25/7/18

Algorithms

Syllabus

1) Analysis
2) Divide & Conquere
3) Greedy Techniques
4) Dynamic Programming
5) Hashing, Tree Traversal & Graph Traversal

Reference book :- Introduction to Algorithm by Cormen

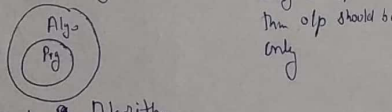
Def: It is a combination of sequence of finite steps to solve a particular problem.

MTN: mul of 2 nos

Ex:
```
MTN( )
{
 1. take 2-nos (a,b)
 2. multiply and store in c.
 3. Pf (c);
}
```

Properties of algorithm

* 1) It should terminate after finite time.
* 2) It should produce atleast one output
3) It should be deterministic
4) It is independent of programming language



after long time if we give i/p a then o/p should be S_2 only

Steps Required to Construct Algorithm

1. Problem Definition
   understand problem clearly ie for every possible i/p what are all possible o/p's.
2. Design the Algorithm ← GP, DP, D&C
   Select suitable algorithms from the existing algorithms

3) Flow Chart : diagrammatic representation
4) Testing (verification) by giving diff cmt i/p's.
5) Coding : using same prog lang ~~implement~~ implement.
6) Analysis (How much time (CPU) & how much memory main)

Analysis (Imp)

→ If a problem contain more than one solution best one will be decided by analysis based on 2 factors :-
1) Time Complexity (CPU time)
2) Space Complexity (Main memory space)

Time Complexity

C(P) = Compile time
R(P) = Run time

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

Time Complexity of problem P
- C(P) ⇓ Compiler ⇓ S/W ⇓ Language of compiler
- R(P) ⇓ CPU ⇓ H/W ⇓ Type of H/W matters

Compiler → set of rules

Analysis

- APosteriori Analysis
- APriori Analysis ~~Profile~~ Analysis

Analysis

Apostiary Analysis

→ Lang of Complier & Type of H/w dependent

~~Prog~~ Analysis

adv → Exact Answer

disadv → Diff answer in different System.

Aprisri Analysis

→ Independent analysis L-C & Type of H/w (Lang of Compiler)

disadv → Approximate answer

adv → Same answer in every System.

Note: Every one can't ~~write~~ buy Super Comp ; but every one can write Super algorithm bcz ~~for~~ everyone having brain . Some people use brain & some dont.

Aprisri Analysis

It is a determination of order of magnitude of a Statement

* While running execute Statement will ~~run~~ how many times

Ex:1

```
Main()
{
1.  x = y + z;        ⟹ 1     TC: O(1)
}
```

↑ Constant Time (complexity)

Ex:2

```
main()
{
1. → x = y+z;                          ——— 1        TC: O(n)
     for(i=1; i≤n; i++)
     {
2.     x = y+z;                        ——— n
     }
}
```

1 time, n+1 time, n time

n time

n+1 = O(n) ↙ take greater

Sysm to sysm constant will change. but n will always be there

Ex:3

```
Main()
{
1.  x = y+z;                           ——— 1
    for(i=1; i≤n; i++)
2.      x = y+z                        ——— n
    for(i=1; i≤n; i++)
    {   for(j=1; j≤n; j++)
        {
3.          x = y+z;                   ——— n²
        }
    }
}
```

$n^2 + n + 1 \Rightarrow O(n^2)$

* part which runs again & again is kept in cache memory

* finding TC means finding loops (Larger loops) ie where CPU spending more time.

26/7/18

Ex-4

```
Main ()
{
1.  i=1                          ~~~~~~ *
    While (i<=n)
2.  {  i = i+1          ------ n
3.  }
```

n times $\rightarrow O(n)$

$i = i+2 \longrightarrow n/2 \rightarrow O(n)$

$i = i+5 \longrightarrow n/5 \rightarrow O(n)$

$i = i+50 \longrightarrow n/50 \rightarrow O(n)$

We always see what is the final change in the value of i

```
{
  i = i+50
  i = i+5      } ---> n/55 --> O(n)
}
```

Addition $\overset{\text{inverse}}{\rightleftharpoons}$ Subtraction

Ex-5.1

```
Main ()
{
  i = n
  While (i >= 1)
  {
    i = i-1          ----> O(n)
  }
}
```

$i = i-10 \longrightarrow \frac{n}{10} \Rightarrow O(n)$

$i = i-100 \longrightarrow \frac{n}{100} \Rightarrow O(n)$

```
{ i = i-100
  i = i-300 } ---> n/300 => O(n)
}
```

Eg.2

```
Main ()
{
  i = n
  While (i > 1)
  {
    i = i-100
    i = i-200     } i = 125
    i = i+325
  }
}
```

It is pgm not algorithm, bcz it will run infinite times

Ex-5

```
Main ()
{
  i = 1
  While (i <= n)
  {
    i = 2*i;          ---> log2 n + 1
  }
}
```

$n=64$	$i=1$
$n=64$	$i=2$
$n=64$	$i=4$
$n=64$	$i=8$
$n=64$	$i=16$
$n=64$	$i=32$

$i=64$, $i=64$ break total $= 6+1$ times

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

$i = 3*i \longrightarrow \log_3 n + 1 \rightarrow O(\log_3 n)$

$i = 5*i \longrightarrow \log_5 n + 1 \rightarrow O(\log_5 n)$

Proof: $1\ 2^1\ 2^2\ 2^3\ 2^4 \ldots\ 2^k = n$ ← at kth iteration i is equal to n

$k \log_2 2 = \log n$

$k = \log_2 n$

While ($i < n$)
↳ $k = \log n$

While ($i \le n$) ← break condition
↳ $k = \log_2 n + 1$

Ex.1

```
Main()
{
  i=10;
  while(i<n)
  {
    i = 3*i;  ----> O(log3 n)
  }
}
```

10, $3^1 \cdot 10$, $3^2 \cdot 10$, $3^3 \cdot 10$, ... $3^k \cdot 10 = n$

$3^k \cdot 10 = n$

$\log_3(3^k \cdot 10) = \log_3 n$

$k + \log_3 10 = \log_3 n$

$k = \log_3 n - 2.5$ $(\because \log_3 10 \approx 2.5)$

Multiplication $\xrightarrow{\text{inverse}}$ Division $\xleftarrow{}$

Ex.2

```
Main()
{
  i=n
  while(i>=1)
  {
    i = i/2   i = i/3   ] -> i/6 => O(log6 n)
  }
}
```

$n, \frac{n}{2}, \frac{n}{2^2}, \frac{n}{2^3}, \ldots\ldots \frac{n}{2^k}$ at k^{th} iteration $i = 1$

$2^k = n$

$2^k = n$

$k = \log_2 n$

Ex.6

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

$2, 2^{2^1}, 2^{2^2}, 2^{2^3}, 2^{2^4}, \ldots\ldots$

$2^{2^k} = n \rightarrow k = \log_2 \log_2 n$

~~$2, 2^2, 2^4, 2^8, 2^{16}, \ldots\ldots (2^k)^2$~~

~~$2^{2k} = n$~~

~~$2k = \log n$~~

~~$2, 2^2, (2^2)^2, (2^{2^2})^2, 2^{2^3}, 2^{2^4}, \ldots\ldots$~~

~~$2, (2^2)^1, (2^2)^2, (2^2)^3, (2^2)^4, (2^2)^5, \ldots\ldots (2^2)^k = n$~~

$2^{2^0}\ 2^{2^1}\ 2^{2^2}\ 2^{2^3}\ 2^{2^4}\ 2^{2^5} \ldots\ldots 2^{2^k} = n$

$2^{2^k} = n$

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

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

$\log 2^k = \log \log n$

$k = \log \log n$

$TC \rightarrow O(\log_2 \log_2 n)$

6.1]

```
Main()
{
  i=2;
  while(i<=n)
  {
    i = i^23
  }
}
```

$2, 2^{23^1}, 2^{23^2}, 2^{23^3}, 2^{23^4}, \ldots, 2^{23^k} = n$ $\quad k = \log_{23}\log_2 n$

6.2]

```
Main()
{
  i=27
  while(i<n)
  {
    i=i^3; i=i^15
    i=i^5
  }
}
```

$27, 27^{15^1}, 27^{15^2}, 27^{15^3}, \ldots, 27^{15^k} = n$ $\quad k = \log_{15}\log_{27} n$

square $\rightleftharpoons$ (inverse) root

6.3]

```
Main()
{
  i=n
  while(i>=2)
  {
    i=√i
  }
}
```

$n, n^{1/2}, (n^{1/2})^{1/2}, n^{1/2^3}, n^{1/2^4}, \ldots, n^{1/2^k} = 2$

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

$\log n = 2^k$

$\log\log n = k$

$k = \log\log n$

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

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

$\log_2\log_2 n = k$

6.4]

```
Main()
{
  i=n
  while(i>=29)
  {
    i=i^1/2
    i=i^1/5
  }
}
```

$n, n^{1/10}, n^{1/10^2}, n^{1/10^3}, \ldots, n^{1/10^k} = 29$ $\quad k = \log_{10}\log_{29} n$

Ex:7

```
Main()
{
  for(i=1; i<=n; i=i+5)          — n/5
  {
    for(j=n; j>=1; j=j/10)       — log10 n
    {
      for(k=5; k<=n^3; k=k^10)   — log10 log5 n^3
      {
        x = y+z;
      }
    }
  }
}
```

$k = k \times 10$, $i = k+10$ — we can neglect these two terms

$T(n) \rightarrow \frac{n}{5} \times \log_{10} n \times \log_{10}\log_5 n^3$

$= \frac{1}{5} n \log_{10} n \times \log(3\log_n 5)$

How we have inner loops, & no dependency is there, so we can multiply their running time to get the effective running time of prg.

Ex: 8

```
Main ()
{   x=0
    for (i=1; i<=n; i=5*i)        — log5n
    {   j=1; x=0;                 i=2*i — log2n
        while (j<=n)              ——— log2n
        {
            j=2*j;
            x=x+n;
        }
    }
```

(log2n)2

x = n(log2n)2

x = n(log2n)

vi) Tc: (log5n × log2n)

Value of x: x = log2n + log2n + ... + log2n

log5n

x = n log2n + n log2n + ... + n log2n

log5n

x = (n log2n) × log5n

x = n(log5n log2n)

Ex: 9 Main ()

Ex: 9

```
Main ()
{   p=0, q=0                                  i) Tc
    for (i=1; i<=n; i=2*i) → log2n            ii) Value of q = ?
        p++;
    for (j=1; j<=p; j=2*j) → log2log2n
        q++;
}
```

larger loop we take

Sol: T(= log2n + log2log2n => O(log2n)

q = log2log2n p = log2n

Ex: 10

```
Main ()
{
    for (i=1; i<=n; i++) ——→ n
    {
        for (j=1; j<=i; j=j++) ——→
        {
            for (k=1; k<=n; k=5*k) → log5n
            {
                pf(hi);
            }
        }
    }
```

here we have dependency

i=1	i=2	i=3	i=4
j=1 time	j=2 time	j=3 time	j=4 time
k=log5n	k=log5n log5n	k=log5n log5n log5n	k=4 log5n time

Sol: i = 1 2 3 4 ---- n

j = 1 time 2 time 3 time 4 time ---- n times

k = log5n + log5n + log5n + log5n + ... + log5n

Scanned by CamScanner

$\log_5 n + 2\log_5 n + 3\log_5 n + 4\log_5 n + \cdots n\log_5 n$

$= \log_5 n \,(1+2+3+4+\cdots+n)$

$\log_5 n \left(\frac{n(n+1)}{2}\right)$

$= O(n^2 \log_5 n)$

Ex-11

```
Main()
{
  for(i=1; i<=n; i++)
  {
    j=1
    while(j<=n)
    {
      j=j+i;
    }
  }
}
```

Sol:- $i=1$: $j = \frac{n}{1}$ times; $i=2$: $j = \frac{n}{2}$ times; $i=3$: $j = \frac{n}{3}$ times; $i=4$: $j = \frac{n}{4}$ times; ..... $i=n$: $j = \frac{n}{n}$ times

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

$= O(n \log n)$ $\left(\because \log_2 n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}\right)$

Ex-12

```
Main()
{
  for(i=1; i<=n; i++)
  {
    if(n%i==0)
    {
      for(j=1; j<=n; j++)
      {
        pf("hi");
      }
    }
  }
}
```

where n is prime no
↳ no divisible by 1 & no itself.

Sol:- $i=1$: $j = n$ times; ..... $i=n$: $j = n$ times

$T(n) = 2n \rightarrow O(n)$

$i=1$	2	3	4	...	n	$\Rightarrow 2n + (n-2)$	constant $O(1)$
↓	↓	↓	↓		↓	↓	
n	$O(1)$	$O(1)$	$O(1)$		n	$2n + n = 3n$	
	↑ constant work bcz of comparison					↓	
						$O(n)$	

Ex 12.1

```
for(i=1; i<=n; i++)
{
  for(j=1; j<=n; j++)
  {
    pf("hi")
    (exit(0))
  }
}
```

$T(n) \rightarrow O(1)$

12.2

```
for(i=1; i<=n; i++)
{
  for(j=1; j<=n; j++)
  {
    pf("hi")
    (break;)
  }
}
```

$T(n) \rightarrow O(n)$

Ex: 13

```
Main()
{ int i;
  for(i=1; i≤n; i++)
  {
    for(i=1; i≤n²; i++)
    {
      for(i=1; i≤n³; i++)
      {
        x = y+2;
      }
    }
  }
}
```

i: 1, 2, 3, 4, ..., n^2

break in n^3 loop → n^3+1
break in n^2 loop → n^3+2
break in n loop → n^3+3

Sol: $T(n) \rightarrow n^3+3 \Rightarrow O(n^3)$

~~i=1~~
~~i=2~~
~~k=n³ time~~
if $i \le n^4$

13-17

```
Main()
{ int i;
  for(i=1; i≤n; i++)
  {
    for(i=1; i≤n⁴; i++)
    {
      for(i=1; i≤n³; i++)
      {
        x = y+2;
      }
    }
  }
}
```

i: 1, 2, 3

n^3
n^3+1 → break in n^3 loop
$(n^3+2 \le n^4) \Rightarrow$ True

1
2
3

n^3
n^3+1
$(n^3+2 \le n^4) \Rightarrow$ True

This loop runs infinite times

Ex: 14

```
Main()
{
  for(i=1; i²≤n; i++)          — n^1/2
  {
    for(j=1; j²⁰≤n⁵; j++)      — n^1/4
    {
      for(k=1; k⁷⁵≤n⁹⁰; k++)   — n^6/5
      {
        x = y+2;
      }
    }
  }
}
```

$n^{1/2} \cdot n^{1/4} \cdot n^{6/5}$
$n^{17/20}$

Sol:

$n=8$
$i=1$ $1^2 \le 8$ $2^2 \le 8$
$j=1$ $1^{20} \le 8$

$i^2 \le n$
$2^2 < n$
$3^2 < n$

$k^2 \le n$
$\log k \le n^{1/2}$
$k^{20} \le n^5$ X

~~$i \le k^2 \le n$~~

for i loop — break condition — Suppose at k^{th} Iteration — $k^2 = n$
loop break — $k = n^{1/2}$

for j loop — $k^{20} = n^5$
$k = n^{1/4}$

for k loop — $k^{75} = n^{90}$
$k = n^{6/5}$

$T(n) \rightarrow n^{1/2} \cdot n^{1/4} \cdot n^{6/5}$
$= O(n^{17/20})$

Ex.15

```
A(n)
{
  if (n ≤ 2)
      return (2);
  else
      return (A(√n))
}
```

n
$n^{1/2}$
$n^{1/2^2}$
$n^{1/2^3}$
⋮
$n^{1/2^k} = 2 \rightarrow \log\log n$

$T(n) \rightarrow O(\log_2(\log_2 n))$

Note:- For every recursive prg we can have, a non-recursive prg. and vice-versa.

27/7/18

Asymptotic Notations

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

Let $f(n)$ & $g(n)$ be two positive functions

1) Big Oh Notation (O)

$f(n) = O(g(n))$ $f(n)$ is order of $g(n)$

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

such that (there exists) $\exists$ 2 positive constants $c > 0$ (↓ speed performance difference from system to system) & $n_0 \geq 1$ → we need at least 1 i/p to analyse

Ex1 $f(n) = n^2 + n + 1$
$g(n) = n^2$

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

$f(n) \leq c \cdot g(n)$ $\forall n, n \geq n_0$

ie we try to make R.H.S greater by taking the help of constant c.

Concepts: take every term as biggest term

$n^2 + n + 1 \leq c \cdot n^2$ $\forall n, n \geq n_0$

$n^2 \; n^2 \; n^2 = 3n^2$

$\therefore c = 3$

$n^2 + n + 1 \leq 3n^2$ $\forall n, n \geq n_0$
$n_0 = 1$

Start analysing n_0 from $n=1$, one by one

$\therefore n^2 + n + 1 = O(3n^2)$ $\forall n, n \geq 1$ $c = 3$

↑ R.H.S greater after taking c's help

Ex.2 $f(n) = n$
$g(n) = n + 10$

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

$n \leq c \cdot n + 10$ $\forall n, n \geq n_0$

↓ already bigger than L.H.S

∴ put $c=1$

$n \le 1\cdot(n+10) \quad \forall n,\ n \ge n_0 \quad n_0 = 1$

$\therefore n = O(n+10) \quad \forall n,\ n \ge 1 \quad c=1$

Ex 3 $f(n) = n+10$

$g(n) = n-10$

$n+10 \le c\cdot(n-10) \quad \forall n,\ n \ge n_0$

$n \quad n = 2n \quad \therefore c = 2$

$n+10 \le 2\cdot(n-10) \quad \forall n,\ n \ge n_0 \quad n_0 = 30$

$n+10 = O(n-10) \quad \forall n,\ n \ge 30$

Ex 4

$f(n) = n^2$

$g(n) = n$

$f(n) \le c\cdot g(n) \quad \forall n,\ n \ge n_0$

$n^2 \le c\cdot n \quad \forall n,\ n \ge n_0$

$\therefore n^2 \ne O(n)$

Since c is a constant, n^2 will always be $\ge n \quad \forall n$.

2) Omega - Notation (Ω)

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

$f(n) \ge c\cdot g(n)$, $\forall n,\ n \ge n_0$

such that $\exists$ two +ve constants $c > 0$ & $n_0 \ge 1$

Ex 1 $f(n) = n \quad g(n) = n+10$

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

$n \ge c\cdot(n+10) \quad \forall n,\ n \ge n_0$

$1 \quad 10 \quad c = 20$ ✗

$n \ge 20\cdot(n+10) \quad \forall n,\ n \ge n_0$

how n is full → $1\cdot n \ge c\cdot(n+10) \quad \forall n,\ n \ge n_0$ ← take half n

$c \downarrow 1/2$

$\therefore n \ge \frac{1}{2}(n+10) \quad \forall n,\ n \ge n_0$ — start analysing from $n_0 = 1$

$n_0 = 10$

$n \ge \frac{1}{2}(n+10) \quad \forall n,\ n \ge 10$

$n = \Omega(n+10) \quad \forall n,\ n \ge 10,\ c = 1/2$

↑ smaller after taking c help

Ex: 2

$f(n) = n+10$

$g(n) = n-10$

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

$n+10 \geq (n-10) \quad \forall n, n \geq n_0 \qquad n_0 = 1$

(1 ↓ ; ↘ already smaller)

$n+10 = \Omega(n-10) \quad \forall n, n \geq 10 \quad c = 1.$

Ex: 3 $f(n) = n^2 \qquad g(n) = n^2+n+10$

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

$1 \cdot n^2 \geq c \cdot (n^2+n+10) \quad \forall n, n \geq n_0$

(c ↓ $1/2$)

$n^2 \geq \frac{1}{2}(n^2+n+10) \quad \forall n, n \geq 4 \qquad n_0 = 4$

$n^2 = \Omega(n^2+n+10) \quad \forall n, n \geq 4 \qquad c = 1/2$

Ex: 4 $f(n) = n \qquad g(n) = n^2$

$n \geq c \cdot (n^2) \quad \forall n, n \geq n_0$

(c ↓ $1/n$ ← but we can not take from n)

c must be Const.

$\therefore n \neq \Omega(n^2)$

Theta - Notation (Θ)

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

1) $f(n) \leq c_1 \cdot g(n)$ (&)
2) $f(n) \geq c_2 \cdot g(n)$

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

Ex: $f(n) = n^2$

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

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

$n^2 \leq c_1 \cdot g(n^2+n+10) \quad \forall n, n \geq n_0$

(c_1 ↓ 1) ($n_0 = 1$)

$n^2 = O(n^2+n+10) \quad \forall n, n \geq 1 \qquad c_1 = 1$

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

$f(n) \geq c_2 \cdot g(n)$

$1 \cdot n^2 \geq c_2(n^2+n+10) \quad \forall n, n \geq n_0$

(c_2 ↓ $1/2$)

$n^2 \geq \frac{1}{2}(n^2+n+10) \quad \forall n, n \geq 4$

$n^2 = \Omega(n^2+n+10) \quad \forall n, n \geq 4 \qquad c_2 = 1/2$

Hence $n^2 = O(n^2+n+10)$ $\forall n$ $n \geq 4$
$c_1 = 1$ & $c_2 = 1/2$
$n_0 = 4$

EX: $f(n) = n^2+10$
$g(n) = n^2-5$

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

$n^2+10 \leq c_1 \cdot (n^2-5)$ $\forall n$, $n \geq n_0$

n^2 $n^2 = 2n^2$ $c = 2$

$n^2+10 \leq 2 \cdot (n^2-5)$ $\forall n$, $n \geq n_0$
$n_0 = 5$

$n^2+10 = O(n^2-5)$ $c_1 = 2$ & $n_0 = 5$

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

$n^2+10 \geq c_2 \cdot (n^2-5)$ $\forall n$, $n \geq n_0$
$\downarrow 1$

$n^2+10 \geq 1/2 (n^2-5)$ $\forall n$, $n \geq 1$ $n_0 = 1$

$n^2+10 = \Omega(n^2-5)$ $c_2 = 1$ $n_0 = 3$

$n^2+10 = O(n^2-5)$ $c_1 = 2$ & $c_2 = 1$ $n_0 = 5$

Ex.3 $f(n) = n$ $g(n) = n^2$

Sol:

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

$n \leq c_1 \cdot n^2$

$\downarrow$

$n \leq 1 \cdot n^2$ $\forall n$, $n \geq n_0$ $n_0 = 1$

$n = O(n^2)$ $c_1 = 1$ $n_0 = 1$

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

$n \geq c_2 \cdot n^2$ $\forall n$, $n \geq n_0$

$\downarrow 1/n$ <- we cannot put it $1/n_0$ (c need to be a constant.

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

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

Ex.4 $f(n) = n$ $g(n) = n$

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

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

$n \leq c_1 \cdot (n)$ $\forall n$, $n \geq n_0$

$\downarrow$

$n \leq 1 \cdot n$ $\forall n$, $n \geq 1$

$n = O(n)$ $c_1 = 1$ $n_0 = 1$

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

$f(n) \geq c_2 \cdot g(n)$

$n \geq c_2 \cdot n$

$\downarrow$

$n \geq 1 \cdot n \quad \forall n \quad n \geq 1$

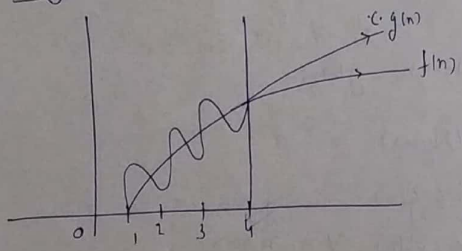
$n = \Omega(n)$

$\therefore n = O(n) \quad c_1 = 1 \quad c_2 = 1 \quad n_0 = 1$ → ie big forms are same

Note:- $a = O(b)$ then $a \& b$ are asymptotically equal

Mathematically equal means asymptotically equal

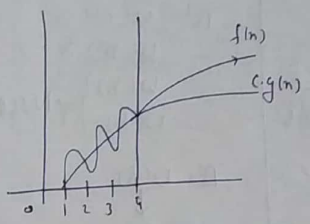
Big Oh Notation



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

$f(n) \leq c \cdot g(n) \quad \forall n, n \geq 4$

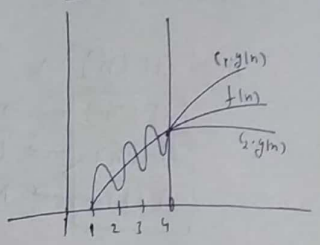
Omega - Notation



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

$f(n) \geq c \cdot g(n) \quad \forall n, n \geq 4$

Theta - Notation



$f(n) = \Theta(g(n)) \quad \forall n, n \geq 4$

$c_1 \& c_2 > 0$

Big oh Notation $(O, \leq)$

$n^2 = O(n)$ ×

$O(n^2)$ → TUB

$O(n^3)$ → Upper bound

$O(n^{10})$ → TNTUB

$A = O(B)$

$\Downarrow$

UB

T NT

TUB → Tightest Upper Bound

NTUB → Non-Tightest Upper Bound

Small Oh - Notation $(o, <)$

$n^2 = O(n)$ ×

$O(n^2)$ ×

$O(n^3)$ → NTUB

$O(n^{10})$ → NTUB

$A = o(B)$

$\downarrow$

NTUB

Omega - Notation (Ω, $\geq$)

$n^2 = \Omega(n^3)$ ✗

$\Omega(n^3)$ → TLB
$\Omega(n)$ → Lower bound
$\Omega(1)$ → NTLB

$A = \Omega(B)$
↓
LB
T NT

Small - omega (ω, $>$)

$n^2 = \omega(n^3)$ ✗
$\omega(n^2)$ ✗
$\omega(n)$
$\omega(1)$ → NTLB

$A = \omega(B)$
↓
NTLB

Theta notation

$n^5 = O(n^5)$ → TUB
$n^5 = \Omega(n^5)$ → TLB

$n^5 = \Theta(n^5)$ → TUB, TLB

$A = \Theta(B)$
↓
TUB TLB

8) Complexity Classes

1) Constant → $O(1)$
2) Logarithmic → $O(\log n)$
3) Linear → $O(n)$
4) Quadratic → $O(n^2)$
5) Cubic → $O(n^3)$
6) polynomial → $O(n^c)$ where c = constant $c > 0$
7) Exponential → $O(c^n)$ where c = constant $c > 1$
8) $2^n < n^n$, $n! < n^n$ → $2^n < n! < n^n$

$\log\log n$, $\sqrt{n}$, $\sqrt{\log n}$, $n\log\log n$, $n\log n$, $n\sqrt{n}$

28/7/18

9) $\log_2 n > \log_3 n$

$\frac{\log_2 n}{\log_2 3} = \frac{\log_2 n}{1.5}$

base ↑ log ↓
$n = 100$
$\log_2 100 \approx 7$
$\log_3 100 \approx 4$

$\log_2 n = O(\log_3 n)$ take $C = 2$
$\log_2 n = \Omega(\log_3 n)$
$\therefore \log_2 n = \Theta(\log_3 n)$

10) $2^n < 3^n$

$(2 \times 1.5)^n \rightarrow 2^n \cdot (1.5)^n$ ← greater by $(1.5)^n$

$2^n = O(3^n)$ $\quad 2^n \neq \Omega(3^n)$ ∴ equal not possible

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

$o \Rightarrow O$
Smallo $\Rightarrow$ bigoh
Subset $\Rightarrow$ superset

$n^2 = O(n^2) \not\Rightarrow n^2 = o(n^2)$
$n^2 = o(n^3) \Rightarrow n^2 = O(n^3)$

11) $n > \log n$ → Verify by taking log both the sides

$n > (\log_2 n)^2$ $\quad \log n > 2 \log_2 \log n$

$n > (\log_2 n)^5$

$n > (\log n)^{100,000}$

$n < (\log_2 n)^{\log_2 n}$ $\quad \log n \quad \log_2 n \log_2 \log n$

$1 < \log_2 \log n$

12) $2^n < n^n$

$2^n > n^{\log n}$ $\quad$ taking log both sides

$n \log_2 2 \quad \log n \log n$

$n > \log n \log n$

$n > (\log n)^2$

Ex1 (Crute)

Check the following statements are True or False

T a) $1000 n \log n = O\left(\frac{n \log n}{1000}\right)$

F b) $\sqrt{\log n} = O(\log \log n)$

T c) if $0 < x < y$ then

$n^x = O(n^y)$

F d) $2^n \neq O(n^c)$ Where c is constants $c > 0$

Sol:- a) ~~false~~ True ← constant c will take care of 1000 take $c = 10^6$

~~put $n = 2^{10}$ $\quad 1000 \times 2^{10} \times 1 \quad \frac{2^{10} \times 10}{1000}$~~

b) $\sqrt{\log n}$ $\quad \log \log n$ $\quad n = 2^{2^{10}}$ ~~2^{10}~~ $2^5 > 1$

false

c) $0 < x < y$

$n^x < O(n^y)$ ✓ $\quad \because x < y$

True

d) $2^n \neq O(n^c)$ $\quad c > 0$

put $n = 2^{10}$ $\quad 2^{2^{10}}$ $\quad (2^{10})^c$

2^{1024} $\quad (2^{10})^c$

log $n \log_2 2 \quad c \log n$

$n > c \log n$

Ex-2

Check T/F

T a) $(n+7)^b = O(n^b)$ where b = constants

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

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

d) $f(n) = O(f(\frac{n}{2}))$

Sol:- b) 2^{n+1} $O(2^n)$

$2^n \cdot 2 = O(2^n)$ diff by const.

c) 2^{2n} $= O(2^n)$ → $2^n \cdot 2^n$ also we can write

$(4)^n \leftarrow (2^2)^n \rightarrow O(2^n)$ diff by const. → $2^{2n} = \omega(2^n)$

a) $(n+7)^b = O(n^b)$

$(n+7)^b \geq c \cdot n^b$ diff by const. Counter example :

$n \to n$ $c = 2n$

F d) $f(n) = O(f(\frac{n}{2}))$ → fail for power forms

$f(n) \leq c \cdot f(\frac{n}{2})$

put $n = 2^{2n}$

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

$f(\frac{n}{2}) = 2^{2n/2} = 2^{n/2} = 2^n$

$f(n) > f(\frac{n}{2})$

Note: by applying log cancel common terms

n^2	n^3	2^{2n}	2^n
1	n	$2^n/2^n$	$2^n/2^n$
		2^n	1

a) $(n+7)^b = O(n^b)$

$\Downarrow$

$(b+1)\, n^b \Rightarrow \leq c \cdot n^b \Rightarrow O(n^b)$

$(n+7)^2 = n^2 + 7^2 + 2 \cdot n \cdot 7$

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

$\leq 3n^2$

$\leq c \cdot n^2 \quad (c=3)$

$(n+7)^b$ = will have $(b+1)$ terms, & can be expressed as $(b+1)\, n^b$ (c)

$(n+7)^b = O(n^b)$ $(c = b+1)$

$(n+7)^b = \Omega(n^b)$ $(c=1)$

$\therefore (n+7)^b = \Theta(n^b)$

Ex-3

Check T/F

F a) $n^5 \cdot 64^{\log_2 n} = O(n^{10})$

(concept: $a^{\log_c b} = b^{\log_c a}$)

F b) $n^5 \cdot 32^{\log_2 n}$ $64^{\log_4 n} \to n^3$ $= O(n^5)$

T c) $\frac{4^n}{2^n} = O(2^n)$

F d) If $f(n) = O(g(n))$ then $2^{f(n)} = O(2^{g(n)})$

Sol:- a) $n^5 \cdot 64^{\log_2 n} = O(n^{10})$

$n^5 \cdot 64^{\log_2 n} \leq c \cdot n^{10}$

$2^{2^{50}} \cdot (64)^{2^{10}} \quad c \cdot 2^{2^{100}}$

$2^{2^{50}} \cdot (64)^{1024} > c\, 2^{2^{100}}$

$n^5 \cdot 64^{\log_2 n} = n^5 \cdot n^{\log_2 64} = n^5 \cdot n^6 = n^{11}$

put $n = 2^{2^{10}}$

$n^{11} \neq O(n^{10})$

F

b) $32^{\log_2 n} \cdot 64^{\log_4 n} = O(n^9)$ ($n^5 \cdot n^3 = n^8$)

$\log_2 n \times 5 + \log_4 n \times 6 = 9 \log n$

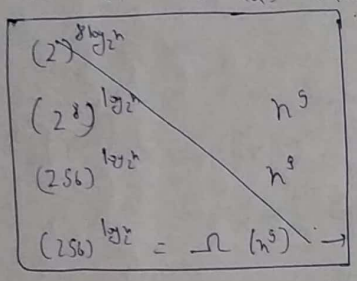
put $n = 2^{2^k}$

$5 \cdot 2^k + 6 \cdot 2^k \quad 9 \times 2^k$

$11 \cdot 2^k > 9 \times 2^k$

$32^{\log_2 n} \cdot 64^{\log_4 n} \Rightarrow (2)^{5\log_2 n} \cdot (2)^{6\log_4 n} = (2)^{5\log_2 n + 6\log_4 n}$

$3^{6\log_2 n}$



$32^{\log_2 n} \cdot 64^{\log_4 n}$

$n^5 \cdot n^3 = n^8$

$n^8 = O(n^9)$ ↓ false

T

c) $\frac{4^n}{2^n} = O(2^n)$ $\left(\frac{4}{2}\right)^n = 2^n$

$\frac{4^n}{2^n} \Rightarrow \frac{2^{2n}}{2^n} \Rightarrow 2^n = O(2^n)$ ✓

d) $f(n) = O(g(n))$ then $2^{f(n)} = O(2^{g(n)})$

$f(n) \le c_1 g(n)$ $2^{f(n)} \quad c_2 \cdot 2^{g(n)}$

$2^{c_1 g(n)} \quad c_2 \cdot 2^{g(n)}$

d). $f(n) = 2n \quad g(n) = n$ ← counter example

$2n = O(n)$

$2n \le c \cdot n$

↓ 2

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

$2^{2n} \le c \cdot 2^n$

$2^{n+n} \le c \cdot 2^n$

↓ 2^n ← not possible

$\therefore 2^{f(n)} \neq O(2^{g(n)})$

for Ω

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

$n = \Omega(2n)$

↓ $\frac{1}{2}$

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

$2^n \ge c \cdot 2^{2n}$

$2^n \ge c \cdot 2^{n+n}$

↓ ← not possible

$\frac{1}{2^n}$ possible

EX:4 Check T/F

T a) $\frac{1}{n} = O(1)$

F b) $\frac{1}{n} = \Omega(1)$

T c) $10000 = O(1)$

F d) $f(n) = O((f(n))^2)$

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

b) $\frac{1}{n} \ge c \cdot 1$

$\frac{1}{n}$ ↓ ← false not possible

Sol: a) $\frac{1}{n} \le c \cdot 1$ ↑ 1

↓ ~~no ← not possible~~

c) ~~$10000 = O(1)$~~

$10000 \le c \cdot 1$ ↳ 10000

$10000 = O(1)$ ✓

$10000 = \Omega(1)$ ✓

$10000 \ge c \cdot 1$ ↓ 1 ∴

d) $f(n) = O((f(n))^2)$

$f(n) \leq c \cdot (f(n))^2$

$f(n) = 2^n$

$f(n) = n/2$

$2n \leq c \cdot 2^{2n}$

$\frac{1}{2}$

false Condition

take $f(n) = \frac{1}{n}$ <- take decreasing fun (counter fun)

$\frac{1}{n} \quad c \cdot \left(\frac{1}{n}\right)^2 = c \cdot \frac{1}{n^2}$

$\therefore \frac{1}{n} \nleq c \cdot \frac{1}{n^2}$

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

- True when $f(n)$ is increasing fun
- False when $f(n)$ is decreasing fun

EX:5

~~check T/F~~

1) $f(n) = \log^*(\log n)$

2) $g(n) = \log(\log^* n)$

Relation b/w $f(n)$ & $g(n)$?

$\log_2^* 1000 = 4$ <- many times log applied → no of times log applied to reach reach termination point ie $\log_2 2 = 1$

~~$\log^* 2^{2^n} = n$~~

Sol: $n = 2^{2^{2^{\cdots^2}}}$ } 100000 times.

