22/08/2017 :

Algorithms :

1. Asymptotic Analysis
   Recursive / Non recursive Algo
2. Asymptotic Notations
3. Recurrence Relations [Methods to solve]
4. Algorithm Design Techniques
   - Divide & conquer
   - Greedy
   - Dynamic
5. Tree and graph representation & traversal
   Priority Queues
6. Sorting algo.

Books → Cormen
→ Sartaj Sahani
→ Mark Allen weiss

ravi.82peddapu@gmail.com
@facebook.com
08074172708

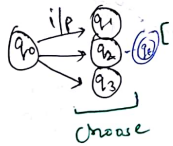
Introduction :

→ Step by step representation of computer program.

Criterias :

1. Finiteness - Algo must terminate in finite amount of time.
2. Definiteness - Each step of algorithm must have unique solution.
   [Deterministic Algorithm], $q_0 \xrightarrow{i/p} q_1$ [possible to implement in comp.]

   Non Deterministic Algo: each step of algo consists of finite no. of solutions and algo should choose correct solution in first attempt.



[Not possible to implement]

3. Effectiveness - Each step of algo should be very basic.

Steps to design algorithm :

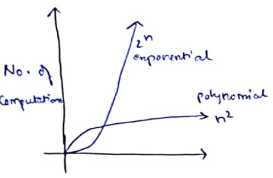
1. Devise Algo : Design algo for given problem using best design technique.
2. Validation of Algo : Test algo correctness.
3. Analysis of algo : Estimation of CPU execution time / main mem space reqd. to complete execution of algo.

Decidable Problem [Tractable] : Problem which can be solved in polynomial time using deterministic algo. [Efficient algo.].
Efficient algo.

No. of computations required → n^2, $n\log n$, n^3, n^k [polynomial time] for n i/p.

Undecidable Problem [Intractable] : Problems which take exponential time to solve using deterministic algo.

n i/p size : # of computations → 2^n, 3^n, $n!$, n^n [Exponential time]



n	n^2 polynomial	2^n exponential
10	100	1024
11	121	2048
12	144	4096
⋮	⋮	⋮
49	$(49)^2$	2^{49} (Halt)
50	$(50)^2$	2^{50} (May not halt)

Algorithm Analysis [Estimation of Time and space complexity]

ⓐ **Time complexity**: Estimation of CPU time [CPU computations] required to terminate algo.

TC of algo = Sum of frequency count of each instruction of algo.

```
Algo sum(a,n)
{ sum = 0 ------------> 1
  for(i=1; i<=n; i++) ---> 2(n+1)
  { sum = sum + a[i]; } ---> n
  return sum; ---------> 1
}
```

$T(n) = 3n + 4$

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

Loop		
i=1	i≤n	i++
	1≤n	1
	2≤n	2
	3≤n	3
	⋮	⋮
	n≤n	n
—	n+1≤n	—
1	n+1	n

ⓑ **Space Complexity**: Estimation of main memory space required to execute algo.

```
sum(a,n) =>  i   : 1
             n   : 1
             sum : 1
             a[] : n
                  ---> n+2 ---> S(n) = space Complexity
```

$S(n) = \Theta(n)$

TC | SC of loops :

① `for (i=1; i<=n; i++)` $2(n+1)$
`printf("Made Easy");` $\frac{n}{3n+2} = \Theta(n)$

TC : $\Theta(n)$
SC : 2 (constant) : $\Theta(1)$

② `for (i=n; i>=1; i--)`
`printf("Made Easy");` — n Times

T.C = $\Theta(n)$
S.C = $\Theta(1)$

③ `for (i=1; i<=n; i=i+5)`
`printf("Made Easy");` → $\lfloor\frac{n-1}{5}\rfloor + 1$ times

$i = \underbrace{1, 1+5, 1+2\times5, 1+3\times5, \cdots, 1+k\times5}_{k+1 \text{ prints}}$

$1 + k\times5 \le n$
$k \le \frac{n-1}{5} \Rightarrow k = \lfloor\frac{n-1}{5}\rfloor$

TC = $\Theta(n)$
SC = $\Theta(1)$

④ `for (i=n; i>=1; i=i-5)`
`printf("Made Easy");` → $\lfloor\frac{n-1}{5}\rfloor + 1$ times

TC = $\Theta(n)$
SC = $\Theta(1)$

⑤ `for (i=1; i<=n; i=i*2)`
`printf("Made Easy");` → $\lfloor\log_2 n\rfloor + 1$ times

$i \Rightarrow \underbrace{1, 2, 2^2, 2^3, \cdots, 2^k}_{k+1 \text{ times}} \le n$

$2^k \le n$
$k\log_2 2 \le \log_2 n$
$k \le \log_2 n \Rightarrow k = \lfloor\log_2 n\rfloor$

TC = $\Theta(\log n)$
SC = $\Theta(1)$

⑥ `for(i=n; i>=1; i=i/2)`
`printf("Made Easy");` → $\lfloor\log_2 n\rfloor + 1$ times

$i = \underbrace{n, \frac{n}{2^1}, \frac{n}{2^2}, \frac{n}{2^3}, \cdots, \frac{n}{2^k} \ge 1}_{k+1 \text{ times}}$

TC = $\Theta(\log_2 n)$
SC = $\Theta(1)$

(7) for (i=1; i≤n; i=i*5)
printf ("Made Easy"); $\lfloor \log_5 n \rfloor + 1$ times.

$TC = \theta(\log_5 n)$

(8) for (i=2; i≤n; i = i^2)
printf("Made Easy"); → $\lfloor \log_2(\log_2 n) \rfloor + 1$ times

$i \Rightarrow 2, 2^2, 2^4, 2^8, \ldots, 2^{(2^k)} \leq n$ (k+1 times)

$2^{(2^k)} \leq n$

$2^k \log_2 2 \leq \log_2 n$

$2^k \leq \log_2 n$

$k \log_2 2 \leq \log_2(\log_2 n) \Rightarrow k = \lfloor \log_2(\log_2 n) \rfloor$

$TC = \log_2(\log_2 n)$
$SC = \theta(1)$

(9) for (i=n; i≥2; i = $\sqrt{i}$)
printf("Made Easy"); → $\lfloor \log_2(\log_2 n) \rfloor + 1$ times

$i = n, n^{1/2}, n^{1/2^2}, n^{1/2^3}, \ldots, n^{1/2^k}$

$n^{1/2^k} \geq 2$

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

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

$TC = \log_2(\log_2 n)$
$SC = \theta(1)$

---

* for(i=1; i≤n; i=i+c) (or) for(i=n; i≥1; i=i-c) → $TC = \theta(n)$

* for(i=1; i≤n; i=i*c) (or) for(i=n; i≥1; i=i/c) → $TC = \theta(\log_c(n))$

* for(i=c; i^c≤n; i=i^c) (or) for(i=n; i≥c; i=$\sqrt[c]{i}$) → $TC = \theta(\log_c \log_c n)$

---

(10) for (i=2; i≤2^n; i=i^2)
printf ("Made Easy"); → $\lfloor \log_2 n \rfloor + 1$

$i \Rightarrow 2, 2^2, 2^{2^2}, \ldots, 2^{2^k} \leq 2^n$

$2^{2^k} \leq 2^n$

$2^k \log_2 2 \leq n \log_2 2$

$2^k \leq n \Rightarrow k = \lfloor \log_2 n \rfloor$

$TC = \theta(\log_2 n)$

$TC = \theta(\log_2 \log_2(2^n))$

$TC = \theta(\log_2(n))$

(11) for (i = n/2; i≤n; i=i*2)
printf ("Made Easy"); $TC = \theta(1)$

$i \Rightarrow \frac{n}{2}, n$

(12) for (i=1; i≤2^n; i=i*2) $TC = \theta(\log_2(2^n))$
printf ("Made Easy"); $TC = \theta(n)$

Nested Loops :

① Independent nested loop: Inner loop variable not depends on outer loop variable.

Eg. for (i=1; i≤n; i++)
  for (j=1; j≤n^2; j=j*2)
    for (k=n; k≥2; k=$\sqrt{k}$)
      printf ("Made Easy");

$TC = n * 2\log_2 n * \log\log n$

$TC = \theta(n \log n \cdot \log\log n)$

Ex. for (i=1; i≤n^2; i=i+10)
  for (j=n; j≥n/2; j=j/2)
    for (k=n^2; k≥n; k=k-10)
      printf ("Made Easy");

$k \Rightarrow n^2, n^2-10, n^2-2*10, n^2-k*10 \geq n$

$n^2 - k*10 \geq n$

n^2

$TC = n^2 * \log n * n^2$

$= \theta(n^4)$

② GATE

Eg. for (i=1; i≤n²; i = i*2)
  for (j=1; j≤n²; j++)
    for (k=n²; k≥1; k=k/2)
      printf("Made Easy");

$k \to \frac{n^2}{2^0}, \frac{n^2}{2^1}, \frac{n^2}{2^2}, \ldots \frac{n^2}{2^k} \geq 1$

$\frac{n^2}{2^k} \geq 1 \Rightarrow 2^k \leq n^2$

$k \leq 2\log_2 n$

i:	$\log_2(n^2)$	$TC = 2\log_2 n * 2\log_2 n * n^2$
	$2\log_2 n$	$= \Theta(n*(\log_2 n)^2)$

③ GATE

Eg. main()
{ n = 2^{2^k};
  for (i=1; i≤n; i++)
  { j = 2;
    while (j≤n)
    { j = j^2; }
  }

$TC = n * \log\log n \quad [n = 2^{2^k}]$

$= 2^{2^k} \cdot \log_2 \log_2 2^{2^k}$

$= 2^{2^k} \cdot k$

$TC = \Theta(k \cdot 2^{2^k}) \Rightarrow TC = \Theta(n \cdot k)$

$j: 2, 2^2, 2^{2^2}, 2^{2^3}, \ldots 2^{2^k} \leq n$

$2^{2^k} \leq n$

$k = \log_2 \log_2 n$

$\log\log n$

④ for (i=n; i≥1; i=i/2; j=j+i) $TC = \Theta(\log_2 n)$

what is the value of j after termination of loop.

$S_n = \frac{a(1-r^n)}{1-r}; |r|<1$

$i = n, n/2, n/2^2, n/2^3, \ldots, n/2^k$

$j = 1 + n/2 + n/2^2 + n/2^3 + \ldots + n/2^k$

$= 1 + n\left[\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \ldots + \frac{1}{2^k}\right]$

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

$\Rightarrow 1 + n\left(1-\frac{1}{n}\right) = \Theta(n)$

Termination condn:
$i \geq 1$
$\frac{n}{2^k} = 1$
$n = 2^k$

Value of j: $\Theta(n)$

Q. i=1; s=1;
GATE while(s≤n)
{ i++;
  s = s+i
}
Time complexity of loop?

i ⇒	1	2	3	...	k
s ⇒	1	1+2	1+2+3		1+2+3..... $k \leq n$

$1+2+3+\ldots+k \leq n$

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

$k^2 \leq n$

$k = \sqrt{n}$

no. of time loop runs

$TC = \Theta(\sqrt{n})$

---

② Dependent Nested Loop: Inner loop variable depending on variable of outer loop.

x = 0
for (i=1; i≤n; i++)
{ for (j=1; j≤i; j++)
    for (k=1; k≤j; k++)
      x = x+1 → ??
}

ⓐ What is frequency count (TC) of loop?
ⓑ If n = 10, what is final value of x?

Expansion of loop:

i ⇒	1	2	3	...	n
j ⇒	1	1,2	1,2,3		1,2,3 ... n
k ⇒	1	1 2	1 1+2 1+3		1 1+2 1+3 ... 1+n

$S_n = x = x+1 \Rightarrow 1 + (1+2) + (1+2+3) + \ldots + (1+2+3+\ldots+n)$

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

$S_n = \sum t_n = \sum \frac{n(n+1)}{2}$

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

$= \frac{1}{2}\left[\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2}\right]$

$\Rightarrow \frac{1}{2}\frac{n(n+1)}{2}\left[\frac{2n+1}{3}+1\right] = \frac{1}{2}\frac{n(n+1)}{2}\cdot\frac{2n+4}{3} = \frac{1}{2}\frac{n(n+1)(n+2)}{6}$

$\Rightarrow \frac{n(n+1)(n+2)}{6} = \Theta(n^3)$ ⓐ

ⓑ $\frac{10 \times 11 \times 12}{6} = 220 = x$

Q. `for (i=1; i≤n; i++)`
`{ for (j=1; j≤n; j=j+i)`
`x = x+1;`
`}`

What is TC of loop?

i	1	2	3	...	n
j	1 to n	~~1 to n~~ n/2 times	n/3 times	...	1

x=x+1: ~~n~~ + n/2 + n/3 + ... + n/n

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

$= \Theta(n \log_e n)$

$1+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n} = \log_e n$

$\int_{x=1}^{n} \frac{1}{n}\,dx = \log_e n$

22/08/2017:

Recursive Algorithm:

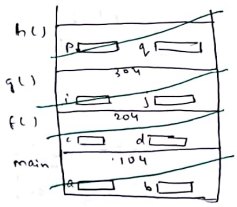
→ Stack DS required to run recursive algo and subroutine calls to store return address.

```
Algo main()          f()              g()              h()
{ Int a,b;           { Int c,d;       { int i,j;       { int p,q;
101 st1              201 st11         301 st21         401 st31
102 if (cond) f();   202 st12         302 st22         402 st32
103 st3;             203 if (cond) g(); 303 if (cond) h()
104 st4              204 st13         304 st23
}                    205 st14         305 st24
                     }                }
```

(true / true / true)

Main() → f() → g() → h() (4 level function calls [4 stack entries])

PC: ~~TOS~~ ~~TOS~~ ~~104~~ 201



~~Overhead of function call:~~
1. ~~Push [stack]~~
2. ~~TOS [stack] ← PC~~
3. ~~PC ← New address [11~~
4.
5.

Overhead of function call: [Microprogram for function call]

1. Top[stack] ← PC
2. Push [stack]
3. PC ← New address [It executes]
4. Pop [stack]
5. PC ← Top [stack]

E.g. How much stack space required to run given program.

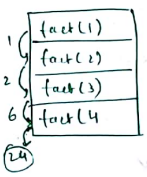
```
main()   f()   g()   h()
{ int p,q;{    {     {
  f();    :    :     :
  g();    }    }     }
  h();
}
```

2 stack entries


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

fact(4) { 4 * fact(3) } fact(3) { 3 * fact(2) } fact(2) { 2 * fact(1) } fact(1) { return 1 }

Sc = Θ(n)



fact(n) → fact(n-1) → fact(n-2) → ... → fact(1⁰)

Depth of recursion: n levels [n stack entries to run]

Q. Algo Rec(n)
```
{ if (n≤1) return(1);   → a
  else return (rec(n-1) + rec(n-1) + n);   → c
}
```

rec(n) → T(n)

{ x = rec(n-1); y = rec(n-1); return (x+y-1); } → 2T(n-1) + c

a. TC of rec(n): T(n) is TC of rec(n)

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

Substitution method to solve recurrence relation

$T(n) = 2T(n-1) + c$ $\quad [T(n-1) = 2T(n-2) + c]$

$= 2[2T(n-2) + c] + c$

$= 2^2 T(n-2) + 2c + c$ $\quad [T(n-2) = 2[T(n-3) + c]$

$= 2^2[2T(n-3) + c] + 2c + c$

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

$\vdots$ k times

$= 2^k T(n-k) + c[2^{k-1} + \dots + 2 + 1]$

$= 2^k T(n-k) + c[2^k - 1] \longrightarrow S_n = \frac{a[r^n - 1]}{r-1}; r>1 = \frac{1(2^k - 1)}{2-1} = 2^k - 1$

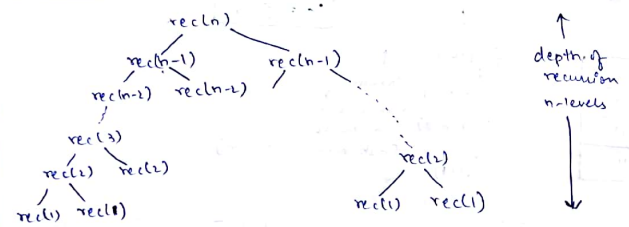
$n - k = 1 \Rightarrow k = n - 1$

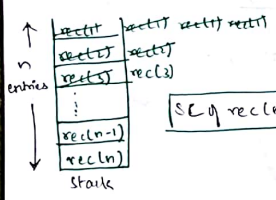
$= 2^{n-1} T(1) + c[2^{n-1} - 1]$

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

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

(B) SC and # of function calls of Rec(n):





SC of rec(n) = $\Theta(n)$

No. of function calls:

$= [1 + 2 + 2^2 + \dots + 2^{n-1}]$

$= 2^n - 1$

Note:
For recursive algorithm if each function call required const. time [except time for subroutine call], then TC of algo. = Θ (# of function calls)

(C) Value returned by algorithm: $rec(n) = \begin{cases} 1 & , \text{if } n \le 1 \\ 2 \cdot rec(n-1) + n & , \text{if } n > 1 \end{cases}$ $\quad \Theta(2^n)$

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

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

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

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

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

Q. Algo rec(n)

```
{ if (n ≤ 1)
    return (1);
  else
    return (2 * rec(n-1) + n)
}
```

(A) TC of rec(n)

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

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

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

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

$\vdots$ k times

$= T(n-k) + k \cdot c$

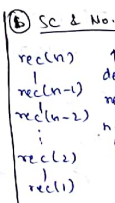
Termination: $n - k + 1 = 1 \Rightarrow k = n - 1$

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

$T(n) = a + c(n-1) \Rightarrow \boxed{T(n) = \Theta(n)}$

```
rec(n)
{ x = rec(n-1)   ⟶ T(n-1)
  return (2*x + n) ⟶ c
}
```

(B) SC & No. of function calls:



SC = $\Theta(n)$

No. of function calls = n

(C) Value returned by algorithm:

$rec(n) = \begin{cases} 1 & ; \text{if } n \le 1 \\ 2 \cdot rec(n-1) + n & ; \text{if } n > 1 \end{cases}$

Q. Algo Rec(n)

```
{ if(n≤1)
    return (1);
  else
  { rec(n-1)
    for ( i=1;i≤n;i++)
      printf("Mode easy");
  }
}
```

(a) TC of rec(n): $T(n) = \begin{cases} a; & n \le 1 \\ T(n-1)+n; & n>1 \end{cases}$

$T(n) = T(n-1)+n$ $\quad [T(n-1) = T(n-2)+(n-1)]$

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

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

: k times

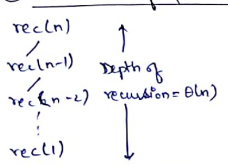
$= T(n-k)+(n-(k-1))+\dots+(n-1)+n$

Termination of $n-k=1$, $k=n-1$

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

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

(b) SC and no. of function calls:



SC of rec(n) = Θ(n) // stack space

No. of function calls = Θ(n)

Q. Algo rec(n)

```
{ if(n≤1)
    return (1)
  else
  { rec(n-1);
    rec(n-1);
    for (i=1;i≤n;i++)
      Printf("Mode Easy");
  }
}
```

(a) TC of rec(n): $T(n) = \begin{cases} a & ; n \le 1 \\ 2T(n-1)+n; & n>1 \end{cases}$

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

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

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

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

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

: k times

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

$n-k=1$, $k=n-1$

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

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

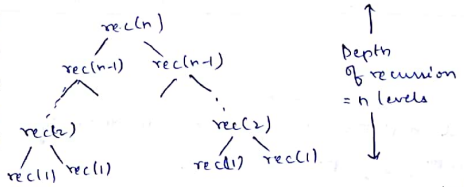
$2T(n) = 2^n\cdot 1 + 2^{n-1}\cdot 2 + 2^{n-2}\cdot 3 + \dots + 2^2(n-1)+2n$

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

$T(n) = \left[\frac{2(2^n-1)}{2-1}\right] - n \Rightarrow T(n) = 2^{n+1}-2-n$ $\quad TC = \Theta(2^n)$

(b) SC and no. of function calls:

SC = Θ(n)



# of function calls $= 1+2+2^2+\dots+2^{n-1} = 2^n-1$

Simplification of A.P. & G.P. series:

Q. $T(n) = 5^n\cdot n + 5^{n-1}\cdot(n-1) + \dots + 5^2\cdot 2 + 5^1\cdot 1$

$5T(n) = 5^{n+1}\cdot n + 5^n(n-1) + \dots + 5^3\cdot 3 + 5^2\cdot 1$

$5T(n)-T(n) = 5^{n+1}\cdot n - 5^n - 5^{n-1} - \dots - 5^3 - 5^2 - 5$

$4T(n) = 5^{n+1}\cdot n - [5^n + 5^{n-1} + \dots + 5^3 + 5^2 + 5]$ $\quad \left[\frac{5[5^n-1]}{5-1}\right]$

$4T(n) = 5^{n+1}\cdot n - \frac{5}{4}(5^n-1)$

$T(n) = \frac{1}{4}\left[5^{n+1}\cdot n - \frac{5}{4}(5^n-1)\right]$ $\quad TC = \Theta(n\cdot 5^{n+1})$

Q. $T(n) = \frac{n}{4^n} + \frac{n-1}{4^{n-1}} + \frac{n-2}{4^{n-2}} + \dots + \frac{2}{4^2} + \frac{1}{4}$

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

$\frac{3T(n)}{4} = \frac{1}{4^n} + \frac{1}{4^{n-1}} + \frac{1}{4^{n-2}} + \dots + \frac{1}{4^3} + \frac{1}{4^2} + \frac{1}{4} - \frac{n}{4^{n+1}}$ $\quad \left[\frac{\frac{1}{4}\left[1-\frac{1}{4^n}\right]}{\frac{3}{4}} = \frac{1}{3}\left[1-\frac{1}{4^n}\right]\right]$

$\frac{3T(n)}{4} = \frac{1}{3}\left[1-\frac{1}{4^n}\right] - \frac{n}{4^{n+1}} \Rightarrow T(n) = \frac{4}{3}\left[\frac{1}{3}\left[1-\frac{1}{4^n}\right] - \frac{n}{4^{n+1}}\right]$

$T(n) = \frac{4}{3}\left[\underbrace{\frac{1}{3}\left(1 - \underbrace{\frac{1}{4^n}}_{\sim 0}\right)}_{\text{leading term is constant}} - \underbrace{\frac{n}{4^{n+1}}}_{\sim 0}\right]$ $\boxed{T(n) = \Theta(1)}$

Q. $T(n) = \frac{1}{3^n} + \frac{2}{3^{n-1}} + \frac{3}{3^{n-2}} + \cdots + \frac{n-1}{3^2} + \frac{n}{3}$ $\Theta(n)$

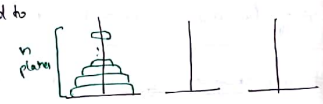
$\frac{T(n)}{3}$

Towers of Hanoi:

Problem statement:

Given 3 towers [X, Y, Z] and n plates in Tower X with descending order of diameter from bottom to top.

