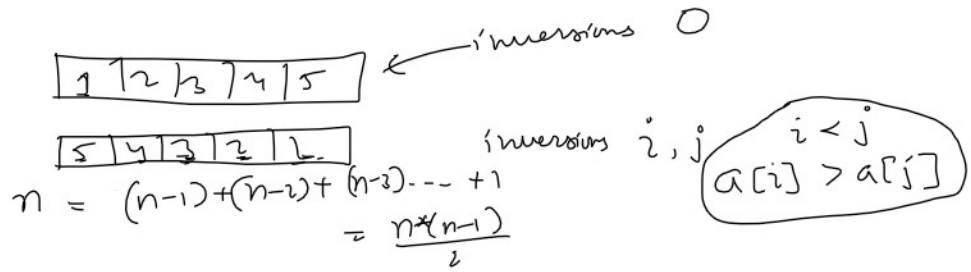
	0	1	2	3
0	0	1	~~3~~ 7	7
1	1	0	2	∞
2	∞	2	0	3
3	7	∞	3	0

k

i j

D(0,3) = 7

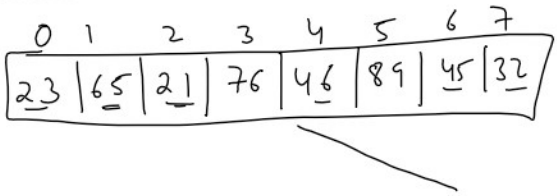
$$D(0,3) \overset{\min}{=} \begin{cases} D(0,3) = 7 \\ D(0,0) + D(0,3) = 7 \\ D(0,1) + D(1,3) = \infty \\ D(0,2) + D(2,3) = 3 + 3 = 6 \end{cases}$$



Week 4:

I. Given an unsorted array of integers, design an algorithm and implement it using a program to sort an array of elements by dividing the array into two subarrays and combining these subarrays after sorting each one of them. Your program should also find number of comparisons and inversions during sorting the array.

Merge Sort



Input Format:
The first line contains number of test cases, T.
For each test case, there will be two input lines.
First line contains n (the size of array).
Second line contains space-separated integers describing array.

Output Format:
The output will have T number of lines.
For each test case T, there will be three output lines.
First line will give the sorted array.
Second line will give total number of comparisons.
Third line will give total number of inversions required.

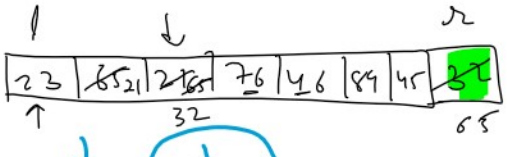
Sample I/O Problem I:

Input:	Output:
3 8 23 65 21 76 46 89 45 32 10 54 65 34 76 78 97 46 32 51 21 15 63 42 223 645 652 31 324 22 553 12 54 65 86 46 325	21 23 32 45 46 65 76 89 comparisons = 16 inversions = 21 32 34 46 51 54 65 76 78 97 comparisons = 22 inversions = 12 22 31 42 46 54 63 65 86 223 324 325 553 645 652 comparisons = 43 inversions =

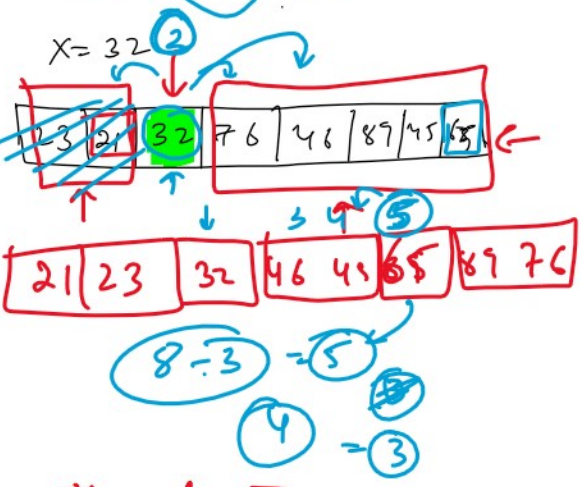
II. Given an unsorted array of integers, design an algorithm and implement it using a program to sort an array of elements by partitioning the array into two subarrays based on a pivot element such that one of the sub array holds values smaller than the pivot element while another sub array holds values greater than the pivot element. Pivot element should be selected randomly from the array. Your program should also find number of comparisons and swaps required for sorting the array.

Divide & Conquer

Quicksort



Input Format:



The first line contains number of test cases, T.
For each test case, there will be two input lines.
First line contains n (the size of array).
Second line contains space-separated integers describing array.

Output Format:
The output will have T number of lines.
For each test case T, there will be three output lines.
First line will give the sorted array.
Second line will give total number of comparisons.
Third line will give total number of swaps required.

Sample I/O Problem II:

Input:	Output:
3 8 23 65 21 76 46 89 45 32 10 54 65 34 76 78 97 46 32 51 21 15 63 42 223 645 652 31 324 22 553 12 54 65 86 46 325	21 23 32 45 46 65 76 89 comparisons = 14 swaps = 10 21 32 34 46 51 54 65 76 78 97 comparisons = 29 swaps = 21 12 22 31 42 46 54 63 65 86 223 324 325 553 645 652 comparisons = 45 swaps = 39

III. Given an unsorted array of integers, design an algorithm and implement it using a program to find Kth smallest or largest element in the array. (Worst case Time Complexity = O(n))

Input Format:
The first line contains number of test cases, T.
For each test case, there will be three input lines.
First line contains n (the size of array).
Second line contains space-separated integers describing array.

Input Format:
The first line contains number of test cases, T.
For each test case, there will be three input lines.
First line contains n (the size of array).
Second line contains space-separated integers describing array.
Third line contains K.

Output Format:
The output will have T number of lines.
For each test case, output will be the Kth smallest or largest array element.
If no Kth element is present, output should be "**not present**".

Sample for Kth smallest:

Input:	**Output:**
3 10 123 656 54 765 344 514 765 34 765 234 3 15 43 64 13 78 864 346 786 456 21 19 8 434 76 270 601 8	123 78

Week 5:

I. Given an unsorted array of alphabets containing duplicate elements. Design an algorithm and implement it using a program to find which alphabet has maximum number of occurrences and

$(k-1)^{th}$ index
$(n-k)^{th}$ index
$O(n \log n)$

$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} \cdots 1$

$n\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots \frac{1}{2^k}\right)$

$2^k = n$
$k = \log_2 n$

$\approx (2n-1) = O(n)$