$\log 2^{2^{2^{\cdots^2}}} = 2^{99,999}$

$f(n) = \log^*(\log n) = \log^*(2^{99,999}) \Rightarrow 99,999$

$g(n) = \log(\log^* n) = \log 10^5 \approx 5 \log_2 10 \approx 5 \times 3 = 15$

$\log_2((\log_2 n)^*)$ ↓ ↑ $\log^* n = 100,000$

$\therefore f(n) \geq g(n) \Rightarrow \boxed{f(n) = \Omega(g(n))}$

$f(n) = \log^*(\log n)$ <- applied first will give big value

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

<- log applied last will decrease value a lot give small result

#Some More Examples

$f(n) = n-2$

$f^*(n) = n/2$ <- how many times we should apply $f(n)$ to get termination point

$f(n) = n/2 \quad f^*(n) = \log_2 n$

$f(n) = \sqrt{n} \quad f^*(n) = \log_2 \log_2 n$

Ex:6

$f(x) = x^{2+\sin x}$ relation b/w $f(x)$ & $g(x)$

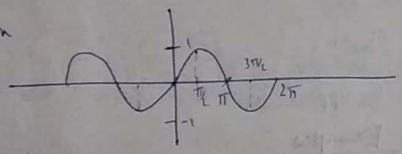
$g(x) = x^{\cos x}$

Sol: $f(x) = x^2 \cdot x^{\sin x}$ → $-1 \le \sin x \le 1$

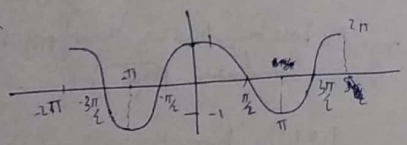
$g(x) = x^{\cos x}$ → $-1 \le \cos x \le 1$

~~$f(x) = \Omega g(x)$~~ ~~$f(x) > g(x)$~~

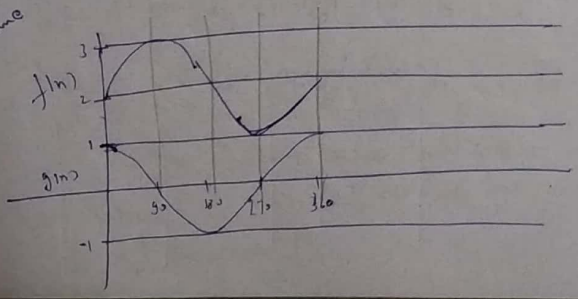
$\sin x$



$\cos x$



base is same



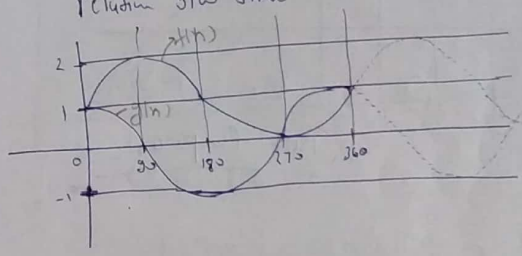
$\therefore f(x) > g(x) \Rightarrow f(x) = \Omega g(x)$

Ex:7

$f(x) = x^{1+\sin x}$ $g(x) = x^{\cos x}$

base is same

relation b/w them

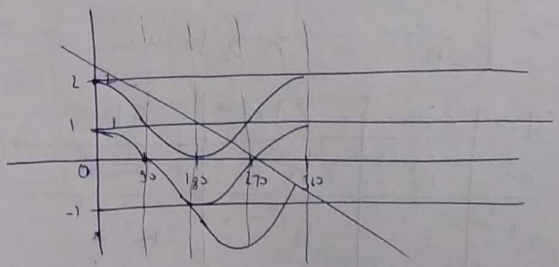


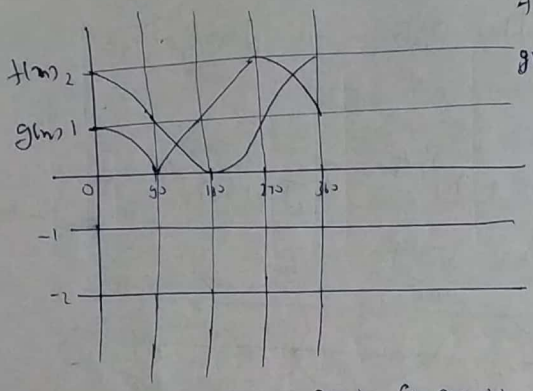
~~$f(x) = \Omega(g(x))$~~ not comparable

Ex:8

$f(x) = x^{1+\cos x}$

$g(x) = x^{1-\cos\sin x}$





$f(n) = 1 + (\sin n)$

$g(n) = n^{1-\sin n}$

not Comparable

Ex.9

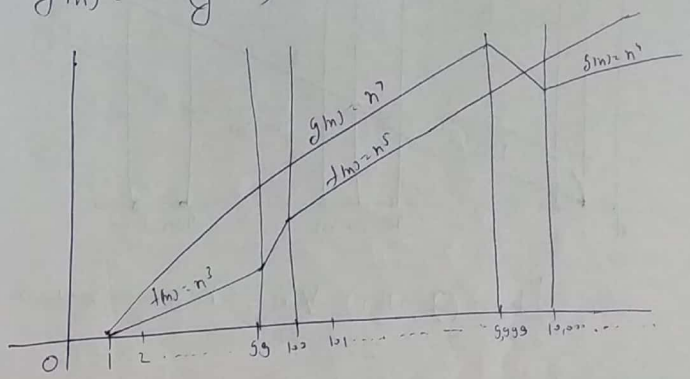
$$f(n) = \begin{cases} n^3 & 0 < n < 100 \\ n^5 & n \geq 100 \end{cases}$$

$$g(n) = \begin{cases} n^7 & 0 < n < 10000 \\ n^4 & n \geq 10000 \end{cases}$$

	<100	≤ 100	<10000	$\geq 10,000$
$f(n)$	n^3	n^5	n^5	n^5
$g(n)$	n^7	n^7	n^7	n^4

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

$f(n) = \Omega(g(n)) \quad \forall n, n \geq 10,000 \quad n_0 = 10,000$



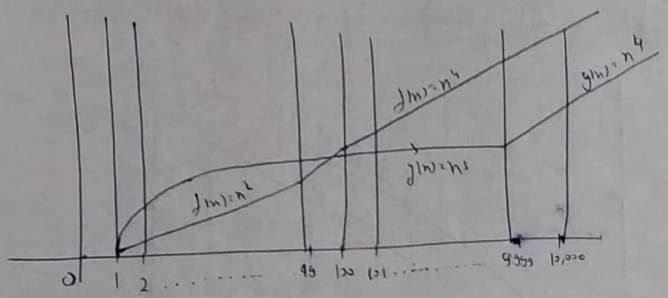
Ex.10

$$f(n) = \begin{cases} n^2 & 0 < n < 100 \\ n^4 & n \geq 100 \end{cases}$$

$$g(n) = \begin{cases} n^3 & 0 < n < 10000 \\ n^4 & n \geq 10000 \end{cases}$$

	<100	≥ 100	$<10,000$	$\geq 10,000$
$f(n)$	n^2	n^4	n^4	n^4
$g(n)$	n^3	n^3	n^3	n^4

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



$f(n) = O(g(n))$ $\forall n$, $n \geq n_0$ · $n_0 = 10{,}000$

$f(n) = \Omega(g(n))$ $\forall n$, $n \geq n_0$ $n_0 = 100$

· Properties of Asymptotic Notations

1) Reflexive Property (Mirror of each other) bcz of = equal

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

$\underbrace{O, \Omega, \Theta}$, o, ω
✓ ✗ ✗

2) Symmetric Property

If $f(n) = O(g(n))$ then $g(n) \neq O(f(n))$

Ex. $n^2 = O(n^3)$ but $n^3 \neq O(n^2)$

O ✓ $\quad \underbrace{O, \Omega, o, \omega}$ ✗

3) Transitive Property

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

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

Ex. $n^2 \leq n^3 \leq n^5$ then $\Rightarrow n^2 \leq n^5$ O

$n^5 \geq n^3 \geq n^2 \Rightarrow n^5 \geq n^2$ Ω

$n^2 < n^3 < n^5 \Rightarrow n^2 < n^5$ o

$n^5 > n^3 > n^2 \Rightarrow n^5 > n^2$ ω

$O, \Omega, \Theta, o, \omega$
✓ all pass.

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

then

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

Ex. $n^3 \leq n^5$

$n^6 \cdot n^3 \leq n^6 \cdot n^5$ ← still holds true $h(n)$ must be a +ve f^n

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

i) ~~$f(n) + d(n) = \max(g(n), e(n))$~~

or

$O(g(n) + e(n))$

ii) ~~$f(n) \cdot d(n) = O(g(n) \cdot e(n))$~~

Reason : When multiplied they might cross $g(n)$ & $e(n)$ individually

EX: 11

Consider the following 3 functions defined below

i) $f(n) = O(g(n))$ & & $g(n) \neq O(f(n))$

ii) $g(n) = O(h(n))$ & & $h(n) = O(g(n))$

then check T/F

$f(n) < g(n)$

equality condition removed here

$g(n) = h(n)$

$f(n) < g(n) = h(n)$

We will take max of $\{f(n), g(n)\}$

T i) $f(n) + g(n) = O(h(n))$

F ii) $f(n) \cdot g(n) = O(g(n) \cdot h(n))$

T iii) $g(n) \cdot h(n) = O(h(n) \cdot h(n))$

T iv) $g(n) + f(n) = \Omega(h(n))$

max $\{g(n), f(n)\}$

Sol: $g(n) = O(h(n)) \cdot h(n) = O(g(n))$

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

i) $f(n) + g(n) = O(h(n))$

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

$f(n) + g(n)$

EX-12

Consider the funs defined below

$T_1(n) = O(f(n))$ & & $T_2(n) = O(f(n))$

then check T/F

T a) $T_1(n) + T_2(n) = O(f(n))$

F b) $T_1(n) = O(T_2(n))$

F c) $T_1(n) = \Omega(T_2(n))$

F d) $T_1(n) = \Theta(T_2(n))$

Sol: $T_1(n) \leq O(f(n))$ $T_2(n) \leq O(f(n))$

Divide & Conquer (Basics)

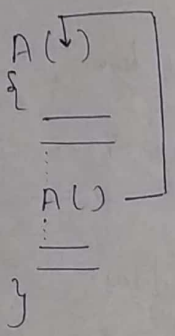
(Basics)

1) Recursion
2) Recurrence relation
3) Recurrence relation solving

Recursion

→ A fun calling itself to solve a particular problem is called recursion

Ex:)



tail recursion
Non-tail recursion
nested recursion
excessive recursion

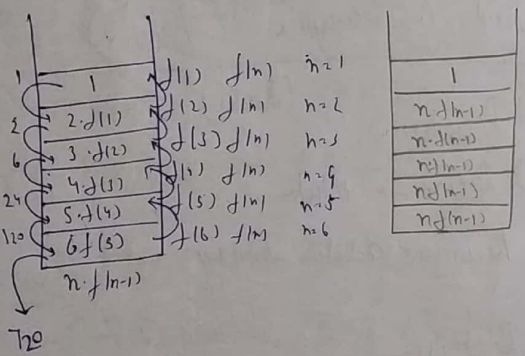
Ex2: f(6) = 6*f(5)
5*f(4)
4*f(3)
3*f(2)
2*f(1)

solving big problems in terms of smaller prob

f(1) = 1
f(n) = factorial of n

2) Recursion is nothing but solving the big problem in terms of smaller problems.

3) → To execute the recursive prg we are using stack data structure.



4) Every recursive prog should have termination condition else we will get error message "Stack Overflow" during "Run Time"

5) In recursion from 1 fun call to another fun call parameter value will change but not no of parameters & names of the parameters.

6) For every recursive prg equivalent non-recursive prg is possible. (but we have to use for loop or while loop)

7) Comparing recursion & non-recursion, recursion will take more stack space bcz of more fun calls.

8) Time complexity of the prg based on the logic but on recursion & non-recursion.

9) Recursive prgs are preferred by user, while non-recursive preferred by the computer.

10) Recurrence Relation

$$f(n) = \begin{cases} 1 & \text{if } n \le 1 \\ n*f(n-1) & \text{if } n > 1 \end{cases}$$

For calculating n!

Recursive prg

```
f(n)
{
  if (n ≤ 1)
    return (1)
  else
  return n*f(n-1)
}
```

TC: O(n)
SC: i/p + extra
= 2B + 8B(n)B
= O(n)

Iterative prg

```
main()
{
  S=1
  for (i=1; i ≤ n; i++)
      S = S*i
  return S;
}
```

EX 2 Write a recursive prg & recurrance relation to multiply 2 positive numbers m & n. (Where m,n>0)

Sol: mul(m,n) = $\underbrace{m+m+m+\dots+m}_{n \text{ times}}$ (or) $\underbrace{n+n+\dots+n}_{m \text{ times}}$

mul(3,4) = 3 + mul(3,3)
↳ 3 + mul(3,2)
↳ 3 + mul(3,1)
↳ 3 + mul(3,0)
return 0

mul(m,n) TC: → O(n)

```
{
  if (m==0 || n==0)
      return 0;
  else
      return (m + mul(m,n-1));
}
```

Recursive Eqn

$$mul(m,n) = \begin{cases} 0 & \text{if } m==0 \text{ or } n==0 \\ m + mul(m,n-1) & \text{otherwise} \end{cases}$$

no of fun calls → n+1

Stack size → n+1

Space Complexity

i/p + extra space

for mul(m,n)

S: 2 integers + stack
= 4B + (n+1)B ⇒ (n+4)B
= O(n)

EX 3 Write a recursive prg & recurrance relation to find n^{th} fibonnacci number.

n	0	1	2	3	4	5	6	7	8	9	10
f(n)	0	1	1	2	3	5	8	13	21	34	55

Sol Recurrance relation

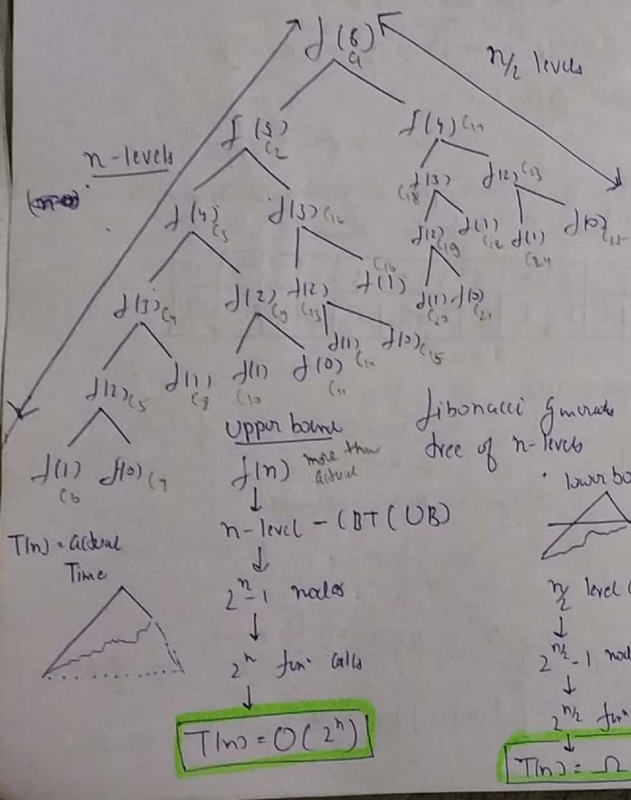
$$f(n) = \begin{cases} 0 & \text{if } n==0 \\ 1 & \text{if } n==1 \\ f(n-1)+f(n-2) & \text{otherwise} \end{cases}$$

(or)

$$f(n) = \begin{cases} n & \text{if } n==0 \text{ || } n==1 \\ f(n-1)+f(n-2) & \text{if } n>1 \end{cases}$$

```
f(n)
{
  if (n==0 || n==1)
      return n;
  else
      return f(n-1)+f(n-2);
}
```

Empty Binary tree.



Fibonacci generate binary tree of n-levels

Upper bound (more than actual)

$f(n)$ → n-level CBT (UB) → $2^n - 1$ nodes → 2^n fun calls

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

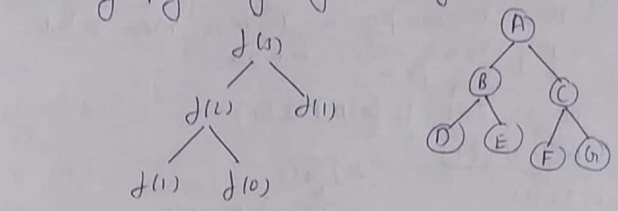
Lower bound (LB) (less than actual)

$\frac{n}{2}$ level CBT → $2^{n/2} - 1$ nodes → $2^{n/2}$ fun calls

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

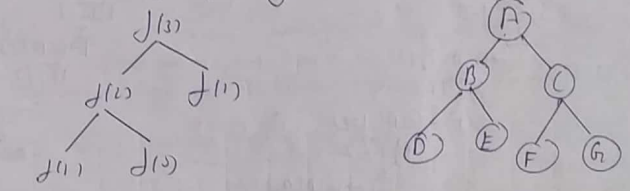
$T(n)$ = Actual Time

In every programming lang fun calling sequence is preorder



Preorder → f(3) f(2) f(1) f(0) f(1) → ABDECFG

In every programming lang. fun execution order is Postorder



→ f(1), f(0), f(2), f(1) f(3) → DEB FG CA

→ Stack Space: is number of levels (bcz all siblings will occupy same stack slot)

Stack space = O(n)

Space = i/p + extra

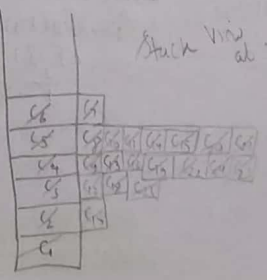
= 2B + n-levels

actual time → ~ O(n)

Best Case: $T(n) = \Omega(2^{n/2})$

Worst Case: $T(n) = O(2^n)$

Avg Case: T(n) =



28/7/2021

EX Write a recursive prg & recurrence relation to find GCD of (m,n) where m,n > 0

Terminating condition

GCD(0,20) = 20
GCD(20,0) = 20

i) GCD(23,59)

23 | 59 | 2
46
13 → GCD(13,23)

13 | 23 | 1
13
10 → GCD(10,13)

10 | 13 | 1
10
3 → GCD(3,10)

3 | 10 | 3
9
1 → GCD(1,3)

1 | 3 | 3
3
0 → GCD(0,1)
↓ return 1

ii) GCD(23,97)

23 | 97 | 4
92
5 → GCD(5,23)

5 | 23 | 4
20
3 → GCD(3,5)

3 | 5 | 1
3
2 → GCD(2,3)

2 | 3 | 1
2
1 → GCD(1,2)

1 | 2 | 2
2
0 → GCD(0,1)
↑ return 1

iii) GCD(43,17)

17 | 43 | 2
34
9 → GCD(9,17)

9 | 17 | 1
9
8 → GCD(8,9)

8 | 9 | 1
8
1 → GCD(1,8)

1 | 8 | 8
8
0 → GCD(0,1)
↓ return 1

43 | 17 | 0
0
17 → GCD(17,43)

GCD(1000, 50,000) :- best case both m,n are multiple of each other

1000 | 50,000 | 50
50000
0 → GCD(0, 1000) ← return 1000

```
GCD(m,n)
{
    if (m == 0)
        return n;
    if (n == 0)
        return m;
    else
        return GCD(n%m, m);
}
```

recurrence relation

$$GCD(m,n) = \begin{cases} n & \text{if } m = 0 \\ m & \text{if } n = 0 \\ GCD(n \bmod m, m) & \text{otherwise} \end{cases}$$

→ min amount of time required to solve a problem

Best Case :- m,n are multiple of each other TC :→ O(1)

→ most amount of time required

Worst Case :- m,n are relatively prime TC :→ O(log n)

Average Case :- behaviour most of the time TC :→ O(log n)

Worst Case = Avg Case

Best Case ≤ Average Case ≤ Worst Case

NOTE: We can write upper bound [UB] for both Best Case & Worst Case in case we are unable to express actual time complexity

Imp (Greate)

Avg Case → A(n)
Worst Case → W(n)
Best Case → B(n)

A(n) = O(W(n))
A(n) = Ω(B(n))

Recurrence Relation Solving

1) Substitution Method
2) Recursive Tree Method
3) Master Theorem

Substitution Method

NOTE: Substituting the given func repeatedly until given func removed is known as substitution method

Ex1: $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + n & \text{if } n > 1 \end{cases}$

Sol:
$$T(n) = T(n-1) + n$$
$$\Downarrow$$
$$T(n-2) + n-1 + n$$
$$\Downarrow$$
$$T(n-3) + n-2 + n-1 + n$$
$$\vdots$$
$$T(n-k) + n-(k-1) + n-(k-2) + \dots \quad n-1+n$$

put $n-k=1 \quad k=n-1$

$$T(1) + n-(n-2) + n-(n-3) + \dots \quad n-1+n$$
$$1 + 2 + 3 + 4 + \dots + n$$
$$= \frac{n(n+1)}{2}$$
$$= O(n^2)$$

Ex2: $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) \times n & \text{if } n > 1 \end{cases}$

Sol:
$$T(n) = T(n-1) \times n$$
$$= T(n-2) \times n-1 \times n$$
$$T(n-3) \times n-2 \times (n-1) \times n$$
$$\vdots$$
$$T(n-k) \times (n-(k-1)) \times (n-(k-2)) \times \dots \quad n$$

put $n-k=1$

$$T(1) \times (n-(n-2)) \cdot 3 \cdot 4 \dots n \qquad n-1=k$$
$$1 \times (k-n+2) \cdot 3 \cdot 4 \dots n$$
$$T(n) = n!$$

$\boxed{T(n) = O(n!)}$ Or if $n!$ not in option take closest upper bound ie $O(n^n)$ or closest lower bound ie $\Omega(2^n)$

$n! = O(n^n)$
$n! = \Omega(2^n)$

Ex3: $T(n) = \begin{cases} 1 & \text{if } n = 0 \\ T(n-1) + \log n & \text{if } n > 1 \end{cases}$

Here we will get series

$\Rightarrow \log 1 + \log 2 + \log 3 + \dots + \log n$

$\Rightarrow \log(1 \cdot 2 \cdot 3 \dots n)$

$\Rightarrow \log(n!)$ or $\Rightarrow \log(n^n) \Rightarrow n \log(n)$

Ex 4

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ T(n-2) + \log_2(n) & \text{if } n > 0 \end{cases}$$

Sol: $T(n) = T(n-2) + \log_2 n$

$= T(n-3) + \log_2(n-1) + \log_2 n$

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

$= T(n-k) + \log_2(n-(k-1)) + \log_2(n-(k-2)) + \dots + \log_2 n$

$=$ put $n-k = 0$, $n = k$

$T(0) + \log_2 2 + \log_2 3 + \log_2 4 + \dots + \log_2 n$

$= 1 + \log_2 2 + \log_2 3 + \dots + \log_2 n$

$= 1 + \log_2(2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \dots n)$

$= 1 + \log_2 n!$

$= 1 + \log_2 n^n$

$= 1 + n \log n$

$= O(n \log n)$

$T(n) = T(n-2) + \log_2 n$ — 1 time, 2^1

$= T(n-4) + \log(n-2) + \log n$ — 2 times, 2^2

$= T(n-6) + \log(n-4) + \log(n-2) + \log n$ — 3 times, 2^3

$= T(n-2k) + \log(n-(2k-2)) + \log(n-(2k-4)) + \dots + \log n$

$=$ put $n-2k = 0$, $\frac{n}{2} = k$

$= T(0) + \log(2) + \log(4) + \log(6) + \log(8) + \dots + \log n$

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

$= T(0) + \log(2) + \log(4) + \log(6) + \dots + \log(n)$

$= 1 + \log(2 \cdot 4 \cdot 6 \cdot 8 \dots n)$

$= 1 + \log\left(2 \cdot 1 * 2 \cdot 2 \times 2 \cdot 3 \times 2 \cdot 4 \times \dots \times 2 \times \frac{n}{2}\right)$ — $\frac{n}{2}$ terms

$= 1 + \log(2 \cdot 1) + \log(2 \cdot 2) + \log(2 \cdot 3) + \dots \log\left(2 \times \frac{n}{2}\right)$

$= 1 + (\log 2 + \log 1) + (\log 2 + \log 2) + (\log 2 + \log 3) + \dots (\log 2 + \log \frac{n}{2})$

$= 1 + \left[\frac{n}{2}\log_2 2 + [\log 1 + \log 2 + \log 3 + \dots + \log \frac{n}{2}]\right]$

$= 1 + \left\{\frac{n}{2} \times 1 + [\log(1 \cdot 2 \cdot 3 \cdot 4 \dots \frac{n}{2})]\right\}$

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

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

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

Ex5

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

Sol. $T(n) = T(n-2) + n^2$ (2.1 → 1 time)

$= T(n-4) + (n-2)^2 + n^2$ (2.2 → 2 times)

$= T(n-6) + (n-4)^2 + (n-2)^2 + n^2$ (2.3 → 3 times)

$\vdots$

$T(n-2k) + (n-(2k-2))^2 + (n-(2k-4))^2 + \dots + n^2$ (k times)

put $n-2k = 0$

$k = n/2$

$T(0) + (2)^2 + (4)^2 + (6)^2 + \dots + n^2$ $\quad [(a+b)^2 = a^2+b^2]$

$= 1 + (2)^2 + (4)^2 + (6)^2 + \dots + n^2$

$= 1 + [(2\cdot1)^2 + (2\cdot2)^2 + (2\cdot3)^2 + \dots + (2\cdot\frac{n}{2})^2]$

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

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

$= 1 + 2^2\left(\frac{n(n+2)(n+1)}{24}\right)$

$= O(n^3)$

Ex6

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

Sol: $T(n) = T(n/2) + n$

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

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

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

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

$n = 2^k$

$\log n = k$

decreasing GP sum can not be > 1

$T(1) + n\left(\frac{1}{2^{k-1}} + \frac{1}{2^{k-2}} + \dots + \frac{1}{2} + 1\right)$ (very small, we neglect this)

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

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

$1 + 2n - (n)2^{1-k}$ $\qquad 1 - \frac{1}{2^{\log_2 n}}$

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

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

Concept: $k = \log n$ ← decreasing G.P series

$T(1) + n\left(\frac{1}{2^{\log n-1}} + \frac{1}{2^{\log n-2}} + \cdots + \frac{1}{2} + 1\right)$ → 0.5

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

$1+n$ [2] or $1+n$ [O(1)]

1 0.5 1 $O(1)$ $(\because n < 1)$

~~this value can not be~~ this is decreasing GP series, it's value is $O(1)$ (2)

$= 1 + n * O(1)$

$= O(n)$

EX7

$T(n) = \begin{cases} 10 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$

Imp: keep brackets

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

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

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

$\vdots$

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

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

$n = 2^k$

$k = \log_2 n$

$2^{\log n} T(1) + 2^{\log n-1} \cdot \frac{n}{2^{\log n-1}} + 2^{\log n-2} \cdot \frac{n}{2^{\log n-2}} + \cdots + 2 \times \frac{n}{2} + n \times 2^0$

$2^{\log n}(10) + \underbrace{n + n + \cdots + n}_{\log n \text{ times}}$

$= 2^{\log n}(10) + n(\log n)$

$= 10n + n(\log n)$

$= O(n \log n)$ take $c = 2$ O possible; take $c = 1$ Ω possible; hence Θ possible

EX8

$T(n) = \begin{cases} 5 & \text{if } n = 1 \\ T(n/2) + c & \text{where } c \text{ is const} \end{cases}$

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

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

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

$\vdots$

$= T\left(\frac{n}{2^k}\right) + \underbrace{c + c + c + \cdots + c}_{k \text{ times}}$ put $\frac{n}{2^k} = 1$, $k = \log n$

$= T(1) + c \log n \Rightarrow O(\log n)$

Ex 4
Imp

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

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

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

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

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

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

$\vdots$

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

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

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

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

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

$= n + n\left[\log n^{\log n} - \log 2^{n-1}\right]$

$= n + n\left[\log n \log n - (n-1)\log 2\right]$

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

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

$= n + n\left[\log\frac{n}{2^{\log n-1}} + \log\frac{n}{2^{\log n-2}} + \cdots + \log\frac{n}{2^0}\right]$

$n + n\left[\log n - \log 2^{\log n-1} + \log n - \log 2^{\log n-2} + \cdots + \log n - \log 2^0\right]$

$n + n\left[\log n - (\log n - 1) + \log n - (\log n - 2) + \cdots + \log n\right]$

$n + n\left[1 + 2 + 3 + \cdots + \log n\right]$

$n + n\,\frac{\log n(\log n+1)}{2}$

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

Ex. 10 (Imp) V V Imp

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

Sol: $T(n) = \sqrt{n}\, T(\sqrt{n}) + n$

~~1~~

~~$\sqrt{n}\,(\sqrt{n}\,T(n^{1/2^2}) + (n)^{1/2}) + n$~~

(n)

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

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

$n^{1/2^3}\left[(n)^{1/2^3} T(n^{1/2^3}) + n^{1/2^2}\right] + n + n$

$n^{1/2^6}\, T(n^{1/2^3}) + n^{1/2}$

put $n = t^2$

$T(t^2) = t\, T(t) + t^2$

t

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

$(n)^{1/2}\left[(n)^{1/2^2} T(n^{1/2^2}) + (n)^{1/2}\right] + n$

$= n^{3/2^2}\, T(n^{1/2^2}) + n^{1/2} + n$ (2 times, 2^{2-1})

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

$= n^{7/2^3}\, T(n^{1/2^3}) + n + n + n$

$= n^{\frac{2^3-1}{2^3}}\, T(n^{1/2^3}) + n + n + n$ (3 times)

$= n^{2^k/2^k}\, T(n^{1/2^k}) + nk$

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

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

$\log\log n = k$

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

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

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

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

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

$= \frac{n}{n^{(1/2\log\log n)}}$

$= \frac{n}{n^{(1/\log_2 n)}}$

$= \frac{n}{n^{\log_n 2}}$ $\left(\because \log_b a = \frac{1}{\log_a b}\right)$

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

$= \frac{n}{2}$

Ex: 11.

$T(n) = \begin{cases} \\ \\ \end{cases}$ ~~$T(n) = 2T$~~

$T(n) = 1 \cdot 2^1 + 2 \cdot 2^2 + 3 \cdot 2^3 + \ldots + (n-1) \times 2^{n-1} + n \cdot 2^n$

$T_n = n \cdot \frac{2(2^n - 1)}{2-1}$

$T_n = n \times \frac{2(2^n - 1)}{1}$

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

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

$=$

$T(n) = 1 \cdot 2^1 + 2 \cdot 2^2 + 3 \cdot 2^3 + \ldots (n-1) \times 2^{n-1} \times n \cdot 2^n$ — ①

AP & GP ⇒ GP

Steps

1) find r of GP
2) multiply r by series
3) substract to eliminate AP

$2T(n) = 1 \cdot 2^2 + 2 \cdot 2^3 + 3 \cdot 2^4 + \ldots (n-1) 2^n + n \cdot 2^{n+1}$ — ②

① − ②

~~$T(n) =$~~

~~$T(n) = T(n) = 1 \cdot 2^1 +$~~

$T(n) = 1 \cdot 2^1 + 2 \cdot 2^2 + 3 \cdot 2^3 + \ldots + (n-1) \times 2^{n-1} + n \cdot 2^n$

$2T(n) = 0 + 1 \cdot 2^2 + 2 \cdot 2^3 + 3 \cdot 2^4 + \ldots + (n-2) \times 2^{n-1} + (n-1) 2^n + n 2^{n+1}$

$T(n) - 2T(n) = [1 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + \ldots + 1 \cdot 2^n] - n 2^{n+1}$

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

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

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

Recursive Tree Method

Note: $T(n) = 2T(\frac{n}{2}) + n \rightarrow n \log n$

irrespective of ceil or floor answer will differ by constant

floor $2T(\lfloor \frac{n}{2} \rfloor) + n \rightarrow n \log n$

ceil $2T(\lceil \frac{n}{2} \rceil) + n \rightarrow n \log n$

$\lceil \frac{n}{2} \rceil$ or $\lfloor \frac{n}{2} \rfloor \Rightarrow \frac{n}{2}$ ↑ We can write as

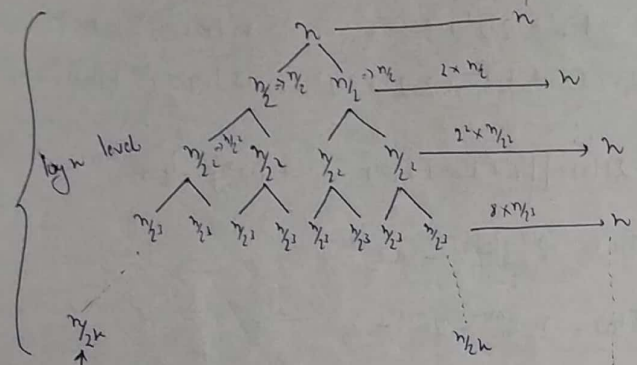
bz there will be a difference of constant.

1 - fun call ← use Substitution

more than 1 fun call (ie fun call ≥ 2) ← use Tree Method

Ex1 $T(n) = T(n/2) + T(n/2) + n$

time taken (or) work done to divide n into n/2

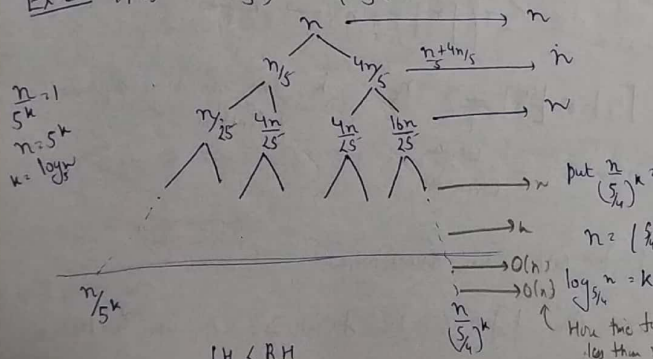


$n/2^k = 1 \Rightarrow n = 2^k \Rightarrow k = \log n$

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

Work done at each level is n
there are "log n" levels
$\therefore$

Ex2 $T(n) = T(n/5) + T(4n/5) + n$



$n/5^k = 1$
$n = 5^k$
$k = \log_5 n$

put $n/(5/4)^k = 1$
$n = (5/4)^k$
$\log_{5/4} n = k$

Here the time will be less than n bcz LH strictly only right elements taken

LH < RH

Left height (LH) = $\log_5 n$ < Right height (RH) = $\log_{5/4} n$

$T(n) = O(n \log_{5/4} n)$ or $T(n) = \Omega(n \log_5 n)$

$T(n) = O(n \log_{4/5} n)$

$\log_{5/4} n > \log_5 n$ — they differ by constant

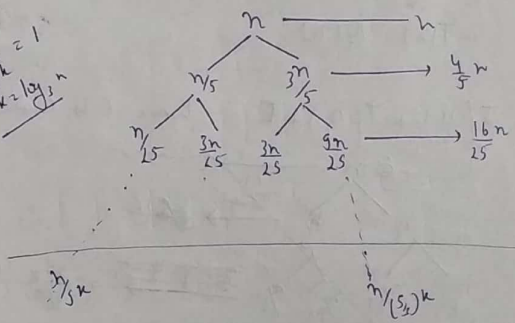
Ex3 $T(n) = T(n/10) + T(9n/10) + n$

$T(n) = O(n \log_{10/9} n)$ $T(n) = \Omega(n \log_{10} n)$

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

$\log_{10/9} n > \log_{10} n$ — they diff a by constant

Ex4 $T(n) = T(n/5) + T(3n/5) + n$



$n/5^k = 1$
$k = \log_5 n$

$n/(5/3)^k = 1$
$k = \log_{5/3} n$

Time taken = Left height $n(1 + 4/5 + (4/5)^2 + (3/5)^3 + \ldots \log_5 n$ times$)$

$= n\left(\frac{1(1-(4/5)^{\log_5 n})}{1/5}\right)$

$= n\left(5(1-(4/5)^{\log_5 n})\right)$

Left Height

decreasing GP

$T(n) \leq n\left\{\underbrace{\left(\frac{1}{3}\right)^0+\left(\frac{1}{3}\right)^1+\left(\frac{1}{5}\right)^2+\cdots+\left(\frac{1}{5}\right)^{\log_5 n}}_{O(1)}\right\}$

$T(n) \leq O(n)$

Right Height

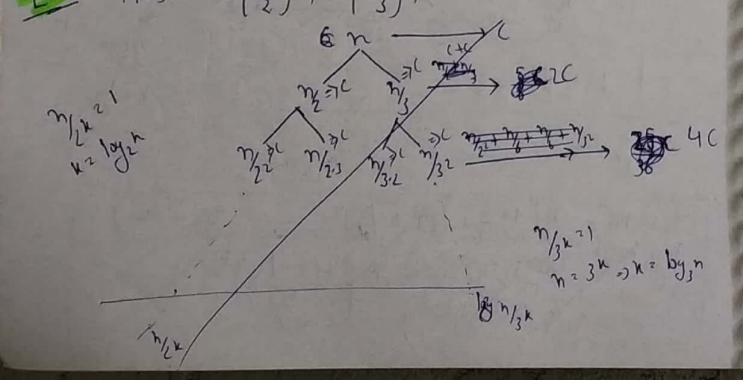
$T(n) \geq n\left\{\underbrace{\left(\frac{1}{3}\right)^0+\left(\frac{1}{3}\right)^1+\left(\frac{1}{5}\right)^2+\cdots+\left(\frac{1}{5}\right)^{\log_5 n}}_{O(1)}\right\}$

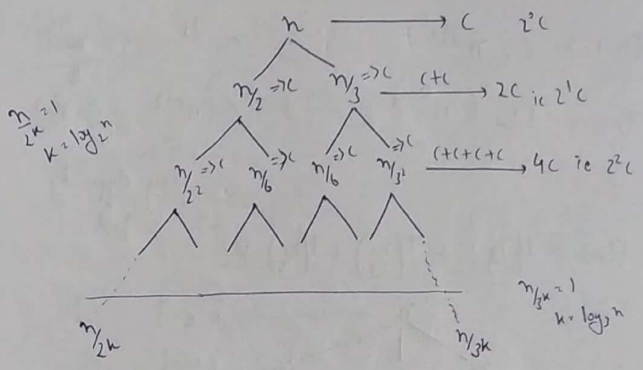
$T(n) \geq n$

$\therefore T(n) = O(n) \quad T(n) = \Omega(n)$

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

EX'S $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{3}\right) + C$ where C is constant





~~$T(n) \leq C + 2C + 4C + \cdots$~~

$T(n) \leq 2^0 \cdot C + 2^1 C + 2^2 C + \cdots + 2^{\log_2 n} \cdot C$

$T(n) \leq C\left(2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^{\log_2 n}\right)$

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

$T(n) \leq C\left[\frac{\left(2 \cdot 2^{\log_2 n}-1\right)}{1}\right]$

$T(n) \leq C\,[2n-1]$

$T(n) \leq O(n)$

$T(n) \geq 2^0 C + 2^1 C + 2^2 C + \cdots + 2^{\log_3 n} \cdot C$

$T(n) \geq C\left[\frac{1\left(2^{\log_3 n}-1\right)}{2-1}\right]$

$T(n) \geq C\left(2^{\log_3 n}\right)$

$T(n) \geq c(n^{\log_3 2})$

$T(n) = \Omega(n^{0.6})$

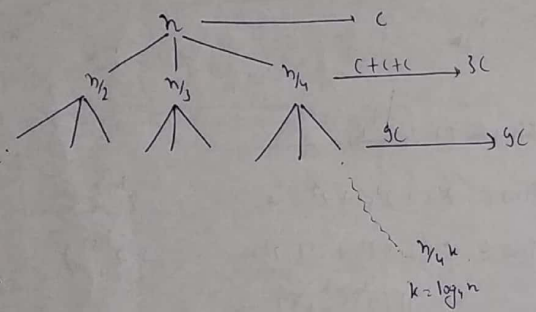
$n^{\log_3 2}$ $\quad$ $n^{\log_2 2}$

$\Downarrow$ $\quad$ $\Downarrow$

$O(n^{0.6})$ $\quad$ $O(n^1)$

they differ by a few

Ex6 $T(n) = T(n/2) + T(n/3) + T(n/4) + c$



Upper bound

$T(n) \geq 3^0 \cdot c + 3^1 \cdot c + 3^2 \cdot c + \dots + 3^{\log_2 n} \cdot c$

$\geq c[3^0 + 3^1 + 3^2 + \dots + 3^{\log_2 n}]$

$\geq c\left[1\left(\frac{3^{\log_2 n} - 1}{3-1}\right)\right]$

$T(n) \geq \frac{c}{2} \cdot (n^{\log_2 3})$

$T(n) = \Omega(n^{1.5})$

~~T(n)~~

Lower bound

$T(n) \leq 3^0 \cdot c + 3^1 \cdot c + 3^2 \cdot c + \dots + 3^{\log_4 n} \cdot c$

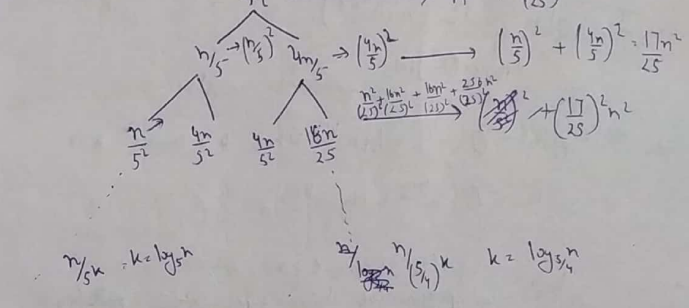
$T(n) \leq c[3^0 + 3^1 + 3^2 + \dots + 3^{\log_4 n}]$

$\leq c\left[1\left(\frac{3^{\log_4 n} - 1}{3-1}\right)\right]$

$\leq c(3^{\log_4 n}) \leq c \cdot (n^{\log_4 3})$

$T(n) = \Omega(n^{\log_4 3})$

Ex7 $T(n) = T(n/5) + T\left(\frac{4n}{5}\right) + n^2 \Rightarrow O(n^2)$



$T(n) \geq n^2\left[\underbrace{\left(\frac{17}{25}\right)^0 + \left(\frac{17}{25}\right)^1 + \left(\frac{17}{25}\right)^2 + \dots + \left(\frac{17}{25}\right)^{\log_{5/4} n}}_{O(1)}\right]$

$T(n) \geq O(n^2)$

$T(n) \leq n^2\left[\underbrace{\left(\frac{17}{25}\right)^0 + \left(\frac{17}{25}\right)^1 + \dots + \left(\frac{17}{25}\right)^{\log_4 n}}_{O(1)}\right]$

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

30/7/18

Masters Theorem (Refer Cormen in case of doubt)

$T(n) = aT(\frac{n}{b}) + f(n)$ Where $a \geq 1, b > 1$ are +ve constants & $f(n)$ is +ve function minimum

$n^{\log_b a}$ → greater by $\frac{n^{\log_b a}}{n^\epsilon}$ polynomial n^ϵ where ϵ is constant $\epsilon > 0$

Case I: $f(n) = O(n^{\log_b a - \epsilon})$

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

Case II: $f(n) = \Omega(n^{\log_b a + \epsilon})$ → $n^{\log_b a} \times n^\epsilon$ → smaller by minimum polynomial n^ϵ

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

Case III: $f(n) = \Theta(n^{\log_b a} * (\log n)^k)$ where k is constant. $k \geq 0$ ← smaller by logarithmic

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

Case I: $n^{\log_b a}$ is polynomial thrice greater than $f(n)$ (n^ϵ) where $\epsilon > 0$

Case 2: $n^{\log_b a}$ is polynomial thrice smaller than $f(n)$ (n^ϵ) where $\epsilon > 0$

Case 3: $n^{\log_b a}$ & $f(n)$ are Asymptotically equal.