Compute no. of plate movements required to transfer n plates from X to Y:

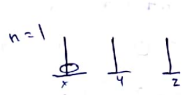


Constraints:

1. Only top plate allowed to move from tower
2. At any time lower diameter plate not below upper diameter plate.

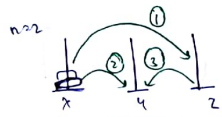
n=1:

# of moves = 1 (X→Y)



n=2:

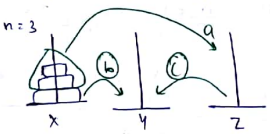
# of moves = 3 [X→Z, X→Y, Z→Y]



n=3:

a: X→Y, X→Z, Y→Z
b: X→Y
c: Z→X, Z→Y, X→Y

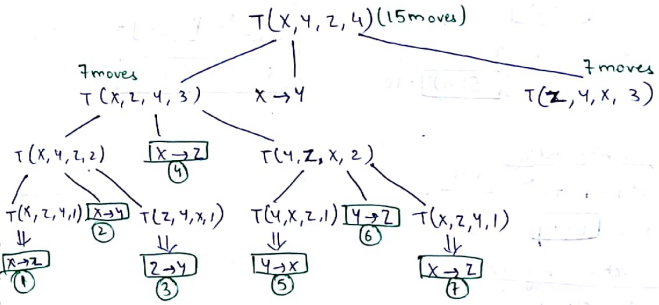
} 7 moves



Procedure to transfer n plates from X to Y using Z temp:

ⓐ Transfer Top (n-1) plates from X to Z using Y is temp.
ⓑ move one plate from X to Y
ⓒ Trans (n-1) plates from Z to Y using X as Temporary.

```
Algo ToH (X, Y, Z, n)
{ //ToH (source, dest, temp, # of plates)
  if (n==1)
     return (X→Y)
  else
   { ToH(X, Z, Y, n-1)
     move X→Y
     ToH(Z, Y, X, n-1)
   }
}
```



T(n): No. of moves to transfer n plates from X to Y:

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

$T(n) = 2T(n-1) + 1$
$= 2[2T(n-2)) + 1] + 1$
$= 2^2 T(n-2) + 1$

$T(n) = \Theta(2^n)$
$T(n) = 2^n - 1$ (Undecidable problem)

SC = $\Theta(n)$

Q. Algo rec(n)

```
{ if (n ≤ 1)
    return(1)
  else
    return (rec(n/2) + rec(n/2) + n)
}
```

$T(n) = \begin{cases} a & ; n \le 1 \\ 2T(n/2) + c & ; n > 1 \end{cases}$

ⓐ $T(n) = 2T(n/2) + c$ $\quad \because T(n/2) = 2T(n/2^2) + c$

$= 2[2T(n/2^2) + c] + c$

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

$= 2^2[2T(n/2^3) + c] + 2c + c$

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

⋮ k times

$= 2^k T(n/2^k) + c[2^{k-1} + \dots + 2^2 + 2 + 1]$

$\Downarrow$ $2^k - 1 = n - 1$

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

$T(n) = nT(1) + c[n-1]$

$\boxed{T(n) = a.n + c(n-1)} \boxed{= \Theta(n)} = TC$

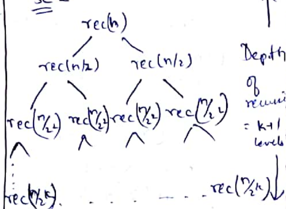
No. of function calls:

$\Rightarrow 1 + 2 + 2^2 + \dots + 2^k = 2^{k+1} - 1$

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

ⓑ SC & no. of function calls:

SC =



k+1 levels $\Rightarrow k = \lfloor \log_2 n \rfloor$

~~SC~~ $= \lfloor \log_2 n \rfloor + 1$

$\boxed{SC = \Theta(\log_2 n)}$

ⓒ Value return by rec(n):

$rec(n) = \begin{cases} 1 & ; n \le 1 \\ 2 \cdot rec(n/2) + n & ; n > 1 \end{cases}$

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

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

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

$= 2^2[2T(n/2^3) + n/2^2] + 2n$

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

k times:

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

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

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

$= n.1 + n\log_2 n \Rightarrow \boxed{n(1 + \log_2 n)}$ $\Theta(n\log n)$

Q. Algo rec(n)

```
{ if (n ≤ 2)
    return;
  else
    return (rec(√n) + rec(√n) + c)
}
```

$T(n) = \begin{cases} a & ; n \le 2 \\ 2T(\sqrt{n}) + c & ; n > 2 \end{cases}$

ⓐ $T(n) = 2T(\sqrt{n}) + c$

$= 2[2T(n^{1/4}) + c] + c$

$= 2^2 T(n^{1/4}) + 2c + c$

$= 2^2[2T(n^{1/8}) + c] + 2c + c$

$= 2^3 T(n^{1/2^3}) + 2^2c + 2c + c$

⋮ k times

$= 2^k T(n^{1/2^k}) + c[2^{k-1} + \dots + 2 + 1]$

$\Downarrow$ 2

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

$n = 2^{2^k}$

$= \log_2 n\, T(2) + c[2^k - 1]$

$= \log_2 n . a + c[\log_2 n - 1]$

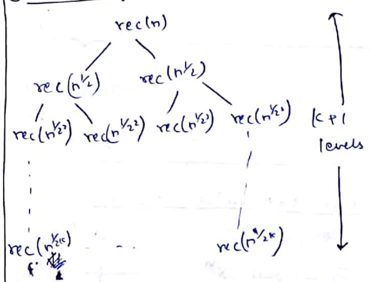
$\boxed{T(n) = a.\log_2 n + c[\log_2 n - 1] = \Theta(\log_2 n)}$

No. of function calls:

$\Rightarrow 1 + 2 + 2^2 + \dots + 2^k = 2^{k+1} - 1$

$\Rightarrow 2\log_2 n - 1 = \boxed{\Theta(\log_2 n)}$

ⓑ SC & no. of function calls:



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

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

$\boxed{SC = \Theta(\log_2(\log_2 n))}$

ⓒ ~~Value returned by rec(n):~~

~~rec(n) = {~~

* **NOTE**

rec function	SC of rec(n)
$rec(n) = rec(n-c) + n$	$\approx \frac{n}{c} = \Theta(n)$
$rec(n) = rec(n/c) + n$	$\approx \log_c n = \Theta(\log_c n)$
$rec(n) = rec(\sqrt[c]{n}) + n$	$\approx \log_c \log_c n = \Theta(\log_c \log_c n)$

Exponential TC rec. Algo:

```
rec(n)
{ rec(n-1)
  rec(n-1)
}
```

```
rec(n)
{ rec(n-1)
  rec(n-2)
}
```

```
rec(n)
{ rec(n-1)
  rec(n-1)
  for(i=1; i<=n; i++)
    x=x+1;
}
```

```
rec(n)
{ rec(n-1)
  rec(n-1)
  rec(n-1)
}
```

```
rec(n)
{ rec(n-c1)
  rec(n-c2)
}
```

Fast Exponential Algorithm:

Compute: x^n

To compute x^n

(a) $y = x^{n/2}$

(b) if n is even
return(y*y)
else
return(x*y*y)

$x^{64} = x^{32} * x^{32}$
$x^{16} * x^{16}$
$x^8 * x^8$
$x^4 * x^4$
$x^2 * x^2$
$x * x$
6 multiplications
$= \log_2 n$

$x^{100} = x^{50} * x^{50}$
$x^{25} * x^{25}$
$x^{12} * x^{12}$
$x^6 * x^6$
$x^3 * x^3$
$x * x * x$
8 multi.
$\log_2 100$

```
Algo FastExp(x, n)
{ // Compute x^n
  if(n==0) return(1);
  else if(n==1) return(x);
  else
  {
    y = FastExp(x, n/2);
    if(n%2==0)
      return(y*y);
    else
      return(x*y*y);
  }
}
```

a

$T(n/2)+c$

$x^5 \cdot x^5$ FastExp(x,10)

$x^5 = x \cdot x^2 \cdot x^2$ FastExp(x,5)

$x \cdot x^2$ FastExp(x,2)

x FastExp(x,1)

TC of Fast Exp Algo:

$$T(n) = \begin{cases} a & ; n \le 1 \\ T(n/2) + c & ; n > 1 \end{cases}$$

$TC = \Theta(\log_2 n)$

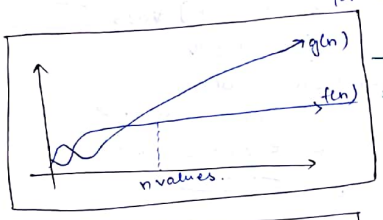
SC of Fast Exp Algo (Recursive) $\Rightarrow \Theta(\log_2 n)$

No. of function calls $= \Theta(\log n)$

ASYMPTOTIC NOTATIONS:

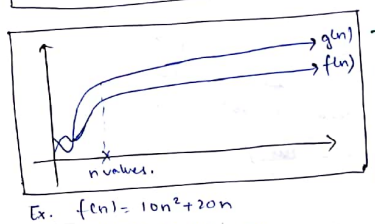
Asymptotic comparison of non negative functions: Growth rate comparison for non negative functions for large n-values.

f(n) g(n) Asymptotic comparison: Growth rate comparison of f(n) & g(n) for large n values.



→ g(n) is asymptotically bigger than f(n)

*[g(n) growth rate more than growth rate of f(n) for large n values]



→ f(n) & g(n) are asymptotically equal.

f(n) & g(n) are asymptotically equal iff $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ = constant.

f(n) & g(n) are asymptotical equal iff $f(n) = O(g(n))$ & $g(n) = O(f(n))$

Ex. $f(n) = 10n^2 + 20n$

$g(n) = 2n^2 + 100n + 100$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^2[10 + 20/n]}{n^2[2 + 100/n + 100/n^2]}$

∴ f(n) & g(n) are asymp. equal

f(n) asymptotically bigger than g(n): iff

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

$f(n) = 5n^2 + 10$

$g(n) = 10n + 20$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n[5n + 10/n]}{n[10 + 20/n]} = \frac{\infty}{10} = \infty$

∴ f(n) asymp. bigger than g(n)

f(n) asymptotically smaller than g(n): iff

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

$f(n) = 2n^2$

$g(n) = n^3 + 10$

$\lim_{n\to\infty} \frac{2n^2}{n^2(n + 10/n^2)} = \frac{2}{\infty} = 0$

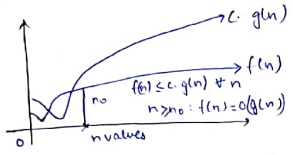
∴ f(n) asymp. smaller than g(n)

Big-O Notation:

→ $f(n)$, $g(n)$ are non negative functions.

→ $f(n) = O(g(n))$ iff

$f(n) \le c \cdot g(n)$ for all n values where $n \ge n_0$.



E.g. $2n+3 = O(n)$

$f(n) = 2n+3$ $g(n) = n$

$[2n+3] \le c[n]$ for all values $n \ge n_0$

$[2n+3] \le 3[n]$ $\forall n \ge 3 \Rightarrow$ $2n+3 = O(n)$

$f(n)$ $c.g(n)$

$f(n) = O(g(n))$ iff
$g(n)$ asymptotically bigger or equal to $f(n)$

Eg. $2n+3 = O(n^2)$

$f(n) = 2n+3$ $g(n) = n^2$

$2n+3 \le c.n^2$ $\forall n$ $n \ge n_0$

$2n+3 = O(n^2)$

E.g. $2n+3 = O(\log_{10} n)$

	10	100	10^3	10^4	10^5
$f(n) = 2n+3$	23	203	2003	20003	200003
$g(n) = 10000 * \log_{10} n$	10000	20000	30000	40000	50000

$2n+3 \le c.\log_{10} n$ [not true for all n values]

$2n+3 \ne O(\log_{10} n)$

Asymptotic Comparisons:

* Decrement Function: (lesser)

$f_1(n) = \frac{c}{n}, \frac{1}{n^2}, \frac{n^2}{2^n}$

$\left[\frac{n^2}{2^n} < \frac{1}{n^2} < \frac{c}{n}\right]$

* Constant Functions: (Bigger than decrement)

$f_2(n) =$ constant

* log functions: (Bigger than constants)

$\log n, (\log n)^k, (\log\log n), (\log\log n)^k$

$[\log\log n < (\log\log n)^k < \log n < (\log n)^k]$

(small ext poly bigger than log largest)

* Polynomial functions:

$n^{0.1}, n^2, n^{0.5}, n\log n, n^3$

$[n^{0.1} < n^{0.5} < n\log n < n^2 < n^3]$

* Exponential Functions: (Bigger than poly)

$2^n, 3^n, n!, n^n, (1.01)^n$

$[1.01^n < 2^n < 3^n < n! < n^n]$

Asymptotic Comp.

dec fun < const fun < log fun < poly fun < exponential fun

GATE:

Q1. Which is false?

(a) $100\log n = \frac{\log n}{100} = O(\log n)$

(b) if $x < y$ then $n^x = O(n^y)$

(c) $n^q = O(a^n)$ $(a > 1$ const$)$

(d) $\sqrt{\log_2 n} = O(\log_2 \log_2 n)$

(2) $f(n) = \log_y n^x$ $(x, y$ const$)$ $f(n) = x \cdot \log_y n = \frac{x}{\log_2 y} \cdot \log_2 n$; $g(n) = \log_2 n$

constant

$g(n) = \log_2 n$

$f(n)$ & $g(n)$ asymp. equal

Which is true?

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

(b) ~~a & b~~ $g(n) = O(f(n))$

(c) a & b

(d) None

(3) $f(n) = \begin{cases} n^3 & ; 0 \le n \le 10000 \\ n & ; n > 10000 \end{cases}$

$g(n) = \begin{cases} n & ; 0 \le n \le 100 \\ n^2 & ; n > 100 \end{cases}$

Which is true?

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

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

(c) a & b

(d) None

(4) $f(n) = \begin{cases} 2^n & \text{for even } n \\ n & \text{otherwise} \end{cases}$

$g(n) = \begin{cases} 2^n & \text{for odd } n \\ n & \text{otherwise} \end{cases}$

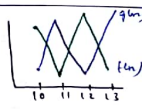
Which is true?

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

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

(c) a & b

(d) None



$f(n)$ & $g(n)$ are asymp. non comparable

$f(n) \ne g(n)$ | $g(n) \ne O(f(n))$

(5) $f(n) = \sqrt{n}$ $g(n) = n^{1+\sin n}$; which is true?

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

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

(c) a & b

(d) None

n	$f(n) = n^{0.5}$	$g(n) = n^{1+\sin n}$
90	$(90)^{0.5}$	$< (90)^2$
180	$(180)^{0.5}$	$< (180)^1$
270	$(270)^{0.5}$	$> (270)^{0.7}$

⑥ $f(n) = n^{1+\sin n}$ $g(n) = n^2$; which is true ?

(a) $f(n) = O(g(n))$ (c) a & b

(b) $g(n) = O(f(n))$ (d) None

***

⇒ $f(n) = n!$ $g(n) = n^n$

$f(n) = n(n-1).(n-2) \ldots 2 \times 1$

$g(n) = n.n.n. \ldots n.n$

n^n asymptotically bigger than $n!$

$n! = O(n^n)$

$n^n \neq O(n!)$

⇒ $f(n) = n!$ | $g(n) = (n-1)!$ | $h(n) = (n-1)! + n$

$(n-1)! + n = (n-1)! < n!$ ⇒ $g(n) = h(n) < f(n)$

⇒ $f(n) = 2^n$ $g(n) = n^{\sqrt{n}}$

Solution: $\log(f(n))$ > $\log(g(n))$

$= \log_2 2^n$ $= \sqrt{n} \cdot \log_2 n$

$= n \log_2 2$ $=$

$= \sqrt{n} \cdot \sqrt{n}$

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

$2^n \neq O(n^{\sqrt{n}})$

If $\log f(n)$ asym.p. bigger than $\log g(n)$, then $f(n)$ asymp. bigger than $g(n)$

→ If $\log f(n)$ asymp bigger than $\log g(n)$ then $f(n)$ asym bigger than $g(n)$.

→ If $f(n)$ asymp. bigger than $g(n)$ then : [$\log f(n)$ asy bigger than $\log g(n)$]

Eg. $f(n) > g(n)$

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

$n^3 > n^2 \Rightarrow \log(n^3) = \log(n^2)$

OR

[$\log f(n)$ asy. equal to $\log g(n)$]

→ If $\log f(n)$ asymp. equal to $\log g(n)$ then $f(n) > g(n)$ asy.

(or) $f(n) < g(n)$ asy.

(or) $f(n) = g(n)$ asy.

→ if $f(n)$ & $g(n)$ asy. equal, then $\log f(n)$, $\log g(n)$ also asy equal

⇒ $f(n) = \log(n!)$

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

Stirling Formula:

$n! \underset{\text{equal}}{\overset{\text{asy}}{\approx}} \sqrt{2\pi n e} \left(\frac{n}{e}\right)^n$

$\therefore \log(n!) \underset{\text{equal}}{\overset{\text{asy}}{\approx}} \log\left(\sqrt{2\pi n e}\left(\frac{n}{e}\right)^n\right)$

$\Rightarrow \frac{1}{2}\log 2\pi n e + n \log\left(\frac{n}{e}\right) = \left[\frac{1}{2}\log 2\pi + \frac{1}{2}\log n + n\log n - n\log e\right]$

$\log(n!) \underset{\text{equal}}{\overset{\text{asy}}{\approx}} (n \log n)$

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

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

$\therefore f(n)$ and $g(n)$ are asymptotically equal.

25/08/2017

Testing if $f(n)$ is polynomially bounded or exponentially bounded :

⇒ if $\log f(n) = O(\log n)$

$\log(n) \leq \log n$ [Asymptotically]

Then $f(n)$ is polynomially bounded.

⇒ if $\log f(n) \neq O(\log n)$

$\log(n) > \log n$ [Asymptotically]

Then $f(n)$ is exponentially bounded.

Eg. $f(n) = n^k$ // Poly fun.

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

Eg. $f(n) = 2^n$

$\log f(n) = n \log 2 \neq O(\log n)$

// Expo. fun.

Q. $f(n) = n!$

$O(n \log n) \neq O(\log n)$

Exponential

Q. $f(n) = (\log n)!$

$\log(n) \approx (\log n)\log(\log n) \leq \log n$

~~Polynomial~~ Exponential

③ $f(n) = (\log\log n)!$

Polynomial

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

⑤ $f(n) = n^{\sqrt{n}}$

$\log f(n) = n^{1/2}\log n \neq O(\log n)$

Exponential

⑦ $f(n) = (\log n)^{\log n}$

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

~~Polynomial~~ // Exponential

⑨ $f(n) = (\log\log n)^{\log n}$

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

// Exponential

④ $f(n) = \log(n!)$

$= n\log n$

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

$= \log n + \log\log n \leq \log n$

// Polynomial

⑥ $f(n) = n^{\log n}$

$\log f(n) = \log n \log n$ & $\log(n)$

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

~~Polynomial~~ Exponential

⑧ $f(n) = (\log n)^{\log\log n}$

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

// Polynomial

⑩ $f(n) = \log(\log n)!$

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

~~Polynomial~~

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

// Polynomial

xxx
- $n!$, $(\log n)!$ — Exponential function
- $(\log\log n)!$ — Polynomial function

$n = 2^{2^{64}}$ // largest no.

$n!$ = too too big

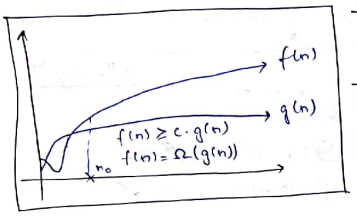
$(\log n)!$ = too big

$\log(\log n)$ = small

Omega Notation (Ω):

⇒ $f(n)$ and $g(n)$ non negative functions.

$f(n) = \Omega(g(n))$ iff $f(n) \geq c \cdot g(n)$ $\forall$ $n \geq n_0$.



→ $f(n) = \Omega(g(n))$ iff $g(n)$ asymptotically smaller or equal to $f(n)$.

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

Eg. 2^n $n^{\sqrt{n}}$

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

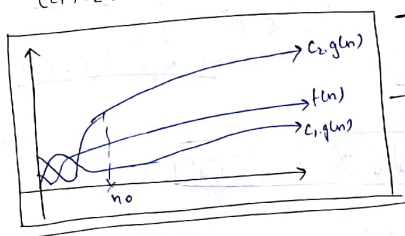
$2^n = \Omega(n^{\sqrt{n}})$

Theta Notation (Θ):

→ $f(n)$ and $g(n)$ non negative functions.

$f(n) = \Theta(g(n))$ iff $c_1 g(n) \leq f(n) \leq c_2 g(n)$ $\forall$ $n \geq n_0$

[c_1, c_2, n_0 → constants]



→ $f(n) = \Theta(g(n))$ iff $g(n)$ asymptotically equal to $f(n)$.

→ $f(n) = \Theta(g(n))$; if $f(n) = O(g(n))$ & $f(n) = \Omega(g(n))$

Little-o Notation:

→ $f(n)$ and $g(n)$ are non-negative functions.

$f(n) = o(g(n))$ iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

$f(n) = o(g(n))$ iff $g(n)$ asymptotically bigger than $f(n)$.

Test:

1. $n! = o(n^n)$ ✓
2. $2^n = o(n!)$ ✓
3. $\log n! = o(n\log n)$ ✗
4. $\log n! = o((\log n)!)$ ✓
5. $n^2 \log n = o(n^{2.01})$ ✓

Little Omega (ω):

→ $f(n)$ & $g(n)$ non negative function.

$f(n) = \omega(g(n))$ iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

→ $f(n) = \omega(g(n))$ iff $g(n)$ asymptotically smaller than $f(n)$.

$\log^k n = (\log n)^k$

Q. Find given true/false:

① $2^{n+1} = \Theta(2^n)$ ✓

② $2^{2n+1} = \Theta(2^n)$ ✗

③ $x^{n+1} = \Theta(x^n)$ ✓ (x const.)

④ $x^{2n+1} = \Theta(x^n)$ ✗ (x const.)

⑤ $n! = \Theta(n^n)$ ✗

⑥ $\log n! = \Theta(n \log n)$ ✓

Q. Write given func.^{n} in asymptotically increasing order:

$\log^k \log\log n$, $\log\log^k \log n$, $\log\log\log^k n$, $\log\log\log n^k$, [k const.]

$\log^k \log\log n = (\log\log\log n)^k$

$\log\log^k \log n = \log(\log\log n)^k$
$= \log(k \cdot \log\log n)$
$= \log k + \log\log\log n$

$\log\log\log^k n = \log\log(k \cdot \log n)$
$= \log\log k + \log\log\log n$

$\log\log^k \log n = \log\log\log^k n = \log\log\log n^k \le \log^k \log\log n$

Q. Write in any increasing order:

$\log n!$; $\sqrt{n}$; $(\log\log n)!$; $(\log n)!$; 2^n ; $2^{n/2}$; 2^{2n} ; 2^{n+1} ; $(\log n)^{\log n}$; $(\log n)^n$; $n^{\log n}$; $n \log n$

Polynomial:
$\log n!$; $\sqrt{n}$; $(\log\log n)!$; $n \log n$;

Exponential:
$(\log n)!$, 2^n , $2^{n/2}$, 2^{2n} , 2^{n+1} , $n^{\log n}$, $(\log n)^{\log n}$, $(\log n)^n$

$\log(\log n)!$
$\log n! = n \log n = \log n + \log\log n$
$n \log n = \log n + \log\log n$
$\sqrt{n} = \frac{1}{2} \log n$
$\log n^n = n \log\log n$
$(\log\log n)! \ge \log\log n \log\log\log n$

$(\log\log n)! < \sqrt{n} < \log n! = n \log n <$

Exp.
$(\log n)! = \log n (\log\log n)$
$n^{\log n} = (\log n)^2$
$(\log n)^{\log n} = \log n \log\log n$.
$(\log n)^n = n \log\log n$

$(\log\log n)! < (\log n)^{\log n} < n^{\log n} < 2^{n/2} < 2^n = 2^{n+1} < 2^{2n} < (\log n)^n$

Q. $O(n) + \Omega(n) + \Theta(n) =$

ⓐ $O(n)$
ⓑ $\Omega(n)$
ⓒ $\Theta(n)$
ⓓ none

$O(n)$: $\le n$ + ; $\Omega(n)$: $\ge n$ + ; $\Theta(n)$: $= n = \ge n$ Ω

Q. $f(n) = O(n)$
$g(n) = \Omega(n)$
$h(n) = \Theta(n)$
$f(n) + g(n) + h(n) = \Omega(n)$

Armstrong Rules over notations:

ⓐ Reflexivity: [O, Ω, Θ allow reflexivity]
$f(n) = O(f(n))$
$f(n) = \Omega(f(n))$
$f(n) = \Theta(f(n))$

ⓑ Transitivity: If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$
similarly Ω, Θ, o, ω also represent transitive property.

ⓒ Asymmetric: ① O, Ω
$f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$

[Transpose symmetry]

② o, ω
$f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$

* $f(n) * c = \frac{f(n)}{c} = \Theta(f(n))$

$f(n) * c = \frac{f(n)}{c} = O(f(n))$

$f(n) * c = \frac{f(n)}{c} = \Omega(f(n))$

Q. find True/False:

$f(n) + g(n) = O(\max(f(n), g(n)))$ True

$f(n) + g(n) = \Omega(\max(f(n), g(n)))$ True

$f(n) + g(n) = \Theta(\max(f(n), g(n)))$ True

$f(n) + g(n) = o(\max(f(n), g(n)))$ false

$f(n) + g(n) = \omega(\max(f(n), g(n)))$ false

Q. Test

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

$f(n) = \frac{1}{n} \Rightarrow \frac{1}{n} = O\left(\frac{1}{n^2}\right)$ False

---

Q. Test

If $f(n) = O(g(n))$ then $2^{f(n)} = O(2^{g(n)})$ ~~True~~ False

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

$2n = O(n) \Rightarrow 2^{2n} = O(2^n)$ (False)

Q. $(x+a)^n$. a is const.

x, n not const.

$(x+a)^n = c_n x^n + c_{n-1} x^{n-1} + \ldots + c_1 x + c_0 = \Theta(x^n)$

Proof: $c_n \cdot x^n \le [x+a]^n \le [c_n + c_{n-1} + \ldots + c_0] \cdot x^n$

$c_n \cdot x^n \le [c_n x^n + c_{n-1} x^{n-1} + \ldots + c_1 x + c_0] \le [c_n + c_{n-1} + \ldots + c_0] \cdot x^n$

const. $g(n) \le f(n) \le$ const. $g(n)$

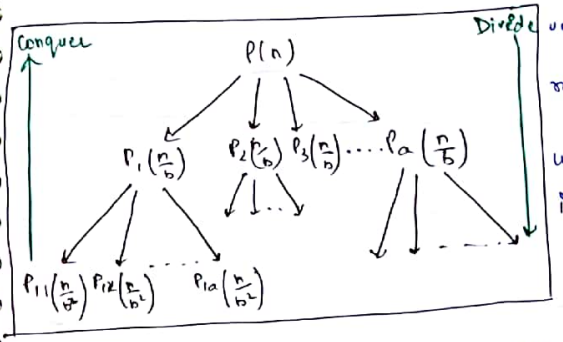
$f(n) = \Theta(g(n))$

$(x+a)^n = \Theta(x^n)$

Divide and Conquer Design Technique:

→ Problem P with "n" inputs if not small problem.

→ Divide P into "a" no. of subproblems [P, P₂ ... Pa] each with $\frac{n}{b}$ i/ps.



until sub problem becomes small

↓

return set of small solution

↓

using conquer combine solutions in bottom up approach.

→ If subproblem not small, divide into sub-subproblems and conquer solution in bottom up approach until returns solution (P).

Until sub-subproblem becomes small return solution of small problem.

Control Abstraction of D & C:

```
Algo DAndC(n)
{
g(n)  If (n==1)
        return (solution (P))
      else
      { //Divide
        Divide Problem P with n i/ps
        into "a" no. of subproblems with
        n/b inputs each.
aT(n/b)  Conquer (DAndC(n/b), DAndC(n/b), ..., DAndC(n/b))
+f(n)          T(n/b)        T(n/b)          T(n/b)
      }
}
```

---

* Recurrence Relation of D and C Algo:

$$T(n) = \begin{cases} g(n) = \Theta(1) & ; \text{if } n=1 \\ aT(n/b) + f(n) & ; \text{if } n>1 \end{cases}$$

a & b constants.

T(n) → TC of DAndC(n)

g(n) → TC to solve small problem

f(n) → TC to divide & conquer

Solving of Recurrence Rel:

1. Substitution method
2. Recursive tree method
3. Master's method
4. Change of variable method
5. Generating function method

Substitution Method:

① $T(n) = \begin{cases} 7T(n/2) + n^2 & n>1 \\ a & n=1 \end{cases}$

$T(n) = 7T(n/2) + n^2$

$= 7\left[7T\left(\frac{n}{2^2}\right) + \left(\frac{n}{2}\right)^2\right] + n^2$

$= 7^2 T\frac{n}{2^2} + 7\left(\frac{n}{2}\right)^2 + n^2$

$= 7^2\left[7T\left(\frac{n}{2^3}\right) + \left(\frac{n^2}{2^2}\right)\right] + 7\left(\frac{n^2}{2^2}\right) + n^2$

$= 7^3 T\left(\frac{n}{2^3}\right) + 7^2\frac{n^2}{2^{2^2}} + 7^1\frac{n^2}{2^2} + n^2$

$\vdots$

$= 7^k T\left(\frac{n}{2^k}\right) + 7^{k-1}\frac{n^2}{2^{2k-1}} + \ldots + \frac{7n^2}{2^2} + n^2$

$= 7^k T\left(\frac{n}{2^k}\right) + n^2\left[\frac{7^{k-1}}{2^{2k-1}} + \ldots + \frac{7}{2^2} + 1\right] \Rightarrow \frac{\left(\frac{7}{4}\right)^k - 1}{\frac{7}{4} - 1} = \frac{4}{3}\left[\left(\frac{7}{4}\right)^k - 1\right]$

$\frac{n}{2^k} = a \Rightarrow \frac{n}{a} = 2^k \Rightarrow k = \log\left(\frac{n}{a}\right)$; $a=1$, $k = \log n$

$= 7^{\log\left(\frac{n}{a}\right)} \cdot a + \frac{4}{3}n^2\left[\left(\frac{7}{4}\right)^{\log n} - 1\right]$

$\Rightarrow 7^{\log n} \cdot T(1) + \frac{4}{3}n^2\left[\left(\frac{7}{4}\right)^{\log_2 n} - 1\right] = a^{\log_c b} = b^{\log_c a}$

$\Rightarrow c \cdot n^{\log_2 7} + \frac{4}{3}\left[n^{\log_2\left(\frac{7}{4}\right)+2} - n^2\right] = c \cdot n^{\log_2 7} + \frac{4}{3}\left[n^{\log_2 7}\right] - \frac{4}{3}n^2$

$= \Theta\left(n^{\log_2 7}\right) \approx \Theta\left(n^{2.81}\right)$

② $T(n) = \begin{cases} 2T(n/2) + n\log_2 n & n>1 \\ c & n=1 \end{cases}$

$T(n) = 2T(n/2) + n\log_2 n$

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

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

$= 2^2 T\left(\frac{n}{2^2}\right) + \frac{n}{2}\log_2 n - \frac{n}{2} + n\log_2 n = 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\log_2\left(\frac{n}{2^2}\right)\right] + n\log_2\left(\frac{n}{2}\right)$

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

$= 2^3 T\left(\frac{n}{2^3}\right) + n\log_2\left(\frac{n}{2^2}\right) + n\log_2\left(\frac{n}{2}\right) + n\log_2 n$

$\vdots$ k times

$= 2^k T\left(\frac{n}{2^k}\right) + n\left[\log_2\left(\frac{n}{2^{k-1}}\right) + \ldots + n\log_2\left(\frac{n}{2}\right) + n\log_2 n\right]$ (1 + 2 + 3 + ...)

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

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

$\log_2\frac{n}{2^{k-2}} = \log_2\frac{4 \cdot n}{2^k} = 2$

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

$T(n) = c \cdot n + n\left[1 + 2 + 3 + \ldots + \log_2 n\right]$

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

③ $T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + \frac{n}{\log_2 n} & ; n > 1 \\ c & ; n = 1 \end{cases}$ $\Theta(n \log\log n)$

④ $T(n) = \begin{cases} \sqrt{n}\, T(\sqrt{n}) + n & ; n > 2 \\ c & ; n \leq 2 \end{cases}$

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

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

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

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

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

$\vdots$

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

$T(n) = 8 n^{1/2} T(n^{1/2}) + n$

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

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

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

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

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

$T(n) = c \cdot \frac{n}{2} + n \log\log n = \Theta(n \log\log n)$

Master's Theorem:

$T(n) = aT\left(\frac{n}{b}\right) + f(n)$ [a & b const.]

Case I: if $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ [$\epsilon > 0$ real no.]

then, $T(n) = \Theta\left(n^{\log_b a}\right)$

Case II: if $f(n) = \Theta\left(n^{\log_b a}\right)$

then, $T(n) = \Theta\left(n^{\log_b a} \cdot \log_b n\right)$

Case III: if $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ [$\epsilon > 0$ real no.]

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

E.g. $T(n) = 7T\left(\frac{n}{2}\right) + n^2$

Case I. $f(n) = n^2 = O\left(n^{\log_2 7 - \epsilon}\right)$

$n^2 \leq n^{2.81 - \epsilon}$ $\Rightarrow$ $T(n) = \Theta\left(n^{\log_2 7}\right)$

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

Case I. $n^2 = O\left(n^{\log_2 4 - \epsilon}\right)$

$n^2 \leq O(n^{2-\epsilon})$ ✗

Case II. $n^2 = \Theta\left(n^{\log_2 4}\right)$

$T(n) = \Theta(n^2 \log_2 n)$

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

Case I. $n^2 = O\left(n^{\log_2 3 - \epsilon}\right)$

$n^2 \leq n^{1.69}$ ✗

Case 2: $n^2 = \Theta\left(n^{\log_2 3}\right)$

$n^2 = n^{1.69}$ ✗

Case 3: $n^2 = \Omega\left(n^{\log_2 3 + \epsilon}\right)$

$n^2 \geq n^{1.69 + \epsilon}$ ✓

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

① $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \text{const.} \Leftrightarrow f(n) \& g(n)$ asymp. equal.

② $\frac{f(n)}{g(n)} = \text{const. or log fun.} \Leftrightarrow f(n) \& g(n)$ are polynomially equal

	f(n)	g(n)	Asy comp.	Poly. comp.
①	$2n^2$	n^2	$f(n)=g(n)$	$f(n)=g(n)$
②	n^2	$n^2 \cdot \log n$	$f(n)<g(n)$	$f(n)=g(n)$
③	n^2	$n^2/\log n$	$f(n)>g(n)$	$f(n)=g(n)$
④	n^2	$n^{2.1}$	$f(n)<g(n)$	$f(n)<g(n)$
⑤	$n^2(\log n)^{10}$	$n^{2.5}$	$f(n)<g(n)$	$f(n)<g(n)$
⑥	n^2	g^n	$f(n)<g(n)$	$f(n)<g(n)$

$T(n) = aT(n/b) + f(n)$

Case I: if $n^{\log_b a}$ polynomially bigger than $f(n)$
then, $T(n) = \Theta(n^{\log_b a})$

Case II: if $n^{\log_b a}$ and $f(n)$ asymptotically equal.
then, $T(n) = \Theta(n^{\log_b a} \cdot (\log_b n))$ depth of recursion

Case III: if ~~log~~ $f(n)$ polynomially bigger than $n^{\log_b a}$
then, $T(n) = \Theta(f(n))$

E.g. $T(n) = 3T(n/2) + n^2$
$f(n) = n^2 \quad n^{\log_2 3} = n^{1.69}$
$\Theta(n^2)$.

$T(n) = aT(n/b) + f(n)$	$n^{\log_b a}$	$f(n)$	$\Theta(n)$
① $T(n) = 2T(n/2) + \sqrt{n}$	$n^{\log_2 2} = n^1$	$n^{1/2}$	$\Theta(n)$
② $T(n) = T(n/2) + \sqrt{n}$	$n^{\log_2 1} = 1$	$n^{1/2}$	$\Theta(\sqrt{n})$
③ $T(n) = 26T(n/3) + n^3$	$n^{\log_3 26}$	n^3	
④ $T(n) = 27T(n/3) + n^3$	$n^{\log_3 27} = n^3$	n^3	$\Theta(n^3 \cdot \log_3 n)$
⑤ $T(n) = T(n/3) + c$	$n^{\log_3 1} = 1$	c	$\Theta(\log_3 n)$
⑥ $T(n) = 4T(n/2) + n \log n$	$n^{\log_2 4} = n^2$	$n \log n$	$\Theta(n^2)$
⑦ $T(n) = 4T(n/2) + \log n$	$n^{\log_2 4} = n^2$	$n^3 \log n$	$\Theta(n^3 \log n)$

Q. $T(n) = 2T(n/2) + n\log n$
$f(n) = n\log n$; $n^{\log_b a} = n$ — $f(n)$ & $n^{\log_b a}$ asymptotically equal.
polynomially equal but not
Modified ✓ $T(n) = n(\log n)^2$
[Master Theorem failed]

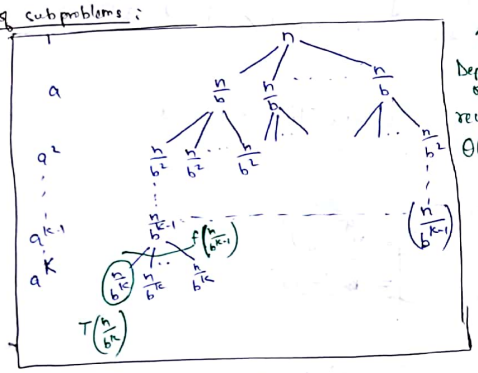
Q. $T(n) = 2T(n/2) + \frac{n}{\log n}$
$T(n) = 4T(n/2) + n^2 \log\log n$
$T(n) = 4T(n/2) + \frac{n^2}{(\log n)^{10}}$
$T(n) = 16T(n/2) + n^4(\log n)^{10}$ $= n^4(\log n)^{11}$
$T(n) = T(n/2) + \log n$ $= n(\log n)^2$
Master Theorem failed

* Modified Master's Theorem:
Case 2: $f(n) = \Theta(n^{\log_b a} \cdot (\log n)^k)$ $k \geq 0$ constant
$T(n) = \Theta(n^{\log_b a} \cdot (\log n)^{k+1})$

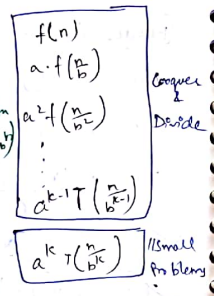
Recursive Tree Method: 26/08/2017

$T(n) = aT(n/b) + f(n)$

# of subproblems :



D & C Time



$T(n) = a^k T(n/b^k) + \sum_{i=0}^{k-1} a^i f(n/b^i)$

$\left[\because \frac{n}{b^k} = 1 \Rightarrow n = b^k \Rightarrow k = \log_b n\right]$

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

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

E.g. $T(n) = 8T(n/2) + n^2 \quad a=8; b=2; f(n)=n^2$

$T(n) = 8^k T\left(\frac{n}{2^k}\right) + n^2\left[1 + 2 + 2^2 + \dots + 2^{k-1}\right]$

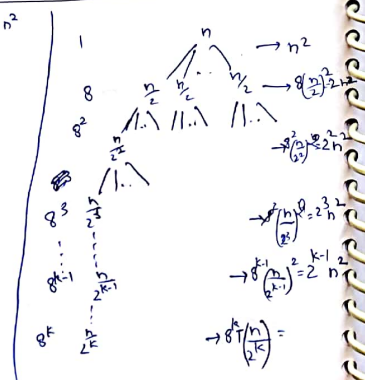
$\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n$

$2^k - 1$

$= 8^{\log_2 n} T(1) + n^2[n-1]$

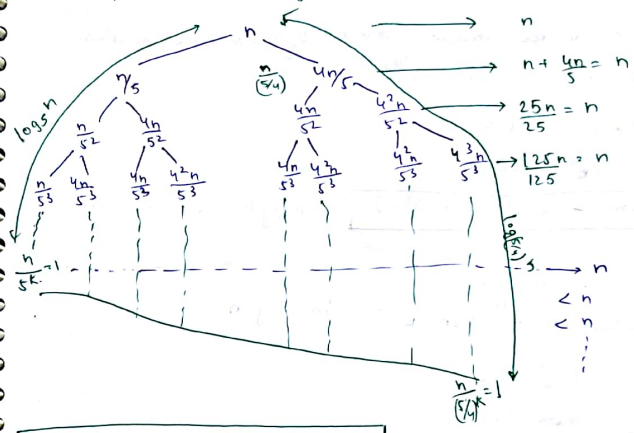
$= n^{\log_2 8} \cdot c + n^3 - n^2$

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



E.g. $T(n) = \begin{cases} T(n/5) + T\left(\frac{4n}{5}\right) + n & n>1 \\ a & n=1 \end{cases}$

Too complex to solve using substitution method.



$T(n) = \left(\log_{(5/4)} n\right) \cdot n = \Theta(n \log n)$

TRY THIS :

Q. $T(n) = \begin{cases} T(n/5) + T\left(\frac{3n}{4}\right) + n & ; n>1 \\ 1 & n \le 1 \end{cases}$

$T(n) = n\left[1 + \frac{19}{20} + \left(\frac{19}{20}\right)^2 + \dots + \left(\frac{19}{20}\right)^k\right]$

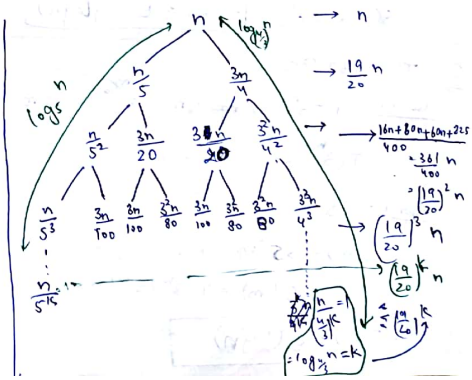
$= n\left[\frac{1 - \left(\frac{19}{20}\right)^{k+1}}{1 - \frac{19}{20}}\right]$

$= 20n\left[1 - \left(\frac{19}{20}\right)^{\log_{5/3} n + 1}\right]$

$= 20\left[n - n\left(\frac{19}{20}\right)^{\log_{5/3} n + 1}\right]$ (negligible decrement)

$\approx 20n$

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



Change of Variables:

If recurrence relation

$$T(n) = aT(\sqrt[b]{n}) + f(n)$$

Convert into master theorem format of recurrence relation.

Assume: $n = b^m \longrightarrow \sqrt[b]{n} = b^{m/b}$

$$T(b^m) = aT(b^{m/b}) + g(m)$$

Assume: $T(b^m) = S(m)$

$$\boxed{S(m) = aS(m/b) + g(m)}$$. Apply Master's theorem.

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

Assume: $n = 2^m \Rightarrow \sqrt{n} = 2^{m/2} \Rightarrow m = \log_2 n$

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

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

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

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

$$S(m) = \Theta(\log_2 m)$$

$$S(m) = T(2^m) = \boxed{T(n) = \Theta(\log\log n)}$$

Q. $T(n) = 3T(\sqrt[3]{n}) + c$

Assume: $n = 3^m \Rightarrow \sqrt[3]{n} = 3^{m/3} \Rightarrow m = \log_3 n$

$$T(3^m) = 3T(3^{m/3}) + c$$

Assume: $S(m) = T(3^m)$

$$S(m) = 3S(m/3) + c$$

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

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

Q. $T(n) = 5T(\sqrt[5]{n}) + \log_5 n$

Assume: $n = 5^m \Rightarrow \sqrt[5]{n} = 5^{m/5} \Rightarrow m = \log_5 n$

$$T(5^m) = 5T(5^{m/5}) + m$$

$$S(m) = T(5^m)$$

$$\Rightarrow 5S(m/5) + m = m \cdot \log_5 m$$

$$\boxed{T(n) = \Theta(\log_5 n \log_5 \log_5 n)}$$

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

Should use generating function method [Discrete Maths]

n^{th} Fib. Num.

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

Solve this rec. relⁿ using generating funcⁿ method.

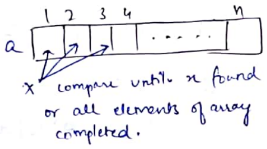
$$T(n) = 2T(n-1) + 1 \quad T(1) = 1$$
$$T(n) = 2T(n-2) + 1 \quad T(0) = 1,\ T(1) = 1,\ T(2) = 2$$
$$T(n) = T(n-1) + n \quad T(1) = 1$$
$$T(n) = 2T(n-1) + n \quad T(1) = 1$$

Using substitution.

Binary Search [DAndC Example] [Only divide no conquer]

* Linear search algo:

a[1....n] n-elements in array
x is search element



Algo Linear Search (a,n,x)
{ for (i=1; i≤n; i++)
  { if (x == a[i]) //x present ith position of array
      return (i);
  }
  return(-1) // Element x not present in array.
}

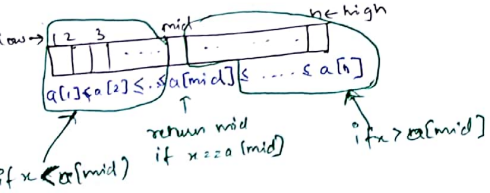
* Best case TC:
  if x present at a[1]
  No. of comparisons = 1 | TC = Θ(1)

* Worst case TC:
  if x not present in array or x present at a[n]
  No. of comparisons = n | TC = Θ(n)

* Average case TC:
  comp. = $\frac{1+2+3+\dots+n}{n} = \frac{n(n+1)}{2n} = \left(\frac{n+1}{2}\right)$ comparisons

TC = Θ(n)

* Binary search: a[1....n] must be in sorted order.



Algo BinSearch (a,n,x)
{ // a[1...n] are n sorted elements
  low = 1 ; high = n
  while (low ≤ high)
  { mid = (low+high)/2 ;
    if (x < a[mid]
      high = mid-1;
    else if (x > a[mid])
      low = mid+1;
    else
      return (mid); // element x present
  }
  return(-1);
}

TC of straight binary search:

* Best case: TC = Θ(1) //x present at middle of array.
* Worst case: TC = Θ(logn)

	low	high	[n elements for search]
1.	low=1	high = $\frac{n}{2}-1$ = mid-1	$\left[\frac{n}{2}\right]$ elements for search]
2.			$\left[\frac{n}{2^2}\right]$ elements for search]
			⋮
K			$\left[\frac{n}{2^k}\right]$ elements for search]

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

Max. $\log_2 n$ while loops to terminate algo.

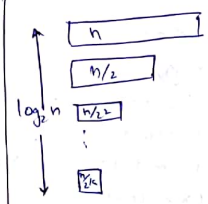
* Average Case: TC = Θ(logn)

SC of straight binary search: Θ(1)
//excluded space required for i/p array.

Recursive Binary Search :

```
Algo RBsearch (l, h)
{ // a[l...h] array elements
  if (l == h) //small (p)
  { if (x == a[l])
      return (l)
    else
      return (-1)
  }
  else
  { m = (l+h)/2
    if (x < a[m])          T(n/2)
      RBsearch (l, m-1)
    else if (x > a[m])     T(n/2)
      RBsearch (m+1, h);
    else
      return (m);
  }
}
```

Worst case TC :



WC TC Recurrence relation of B.S.

$$T(n) = \begin{cases} a & ; n = 1 \\ T(n/2) + c & ; n > 1 \end{cases} = \boxed{O(\log_2 n)}$$

Avg case : $= \Theta(\log n)$

Best case : x in middle.

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

SC. of B.S. : $= \Theta(\log n)$ //stack space.

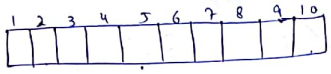
//excluded input array space.

** Because TCs of Recursive and non recursive binary searches are same, non-recursive binary search is preferred because no stack space required and no overhead of function calls.

Problems:

① If array of 10 elements in sorted order.
Average # of element comp. to search element using binary search.

(a) 2.9 (b) 3 (c) 4 (d) 2.



Avg. $= \frac{29}{10} = 2.9$

index i	no. of comp.
1	3
2	2
3.	3
4	4
5	1
6.	4
7.	3
8.	2
9.	3
10	4
Total	29

② Modify binary search algorithm such that algorithm should return position of first occurrence of searching element whose worst case TC should not exceed "log n".

1	2	3	4	5	6	7	8	9
12	12	14	14	14	14	15	15	20

```
low = 1 ; high = n
while (low ≤ high)
{ mid = (low + high) / 2
  if (x < a[mid])
    high = mid - 1
  else if (x > a[mid])
    low = mid + 1
  else if (a[mid] == a[mid-1])
    high = mid - 1;
  else
    return (mid)
}
```

WC/AC TC $= \Theta(\log n)$

③ Leading element of array is element which repeats more than $n/2$ times.

(i) What is TC to find if leading element exists or not in array of n sorted elements.

(a) $\Theta(\log n)$ (b) $\Theta(n)$ (c) $\Theta(n \log n)$ (d) $\Theta(n^2)$

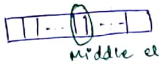
(ii) What is TC to find leading element exists or not in array of n elements whose values range [0 ... n]

(a) $\Theta(\log n)$ (b) $\Theta(n)$ (c) $\Theta(n \log n)$ (d) $\Theta(n^2)$

(iii) What is TC to find leading element exists or not in array of n elements.