EX1

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

$a = 8 \quad b = 2$

$f(n) = n^2$ | $n^{\log_b a} = n^{\log_2 8} = n^3$ → greater by polynomial

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

EX2

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

$a = 2 \quad b = 2$

$f(n) = n^2$ | $n^{\log_b a} = n$ → smaller by polynomial

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

EX3

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

$a = 2, \quad b = 2$

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

both are equal ⇓ $n \cdot (\log n)^0$

$T(n) = \Theta(n(\log n)^{0+1})$

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

← we have to multiply by $(\log)^k$ to make both equal

Ex4

$T(n) = 64\,T(n/2) + n^4$

$a = 64 \quad b = 2$

$f(n) = n^4$ | $n^{\log_b a} = n^6$ → greater by polynomial | $T(n) = \Theta(n^6)$

Ex5

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

$f(n) = n\log n$ | $n^{\log_b a} = n \Rightarrow n(\log n)^1$ → smaller by log rathmic go to Case 3 | $T(n) = \Theta(n(\log n)^{1+1}) = \Theta(n(\log n)^2)$

Explain

If $f(n) = n$ | $n^{\log_b a} = n\log n \times (\log n)^{-1}$ (not allowed b/c $k \geq 0$) → greater by log rathmic | MT not possible

$f(n) = n$ | $n^{\log_b a} = n^n$ → by exponential still work | $T(n) = \Theta(n^n)$

Ex6

$f(n) = n$ | $n^{\log_b a} = n \cdot 2^n$ → greater by exponential | $T(n) = \Theta(n \cdot 2^n)$

Ex7

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

a is not constant.

Hence MT not applicable.

Ex8 $T(n) = 2T(n/2) - n$ ← -ve $f(n)$

Hence MT is not applicable

NOTE: If recurrence relation containing Root operator

Root operator

Ex: $T(n) = T(\sqrt{n}) + c$ $\quad c$ = constant

Step 1 : ~~put $n = 2^k$~~ assume $n = 2^k$

Step 2 $\quad T(2^k) = T(2^{k/2}) + c$

Step2: assume $T(2^k) = S(k)$

$S(k) = S(k/2) + C$

$a=1 \quad b=2$

$f(n) = C \mid n^{\log_b a} = k^0 = 1 \cdot (\log k)^0$

$S(k) = O(C \cdot (\log k)^{0+1})$

$S(k) = O(\log k)$

Step3: $T(2^k) = O(\log k)$

Step4: $T(n) = O(\log \log n)$ $\quad (\because n = 2^k, \; k = \log n)$

Ex3: $T(n) = 2T(\sqrt{n}) + \log n$

Sol: assume $n = 2^k$

$T(2^k) = 2T(2^{k/2}) + \log_2 2^k$

assume $T(2^k) = S(k)$

$S(k) = 2S(k/2) + k$

$a=2 \quad b=2$

$f(k) = k \mid k^{\log_b a} = k (\log k)^0$

$S(k) = O(k \cdot (\log k)^{0+1})$

$S(k) = O(k \cdot (\log k))$

$T(2^k) = O(k \cdot (\log k))$

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

Divide & Conquer

Divide :

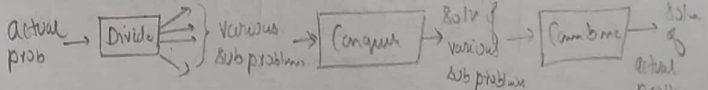
Divide the prob into some sub problems

Conquer

Force the sub problem by calling recursively until sub problems are solved.

Combine

Combine the sub problem soln so that we will get final problem soln.



DAC - Abstract Algorithm

DAC (a, i, j) — passing an array which contains elements from i to j

```
{
  small prob { if (small(a,i,j))
                 return (Solution(a,i,j));
  else
  {
    big prob  m = divide(a,i,j);      → divide
              b = DAC(a,i,m);         → conquer
              c = DAC(a,m+1,j);
              d = Combine(b,c);       → combine
    return(d);
  }
}
```

```
small(a,i,j)
{
  :
}
```

```
Solution(a,i,j)
{
  :
}
```

```
divide(a,i,j)
{
  :
}
```

```
Combine(b,c)
{
  :
}
```

DAC Algorithm Time Complexity finding

$$T(n) = \begin{cases} O(1) & \text{if } n \text{ is small} \\ f_1(n) + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + f_2(n) & \text{if } n \text{ is big} \end{cases}$$

($f_1(n)$: divide time; $f_2(n)$: combine time)

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

no of subproblems: a; size of each subproblem: $\frac{n}{b}$; time for divide & combination: $f(n)$

$a \geq 1 \quad b > 1$

Recurrence relation have many ways

To find value → write recursive relation to find value

To find Time → write recursive relation to find time.

31/7/18

Applications of DAC

1) Finding MAX & MIN
2) Power of an element
3) Binary Search
4) Merge Sort
5) Quick Sort
6) Selection Procedure
7) Finding no. of Inversions
8) Strassen's Matrix Multiplication

Finding MAX & MIN

i/p: Array of n-elements

o/p: Find — min, — max

Array Sorted in ascending order always 1 comparison

Best case: $1.(n-1)$ → always 2 comparisons

Worst case: $2.(n-1)$ Array Sorted in descending order

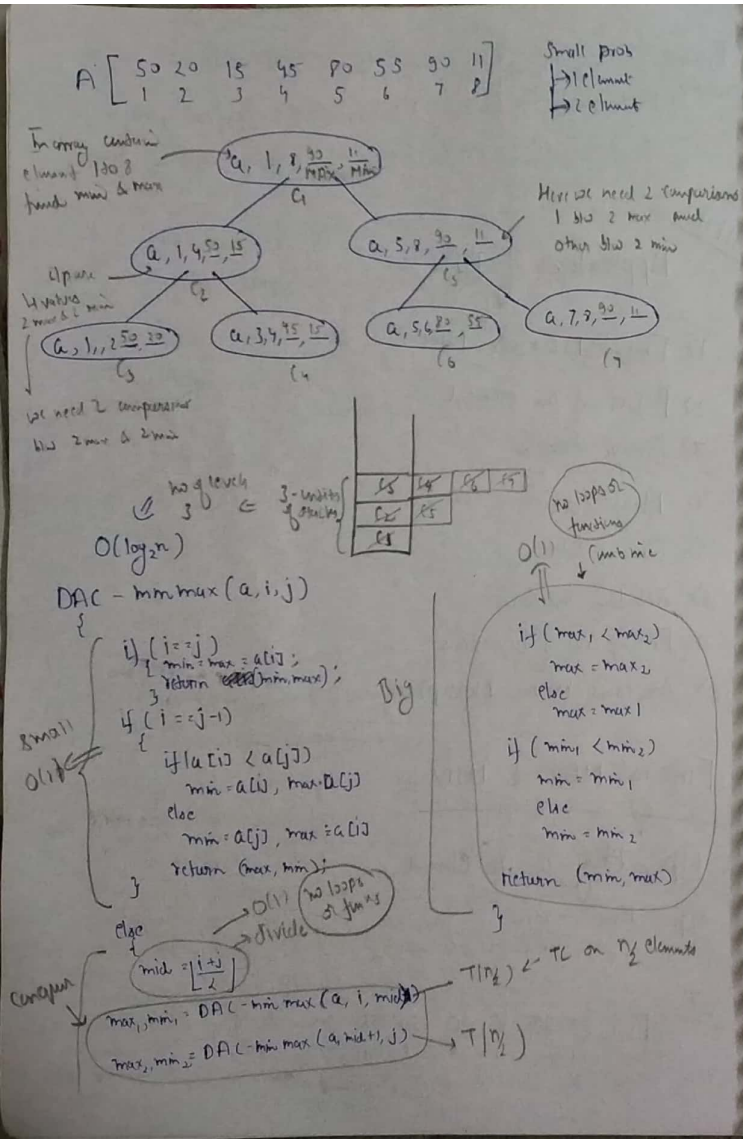
$T(n) \rightarrow \Theta(n)$ [Every case] → because have to compare with every element

Ex: A [50 10 15 45 80 55 90 11]
1 2 3 4 5 6 7 8

min = max = 50

if (a[i] > max) max = a[i]

else if (a[i] < min) min = a[i]



Let T(n) = Time complexity of above algorithm on n-element array.

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \text{ or } 2 \\ O(1) + 2T(n/2) + O(1) & \text{if } n > 2 \\ \Downarrow \\ 2T(n/2) + C & (C \text{ is constant} \end{cases}$$

divide time & combine time for top level

Solving by substitution

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

$$2\left[2T(n/2^2) + C\right] + C$$

$$2^2 T(n/2^2) + 2C + C$$

$$2^2\left[2T(n/2^3) + C\right] + 2C + C$$

$$2^3 T(n/2^3) + 2^2 C + 2C + C$$

$$\vdots$$

each term represents cost at each level

this is stack space

$$2^k T(n/2^k) + 2^{k-1}C + 2^{k-2}C + \dots + 2C + C$$

put $n/2^k = 1$ $\quad n = 2^k \Rightarrow k = \log n$

small problem cost

big problem cost

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

$$= n + n \cdot 2^{\log n} + 2^{\log n - 1} + n \cdot 2^{\log n - 2} + \dots + 2^1 + 2^0$$

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

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

$$= O(n)$$

Every case bz escaping not possible in between

Space Complexity

i/p + Extra

$(n)_B + (\log_2 n)_B$

$= O(n)$

Let $T(n)$ = no of comparisons required for the above algorithm to find the elements on n-elements array

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

$\Downarrow$

$2T(n/2) + 2$

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

$2\left[2T(n/2^2) + 2\right] + 2$

$2^2 T(n/2^2) + 2^2 + 2$

$2^2\left[2T(n/2^3) + 2\right] + 2^2 + 2$

$2^3 T(n/2^3) + 2^3 + 2^2 + 2$

$\vdots$

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

put $\frac{n}{2^k} = 2$ ↑ 2 will come first

$n = 2^{k+1}$

$\log_2 n = k+1$

$k = \log_2 n - 1$

$2^{\log_2 n - 1} T(2) + 2^{\log_2 n - 1} + 2^{\log_2 n - 2} + \dots + 2^2 + 2^1$

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

$n/2 + 2((n/2) - 1)$

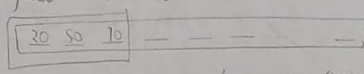
$= n/2 + n - 2$

$= \boxed{\frac{3n}{2} - 2}$ Every case.

$= \boxed{1.5n - 2}$

Given an array of n-distinct elements

i) find 1 element which is neither min nor max $TC \to O(1)$



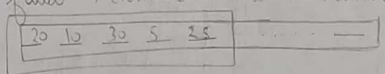
how we take first 3 elements

take any 3 elements from an array & sort them 10, 20, 50

Mid element is the required element bz it is neither min nor max

Since $10 < 20$ & $50 > 20$.

ii) find 1 element which is neither 2nd max nor 2nd min



take any 5 elements from an array & sort them $TC \to O(1)$

5 10 (20) 25 30

↑ required element

~~Power of an ★ Element~~

DAC – min max : Here we divide actual problem into 3 subproblems

RR for TC $T(n) = O(1) + 3T(n/3) + O(1)$ ← Compare 3 min & 3 max. in Combining step.

RR for no of comparisons

$T(n) = 0 + 3T(n/3) + (2+2)$ ← 2 comparisons for min, 2 comparisons for max

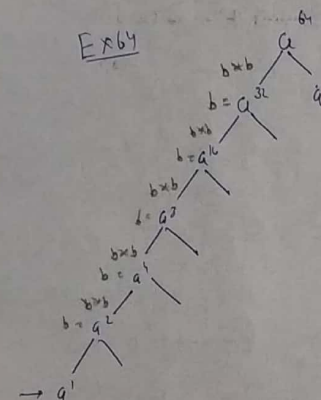
Power of an element

i/p : 2 integers $a \geq 2$, $n \geq 1$

o/p : Find a^n

$a^n = a^{n/2} \cdot a^{n/2}$

Ex: 64



Small problem stop if (n==1) return(a);

```
DAC-power(a,n)
{
  if (n==1)
    return a;
  else
  {
    b = DAC-power (a, n/2)
    return (b*b);
  }
}
```

For odd power

```
else
{
  mid = ⌊n/2⌋
  b = DAC-power (a, mid)
  if (n>1 && n%2 == 0)
    return (b*b);
  else return (b*b)*a;
}
```

```
else
{
  mid = n/2 ;        → divide
  b = DAC-power (a, mid) ;   → conquer
  c = b*b ;          → combine
  return c;
}
```

Let T(n) = TC of above algorithm

Recurrence relation

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

$\Downarrow$

$T(n/2) + c$ ← divide once & combine once at top level

Substitution

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

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

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

⋮

$T(n/2^k) + ck$

put $n/2^k = 1$ ($k = \log_2 n$) ← Stuck size

$T(1) + c\log_2 n$ ← big problem cost

(small problem cost)

$O(1) + c\log_2 n$

$= O(\log_2 n)$ → (Every time) bcz no escape in b/w return at end

Space Complexity

i/p + Extra

$4B + (\log n)B$

$= O(\log n)$

1/8/13

No of Multiplications

Let $T(n)$ = no of multiplications for the above program to find a^n.

Recurrence Relation

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

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

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

$T(n/2^3) + 1 + 1 + 1$

$\vdots$

$T(n/2^k) + 1 \cdot k$

$T(1) + 1 \cdot \log n$

$= O(\log n)$

If division is done in 3 parts

$a^n = a^{n/3} \cdot a^{n/3} \cdot a^{n/3}$

$T(n)$ = no of multiplications

$T(n) = 0 + T(n/3) + 2$

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

$T(n) = O(2\log_3 n)$

$T(n) = O(\log_3 n)$

NOTE: To apply DAC $n = 2^k$ i.e. n should be power of 2

Searching

Linear Search

i/p : an array of n elements ; element - x to be found

o/p : position of x if found else return (-1)

Ex:

	1	2	3	4	5	6	7	8	9	10	x = ?
A[	50	70	40	20	30	80	90	60	25	13]	

$x = 30$	$x = 13$	$x = 99$	$x = 50$
5	10	-1	1

BC $\rightarrow O(1)$ ← first element or near first element

WC $\rightarrow O(n)$ ← last element or near last element

AvgC $\rightarrow \dfrac{1 + 2 + 3 + 4 + \ldots + n}{n}$ ← all possible cases

$= \dfrac{n(n+1)/2}{n} = \dfrac{n+1}{2} \Rightarrow O(n)$

B. first element is target element + second element is target element + third element is target element + ... + n^{th} element is target element

```
for (i=1; i<=n; i++)
    if (a[i] == x)
        return i
```

Binary Search (Application of Partial DAC, bcz no need to combine)

i/p: A sorted array of n elements ; element to be eliminated - x

o/p: find position of x if found, else return (-1);

Ex:

A[10 20 30 40 50 60 70]
1 2 3 4 5 6 7

x = 55

mid = $\frac{1+7}{2}$ = 4

$\frac{5+7}{2}$ = 6

Program

```
BSTO BinarySearch(a, i, j, x)
{
    if(i == j)                          // O(1)
    {
        if(a[i] == x)
            return i;
        else
            return -1;
    }
    else
    {
        if(a[i] < x)
            return i;
        if(a[i] > x)
            return BinarySearch(
    }
    Mid = ⌊(i+j)/2⌋                     // => C
    if(a[mid] == x)
        return mid;
    else
    {
        if(a[mid] > x)
            return BinarySearch(a, i, mid-1, x)   -> T(n/2)
        else
            return BinarySearch(a, mid+1, j, x)   -> T(n/2)
    }
}
```

Note:

In case of ternary search

$T(n) = T(n/3) + C$

(1/3 | 2/3 | 3/3)

mid1

T(n) = 99

T(n/3) = 33

if(a[mid1] < x)

$mid_2 = 2 \times n/3 = 2 \times 33 = 66$

if(a[mid2] < x) go to III

else go to II

TC → $O(\log_3 n)$

Let T(n) = TC of above program on n elements array

Then Recurrence relation (Worst case)

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

(same constant time to decide which side to go.)

(C (finding mid, 3 → comparing left or right, 2 → mid my element or not))

by Substitution

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

$= T(n/2^2) + C + C$

$= T(n/2^3) + C + C + C$

$\vdots$

$T(n/2^k) + C \cdot k$ put $n/2^k = 1$ ($k = \log n$) (Stack size)

$T(1) + C \cdot \log n$

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

by using non-recursive Stack space can be reduced to O(1)

Space Complexity

i/p + extra

(n)B + (log n)B ⇒ O(n)

SC = O(n)

for non-recursive prog. stack space is O(1)

report this answer in gate for stack space

Scanned by CamScanner

Best Case mid element is the required element x

$T(n) = O(1)$

Average Case Most of the time worst case will happen

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

NOTE: For tailed recursive prog, its corresponding non-recursive prog will have constant stack space $O(1)$

(by turn (all write everything in while loop)

EX1

i/p : A sorted array of n ~~elements~~ distinct integers (+ve or -ve)

o/p : find any element A[i] such that A[i] = i

A[-100	-70	-60	-50	-30	-10	-5	-3	0	7	9	10	13	15	20	25	50	70	90	100	150
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

170 | 200]
22 | 23

Linear Search → O(n)

We use binary search

Concept: V = 13, P = 13 (possible) | V = 20, P = 14 (more go left) | to left (less than 20) | to right more than 20 we can not get V = 15, P = 15

If we go to left: V < 15 | V = 15 → go to left or right | V > 15 | V = 21, P = 21 possible

V = 19, P = 19 not possible | P = 20 | More go to right

algo → if (P < V) go left else go right

BS → TC (O(log n))

Case - II

A sorted array of n integers (not distinct +ve or -ve)

10	10	11	V ≤ 10 go to left	V = 10	11	12	13	13
10	11			P = 12				

More we can not apply Binary Search

EX 2

i/p : An array of n-elements in which ~~same positions~~ initially all are integers and afterward all are ∞.

o/p : Find position of first ∞

Sol: (logically sorted)

LS → O(n)

A[-100	70	-60	50	-20	10	-5	100	-200	26	55	92	54	-150	∞	∞	∞	∞	∞	∞	∞
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

(go to right ∞ after integers)

(compare element by this finger to check whether ∞ or not)

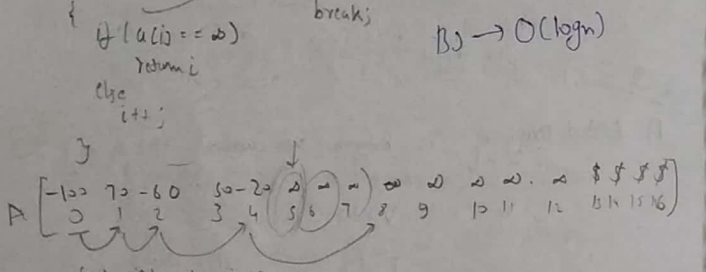
BS → O(log n)

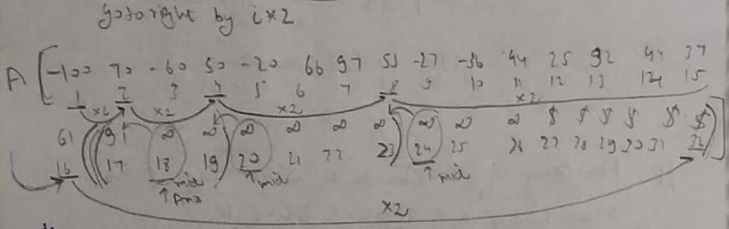
EX3

i/p An array of n-elements, in which until some position all are integers and afterwards all are ∞ (where n is unknown and after that all are $'s)

o/p: Find the position of 1st-∞

```
While(1)                    LS → O(n)
{
  if(a[i]==$)
    break;
  if(a[i]==∞)               BS → O(logn)
    return i
  else
    i++;
}
```





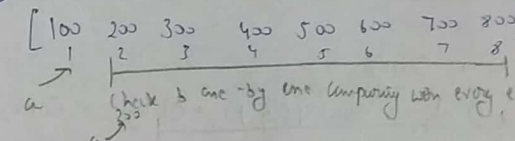
EX4

Here we multiply position by 2 in every iteration and move right, hence acts as BS $T(n) \to O(\log_2 n)$ [i.e. $2^1, 2^2, 2^3, 2^4, \ldots 2^k$]

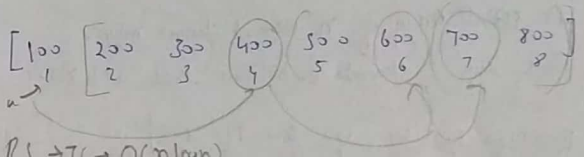
i/p : A sorted array of n-Integers

o/p: Find any 2 elements (a & b) such that $a+b == 1000$

EX2 Linear Search $(n-1)+(n-2)+(n-3)+\ldots+1 = \frac{n(n-1)}{2} = O(n^2)$

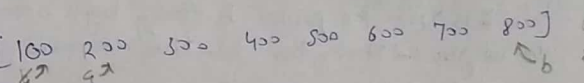


Here set a = first element and apply LS for n-1 elements. If b not found then set a = 2nd element and apply LS for (n-1) elements. In worst case we might end up applying LS for every element $T(n) \to n \times O(n) \to O(n^2)$



BS → $T(n) \to O(n \log n)$

Here fix a = first element and apply BS in remaining (n-1) elements. If none of them turn out to be "b", then set a = 2nd element and apply BS in remaining (n-1) elements. This procedure continues until we don't get $a+b == 1000$. In worst case we might end up applying BS for every element



Greedy Algo $T(n) \to \cancel{O(1)}\ O(n)$

```
1. a - 1st Etgts element
   b - last element
While (a != b)
{
  if(a+b == 1000)
      return (a,b)
  else
    if(a+b > 1000)
      b=b-1
    else
      a=a+1
}
```

Case - II

2/8/18

if array not sorted

1) apply LS → $O(n^2)$

2) o sort → $O(n \log n)$ then apply BS → $O(n \log n)$ ⇒ $O(n \log n)$

best → 3) sort → $O(n \log n)$ then apply Greedy Algo → $O(n)$ ⇒ $O(n \log n)$

Ex-5

i/p: A sorted array of n element distinct integers.

o/p: Find any 2 elements (a, b) such that $a+b > 1000$

A [100 200 300 400 500 600 700 800]
1 2 3 4 5 16 7 8

Sol since it is sorted array, add last 2 elements
$a[n] + a[n-1]$ $T(n) \to O(1)$ constant time

if $a[n] + a[n-1] \not> 1000$

then $a+b > 1000$ not possible bcz if adding largest 2 elements does not give sum > 1000 then $a+b \not> 1000$ never possible

Case II

i/p array not sorted

Algo ①

1) Sort + last 2 elements ⇒ $O(n \log n) + O(1) \Rightarrow O(n \log n)$

Algo ②

1) Find 1st max + remove it + find 2nd max + add both

DAC - maximo → $O(n) + O(1) + O(n) + O(1) \Rightarrow O(n)$

Find max_1 & max_2 in a single pass

20 | 15

$max_1 = 10$ | $max_1 = 15$

$max_2 = 8$ | $max_2 = 15$

```
if(a[i] > max1)
    max1 = a[i]
else {
    if (a[i] > max2)
        max2 = a[i]
    else
        continue;
}
```

Algo ③

1) apply 2 passes of bubble sort to get 2 max elements

$O(n) + O(n) = O(n)$

2) (max_1 + max_2) return $O(1)$

$T(n) \to O(n) + O(1) \to O(n)$

Case - III $a+b+c = 1000$

LS → n^3

BS → $O(n^2 \log n)$ fix a: 1st element, b = 2nd element & apply BS for (n-2) elements to find c

Greedy → $O(n^2)$ fix a = 1st element & apply greedy b/w (n-1) elements to find b & c

Case - IV $a+b+c < 1000$

Greedy → $O(1)$

return first 3

Ex-6

i/p: An array of n-ele in which every element is 20 or 30.

o/p: Find sorted array.

[20 30 20 20 30 30 20 30]
1 2 3 4 5 6 7 8

$T(n) \to O(n)$ Every case

(create one extra array)

[20 20 20 20 30 30 30 30]

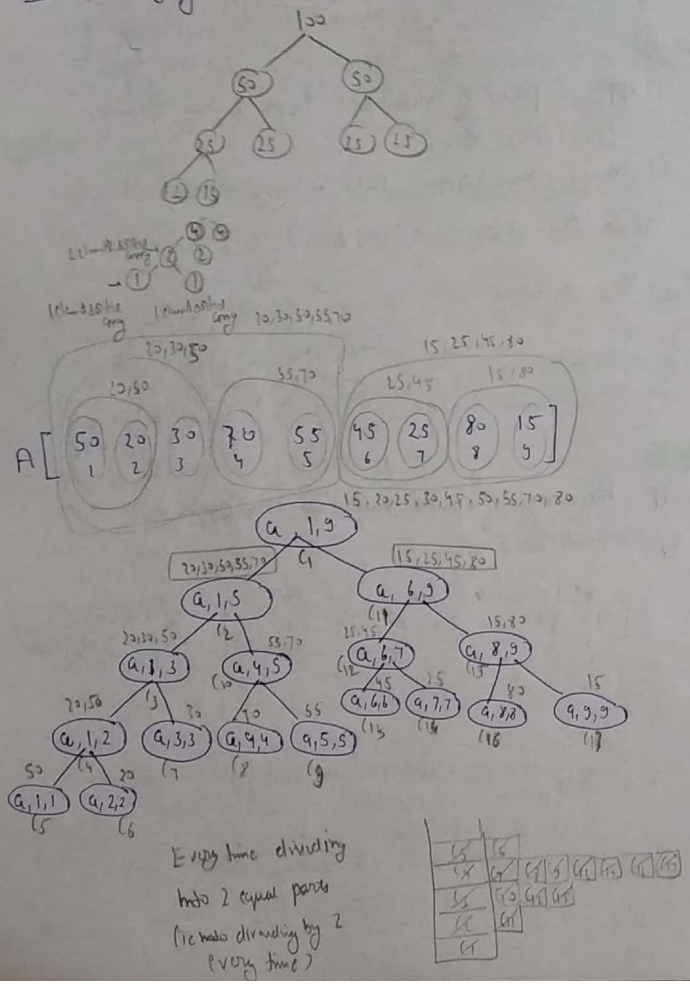
i = 0 & j = n-1

if (a[i] == 20) a[i] = 20; i++

if (a[j] == 30) a[j] = 30; j--;

Merge Sort

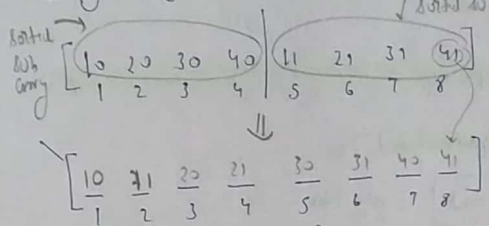
Note: Merging 2 sorted sub arrays



Every time dividing into 2 equal parts (ie mostly dividing by 2 every time)

Merge Algorithm (Out-place Algorithm)

Input for 1st place



10,11 → 10
11,20 → 11
20,21 → 20
30,21 → 21
30,31 → 30
40,31 → 31
40,41 → 40
no comparisons for last element

Comparisons (Worst Case)

$n_1, n_2 \rightarrow n_1 + n_2 - 1 = n - 1$

$m, n \rightarrow m + n - 1$

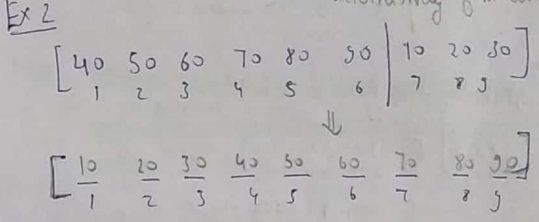
Moves

$n_1, n_2 \rightarrow n_1 + n_2 = n$

$m, n \rightarrow m + n$

Here we have worst case elements are going out alternatively from both the groups

Ex 2



40,10 → 10
40,20 → 20
40,30 → 30

Here we have Best case, bz all the elements are going from one sub array and that too smaller subarray.

Comparisons (Best Case)

$n_1, n_2 \rightarrow n_2$

$m, n \rightarrow \min(m, n)$

Moves

$n_1, n_2 \rightarrow n_1 + n_2$

$m, n \rightarrow m + n$

For every element elements there is 1 comparison and 1 movement.

TC → O(comparison + movement) but moves > comparison

⇓

O(movements)

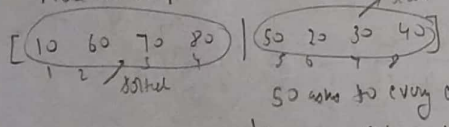
TC → O(m+n) Every case

In-place Merge Algorithm

[50 60 70 80 | 10 20 30 40]
1 2 3 4 | 5 6 7 8

50,10 → 10

Here we swap both 50 & 10



50 asks to every element

Hence O(n) time is taken

[10, 60 70 80 | 20 30 40 50]
1 2 3 4 | 5 6 7 8

60,20 → 20

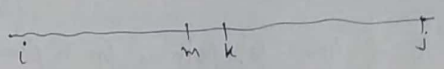
∴ { 50,10 → 10
60,20 → 20

O(n) Here sorting is done n times
O(n) ← m × O(n)
O(n)
O(n)
O(n)

TC = O(mn)

Here Here after every comparison 2nd sorted subarray gets disturbed, to sort it every time O(n) time required

p = 1 ← keep track of index of array B




```
While (i ≤ m && k ≤ j)
{
    if (a[i] < a[k])
    {
        B[p] = a[i];
        p++; i++;
    }
    else
    {
        B[p] = a[k];
        p++; k++;
    }
}
Copy remaining elements to B
```

Note: Merging 2 sorted sub arrays each of size m & n will take O(m+n) time [Every case] bcz of moves.

Merge Sort - Algorithm

```
Mergesort (a, i, j)
{
    if (i == j)                          O(1)
        return a[i];
    else
    {
        mid = [(i+j)/2]                  ⇒ O(1)
        mergesort (a, i, mid);           → T(n/2)
        Mergesort (a, mid+1, j);         → T(n/2)
        merge (a, i, mid, mid+1, j);     → O(m+n) = O(n/2 + n/2) = O(n)
    }
    return a;
}
```

Copy all elements from B to A.

Let
$T(n)$ = TC of above algorithm on n elements array

Recurrence Relation for TC

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

$T(n) = 2T(n/2) + n$ ← top level divide & Combine time

Substitution

$T(n) = 2^2T(n/2^2) + n/2 + n$ ← divide Time & Combine time for 2nd level

$2^2[2T(n/2^3) + n/2^2] + n + n$

$2^3T(n/2^3) + n + n + n$

⋮

$2^kT(n/2^k) + nk$ ← stack size

put $n/2^k = 1$

$k = \log n$

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

$n \cdot O(1) + n\log n$ ← to solve n small problem

$= O(n\log n)$ [Every case] bcz return at last no escape in this

Space Complexity

auxiliary array space (B)

i/p + Extra

$(n)B + (n)B + (\log n)B$ → Extra space

↓ stack space

$= O(n)$

For In place Merge sort algorithm

$T(n) = 2T(n/2) + O(n^2)$ if $n>1$

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

Using MT $f(n) = n^2$ | $n^{\log_2 2} = n$ $T(n) = O(n^2)$

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

Note: 1) Merge sort is out-place algo bcz in the merge algorithm we are taking $O(n)$ space extra.

Outplace algo → if any algo takes extra space > $\log n$ (more than)

In place algo → if any algo takes extra space ≤ $\log n$

2) If the array size is very large go to merge sort

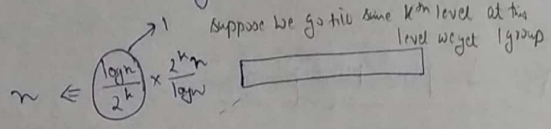
Otherwise go to Insertion sort.

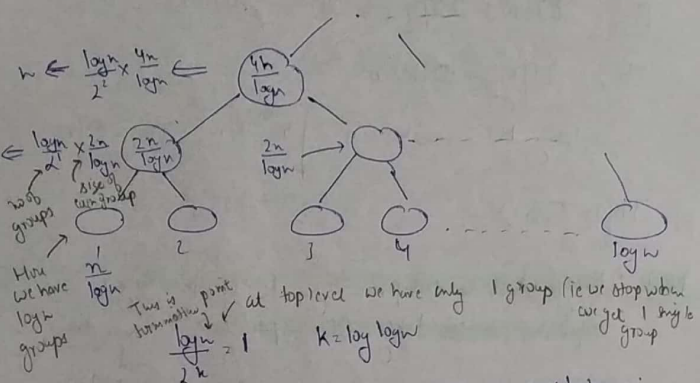
3/8/18

Ex1

i/p: $\log n$ sorted subarrays each of size $\frac{n}{\log n}$

o/p: Find single sorted array with n elements





$\frac{n}{\log n}$

This is termination point

at top level we have only 1 group i.e we stop when we get 1 single group

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

Cost at each level is n i.e at every level we have n movements

$\therefore TC \rightarrow n \times \log \log n = O(n \log \log n)$

To actual merge sort we have n sorted subarrays each of size 1.