(a) $\Theta(\log n)$ (b) $\Theta(n)$ (c) $\Theta(n \log n)$ (d) $\Theta(n^2)$

(i) Sorted array:

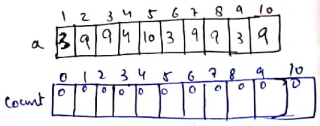


middle element must be leading element if leading element exists.

```
x = a(mid)
i = call first bin search (low, high, x)
if( a[i + n/2] == x )
    Then leading element present
else
    Then leading element not exists.
```

TC = $\Theta(\log n)$

(ii) Unsorted array of n elements.
Range of elements [0 ... n]




```
for (i = 0; i ≤ n; i++)
    count[i] = 0;
for (i = 1; i ≤ n; i++)
    count[a[i]] = count[a[i]] + 1;
for (i = 0; i ≤ n; i++)
    { if (count[i] ≥ n/2)
        then return (true)
    }
```

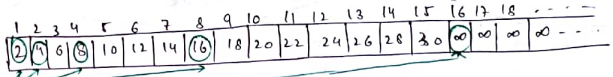
TC = $\Theta(n)$

(iii) Array of n elements:

1. sort array using best sorting algo.] $\Theta(n \log n)$
2. find leading element in sorted array] $\Theta(\log n)$

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

(iv) Assume size of array unknown [Infinite]. first few elements in sorted order remaining elements are "∞".
Write binary search to search "x" in array. or
to find last position of defined number.



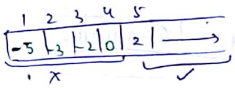

```
i = 1
while (a(i) ≠ ∞)
{ i = 2*i
}
low = i/2 ; high = i
call BS (low, high)
// for n elements
```

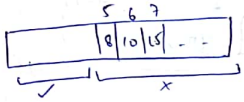
$\Theta(\log n)$ + $\Theta(\log n)$

TC = $\Theta(\log n)$

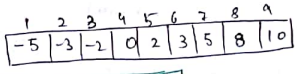
(v) Array n distinct sorted elements which can be -ve or +ve. Find any element "x" whose index "x" present in array or not?

i = a[i]	Index(x) = x





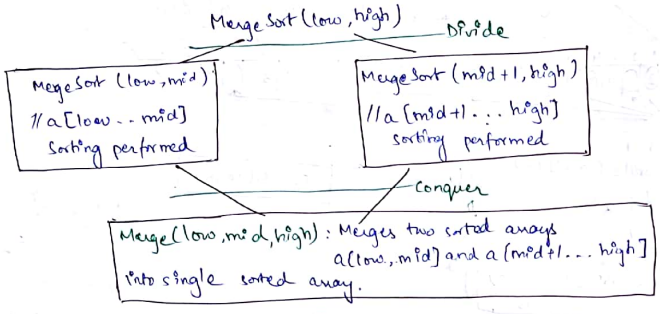
Worst case TC = $\Theta(\log n)$



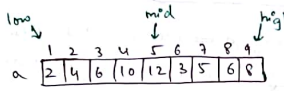

```
low = 1  high = n
while (low ≤ high)
{ mid = (low + high)/2 ;
  if (mid < a[mid])
      high = mid - 1;
  else if (mid > a[mid])
      low = mid + 1;
}
return (-1);
```

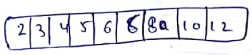
Merge Sort Algo [Fully DandC Approach]:

a[low.... high] array of n elements.




```
Algo MergeSort (low, high) → T(n)
{ // a[low...high] are n elements.
  if (low < high)
  { mid = (low + high)/2
    MergeSort (low, mid);      → T(n/2)
    MergeSort (mid+1, high);   → T(n/2)
    //conquer
    Merge (low, mid, high);    → θ(n)
  }
```

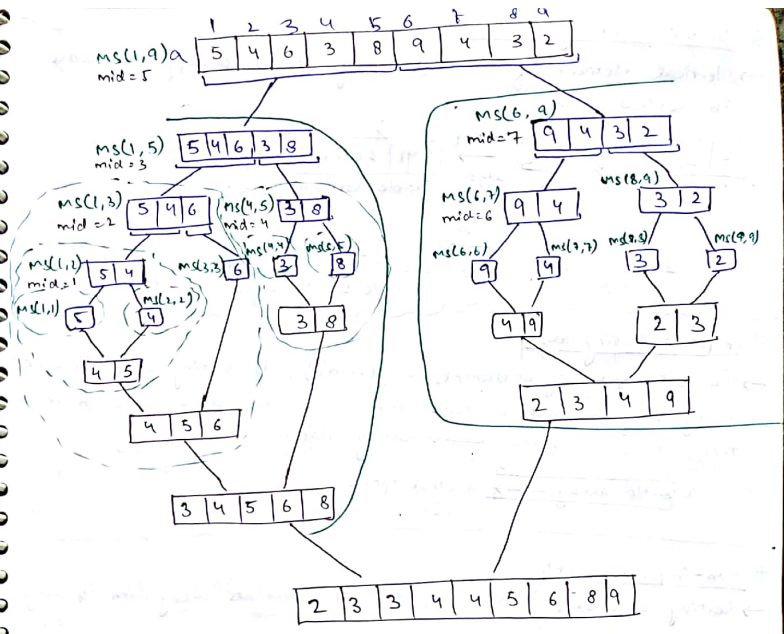


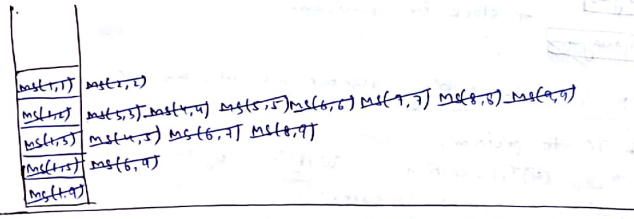


$$TC = \begin{cases} 1 & ; n=1 \\ 2T(n/2) + n.c & ; n>1 \end{cases}$$

```
Algo Merge (low, mid, high)
{ // To merge two sorted arrays a[low.. mid] and a[mid+1.. high] into single sorted array
  i=low; j=mid+1; k=low
  while (i ≤ mid && j ≤ high)
  { if (a[i] ≤ a[j])
    { b[k] = a[i]; i++; }
    else
    { b[k] = a[j]; j++; }
    k++;
  }
  if (i > mid)
    for (x = j; x ≤ high; x++)
    { b[k] = a[x]; k++; }
  else
    for (x = i; x ≤ mid; x++)
    { b[k] = a[x]; k++; }
  for (x = low; x ≤ high; x++)
  { a[x] = b[x]; }
}
```

n-1 n

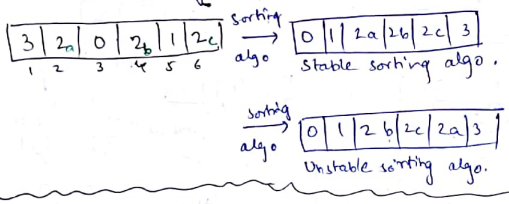




* TC to merge two sorted array of $n/2$ and $n/2$ elements each into single sorted array = $\theta(n)$
* TC of Merge sort = $\theta(n \log n)$ [All cases]
* Depth of recursion = $\theta(\log_2 n)$ [of Merge sort]
* SC of merge sort: n entries b[1... n] & [log n stacks space]
  $= \theta(n + \log n)$

Stable Sorting Algo:

→ Identical elements of unsorted array should occupy same order in sorted array.

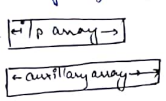


In place sorting algo:

→ To sort array of n elements {a[1...n]}; If sorting algo uses input array only, to sort elements and may use stack space for recursion then, it is in place algo. sorting algo.

Non-in place sorting:

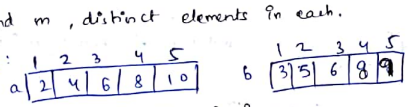
→ Sorting algo uses extra array to sort elements along with i/p array.



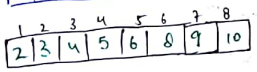
→ Merge sort is stable sorting algo but not in place sorting algo.

Q. What is TC to perform:
(i) Union (ii) Intersection (iii) Minus of two sorted arrays, n and m, distinct elements in each.

Solution:



c: a∪b



(i) Union → TC = Θ(n+m)

```
i=1  j=1  k=1
while (i≤n && j≤m)
{ if (a[i] < b[j]) {c[k] = a[i]; i++;}
  else if (a[i] > b[j]) {c[k] = b[i]; j++;}
       else
         {c[k] = a[i]; i++; j++;}
  k++;
}
if (i>n) for (x=j; x≤m; x++)
           {c[k] = b[x]; k++;}
else for (x=i; x≤n; x++)
       {c[k] = a[x]; k++;}
```

(ii) Intersection:

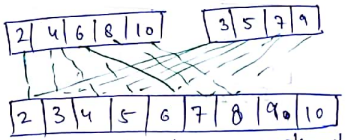
```
i=1  j=1  k=1
while (i≤n && j≤m)
{ if (a[i] < b[j]) {i++;}
  else if (a[i] > b[j]) {j++;}
       else
         {c[k] = a[i]; i++; j++;}
  k++;
}
```

TC = Θ(n+m)

(iii) Minus: TC = Θ(n+m)

Q. How many maxm element comp. required to merge two sorted arrays n & m elements each into single sorted array.

(a) n or m (b) n+m+1 (c) n+m-1 (d) n+m



Q. Given a[1...n] array of n elements. Inverse defined as two elements of array such that $i < j$ && $a[i] > a[j]$. What is TC to find no. of inverses of array of n-elements.

FATE

(a) $\Theta(n^2)$ (b) $\Theta(n \log n)$

(c) $\Theta(\log n)$ (d) $\Theta(n)$

1	2	3	4	5	6	7	8
6	5	3	8	2	9	3	1

Inv 5+4+2+3+1+2+1+0 = 18 Inverses.

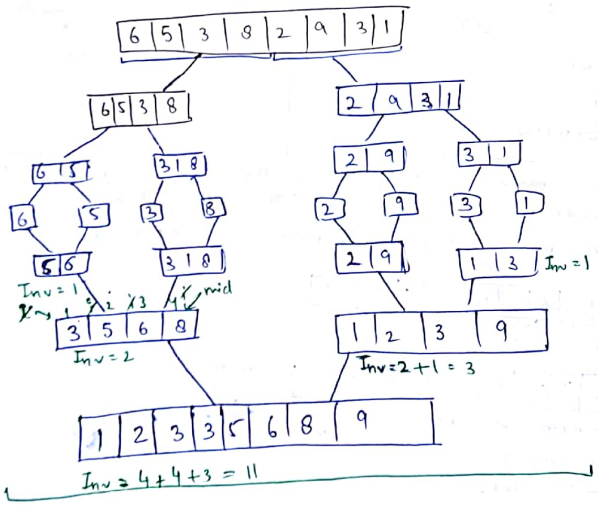
```
Inv = 0
for(i=1; i<=n-1; i++)
  { for(j=i+1; j<=n; j++)
      if (a[i] > a[j])
      { Inv++; }
  }
return(Inv);
```

TC : $\Theta(n^2)$

① Using mergesort:

```
while (i <= mid && j <= high)
{ if (a[i] <= a[j])
  { b[k] = a[i]; i++; }
  else
  { b[k] = a[j]; j++;
    Inv = Inv + (mid - i + 1);
  }
  k++;
}
```

// a[i] Inv with every element from a[i] to a[mid]



Total inverses = 18

Q. $f_0 = 1$; $k = 0$

$f_1 = 1$; $k = 1$

$f_k = f_{k-1} + f_{k-2}$; $k \geq 2$;

k =	0	1	2	3	4	5	6
f_k =	1	1	2	3	5	8	13

Algo MergeSort (f_k i/ps)

```
{ if (k > 1)
  { //Divide
    MergeSort (f_{k-1} i/ps)
    MergeSort (f_{k-2} i/ps)
    //Conquer
    Merge (F_{k-1}, F_{k-2})   cost of Merge f_k.
  }
}
```

Analyse TC of FibMergeSort?

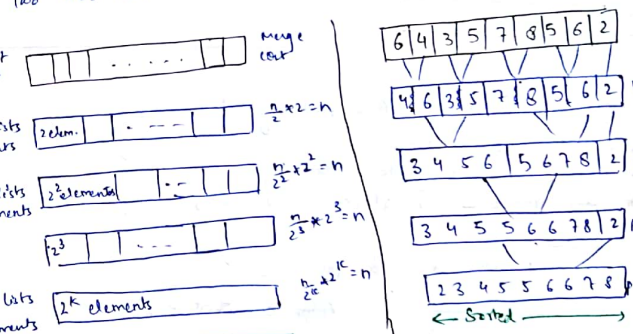
Use golden ratio = 1.618

F_k

27/08/2017

#2 way Merge Sort: [Straight Merge Sort]

→ Initially a[1...n] consider as n sorted lists each list one element. Merge two lists until 'n' sorted lists become single sorted list.



Q. Array consists $\log_2 n$ sorted lists each list $\frac{n}{\log_2 n}$ elements. What is TC to merge an array into single sorted list?

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

$= n$

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

$\log_2 n = k$

$TC = n \cdot \log_{\log_2 n} n$

Q. Array consists of n strings each string of n characters. What is TC to sort n strings in lexicographic order?

n

① $\log_2 n$ lists, $\frac{n}{\log_2 n}$ elem. | $\frac{n}{\log_2 n}$ | ... | — merge cost

$\frac{n}{\log n}$ lists, $\frac{2n}{\log n}$ elem. | $\frac{2n}{\log_2 n}$ | ... | — $\frac{\log n}{2} * \frac{2n}{\log_2 n} = n$

$\frac{\log n}{2}$ lists, $\frac{2^2 n}{\log n}$ elem. | $\frac{2^2 n}{\log_2 n}$ | ... | — $2n$

$\frac{\log n}{2^{1c}}$ lists. $\frac{\log n}{2^{1c}} = 1 \Rightarrow \log_2 n = 2^{1c} \Rightarrow k = \log_2 \log_2 n$

TC = $n \log_2 \log_2 n$

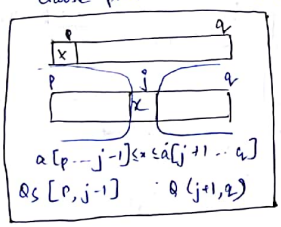
② TC = $\theta(n^2 \cdot \log n)$

Quick Sort Algo : [Also partially D and C]

* widely used sorting algo
* Inplace and not stable sorting algo
* Tony George [1970] {Turing Award 1985}

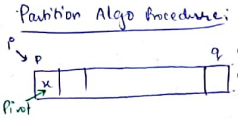
Basic Policy :

Choose pivot element and place in correct position.



Algo Quicksort (p, q)
{ // a[p ... q] array of n elements
if (p < q)
{ j = partition (a, p, q);
Quicksort (p, j-1);
Quicksort (j+1, q);
}

Partition Algo Procedure:

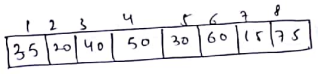


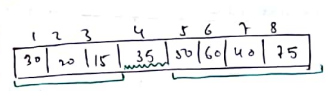
To away out of bound access if pivot element bigger than all other element.

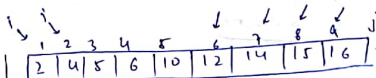
1. Increment i until a[i] ≥ x
2. Decrement j until a[j] ≤ x
3. If i < j swap (a[i], a[j])
4. Repeat ①②③ until i ≥ j
5. swap (a[p], a[j])
6. return j

Algo partition (a, p, q) $\theta(n)$

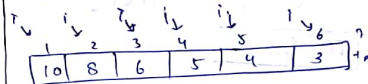
```
{ a[q+1] = +∞;
  i = p ; j = q+1 ; x = a[p];
  do
  { 1. do
       { i = i+1;
       } while (a[i] < x)
    2. do
       { j = j-1;
       } while (a[j] > x)
    3. if (i < j) swap (a[i], a[j])
  } while (i < j);
  swap (a[p], a[j])
  return j;
}
```







(a[i] < x) : 1 comp
(a[j] > x) : n comp.
n+1 comp.



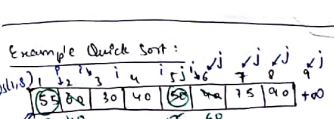
a[i] < x : n comp
a[j] > x : 1 comp
n+1 comp.

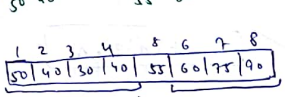
Unsorted

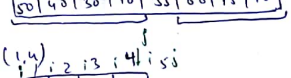
a[i] < x : n/2
a[j] > x : n/2 + 1
n+1 comp.

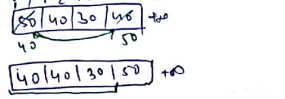
∴ TC of partition algo all cases = $\theta(n)$

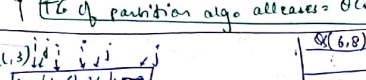
Example Quick Sort :

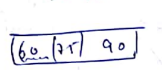


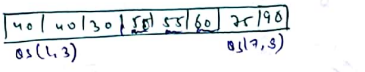






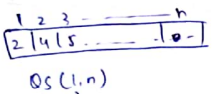






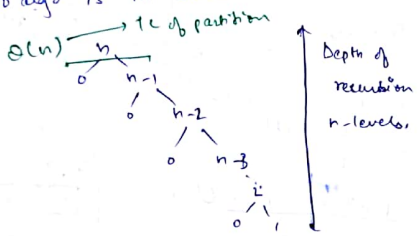
TC of QuickSort :

① Worst case TC of QS: QS behaves as worst case if given i/p array to algo is in sorted order.



QS(1,n)

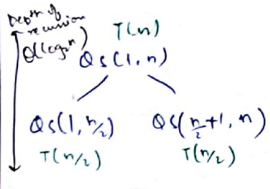
QS(1,0) — 0 elements; QS(2,n) — n-1 elements

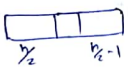


$$TC = \begin{cases} T(n-1)+n & ; n>1 \\ 1 & ; n\le 1 \end{cases}$$ Worst case TC recurrence relation.

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

② Best case TC of QuickSort: If pivot element places middle of list in every function call, then QS behaves as best case.

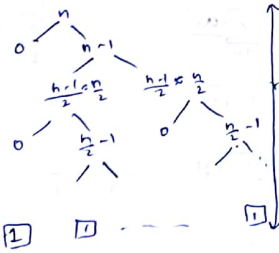




$$T(n) = \begin{cases} 2T(n/2)+n & ; n>1 \\ 1 & ; n\le 1 \end{cases}$$

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

② Average case TC of QuickSort: $\Theta(n \log n)$

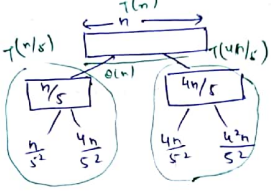


Case 1, Partition algo divides list as best case & worst case and so on...

2 log n depths of recursion

$$TC = \Theta(n \log_e n)$$

Case 2: Every time partition divides list into n/5 elements & 4n/5 elements for given n elements.



Recurrence relation:

$$T(n) = \begin{cases} T(n/5) + T(4n/5) + n & ; n>1 \\ 1 & ; n\le 1 \end{cases}$$ $T(n) = \Theta(\log_{5/4} n)$

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

Average case recurrence relation of QS:

$$T(n) = \begin{cases} T(\alpha n) + T((1-\alpha)n) + n & ; n>1 \\ 1 & ; n\le 1 \end{cases} \qquad 0<\alpha<1$$

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

$$T(n) = T\left(\frac{n}{20}\right) + T\left(\frac{19}{20}n\right) + n$$

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

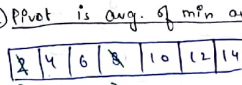
$T(n) = \Theta(n \log n)$

Pivot Selection Methods:

① First / last element of array is pivot: Worst case TC $\to \Theta(n^2)$

If i/p array is in sorted order.

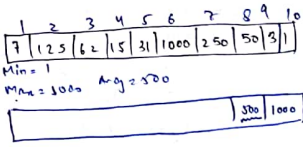
② Pivot is avg. of min and max of array:



$\Theta(n)$

```
Algo QS(p,q)
{ if (p<q)
  { k = Pivot set (a,p,q) → Θ(n)
    swap (a[k], a[p])
    j = partition (a,p,q) → Θ(n)
    QS (p, j-1)   → T(n-2)
    QS (j+1, q)   → T(1)
  }
}
```

W.C.



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

$= \Theta(n^2)$

③ Median element*

Median element of array "x" such that:

$\frac{n}{2}$ elements $\leq x \leq \frac{n}{2}$ elements

If median element of array is pivot element:
then worst case TC of QS :: $\Omega(n\log n)$ [more than $n\log n$]
↳ which depends on TC to find median.

```
Also QS(p,q)
{ if (p<q)
  { k = median(a,p,q) → Ω(n)
    swap(a[k], a[p])
    j = Partition(a,p,q) → Θ(n)
    QS(p, j-1)   → T(n/2)
    QS(j+1, q)   → T(n/2)
  }
}
```

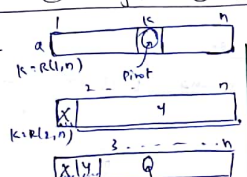
$TC = \Omega(n\log n)$ (All cases)

Q. WC TC of Quick sort using median pivot selection?

(a) $O(n^2)$ (b) $O(n)$ (c) $\Theta(n\log n)$ (d) $O(n\log n)$

---

④ Random Pivot selection:




```
Algo QS(p,q)
{ if (p<q)
  { k = Random(p,q)
    swap(a[k], a[p])
    j = partition(a,p,q)
    QS(p, j-1)
    QS(j+1, q)
  }
}
```

WC TC of QS
$= O(n^2)$
// Every time randomly chooses min^m element as pivot unknowingly.

*Probability of min/max element selected by Random Pivot selection is almost 0.

Q. what is WC TC of QS using Random Pivot selection:

(a) $O(n^2)$ (b) $O(n\log n)$ (c) $\Theta(n\log n)$ (d) $\Theta(n)$

Q. In pivot selection algo, $\Theta(n)$ time to choose which is $(\frac{n}{2})^{th}$ biggest element of array. Then what is ~~worst~~ WC TC of QS.

(a) $\Theta(n^2)$ (b) $\Theta(n\log n)$ (c) $O(n)$ (d) $\Theta(n)$

Q. If all elements of array is identical. what is TC of QS?

(a) $\Theta(n\log n)$ (b) $\Theta(n^2)$ (c) $\Theta(n)$ (d) $O(n)$

//Ans - $\Theta(n\log n)$ or $\Theta(n^2)$
Depends on implementation of partition.
Our algo - $\Theta(n\log n)$

Q. $list_1 = 1, 2, 3, \ldots n$
$list_2 = n, n-1, n-2, \ldots 1$
if c_1 & c_2 comp. required to sort ~~them~~ $list_1$ & $list_2$ in ascending order using quick sort algo. which is true?

(a) $c_1 < c_2$ (b) $c_1 > c_2$ (c) $c_1 = c_2$ (d) $c_1 = c_2 + n\log n$

Q. Write algo to find k^{th} smallest element in array of n distinct element.

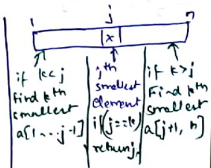
Method 1.
① Sort array using sorting algo — $\Theta(n\log n)$ All cases
② return a[k]

Method 2.
① Find pos of min from a[1] to a[n] & swap with a[1]
② find pos of min(a[2] to a[n]) swap with a[2]
:
k times

TC = $\Theta(k.n)$
BC = $\Theta(n)$
WC ⇒ $\Theta(n^2)$
AC ⇒ $\Theta(n^2)$

method 3.
Best solution: using partition algo.

```
Algo k^th smallest(p,q,k)
{ if (p==q)
  { if(k==p)
      return(a[k])
    else
      return(false)
  }
else
{ j = partition(a,p,q)   Θ(n)
  if (k<j)
    k^th smallest(p,j-1,k)
  else if(k>j)
    k^th smallest(j+1,q,k)
  else return(a[j])
}
```



Best case TC:

$\Theta(n)$

One function call to terminate algo if $j == k$ after one partition.

Average Case TC:

$T(n) = T(n/5) + n \cdot c$

$T(n) = T(\frac{4n}{5}) + n \cdot c$ $\Theta(n)$

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

Assume '$\alpha \cdot n$', '$(1-\alpha)n$'

$0 < \alpha < 1$ is partition sizes

Worst Case TC:

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

$= \Theta(n^2)$

* Every time partition algo divides c & $(n-c)$ for i/p array.
* k^{th} smallest element at n.

Max-Min Algo:

→ a[1...n] array of n elements. find max & min element of array.

Straight Max Min Algo:

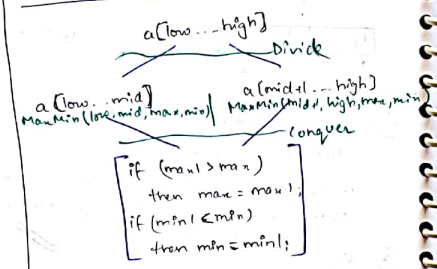
```
Algo MaxMin(a,n)
{ Max = min = a[1];
  for (i=2; i<=n; i++)
  { if (a[i] > max)
      max = a[i];
    else if (a[i] < min)
      min = a[i];
  }
  return (max, min);
}
```

	1	2	3	4	5
a	4	5	2	3	6

* Min comp. required to find Max & Min = $(n-1)$ comp.
* Max comp. required to find Max & Min = $2(n-1) = 2n-2$ comp.

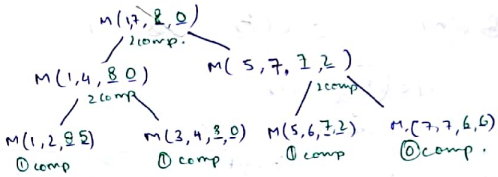
$\frac{(n-1)}{2} * 1 + \frac{(n-1)}{2} * 2 = \frac{3}{2}n - \frac{3}{2}$

D And C Max-Min Algo:




```
Algo Max-Min(low, high, max, min)
{
  If (low == high)                          n=1
    max = min = a[low];
  else if (low == high-1)                   n=2
    { if (a[low] ≤ a[high])
      { max = a[high];
        min = a[low];
      }
      else
      { max = a[low]
        min = a[high]
      }
    }
  else                                      n>2
    { mid = (low + high)/2
      Max-Min(low, mid, max, min)
      Max-Min(mid+1, high, max1, min1)
      If (max1 > max) max = max1;
      If (min1 < min) min = min1;
    }
  return (max, min)
}
```

	1	2	3	4	5	6	7
a	8	5	3	0	2	7	6



$T(n)$: # of comparisons to find max-min using D And C.

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

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

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

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

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

$\vdots$

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

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

$n = 2^{k+1} \cdot 2 \Rightarrow 2^k = \frac{n}{2}$, $\therefore k = \log_2(\frac{n}{2})$

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

$= \frac{n}{2} \cdot 1 + n - 2 = \boxed{\frac{3n}{2} - 2 = T(n)}$

Max-Min Algo	Min Comp	Avg. Comp.	Max Comp.
Straight	$(n-1)$	$\frac{3}{2}n - 1.5$	$2n-2$
D & C	$\frac{3}{2}n - 2$	$\frac{3n}{2} - 2$	$\frac{3n}{2} - 2$

Q. Min element comp. to find max-min of array of n elements.

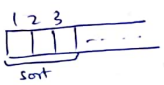
(a) $n-1$ (b) $2n-2$ (c) $\frac{3n}{2} - 2$ (d) none

Q. Max element comp. to find Max and Min of array of n elements.

(a) $n-1$ (b) $2n-2$ (c) $\frac{3}{2}n - 2$ (d) none

Q. What is TC reqd. to return element from array of n distinct elements, which is neither max nor min?

(a) $\Theta(\text{sign})$ (b) $\Theta(n)$ (c) $\Theta(n^2)$ (d) $\Theta(1)$



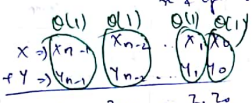
$a[1] < a[2] < a[3]$

$a[2]$ → neither max nor min

Integer Multiplication Algo:

(a) Straight integer multiplication
(b) D & C integer multiplication
(c) Strassen's D&C int mult.

* Integer Addition: x, y are two n bit integers. TC required to add x & y : $\boxed{O(n)}$ [no. of bit operation]



(a) Straight integer multiplication:

X = 25 Y = 5 Z = 0

① if x is odd, then z = z+y = 5
x = ⌊x/2⌋ = 12 y = 2y = 10

② if x is even then no add
x = ⌊x/2⌋ = 6 y = 2y = 20

③ if x is even, then no add
x = ⌊x/2⌋ = 3 y = 2y = 40

④ if x is odd, z = z+y = 45
x = ⌊x/2⌋ = 1 y = 2y = 80

⑤ If x is odd, z = z+y = 125
x = ⌊x/2⌋ = 0 y = 2y = 160

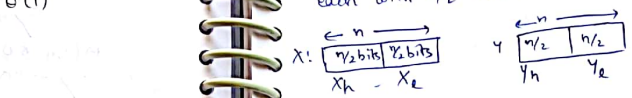
```
Algo IntMul(X,Y)
{ //X,Y two n-bit integers
  Z = 0
  while (X>0)          O(1)
  {                    [log2 X] = n = # of bit for X
    if ((X%2) == 1)
      { Z = Z+Y; } ← O(n)
    X = ⌊X/2⌋; //Right shift O(1)
    Y = 2Y ; //Left shift O(1)
  }
  return (Z);
}
```

# of bits required to represent int x is $\lceil \log_2 x \rceil$ bits.

$\boxed{TC = \Theta(n^2)}$ # of bit operations.

D&C int multiplication:

→ X, Y are n bit numbers, if n>1 divide x & y into two sub Int each with n/2 bits.



$X \cdot Y = (X_h \cdot 2^{n/2} + X_l)(Y_h \cdot 2^{n/2} + Y_l)$

$X \cdot Y = X_h * Y_h \cdot 2^{n} + (X_h * Y_l + X_l * Y_h) \cdot 2^{n/2} + X_l * Y_l$

(2^n: n left shift; $2^{n/2}$: n/2 left shift)

$(011|010)_2$ = 26
$X_h = (011)_2 = 3$ $X_l = 010 = 2$

$X = X_h \times 2^3 + X_l$
$= 3 \times 2^3 + 2$
$= 24 + 2 = 26$

To multiply two n bit int. ~~x & y~~ X & Y.

- 4 subInt Mul of n/2 bits
- 3n/2 left shift opr.
- 3 int additions

Algo IntMul(X, Y, n) ⇒ T(n)

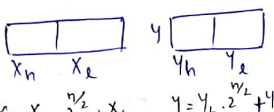
```
{ if (n≤1) //small(P)        } a
    return (X*Y)
  else
  { //Divide
    Xh Xl are two sub int of X
    Yh Yl are two sub int of Y
    P = IntMul(Xh, Yh, n/2)   }
    Q = IntMul(               } 4T(n/2)
    R = IntMul(               }
    S = IntMul(Xl, Yl, n/2)   }
    //Conquer
    return (P.2^n + (Q+R)2^(n/2) + S) ⇒ c·n
```

D & C recurrence relations:

$$T(n) = \begin{cases} 4T(n/2) + c \cdot n & ; n > 1 \\ a & n \le 1 \end{cases}$$

$= O(n^2)$

Strassen's D&C:



$X = X_h \cdot 2^{n/2} + X_l \qquad Y = Y_h \cdot 2^{n/2} + Y_l$

$P = X_h * Y_h$

$Q = X_l * Y_l$

$R = (X_h + X_l) * (Y_h + Y_l)$

$X.Y = P.2^n + [R - P - Q].2^{n/2} + Q$

To multiply X & Y of n bits each: $T(n)$

- 3 sub int mult. of $n/2$ bits each $\rightarrow 3.T(n/2)$
- $\frac{3n}{2}$ left shifts $\rightarrow O(n)$
- 6 int add/sub $\rightarrow O(n)$

Strassen's Int Mult. D&C Recurrence relation:

$$T(n) = \begin{cases} 3T(n/2) + c.n & ; n > 1 \\ a & ; n \le 1 \end{cases}$$

$$= \Theta(n^{\log_2 3})$$

$$= \Theta(n^{1.59})$$

Straight Matrix Multiplication:

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ a_{21} & & \\ \vdots & & \\ a_{n1} & \cdots & a_{nn} \end{bmatrix}_{n \times n} * \begin{bmatrix} b_{11} & \cdots & b_{1n} \\ b_{21} & & \\ \vdots & & \\ b_{n1} & \cdots & b_{nn} \end{bmatrix}_{n \times n} = \begin{bmatrix} c_{11} & c_{21} & \cdots & c_{1n} \\ c_{21} & & & \\ \vdots & & & \\ c_{n1} & \cdots & & c_{nn} \end{bmatrix}_{p \times r}$$

$p \times q \qquad q \times r$

$c_{11} = a_{11} * b_{11} + a_{12} * b_{21} + \cdots + a_{1n} * b_{n1}$

n element multiplications for each element of ~~matrix~~ result matrix.

```
for (i=1; i<=p; i++) -> p
{ for (j=1; j<=r; j++) -> r
  { c[i,j] = 0
    for (k=1; k<=q; k++) -> q
      c[i,j] = c[i,j] + a[i,k] * b[k,j]
  }
}
```

$T(n) = \Theta(n^3)$ or $\Theta(p.q.r)$

Q. $A_{p \times q}$ $B_{q \times r}$ are two matrices. How many element multiplication reqd. to multiply matrices A & B? $p.q.r$ [No of element mul.]

(cost of matrix multiplication.

Matrix Multiplication Algo:

(a) Straight Matrix Multiplication

(b) D&C Matrix Multiplication

(c) Strassen's D&C Matrix multiplication

* Matrix Addition:

$C_{n \times n} = A_{n \times n} + B_{n \times n}$

$O(n^2)$

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    c[i,j] = A[i,j] + B[i,j];
```

$$\begin{bmatrix} c_{11} & c_{12} & \cdots \\ c_{21} & & \\ \vdots & & \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots \\ a_{21} & & \\ \vdots & & \end{bmatrix}_{n \times n} + \begin{bmatrix} b_{11} & b_{12} & \cdots \\ b_{21} & & \\ \vdots & & \end{bmatrix}_{n \times n}$$

D&C Matrix Multiplication:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}_{n \times n} \quad \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}_{n \times n} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}_{n \times n}$$

(A_{11}, B_{11}, C_{11}: $\frac{n}{2} \times \frac{n}{2}$)

→ if $n > 2$ divide each matrix into 4 submatrices with $\frac{n}{2} \times \frac{n}{2}$ order.

$C_{11} = A_{11} * B_{11} + A_{12} * B_{21}$

$C_{12} = A_{11} * B_{12} + A_{12} * B_{22}$

$C_{21} = A_{21} * B_{11} + A_{22} * B_{21}$

$C_{22} = A_{21} * B_{12} + A_{22} * B_{22}$

```
Algo D&C(A,B,n)
{
  if (n ≤ 2)
    return (A·B)                    ] a
  else
    // Divide
    Divide A & B into 4 sub matrices each
    p = D&C(A11, B11, n/2)
    q = D&C(A12, B21, n/2)
    r = D&C(A11, B12, n/2)
    s = D&C(A12, B22, n/2)          ] 8T(n/2)
    t = D&C(A21, B11, n/2)          ] + 4n^2
    u = D&C(A22, B21, n/2)
    v = D&C(A21, B12, n/2)
    w = D&C(A22, B22, n/2)
    C11 = p+q
    C12 = r+s
    C21 = t+u
    C22 = v+w
    return [C11 C12; C21 C22]
}
```

* D&C Matrix mul. Recurrence relation

$$T(n) = \begin{cases} 8T(n/2) + n^2 & ; n > 2 \\ a & ; n \le 2 \end{cases}$$

$TC = \Theta(n^3)$

Strassen's Matrix Multiplication: (Most effficient algo)

→ Divide each sub matrix into 4 sub matrices:

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

* To multiply matrices of $n \times n$
  - 7 submatrix mult. of $n/2 \times n/2$
  - 18 matrix add/sub

$$T(n) = \begin{cases} 7T(n/2) + c \cdot n & ; n > 2 \\ a & ; n \le 2 \end{cases}$$

$= \Theta(n^{\log_2 7})$

$T(n) = \Theta(n^{2.8})$

$p = (A_{11} + B_{22}) \cdot (B_{11} + B_{22})$
$q = (A_{21} + B_{22}) \cdot B_{11}$
$r = A_{11} \cdot (A_{12} - A_{22})$
$s = A_{22} \cdot (B_{21} - B_{11})$
$t = (A_{11} + A_{12}) \cdot B_{22}$
$u = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$
$v = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$

$C_{11} = p + s + t + v$ | $C_{21} = q + s$
$C_{12} = r + t$ | $C_{22} = p + r - q + u$

25/08/2017

- → Graph and Tree representation
- → Priority Queue [Min heap / Max heap]
- → Set algorithms [Union/Find Algo]

] DS Topic

Tree

- → Single rooted (Starting point). Every tree operation starts from root.
- → Always connected.
- → Always acyclic.
- → Always directed.

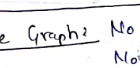
[Tree with n vertices must be exactly (n-1) edges]

Graph

- → Any vertex can be used as starting point.
- → May / may not connected.
- → May / may not acyclic.
- → May / may not directed.

[Simple graph with n vertices can be
- atleast 0 Edge [Null graph].
- atmost $\frac{n(n-1)}{2}$ Edges [complete connected graph]]

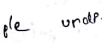
* Simple Graph: No self loops. No parallel edges.



] Not simple graph.

Q. How many simple undirected graph possible with n vertices?

$n = 4$ ~~$n(n-1)/2$~~ 6
$e = 6$



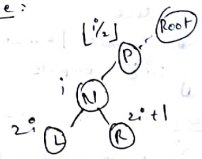
$= {}^6C_0 + {}^6C_1 + {}^6C_2 + {}^6C_3 + \dots + {}^6C_6 = 2^6$

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

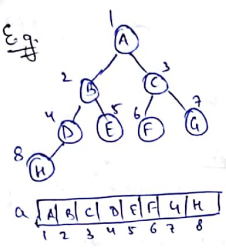
Tree Representation: (a) Using array (b) Using linked list

* Array representation to store binary tree:

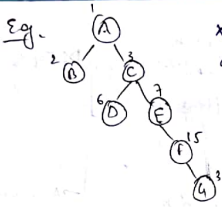
Root node index = 1
If Node N array index i
then left child of N is $2i$ index
right child of N is $(2i+1)$ index
parent of N is $\lfloor i/2 \rfloor$ index

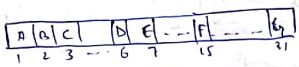


Eg.



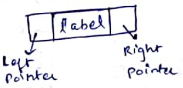
Eg.

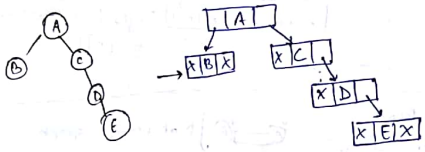


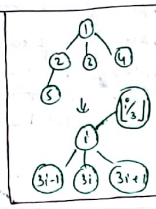


* Array rep. not prefered due to wastage of m/m space.

* Linked List representation:







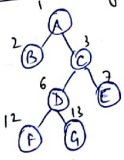
→ To store binary tree of n nodes: 2n pointers
[n+1 null pointers]
& [n-1 non-null pointers]

* If node indexes are continuous integer from 1 to n, then array representation preferred. [No overhead of pointers].

Binary Trees:

① Strict binary tree
② Complete binary tree | Almost complete → In some books
③ Full binary tree | Complete

* Strict binary tree: Every node of tree must have 0 or 2 children.

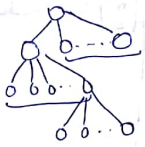


* Linked list list representation prefered to store strict binary tree.

Q. Complete k-ary, every internal node exactly k child nodes. How many # of leaf nodes with n internal nodes.

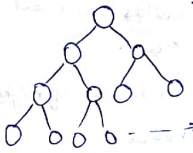
(a) nk (b) n(k-1)+1 (c) (n-1)k+1 (d) n(k-1)

0 → 1
1 → 2
2 → 2k-1
3 → 3k-2



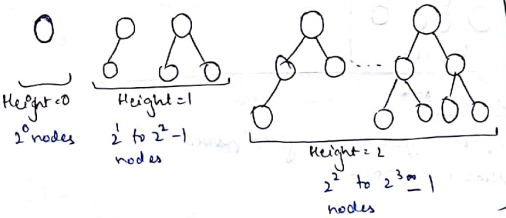
* Complete Binary Tree:

→ Complete binary tree of h levels must be (h-1) levels fully occupied.
→ hth level nodes can add left to right.



h-1 levels fully occupied.

hth level nodes filled from left to right

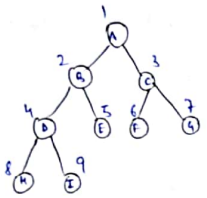


Height=0 → 2^0 nodes
Height=1 → 2^1 to 2^2-1 nodes
Height=2 → 2^2 to 2^3-1 nodes

Complete binary tree of height "h": (2^h to 2^(h+1)-1) nodes.

* Array representation preferred to store complete binary tree & FBT

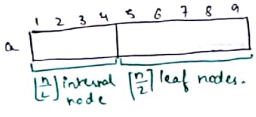
→ Complete binary tree of "n" nodes: ⌊n/2⌋ # of internal nodes
or
FBT
⌈n/2⌉ # of leaf nodes.



$n = 9$

$\lfloor \frac{n}{2} \rfloor$ internal nodes $= \lfloor \frac{9}{2} \rfloor = 4$

$\lceil \frac{n}{2} \rceil$ leaf nodes $= \lceil \frac{9}{2} \rceil = 5$



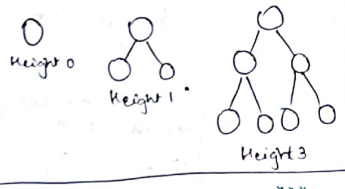
* Height of CBT for "n" nodes: $n = 2^h \Rightarrow h = \log_2 n$ (min nodes)

or FBT

$n = 2^{h+1} - 1 \Rightarrow h = \log_2(n+1) - 1$ (max nodes)

$\Theta(\log_2 n)$

* Full Binary Tree: Every nodes 0 & 2 childs and all leaf nodes at the same level.



No. of nodes in FBT of height "h".

$(2^{h+1} - 1)$ nodes.

Priority Queue Data Structure: ***

→ DS which should support efficient way to perform:

- Repeated min/max element deletion
- Repeated Insertions
- May required to store duplicate element.

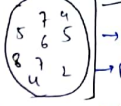
Priority queue use for process scheduling

Max val: high priority

→ Repeated max deletions

→ Repeated insertion

→ Able to store duplicate values



Min val: high priority

→ Repeated min deletions

→ Repeated insertions

→ Support to store duplicate values.

Q. Algo

(1) Build list for n elements [Assume elements should insert one after and may not be distinct]

(2) Repeat Min del. over list n times

	Sorted array // Descending	Unsorted array	Sorted Linked list // Asc.	Unsorted Linked list	Balanced binary Search Tree	Min heap. * Best DS to implement priority queue
(1)	$1+2+3+\dots+n$ comp & shift $= \Theta(n^2)$	$n \times \Theta(1) = \Theta(n)$	$1+2+3+\dots+(n-1)$ comp. $= \Theta(n^2)$	$n \times \Theta(1) = \Theta(n)$ insert * DS not suitable for duplicate elements. * Uses linked list.	$\Theta(n \log n)$	$n \times \Theta(\log n) = \Theta(n \log n)$ * Used to maintain duplicate value * No overhead of pointers [array based]
(2)	$n \times \Theta(1) = \Theta(n)$	$(n-1)+(n-2)+\dots+2+1 = \Theta(n^2)$	$n \times \Theta(1) = \Theta(n)$	comp: $n+n-1+\dots+1 = \Theta(n^2)$	$\Theta(n \log n)$	$n \times \{\Theta(\log n) = \Theta(n \log n)$
T(n)	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n \log n)$ *	$\Theta(n \log n)$

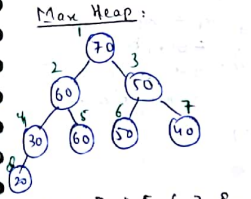
Note:

* Balanced BST best DS for: (1) searching element.
(2) Insert / delete
(3) discard duplicate elements.

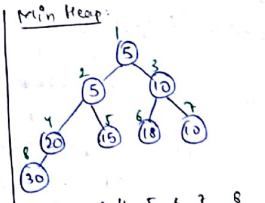
Max (Min) Heap:

→ Complete binary tree with every parent must be max (min) than all left and right subtree nodes.

Max Heap:

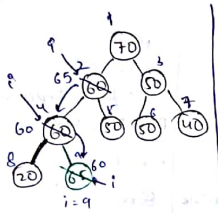


Min Heap:



* Insertion into Maxheap: a[1...n-1] array elements already in max heap.

Insert element i into Max heap.

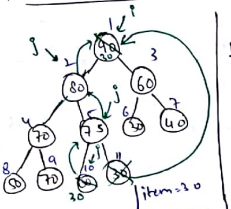