At every level, for every element there is comparison & movement, Hence Cost (or) Time $= O($comparisons + movements$)$

$= O($movements$) = O(n)$

bz in WC comparisons $= m+n-1$

& in every case movement will always be $= (m+n)$

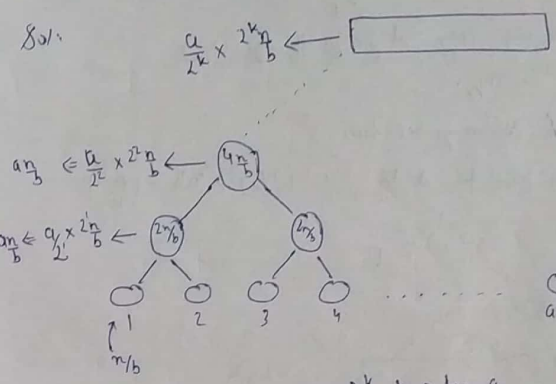
Ex2

i/p: a sorted subarrays each of size n/b.

o/p: find single sorted subarrays each of size $\frac{an}{b}$.

$\downarrow$ TC

Sol:



$\frac{a}{2^k} = 1 \qquad a = 2^k \quad k = \log_2 a$

Cost at each level $= a(\frac{n}{b})$

$\therefore TC \rightarrow O\left(\frac{an}{b} \log_2 a\right)$

Ex3

i/p: nk sorted sub arrays each of size $\frac{n}{k}$

o/p: Find single sorted array with n^2 elements

Sol: at k^{th} level $\frac{nk}{2^k} = 1 \quad k = \log_2 nk$ & Cost at each level $= n^2$

$TC \rightarrow n^2 * \log nk$

$O(n^2 \log_2 nk)$

Case - II If each subarray is unsorted

i/p : nk subarrays each of size n/k

o/p : find single sorted array with n^2 elements

Initially subarrays are sorted using bubble sort.

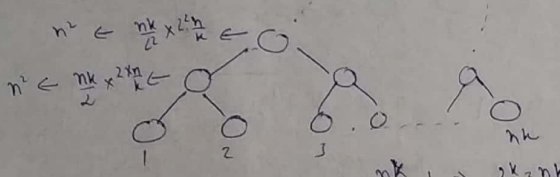
Sol: Initial to sort every group of size (n/k)

Time $\rightarrow (n/k)^2$

nk such subarrays are there.

Hence total time for sorting is $(n/k)^2 \times nk = \left(\frac{n^3}{k}\right)$

$\frac{nk}{2^k} \rightarrow 1$, $2^k \rightarrow \frac{n}{k}$



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

levels $\Rightarrow \log nk$

Cost at each level $\rightarrow n^2$

TC $\rightarrow (n^2 \times \log nk)$

Total TC $\Rightarrow O(n^2 \log nk) + O\left(\frac{n^3}{k}\right)$

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

Ex (Important)

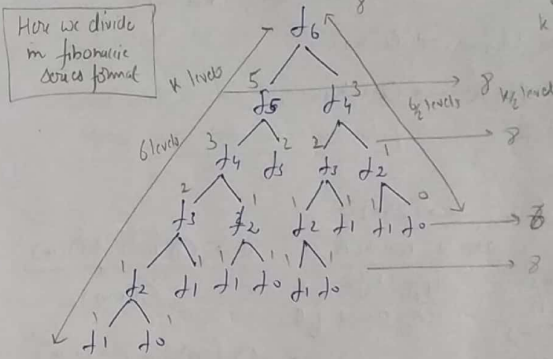
n	0	1	2	3	4	5	6	7	8	9	10
f(n)	0	1	1	2	3	5	8	13	21	34	55

f_k-elements to sort using fibonaccie-Merge sort find TC?

f_k-element $\Rightarrow$ fibonaccie of k elements

$f_k = f_{(k-1)} + f_{k-2}$ (k levels, $k/2$ levels)

Here we divide in fibonaccie series format



Cost at each level is f_k i.e 8. (this is merging time)

Here we have k such levels.

$TC \rightarrow O(k \cdot f_k)$

Ex: 5 (Imp)

i/p: 2 sorted arrays A & B where A contains m distinct elements & B contains n distinct elements

o/p: find $A \cap B$

$A \cup B$

A: 10 20 30 40 50 60 70

B: 5 10 15 25 25 40 55 70 80 90

10 20

A∩B: We use merge algorithm while comparing elements, skip smaller element. if both the elements are same, print once and skip both.

Imp

1) TC → Merge Algo → TC = O(m+n) (Worst Case)
min(m,n) (Best Case)

2) m × LS(n) → O(mn)

3) m × BS(n) → O(m log n)

A∪B

Algo 1) add A elements → O(m)
add B element also if not in A
ie apply n times LS in array A
TC → n × O(m)
→ O(mn)
TC → O(mn) + O(m)
O(mn)

Algo 2) for each add A elements
for every of B apply BS in A
→ O(n log m) + O(m)
TC ⇒ O(n log m)
apply LS for every element of B

Algo Imp 3) Using Merge Algorithm TC → O(m+n) [Every Case]

if A=B Compare A & B
print Smaller
Compare A & B
if both equal
print once and skip both

Case - II B & C array not sorted each of size n

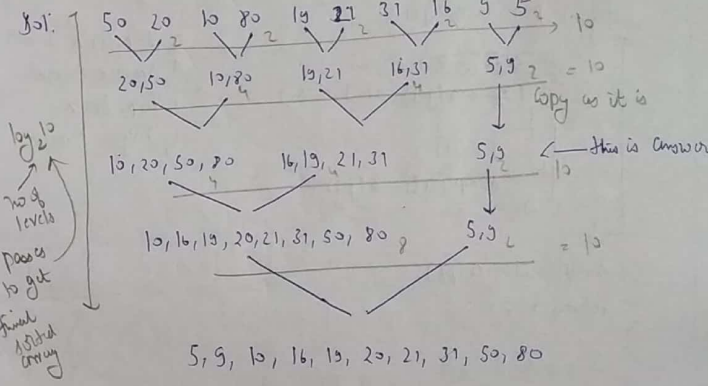
Algo 1) Using LS → O(n²)

Algo 2) Better Algo 1) sort both array O(n log n) } O(n log n)
2) Merge both array O(n)

EX-6 Consider following array

50 20 10 80 19 21 31 16 9 5

What will be the O/p after second pass of straight 2 way Merge sort algorithm. → take two-two element & merge them



5, 9, 10, 16, 19, 20, 21, 31, 50, 80

TC → (10 · log₂10)

TC → (n log n)

Quick Sort

1. In-place algorithm
2. Not - stable
3. It is practical sorting algorithm

Partition Algorithm

Partition (a, p, q) ⇒ O(n) [E C]

```
{
    x = a[p] ; //select pivot as 1st element
    i = p
    for (j = p+1; j < q; j++)
    {
        if (a[j] ≤ x)
        {
            i = i+1;
            swap (a[i], a[j])
        }
    }
    swap (a[i], a[p]);
    return i ;
}
```

n-1 times

i keeps track of smaller element
j keeps track of bigger element

NOTE: Before starting partition, always fix p, q, i, j
after partition pivot will go to its correct place.

Important: Apply Quick Sort only if given array is unsorted. else if applied in sorted array Quick sorts will give worst time.

Ex) QS(a, 1, 10)

A [50 70 90 30 25 45 88 78 98 42]
  1 2 3 4 5 6 7 8 9 10

p, i, j → move forward

return 5

pivot | smaller | bigger

(42 30 25 45) (50) (70 88 78 98 90)

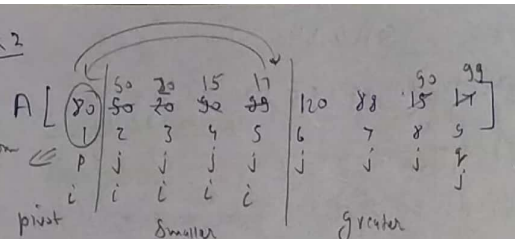
smaller | greater

QS(a, 1, 4) QS(a, 6, 10)

pass 1st: 50 (70) 90 30 25 45 88 78 98 42
pass 2nd: 50 70 (90) 30 25 45 88 78 98 42
pass 3rd: 50 70 90 (30) 25 45 88 78 98 42
pass 4th: 50 30 70 90 (25) 45 88 78 98 42
pass 5th: 50 30 25 70 90 (45) 88 78 98 42
pass 6th: 50 30 25 45 90 70 (88) 78 98 42
pass 7th: 50 30 25 45 90 70 88 (78) 98 42
pass 8th: 50 30 25 45 90 70 88 78 (98) 42
pass 9th: 50 30 25 45 90 70 88 78 98 (42) 90
pass 10th: [42 30 25 45 50 70 88 78 98 90] j

(42 30 25 45) (50) (70 88 78 98 90)

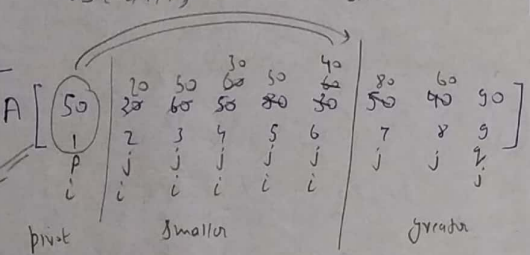
EX2



(17 50 20 15)(80)(120, 88 90 99)
Smaller pivot bigger
QS(a,1,4) QS(a,6,9)

NOTE: after partition pivot will be ~~at its~~ sorted, it will be present at its correct place.

EX3



(40 20 50 30 50)(50)(80 60 90)
Smaller or equal pivot greater

5/8/18 **Quick Sort** (In-place Algorithm)

```
QS(a, p, q)
{
  if(p == q)
    return(a[p]);
  else
  {
    m = partition(a, p, q)        → T(m-p)
    QS(a, p, m-1)                 → T(q-m)
    QS(a, m+1, p)
  } } return a;
```

(p ... m-1) m (m+1 ... q)
m-1-p+1 (m-p) q-(m+1)+1 (q-m)

It is non tailed recursion bcz after 1st fun call 2nd fun is called.

In tailed recursion after fun call, there is no work.

Let T(n) be TC of above algorithm

Recurrence relation (Every case)

$$T(n) = \begin{cases} O(1) & \text{if } n=1 \\ O(n) + T(n_1) + T(n_2) & \text{if } n>1 \\ T(m-p) + T(q-m) \end{cases}$$

(4) 70 10 30 20 60 50
(10 20 30) (40) (70 60 50)

(7) 20 50 10 40 30 60
(20 50 10 40 30 60)(70)(0)

(2) 70 30 50 60 10 40
(10)(20)(70 30 50 60 40)

$$T(n) = \begin{cases} O(1) & \text{if } n=1 \\ n + T(m-p) + T(q-m) & \text{if } n>1 \end{cases}$$

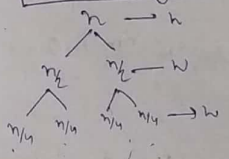
divide Time only

(Best Case) most of the time QS behaves best case (middle element)

$T(n) = n + T(n/2) + T(n/2)$ Hence / This is also the Average Case

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

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



Stack size = $\log n$
Cost at each level = n

BC → $O(n \log n)$
AC → $O(n \log n)$
WC → $O(n^2)$

(Worst Case)

$T(n) = T(n-1) + n + T(0)$

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

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

$1+2+3+\dots+n = \frac{n(n+1)}{2} = O(n^2)$

Actual stack space is $O(n)$ bcz bcz of tail recursion its equivalent non-recursive prg will compress stack space to $O(1)$ (max)

EX 1

Consider the following 3 i/p's. QS will give worst case when array is already sorted

(i) 1 2 3 4 ..... n-1 n (ascending)

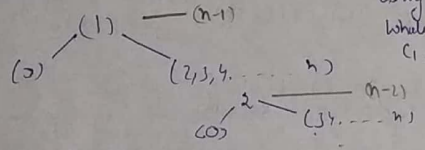
(ii) n, n-1, n-2, ... 2, 1 (descending)

(iii) n, n, n, ... n, n (all equal)

Let C_1, C_2, C_3 be the no of comparisons made for ip's (i) (ii) & (iii) respectively to sort in ascending order using QS. Then what is relation b/w C_1, C_2, C_3?

Sol (i) 1 2 3 4 ..... n-1 n

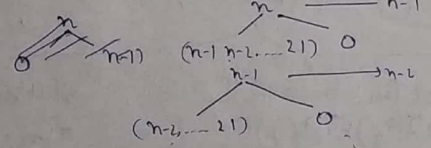
no of swaps = 1+1+1+ ..... n times = O(n)



~~Then what is the rela~~

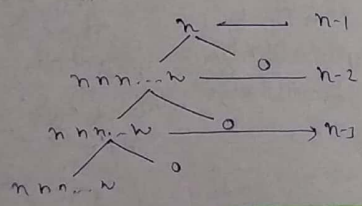
$C_1 = \frac{n(n-1)}{2} = O(n^2)$

(ii) no of comparisons at top level



$C_2 = \frac{n(n-1)}{2} = O(n^2)$

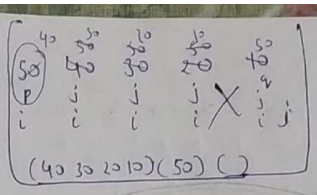
(iii)



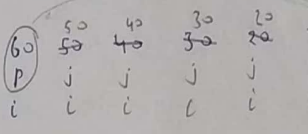
$C_3 = \frac{n(n-1)}{2} = O(n^2)$

$\therefore C_1 = C_2 = C_3$

(i)



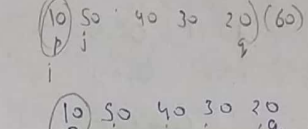
(ii)

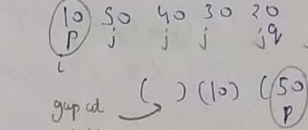


comp = n-1 = 5

swap = n = 6

gap at right





comp = n-2 = 4

swap = 1

gap at right

(40 30) (50) ( )

comp = 3

swap = 4

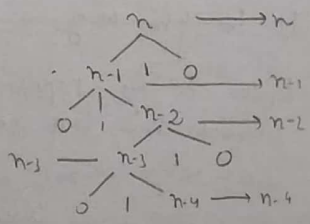
Total comp = n-1 + n-2 + n-3 + ... + 2 + 1

= $O(n^2)$

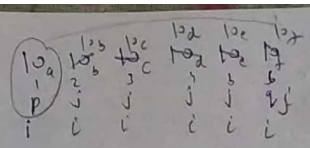
Total swaps = 2, 4, 6, 8, ... $n/2$

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

$= O(n^2)$



(iii)



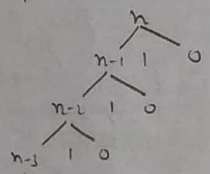
$(l_{0_f}\ l_{0_b}\ l_{0_c}\ l_{0_d}\ l_{0_e})(l_{0_a})(\ ) \longrightarrow 6S\ \ 5C$

$(l_{0_e}\ l_{0_b}\ l_{0_c})\ (l_{0_f})(\ ) \longrightarrow 5S\ \ 4C$

$(l_{0_c}\ l_{0_b})\ (l_{0_e})\ (\ ) \longrightarrow 4S\ \ 3C$

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

~~Not~~ Swaps $= n-1+n-2+\cdots+2+1 = O(n^2)$



$C_1 = C_2 = C_3$

$S_1 < S_2 < S_3$ <- mathematically

NOTE:

1) If array is already sorted go for Insertion sort where $TC \rightarrow O(n)$, (Quick sort never preferred bcz it will give worst case: $TC \rightarrow O(n^2)$

(or) almost sorted

~~If the given array is already sorted.~~

2) After sorting repeated elements order is not changed then that sorting technique is called stable sorting technique.

3) Quick sort is not stable.

Best Case & Average Case

$T(n) = n + T(n/2) + T(n/2) = O(n \log n)$

$T(n) = n + T(n/5) + T(\frac{4}{5}n) = O(n \log_{5/4} n)$

ie after partition fraction of n one side and running otherside

$T(n) = n + T(\alpha n) + T((1-\alpha)n)$ → RH $= \log_{\frac{1}{1-\alpha}} n$ Stack space = max{LH, RH}

LH $= \log_{\frac{1}{\alpha}} n$ $\quad 0 < \alpha < 1$ $\quad \alpha$ is fraction

Ex: $T(n) = n + T(n/3) + T(\frac{2}{3}n) = O(n \log_{\frac{3}{2}} n)$

Ex: $T(n) = n + T(\frac{n}{10}) + T(\frac{9n}{10}) = O(\log_{\frac{10}{9}} n) = O(n \log_{\frac{10}{9}} n)$

Stack space $= \max\left\{ \log_{\frac{1}{\alpha}} n\ ,\ \log_{\frac{1}{1-\alpha}} n \right\}$

Worst Case

ie after partition constant elements one side and running other side

$T(n) = n + T(1) + T(n-1) \Rightarrow \frac{n}{c} \times n = O(n^2)$

(↙ time of size (n-1) (stack size))

Stack size $= n/1$ → but due to NRP

$O(\log n)$ [max]

$O(1)$ [min]

Stack size reduced to

NRP = non-recursive program

Ex: $T(n) = n + T(2) + T(n-2) = O(n^2)$ $\quad n/2 \times n$

$T(n) = n + T(n-10) + T(10) = O(n^2)$ $\quad n/10 \times n$

Ex2

Q In the quick sort, sorting of n numbers the $n/5^{th}$ smallest element is selected as pivot using $O(n^2)$ TC algorithm. Then what will be the worst case TC of Quick sort?

a) $n \log n$ b) n^2 c) n^3 d) n

Sol: $T(n) = O(n^2) + T(n/5) + T(4n/5) + O(n)$

$= O(n^2) + O(\log_{5/4} n) \times n$

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

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

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

$= n^2 \left[\left(\frac{17}{25}\right)^0 + \left(\frac{17}{25}\right)^1 + \left(\frac{17}{25}\right)^2 + \cdots \left(\frac{17}{25}\right)^{\log_{4/5} n} \right)$

$= O(n^2)$

Ex3 In quick sort, sorting of n numbers the $n/5^{th}$ element is selected as pivot using $O(n)$ TC algorithm then what will be the worst case TC of quick sort?

Sol: $T(n) = O(n) + O(n) + T(n-1) + T(0)$

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

$= T(n-1) + n$

$= O(n^2)$

Ex3 In quick sort, the sorting of n numbers the $n/5^{th}$ largest element is selected as pivot using $O(n^2)$ TC algorithm then what will be the worst case TC of algorithm?

Sol: $T(n) = O(n^2) + T(4n/5) + T(n/5)$

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

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

$= O(n^2)$

Ex4 In quick sort, the sorting of n numbers the 25^{th} largest element is selected as the pivot, then what will be the worst case TC of quick sort?

Sol: $T(n) = O(1) + O(n) + T(n-25) + T(25)$

$= O(n) + T(n-25) + T(25)$

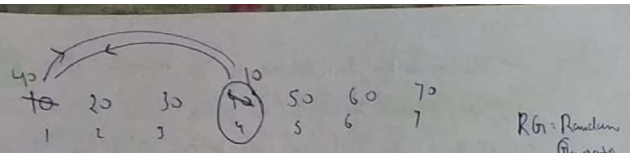
$= O(n^2)$

Note: middle element = $n/2^{th}$ element → $O(n^2)$

Median = $n/2^{th}$ smallest element → $O(n \log n)$

Randomized Quick Sort

Selecting pivot element randomly is known as Randomized Quick sort



RG = Random Generator

$r = 4 = RG(a, 1, 7)$

swap [a[p], a[r]]

10 (26) 30 10 (40) (50 60 70)

$1 = r = RG(a, 1, 3)$

= Swap[a[p], a[r]]

(10) (20) (30)

$7 = r = RG(a, 5, 7)$ ~~Not~~

(50 60) (70) ( )

(50) (60) ( )

RQS

$T(n) \rightarrow O(n \log n)$ Normal QS $T(n) \rightarrow O(n^2)$

	QS	RQS
BC	~~BC~~ $O(n \log n)$	$O(n \log n)$
AC	$O(n \log n)$	$O(n \log n)$
WC	$O(n^2)$ [10] ↑ happen 10 times	$O(n^2)$ [2] ↑ happen (very rare) 2 times

In randomize QS WC will happen but very rarely

Selection Procedure (Imp)

i/p : An array of n - element and integer - k

o/p : Find k^{th} - smallest element

Ex:

A[70 30 100 80 25 99 55 79 81 91 46 56] k=6 k=7

1 2 3 4 5 6 7 8 9 10 11 12

⇓ partition

m = 6 ⇐ (25 49 55 46 56) (70) (30 100 80 79 81 91)

1 2 3 4 5 6 7 8 9 10 11 12

Check k == m

if (k < m) go left

else go right

(50 100 80 79 81 91)

7 8 9 10 11 12

m = 10 ⇐ (30 79 81) (90) (91 100)

7 8 9 10

10 ≠ 7

go to left

m = 8 ⇐ (79) (80) (91)

7 8 9

go to left

(m = 7) 70 → return 70

Selection Procedure

```
SP(a, p, q, k)
{
    if (p == q)                          => O(1)
        return a[p];
    else
    {
        m = partition(a, p, q)           -> O(n)
        if (k == m){
            return a[m];}
        else{
            if (k < m)
                return SP(a, p, m-1, k)  => T(m-p)
            else
                return SP(a, m+1, q, k-m+p-1)  -> T(q-m)
        }
    }
}
```

Let T(n) = TC of above algorithm.

$$T(n) = \begin{cases} O(1) & \text{if } n == 1 \\ O(n) + T(n-p) \\ \quad (\text{or}) \\ \quad O(n) + T(q-n) & \text{if } n > 1 \end{cases}$$

Best Case

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

$= n\left[\frac{1}{2^0} + \frac{1}{2^1} + \frac{1}{2^2} + \cdots\right]$ ($O(1)$)

$= O(n)$

(or)

if after 1st partition k == n no further procedure

Worst Case

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

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

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

$= O(n^2)$

Stack size = O(1)

bcz of equivalent NRP

NRP = non-recursive program

Space Complexity

i/p + extra

n B ~~(log n) B~~ O(1) B

= O(n)

Counting Number of Inversions (Imp)

i/p: An array of n elements

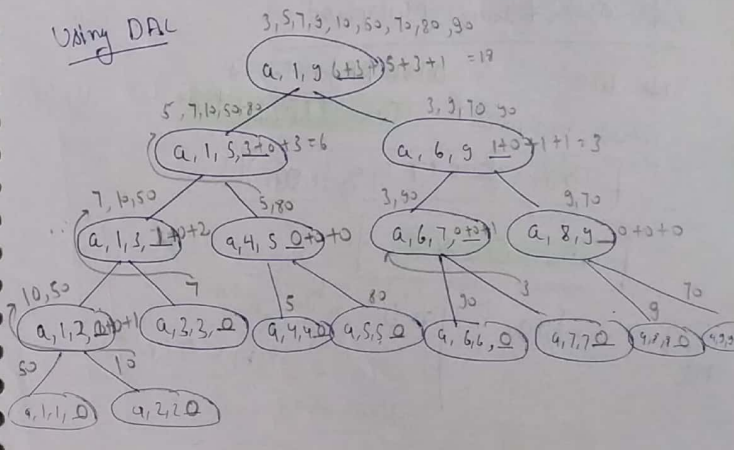
o/p: Find inversions. [50 20] (1, 2) 1 < 2, a[1] > a[2]

EX [50 10 7 5 80 90 3 9 70] (1 2 3 4 5 6 7 8 9)

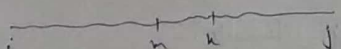
Inversion for 50 → 10, 7, 5, 3, 9
Inversion for 10 → 7, 5, 3, 9
Inversion for 7 → 5, 3
Inversion for 5 → 3
Inversion for 80 → 3, 9, 70
Inversion for 90 → 3, 9, 70

total inversions = 18

Using DAC



We will use merge Algorithm (with modification)




```
NI = NIL + NIR
While (i ≤ m && k ≤ j)
{
    if (a[i] < a[k])
    {
        b[p] = a[i];
        p++; i++
    }
    else
    {
        b[p] = a[k]
        p++; k++
        NI = NI + m - i + 1;
    }
}
return NI;
```

mid = (i+j)/2

```
NIL = (NI(a, i, mid)
NIR = (NI(a, mid+1, j)
NI = NIL + NIR
NI = Merge(a, i, mid, mid+1, j)
```

Strassen's Matrix Multiplication

With DAC

Matrix multiplicati.

Actual RR for Tmm is $T(n) = 7T(n/2) + 18n^2$

$T(n) = 7T(n/2) + O(n^2)$

$T(n) = O(n^{2.81})$

Using MT $f(n) = n^2$ | $n^{\log_b a} = n^{\log_2 7} = n^{2.81}$

$T(n) = \Theta(n^{2.81})$

Without DAC Matrix addition

$A = [\ ]_{m \times n}$ $B = [\ ]_{m \times n}$

$C = A + B$ $C = [\ ]_{m \times n}$

$T(n) \to O(n^2)$

```
for (i = 1 to n)
    for (j = 1 to n)
        C[i][j] = A[i][j] + B[i][j]
```

Without DAC Matrix multiplication

$A_{m \times n} * B_{n \times p}$

$C = A * B$ $C_{m \times p}$

$T(n) \to O(n^3)$

```
for (i = 1 to m)
    for (j = 1 to n)
        for (k = 1 to n)
            C[i][j] = C[i][j] + A[i][k] * B[k][j]
```

H.W (Imp)

(P1) i/p : An array of n elements in which until some position n elements in ascending order and after wards in descending order

o/p : finds x

LS = $O(n^2)$
BS = $O(n \log n)$

(P2) i/p : An array of n points in x-y plane.

o/p : finds closest pair

LS : $O(n^2)$
DAC : $O(n \log n)$

(P3) i/p : 2 - Sorted arrays — A - m distinct elements; B - n distinct elements

o/p : finds k^{th} smallest element in Union sorted array

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

Algo ① DMerge - Union $\Rightarrow O(m+n)$
Return k^{th} elem $\Rightarrow O(1)$
$\} O(m+n)$

Algo (2) Apply BS without merging $O(\log m + \log n)$

Ex:- A [50 60 70 80 90 40 30 20]
1 2 3 4 5 6 7 8

LS → Tc = O(n)

for (i = 1 to n)
if (a[i] < a[i+1])
i = i+1;
else
return i+1;

[50 60 70 80 90 40 30 20]
1 2 3 4 5 6 7 8

BS(a, i, j) Tc → O(log n)
{
if (i == j)
return i;
else
{
mid = $\lfloor \frac{i+j}{2} \rfloor$
if (a[mid] < a[mid+1])
return BS(a, mid+1, j);
else
return BS(a, i, mid-1);
}
}

(P3) finding k^{th} smallest element in Union Sorted arrays i.e A ∪ B

Arr(A) = m
Arr(B) = n

Method, applying Binary Search in both the arrays without performing Union

A = [10 20 30 40 50 60]
0 1 2 3 4 5

B = [5 15 25 35 45 55]
0 1 2 3 4 5

find k^{th} smallest
mid_1 = index of mid element of A
mid_2 = index of mid element of B.

if ($mid_1 + mid_2 + 3 < k$)
if ($mid_1 > mid_2$)
ignore first half of array B.
else
ignore first half of array A
else if ($mid_1 + mid_2 > k$)
if ($mid_1 > mid_2$)
ignore second half of array A
else
ignore second half of array B

5/8/18

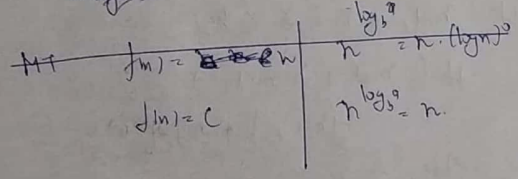
Consider the following C program

```
A(n)
{
  if (n ≤ 1)
    return 1
  else
    return (A(n/2) + A(n/2) + n);
}
```

a) O(nlogn)
b) O(n²)
c) O(n)
d) O(logn)

Sol: $T(n) = \begin{cases} 1 & n \le 1 \\ A(n/2) + A(n/2) + C & n > 1 \end{cases}$

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



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

Consider the following C program

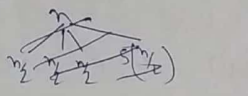
```
A(n)
{
  if (n ≤ 1)
    return (n²+n+1);
  else
  {
    return (3A(n/2) + 5A(n/2) + (MA(n)))
  }
}
```

Sol:

$$T(n) = \begin{cases} O(1) & n \le 1 \\ 3T(n/2) + 5T(n/2) + O(n^2) \\ T(n/2) + T(n/2) + O(n^2) \end{cases} \quad n > 1$$

b = A(n/2)
c = 3×b
d = A(n/2)
e = 5×d
f = c+e
g = MA[n]
h = f+g
return h

$T(n) = 3T(n/2) + 5T(n/2)$ ✗



$T(n) = 2T(n/2) + MA[n]$ (MA[n] → $O(n^2)$)

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

$= O(n^2)$

MT $f(n) = O(n)^2$ | $n^{\log_2 2} = n$

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

Note - While writing recurrence relations we consider only four calls

Ex

```
A(n)
{
  if (n ≤ 1)
    return (n²+n+1)
  else
    return (3A(n/2) * 5A(n/2) - 5A(n/2)/3A(n/2) + BS(n))
}
```

a = A(n/2)
b = 3×a
c = A(n/2)
d = 5×c
e = b*d

Sol: $T(n) = 4T(n/2) + O(\log n)$

MT $f(n) = \log n$ | $n^{\log_2 4} = n^2$

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

NOTE: While finding TC we will consider only func calls and loops.

func calling takes time

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

```
else
{
  m = n/2
  T(m)
}
```

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

EX: Consider the following

```
A(n)
{
  if (n ≤ 1)
     return 1
  else
   return (n * A(n-1));
}
```

Time R.R

Sol: $T(n) = T(n-1) + c$

$= T(n-2) + ~~T(n-1)+~~ c + c \quad T(n-2) + c + c$

$= T(n-3) + ~~T(n-2) + T(n-1) + c + c + c~~ \; c + c + c$

⋮

~~n-1 n-2 n-3 .... 1~~ = nc

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

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

Value $T(n) = n\,T(n-1)$

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

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

⋮

~~(n-(k-1)) T(n-k)~~

$(n-(k-1))\,T(n-k) + c\,(n-(k-1) + n-(k-1) + \dots + 1)$

$n-k = 1$

$n-1 = k$

$2 + c(2+3+4+ \dots\dots$

Value R.R

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

$n \times (n-1) \times T(n-2)$

$n \times (n-1) \times (n-2) \times T(n-3)$

⋮

$n \times (n-1) \times (n-2) \times \dots (n-(k-1))\; T(n-k)$ put $n-k = 1$, $n-1 = k$

$n \times (n-1)(n-2) \dots (2) \cdot T(1)$

$= n!$

Multiplication R.R

$$T(n) = \begin{cases} 0 & \text{if } n \le 1 \\ T(n-1) + 1 & n > 1 \end{cases}$$

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

$= 0 + \underbrace{1 + 1 + \dots 1}_{n-1 \text{ times}}$

$= O(n)$

EX

```
A(n)
{
 if (n == 0 || n == 1)
     return (n)
  else
   return (A(n-1) + A(n-2));
}
```

Write R.R for

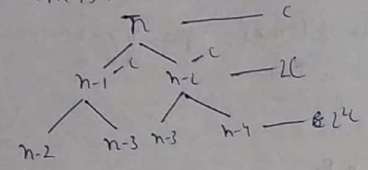
i) TC ? ~~T(n) = T(n-1) + T(n-2)~~

ii) Value?

iii) Add?

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

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



$= c(1 + 2 + 2^2 + 2^3 + \dots + 2^n)$

$= c[2^n]$

Value

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

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

$T(n) = 1[2^1 + 2^1 + 2^2 + \dots + 2^n]$

$= 1(2^n)$

$= O(2^n)$

Except

Comparing left & right right must be greater

Bubble Sort

i/p: 70 20 50 40 90 5 15
(1 2 3 4 5 6 7)

Pass P_1: 20 50 40 70 5 15 (90) — 6C. S — WC–6, BC–0

Pass P_2: 20 40 50 5 15 (70) 90 — 5C. S — WC–5, BC–0

P_3: 20 40 5 15 (50) 70 30 90 — S < 4

P_4: 20 5 15 (40) 50 70 30 3C — S < 3

P_5: 5 15 (20) 40 50 70 90 2C — S < 2

P_6: 5 15 20 40 50 70 90 1C — S → 0 Here we have zero swaps

stop

NOTE — using bubble sort

1) To sort n elements ∴ (n-1) passes are required.

2) Total comparisons $= n-1 + n-2 + n-3 + \dots + 2 + 1$
$= \frac{n(n-1)}{2} = O(n^2)$ · [E.C]

3) Total swaps — Worst case → For every comparison there is swap $= \frac{n(n-1)}{2} = O(n^2)$ [WC]
Best case = 0

4) Time Complexity = C + S
$= O(n^2)$ [Every case]

5) It is In-place & Stable algorithm

BC → i/p array sorted in increasing order [30 40 50 60 70 80]

WC → i/p array sorted in decreasing order [80 70 60 50 40 30]

While performing BS, in any pass no of swaps = 0
then stop the algorithm, bcz array is already sorted.
With this condition BS Best case O(n) [rarely occurs]

WC & AC $O(n^2)$

i/p 50 10 20 30 40

P1 10 20 30 40 50 S = 4

P2 10 20 30 40 50 S = 0 <- stop

Selection Sort

i/p : 80 20 5 43 91 3 25 17
1 2 3 4 5 6 7 8

P1 min=1 (3) (20 5 43 91 80 25 17) 7C 1S

P2 min=2 3 (5) (20 43 91 80 25 17) 6C 1S

P3 min=3 3 5 (17) (43 91 80 25 20) 5C 1S

P4 min=4 3 5 17 (20) (91 80 25 43) 4C 1S

P5 min=5 3 5 17 20 (25) (80 91 43) 3C 1S

P6 min=6 3 5 17 20 25 (43) (91 80) 2C 1S

P7 min=7 3 5 17 20 25 43 (80) 91 1C 1S

Swapping a=20 b=10 i) Using Temporary Variable iii)
t = 20
a = 10
b = t

ii) without using Temporary variable
a = a+b
b = a-b
a = a-b

NOTE:

i) To sort n-elements selection sort will take (n-1) passes

ii) To comparsion = (n-1) + (n-2) + ... + 2 + 1
Total = $\frac{(n-1)n}{2}$
= $O(n^2)$ [EC]

(Imp) Total Swaps = n-1 [EC] = O(n)

iii) TC = C + S
= $O(n^2)$ [EC]

Imp Selection sort minimize total no of swaps ie in every case no of swaps are (n-1)

iv) Stable & Inplace sorting algorithm

Insertion Sort (Important)

i/p: 50 70 90 75 55 100 150 40 78
1 2 3 4 5 6 7 8 9

Concept: Insert every new element at end and then sort

P1: 50 70 => 50 70

P2: 50 70 90 => 50 70 90

P3: 50 70 90 75 => 50 70 75 90

P4: 50 70 75 90 55 => 50 55 70 75 90

P5: 50 55 70 75 90 | 100 => 50 55 70 75 90 100

P6: 50 55 70 75 90 100 | 150

P7: 50 55 70 75 90 100 150 | 40 => 40 50 55 70 75 90 100 150

P8: 40 50 55 70 75 90 100 150 | 78 =>

P9: 40 50 55 70 75 78 90 100 150

Time Complexity

BC (Best Case) $T(n) \rightarrow O(n)$

i/p: 10 20 30 40 50 <- increasing sorted array

		C	S
P1	10 20 |	1	0
P2	10 20 30 |	1	0
P3	10 20 30 40 |	1	0
P4	10 20 30 40 50 |	1	0
		n-1	0

NOTE:

1) To sort n elements insertion sort algorithm will take n-1 comparisons and 0 swaps in BC.
$T(n) \rightarrow O(n)$ in BC

Imp

2) For BC, insertion sort is better than any sorting algorithm

3) If array already sorted or almost sort then Insertion sort algorithm will take O(n) time to sort that array. but quick sort will take $O(n^2)$

10 20 30 40 50 S

$O(n) + O(n) = O(n)$

i) Stable
ii) Inplace

-> Worst Case occurs if elements are sorted in reverse order

3) No. of ~~Swap~~ inversions in the given array = no of swaps in insertion sort (operations)

5
10 20 30 40 50 | 5

4) If at all in the given array at most n-inversions are there, then to sort that array apply insertion sort bcz array is almost sorted. So TC is $O(n)$

Worst Case TC = $O(n^2)$

i/p : 50 40 30 20 10

		C	S
P1	50 | 40	1	1
P2	40 50 | 30	2	2
P3	30 40 50 | 20	3	3
P4	20 30 40 50 | 10	4 / n-1	4 / n-1

total Comparisons = $(n-1) + (n-2) + \dots + 1$
$= O(n^2)$

total Swaps = $(n-1) + (n-2) + \dots + 1$
$= O(n^2)$

TC = C + S ⇒ $O(n^2)$

Avg Case

$\frac{n}{2} \cdot BC + \frac{n}{2} WC \Rightarrow \frac{n}{2} \cdot 1 + \frac{n}{2} \cdot n \Rightarrow O(n^2)$

(for 1 element) (for 1 element)

Gate Ex

10 20 30 40 50 60 70 | 15

In worst case if we apply BS to find correct position of 15 in sorted array. Then worst case TC of insertion sort will

Sol. 10 20 30 40 50 60 70 | 15

Using BS
C → $O(\log n)$ } for 1 element
S → $O(n)$

Using LS
C → n
S → n
n+n
n-element ⇒ n×2n = $O(n^2)$

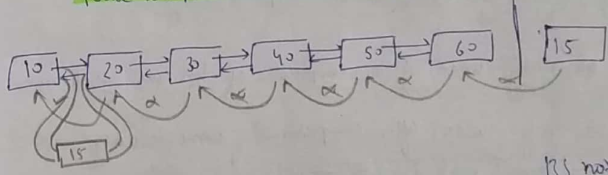
∴ for n element $n * (O(\log n) + O(n)) = O(n^2)$

No of swaps for every element is $O(n)$ hence WC TC does not change.