```
Algo InsertMaxheap(a,n,x)
{ i=n;                          parent
  item=x;                       ↑
  while (i>1 && a[i/2]<item)
  { a[i]=a[i/2];
    i=i/2;
  }
  a[i]=item;
}
```

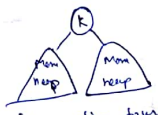
→ Comp. and shift operations insert element into max heap: $\theta(\log n)$ ($\log n \geq$ height)

→ TC to insert element into max heap: $\theta(\log n)$ in worst & avg case.
$\theta(1)$, in best case.

* Delete Max from Max heap: $\theta(\log n)$



- maxele = a[1]
- a[1] = a[n]
- Adjust (a, i, n-1)



[Assumption to use adjust fn]

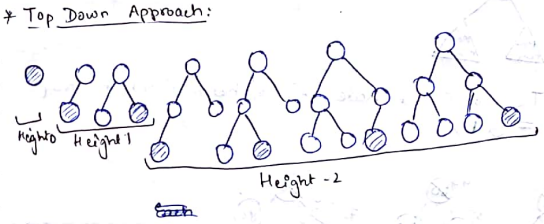
```
Algo DelMaxHeap(a,n) ⇒ TC to delete max from heap of n elements: θ(log n) → WC & Avg C
{ MaxEle = a[1];                                                  θ(1) → BC
  a[i]=a[n];
  Adjust(a,1,n-1)
  return(MaxEle)
}
Algo Adjust(a,i,n) ⇐ TC = θ(log2 n) WC & Avg case.
{ j=2*i; item=a[i];                                  θ(1) in BC.
  while(j≤n)                  # of comp & shift operations:
  { if(j<n && a[j]<a[j+1])      = θ(h) time.
    { j=j+1; }                  adjust of element "x" at height "h".
    if(item≥a[j]){break;}
    a[j/2]=a[j];
    j=2*j;
  }
  a[j/2]=item;
}
```

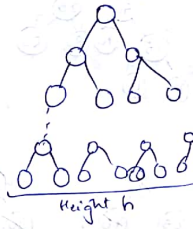
* Building of heap: 29log12017:

① Top Down approach: [To build max heap of elements [keys] should enter one after another] $\theta(n \log n)$

② Bottom Up approach: [To build max heap if all elements are available] $\theta(n)$

* Top Down Approach:





* Each insertion of height h [h comp. & h shift]

Height	No. of nodes	Cost per insertion [comp. & shift]
0	1	0
1	2	1
2	2^2	2
3	2^3	3
⋮	⋮	⋮
h	2^h	h

Cost of building max heap in top down approach.

$T(h) = 2*1 + 2^2*2 + 2^3*3 + \cdots + 2^h*h$

$-2T(h) = 2^2*1 + 2^3*2 + 2^4*3 + \cdots + 2^{h+1}*h$

$-T(h) = 2*1 + 2^2*1 + 2^3*3 + \cdots + 2^h*h$

$-T(h) = 2*1 + 2^2 + 2^3 + 2^4 + \cdots + 2^h + 2^{h+1} - h$

$-T(h) = (2 + 2^2 + 2^3 + 2^4 + \cdots + 2^h) + 2^{h+1} - h$

$T(h) = 2^{h+1}.h - [2 + 2^2 + 2^3 + 2^4 + \cdots + 2^h]$

$\Rightarrow 2^{h+1}.h - 2^{h+1} + 2 = \left(2^{h+1}\right)\left(\sum_{i=0}^{h} 1\right) - \left[2^{h+1} - 1\right] + 1 \Rightarrow (n+1)\left[\log_2(n+1) - 1\right] - n + 1$

$\frac{2\left[2^h - 1\right]}{2-1} = 2^{h+1} - 2$

In elements in max heap

$2^{h+1} - 1 = n$

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

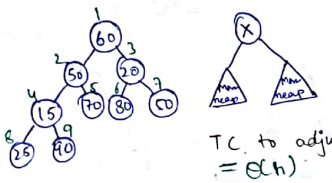
$h = \log_2(n+1) - 1$

$T(n) = n\log_2 n$

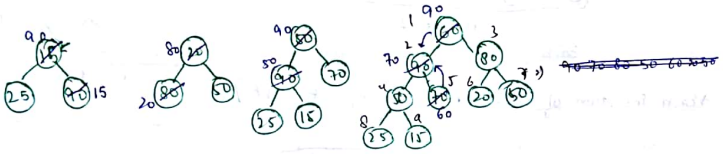
* Bottom up approach:

	1	2	3	4	5	6	7	8	9
a	60	50	20	15	70	80	50	25	90

internal nodes $\lfloor n/2 \rfloor$ | leaf nodes $\lceil n/2 \rceil$



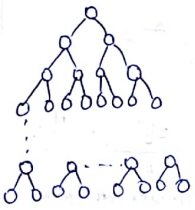
TC to adjust node "x" of height "h" into maxheap:
$= \Theta(h)$



90	70	80	50	60	20	50	25	15

max heap array

* TC to build maxheap of height "h" using bottom up.



Height	No. of nodes of given height	Cost per adjustment [# of shift]
h	1	h
h-1	2	h-1
h-2	2^2	h-2
⋮	⋮	⋮
1	2^{h-1}	1
0	2^h	0

$T(h)$ = TC to build max heap in bottom up approach of height (h).
[sum of heights of all nodes in CBT of height h]

$T(h) = 1*h + 2*(h-1) + 2^2*(h-2) + \ldots + 2^{h-1}*1$

$2T(h) = 2*h + 2^2*(h-1) + 2^3*(h-2) + \ldots + 2^h*1$

$T(h) = -h + 2 + 2^2 + 2^3 + \ldots + 2^{h-1} + 2^h$

$\Rightarrow [2 + 2^2 + 2^3 + \ldots + 2^h] - h = 2[2^h - 1] - h$

$2^{h+1} - 1 = n$

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

$\Rightarrow 2[2^h - 1] - h = 2^{h+1} - 1 - 1 - (\log_2(n+1) - 1)$

$\Rightarrow h - h \le \log_2(n+1) + 1 = n - \log_2(n+1)$

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

* Algo to build max heap top-down approach:

```
Algo MaxheapTD(a,n)
{ for(i=2; i<=n; i++)
    { InsertMaxheap(a, n, a[i])
    }
}
```

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

* Algo to build max heap bottom up approach:

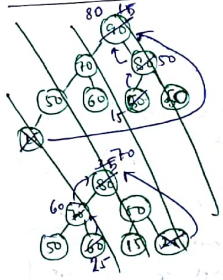
```
Algo MaxheapBU(a,n)
{ for(i=⌊n/2⌋; i>=1; i--)
    Adjust(a,i,n);
}
```

TC = $\Theta(n)$

* Most operations (comp.) take const. time. few take logn time.
*. overall $\Theta(n)$

$\Theta(n\log n)$

\# Heap Sort Algorithm:



1	2	3	4	5	6	7	8	9	
60	50	20	15	70	80	90	25	90	i/p array

⇓

1	2	3	4	5	6	7	8	9
90	70	80	50	60	20	50	25	15

⇓

80	70	50	50	60	20	15	25	90

max heap | sorted

⇓

70	60	50	50	25	20	15	80	90

```
Algo Heapsort(a,n)
{ // build maxheap                          ↑
  for(i=⌊n/2⌋; i>=1; i--)                   Θ(n)
  { Adjust(a,i,n);                          ↓
  }
  // delete max from a[i] to a[1] & place max at a[i] where i from n to 2.
  for(i=n; i>=2; i--)                       ↑
  { a[i] = delMax(a,1,i);                   Θ(nlogn)
  }                                         ↓
}
```

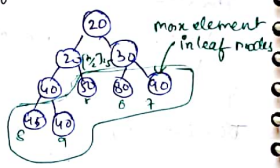
* TC of Heap Sort = $\Theta(n \log n)$ [All cases]
→ In place ~~sorta~~ sorting algo.
→ Not stable sorting algo.

Heap Operations:

① TC to find min element in min heap of n elements: $\Theta(1)$ — Efficient operation
② TC to delete min element from min heap of n elements: $\Theta(\log n)$ — Efficient operation
③ TC to find max element in min heap of n element: $\Theta(n)$

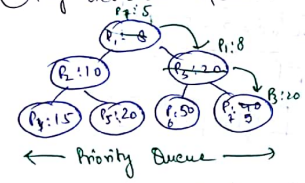
```
max = a[⌈n/2⌉]
for (i=⌈n/2⌉+1; i≤n; i++)
{ if (a[i] > max)
    max = a[i];
}
```

*Not efficient operation



④ TC to delete max from min heap of n elements:
→ Find max pos in heap. — $\Theta(n)$
→ del max — $\Theta(\log n)$
$\Theta(n)$

***
⑤ Key decrement operation from min heap of n elements: $\Theta(\log n)$



$P_7: 40 \to 5$ Minval: high priority.
[Key decrement of min heap used for increasing priority of same process]
* Insert fun^c
Shift down → Insert
Shift Up → Adjust
← Priority Queue →

⑥ Key increment operation TC from min heap of n elements: $\Theta(\log n)$
→ use adjust fun^c.

⑦ Find k^{th} smallest element in min heap of n ~~ele~~ distinct elements:
Method-1: ① delete min from min heap (k-1) times — ② $\Theta(k \log n)$
② Root is k^{th} minimum

Method 2:

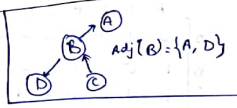
	comp.
1st min is a[1]	
2nd min {a[2], a[3]}	1
3rd min {a[4], a[5], a[3]}	2
4th min {a[8], a[9], a[5], a[3]}	3
⋮	⋮
k^{th} min { k elements }	k-1

Find k^{th} min of min heap	If k is constant	if k in terms of n {k = O(n)}
Method 1 $O(k \log n)$	$\Theta(\log n)$ efficient	$\Theta(n \log n)$ efficient
Method 2 $\Theta(k^2)$	$\Theta(1)$	$\Theta(n^2)$

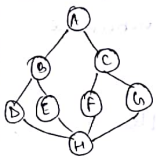
Q. What is TC to find 7th min in min heap of n distinct elements:
(a) $\Theta(\log n)$ (b) $\Theta(n \log n)$ (c) $\Theta(1)$ (d) $\Theta(n^2)$

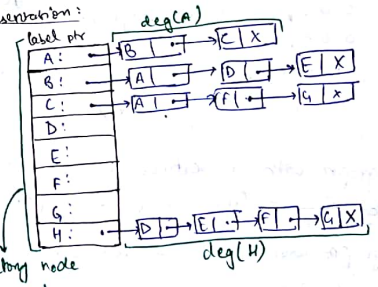
Q. What is TC find $\frac{n}{10}^{th}$ min in min heap of n distinct elements:
(a) $\Theta(\log n)$ (b) $\Theta(n \log n)$ (c) $\Theta(1)$ (d) $\Theta(n^2)$

Graph Representation:
(a) Adjacency list representation. [using linked lists]
(b) Adjacency matrix representation. [using 2-D array]



(a) Adjacency list representation:





Using adjacency list representation of graph:

① TC to find adj of given vertex (v): Adj(v) = $\Theta(deg(v))$ $\{\underset{BC}{\Omega(1)}$ to $\underset{WC}{O(n)}\}$

② TC to find adj of all vertices of graph:

$Adj(v_1) + Adj(v_2) + \ldots + Adj(v_n) =$
$deg(v_1) + deg(v_2) + \ldots + deg(v_n) = 2 \cdot e$
$= O(n+e)$

```
for(i=1; i<=n; i++)
{ find Adj(i); }
```
$\Theta(n+e)$

$\Omega(n)$ to $O(n^2)$

* Best case is when graph is tree ←

Graph is complete connected graph

③ SC to store graph with n vertices and edges:

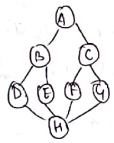
$O(n+e)$ $\{\Omega(n)$ to $O(n^2)\}$

n: space for directory node
2e: space for all linked lists.

ⓑ Adjacency Matrix Representation: ⓑ

define 2D array Adj[1..n, 1..n]

$Adj(i,j) = \begin{cases} 1 & \text{if } (i,j) \in \text{edge} \\ 0 & \text{if } (i,j) \notin \text{edge} \end{cases}$



	A	B	C	D	E	F	G	H
A	0	1	1	0	0	0	0	0
B	1	0	0	1	1	0	0	0
C	1	0	0	0	0	1	1	0
D	0	1	0	0	0	0	0	1
E	0	1	0	0	0	0	0	1
F	0	0	1	0	0	0	0	1
G	0	0	1	0	0	0	1	
H	0	0	0	1	1	1	1	0

Using adjacency matrix representation of graph:

① TC to find adj of given vertex: $\Theta(n)$

② TC to find adj. of all vertices in graph: $\Theta(n^2)$

```
for(i=1; i<=n; i++)
{ find Adj(i); }
```
$\Theta(n^2)$

③ SC to store graph with n vertices & e edges: $\Theta(n^2)$

Sparse Graph: graph with n vertices and O(n) edges [$\leq n$ edges]

E.g. ~~Sparse~~ Tree is sparse graph.

Dense Graph: graph with n vertices and $O(n^2)$ edges [max^m possible edges]

E.g. Complete connected graph.

→ less SC & TC.

→ Adjacency list representation preferred to store sparse graphs.

→ Adjacency matrix representation prefered to store dense graphs. [no overhead of pointers]

GREEDY ALGORITHMS:

→ Used to solve optimization problems.

→ Optimization problem: maximization or minimization of result of solution.

→ Usually no. of feasible solution to solve problem with n i/p:
⇒ 2^n feasible solution [subset problem] [0-1 knapsack]
⇒ n! feasible solution [permutation problem] [TSP]

→ Greedy algorithm solves optimization problem by using predefined strategy [static approach].

→ Greedy solution chooses one feasible solution from 2^n and n! feasible solutions and retain cost of solution. [statically].

→ Always run in polynomial TC.

→ Every optimization problem may not resolved using greedy approach. [Some optimization problem may not result optimal value using greedy algo.]

E.g. Coin change problem:

Denomination of coins {50, 25, 20, 10, 5, 2, 1} Rs.

Q. How many minimum coins to denominate Rs.143?

1st dec.	2 × 50	= 100	143
2nd dec.	1 × 25	= 25	118
3rd	0 × 20	= 0	
4th	1 × 10	= 10	8
5th	1 × 5	= 5	3
6th	1 × 2	= 2	1
7th	1 × 1	= 1	0

4 coins ← More denomination coin place first [strategy]

* Coin change problem failed to solve using greedy algorithm.

Dynamic: → Compute all possible solutions.
→ Return optimal solution.

2 × 50
0 × 25
2 × 20
0 × 10
0 × 5
1 × 2
1 × 1

6 coins.

50 : 1 2
25 : 0 1 2 3
20 : 0 1

Single Source Shortest Path algorithms 30/05/2017

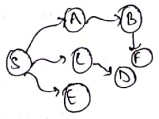
① Djikstra's Algorithm [Greedy]
② Bellman Ford algorithm [Not greedy]

① Djikstra's Algo:

Graph (n,e) n: no. of vertices
e: no. of edges

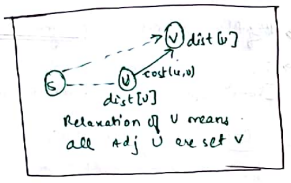
cost [1...n, 1...n] // edge costs

$$cost[i,j] = \begin{cases} \text{Edge cost} & \text{if } (i,j) \in \text{Edge} \\ \infty & \text{if } (i,j) \notin \text{Edge} \end{cases}$$



S: source vertex

→ Compute shortest path from source (s) vertex to every vertex of graph. Determine dist [1...n] array s.t. dist [i] shortest path cost from s to i.

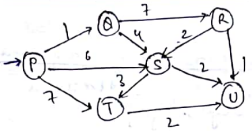


Djikstra's Procedure

// Initialisation
1. dist [1...n] = +∞;
prev [1...n] = -1;
dist[s] = 0
. List of all vertices which are not completed relaxation
// all n vertices in list.

// Relaxation Procedure
2. Choose vertex (u) from list which is min dist & which is not relaxed before. } Greedy Solution
v is set of Adj. of vertex (u)
for all v set
{ if (dist [u] + cost (u,v) < dist [v])
{ dist [v] = dist [u] + cost (u,v)
prev [v] = u
}
}
3. Repeat ② for all vertices

Q.



Dist

P	Q	R	S	T	U
0	∞ 1	∞ 8	∞ 5	∞ 7	∞ 7

Prev

P	Q	R	S	T	U
-1	N P	N Q	N Q	N S	N S

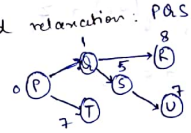
List // Min heap

P	Q	R
S	T	U

P : Q : S : T : U : R

① Order of vertices completed relaxation: PQSTUR (or) PQSUTR

② Djikstra's spanning tree:



*Algo Djikstra (Graph(V), n, e, cost [ ][ ], s)
{ // Initialisation
for (i=1; i≤n; i++) } O(n)
{ dist [v,i] = +∞;
prev [v,i] = -1;
}
dist [s] = 0
O(n) [Build Minheap (Q) for all vertices based dist. of vertex is priority.

// Apply edge relaxation for each vertex
while (Q is not empty) } n
{ del vertex (u) whose dist minimum. [O(logn)]
Find Adj of vertex (u) are v set [TC: deg(u) or O(n)] Adj. List / Adj. Matrix
for (each vertex of v set) } deg(v)
{ if (dist [u] + cost (u,v) < dist [v])
{ dist [v] = dist [u] + cost (u,v);
prev [v] = u
key decrement (Q, v, dist [v]); O(logn)
}
}
}
return (dist [ ], prev [ ])
}

TC of Dijkstra's Algo:

(a) Assume graph G is Adj list representation & Min heap used. (Preferred)

$$\Theta(n) + \Theta(n) + n\left[\log n + \deg(u) + \deg(u).\log n\right]$$

- $\Theta(n)$ → Initialisation of dist[]
- $\Theta(n)$ → Build min heap
- $\log n$ → del min
- $\deg(u)$ → fig find $\frac{\deg}{\text{adj}}$ of u
- $\deg(u).\log n$ → key decrement.

$$= \Theta(n) + \Theta(n) + n\log n + \sum_{i=1}^{n} \deg(U_i) + \sum_{i=1}^{n} \deg(U_i).\log n$$

$$= \Theta(n) + \Theta(n) + n\log n + e + e.\log n$$

$$= \Theta(n+e).\log n$$

(b) Assume graph G is Adj. matrix representation & Min heap used

$$\Theta(n) + \Theta(n) + n\left[\log n + \Theta(n) + \deg(U).\log n\right]$$

$\Theta(n)$ → Find Adj of U

$$= \Theta(n) + \Theta(n) + n\log n + \Theta(n^2) + \Theta(e).\log n$$

~~$= \Theta(n^2)$~~ $= \Theta(n^2 + e\log n)$

(c) Assume graph G is adj list representation & sorted linked list used for priority queue.

$$= \Theta(n) + \Theta(n\log n) + n\left[\Theta(1) + \deg(u) + \deg(u).n\right]$$

- $\Theta(n)$ → Initialisation of dist[]
- $\Theta(n\log n)$ → Build sorted LL

$$= \Theta(n) + \Theta(n) + n + \Theta(e) + e.n$$

~~$= \Theta(e.n)$~~ $= \Theta(e.n)$ $[\Omega(n^2)$ to $O(n^3)]$ Bc wc

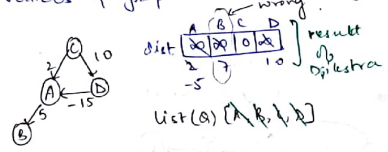
(d) Assume graph G in adj. list and unsorted array used as priority queue.

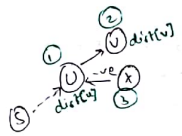
$$= \Theta(n) + \Theta(1) + n\left[n + \deg(u) + \deg(u).\Theta(1)\right]$$

$$= \Theta(n^2 + e) = \Theta(n^2)$$

Limitations of Dijkstra's Algo:

(1) Dijkstra's algo may fail to find shortest path from source to other vertices if graph consists -ve edges. [Assume no -ve cycle]

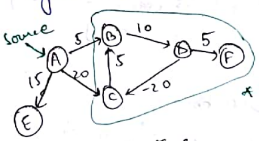




① vertex u completed relaxation
② dist[v] depends on dist[u]
③ because of relaxation vertex (x), dist[u] reduces

then only Dijkstra fails.

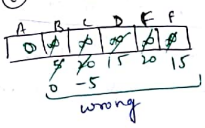
(2) Dijkstra's algo fails to detect negative cycle if ~~neg~~ negative cycle reachable from source.



A → B → D → F = 20
A → B → D → C → B → D → F = 15 ↓10 ↓5

* Not possible to find shortest path from source (A) to {B, C, D, E, F}

[Algo should detect -ve cycle]
↳ Not possible using Dijkstra's

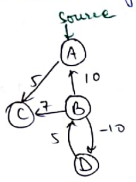


Not possible to compute shortest path from A to BCDF.
// (Cycle detection required)

② Bellman Ford Algo: [Not greedy method]

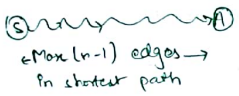
→ Compute shortest path cost from source correctly even -ve edge exists. [Assume no -ve cycle]

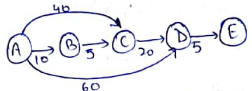
→ Detect negative cycle if negative cycle reachable from source.



* Bellman Ford algo also doesn't detects -ve cycle.

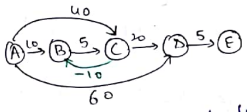
* If no negative cycle exists, shortest path from source(s) to vertex(A) should consist at most (n-1) edges. [n-vertices]





A to E ⇒ 4 edges b/w shortest path from A to E.

→ If distance from source with (n-1) edges is more than dist[A] from source with more than (n-1) edges then vertex "A" reachable from source through -ve cycle.



A-B-C-D-E → 40 (n-1) edges

A-B-C-B-C-D-E → 35 [more than (n-1) edges]

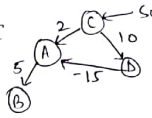
* Procedure of Dijkstra's Bellman Ford:

① Initialize
dist[v°] = ∞
Prev[v°] = -1 for all vertices
dist[s] = 0

② Apply edge relaxation for all edges of graph.
&
Repeat (n-1) times.

③ Cycle Detection
Apply edge relaxation nth time:
If any vertex dist[v] reduces then negative cycle going through source.

Eg



	A	B	C	D
dist	-5	7	0	10

Djikstra wrong

Bellman Ford:

	A	B	C	D
dist	∞	∞	0	∞

edges:
C-A
C-D
A-B
D-A

(n-1) times

i=1	-5	7	0	10
i=2	-5	0	0	10
i=3	-5	0	0	10

dist result after (n-1) times of edge relaxation

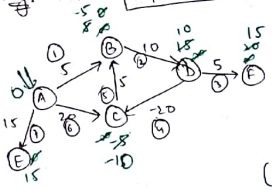
nth time i=4 | -5 | 0 | 0 | 10 | Correct dist value

No -ve cycle

Eg 2. dist

A	B	C	D	E	F
0	0	-5	15	15	20

Result of Djikstra



* BCDF → going through -ve cycle from source.

	A	B	C	D	E	F
dist	0	∞	∞	∞	∞	∞
i=1	0	0	-5	15	15	20
i=2	0	-5	-10	10	15	15
i=3	0	-10	-15	5	15	10
i=4	0	-15	-20	0	15	5
i=5	0	-20	-25	-5	15	0
nth i=6	0	-25	-30	-10	15	-5

(n-1)

A B C D E F

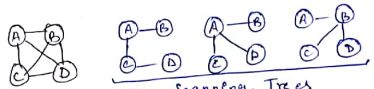
Algo BellmanFord (n, e, cost[][], s, E[][])
// edges

```
{ //Initialisation
O(n)  for (i=1; i<=n; i++)
      { dist[vi] = +∞
        prev[vi] = -1
      }
  dist[s] = 0
  //Apply edge relaxation  for each edge of graph and repeat (n-1) times.
  for (i=1; i<=n-1; i++)        (u)→(v)
O(n.e)  { for(All edges of graph(u,v))
(n-1)     { if(dist[u] + cost[u,v] < dist[v])
times       { dist[v] = dist[u] + cost[u,v]
              prev[v] = u;
            }
        }
  // testing if negative cycle reachable from source exists or not.
  for (all edges of graph (u,v))
O(e)  { if(dist[u] + cost[u,v] < dist[v])
        { //Negative cycle exists
          return (FALSE);
        }
  }
  return (dist[], prev[])
  //No negative cycle
}
```

* TC of Bellman ford Algo : $\theta(n.e)$
  WC TC $\Rightarrow O(n^3)$

Minimal Spanning Tree : 31/08/2017

→ Spanning Tree : If graph $G(V,E)$ is G' such that $G'(V,E')$ where $E' \subseteq E$ and G' is connected and no cycle.



* n^{n-2} spanning trees possible in complete connected graph with n-vertices.

↳ Complete ~~complet~~ connected 4 nodes $\Rightarrow 4^{4-2} = 16$ spanning tree

* Kirchoff Theorem: [To find no. of spanning trees of graph]

① Represent Graph G in adjacency matrix format:

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 1 & 1 & 0 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 1 & 1 & 0 & 1 \\ 4 & 0 & 1 & 1 & 0 \end{array}$$

② Multiply (-1) to adj. matrix

③ Diagonal element replace deg. of adj.(i,i)

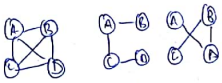
$$\begin{bmatrix} 0 & -1 & -1 & 0 \\ -1 & 0 & -1 & -1 \\ -1 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 \end{bmatrix} \quad \begin{bmatrix} 2 & -1 & -1 & 0 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 3 & -1 \\ 0 & -1 & -1 & 2 \end{bmatrix}$$

Cofactor of $Adj(1,1) = (-1)^{1+1} \det \begin{bmatrix} 3 & -1 & -1 \\ -1 & 3 & -1 \\ -1 & -1 & 2 \end{bmatrix}$

$= \{3(6-1) - (-1)(-2-1) + (-1)(1+3)\}$

$= 8$ (ST's of graph)

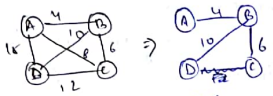
Ques. How many edge disjoint ST's possible in complete connected graph of n vertices ? $[n \geq 2]$

Complete connected graph	No. of edge disjoint ST.
	1
	1
	2
⋮	

$\frac{n(n-1)}{2} \Rightarrow \frac{\frac{n(n-1)}{2}}{(n-1)} = \left\lfloor \frac{n}{2} \right\rfloor$ ST's (edge disjoint)

Minimal Spanning Tree:

→ Sum of cost of edges of ST of graph is minimum.



⇒

MST.

Q. Distinct edge costs. ∀ [1,2,3,4,5,6]

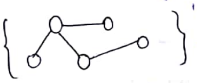
More cost of MST?

(a) 6 (b) 7 (c) 9 (d) 10

① Prim's Algo] Greedy
② Kruskal's Algo] Algorithms.

Prim's Algorithm:

→ MST constructs edge by edge set edges which are included in MST must be always connected.



'x' is set of edges included into MST so far which must be connected.

(u,v) next edge of MST connected s.t. {x ∪ (u,v)} must be connected & min inc. in sum of edge costs.

dist

1	2	3	4	5	6	7
∞	∞	∞	0	∞	∞	∞
∞	25	22		16	12	15
10						

prev

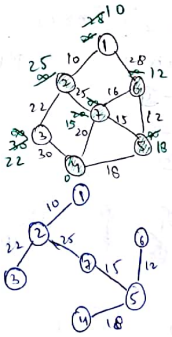
1	2	3	4	5	6	7
-1	-1	-1	-1	-1	-1	-1
				4	5	4/6
2	7	2				

color

1	2	3	4	5	6	7
W	W	W	W	W	W	W
B	B	B	B	B	B	B

color[v_i] = { W; if v_i not included into MST; B; if v_i included into MST

Minheap(Q): All vertices based on dist(v) is priority [dist]



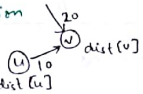
MST cost = 102

"4" "5" "6" "7" "2" "1" "3"

MST using prev[v]

cost MST: using dist[v]

Relaxation