Imp!

Total Comparison = $n * O(\log n) = O(n \log n)$

Total Swaps = $n * O(n) = O(n^2)$



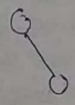
BS not possible on Linked List

for 1 element
C → n ⇒(n-element) $n * n = O(n^2)$
S → 0

↑ bcz it is doubly Linked List, we just need to adjust the pointers.

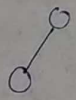
Heap Sort

A heap is almost Complete binary Tree.



BT ✓

✗ ACBT (without filling Left don't go to right)



BT ✓

ACBT ✓

BT ✓

✗ ACBT (without completing current level don't go to next level

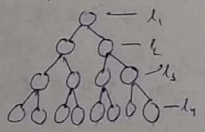
BT ✓

ACBT ✓

A binary Tree is said to be almost complete Binary Tree if

i) at every node, after completion of left only go to right.

ii) at every node, after completion of current level only go to next level.



← CBT

$2^0 + 2^1 + 2^2 + 2^3 \Leftarrow \frac{1(2^4-1)}{2-1} = 2^4-1$

1) If CBT contains contains k-level and n nodes

$n = 2^k - 1$

$k = \log_2(n+1)$ ← no of levels

2) Height = levels − 1 $= k-1$

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

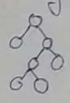
(or) almost BT

3) If CBT contains n-nodes

leaf nodes = $\lceil \frac{n}{2} \rceil$

non-leaf = $\lfloor \frac{n}{2} \rfloor$

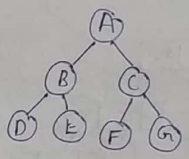
Strict BT → exactly 2 or 0 child



leaf-node – means 0 child

Internal node – means at least 1 child

Binary Tree Representations



A	B	C	D	E	F	G
1	2	3	4	5	6	7

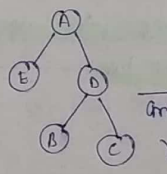
i^{th} position node

parent(i) = $\lfloor \frac{i}{2} \rfloor$

left-child = $2i$

right-child = $2i+1$

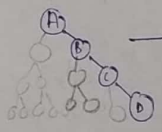
Ex



Array representation →

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

Ex

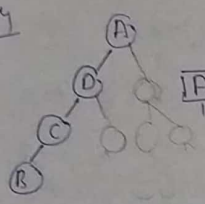


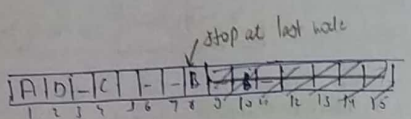
→

A	B			C										D
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Array size = $2^k - 1 = 2^4 - 1 = 15$

Ex:-1





Bt - n - nodes

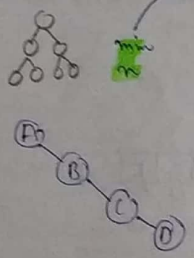
⇓

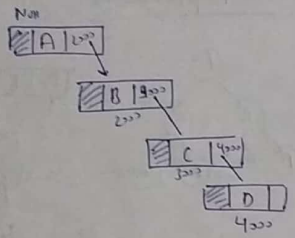
array → best when given tree is CBT or almost CBT

worst case skew tree

min n

max $2^n - 1$



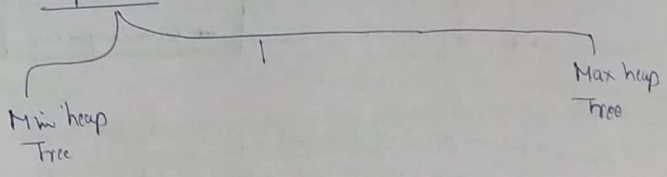


NOTE

Given BT is CBT or almost CBT, then array representation is best otherwise Linked List.

~~Width & Links not mentioned~~

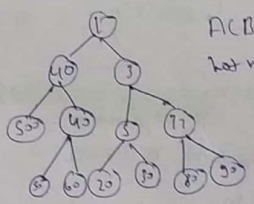
Heap Tree



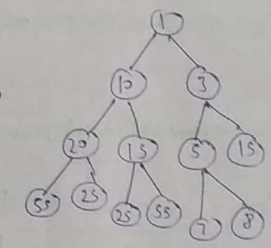
Min Heap Tree

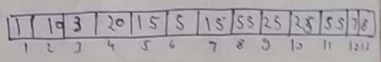
(or) CBT

In the given almost CBT & at every node root is minimum or equal & compared with children, then it is called Min Heap Tree.



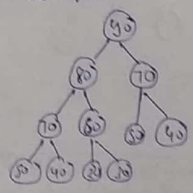
ACBT ✗
not min heap





Max Heap Tree

In the given ~~atleast~~ almost CBT (or) CBT at every node root is maximum or equal with children, then it is called Max Heap Tree



Imp

no of elements Comparisons fighting

1st min — 1 — 0 — O(1)

2nd min — 2 — 1 → O(1)

3rd min → 3 — 2 — O(1)

children of 2nd min
↳ last 2nd min position

Min Heap is used to find minimum elements at with cheaper costs

1st min → 100 — 99 → O(1) total comparisons = $\left(\frac{99 \times 100}{2}\right)$ ∴ O(1)

500th min → 500 — 499 → O(1) total comparisons = $\left(\frac{499 \times 500}{2}\right)$ = O(1)

To find Kth minimum element, K elements will fight

total comparisons = $\sum(k-1) = \frac{K(k-1)}{2}$ *

To find nth minimum (or) maximum element = $O(n^2)$

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

(or)

better

↳ apply bubble sort 1 pass to find (maximum element) T(→O(n)

To nth minimum

Min Heap Insertion (T(→O(logn)) for n elements.

Min Heapify Bottom (Child is working) or simply Min Heapify

N = 21 (array size)
M = 20 (No of elements)

O(logn)

BC → O(1)
WC → O(logn)
AC → O(logn) = O(logn) / 2 + O(logn)

new element

B(O(1))

10	20	30	40	50	60	70	80	90	100	110	120	130	140	150	160	170	180	190	200	5
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

10	20	30	40	50	60	70	80	90	100	110	120	130	140	150	160	170	180	190	5
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

always dividing by 2 T(→O(logn))

NOTE

Inserting a element into min Heap (or) max heap which already contains with take n elements will take O(logn) time (WC & AC)

O(1) time in BC

Min Heap Deletion

Min Heapify top

O(logn)

n = 20 (array size)
m = 20 (no of elements)

x = a[1]
a[1] = a[m]
m = m-1

C → 2(logn)
S → 1(logn)
3(logn)
O(logn)

10	20	30	40	50	60	70	80	90	100	110	120	130	140	150	160	170	180	190	200
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Here parent will work both the children

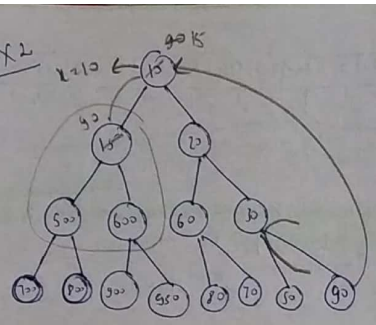
T(→ O(logn) [EC]

1st min will go last place bcz space there
2nd min will go to last 2nd last place bcz space there

Steps

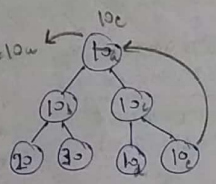
1) x = a[1]
2) a[1] = a[m]
3) m = m-1
4) Min Heapify

EX2



BC → O(1)

EX3



↙ Unstable

NOTE:

1) Deleting a element from min heap or max heap which contains already n elements will take O(log n) (WC & AC) and O(1) BC.

2) Min heap tree will give descending order on applying Heap Sort
Max heap tree will give ascending order on applying Heap Sort

3) It is in place ✱

4) It is not stable

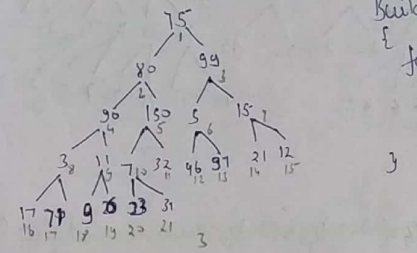
6/8/18

Mathematical Derivation (Imp Interview)

Build-Heap [Constructing min Heap with O(n) time] [BC]

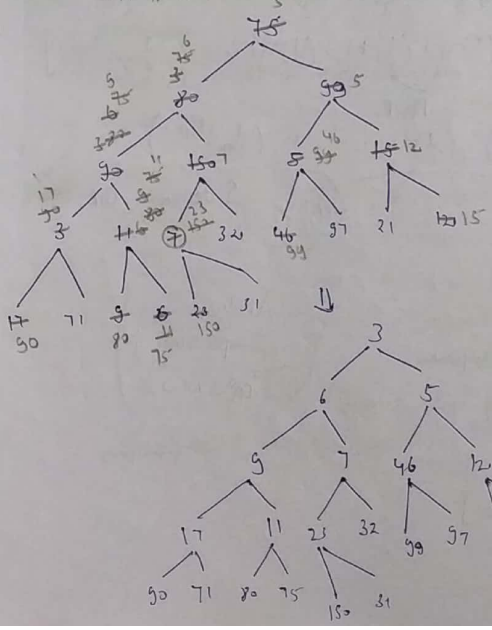
EX1

i/p : 75 80 99 90 150 5 15 3 11 7 32 46 97 21 12 17 9 6 23 31



```
Build Heap ( )
{
  for(i = ⌊n/2⌋ ; i ≤ 1 ; i--)
  {
    MinHeapify Top(i);
  }
}
```

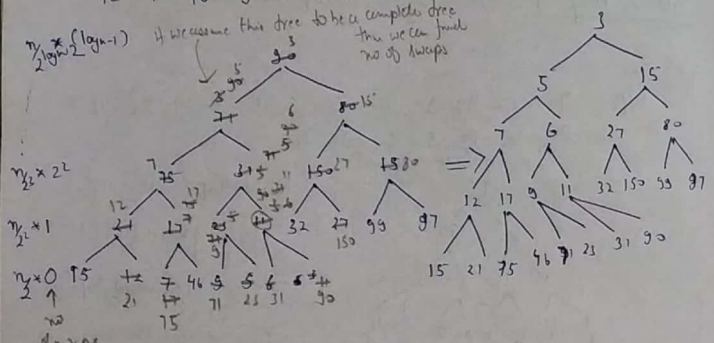
We start from non-leaf node.



Ex

90 71 80 75 31 150 15 21 17 23 11 32 27 99 97 15

12 7 46 9 5 6 3



Total Swaps = $n_{2^1} \times 1 + n_{2^2} \times 2 + n_{2^3} \times 3 + n_{2^4} \times 4 + \cdots + \frac{n}{2^{\log n}} \times (\log n - 1)$

$= n_2 \left[\left(\frac{1}{2}\right)1 + \left(\frac{1}{2}\right)^2 2 + \left(\frac{1}{2}\right)^3 3 + \left(\frac{1}{2}\right)^4 4 + \cdots + \frac{1}{2^{\log n - 1}} (\log n - 1) \right]$

AGP

$= \frac{n}{2} \left(\underbrace{\frac{1}{2} + \left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^3 + \cdots + \left(\frac{1}{2}\right)^{\log n - 1}}_{O(1)} \right)$ ← decreasing GP

$\frac{n}{2} \times O(1)$

$= \frac{n}{2}$ ✱

(Analysis Important for Interview)

TC = Swaps + Comparisons

$= \frac{n}{2} + 2 \times \frac{n}{2}$ → for every swap there are 2 comparisons

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

$= O(n)$ [EC]

Heap Sort Algorithm

I/p: An array of n-elements.

1) Build Heap (Using build Heap create max heap) → $O(n)$
2) Delete element one by one and store from RHS → $O(\log n) \times n$

Continue n times

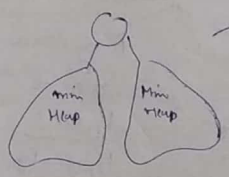
⇒ TC → $\underbrace{O(n) + O(n \log n)}_{O(n \log n)}$ [EC]

Note: If all the n elements are in the Heap are same, then TC → $O(n)$

1) Heap sort on Max-Heap will give in ascending order sorted Heap.
2) Heap sort on Min-Heap will give descending order sorted Heap

Note: Find Y on min → $O(1)$ 26 comparisons. ie 21 comparisons
Extract Y on min → $O(\log_2)$ Delete operation

Note: We have tree

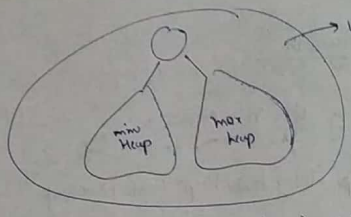


→ but overall not min Heap

to make overall min Heap → $O(\log n)$

Best Case → $O(1)$

Q Given



→ we have to make overall min Heap

Apply Build Heap on right → $O(n_2) = O(n)$

then apply min-heapify top at root → $O(\log)$

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

Q We have min Heap with n-elements, now n more elements are inserted into the min-Heap. Find TC ?

Sol. i) add n-more elements
ii) apply build Heap on (2n) elements
} → TC → O(n)

Min Heap Insertion

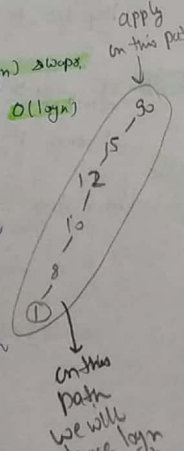
Usually - we have ($\log n$) comparison & $O(\log n)$ swaps

$T(n) \rightarrow (+S) \Rightarrow O(\log n) + \log n = 2\log n \rightarrow O(\log n)$

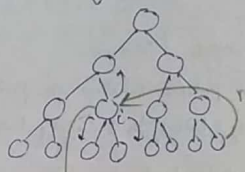
~~If we~~

Min Heap Insertion

LS	Binary Search
$\log n - C$	$\log\log n - C$
$\log n - S$	$\log n - S$
$2\log n$	$\log\log n + \log n$
⇓	⇓
$O(\log n)$	$O(\log n)$



delete ith element from Heap



it will ask both parent & children, hence it can go to top or bottom $T(n) \rightarrow O(\log n) \Rightarrow O(\log n)$

Sorting Techniques

Comparison based S.T

Non-comparison based S.T

S.T	BC	WC	AC
1. Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
2. Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
3. Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
4. Merge Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$
5. Quick Sort	$O(n\log n)$	$O(n^2)$	$O(n\log n)$
6. Heap Sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$

1. Counting Sort
2. Radix Sort
3. Bucket Sort

} $O(n)$ [EC]

If all elements are in the range $[1 \ldots n]$

apply Counting Sort for each digit

apply if all elements are in range $[1 \ldots n^k]$

If all elements are floating point numbers and are uniformly distributed across a range

Given NOTE: In all Comparison based S.T in WC best one is (lower bound) $O(n\log n)$

In all Comparison based Sorting techniques in BC ; best one is (lower bound) is $O(n)$

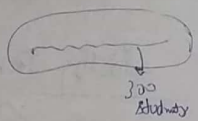
Greedy Technique

Basics of Greedy Technique

1. Solution Space : Set of all possible solutions over given n no. of i/p's.
2. Feasible Solution : Set of those solutions which satisfy our conditions.
3. Optimal Solution : It is one of the feasible soln which will optimize our goal. (MAX AVG). [Need not be unique] *

1 Solution space

$G_1\ G_2\ G_3 \ldots G_{300C_{10}}$ ← all possible solns



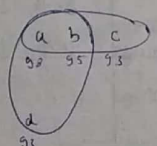
$^{300}C_{10}$ ways to select a group of 10

2) Feasible Solution

$G_1 G_2 G_3 \ldots G_{50C_5}$ ←

3) Optimal Solution

G_1 ← top 10 people



NOTE: Optimal solution need not be unique

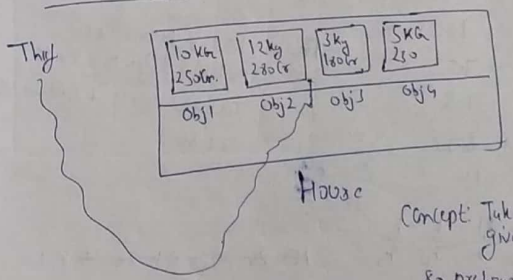
NOTE: Most of the problems in Greedy contains n no. of i/p's. And our objective is finding a subset (strict) which will satisfy our conditions, & which will optimize our goal.

Applications of Greedy

1) Knapsack problem [fractional]
2) Job Sequencing with deadline.
3) Huffman Coding
4) Optimal Merge Pattern
5) Minimum Cost Spanning Tree [MST]
   i) Kruskal
   ii) Prim's
6) Single Source Shortest Path
   i) Dijkstra's (all +ve edge weights)
   ii) Bellman ford (contains some -ve edge weights)
   iii) BFT (Breadth First Traversal) (no edge weights)

Knapsack Problem

M = 23



Thief has a bag with capacity 23. And he wants to maximize his profit

Concept: Take that obj which gives max profit per weight

obj1 → Profit/weight = $\frac{250}{10} = 25$

obj2 → $\frac{280}{12} = 23.3$

obj3 → $\frac{180}{3} = 60$

obj4 → $\frac{250}{5} = 46$

So preference order of objects is obj 3, obj 4, obj 1, obj 2

$\frac{1}{x_1}\ \frac{5/12}{x_2}\ \frac{1}{x_3}\ \frac{1}{x_4}$

Each position denotes fraction of object taken

M = 23 - 3 = 20 - 12 = 8 - 5 = 3

Profit gained

$1 \cdot 250 + \frac{5}{12} \cdot 280 + 1 \cdot 180 + 1 \cdot 230 =$

$250 + 116.6 + 180 + 230$

$= 776.6$

7/8/18

$M = 26$

$N = 7$

Objects	1	2	3	4	5	6	7
profit	75	25	40	30	150	19	55
weight	10	4	3	4	2	3	5

Merge sort(or) Heap sort $\Rightarrow O(n \log n)$

I $\Rightarrow O(n)$

Sol:
1 – Pi/wi = 7.5
2 – 25/4 = 5.75
3 – 40/3 = 13.33
4 – 30/2 = 7.5
5 – 150/2 = 7.5
6 – 19/3 = 6.33
7 – 55/5 = 11

We have multiple answers how

II Sort in decreasing Order

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

Profit = 1×75 + 1×40 + 1×30 + 1×15 + 2/3×19 + 1×55

= 75 + 40 + 30 + 15 + 12.66 + 55

= 227.66

$\frac{1}{x_1}$ $\frac{1/2}{x_2}$ $\frac{1}{x_3}$ $\frac{1}{x_4}$ $\frac{1}{x_5}$ $\frac{1}{x_6}$ $\frac{1}{x_7}$

M = 26 24 21 16 14 1

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

$\frac{1}{x_1}$ $\frac{0}{x_2}$ $\frac{1}{x_3}$ $\frac{1}{x_4}$ $\frac{1}{x_5}$ $\frac{2/3}{x_6}$ $\frac{1}{x_7}$

3, 7, 5, 4, 1

M = 26 23 +8 +6 +2 1

Steps to follow

	obj 1	obj 2	obj 3	obj 4	obj 5	obj 6	obj 7	
1) Pi/wi	7.5	5.75	13.33	7.5	7.5	6.3	11	$\Rightarrow O(n)$
Sort Decreasing order	3	7	1	4	5	6	2	$\Rightarrow O(n \log n)$

TC = O(n log n)

M = 26	P = 0
26 − 1×5 = 23	0 + 40×1
23 − 1×5 = 18	40 + 55×1 = 95
18 − 1×10 = 8	95 + 1×75 = 170
8 − 1×4 = 4	170 + 30×1 = 200
4 − 1×2 = 2	200 + 1×15 = 215
2 − 2/3×3 = 0	215 + 2/3×19 = 227.6

$\frac{1}{x_1}$ $\frac{0}{x_2}$ $\frac{1}{x_3}$ $\frac{1}{x_4}$ $\frac{1}{x_5}$ $\frac{2/3}{x_6}$ $\frac{1}{x_7}$

TC of knapsack = O(n log n)

i/p format

```
struct knapsack
{
    int objno;
    int profit;
    int weight;
};
```

NOTE: In greedy knapsack we will get optimal soln by giving priority to profit & weight

Job Sequencing With Deadline

Basics

1) single CPU
2) no preemption R.R fails
3) same-arrival time FCFS fails
4) One-unit running time to each job. SJF fails

EX:①

n=4

Jobs:	J_1	J_2	J_3	J_4
profit:	25	75	65	45
dead line:	2	1	2	1

All feasible soln possible

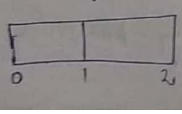
no work zero profit → ( ) ⇒ 0

$(J_2, J_1) = 100$	$(J_4, J_3) = 110$	$(J_1) = 25$
~~$(J_2, J_4) = X$~~	$(J_4, J_1) = 70$	$(J_2) = 75$
~~$(J_1, J_2) = X$~~	$(J_1, J_3) = 90$	$(J_3) = 65$
$(J_2, J_3) = 140$	$(J_3, J_1) = 90$	$(J_4) = 45$

→ optimal soln

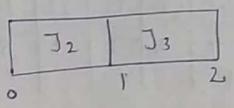
I → scan dead line. Find max deadline

II → Create an array of size = max deadline



III → sort in decreasing profit order

J_2 J_3 J_4 J_1
75 65 45 25



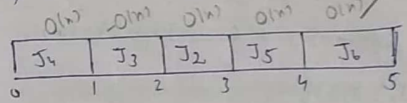
EX:2

n=7

Jobs	J_1	J_2	J_3	J_4	J_5	J_6	J_7
profits	25	75	85	55	65	45	35
deadlines	5	3	2	3	4	5	3

find max profit & sequence

Jobs with deadline ≥ 5 can fight for this position



sort in dec profit

J_3 J_2 J_5 J_4 J_6 J_7 J_1

Sequence → J_4, J_3, J_2, J_5, J_6

max profit → 55 + 85 + 75 + 65 + 45

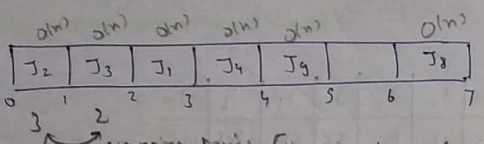
= 325

- TC ⇒ O(n)

EX3

$n=9$

Jobs	J_1	J_2	J_3	J_4	J_5	J_6	J_7	J_8	J_9
profits	55	50	65	60	45	40	50	55	70
deadlines	5	3	2	5	1	4	3	7	5



3 ↔ 2 swapping possible [If given Seq^n answer does not match with the option look for possible swaps]

Sort dec order profit

J_9 J_3 J_4 J_1 J_8 J_2 ~~J_7~~ ~~J_5~~ ~~J_6~~

Seq → $J_2, J_3, J_1, J_4, J_9, J_8$

profit → $50 + 65 + 55 + 60 + 70 + 55 = 355$

Jobs leftout → J_7, J_5, J_6

TC for Best Case → $O(n \log n)$

Worst Case → $O(n^2)$

Note: Sorting is done based on profit, but searching is done based on deadline. Hence we have to go for Linear Search. We can not apply Binary Search.

Huffman Coding

1) Encoding Technique
2) Data Compression Technique

$n = 100$

$a = 45$ $b = 1$
$c = 6$ $d = 30$
$e = 15$ $f = 3$

i) Uniform coding

ASCII
a - 45 × 8
b - 1 × 8
c - 6 × 8
d - 30 × 8
e - 15 × 8
f - 3 × 8

100 char → 800 bits

1 char (Avg) = $\frac{800}{100}$ = 8 bits/1 char

But we can Compress it further using Huffman coding

ii) Non-Uniform Coding

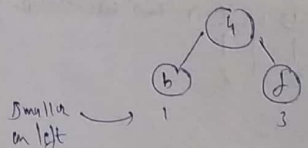
iii) More Freq char ⇒ use less bits

iii) less freq char ⇒ use more bits

- Build 2-way Huffman ~~Encoding~~ Encoded Tree

Step-1: take first 2 min and add them.
Store the result back

Step2: repeat this procedure until complete tree is built

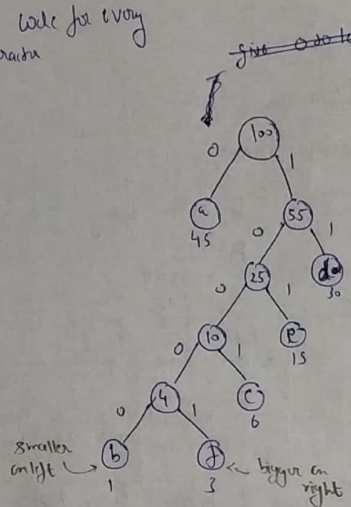


Smaller on left

Huffman code for every character

a = 0
b = 10000
c = 1001
d = 11
e = 101
f = 10001

~~first 0 to left & 1 to right~~



We always assume 0 to left branch & 1 to right branch, if labelling of edges not mentioned in the question.

If ans does not match go for opposite labelling.

Avg no of bits / char = $\frac{1\times45 + 5\times1 + 4\times6 + 30\times2 + 15\times3 + 5\times3}{100}$

* always divide by root value

$= \frac{194}{100}$

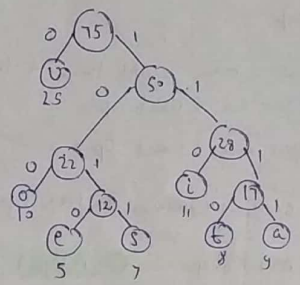
$= 1.9$ bits/char

EX2 n=7

m = (a, e, i, o, u, s, t)
f = (9, 5, 11, 10, 25, 7, 8)

~~n=7 ie 7:0~~

f: 5, 7, 8, 9, 10, 11, 25
e s t a o i u



Huffman coding
u = 0
o = 100
e = 1010
s = 1011
t = 1110
a = 1111
i = 110

Avg bits / char = $\left(5\times4 + 7\times4 + 8\times4 + 9\times4 + 10\times3 + 11\times3 + 25\times1\right) / 75$

$= \frac{20 + 28 + 32 + 36 + 30 + 33 + 25}{75}$

$= \frac{204}{75}$

$= 2.07$ bits/char

Steps

~~B~~ → Every time 2 min elements are we are taking so go for Min-Heap.

I → Build-Heap

II → ~~Z~~ Extract 2 min, add them & then return result.

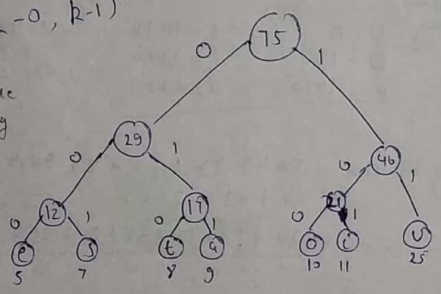
$O(\log n) \times 2$ ← 2 delete opn

$O(\log n) \times 1$ ← 1 insert opn

Here in every iteration ⊛ $3 * O(\log n)$ work done.

$T(n) \rightarrow n + (n-1)\,3\log n = O(n \log n)$

assume (k=0, k=1)



To extract 1st min selection sort will take $O(n)$ time while using Min Heap will take $O(n) + O(\log n)$ time

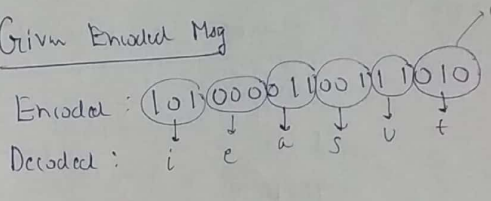
Huffman code

e = 000
s = 001
t = 010
a = 011
o = 100
i = 101
u = 11

Avg bits/char = $\frac{200}{75}$

Note: - prefix of one code can not be Huffman code for another character. (for security purpose & removing ambiguity)

Given Encoded Msg



8/8/18

Optimal Merge Pattern (Here we have to minimize no of record movements)

Ex)

⇒ Collection of records

n = 3 (3 files)

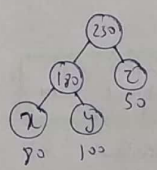
x = 80

y = 100

z = 50

[2-way merge Tree] - there are 3 ways $3_{C_2} = 3$

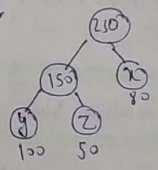
I



no of record movements

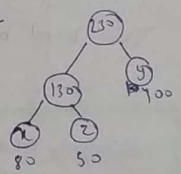
180 + 230 = 410

II



record mov = 150 + 210 = 380

III



record mov. 130 + 230 = 360

II

Optimal merge pattern

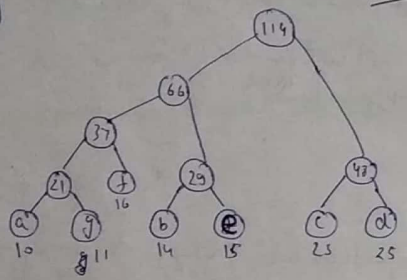
Ex2

n = 7

a = 10
b = 14
c = 23
d = 25
e = 15
f = 16
g = 11

2-way Merge Tree

eg



min
Worst record mov = 21 + 29 + 48 + 37 + 66 + 114
= 315

Minimum record movements = sum of all internal nodes

build-heap
TC (O.M.P) = n + (n-1) 3 × log n → 2 deletion, 1 insertion
= O(n log n)

Minimum Cost Spanning Tree (MST)

Graphs G(V, E)
| | set of edges
set of vertices

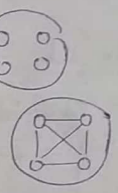
Note: In n vertex simple graph max deg of any vertex is n-1 & min deg is 0.

In n vertex ~~any~~ multi graph max deg of any vertex is ∞ & min deg is 0.

Types of simple graphs

For TC point of view

for min TC think about Null Graph
for max TC think about Complete Graph →



Note:
Let G(V, E) be a simple Graph

$|E| \leq \frac{V(V-1)}{2}$

$|E| \leq C_1 \cdot V^2$

$|E| = O(V^2)$

$E = O(V^2)$

$\log E = O(2 \log V)$

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

With n vertices, how many simple Graphs possible

$^nC_2$ edges are there, & each edge has 2 options, either present or absent

$$^nC_0 + {}^nC_1 + {}^nC_2 + \cdots + {}^nC_{nC_2} = 2^{^nC_2}$$

$^nC_0$: SG with 0 edge; $^nC_1$: SG with 1 edge; $^nC_2$: SG with 2 edges

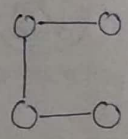
$\frac{n(n-1)}{2}$ ← most possible simple Graphs

EX: $n=4$

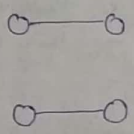
$$\max\text{-}SG = 2^{4C_2} = 2^6 = 64$$

Undirected Graphs

1) Connected

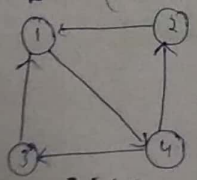


2) Disconnected Graph



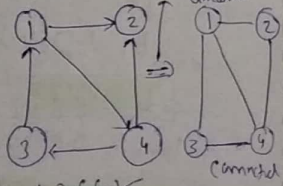
Directed Graph

1) Strongly Connected (b/w every pair of vertices there exists a path)



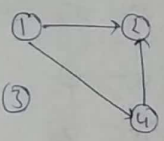
SCG ✓
WCG ✓

2) Weakly Connected Graph.



WCG ✓

4 is directed graph after removing direction becomes connected.



this graph is neither SCG nor WCG

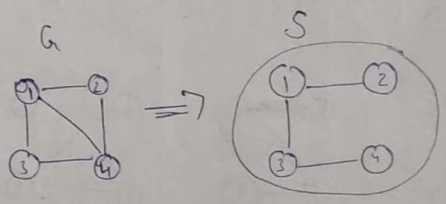
SCG ✗
WCG ✗

Spanning Tree

A subgraph S of the given graph G(V,E) is said to be Spanning Tree iff

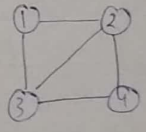
1) S should contain all vertices of G
2) S should contain V-1 edges. Where V = vertices.
3) S can not contain cycle. (seems doing same thing) ie we are doing extra work

EX1

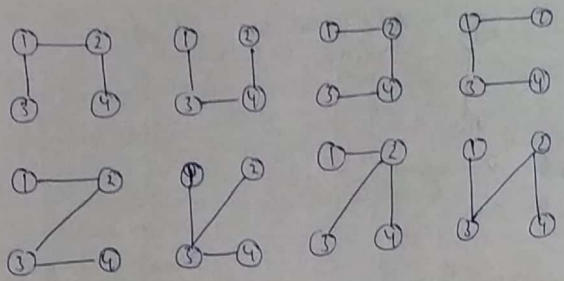


NOTE: MST possible only for Undirected Graph

EX2

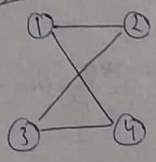


find all possible Spanning tree

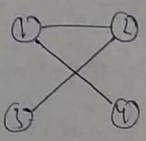


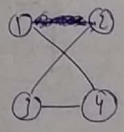
Always remember → look for cycles

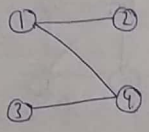
EX3

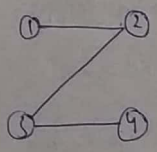


find no of Spanning Trees

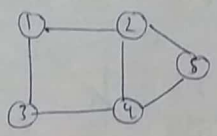




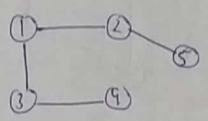


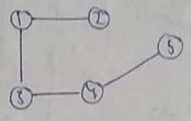


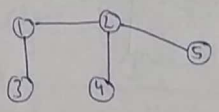
EX4

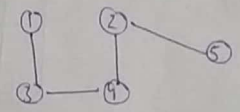


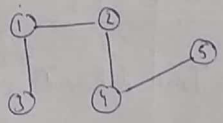
find no of Spanning Trees.

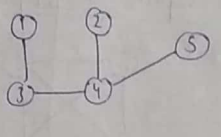


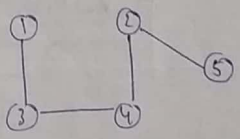


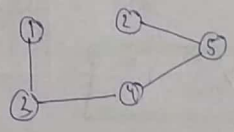


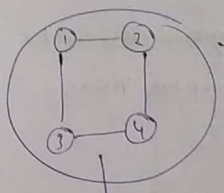










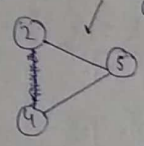


2 Ways 2 connect to these 4 cycles

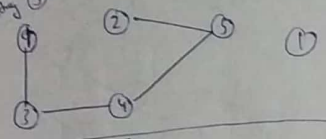
total we get 4+4+2 cycles

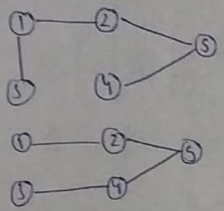
Join (4 we will gets 4 S.T.

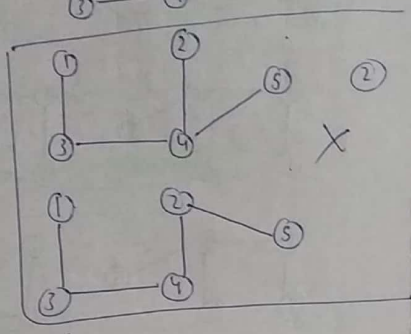
3 more possible for



both edges are present connecting



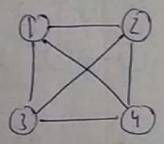




Ex:

for Complete ~~Graph~~ Graph

K_4 =



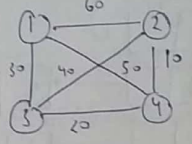
Total S.T = n^{n-2}

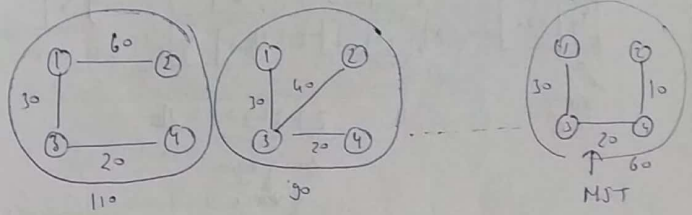
$ST(K_4) = 4^{4-2} = 4^2 = 16$

$ST(K_n) = n^{n-2}$

Note: ST ϕ is not possible if Graph is disconnected because we can never cover disconnected vertex include

Minimum Cost ~~S~~ ST





To construct MST we have 2 algorithms

i) Prims

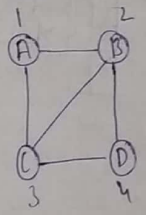
ii) Kurus Kal's

~~Prims Algorithm~~

Basics

Graphs Representations

Ex:



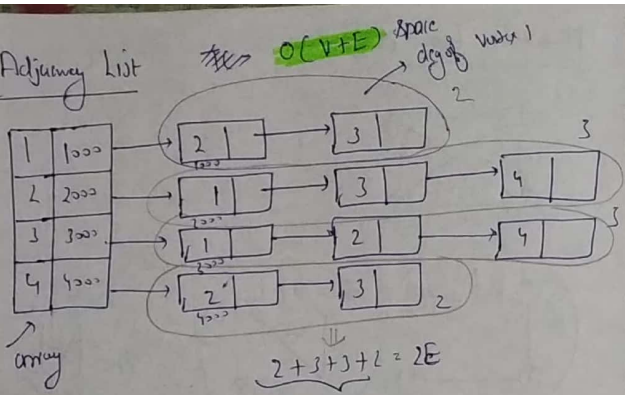
Adjacency Matrix

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

$O(V^2)$ Space

$(E \cdot C)$

Adjacency List



$2+3+3+2 = 2E$

Sum of degree of vertex

$V + 2E \Rightarrow O(V+E)$

$O(E)$

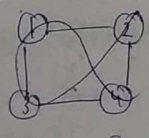
Imp

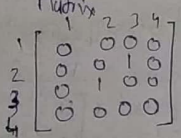
	Matrix	List
1) V_1 & V_2 adjacent	$O(1)$	$O(1)$, $O(V)$
2) V_1 → degree	$O(V)$	$O(1)$, $O(V)$
3) no of edges	$O(V^2)$	$O(V+E)$

3) Scan the matrix row by row and count all 1's → this will give sum of deg of all vertices. Divide this by 2 to get no of edges

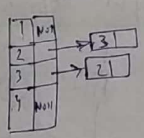
3) added up all length of lists corresponding to every array index. This will give sum of degree of edges. Divide this by 2

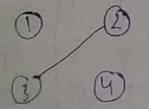
ex-2





adj List





---

* For Graph with more edges (i.e complete Graph) or for Dense Graph go for adj. Matrix representation

* For Graph with less edges (or) sparse graph go for adj List Representation.

Grate

B) Given a directed Graph, a vertex is said to be a Universal ~~Sink~~ sink iff

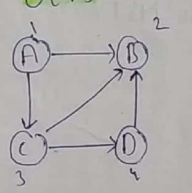
1) In-degree = V-1
2) Out-degree = 0

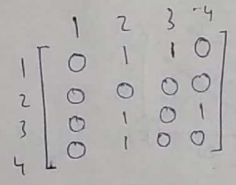
Grate

C) How much time it will take to find universal sink if given graph is represented using adj Matrix?

Sol: $O(V)$

This is a sparse matrix (means non-zero entries are very less)





~~1~~ — 2 ⇒ 1 → means out-deg of 1 = 1 hence 1 can not be sink bcz sink has 0 out-deg

2 — ~~2~~ ⇒ 0 → means out-deg of 2 = 0 (hence it may be sink bcz out deg of sink is 0) (*) we can say that In-deg of 3 is 0. Hence 3 can not be sink bcz its In-deg should be V-1.

2 — ~~3~~ ⇒ 0

In-deg of 4 is 0, hence it can not be sink.

Hence 2 is sink. T(→ O(V).

(While comparing if vertex has outdegree it can not be sink)

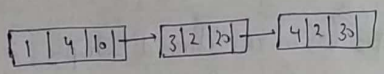
NOTE: to find sink, look for vertex with non-zero outdegree and we try to eliminate them

Similarly to find universal sink in adjacency list

$O(V+E)$ [W.C]

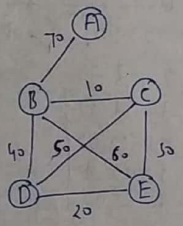
To represent sparse matrix go for linked lists representation

	1	2	3	4
1	0	0	0	10
2	0	0	0	0
3	0	20	0	0
4	0	30	0	0



9/8/18

Kruskals Algorithm



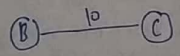
find MST?

Algo

i) Build a Min Heap of edges – $O(E)$

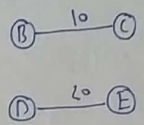
ii) Take 1st min edge 10_{BC} – $O(\log E)$

put it into MST



iii) Take next min edge 20_{DE} – $O(\log E)$

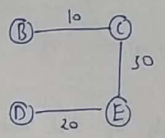
add to MST



iv) Take next min edge → from this step there is always cycle checking

add to MST if no cycle

30_{CE} – $O(\log E)$ + $O(\log E)$ ← cycle checking



v) Repeat previous step

1) 40_{BD} ✗ $(\log E)$ + $O(\log E)$

2) 50_{DC} ✗ $(\log E)$ + $O(\log E)$ } time wasted

3) 60_{BE} ✗ $(\log E)$ + $O(\log E)$

4) 70_{BA} ✓

Time Complexity

one extra $\log E$ for cycle checking

Best Case $2\log E$

$E + (V-1)(\log E)\times 2)$

$E + V\log E$

$O(E + V\log V)$

if dense graph $E = O(V^2)$ $T(\quad) = O(V^2)$ if sparse graph $T(\quad) = O(V\log V)$

Worst Case

We might end up deleting all the edges

$E + E(\log E)$

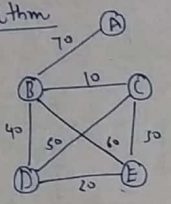
$O(E\log E)$ ← $(E + E(\log V))$

$O(E\log V)$

from step (iii) we have cycle checking also using Disjoint Sets

Prims Algorithm

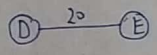
~~It is~~ tak.

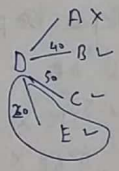


find MST!

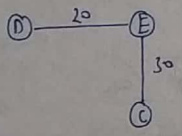
Algorithm

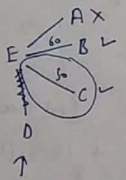
1) Take any vertex and find adj of that vertex





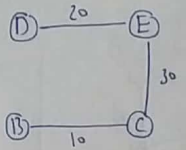
2) Find adj vertices to newly added vertex and take min these and previous

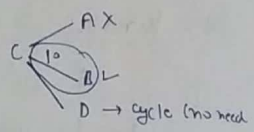




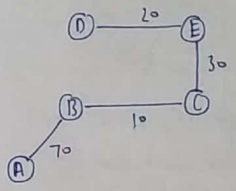
↑ no need to check

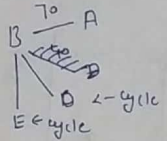
3) Find adj vertices to newly added vertex and take min these and previous if no cycle





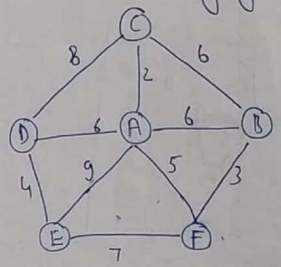
4) Repeat previous step



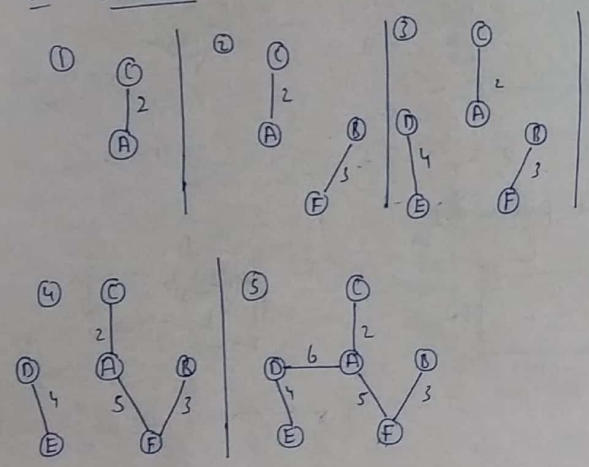


Note: In every step of Prims algorithm we will get connected graph but in Kruskals algorithm we may get disconnected graph at some steps. And finally both will give connected graphs only

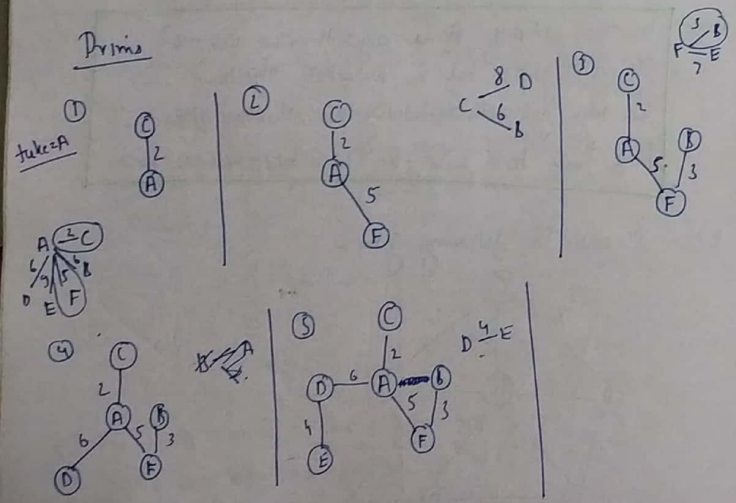
Ex) Consider the following graph



Sol: Kruskal



Prims



Ex1

Prims - Algo - Edge Sequence

(A) (E,D), (D,A), (F,B), (A,F) (A,C) — (F,B): no adj

(B) (A,C) (A,F) (F,B) (A,D) (D,E) adj satisfied

(C) (A,C) (A,F) (D,E) (A,D) (F,B) — (D,E): no adj

(d) (A,C) (A,F) (F,B) (D,E) (A,D) — (D,E): no adj

(e) (A,C) (A,F) (A,D) (D,E) (F,D) adj satisfied, now go for min property also the edge B & E

First check Adjacency property

if more than one answer satisfy the adj property then go for min property

option c) fails min property (A,C) (A,F) (A,D) X

Ex3

Let Gr be an Undirected Weighted Connected Complete Graph with distinct edge weights, e_{min} be the min edge weight, e_{max} be the max edge weight among all edge weights

check T/F?

a) Gr - Connected Unique - MST T

b) Every MST of Gr - must contain e_{min} T

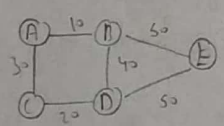
c) " " of Gr - must contain e_{max} F

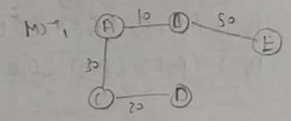
d) " " Gr may contain e_{max} T

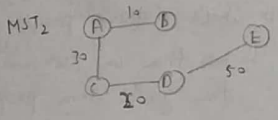
e) If e_{max} is in MST then its removal from Gr must disconnect Gr. T

Note: i) If for a given graph more than 1 ~~MST~~ MST possible then in that graph same edge weights should repeat. (Necessary condition)

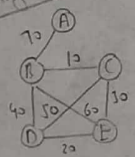
ii) For the given graph more than 1 MST may be possible but cost must be same.

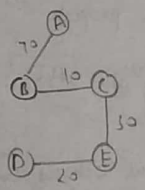






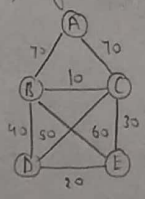
Assuming for option a)

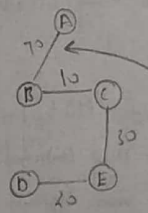




Here we have no other option but to take edge with weight 70 because it is the only edge that covers A. If we don't take it graph will be disconnected.

If edges are not distinct weights





Here if we remove Cmax G will not get disconnected bcz we can cover A by A — C

EX-4 U – Undirected W – weighted C – connected Gr – graph

Let Gr be an UWCG with n-vertices, w be the min weight among all edge weights. and e be a specific edge with weight w.

Check T/F?

a) Gr contains U – MST F

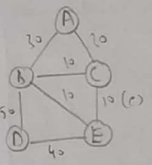
b) Every MST of Gr must contain e F

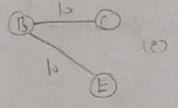
T c) " " " " " atleast one edge with weight w

d) Some MST's of Gr may contain e T

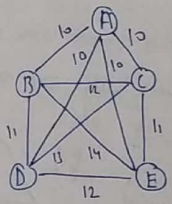
e) If e is not in MST then in that cycle all edges contain weight w T

Ex:



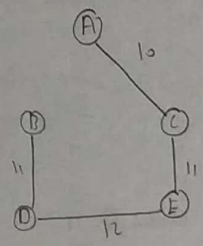


Ex 5 Consider the following Graph



What is cost of MST for this Graph?

a) What will cost of MST for above graph ② in such a way that A is leaf node. ? given undirected graph → deg = 1; given directed graph → In-deg = 1, Out-deg = 0



Cost = 44

Ex:6

Let G be an graph with n-vertices shown below ~~in form of~~ by an adj matrix in which

(i) all diagonal elements — 0's

(ii) all non-diagonal elements — 1's

Check T/F

a) G contain U-MST with cost n-1 F

b) G- cannot have MST F

F c) G- Contain multiple MST's ~~each of cost~~ with diff cost

d) G " " " " Each with cost n-1 T

Sol:

$$\begin{bmatrix} 0 & 1 & 1 & \cdots & 1 \\ 1 & 0 & 1 & \cdots & 1 \\ \vdots & & & & \vdots \\ 1 & 1 & 1 & \cdots & 0 \end{bmatrix}_{n \times n}$$

⇒ Complete graph

Ex:

n=3

n=4

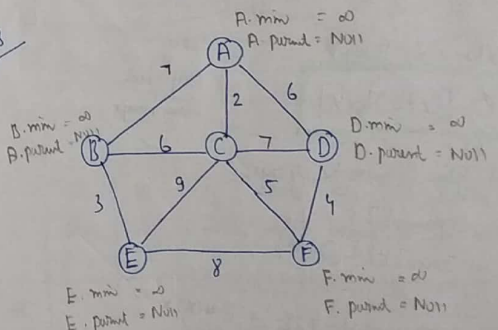
Time Complexity of Prims

Basic Concept

Decrease key & increase key operation's in ~~Min~~ max Heap or ~~max~~ min Heap will take O(logn) (WC & AC) & O(1) in BC

Decrease key	Data structure
O(1)	Unsorted array
O(n)	Sorted array
O(1)	Unsorted double linked list
O(n)	Sorted double linked list
O(logn)	Min Heap
O(logn)	Max Heap

10/8/18



A.min: means A adjacent better i.e. better node than where A

Present in Min Heap

	A	B	C	D	E	F	
pick any vertex	ϕ0 N	∞ N	∞ N	∞ N	∞ N	∞ N	O(V)
say A, set A min=0		7 A	2 A	6 A	∞ N	∞ N	3 + 3 log V
(log V) A							
(log V) C		6 C		6 A	9 C	5 C	5 + 4 log V
(log V) F		6 C			8 F		3 + 2 log V
(log V) D					8 F		3 + 0 log V
(log V) B							3 + 1 log V
(log V) E							3 + 0 log V

delete

- min Heap
- search for adjacent nodes. 3 nodes updated ie 3 decrease key operations

E degree $\leq$ deg(V) $\Rightarrow$ 2E

V log V

E log V

✓ for constructing min Heap

✓ explore degree of vertex

T_C (prims) = O(V) + V log V + 2E + E log V

= V log V + E log V

$T_C = O[(V+E) \log V]$

Using Adj. List Min Heap

if given connected graph

min (E) = V-1

max (E) = $O(V^2)$

ii) Using Adj List Array

$T_C = V^2 + 2E + E * O(1)$

$= O(V^2)$

iii) Using Adj Matrix Sorted Doubly linked list

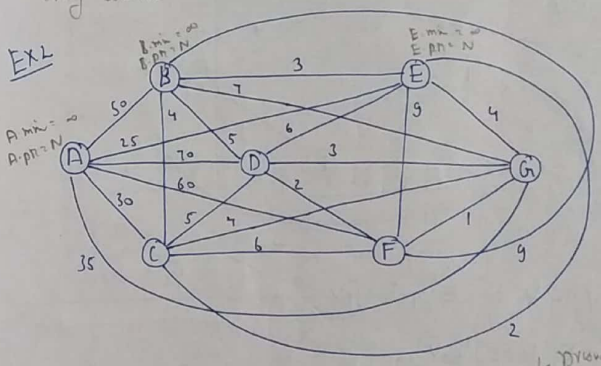
$T_C = V * O(1) + V * O(V) + E * O(V)$

$= V^2 + EV$

$= O[V^2 + EV]$

To get min MST look for last values of each node in every column

Ex:

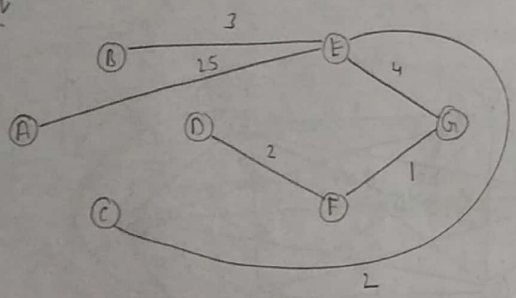


↓ present in min Heap

	A ϕ0 N	B ∞ N	C ∞ N	D ∞ N	E ∞ N	F ∞ N	G ∞ N	O(V)
(log V) A		50 A	30 A	70 A	25 A	60 A	35 A	⇒ 6 + 6 log V
(log V) E		3 E	2 E	6 E		9 E	4 E	6 + 5 log V
(log V) C		3 E		5 C		6 C	4 E	6 + 4 log V

	A	B	C	D	E	F	G	
C		3 E		5 C		6 C	4 E	$6+4\log V$
B				5 C		6 C	4 E	$6+3\log V$
G				3 G		1 G		$6+2\log V$
F				2 F				$6+1\log V$
D								$6+0\log V$

$V \log V$ ↓ $2E$



Cost = 37

TC → $V \log V + 2E + E \log V$

$= O[(V+E) \log V]$

ii) Using Adj Matrix
Sorted Array

$V \times O(V)$

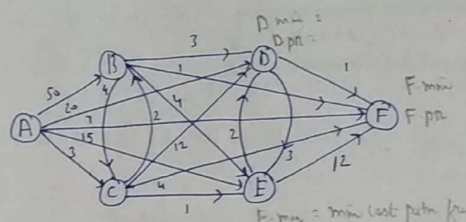
TC = $V * O(1) + VE + E * O(V)$

$= V + V^2 + EV$

$= O(V^2 + EV)$

Single Source Shortest Path

Ex)



E.min = min cost path from source to E
E.parent = Null

manually

A → A = 0
A – B = 5
A – C = 3
A – D = 6
A – E = 4
A – F = 6

Note: While Calculating E.min = previous cost + cost of edge added
C.min + edge weight (C–E)
= 3 + 1
= 4

Using Adj List Min Heap

Dijkstra's Algorithm — Present in Min Heap

	A	B	C	D	E	F	
source	0	∞	∞	∞	∞	∞	$O(V)$
	N	N	N	N	N	N	
$\log V$ A		50 A	3 A	20 A	15 A	7 A	$\Rightarrow 5 + 5\log V$
$\log V$ C		5 C		15 C	4 C	7 A	$\Rightarrow 4 + 4\log V$
$\log V$ E		5 C		6 E		7 A	$\Rightarrow 2 + 2\log V$
$\log V$ B				6 E		6 B	$\Rightarrow 4 + 2\log V$
$\log V$ D						6 B	$\Rightarrow 2 + 1\log V$
$\log V$ F							$0 + 0\log V$

directed graph → O / E $E \log V$

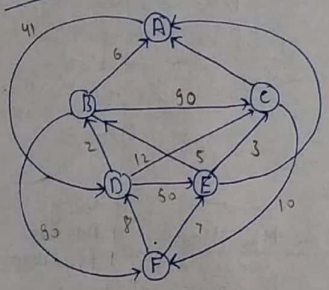
TC (Dijkstra's) = $\sum_{V+}$ V log V + E + E log V

= V log V + E log V

TC = O[(V+E) log V]

To get the path check the final parent (ie last value in the column)

EX 2



Q1) Print the sequence of vertices identified by Dijkstra's algo. When algo. starts from Vertex B.

BADCFE

Q2) What will be cost of shortest path from B to E?

Cost (B-A-D-C-F-E) = 76

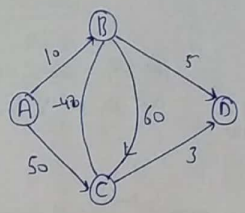
Q3) What will be shortest path from B to E?

B-A-D-C-F-E

1

	A	B	C	D	E	F
	∞ N	∞ N	∞ N	∞ N	∞ N	∞ N
B	6 B		90 B	41 N	∞ N	90 B
6 A			90 B	47 A	∞ N	90 B
47 D			59 D		97 D	90 B
59 C					97 D	69 C
69 F					76 F	
E						

EX 3



Manually

A-A = 0

A-B = 2

A-C = 50

A-D = 7

Applying Dijkstra's Algorithm

Priority in Min Heap

	A	B	C	D
	∞ 0 N	∞ N	∞ N	∞ N
A		10 A	50 A	∞ A
10 B			50 A	15 B
15 D			50 A	
C				

To check the wrong vertices (ie vertices for which we get wrong cost) Whenever Dijkstra's says not to visit a particular vertex, go and visit that vertex and check whether you get better path or not.

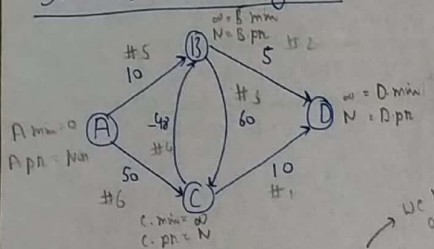
Relaxation operation on a vertex A

→ Whichever vertices are adjacent to A, perform decrease key operation.

In Dijkstra's algorithm every vertex is relaxed exactly once

Here we are going vertex by vertex, by performing relaxation operation on every vertex once

Bellman - Ford Algorithm



It will cover all the edges. V times

I step → give edge no's. Present in Array

We have taken array here we are not taking min every time

1 length, 2 length better

	A	B	C	D
all 1 length path	0	∞	∞	∞
	N	N	N	N
all 2 length path, n=1	0	10	50	∞
	N	A	A	N
n=2	0	2	50	15
	N	C	A	B
n=3	0	2	50	7
	N	C	A	B
n=4	0	2	50	7
	N	C	A	B

decrease key in array

E * O(1) = E

E * O(1) = E

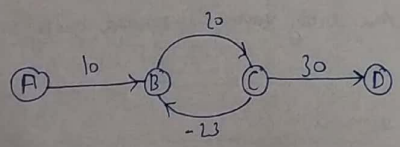
E * O(1) = E

E * O(1) = E

TC = O(VE)

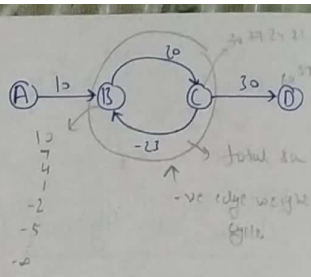
algorithm will do V rounds.
Here V-1 rounds are required.
last round (ie one more extra round) is verification round.

EX2



Sol: Manual

A - A = 0
A - B = 10 - ∞
A - C = - ∞
A - D = - ∞



~~Apply Bellman ford~~

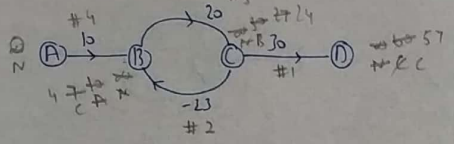
A	B	C	D	
0	∞	∞	∞	Present in array
N	N	N	N	
0	-∞	-∞	-∞	← Manual results.

depend on -ve edge weight cycle
(it) part of -ve edge weight cycle

Dijkstra Algorithm

A	B	C	D	
0	∞	∞	∞	Present in Min Heap O(V)
N	N	N	N	
A	10 A	∞ N	∞ N	log(V)
10 B		30 B	∞ N	log V
30 C			60 B	log V
60 D				log V

How B, C & D are wrong answer but Dijkstra did only 1 round.



Bellman ford Algorithm (V-rounds)

	A	B	C	D
	0	∞	∞	∞
	N	N	N	N
n=1	0	10	∞	∞
	N	A	N	N
n=2	0	10	30	∞
	N	A	B	N
n=3	0	7	27	60
	N	C	B	C
n=4	0	4	24	54
	N	C	B	C
		-∞	-∞	-∞

} proved in array

Note:

→ In verification round value should not change but here values changed. This indicate presence of -ve weight edge cycle.

B, C, & D values changed, have they are part of -ve weight edge cycle (or) depend on it.

→ A is not part of -ve weight cycle. bcz its value did not change

Note:

1) if the graph contain all +ve edge weights, Dijkstra's algorithm will always give correct answer. but Dijkstra's may fail if graph contain -ve edge weights.

2) If the graph contain +ve (or) -ve (or) both then Bellman ford will always give correct answer.

3) If graph contain -ve edge weight cycle then Bellman ford algorithm will give correct answer for every vertex. but for some vertices answer is -∞ (Undefined)

Note:

4) if graph is Complete Graph

$TC = O(EV) = O(V^3)$ [$\because E = O(V^2)$]

Note:

5) Bellman ford will detect -ve edge weight cycle only if it is reachable. (all) from the source

BFT (Breadth First Traversal)

if the graph is unweighted graph we can find out SSSP (Single Source Shortest Path) using BFT $TC \Rightarrow O(V+E)$

→ TC for computing Transitive closure of a binary relation on set of n-elements. is $O(n^3)$.

→ Radix sort sorts the elements digit by digit. Each sorting takes $O(n)$ time. M digits takes $O(mn)$ time.
m = no of digits
n = no of elements.

→ Comparison sort makes $\lceil \log n! \rceil$ comparisons to sort n elements. This algorithm gives minimum no. of comparisons to sort n elements.

→ Priority Queue is ~~always~~ implemented using ~~Heap~~ Heap (min or max)

→ To implement Dijkstra's algorithm for linear running time complexity "Queue" data structure is used

→ Shortest path from source to node to every other node is computed by Breadth First Search efficiently in terms of time complexity

→ If every edge weight is same in a weighted graph then BFS will search efficiently shortest path from source to destination.

→ DFS of undirected graph will always have tree edges and back edges, but will never have forward edge and cross edge

Average case of linear search occurs when the item to be searched is in some where middle of array.

→ Average height of BST is $O(\log n)$

Dynamic Programming

Gr T : Cover few possibilities	; take less time	Sometimes wrong answer
D P : Cover all possibilities	; takes more time	Every time correct answer

Applications of Dynamic Programming

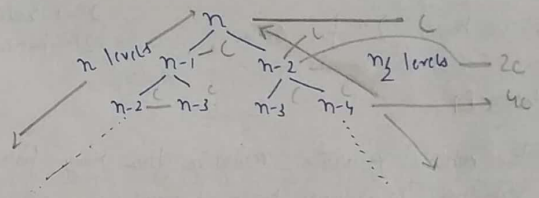
1) Fibonacci Series
2) Largest Common Subsequence
3) Matrix Chain Multiplication
4) 0/1 knapsack problem
5) Sum of Subset Problem
6) All pair shortest Path
7) Optimal Cost BST

Fibonacci Series

n	0	1	2	3	4	5	6	7	8
fib(n)	0	1	1	2	3	5	8	13	21

```
fib(n)
{
  if (n==0 || n==1)
      return n;
  else
      return (fib(n-1) + fib(n-2));
}
```

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



$T(n) = 2^0C + 2^1C + 2^2C + \ldots\ldots 2^nC$

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

R.R value

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

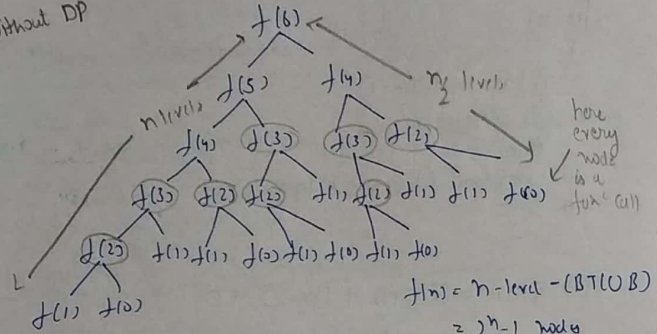
Space

i/p + extra

2B + nB

$= O(n)$

Without DP



$f(n) = n\text{-level} - (BT(OB))$

$= 2^n - 1$ nodes

$= 2^n$ fun calls

Here we have 2^n fun calls.

~~if we use DP~~

→ In the above recursive recursive tree many fun calls are repeating [which are known as overlapping sub problems]

→ What is the need of computing same fun again and again.!

→ In DP we compute only distinct fun calls. bcz as soon as we compute any fun we store its value in ~~some~~ data structure so that we can reuse it if it is needed after words

✱ In fibonacci of n how many distinct fun calls are there?

✱ Ans: $f(6) \to f(5) \to f(4) \to f(3) \to f(2) \to f(1)$, $f(0)$

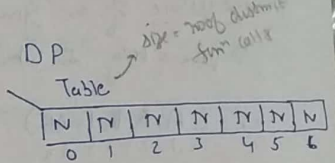
∴ $f(n) = n+1$ distinct fun calls.

$f(n) = n+1 - DFC$ ↓ every fun call cost is constant

$= (n+1) * O(1)$

$= O(n)$

$T(n) = O(n)$ with DP




```
DP-fib(n)
{
  if(n == 0 || n == 1)
      return n;
  else
  {
    if(Table[n-1] == NULL)
      Table[n-1] = DP-fib(n-1);
    if(Table[n-2] == NULL)
      Table[n-2] = DP-fib(n-2);
    else
      Table[n] = Table[n-1] + Table[n-2];
    return Table[n];
  }
}
```

To find TC here we can not write R.R bcz it is conditional Recursion. Here find out distinct fun calls.

TC = no of distinct fun calls.

With DP - Space

i/p + extra → stack → table

$2B + nB + \cancel{nB}\ (n+1)B$

$O(n)$

Longest Common Subsequence (LCS)

Subsequence of a given sequence is just the given sequence only in which 0 or more symbols left out.

S = (A, B, B, A, B, B)
1 2 3 4 5 6

SS_1 = (A, A)
1 4

SS_2 = (B, B, B, B)
2 3 5 6

SS_3 = ( )

SS_4 = (A, B, B, A, B, B)

SS_5 = (B, A, B, A) ← not subsequence
3 4 5
relative position of elements should not change.

Common Subsequence

X = (A, B, B, A, B, B)
1 2 3 4 5 6

Y = (B, A, A, B, A, A)
1 2 3 4 5 6

CS_0 = ( ) CS_1 = (A) CS_2 = (A, B)

CS_3 = (AAB) ⇒ LCS.

EX2

X = (A, B, A, B, A, B)

Y = (B, A, B, A, B, B)

CS_0 = ( ) CS_1 = (A) CS_2 = (A, B) CS_3 = (A, B, A)

CS_4 = (B, A, B, B) CS_5 = ~~(A, B, B, B)~~ (B, A, B, A, B) ⇒ LCS

How to find out LCS?

LCS(m, n) = Length of longest common subsequence possible with the two sequences X & Y where X contain m symbols and Y contain n symbols. and we are at mth symbol. and we are at nth symbol

EX3

X = (B A B A B A)
1 2 3 4 5 6

Y = (B A A B A)
1 2 3 4 5

LCS(6, 5) = 1 + LCS(5, 4)
1 + LCS(4, 3)
1 + LCS(3, 2)
2 ← max { LCS(3, 1), LCS(2, 2) }
BAB → 1
BA, BA → 2

Recursive Program

```
LCS(m,n)
{
  if (m==0 || n==0)
      return 0;
  else
  {
    a = LCS(m-1,n)
        LCS(m,n-1)

LCS(m,n)
{
  if (m==0 || n==0)
      return 0;
  else
    if (x[m]==y[n])
        return 1+LCS(m-1,y-1)
    else
    {
      a = LCS(m-1,n)
      b = LCS(m,n-1)
      c = max(a,b)
      return c;
    }
}
```

best case min (m,n)

c constant time

R.R for Value

$$T(m,n) = \begin{cases} 0 & \text{if } m==0 \text{ or } n==0 \\ 1+T(m-1) & \text{if } x[m]==y[n] \\ \max\{T(& \end{cases}$$

$$T(m,n) = \begin{cases} 0 & \text{if } m==0 \text{ or } n==0 \\ 1+T(m-1,n-1) & \text{if } x[m]==y[n] \\ \max\{T(m-1,n), T(m,n-1)\} & \text{otherwise} \\ & \text{i.e. } x[m] \neq y[n] \end{cases}$$

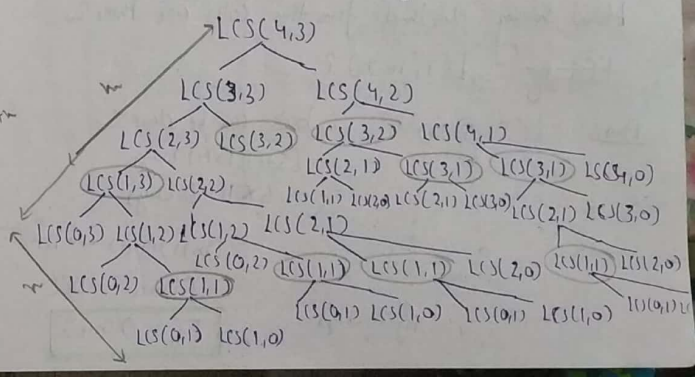
R.R for Time

$$T(m,n) = T(m-1,n) + T(m,n-1) + C$$

For Best Case

$$T(m,n) = \min(m,n)$$

ex $x = (A_1, A_2, A_3, A_4)$ $y = (B_1, B_2, B_3)$



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

$= 2^0C + 2^1C + 2^2C + \dots + 2^{m+n}C$

$= C[1 + 2 + 2^2 + 2^3 + \dots + 2^{m+n}]$

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

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

Space

i/p + extra (stack size)

$(m+n)B$ + $(m+n)B$

$= O(m+n)$

In the above recursive tree many fun calls are repeating, hence we will go to DP. which will solve only distinct fun calls.

How many distinct function calls are there in ~~tfs~~ LCS(m,n) ?

Ans:- LCS(5,10)

If 5 reduce to (ie decreases) 0

5 → 4 3 2 1 0 → 5+1 ways

10 → 9 8 7 6 5 4 3 2 1 0 → 10+1 ways

both can be done in $(5+1)(10+1)$ ways

$6 \times 11 = 66$ ways

$\therefore (m+1)(n+1) \rightarrow DFC$

$O(mn) \rightarrow DFC$

$T(= O(mn)$

```
DP-LCS(m,n)
{
  if(m==0 || n==0)
      return 0;
  else
  {
    if(x[m]==y[m])
    {
      if(Table[m-1][n-1] == NULL)
      {
        Table[m-1][n-1] = LCS(m-1,n-1)
        ~~Table[m][n] = (1+Table[m-1][n-1]);~~
      }
      Table[m][n] = 1 + Table[m-1][n-1];
      return Table[m][n];
    }
    ~~return Table[m-1][n-1]~~
    else
    {
      if(Table[m-1][n] == NULL)
          Table[m-1][n] = DP-LCS(m-1,n)
      if(Table[m][n-1] == NULL)
          Table[m][n-1] = DP-LCS(m,n-1)
      Table[m][n] = max(Table[m-1][n], Table[m][n-1])
      return Table[m][n];
    }
  }
}
```

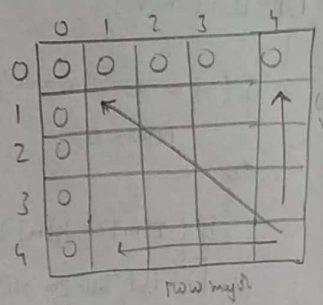
Space

i/p + extra

m+n S T

m+n mxw

= O(mn)



If both matching fill diagonally
If both not matching every time fill either column wise or row wise.

0/1 Knapsack Problem

Ex

M = 10

Objects :	obj 1	obj 2	obj 3
profit :	130	90	60
weight :	7	6	4

manually

obj 1	obj 2	obj 3	
0	0	0	= 0
0	0	1	= 60
0	1	0	= 90
0	1	1	= 150 ⇒ Optimal (best)
1	0	0	= 130
1	0	1	= X
1	1	0	= X
1	1	1	= X

Greedy

obj 1 → 18.56
obj 2 → 15
obj 3 → 15

(1 0 0)
x1 x2 x3

profit = 130

0/1 knapsack will give wrong answer, so we will use DP which will cover every possibility.

Ex

M = 16 N = 5

objects :	obj 1	obj 2	obj 3	obj 4	obj 5
profit :	25	75	80	35	55
weight :	3	4	3	7	3

OIKS(m,n) = maximum profit we will get in 0/1 knapsack problem where capacity of knapsack is M and no of objects are N.

R.R for value

i) OIKS(m,n) = 0 if m == 0 or n == 0
16, 0
0, n
0, 0

ii) OIKS(m,n) = OIKS(m,n-1) if w[n] > m
(weight of obj is more than available weight in knapsack)

iii) OIKS(m,n) = max { OIKS(m-w[n], n-1) + P[n] (weight we took here ← profit we got)
                     { OIKS(m, n-1) if w[n] < m

```
01KS(m,n)
{
  if (m==0 || n==0)
      return 0;
  else
  {
    if (w[n] > m)        ⇒ best (we can) O(n)
        return 01KS(m,n-1);
    else
    {
      a = 01KS(m-w[n], n-1) + P[n]
      b = 01KS(m,n-1)        } n definitely decreasing (always)
      c = max(a,b);          so stack size O(n)
      return c;              i.e we get n level tree
    }
  }
}
```

TC ⇒ $T(m,n) = T(m-w[n], n-1) + T(m, n-1) + C$

$= 2^0C + 2^1C + 2^2C + 2^3C + \dots + 2^nC$

$= C[2^0 + 2^1 + 2^2 + \dots + 2^n]$

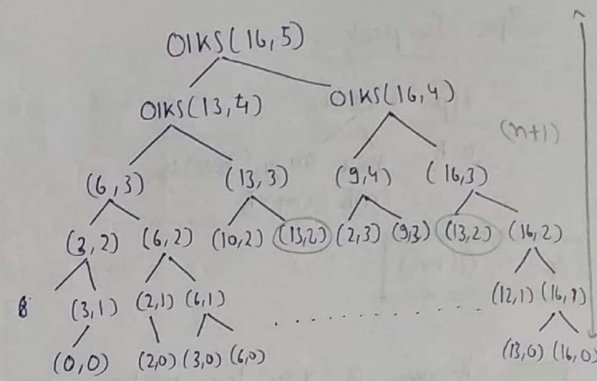
$= C \cdot 2^n$

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

Space Complexity

i/p + extra + stack

$(3n)B + (n)B \Rightarrow O(n)$



In the above recursive tree same fun calls are repeating so we will go to DP. which will solve only Distinct fun calls.

How many distinct fun calls are there in 01KS(m,n) ?