```
if (dist[v] > cost(u,v) && color[v] == W)
{ dist[v] = cost(u,v)
  prev[v] = u
}
```

```
Algo Prims (Graph, n, e, cost[][])
{ for (i=1; i≤n; i++)
  { dist[v_i] = ∞                    O(n)
    prev[v_i] = -1
    color[v_i] = W
  }
  Choose one vertex as source dist[s] = 0
  [Build minheap(Q) for all vertices based on dist[vertex] priority.   O(n)
  while (Q not empty)
  { dele vertex x(u) which min dist. from minheap.
    color[u] = B  // u included into MST.
    v is adj of u
    for (all vertices (v) adj of u)
    { if (dist[v] > cost(u,v) && color[v] == "W")
      { dist[v] = cost(u,v)
        prev[v] = u
        { keydec(Q, v, dist[v])
      }
    }
  }
  return(prev[], dist[])
}
```

#TC of Prim's Algo:

(a) Assume Adj list graph && priority queue min heap : $O((n+e).\log n)$

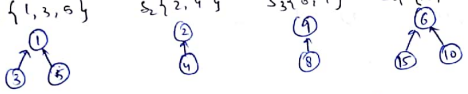
(b) Assume Adj matrix graph && priority queue min heap : $O(n^2 + e\log n)$

(c) Assume Adj list graph && priority queue unsorted array : $O(n^2)$

SET Algo: [used in Kruskal's algo to detect cycle in MST construction.

Representation of sets [disjoint set]:

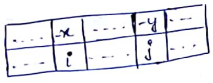
$S_1\{1,3,5\}$ $S_2\{2,4\}$ $S_3\{8,9\}$ $S_4\{6,7,10\}$



P(i)	-3	-2	1	2	1	-3	9	-2	6	6
i	1	2	3	4	5	6	8	9	10	15

Representation of disjoint sets using array.

* Union Algo:



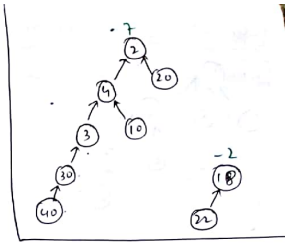
if set with root i has more elements than set with root j then i becomes parent of j.

Algo Union(i,j)
{
  z = p[i] + p[j]
  if (p[i] ≤ p[j]) //set with root i has more elements.
  { p[j] = i;
    p[i] = z;
  }
  else //set with root j has more elements.
  { p[i] = j
    p[j] = z
  }
}

TC of Union = $O(1)$

* Find Algo:

	-7	4	2	4	2	3	30	-2	18	p[i]
i	2	3	4	10	20	30	40	18	22	



Find(i): should return root set tree of element i present.

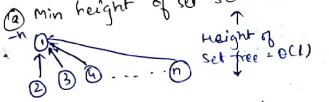
Find(24) = 18
Find(10) = 2
Find(40) = 2
Find(2) = 2

Algo Find(i)
{ while(p[i] > 0) TC of Find operation: $O(\text{height of set tree})$
  i = p[i];
  }
  return(i);
}

=> Initially n sets, each set one element

i	1	2	3	4	5	...	(n-1)	n
p(i)	-1	-1	-1	-1	-1	...	-1	-1
	1	2	3	4	5	...	n-1	n

(a) Min height of set tree if n sets merge into one set using union op.

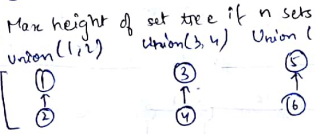


Union(1,2)
Union(1,3)
Union(1,4)
...
Union(1,n)

$O(1) \times$ min height

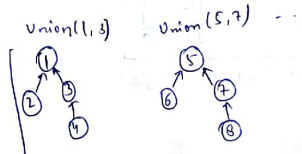
(b) Max height of set tree if n sets merge into one set using union op. $O(\log n) \times$ max height

Union(1,2) Union(3,4) Union(5,6) ... Union(n-1,n)



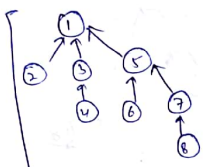
$\frac{n}{2}$ sets, each set 2 elements
//Height = 1

Union(1,3) Union(5,7) ... Union(n-3,n-1)



$\frac{n}{2^2}$ sets, each set 2^2 elements
//Height = 2

Union (1,5). ... Union (n-7, n-3)



$\frac{n}{2^3}$ sets : Each set 2^3 elements.

//Height = 3

$\frac{n}{2^k}$ sets each set 2^k elements.

//Height : $k = O(\log_2 n)$

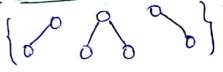
$\frac{n}{2^k} = 1 \Rightarrow k = \log_2 n$

* O(logn) worst case TC of find operation.

Kruskal's Algorithm:

→ Constructs MST edge by edge.

→ Let "x" be set of edges included into MST so far. These "x" edges may not connected.

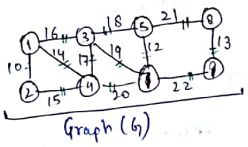


* (u,v) next edge into MST such that {x ∪ (u,v)} should not form cycle & min increment in sum of edge cst.

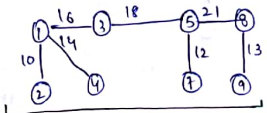
Procedure of Kruskal's Algo:

n : vertices of graph
e : edges of graph.

① Del Min cost edge (u,v) from graph G.
② Add (u,v) into MST if (u,v) not forms cycle in MST
③ Repeat ① & ② until MST becomes (n-1) edges or graph G becomes 0 edges.



Graph (G)



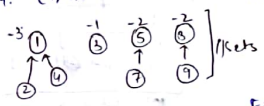
MST (G') n-1 edges.

* Set Algo used to detect cycle in MST construction using Kruskal's Algo.

if (u,v) edge min cost edge deleted from graph
  if (find(u) != find(v))
    // (u,v) not forms cycle in MST
  else
    //(u,v) forms cycle in MST

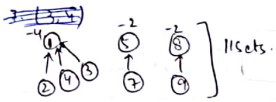
} //Initially each vertex is one set

Edges deleted from graph		Edge forms cycle in MST
1. (1,2)	find(1) ≠ find(2)	NO [Add (1,2) to MST]
2. (5,7)	find(5) ≠ find(7)	NO [Add to MST]
3. (8,9)	find(8) ≠ find(9)	NO "
4. (1,4)	find(1) ≠ find(4)	NO "



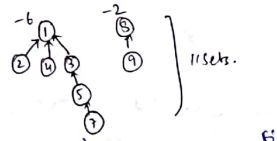
//sets

5. (2,4)	Find(2) = find(4)	YES [Discard (2,4)]
6. (1,3)	Find(1) ≠ find(3)	NO [Add to MST]



//sets.

7. (3,4)	Find(3) = find(4)	YES [Discard (3,4)]
8. (3,5)	find(3) ≠ find(5)	NO [Add to MST]



//sets.

(9). (3,7)	Find(3) = Find(7)	YES
10. (4,7)	Find(4) = find(7)	YES
11. (5,8)	Find(5) ≠ Find(8)	NO

* Continue until MST (n-1) edges or graph becomes '0' edges.

Algo Kruskal (n, e, cost[][])

O(e) { build minheap(Q) for all edges of graph based on priority edge cost.

O(n) { Initialise set array each vertex of graph is one set. // Initially n sets each set n-elements

```
i=0 // No. of MST edges.
MinCost = 0 // Cost of MST
while ( i < (n-1) && Q not empty)
{ Del MinCostEdge (u, v) from Minheap (Q) log e
  X = Find(u) , Y = Find(v) log n
  if ( X ≠ Y ) // (u,v) not forms cycle in MST
  { MinCost = MinCost + cost (u,v)
    i++;
    t[i,1] = u ; t[i,2] = v
    union(X, Y);
  }
}
if (i < n-1) // No spanning tree in graph
  return (FALSE);
else
  return (t[][], MinCost);
}
```

e times, d(1)

	1	2
1	1	2
2	5	7
3	8	9
4	1	4
:	:	:
n-1		

t[1...n-1, 1...2]

MST edge array

* TC of Kruskal's = $\Theta(e \log e) \approx \Theta(e \log n)$ — BC = $\Theta(n \log n)$, WC = $\Theta(n^2 \log n)$

Huffman Coding: [Optimal Prefix encoding]

→ Compression Technique. && Encoding technique.

→ Encoding: character to binary.

① Fixed length encoding

② Prefix Encoding.

*Fixed Length encoding: Every character of message encoded by same no. of bits.

n bits ⇒ 2^n distinct char. can encode.

E.g. [***] → 8 characters can encode. E.g. ASCII → 7 bits

To encode n characters → $\lceil \log_2 n \rceil$ bits per char.

---

aaababbaaacdaa
bbbcxxyyzaabcc
cddyzz

11	a	7 distinct ⇒ $\lceil \log_2 7 \rceil$ = 3 bits characters
7	b	
5	c	
4	d	
2	x	Using fixed length encoding message length = [11+7+5+4+2+3+3] × 3 = 35 × 3 = 105 bits.
3	y	
3	z	

Disadvantage

*Message length is high because no difference between most frequently used && least frequently used with respect to no. of bits.

*Prefix Encoding [Huffman Encoding]:

→ Most frequently used character represented by less # of bits.

→ Least frequently used character represented by more # of bits.

→ Proper prefix any char code is not code of any other character.

IP: [11010]

should not code for any other character: 11, 110, 1101

p: 11010
q: 110
r: 11

(11)(0)(1)0 11010110 ⇒ Decoding not possible

Advantage: Overall message length [sum of frequency counts] is optimal [min] if Huffman coding used for encoding.

2 way of optimal merge tree: [O(n log n)] // Min heap used for all n distinct char. & priority is frequency count of character.

n distinct char

① each char denote tree with weight is frequency count.

② del 2 min cost trees && merge into one tree with weight = sum of weights of both trees && insert merged tree into list of trees.

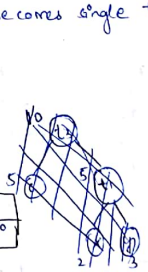
③ Repeat ② until list becomes single tree.

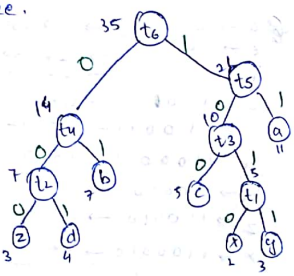
char a b c d x y z

freq. 11 7 5 4 2 3 3

{ a b c d x y z }

a	b	c	d	x	y	z
11	01	100	001	1000	1011	000





① Sum of frequency counts [message length] using Huffman coding:

$= 11*2 + 7*2 + 5*3 + 4*3 + 2*4 + 3*4 + 3*3$

$= 92$ bits.

② Avg. bits per character using huffman code:

$= \frac{\text{sum of freq.}}{\text{\# of char.}} = \frac{92}{35} = 2.63$

③ Min. & Max. bits for character [Min → 2 bits Max → 4 bits]

Q.

char	a	e	i	o	u	sp	nl	z	x
freq.	200	150	80	40	12	5	8	10	4

① Sum of freq. count using fixed length,

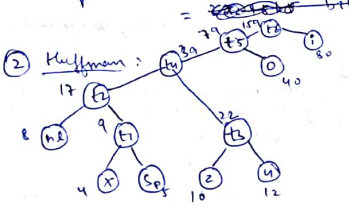
② " " " " Huffman.

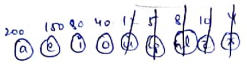
① 4 bits/character

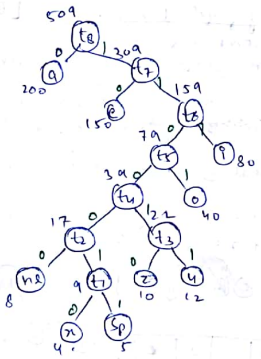
freq. count = $4 * [200 + 150 + 80 + 40 + 12 + 5 + 8 + 10 + 4]$

$= 2036$ bits

② Huffman:







a → 0	→ 1 * 200	200
e → 10	→ 2 * 150	300
i → 111	→ 3 * 80	240
o → 1101	→ 4 * 40	160
u → 110011	→ 6 * 12	72
sp → 11000011	→ 7 * 5	35
nl → 110000	→ 6 * 8	48
z → 110010	→ 6 * 10	60
x → 1100010	→ 7 * 4	28

1143 bits $= t_1 + t_2 + t_3 + t_4 + t_5 + t_6 + t_7 + 8$

$= 9 + 17 + 22 + 39 + 79 + 159 + 309 + 509$

③ Avg. bits/char $= \frac{1143}{509} = 2.25$ bits.

2 way Optimal Merge Tree TC:

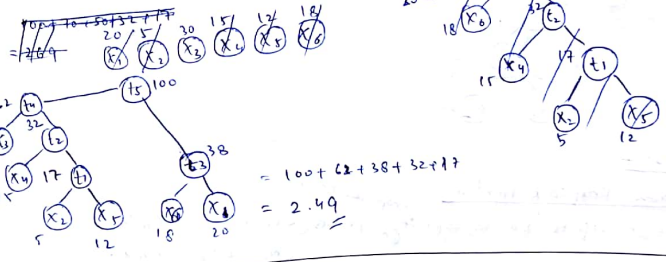
① Build min heap → $\Theta(n)$

② [2 Min del && Insertion] repeat $(n-1)$ times. → $(n-1)[2\log n + 1\log n]$

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

Q. char $[x_1\ x_2\ x_3\ x_4\ x_5\ x_6]$

freq $[20\%\ 5\%\ 30\%\ 15\%\ 12\%\ 18\%]$



$= 100 + 62 + 38 + 32 + 17$

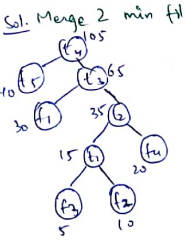
$= 2.49$

Q. Assume $(n+m)$ record moves required to merge two files with n & m sorted records each into single file of sorted record. How many minimum record moves required to merge n=5 (files)

$\langle f_1\ f_2\ f_3\ f_4\ f_5 \rangle$ with for

$\langle 30\ 10\ 5\ 20\ 40 \rangle$ sorted records each into single sorted file.

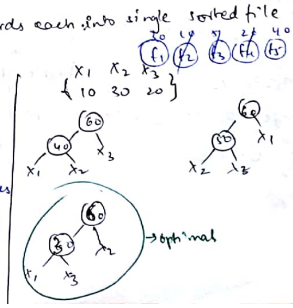
Sol. Merge 2 min file & so on.



$= 105 + 65 + 35 + 15$

$= 220$

[Minimum record moves to merge 5 files into single file]

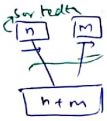


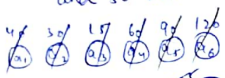
DATE

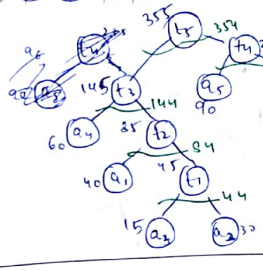
Q. 6 sorted arrays $[a_1\ a_2\ a_3\ a_4\ a_5\ a_6]$
$\{40\ 30\ 15\ 60\ 90\ 120\}$ elements in sorted order.

How many minimum comparisons required to merge 6 sorted arrays into single sorted array using optimal algorithm?

Sol. Merge two array which are min. elements







$= 354 + 209 + 144 + 84 + 44$

$= 835$ comparisons.

01/09/2017

Fraction knapsack Problem:

→ Given n objects & a m capacity knapsack (bag)
profits $\langle P_1\ P_2\ P_3 \dots P_n \rangle$
weights $\langle W_1\ W_2\ W_3 \dots W_n \rangle$.

If x_i fraction of obj. 'i' included into knapsack which require $w_i * x_i$ weight and results $P_i * x_i$ Profit.
where $0 \le x_i \le 1$.
Determine fraction values $\{x_1, x_2 \dots x_n\}$ such that ~~from weights~~ [sum of weights included into knapsack] $\le m$ & maximize profit?

Maximize $\sum_{i=1}^{n} P_i * x_i$ Profit
subjected to $\sum_{i=1}^{n} w_i * x_i \le m$
& $0 \le x_i \le 1$

} fractional knapsack Problem

⇒ n = 6 [no. of objects] & m = 90

$\langle P_1\ P_2\ P_3\ P_4\ P_5\ P_6 \rangle$
$\langle 50\ 30\ 70\ 40\ 60\ 30 \rangle$
$\langle W_1\ W_2\ W_3\ W_4\ W_5\ W_6 \rangle$
$\langle 20\ 10\ 20\ 25\ 35\ 50 \rangle$

Soln. i^{th} object
w_i unit weight → P_i profit
Profit/unit weight = $\frac{P_i}{w_i}$

Greedy sol^n = Max P/w objects should choose first and so on.

$\langle \frac{P_1}{W_1} \frac{P_2}{W_2} \frac{P_3}{W_3} \frac{P_4}{W_4} \frac{P_5}{W_5} \frac{P_6}{W_6} \rangle = \langle \frac{50}{20} \frac{30}{10} \frac{70}{20} \frac{40}{25} \frac{60}{35} \frac{30}{50} \rangle$

$\{\langle 2.5, 3, 3.5, 1.6, 1.7 \}, 1.6 \rangle = \langle O_3, O_2, O_1, O_5, O_4, O_6 \rangle$

* fill knapsack by obj. in descending order of P/w ratios:

$w_i * x_i$ (≤ 90) ⇒ $20*1 + 10*1 + 20*1 + 35*1 + 25*\frac{5}{25} = 90$

$P_i \times w_i$ ⇒ $70*1 + 30*1 + 50*1 + 60*1 + \boxed{40*\frac{5}{25}}$ 8 = 218 = Max profit

Optimal profit

x_1	x_2	x_3	x_4	x_5	x_6
1	1	1	1/2	1	0

Optimal solution

Algo knapsack(P[], W[], n, m) ⇒ O(n log n) TC of fraction knapsack.

```
{ 1. sort objects based on decreasing order of their P/w ratios   O(nlogn)
  2. for (i=1; i<=n; i++)                                          O(n)
     { if (w_i <= m_i)
       { x_i = 1;
         profit = profit + P_i;
         m_i = m_i - w_i
       }
       else
       { x_i = m_i / w_i;
         profit = profit + P_i * x_i;
       }
     }
  3. return (x[], profit)
}
```

profit = 0, m_i = m

0/1 Knapsack Problem:

$\sum_{i=1}^{n} p_i x_i$ Maximise profit

$\sum_{i=1}^{n} w_i x_i \le m$ [Subjected to]

$x_i = 0$ or 1

Eg. $n = 3$

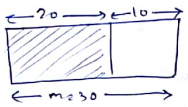
$\langle P_1, P_2, P_3 \rangle = \langle 60, 40, 30 \rangle$

$\langle w_1, w_2, w_3 \rangle = \langle 20, 10, 15, 11 \rangle$

Profit = 60 $\langle x_1\ x_2\ x_3 \rangle$

Not optimal $\langle 1, 0, 0 \rangle$

order obj $\langle 0_1\ 0_3\ 0_2 \rangle$



Brute Force Algorithm:

→ Compute cost of all feasible solution and return feasible set whose result is optimal.

→ 0/1 Knapsack Problem [Brute Force Method] $TC = \Theta(2^n)$

Compute profit for every subset of object.

sum of weights $\le m$

sum of profits maximum

obj	profit	weight
{ }	0	0
$\{0_1\}$	60	20
$\{0_2\}$	40	15
$\{0_3\}$	30	11
$\{0_1, 0_3\}$	90	31 ✗
$\{0_1, 0_2\}$	100	35 ✗
$\{0_2, 0_3\}$	70	26 ← optimal solution.
$\{0_1, 0_2, 0_3\}$	130	46 ✗

* ① → SSSP.
② → MST
③ → Huffman
imp.

Job Scheduling based on deadline:

→ given "n" tasks $[T_1, T_2 \ldots T_n]$ & Profits $\langle P_1, P_2 \ldots P_n \rangle$ & deadline $\langle d_1, d_2 \ldots d_n \rangle$

If T_i executed on/before d_i time, then resultant profit P_i, otherwise no profit.

Maximise profit in order to schedule tasks with constraints:

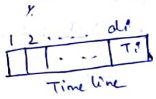
ⓐ only one task can execute at a time.
ⓑ every task ready to execute.
ⓒ one unit execution time for each task.

E.g.

	OS	CO	Algo	DS	DBMS
Marks	20	15	25	30	18
Deadline	3	2	3	1	2

} days. Max marks?.

→ Schedule assignments to get maximum marks.



day1	day2	day3
DS	OS	Algo

30 + 20 + 35 = 75

→ Choose Max Marks task $[T_i]$ & d_i

Greedy solution: Job scheduling ⇒ TC = $\Theta(n^2)$

① Sort tasks based on decreasing order of profits. $\Theta(n \log n)$

$\{T_1\ T_2\ T_3 \ldots \ldots T_n\}$ order of tasks

$P_1 \ge P_2 \ge P_3 \ge \ldots \ldots P_n$

② Choose task T_i in above order (d_i deadline) & schedule T_i at d_i^{th} time, if free.

if d_i time not free $(d_i - 1)^{th}$ time and so on.

* If no time slot free from d_i to 1 then discard T_i.

Time to schedule one task $\Theta(n)$ × n × #of tasks

HW. Pg 16. ②

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
	15	20	30	18	18	10	23	16	25
	7	2	5	3	4	5	2	7	3

20 + 23 + 25 + 18 + 30 + 15 + 16 = 147.

T_2	T_7	T_9	T_5	T_3	T_1	T_8
1	2	3	4	5	6	7

② → d.
③ → a.

Sorting Algorithms:

① Comparison based sorting algo:

Algorithm	Time complexity BC	Time complexity AC	Time complexity WC	Stable Sorting	Inplace Sorting
① Quick Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n^2)$	No	YES
② Merge Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	YES	NO
③ Heap Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	No	YES
④ Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	YES	YES
⑤ Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	No	YES
⑥ Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	YES	YES

* WC TC of any comparison based sorting algo = $\Omega(n\log n)$

② Non-comparison based sorting algo:

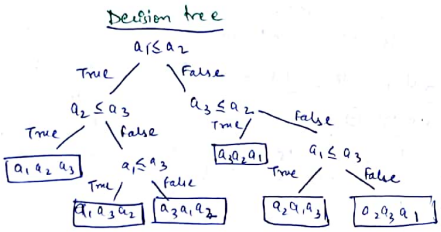
① Counting Sort: TC = $\Theta(n+k)$
- Stable
- Not in place

② Radix / Bucket Sort: TC = $\Theta(d \times n)$
- Stable
- In place

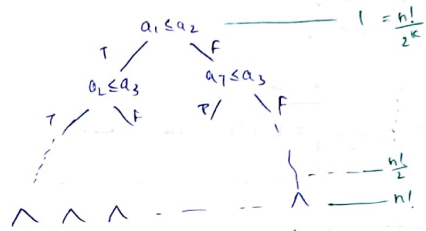
*** Theorem: Worst case time complexity of comparison based sorting algo for n elements is $\Omega(n\log n)$.

n=3 $[a_1\ a_2\ a_3]$

Decision tree



Given n element decision tree:



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

* WC atleast nlogn comparisons required for any comparison based sorting algorithm.

#Bubble Sort Algo: Compare and exchange adjacency elements. Stable & Inplace

*Procedure:

Pass 1: comp a[j] & a[j+1]
if a[j] > a[j+1] then swap(a[j], a[j+1])
for j=1 to n-1
//a[n] gets sorted.

Pass 2: comp a[j] & a[j+1]
if a[j] > a[j+1] then swap(a[j], a[j+1])
for j=1 to n-2
// a[n-1], n) get sorted.

Pass (n-1) Repeat

Eg: a

1	2	3	4	5	6
40	30	20	30	50	10
30	40	40	60	30	70
	20			10	

Pass 1

30	20	40	60	10	70
20	30		10	60	sorted

Pass 2

20	30	40	10	60	70
		10	40		

Pass 3

20	30	10	40	60	70
		10	30		

Pass 4

20	10	30	40	60	70
10	20				

Pass 5

10	20	30	40	60	70