Ans: 01KS(m,n)

16 5
15 4
14 3
13 2
12 1
11 0

5+1 ways

0

16+1 ways

(m+1)(n+1) → DFC

O(mn) → DFC

$TC = O(mn)$

Space Complexity

i/p + extra

(n)B — stack (n)B — array (Table) size (mn)B

Space = O(mn)

Note: Because of very less repetitions TC of 0/1 knapsack using DP O(mn) approximately equal to $O(2^n)$

i.e. $O(mn) \cong O(2^n)$

Hence 0/1 knapsack is considered to be one of the N-P Complete problems.

Sum of Subsets Problem

i/p : Set of n integers (all +ve integers) and an integer m

o/p : Find any subset, so that its sum m.

ex: [50 10 30 70 100 25 35 60 40 80]
1 2 3 4 5 6 7 8 9 10

12/8/18

SOS(m,n) = finding a subset from the given array of n elements so that its sum is m, and currently I am at nth element.

initially SS = { } ← subset we need to find

i) SOS(m,n) = SS
0,5
0 0

ii) SOS(m,n) = error. ⇒ O(n)
20,0

iii) SOS(m,n) = SOS(m,n-1) if A[n] > m ⇒ Best case
(if current element is greater than required sum)
300, 10
300, 9
⋮
300 0

iv) SOS(m,n) = { SOS(m-A[n], n-1), SS ∪ A[n] if A[n] ≤ m ⇒ worst case
SOS(m,n-1)

In worst case
⇓
n-level - CBT

$T(m,n) = T(m-A[n], n-1) + T(m,n-1) + C$

$= 2^0C + 2^1C + 2^2C + \dots + 2^nC$

$= C[2^0 + 2^1 + 2^2 + \dots + 2^n)$

$= C.2^n$

$\sim O(2^n)$

Space ⇒ i/p + extra

n n ⇒ O(n)

Using DP ~~we can complete only distinct functional calls.~~

TC ⇒ SOS(m,n)

m-1 n-1
m-2 n-2
m-3 n-3
⋮ ⋮
0 0

Loop ⊢ m+1 n+1 ways

TC = O(mn)

Space ⇒ i/p + extra

(m)B ⊢ T

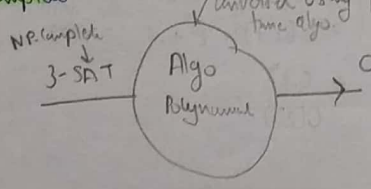
O(n)B O(mn)B

SC = O(mn)

Note: both OIKS & SOS are same;

OIKS polynomial time reduced to SOS problem

NOTE: OIKS is Polynomially converted into SOS problem that means OIKS is NP-Complete so SOS is also NP-Complete



Matrix Chain Multiplication (Optimal Parenthesization)

EX1 $A_{2\times3}$ $B_{3\times4}$ $C_{2\times4}$

$C_{2\times4} = A_{2\times3} * B_{3\times4}$

↓

C contains 8 elements ⇒ 2×3×4 = 24 multiplications required

EX2 $A_{2\times5}$ $B_{5\times7}$ $C = A_{2\times5} * B_{5\times7}$

↓

2×7 elements

2×7×5 = 70 multiplications required

EX3 $A_{3\times4}$ $B_{4\times2}$ $C_{2\times5}$

D = ABC

(A×B)×C ↓ 24 + 30 = 54 OR A×(B×C) ↓ 60 + 40 = 100

# of ways we can parenthesis the given matrix chain $\frac{2nCn}{n+1}$

put n-1 value in place of n

EX4 $A_{5\times2}$ $B_{2\times3}$ $C_{3\times4}$ $D_{4\times2}$ E = ABCD ?

A(B(CD)) (AB)(CD) (ABC)D

A(B(CD)): 20 + 12 + 24 = 56 Ans

A(B(CD)): 24 + 16 + 20 = 60

(AB)(CD): 30 + 24 + 30 = 84

((AB)C)D: 30 + 60 + 40 = 130

(A(BC))D: 40 + 24 + 40 = 104

EXS

Way of parenthesizing ↓

$A_{1\times2}$ $B_{2\times3}$ $C_{3\times1}$ $D_{1\times2}$

E = ABCD
E = (A (BC)) D

8+4=12 ← A(B(CD)) (10)
(AB)(CD): 6+6 + 6 = 18
(ABC) D: 8+2 = 10

(BC) D: 6 + 2 … ⑧ 10
B(CD): 6, 12+6 = 18
(AB)C: 6+3 = 9
A(BC): 2+6 = ⑧ 2

MCM(1, n) = the min of no. of multiplications required to multiply 1 to n matrices.

Assume Matrix M_i has P_{i-1} rows & P_i columns

M_i: P_{i-1}, P_i M_{10}: P_9, P_{10} $M_5 M_6 M_7 M_8$ ⇒ size = $P_4 \times P_8$ (resultant matrix size); P_4 P_5 … P_7 P_8

$$MCM(1,4) = \min \begin{cases} MCM(1,1) + MCM(2,4) + P_0 * P_4 * P_1 \\ MCM(1,2) + MCM(3,4) + P_0 * P_4 * P_2 \\ MCM(1,3) + MCM(4,4) + P_0 * P_4 * P_3 \end{cases}$$

$$MCM(i,j) = \begin{cases} \min \{ MCM(1,k) + MCM(k+1,j) + P_{i-1} * P_j * P_k \quad \forall k \; i \le k \le j-1 \\ ~~0 \quad if \; i == j~~ \\ 0 \quad if \; i == j \end{cases}$$

~~MCM(i,j)~~ = { MCM(…

$$MCM(i,j) = \min \begin{cases} MCM(i,i) + MCM(i+1,j) + P_{i-1} * P_j * P_i \\ MCM(i,i+1) + MCM(i+2,j) + P_{i-1} * P_j * P_{i+1} \\ MCM(i,i+2) + MCM(i+3,j) + P_{i-1} * P_j * P_{i+2} \\ \vdots \\ MCM(i,j-1) + MCM(j,j) + P_{i-1} * P_j * P_{j-1} \end{cases}$$

⇓

$$MCM(i,j) = \begin{cases} 0 & if \; i == j \\ \min(MCM(i,k) + MCM(k+1,j) + P_{i-1} * P_j * P_k) & if \; i < j \\ \forall k, \; i \le k \le j-1 \text{ or } \boxed{i \le k < j} \end{cases}$$

Will generate n-level n-ary tree

O(n) time [out of n finding minimum]

NOTE:
i) MCM(1,n) will generate n level n-ary tree.

R.R for TC

n fun calls each of size (n-1)

O(n) takes to find minimum among n

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

$= n^1 + n^2 + n^3 + n^4 + \dots + n^n$

$= O(n^n)$ [UB]

n-levels ⇑ Stack space = n

Space

i/p + Extra

we need only size of matrix: 4nB, (n)B

= O(n) → row & column 2B 2B

Using DP (Dynamic Programming)

TC Time Complexity

MCM (1, n)

2 n-1
3 n-2
4 n-3
.
.
n-1 2
n 1

n * n - DFC

How we have n^2 - DFC
Each fun call will take O(n) time

$TC = O(n^2) * n^2 = O(n^3)$

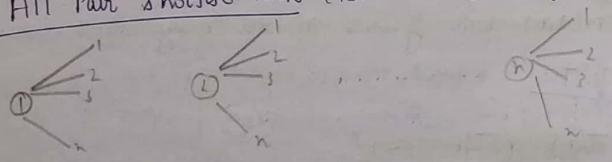
$TC = O(n^3)$

Space Complexity
i/p + extra
(4n)B
S T
n + n^2

$= O(n^2)$

$SC = O(n^2)$

All Pair Shortest Path (Sudo code from Gfg)



i) +ve edge weights
|V| * Dijkstra's
|V| * (V+E) log V) Adj List Min Heap

ii) -ve edge weights
|V| * Bellman Ford
|V| * (VE)

iii) Using Un weighted Graph
= |V| * (V+E)
(|V| * BFT) ← apply V times BFT
i.e

iv) Floyd Warshall Algorithm Using DP

$TC = O(V^3)$ [+ve -ve] works for both +ve & -ve edge weights.

NOTE: This algorithm will not give correct result if graph contains -ve edge weight cycle.

Tree Traversals & Graph Traversals

1) Tree Traversals

i) Pre Order (Root LST RST)
ii) In Order (LST Root RST)
iii) Post Order (~~Root~~ LST RST Root)

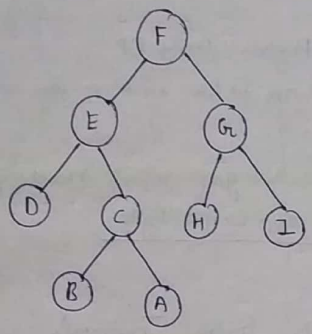
```
PreOrder(Root)          => T(n)
{
  Print(Root->data);    => O(1)
  PreOrder(LST);        => T(n/2)
  PreOrder(RST);        => T(n/2)
}
T(n) = 2T(n/2) + c
T(n) = O(n)
```

```
InOrder(Root)
{
  InOrder(LST);
  print(Root->data);
  InOrder(RST);
}
```

```
PostOrder(Root)
{
  PostOrder(RLST);
  PostOrder(RST);
  print(Root->data);
}
```

NOTE: In Order, PreOrder, Post Order on n nodes binary tree will take O(n) time. [EC]

EX1

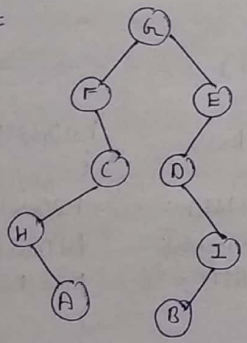


Pre: F E D C B A G H I

Post: D B A C E H I G F

In : D E B C A F H G I

EX2



Post:

Post: A H C F B I D E G

Pre: G F C H A E D I B

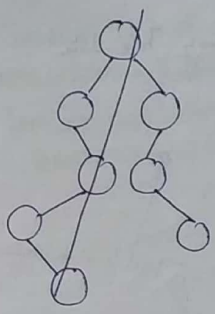
In : F H A C G D B I E

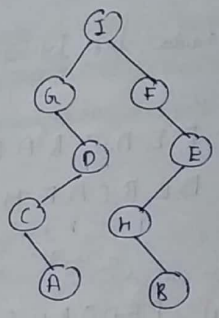
Stack size

min → log n (balanced)

max → n (unbalanced)

EX4





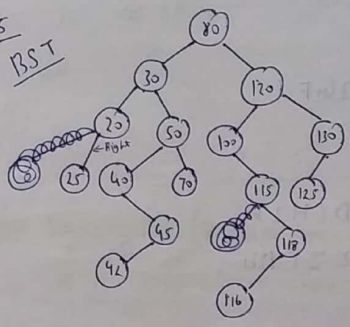
In: G C A D I F H B E

Pre: I G D C A F E H B

Post: A C D G B H E F I

EX5

BST



NOTE: (Greate)

Inorder Traversal of BST always gives Ascending order.

Pre: 80 30 20 25 50 40 45 42 70 120 100 115 118 116 130 125

Post: 25 20 42 45 40 70 50 30 116 118 115 100 125 130 120 80

In: 20 25 30 40 42 45 50 70 80 100 115 116 118 120 125 130

EX1 Consider the following binary Tree data

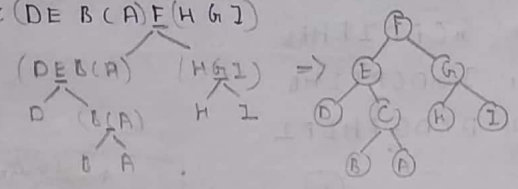
Pre: F E D C B A G H I

In: D E B C A F H G I

Post: ?

Sol: Pre: F E D C B A G H I → Find root → O(1)

In: (D E B (A) F (H G I)) → Find root in Inorder → O(n)



Post: → D B A C E H I G F

EX2:

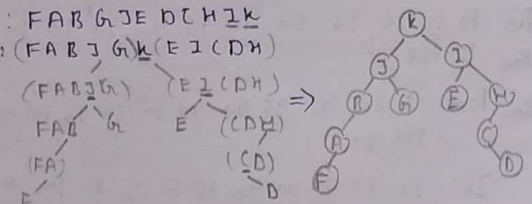
Post: F A B G I E D C H I K

In: F A B I G K E I C D H

Pre:

Sol: Post: F A B G I E D C H I K

In: (F A B I G) K (E I C D H)



Pre: K I B A F G I E H C D

NOTE: To create BT for the given (Preorder, Inorder) or (Postorder, Inorder) will take $O(n^2)$ time [Worst Case]

(Here we apply n times Linear Search to find out LST & RST)

EX-3 Consider the following ~~BST data~~ Binary Tree data

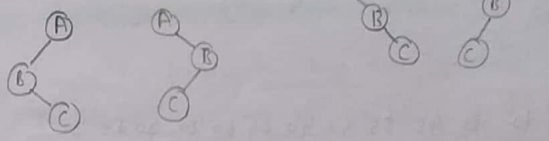
Pre: A B C

Post: C B A

In: ?

Sol: Pre: A B C

Post: C B A



Great NOTE: For a given (Preorder and Postorder) Unique BT not possible

To get unique binary Tree Inorder compulsory

Pre + In → BT possible → UBT possible → Unique

Post + In → BT ✓ UBT ✓

Pre + Post → BT ✓ UBT ✗

Unique BT not possible

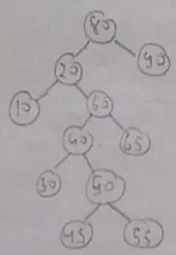
EX4 Consider the following BST data

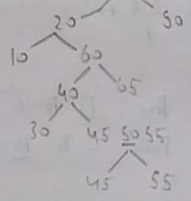
Pre : 80 20 10 60 40 30 50 45 55 65 90

Post : ?

Sol: Pre: 80 20 10 60 40 30 50 45 55 65 90

In : (10, 20, 30, 40, 45, 50, 55, 60, 65, 80, 90)





Post: 10 30 45 55 50 40 65 60 20 90 80

NOTE: Here root finding O(1) time, then in Inorder apply binary search to find root exact position in O(logn) time.

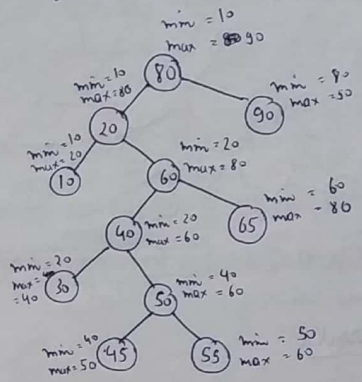
T(n) = n * O(logn) = O(nlogn) [n time apply BS]

better Method — O(n) time.

Keep track of min & max.

Step 1 → find min & max O(n) time

Step 2 → ask every node & keep track of min & max



When going Right min will change

When going left max will change

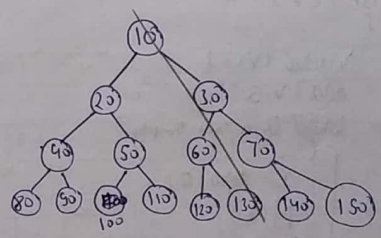
NOTE: To construct BST O(n) is taken.

Ex: Consider the following Min Heap data Tree data.

In: (120 60 110 50 100 70 80 10 90 30 130 20 140 40 150)

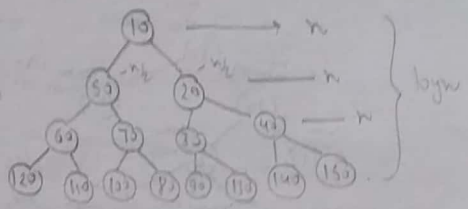
Post : ?

Sol:-



To find, not O(n) time is fine, but it is big

In: 120 60 110 50 100 70 80 10 90 30 130 120 140 40 150



T(n) = O(n log n)

Graph Traversals

1. BFT
2. DFT

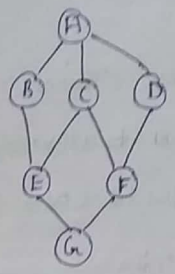
BFT (Breadth First Traversal) [It uses Queue]

$T(n) = O(V+E)$

```
BFT(V)
{
    visited(v) = 1
    add(v, Q);          => (over vertices runs V times)
    while(Q is not empty)
    {
        x = delete(Q)
        pf(x)           => (over edges runs 2E times)
        for all w adj to x
        {
            if(w is not visited)
            {
                visited(w) = 1
                add(w, Q);
            }
        }
    }
}
```

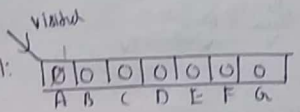
O(V+E)

Ex:



O/P → A B C D E F G

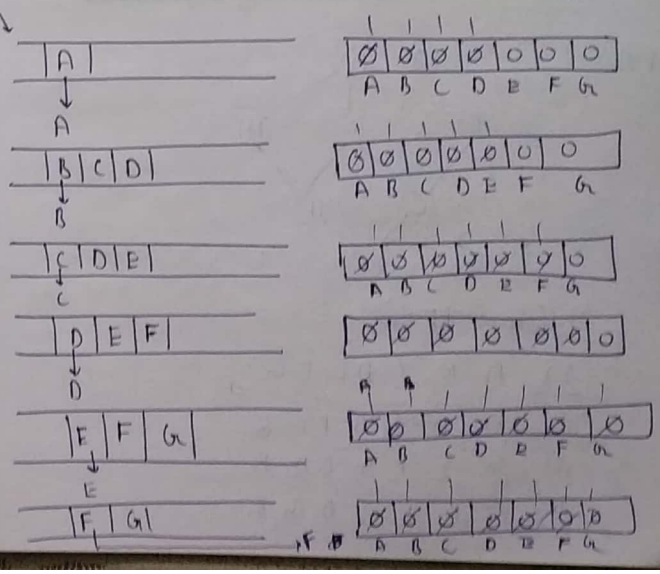
Sol:



for every vertex maintain visited array to avoid visiting same element twice.

BFT will use Queue

Visited array



NOTE:

1) BFT uses Queue data structure.
2) TC of BFT is equal to $O(V+E)$ [adjacency List]
3) TC of BFT is equal to $O(V^2)$ [using adj Matrix]
4) SC of BFT i/p + extra

list ⇓ Queue ⇓ + array ⇓ = ⇓

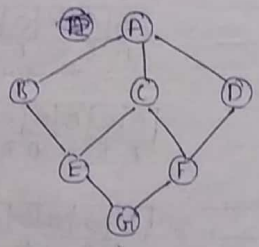
$O(V+E)$

$= O(V+E)$

if adj matrix then $= O(V^2)$

5) BFT is also known as Level Order Traversal.

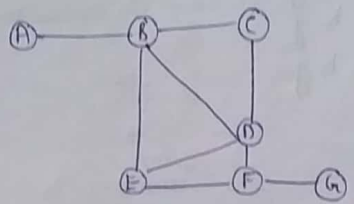
Ex1)



4 Correct BFTs

1) A B C D E F G
2) A D C B F E G
3) G E F B C D A
4) C F E A D G B

Ex2)

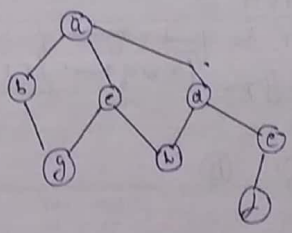


BFT – T/F

1) A B C D E F G T
2) D B C E F G A F
3) G F D E C B A T
4) C B D E A F G F T

Sol: 1) A | B | C D E | F | G
2) D | B C E F | A | G
3) G | F | D E | C | B | A
4) ~~C | B D | A | E F~~ C | B D | A E | F | G

13/8/18



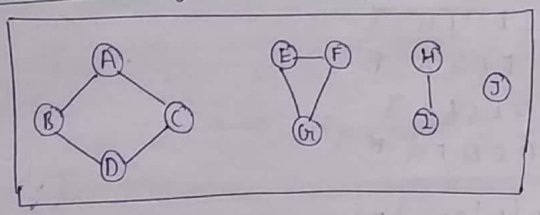
4 BFT sequences starting from F

~~f e d a b e g~~
~~f e d h e b g~~
i) f e d a h b e g
ii) f e d h a b e g
iii) ~~f e d a h e b g~~
iv) ~~f e d h a e b g~~

13/8/18

Sol: a) f e d a h b e g
b) f e d a h e b g
c) f e d h a e b g

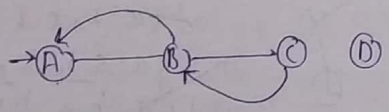
Applications of BFT (Grade) V.V. Imp (Refer G/G)



visited

0	0	0	0	0	0	0	0	0	0
A	B	C	D	E	F	G	H	I	J

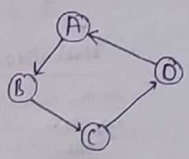
1. We can apply BFT to check graphs is connected or not. TC → O(V+E)
2. We can also used BFT to find number of Connected components of a graph (ie no of times BFT applied)



Using BFT we can check whether a state is reachable or not in an Automata, (directed Graph)

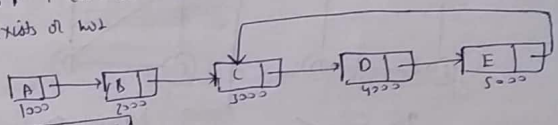
Connected Component : A maximal subgraph which is connected is known as connected component.

3. We can verify given graph contain cycle or not. TC = O(V+E)



NOTE:
All the above applications of BFT will take O(V+E) time

4. BFT can be used in Linked list to verify cycle exists or not

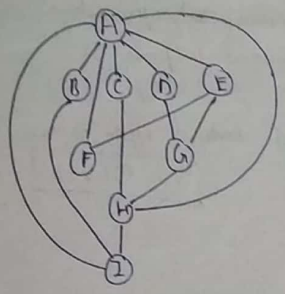


TC - O(V+E)

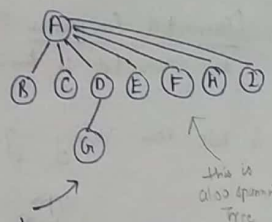
5. Using BFT we can find out shortest path from given source to every other vertex in given Unweighted graph in O(V+E) time.

6. BFT can also be used to generate Spanning Tree for a given unweighted graph in O(V+E) time.

NOTE: If edge weights are not given by default edge weight is considered 1.



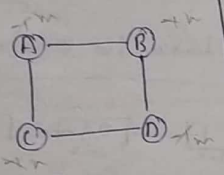
=)



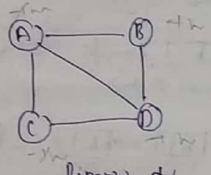
level order Traversal

All the nodes which are adj to A are present at next level.

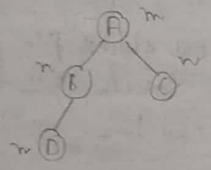
5) Using BFT we can verify given graph is bipartite (k=2) graph or not. in O(V+E) time.

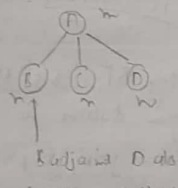


Bipartite ✓



Bipartite ✗



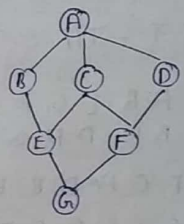


Bcoz in this D also but time D should be go to m but B & D are present in same set.

DFT (Depth First Traversal)

```
DFT(V)
{
  visited(v)=1
  Printf(v);
  for all w adj to v        → cover edges
  {
    if(w is not visited)
      DFT(w);               → cover vertices
  }
}
```

because of recursion, stack is used



visited

0	0	0	0	0	0	0
A	B	C	D	E	F	G

Stack is used by DFT

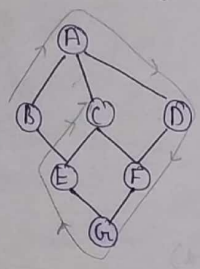
Stack size = O(V)

DFT(A)
w = B C D — A
DFT(B)
w = A E — A B
DFT(E)
w = B G C — A B E
DFT(G)
w = E F — A B E G
DFT(F)
w = G D — A B E G F
DFT(C)
w = F E F — A B E G F C
↓ pop

NOTE:

1) To implement DFT we are using Stack Data Structure.
2) TC = O(V+E) [adj list]
   TC = O(V²) [adj Matrix]
3) Space : i/p + extra
   ↓
   O(V+E) + Stack array
   V V
   = O(V+E)
4) TC = SC = O(V+E) (adj List) | O(V²) (adj Matrix)

Ex2 Consider the following Graph



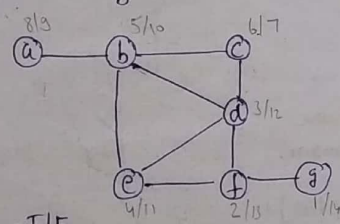
To get max stack size
find a path with
no pop in b/w

DFT - T/F

problem

① A B E G F D C
② B A D F G E C
③ F C A D B E G
④ G E C A D F B
⑤ C A B E D F G
⑥ F C A B E G D T

Ex3



DFT - T/F

a) f e d b c a g T
b) g f d e b c a T
c) b c d e g f a F
d) d f g e c b a F

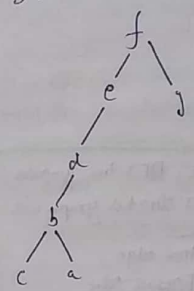
g f d e b c a
1/14 2/13 3/12 4/11 5/10 6/7 8/9

arrival time (push time)
finishing time (pop time)

In a graph after ⑨ ⑧ came what is relation b/w them?

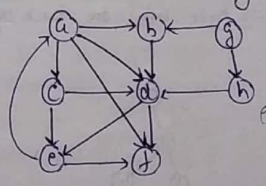
There is no relation b/w them. We can not say that they are adjacent, bcz ⑧ might have been pushed due to backtrack

DFT Tree (Spanning Tree)



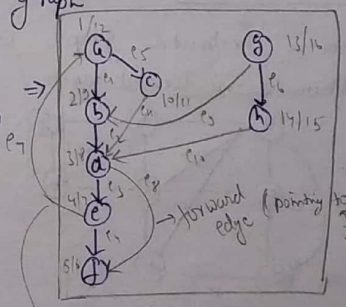
NOTE:
for a given graph many DFT tree possible

Ex3 Consider the following graph



edges present in DFT tree are called tree edges
($e_1, e_2, e_3, e_4, e_5, e_6$)

(Back edge (e_7)
Forward edge (e_8))

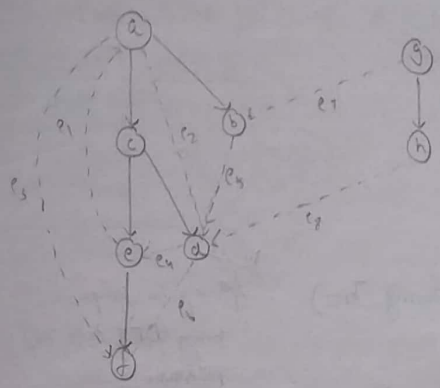


(e_9, e_{10} → cross edge)
(e_{11})

→ forward edge (pointing to Descendant not one)

DFT tree

edge pointing to ancestor (already visited vertex) back edge



Check for edges which are not present in DFT tree

forward edge - e_2, e_3

back edge - e_1

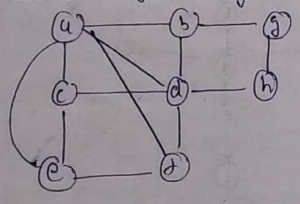
Cross edge - e_4, e_5, e_6, e_7, e_8

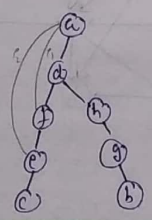
NOTE: DFT tree generated from a directed graph will contain
i) tree edge
ii) Forward edge
iii) back edge
iv) Cross edge

Undirected Graph (Imp)

Here in undirected graph we have only ~~tree~~ back edge

There is no forward edge





e_1, e_2 are back edges

Note: DFT tree generated from an undirected graph will contain
i) tree edge ii) back edge

Application of DFT (Imp) V.V.Imp

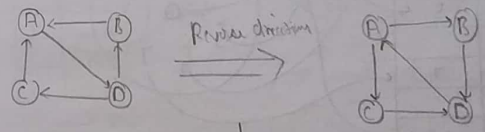
1) We can verify given graph is connected or not.

2) We can find out connected components in the given graph

3) We can verify given graph contain cycle or not. (if back edge present)

4) We can not find out Single Source Shortest Path in the given unweighted graph using DFT.

5) Using DFT we can verify given directed graph is strongly Connected or not.



DFT: ADCB

DFT: ACDB

apply DFT on A from A all reachable

apply DFT on A from A all reachable

TC => O(V+E)

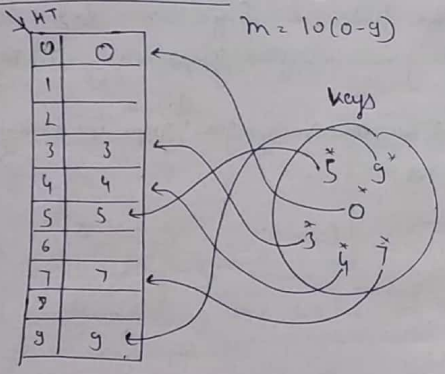
6) Using DFT we can find out number of articulation points in the given graph

14/8/18

Hashing

It is a searching technique where WC TC => O(1)

Direct Address Table (DAT)



1) According to DAT, key itself is the address without any calculation.

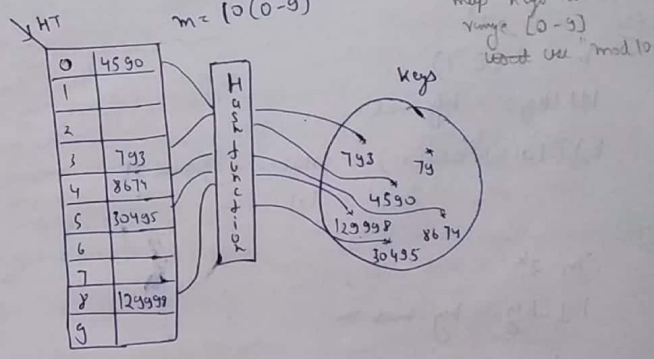
2) Searching Time is O(1) [every case]

3) The drawback with DAT is even though no. of keys are very less but one of the keys may be very large (2^{1000}) then there is a need of HT of size approx. 2^{1000}.

To eliminate above drawback we are going to the Hash function

Hash Function

How we have to map keys within range [0-9] we use 'mod 10'



Drawback

Collision - more than 1 key mapped to same Hash address is called collision.

Types of Hash functions

1. Division Modulo Method
2. Mid Square Method
3. Digit extraction Method
4. Folding Method

1. Division Modulo Method

Ex1

$m = 1000\ (0-999)$

$key = 789456123$

$H.f(k) = key \bmod m$

$= 789456123 \bmod 1000$

$= 123$

Ex2

$m = 8\ (0-7)$ (2^3)

$HF(key) = key \bmod m$

$hf(1010100101101) = 1010100101101 \bmod 2^3$

$= 101$

When binary number divided by 2^k we get remainder as Least significant k-bits ie (LSB k-bits)

Ex3

$m = 2^k$

$hf(key) = key \bmod m$

$key = 1010100101011100100$

$hf(key) = 1010100101011100100 \bmod 2^k$ (LSB k-bits)

$= 11001100$

NOTE:

1) dont take m value (exactly powers of 2, ie table size) bcz if $m = 2^k$ then $Hf(k)$ is always LSB k-bits.

2) Pick m value which is a prime no but it can not be close to powers of 2.

$k = 63\ (64)$ ✗ ← not suitable prime no bcz close to 2^6 ie 64.

$k = 511\ (512)$ ✗

$k = 713$ ✓

2) Mid Square Method (first square the key, then take middle)

Ex $m = 1000\ (0-999)$

$key = 84327$

$hf(key) = (84327)^2$ → 2 options for middle

$= 7111042929$

$= 104$

We take 3 digits from middle bcz it covers more address range, hence less collision

3) Digit Extraction Method (it is worse than division modulo) → (Truncation Method)

Ex $m = 1000\ (0-999)$

$key = 789123654$ (positions 1 2 3 4 5 6 7 8 9)

$Hf(key) = 934$

How digits are extracted will be mentioned in the question

→ It is worse than all the 2 above methods

4) Folding Method

1) Fold boundary → majority address 3 digit

$m = 1000\ (0-999)$

$key = 789\ 321\ 456$

Key = 789 3 21 456

h.f(key) = 789
456
1245
124
5
129 — 789321456

Result should be 3 digit but it came 4 digit, now again fold them

ii) Fold Shifting

m = 1000 (0-999) Here fold every 3 digit.

Key = 789 321 456

h.f(key) = 789
321
456
1566
156
6
162 — 789321456

Collision Resolution Techniques

1) Chaining (storing outside)
2) Open Addressing (storing inside)
   - linear probing
   - quadratic probing
   - double Hashing

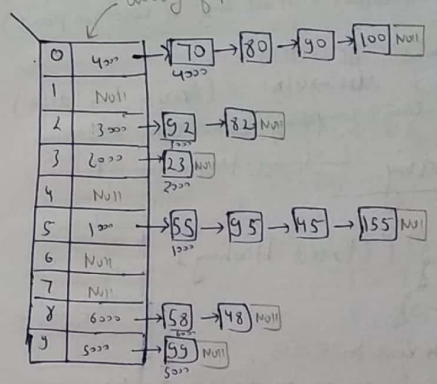
1) Chaining (Open Hashing)

Ex! m = 10

h.f = key mod m

keys = 70, 55, 92, 95, 80, 82, 45, 155, 58, 99, 48, 90, 100, 23

C.R.T = Chaining



length of largest chain = 4

length of smallest chain = 0

Average chain length = $\frac{4+2+1+4+2+1}{9}$

NOTE:

1) We are wasting lot of space outside in the form of linked list even though space available inside.
2) The largest chain possible with n keys is equal to n.

3. Searching time => BC = O(1)
WC = O(n)

4. The greatest advantage with the chaining is it can handle infinite number of collisions. (bcz of linked list)

5. Insertion time O(1) every case
* but with condition (insert only if node not present) → O(n)

6. Deletion time BC → O(1)
WC → O(n) (search then delete)

7. Deletion is easy comparing with Open Addressing

Open Addressing (~~Closed Hashing~~)

1. Linear Probing (Closed Hashing)

Ex: m = 10 (0-9)
Hf(k) = k mod m
keys = 75, 64, 33, 84, 93, 85, 69, 89, 43
C.R.T = Linear Probing

attempt no.

$$LP(k,i) = (hf(k) + i) \bmod m \quad \forall i, i \in \{0,1,2,\ldots m-1\}$$

0	89
1	43
2	
3	33
4	64
5	75
6	84
7	93
8	85
9	69

LP(75) = (hf(75) + i) / 5 + 0 = 5

LP(84) = 4 + 0 = 4
4 + 1 = 5
4 + 2 = 6
↑ 2 collision / 3 attempts

i = also tells no of collisions

Quadratic Probing

m = 10 (0-9)
hf(key) = k mod m
keys = 75, 64, 33, 84, 93, 85, 69, 89, 43
C.R.T = Quadratic Probing
Here assume $c_1 = c_2 = 1$

$$QP(key,i) = (hf(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$$

0	
1	69
2	
3	33
4	64
5	75
6	84
7	85
8	
9	93

QP(75,0) = 5 + 1·0 + 1·0² = 5
QP(64,0) = 4 + 1·0 + 1·0² = 4
QP(84,0) = 4 + 1·0 + 1·0² = 4
(84,1) = 4 + 1·1 + 1·1² = 6
QP(93,0) = 3 + 1·0 + 1·0² = 3
1 = 3 + 1·1 + 1·1² = 5
2 = 3 + 1·2 + 1·2² = 9
QP(85,0) = 5 + 1·0 + 1·0² = 5
5 + 1·1 + 1·1² = 7
QP(69,0) = 9 + 1·0 + 1·0² = 9
9 + 1·1 + 1·1² = 11 mod 10 = 1
QP(89,0) = 9 + 1·0 + 1·0² = 9
9 + 1·1 + 1·1² = 11 mod 10 = 1
9 + 1·2 + 1·2² = 15 mod 10 = 5
9 + 1·3 + 1·3² = 21 mod 10 = 1
9 + 1·4 + 1·4² = 29 mod 10 = 9
9 + 1·5 + 1·5² = 39 mod 10 = 9
9 + 1·6 + 1·6² = 51 mod 10 = 1
9 + 1·7 + 1·7² = 5
9 + 1·8 + 1·8² = 1

QP(43,0) = 3 + 1·0 + 1·0² = 3
1 = 3 + 1·1 + 1·1² = 5
2 = 3 + 1·2 + 1·2² = 9
3 = 3 + 1·3 + 1·3² = 5
4 = 3 + 1·4 + 1·4² = 3
5 = 3 + 1·5 + 1·5² = 3
6 = 3 + 1·6 + 1·6² = 5
7 = 3 + 1·7 + 1·7² = 9
43, & 89 will not be mapped anywhere.

9 + 1·9 + 1·9² = 9
How we tried m-1 probes

15/8/18

Double Hashing

Ex: m = 10 (0-9)

keys = 75, 64, 33, 84, 93, 85, 69, 89, 43 → want to be mapped to any slot

$hf_1(k) = k \bmod m$ $\quad hf_2(k) = 1 + (k \bmod (m-2))$

C.R.T = double Hashing

$DH(k,i) = (hf_1(k) + i\,hf_2(k)) \bmod m$

$\forall i \quad i = \{0,1,2,\ldots m-1\}$

0	~~93~~
1	93
2	
3	33
4	64
5	75
6	
7	85
8	
9	84

$DH(75) = hf_1(75) + 0 \cdot hf_2(75) = 5$

$5 + 0 \cdot = 5$

$DH(64) = hf_1(64) + 0 \cdot hf_2(64) = 4$

$DH(84) = hf_1(84) + 0 \cdot hf_2(84) = 4 + 0 = 4$

$hf_1(84) + 1 \cdot hf_2(84) = 4 + 1 \cdot 5 = 9$

$1 + 84 \bmod 8$

$DH(93) = 3 + 0 \cdot hf_2(93) = 3$

$3 + 1 \cdot 5 = 19 \bmod 10 = 9$

$3 + 2 \cdot 5 = 4$

$3 + 3 \cdot 5 =$

$3 + 3 \cdot 6 =$

$DH(93) = 3 + 0 \cdot hf_2(93) = 3$

$= 3 + 1 \cdot 6 = 9$

$= 3 + 2 \cdot 6 = 5$

$= 3 + 3 \cdot 6 = 1$

$DH(89) = 9 + 0 = 9$

$9 + 1 \cdot 8 = 17$

$9 + 2 \cdot 10 = 9$

$9 + 3 \cdot 10 = 9$

$9 + 9 \cdot 10 = 9$

not to be mapped

$DH(85) = 5 + 0 = 5$

$5 + 1 \cdot 6 = 1$

$5 + 2 \cdot 6 = 7$

$DH(43) = 3 + 0 = 3$

$3 + 1 \cdot 4 = 7$

$3 + 2 \cdot 4 = 1$

$3 + 3 \cdot 4 = 5$

$3 + 4 \cdot 4 = 9$

$3 + 5 \cdot 4 = 3$

$3 + 6 \cdot 4 = 7$

$3 + 7 \cdot 4 = 1$

$3 + 8 \cdot 4 = 5$

$3 + 9 \cdot 4 = 9$

WB Ex: m = 11 (0-10)

```
hf(k) =
{
  int x;
  x = (key + 5)(key + 5)
  x = x / 16
  x = x + key
  x = x % 11
  return x;
}
```

C.R.T = Linear probing

keys 42, 23, 1, 0, 15, 31 4, 7, 11, 3

Sol:-

$hf = \left(\left[\frac{(k+5)(k+5)}{16}\right] + k\right) \bmod 11$

$hf(k) = \left(\frac{(k+5)^2}{16} + k\right) \bmod 11$

0	
1	0
2	31
3	1
4	42
5	~~17~~
6	23
7	15
8	~~11~~
9	4
10	3

$hf(42) = 170 \bmod 11 = 4$

$hf(23) = 72 \bmod 11 = 6$

$hf(1) = 3$

$hf(0) = 1$

$hf(15) = 7$

$hf(31) = 2$

$hf(4) = 9$

$hf(7) = 5$

WB Ex

m = 10 (0-9)

$hf(k) = k \bmod m$

C.R.T = LP

After inserting 6 values into an empty hash table shown below

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

1) Find sequence to get above HT?

a) 46, 42, 34, 52, 23, 33
b) 34, 42, 23, 52, 33, 46
c) 46, 34, 42, 23, 52, 33
d) 42, 46, 33, 23, 34, 52

b) Imp

How many insertion sequences possible to get above HT?

a) 10 b) 20 c) 30 d) 40

46 34 42 23 52 33

52 − 3

$4! = 24 \times 2 + 6$

46 34 | 42 52 | 23 33 → last

2! 1 1

46 34 42 | 23 | 52 | 33

46 34 42 23 — — —

33 make its journey from 3 to 7
52 makes its journey from 2 to 5

34, 42 & 23 have problem with 52 bcz if 52 comes by then it will occupy their position.

2 cases are possible Here

How we will know, they directly map to their respective places bcz they can come in any order

I-Case

46 34 42 23 — 52 33

41 ways — they have problem with 52 — 2,3,4,5 — 1 way

46 34 42 23 — 6 4 2 3

$= 4!$
$= 24$

no problem with 52 so we can swap to 2nd group

II-Case

3! — 34 42 23

1! — 52 46 33

33 can not come by 52 come

46 52 36 — covering I Case

33 can not come by 46 else it will occupy its position

$= 6$

total = I + II = 24 + 6 = 30

(Linear Probing) (Limitations)

Ex: Linear Probing

m = 10 (0-9)

h(k) = k mod m

C.R.T = LP

keys = 52, 80, 93, 31, 60, 50

60 — 50

0	80
1	31
2	52
3	93
4	60
5	50
6	
7	
8	
9	

Primary Clustering

i) If 2 keys contain same starting hash address, then they both will follow same path unnecessarily (redundant work) in linear manner. bcz of this reason avg. searching time will increase. This problem is known as primary clustering.

ii) Linear Probing is having the drawback of primary clustering

iii) Search Time
BC → O(1)
WC → O(m)
AC → ?

ST = IT = DT

1st element 1 comparison, 2nd element 2 comparison

$$\frac{1+2+3+\dots+m}{m} = \frac{m(m+1)}{2m} = \frac{m+1}{2} = O(m)$$

iv) I.T = BC → O(1)
WC → O(m)
AC → O(m) [same as above]

I.T = Insertion Time

v) Deletion Time
BC → O(1)
WC → O(m)
AC → O(m)

vi) Deletion difficult (it will effect the other keys) but we can manage with special symbol $. If mark "$", re-hash again in (clear the HT & do re-hashing)

Quadratic Probing (Limitation)

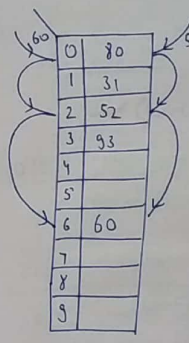
m = 10(0-9)

hf(k) = k mod m

QP = (hf + c1 i + c2 i²) mod m
assume c1 = c2 = 1

QP = Quadratic Probing

keys = 52, 80, 93, 31, 60, 50



QP(60,0) = hf(60) + 1·0 + 1·0²
= 0 + 0 + 0 = 0
0 + 1·1 + 1·1² = 2
0 + 1·2 + 1·2² = 6

QP(50,0) = 0 + 1·0 + 1·0² = 0
0 + 1·1 + 1·1² = 2
0 + 1·2 + 1·2² = 6
0 + 1·3 + 1·3² = 2
0 + 1·4 + 1·4² = 0
0 + 1·5 + 1·5² = 0
0 + 1·6 + 1·6² = 2
0 + 1·7 + 1·7² = 6
0 + 1·8 + 1·8² = 2
0 + 1·9 + 1·9² = 0

does same work many times with same entries

Hence here also we have redundancy

Secondary Clustering

i) If 2 keys contain same starting hash address they both will follow same path unnecessarily (in quadratic manner). bcz of this avg searching time will increase. This is called secondary clustering.

ii) Quadratic Probing is having drawback of Secondary Clustering.

3) Searching time = Insertion time = Deletion Time

BC → O(1) WC → O(m) AC → O(m) i.e. $\frac{m+1}{100}$

decreases by constant factor compared to Linear probing.

Double Hashing (

m = 10 (0-9)

$hf_1(k) = k \bmod m$ $hf_2(k) = 1 + (k \bmod (m-2))$ mod

C.R.T = Double Hashing

keys = 52, 80, 93, 31, 60, 50

0	80
1	31
2	52
3	93
4	
5	60
6	50
7	
8	
9	

DH(60) = 0 + 0·(5) = 0
0 + 1·5 = 5

DH(50) = 0 + 0·3 = 0
= 0 + 1·3 = 3
= 0 + 2·3 = 6

DH(70) = 0 + 0·7 = 0
0 + 1·7 = 7
0

i) If 2 keys contain same starting Hash address & they both will follow different paths. bc (most probably) bez of 2nd Hash fun. bez of this reason Any searching time is O(1).

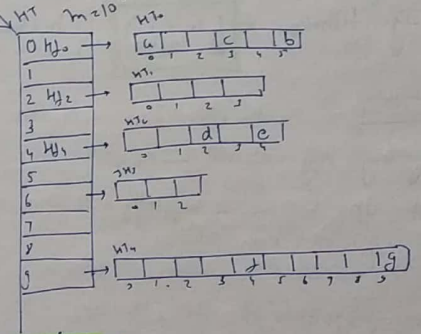
ii) Searching Time = Insertion Time = Deletion Time

BC → O(1) WC → O(m) AC → O(1)

iii) In double hashing secondary clustring problem is eliminated 99.1. (1.1. left out)

NOTE: Using perfect Hashing we will get WC searching time O(1).

Perfect Hashing



Load factor

M - Slots N - keys

m slots = N - keys

1 slots = $\frac{N}{m}$ keys/slot

↑

Avg

(or)

Load factor → no of keys per slot

~~$\alpha \leq 1$~~ $0 \leq \alpha \leq 1$ → (Open Addressing)

~~$0 \leq \alpha \leq \infty$~~ $0 \leq \alpha < \infty$ → (Chaining)

NOTE:

~~The instructor~~

1) The expected no of probes in an Unsuccessful search of Open addressing technique = $\frac{1}{1-\alpha}$

2) The expected no of probes in an successful search of open addressing technique = $\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$ ln = base e

Universal Hash fun

Independent of any key it will work.

irrespective of the i/p, whether it is integer or character or float. it will give the hash address

My NOTES

Important Questions

Gate (2010) [WB = Ch-2 Q. 22]

Q The weight of a sequence $a_0 a_1 a_2 \ldots a_{n-1}$ of real numbers is defined as $a_0 + \frac{a_1}{2} + \frac{a_2}{2^2} + \cdots + \frac{a_{n-1}}{2^{n-1}}$.

A subsequence of a sequence is obtained by deleting some elements from the sequence keeping the order of the remaining elements the same. Let

X = maximum possible weight of a subsequence of $a_0 a_1 a_2 \ldots a_{n-1}$

Y = maximum possible weight of a subsequence of $a_1 a_2 a_3 \ldots a_{n-1}$

Then X = ?

Sol: Let S = 30, 20, 40, 32 $a_0 = 30$

X = a_0 + maximum possible weight of subsequence of $a_1 a_2 \ldots a_{n-1}$

$X = a_0 + Y/2$

$X = 30 + \frac{20}{2} + \frac{40}{2^2} + \frac{32}{2^3} = 30 + (24) = 30 + \frac{Y}{2}$

$Y = 20 + \frac{40}{2} + \frac{32}{2^2} = 48$

If we add a_0 we have to halt the remaining sequence.

Here we have two options

X < $a_0 + Y_2$ (we can take a_0)

Y (we don't take a_0)

$\because a_0 \in R$ R = Real no.

a_0 can be +ve or -ve.

Ans: $X = MAX(a_0 + Y_2, Y)$

Ques: CLRS (P-18)

Q Given 2-array of numbers $a_1, a_2 \ldots a_n$ and $b_1, b_2 \ldots b_n$ where each number is either 0 or 1.
The fastest algorithm to find the largest span (i,j) such that $a_i + a_{i+1} + \ldots + a_j = b_i + b_{i+1} + \ldots + b_j$ (or) that there is no such span? $T(\rightarrow O(n))$

Sol: Concept: longest span with same sum in 2 binary arrays.

→ maximum possible sum is n.
→ Diff b/w prefix sum can vary -n to n
so we create an auxillary array of size 2n+1 which will cover all the possible diff b/w prefix sum in the arrays.

Starting from 0 if at any point diff b/w prefix sum is same. then the required span (i,j) [or] subarray with same sum is from 0 to current index, at which sum is same

n=7

arrA → 0 1 0 1 1 1 1 (0 1 2 3 4 5 6)

arrB → 1 1 1 1 1 0 1

Auxillary array to cover all possible diff of size 2n+1

-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 (0 1 2 3 4 5 6 7 8 9 10 11 12)

```
maxlen = 0; // keep track of longest span
presum1 = presum2 = 0;
curr_diff = presum1 - presum2  // -n to n
diff_index = n + curr_diff  // index can't be -ve
len = 0; // keep track of current length

for i = 0 to n-1
    presum1 += arrA[i]
    presum2 += arrB[j]
    curr_diff = presum1 - presum2
    diff_index = n + curr_diff

    if (curr_diff == 0)
        maxlen = i + 1;
    else if (diff[diff_index] == -1)
        diff[diff_index] = i
    else
        int len = i - diff[diff_index]
        if (len > maxlen)
            maxlen = len;
```

Gate 2008 [WB = [n-1, QS]

Q Find minimum no of comparisons to find majority element in sorted array of n elements

Sol: majority element → element which occurs more than $n/2$ times.

Concept: if array is sorted, then majority element will definitely be present at middle index position i and additionally to that it will also be present at either of the neighbouring places namely i-1 and i+1.

[1 2 3 3 4 4 4 6] apply BS to count the occurrences of 3

Using binary search in O(logn) time we can find the occurrences of majority element.

Minimum no of comparisons : O(logn)

Q Find: majority element in an array of n elements if exists otherwise print "No majority element". TC → O(n)

Sol: Concept: Moore's Voting Algorithm.

```
int find Candidate (a, n)
{ majority_index = 0 , count = 1
  for i = 1 to n-1
  { if ( a[majority_index] == a[i])
        count ++;
    else
        count --;
    if ( count == 0)
      { majority_index = i
        count = 1;
      }
  }
  // after for loop
  return a[majority_index];
}

bool isMajority ( a, n, candidate)
{ count = 0;
  for i = 0 to n-1
  { if (a[i] == candidate)
        count ++;
  } else
  if ( count > n/2)
      return 1
  else
      return 0
}
```

Q1.45 P-386

Sol: I> Apply search in BBST for L

To search L it will take O(logn)

II> From L apply inorder traversal (& while traversing add all the nodes) upto we reach H. Stop here → It will take O(m) time bcz there are m elements b/w L — H.

∴ total TC = O(logn) + O(m)

∴ $O(n^a \log^b n + m^c \log^d n)$ → a=0 b=1 c=1 d=0

a+1ob +1ooc +1oood = 10 +100 = 110

Q 3.44 · P-411

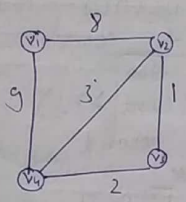
Sol: $G_1 = (V, E)$ CUEWG

Connected Undirected Weighted Graph.

Weights are +ve & distinct

I. True

II. False. Ex:



Shortest path b/w v_2 & v_4

i) $v_2 — v_4$

ii) $v_2 — v_3 — v_4$

Q 3.43 P-410

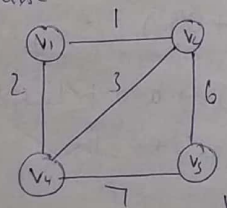
Sol: $G = (V, E)$ USG

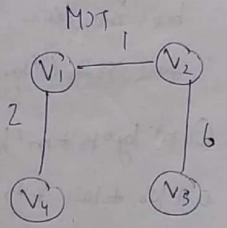
Undirected Simple graph

distinct weights

e → particular edge

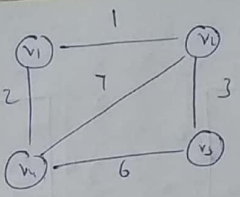
I. False

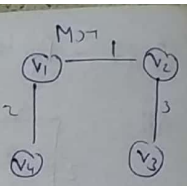




We can not take $(v_4 v_2)$ bcz cycle will be formed.

II: True





Q 3.41 P-410

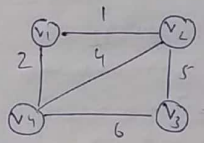
Sol: $G = (V, E)$ WCDG

Weighted Connected Directed Graph

distinct +ve edge weights

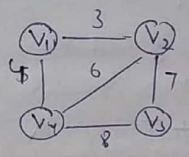
P: True Q: False Ex

increase by 2



Shortest path b/w v_2 & v_4

$v_2 \to v_1 \to v_4$

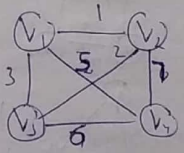


new shortest path

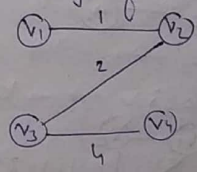
$v_2 \to v_4$

3.42 P-410

Sol:



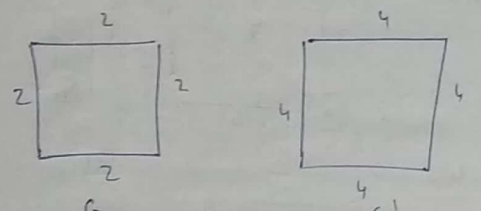
max weight of MST



Max weight = 1+2+4 = 7

3.35 - P- 409

Sol.

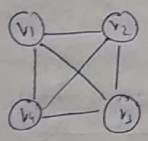


$t = 6$ $t' = 12$

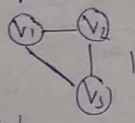
$T \neq T'$ & $t' \neq t^2$

3.36 P- 408

Sol: $n=4$ $|E|=6$



a) True $k=3$

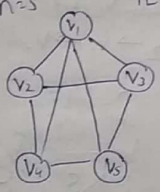


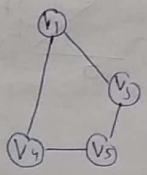
$|E| = 3$

$|E| \leq 2k-2$

$|E| \leq 4$

$n=5$ $|E|=8$





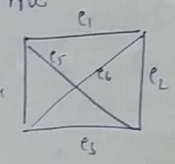
$|E| = 4$

$|E| \leq 2k-2$

$|E| \leq 6$

Induced subgraph means all the vertices and their corresponding edges

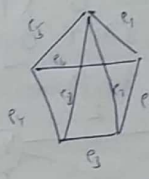
b) True



min cut ≥ 2

edge cut to disconnect graph

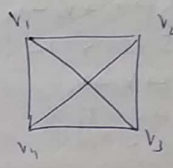
$\{e_1, e_2, e_6\}$

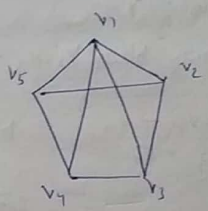


min cut ≥ 2

$\{e_1, e_2, e_4\}$

c) True





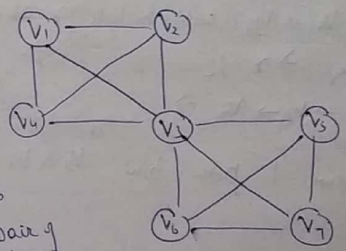
$(V_1, V_2) \rightarrow V_1 \rightarrow V_4 \rightarrow V_2$

$V_1 \rightarrow V_3 \rightarrow V_2$

$(V_1, V_3) \rightarrow V_1 \rightarrow V_2 \rightarrow V_3$

$V_1 \rightarrow V_5 \rightarrow V_4 \rightarrow V_3$

d) False

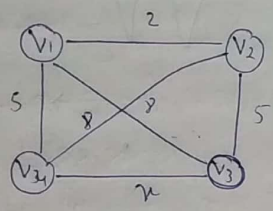


no vertex disjoint path b/w every pair of vertex.

Q 4.12 P-421 WUG

Sol:

$$W = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 0 & 2 & 8 & 5 \\ 2 & 0 & 5 & 8 \\ 8 & 5 & 0 & x \\ 5 & 8 & x & 0 \end{bmatrix} \end{array}$$



max value of x

~~Shortest~~ path b/w V_3, V_4 → i) $V_3 \to V_2 \to V_1 \to V_4$
$5 + 2 + 5 = 12$

iii) $V_3 \to V_1 \to V_4$
$8 + 5 = 13$

ii) $V_3 \to V_2 \to V_4$
$5 + 8 = 13$

~~iii~~ iv

Shortest path b/w $(V_3, V_4) \Rightarrow 12$

max value of x can be 12., now shortest path b/w V_4 & V_3 will include x also

i) $V_3 \to V_4 \Rightarrow x = 12$

$V_3 \to V_2 \to V_1 \to V_4 \Rightarrow 12$

Hence at least 1 path b/w V_3 & V_4 include x.

4.14 P-421

Sol: In parenthesization of $F_1 F_2 F_3 F_4 F_5$ we have to find explicit computed pairs

Here check with options

~~$F_1 F_2$~~ $(F_1)_{2\times25}$ $(F_2)_{25\times3}$ $(F_3)_{3\times16}$ $(F_4)_{16\times1}$ $(F_5)_{1\times1000}$

$F_1 F_2 \to 2\times25\times3 = 150$
$F_3 F_4 \to 3\times16\times1 = 48$
$F_2 F_3 \to 25\times3\times16 = 1200$
$F_4 F_5 \to 16\times1\times1000 = 16000$

Hence Options (b) & (d) are eliminated.

Check (a) & (c)

(a) $F_1 F_2 \; F_3 F_4$

$(F_1 F_2)(F_3 F_4) F_5$

$150 + 48 + 2000 = 2198$

(b) $F_1 F_2 (F_3 F_4) F_5 \Rightarrow \big((F_1 (F_2 (F_3 F_4))) F_5\big)$

$48 + 75 + 50 + 2000 = 2173$

Hence (b) is answer.

6.18 P-428

Sol: Find all leaders in an array

Scan from right and keep track of maximum till now. When max changes its value print it.

Ex: arr[] = {16, 17, 4, 3, 5, 2}

~~for i=n-2~~

max_from_right = a[n-1]

// last element will always be leader

print (max_from_right)

```
for (i = n-2 ; i > 0 ; i--)
{
    if (max_from_right < a[i])
    {
        max_from_right = a[i];
        print (max_from_right);
    }
}
```

TC → O(n) SC → O(n)

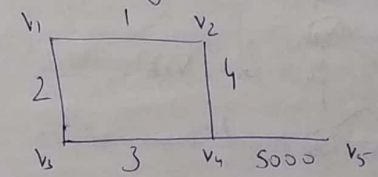
Q26 Chp-2 P-19 (WB)

Sol: Consider the following statements

I) Let T be a minimum spanning tree in a graph G. Then for any two vertices v and v the path v to v in T is the shortest path v to v in the graph G.

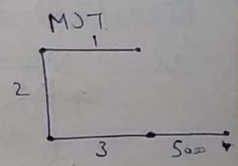
II) Suppose that average edge weight for a graph G is Aavg. Then the minimum spanning tree of G will have weight at most (n-1) Aavg. Where n is number of vertices in graph G.

Sol:-



G

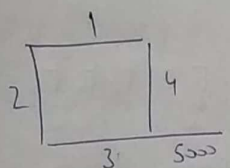
Shortest path b/w V_2 & V_4 = 4



T

Shortest path b/w V_2 & V_4
1+2+3 = 6

Hence I) False



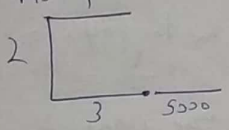
G

$$\text{Avg} = \frac{1+2+3+4+5000}{5}$$

$$= \frac{5010}{5}$$

$$= 1002$$

MST



max weight in T = 5000

$5000 \leq (n-1)\text{Avg}$

$5000 \leq (5-1)1002$

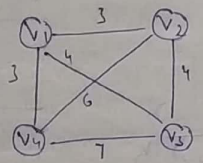
$5000 \leq 4 \times 1002$

$5000 \nleq 4008$

Hence II > False.

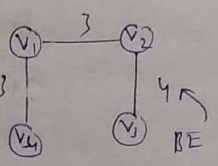
Q20 (TS) - 2019

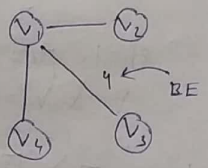
Sol.



$G = (V, E)$

MST

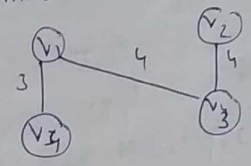




BE = Bottle neck edge

BT → bottle neck tree

min bottle neck tree



it is not ST.

ST -> spanning tree

Q19

Sol. given degree seq. of n vertices

TC to determine simple graph possible (or) not

we use Havalle - Hakimi algorithm

TC → $O(n^2 \log n)$

Q7

Sol: Multiplying 2 large no. in binary format

we use Karatsuba algorithm (~~divide~~ & conquer)

based on divide & conquer)

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

$$T(n) = O(n^{1.59})$$

Imp

Q find the median of 2 sorted array of same size.

i/p : arr1 { 1, 12, 15, 26, 38 }

arr2 { 2, 13, 17, 30, 45 }

after merging 2 arrays

{ 1, 2, 12, 15, 15, 17, 26, 30, 38, 45 }

median = $\frac{15+17}{2}$ = 16

Size of ~~sorted~~ array after merging = n+n = 2n

Since size is even median = $\frac{mid1 + mid2}{2}$

Algorithm TC → O(log n)

1. Calculate medians m1 and m2 of the i/p arrays arr1[] and arr2[] respectively
2. If m1 and m2 both are equal, then ~~medians~~ is we are done. return m1 (or) m2.
3. If $m_1 > m_2$, then median is present in one of the below 2 subarrays
   a) From arr1[0] to m1 (ie arr1[0] to arr1 + $n_1/2 - 1$)
   b) From m2 to last element of arr2
      ie (arr2 + $n_1/2 - 1$ to arr2[n-1])
4. If $m_2 > m_1$, then median is present in one of the below two sub arrays.
   a) From m_1 to last element of arr1
      (ie arr + $n_2/2 - 1$ to arr[n-1])
   b) From first element of arr2 to m_2
      (ie arr2[0] to arr2 + $n_2/2 - 1$)
5. Repeat above process until size of ~~medi~~ both subarrays becomes 2.
6. If size of 2 ~~sub~~arrays is 2 then we can calculate median as

$$\text{median} = \frac{\max(arr1[0], arr2[0]) + \min(arr1[1], arr2[0])}{2}$$

Ex: arr1[] = { 1, 12, 15, 26, 38 }

arr2[] = { 2, 15, 17, 30, 45 }

$m_1 = 15$ $m_2 = 17$ ie $m_2 > m_1$

[15, 26, 38] and [2, 15, 17]

$m_1 = 26$ $m_2 = 15$

$m_1 > m_2$

[15, 26] [15, 17] ← size is 2

$$\text{median} = \frac{\max(15,15) + \min(26,17)}{2} = \frac{15+17}{2} = 16$$

⇒ # of back edges in undirected graph = e-n+1

⇒ BFS / DFS can be used to find no of biconnected components of undirected graph $O(n^2)$

⇒ If graph contain "k" cut vertices, then graph has atleast k+1 biconnected components

⇒ Biconnected graph : in which every pair of vertex must be connected by two distinct paths.

Selection procedure

Finding kth smallest/largest element in unsorted array

I) Quick sort partition procedure
→ $O(n^2)$ ← WC
$O(n)$ ← AC

II) Using Min Heap
$O(n) + k(\log n) \Rightarrow O(k \log n)$
apply k times extract key operation

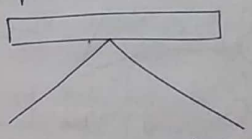
III) Using Max Heap
$O(k) + (n-k)(\log k)$

for remaining (n-1) elements compare with root, if less than make it root and apply Heapify. opn. If bigger than ignore it.

Maximum Sub array Sum

find sub array with maximum sum in a given 1-D array

→ Divide & Conquer



divide given arry into 2 half.

return max { max subarray sum in left half
max subarray sum in right half
max subarray sum such that subarray cross mid point }

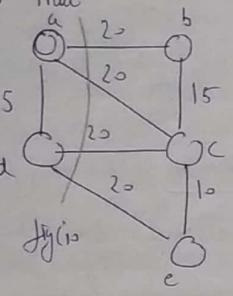
$T(n) \rightarrow O(n \log n)$

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

Using Kadane's algorithm $T(n) \rightarrow \underline{O(n)}$

Q 3.11 P- 406 (Gate 2005)

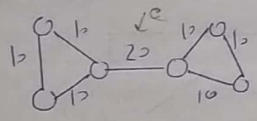
Sol: a) True



Cut set
{(a,b) (a,c) (d,c), (d,e)}

if MST contains edge with max weight than there exists a cut with all edges with weight equal to heavy weight.

b) False (Not always true)



Worst case no other option but to include 2.

c) $f_3(i)$ d) not always true option b)

Test GrM-2

Q
```
for (int i1=1 ; i1≤n ; i1++)
    for (i2 = 1 ; i2 ≤ i1 ; i2++)
        for (i3 = 1 ; i3 ≤ i2 ; i3++)
            .
            .
            .
                for (im = 1 ; im ≤ im-1 ; im++
                    : count ++
```

Initially count = 0

Value returned by count = $^{n-1+m}C_m$

for $n=8$ $m=6$

count will return $^{8-1+6}C_6 \Rightarrow {}^{13}C_6 = 1716$

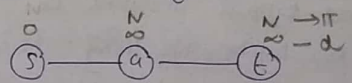
FST(A-6) 2019

Q given an array of n natural no.
A[n] $n \in N$
A contains exactly n no. of inversions
if all the no's are made -ve
TC to find inversion pairs ?

Sol: Modified array will contain $\binom{n}{2} - n$ inversions
TC → ~~O(n)~~ O(1) bcz we need to compute $\binom{n}{2} - n$ which takes constant time.

→ The running time of matrix sort is effectively independent of whether the i/p is already sorted → True

→ Changing the RELAX function to update is $d[v] \geq d[u] + w(u,v)$ (instead of strictly greater than) may produce shortest path, but will not effect the correctness of Bellmann ford algorithm → False



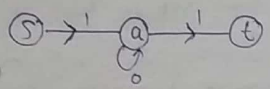
relax (s,a) $d[a] \geq d[s] + w(s,a)$

$\geq 0+1$

≥ 1

$d[a] = 1$ $\pi[a] = s$

the parent pointer may not lead back to the source node if a zero length cycle exists.



Relax (a,a)

(s,a) $d[a] \geq d[a] + w(a,a)$

$1 \geq 1+0$

$1 \geq 1$ $\quad d[a] = 1 \quad \pi[a] = a$

MT 7(A)

Given array A[n] of natural no. Tc to find max sum sub array

Sol: O(1) bcz all elements are natural number Hence array itself will be subarray with maximum sum.

Bit Algorithm

Important points

→ Java does not support unsigned no.

→ Signed no's are represented using 2's comp.

→ Ex: -2 rep: Suppose 4 bits are used to rep no.

$2^4 - 2 = 14 = 1110$

-5 $\quad 2^4 - 5 = 11 = 1011$

Bit wise operator

&, !, ~, ^, >>, <<

↓ bit wise complement

↓ bit wise XOR

>> → right shift $\equiv \left[\frac{x}{2^k}\right] \equiv x >> k$

<< - left shift $\equiv x * 2^k \equiv x << k$

Q Check whether k^{th} bit from right is set or not.

Sol: i/p n = 5 101

k = 1 $\quad$ k = 2

o/p - True $\quad$ o/p = False

Concept: generate a no in which only k^{th} bit from right is set. and perform bit wise & b/w generated no. and given no. generate second no.

```
if (num & (1 << (k-1)) != 0)
    print True
else print False
```

Q Check for power of 2.

Sol: i/p = 4 $\quad$ i/p = 10 $\quad$ i/p = 0

o/p - True $\quad$ o/p → False $\quad$ o/p → False

Method - 1 Keep dividing no by 2, if at any step (num%2 != 0) given num is not power of 2

Method - 2 !(num & (num-1))

↳ return 0 if power of 2
return ≠0 if not power of 2

```
if (!(x&(x-1)))        if (x && !(x&(x-1)))
    print True             print True
else                   else
    print False            print False
```

Q Counting no of set bits in a given integer.

Sol. Standard algorithm

Brian Brian Kernighan algorithm

TC → O(no of set bits)

Unset right most set bit

```
x = 26   11010
int res = 0
While (x>0)
{
   x = x & (x-1)
   res++
}
return res;
```

11010
11000
10000
00000

Q given an array arr[1...n] of n integers in which all elements appear even no of times exp except for one element which appear odd times. find that no element.

elements which occurs even no of times will cancle out each other

arr[] = {3,3,3,4,5,4,5}

property of XOR

$x \oplus 0 = x$

$x \oplus 1 = \bar{x}$

$x \oplus y = x\bar{y} + \bar{x}y$

$(x \oplus y) \oplus z = x \oplus (y \oplus z)$

```
res = 0
for (int i=0; i<n; i++)
    res = res ^ arr[i]
```

Q Generating power set of a set?

Sol. Str = "ABC" <- given set

Pow(str) = {"", "A", "B", "C", "AB", "AC", "BC", "ABC"}

We can find the size of set. say n

if we generate binary no's from 0 to $2^n - 1$

Every binary no. corresponds to a ~~set~~ subset of given set n=3 0 to 7

000 → ϕ
001 → A
010 → B
011 → AB
100 → C
101 → AC
110 → BC
111 → ABC

given size of set n

Calculate (2^n)

```
int count = (1<<n)          // generate no from 0 -(2^n - 1)
for(int i=0; i<count; i++)
    for(int j=0; j<n; j++)  // loop through every bit in no. only
        if((i & (1<<j)) != 0)   // generate no where k'th bit is set
            print(str[j])   // checking whether k'th element from right in set is present in the subset or not.
```

Exercise

Q given a number in binary representation find largest span of 1's.

Done

Q given an array of n integers, in which all elements appear even no of times except for 2 elements which appear odd no of times. find these 2 elements.

Mathematics

finding no of digits in a number

```
while(n!=0)
{
    count++;
    n = n/10;
}
```

```
countDigit(int n)
if(n==0)
    return
else
    1 + countDigit(n/10)
```

use logarithmic

count = $\lfloor \log_{10}(n) + 1 \rfloor$

Recursion

We solve a problem assuming s/n's to smaller subproblems are available.

Q given a number n, print nos from n to 1 without using loop.

I)
```
printNum(int n)
{
    if(n==0)
        return
    else
        print(n)
        printNum(n-1)
}
```

failed recursion

```
print 1 to n
{
    printNum(int n)
    if(n==0)
        return
    else
        printNum(n-1)
        print(n)
}
```

Advantage of failed recursion

Compiler optimization Recursive —Convert→ Non-recursive (space saved)

Q given a rope of length n and three values a,b,c we need to make maximum pieces such that every piece has length in set {a,b,c}. & no piece is wasted.

i/p n=5 a=1, b=2, c=3

o/p 5

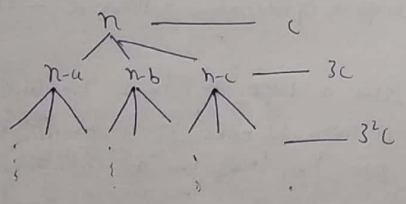
i/p n=5 a=2, b=4, c=6

o/p : -1

```
int getMax(int n, int a, int b, int c)
{
    if (n < 0)
        return -1
    if (n == 0)
        return 0

    int A = getMax(n-a, a, b, c)
    int B = getMax(n-b, a, b, c)
    int C = getMax(n-c, a, b, c)
int res return = max(A, B, C)
    if (res == -1)
        return -1
    else
        return res+1
}
```

TC $T(n) = T(n-a) + T(n-b) + T(n-c) + C$

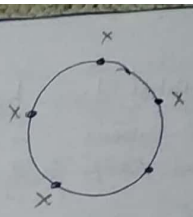


$TC = C + 3C + 3^2C + \cdots 3^nC$

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

$= O(3^n)$

Q Josephus Problem



o/p n = 5 k = 2

o/p → 3

n people standing in a circle, in every iteration kth person is killed. We need to find luck place

$$JP(n,k) = (JP(n-1,k) + k - 1) \% n + 1$$

$JP(1,k) = 1$

position of the person who will hold the gun in next iterations with n-1 people

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

$O(n)$

Q Printing all the subsets of a set

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

$O(2^n)$

Set = "ABC"

Subsets = {"", "A", "B", "C", "AB", "AC", "BC", "ABC"}

```
void printSubset(string str, int index = 0, string curr_subset = "")
{
    int n = str.length()
    if (index == n)
        print(curr_subset)
        return
    printSubset(str, index+1, curr_subset)
    printSubset(str, index+1, curr_subset + str[index])
}
```

Q Exercise: given an integer set and a number. print all subsets whose sum is equal to given number

i/p arr[] = {2, 3, 8, 5, 4, 9}, n = 7

o/p: {2,5} {3,4}

Imp

Q Tightest lower bound on the number of comparisons ; in the worst case, of comparison based sorting is of the order of

$O(n \log n)$

Sol:- Given a list of n distinct numbers $n!$ permutations are possible, out of which only one is sorted.

Sorting algorithm must gain enough information from comparisons to identify the correct permutation.

If algorithm always completes after $f(n)$ steps, then it can not analyze (distinguish) more than $2^{f(n)}$ cases (bcz when comparing, each comparison has only 2 possible outcomes) → either swap / no swap

Hence $2^{f(n)} \geq n!$

$\log 2^{f(n)} \geq \log n!$

$f(n) \geq \log n! \rightarrow f(n) = \Omega(n \log n)$

Q You have to sort 1GB of data with only 100MB of available main memory. Which sorting technique will be most appropriate?

Sol:- The data can be sorted using any external sorting technique which uses merging technique

1. Divide 1GB data into chunks of 100MB.
2. Sort each group and write them to disk
3. load 10 items from each group into main memory. Output smallest item from main memory to disk.
4. Load next item from the group whose item was chosen
5. loop 4 step until all items are not outputted

Sort nearly sorted array

Given an array of n elements, where each element is at most k away from its correct position in sorted array.. Devise an algorithm that sort in $O(n \log k)$ times.

Insertion Sort

i/p: arr[] = {6, 5, 3, 2, 8, 10, 9}
k = 3

o/p: arr[] = {2, 3, 6, 5, 8, 9, 10}

```
for(i=1; i<n; i++)
    key = arr[i] ; j = i-1
    while( j>=0 && arr[j] > key)
        arr[j+1] = arr[j]
        j = j-1
    arr[j+1] = key
```

While loop will run at most k times for each element. $T(\rightarrow O(nk)$

Efficient method → Heap data structure

1) Create min Heap of size k+1 with first k+1 elements. → O(k) time

2) One by one remove min element from heap, put it in result array. And add a new element to heap from remaining elements.

delete & insert

insertion

$T(\rightarrow O(k) + \cancel{n \log k} + (n-k)\log k$

$T(\rightarrow O(n \log k)$

Segregating numbers

Q given arr[1..n] consists of only +ve & -ve no's. Segregate no's having same sign altogether.

Sol:- We can use partition algo of quick sort.

$T(\rightarrow O(n)$

# of comparisons = n-1