```
Algo BubbleSort(a, n)
{
  for(i=1; i≤n-1; i++)
  {
    for(j=1; j≤n-i; j++)
    {
      if(a[j] > a[j+1])
      {
        swap(a[j], a[j+1])
      }
    }
  }
}
```

	# of comp.	# of swap.
Pass 1	n-1	[0 to n-1]
Pass 2	n-2	[0 to n-2]
⋮		
Pass(n-1)	1	(0 to 1)

TC = $\frac{n(n-1)}{2}$ + {0 to $\frac{n(n-1)}{2}$} = $\theta(n^2)$ in all cases.

Selection Sort Algorithm:

→ Find position of min element from a[j] to a[n] and swap(a[minpos], a[j])

where j is 1, 2, ... n-1 for n-1 passes.

* Procedure:

Pass 1 : Find position of min element a[1] to a[n] and swap with a[1]

Pass 2 : Find position of min element a[2] to a[n] and swap with a[2]

⋮

Pass n-1 : Find position of min element a[n-1] to a[n] & swap with a[n-1].

E.g.

	1	2	3	4	5	6	7
	60	50	30	60	50	10	20
Pass 1	10	50	30	60	50	60	20
Pass 2	10	20	30	60	50	60	50
Pass 3	10	20	30	60	50	60	50
Pass 4	10	20	30	50	60	60	60
Pass 5	10	20	30	50	50	60	60
Pass 6	10	20	30	50	50	60	60

```
Algo selectionsort(a,n)
{ for(i=1; i≤n-1; i++)
  {//find pos of min element a[i] to a[n]
   MinPos = i;
   for(j=i+1; j≤n; j++)
   { if(a[j]<a[MinPos])
       MinPos=j;
   }
   swap(a[i], a[MinPos])
  }
}
```

	# of comp	# of swap
Pass 1	n-1	1
Pass 2	n-2	1
⋮	⋮	⋮
Pass n-1	1	1

TC = $\left\{\frac{n(n-1)}{2} + (n-1)\right\} = \theta(n^2)$ All cases.

⇒ Selection Sort can be treated as special case of Quick sort where pivot element is minimum.

* * *

→ Selection sort TC → $c_1 \cdot n^2$ [AC]

Quick sort TC → $c_2 \cdot n\log n$ [AC]

[Selection sort runs faster than Quicksort for less i/p array]

[Quick sort runs faster than selection sort for large i/p array]

```
Sorting(a,n)
{ if (n<20)
    Selection sort(a,n)
  else
    Quick sort(a,n)
}
```

Insertion Sort:

→ a[i] placed in correct position in sorted part of array a[1...i-1].

where i=2 to n for (n-1) passes.

Procedure:

Pass 1 : place a[2] in correct position of sorted array a[1...1]
//a[1...2] in sorted order.

Pass 2 : place a[3] in correct position of sorted array a[1...2]
//a[1...3] in sorted order.

⋮

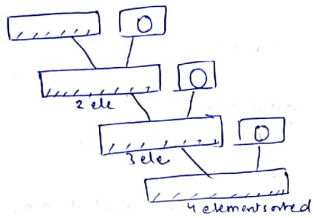
pass n-1 : place a[n] in correct position of sorted array a[1...n-1]
//a[1...n] in sorted order.

	1	2	3	4	5	6	7
	60	50	30	60	50	10	20
Pass 1	50	60	30	60	50	10	20
Pass 2	30	50	60	60	50	10	20
Pass 3	30	50	60	60	50	10	20
Pass 4	20	50	50	60	60	10	20
Pass 5	10	30	50	50	60	60	20
Pass 6	10	20	30	50	50	60	60

```
Algo Insertion sort(a,n)
{
 for(i=1; i≤n-1; i++)
  { item=a[i+1]; j=i;
    while(j≥1 && a[j]>item)
    { a[j+1]=a[j]
      j=j-1
    }
    a[j+1]=item;
  }
}
```

	Min & Min Comp.	Min shift	Max & Max Comp	Max shift	Avg & Avg Comp	Avg shift
Pass1	1	0	1	1	1	1
Pass 2	1	0	2	2	2/2	2/2
Pass 3	1	0	3	3	3/2	3/2
⋮	⋮	⋮	⋮	⋮	⋮	⋮
Pass(n-1)	1	0	(n-1)	(n-1)	(n-1)/2	(n-1)/2
	(n-1) comp	0	n(n-1)/2 comp. 2	n(n-1)/2 shift	n(n-1)/4 comp	n(n-1)/4 shift
	BC TC of insertion sort = Θ(n)		WC TC of insertion sort = Θ(n²)		AC TC of insertion sort = Θ(n²)	

* Insertion sort : special case of merge sort where every merge is merging of two sorted array with only one element in 2nd sorted array.



→ If input array with less elements, insertion sort runs faster than merge sort.

→ If i/p array with large more elements, merge sort runs faster than insertion sort.

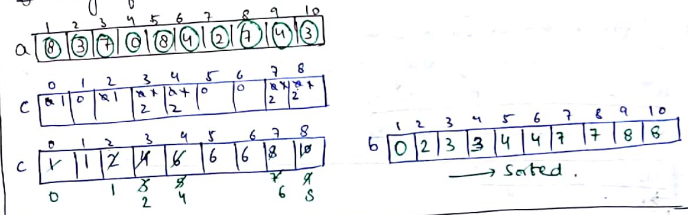
Non-comparison based sorting :

① Counting Sort Algo : array a[1...n] are n integers where element values range 0...k.

1. Initialize count array c[0...k] by 0's
2. for each element of array "a" increase count val. of count array
3. increase count array by previous index of count array

c[i] = c[i] + c[i-1]

E.g. array of 10 elements (int) whose range 0-8.



4. a[i] element place b[c[a[i]]] & decrease c[a[i]] by 1.

```
Algo CountingSort(a,n,k)
{ //a[1..n] are n integers whose element
  //range 0 to k.
  for(i=0; i<=k; i++)                    } Θ(k)
  { c[i]=0;                              }
  }
  for(i=1; i<=n; i++)                    } Θ(n)
  { c[a[i]]=c[a[i]]+1;                   }
  for(i=1; i<=k; i++)                    } Θ(k)
  { c[i]=c[i-1]+c[i];                    }
  for(i=n; i>=1; i--)                    } Θ(n)
  { b[c[a[i]]]=a[i];                     }
    c[a[i]]=c[a[i]]-1;
  }
  return(b[1..n]);
}
```

* TC of counting sort = Θ(n+k)

* Best case TC = Θ(n) [if k≤n]

* Stable sorting
* Not inplace sorting

② Radix Sort Algo:

Radix (r): r different symbols used to represent any number using number systems symbol val[0, 1, 2, ... r-1]

Hex: 16 [0, 1, 2, 3, ... 9, A, B, C, D, E, F]

Radix: 20 [0, 1, ---- 9, A, B, C, D, E, F, G, H, I, J]

Radix Sort: n: # of elements of array [1..n]
r: is radix of Number system
d: # of digits of each element.

Step 1: define r - Queues [Queue 0, Queue 1, ... Queue (r-1)]

Step 2: For each element of array place element into Queue based on k^{th} least significant digits of element.
k is 1 to d for d passes
and retrieve elements from Queue 0 to Queue (r-1) & store in array 'a'.
↳ Repeat d times.

E.g. n=8 d=3 r=10 (decimal numbers)

	1	2	3	4	5	6	7	8
a	956	743	143	650	836	985	143	743

Pass1:

Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9
650			743 143 143 743			836 985		956	

	1	2	3	4	5	6	7	8
	650	743	143	143	743	985	956	836

n Queue Insertions & n Queue delete operations

Pass2:

Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9
			836 743 143 143	650 743		956		985	

	1	2	3	4	5	6	7	8
	836	743	143	143	743	650	956	985

Pass3:

Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9
	143 143			650	743	836	743	956	985

	1	2	3	4	5	6	7	8
	143	143	650	743	743	836	956	985

→ Sorted

TC of Radix Sort: $n \rightarrow$ # of elements of array
$d \rightarrow$ # of digits of each element
$r \rightarrow$ radix

$\Theta(d.n)$

Best Case Tc of Radix Sort = $\Theta(n)$ [if d is const.]

Q. WB Pg 5 .①:
Element val (x) = n^3
# of digits to denote x in decimal = $\log_{10} n^3 = 3 \log_{10} n$
$x = n^3$
# of digits for x using radix = n (Number sys.)
$\log_n x = 3 \log_n n^3 = 3$ digits.

* Radix Sort Algo:

a[1..n] n elements in decimal val. of each element < 0..n^3>

① Convert each element of array from radix -10 to radix - n. [x is element of array]
$\log_n x = \log_n n^3 = \underbrace{3 \text{ digits}}_{\text{constant}}$ — $\Theta(n)$

② Run Radix sort for above resulted array. — $\Theta(n)$

③ Convert each element of above result from radix-n to radix-10 — $\Theta(n)$

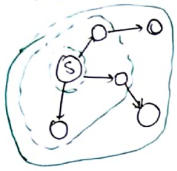
$\Theta(n) = TC$

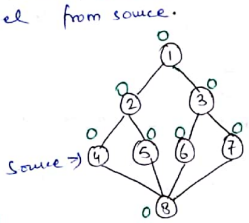
Graph Traversal Algorithm: — BFS / DFS

02/09/2017

① Breadth First Search: [Level order traversal] [Queue]

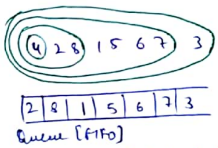
→ Graph vertices visits level by level from source.





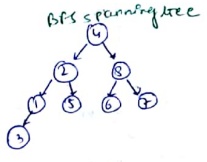
- Exploring vertex(u): Vertex "u" visited but adj: "v" not visited yet. [Grey]
- Explored vertex(u): vertex "u" visited Adj "v" also visited. [Black]

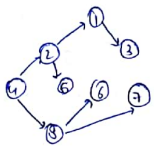
- Visited[vi] = { 0; Vi not visited / 1; Vi visited
- Unexplored vertex: Vertex "u" not visited. [White] (Not started exploring)



2	8	1	5	6	7	3

Queue [FIFO]





4,4e 2,2e 8,8e 1,1e 5,5e 6,6e 7,7e 3,3e

```
Algo BFS(G,n,e,S)
{ for(i=1; i≤n; i++)                    θ(n)
  { visited[vi] = 0
  }
  //Source S
  visited[s] = 1
  while(TRUE)
  { v is set of Adj. of s              deg(s) → θ(n)
    for(all vertices of v)             n
    { if(visited[v] == 0)              deg(s)
      { visited[v] = 1
        InsertQueue(Queue(Q), v);
      }
    }
    if(Empty(Queue(Q))) return ;
    else s = delete(Queue(Q))
  }
}
```

BFS

TC of BFS:

(a) Using adj. list graph: $O(n+e)$

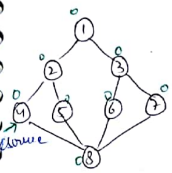
(b) Using adj. matrix graph: $\theta(n^2)$

SC of BFS: $O(n)$ Queue DS space.

Depth First Search: [stack]

→ Graph vertices visited based on depth.

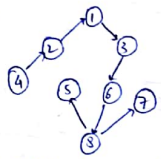
→ Vertex(v) which started exploration first is the vertex that completes exploration last.



DFS(4) Adj(4) = {2, 8}
  DFS(2) Adj(2) = {1, 4, 5}
    DFS(1) Adj(1) = {2, 3}
      DFS(3) Adj(3) = {1, 6, 7}
        DFS(6) Adj(6) = {3, 8}
          DFS(8) Adj(8) = {4, 5, 6, 7}
            DFS(5) Adj(5) = {2, 8}
            DFS(7) Adj(7) = {3, 8}

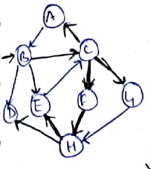
4s 2s 1s 3s 6s 8s 5s 5e 7s 7e 8e 6e 3e 1e 2e 4e

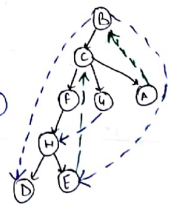
DFS Sequence = 4,2,1,3,6,8,5,7
[visited order]



Source: B

Back Edge: Graph edge which is connected child to parent.





---→ Forward Edge
---→ Back Edge
——→ Tree Edges

Cross Edge = 0
Back Edge = 2
Forward Edge = 3
Tree Edges = 7

- DFS Spanning Tree Edges: Tree Edges | Cross Edge: Graph edges which connect same level tree vertices.
- Forward Edge: Edge from prev. level to next level.

```
Algo DFS(G,n,e, s)
{ for(i=1; i≤n; i++)
  { visited [vi]=0 }                    } O(n)
  // s is source
  DFS(s)                                  → started empty
  { print (s);
    visited [s] = 1;
    V is set of adj of vertex (s) → deg(s)
    for (vertices of v set)
    { if (visited [v] == 0)               } deg(s)
      { DFS(v)                            } n func calls.
    }
  } print (s) → ended empty.
}
```

TC of DFS:

Adj. list → $O(n+e)$

Adj. matrix → $O(n^2)$

SC of DFS:

$= O(n)$

```
X=0
for(i=1; i≤n; i++)
  visited [vi]= 0
for(i=1; i≤n; i++)
{ if (visited [Vi]==0)
  { call DFS (vi);
    x++;
  }
}
return(x);
```

1	2	3	4	5	6	7	8
∅	∅	∅	∅	∅	∅	∅	∅
1	1	1	1	1	1	1	1

x = 1 DFS(1)
2 DFS(2)
3 DFS(3)
} $O(n+e)$

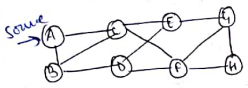
G has 3 connected components.

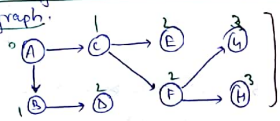
# of DFS Algo calls to be come all elements

****

Applications of BFS / DFS:

① BFS Algo can be used as single source shortest path algo for unweighted ~~edges~~ edges graph.





A	B	C	D	E	F	G	H
0	1	1	2	2	2	3	3

dist.

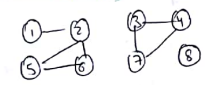
Result of single source shortest path.

TC = $(n+e)$ or (n^2)

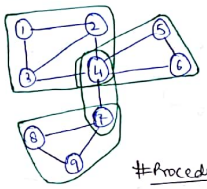
Q. which DS used to compute single source shortest path algo for unweighted graphs?

(a) Min heap (b) stack (c) Queue (d) linked list

② BFS / DFS can be used to test no. of connected components of undirected graphs.



③ BFS / DFS can be used to test # of bi-connected components of undirected graph.

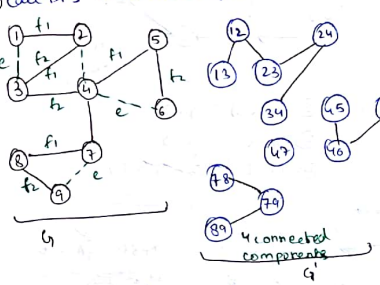


→ If graph G with no cut vertex [articulation point] then graph G is single bi-connected component.

→ If graph G has "k" cut-vertices, then G has at least (k+1) Bi-connected components.

Procedure to test # of bi-connected components of G:

① Call DFS & divide edges as Tree ~~edges~~ edges & (back edges) → all other edges except tree edges.



Auxillary Graph "G'"

→ Edges of G' are vertices of G'

→ Edges of G' are (e, f1, f2 ... fx) loop. back edge, tree edges

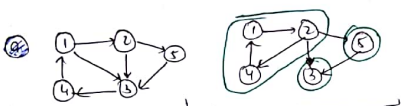
in G then ef1, ef2, ... efx are ~~tree~~ edges of G'.

② Construct auxilary graph.

③ # of connected components of G' is # of bi-connected edges of G.

④ DFS algo can be used to test # of strongly connected components of Directed Graph.

Strongly connected component: If every pair of vertices (u, v), there must be path from u to v & v to u.



one strongly connected

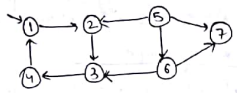
3-strongly connected.

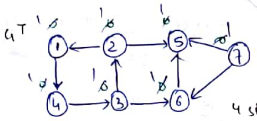
* Procedure to test # of strongly connected components.

TC : [O(n+e)]

(a) Call DFS and store finish time of exploration of vertices in stack(s)
(b) Compute G^T.
Transpose of G : G^T
(c) Call DFS for G^T based on unvisited vertex from top of stack as source.
[# of DFS calls to ~~visit~~ visit all vertices of G^T is # of strongly connected component of G].

Eg





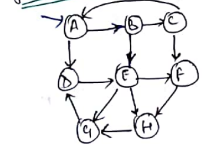
4 strongly connected components.

① Call DFS for G

5
6
7
1
2
3
4

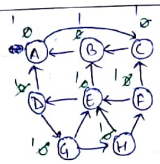
① DFS(5) = {5}
② DFS(6) = {6}
③ DFS(7) = {7}
④ DFS(1) = {1, 4, 3, 2}

Pg. 20.Q31.



①

A
B
C
F
H
G
D
E



① DFS(A) = {A, C, B}
~~DFS(G) =~~
~~DFS(H) =~~
DFS(E) = {F, E, D, G, H}

2 strongly connected components.

⑤ DFS Algo can be used to detect cycle in directed graph.
⑥ DFS Algo can be used to detect cycle in undirected graph.

TREE TRAVERSALS:

⇒ In order:

[Traverse Left subtree, Visit Root, Traverse Right subtree]

⇒ Pre-order:

[Visit Root, Traverse Left subTree, Traverse Right subtree]

⇒ Post-Order:

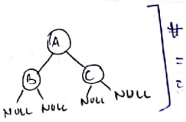
[Traverse Left subTree, Traverse Right subtree, Visit Root]

* Algo Inorder (t):

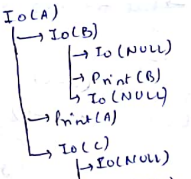
```
{ ~~Algo~~ if (t==NULL)
      return
  else
  { In-order (t→left);
    print (t. element);
    In-order (t→Right);
  }
}
```

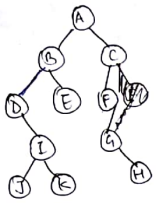
TC = O(n) [2n+1 fn calls, each call O(1) time]

SC = Ω(log n) to O(n)

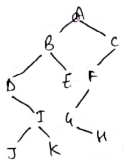


# of func^n calls
= n + [n+1]
= 2n+1





In order ⇒ ~~A B C D~~ ~~E F G H~~ I J K
D J I K B E A G H F C



Pre order ⇒ A B D I J K E C F G H

Post order ⇒ J K I D E B H G F C A

***

→ If [In order & Pre order] or (In order & Post order) sequence given, can construct unique binary tree.

→ If [Pre order & post order] sequence given, not possible to construct unique binary tree.

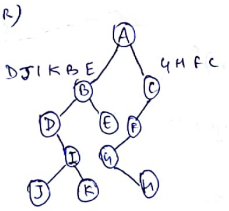
→ In-order sequence of Binary Search tree is ascending order of keys

→ If pre order or post order of binary search tree given, can construct unique binary tree.

⇒ Expression Tree:

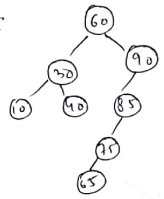
In order sequence → Infix expression
Pre order sequence → Prefix expression
Post order sequence → Post fix expression.

Q. What is post order sequence for binary tree with ~~In order~~:
In order: D J I K B E A G H F C (LNR)
pre order: A B D I J K E C F G H (NLR)
↑ root of binary tree



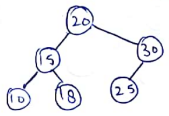
D J I K B E G H F C

Eg.

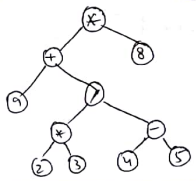


In order: 10, 30, 40, 60, 65, 75, 85, 90
(Ascending order)

Q. What is post order sequence of BST with given pre order.
20, 15, 10, 18, 30, 25.
In order = 10, 15, 18, 20, 25, 30
LRN Post order:
10, 18, 15, 25, 30, 20

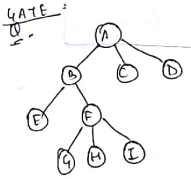


Expression Tree: (9+((2*3)/(4-5)))*8



IO ⇒ 9 + 2 * 3 / 4 − 5 * 8 [Infix]
Pre ⇒ * + 9 / * 2 3 − 4 5 8 [Prefix]
Post ⇒ 9 2 3 * 4 5 − / + 8 * [Postfix]

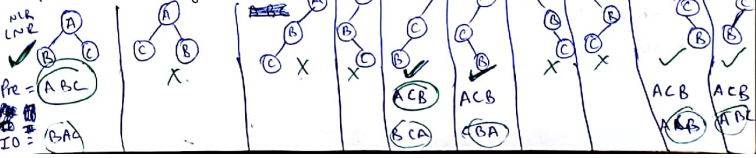
GATE:



In order: ① Traverse 1st subtree, Print Root, Traverse 2nd subtree, Traverse 3rd subtree
~~Traverse~~
E B G F H I A C D

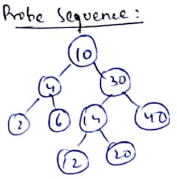
② Traverse 1st s.t., Traverse 2nd s.t., Print root, Traverse 3rd subtree
E G H F I B C A D

Q. If post order sequence is "BCA", How many different preorders possible for given post order.
2 different ~~post or~~ pre orders possible



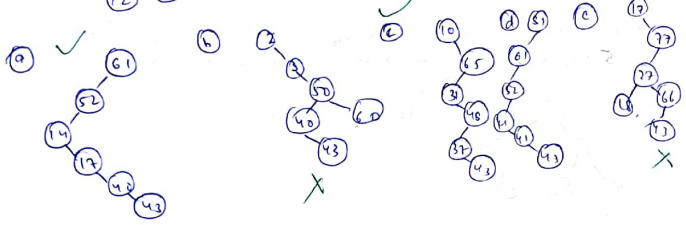
P97
WB

Q22. Probe sequence:

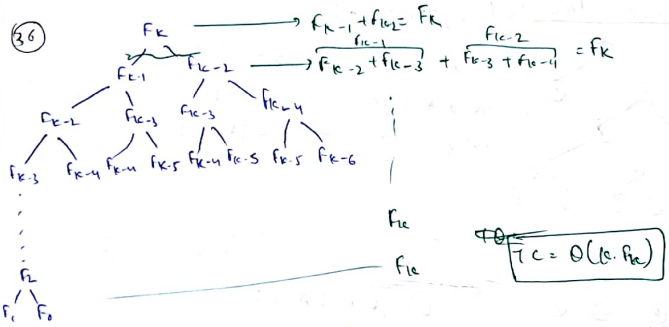


x = 12

10, 30, 14, 12 → Probe sequence for 12



(36)



$F_k \rightarrow F_{k-1} + F_{k-2} = F_k$

$F_{k-1} \rightarrow F_{k-2} + F_{k-3}$

$F_{k-2} \rightarrow F_{k-3} + F_{k-4}$

$= f_k$

F_k

F_{k-1} F_{k-2}

F_{k-2} F_{k-3} F_{k-3} F_{k-4}

F_{k-3} F_{k-4} F_{k-4} F_{k-5} F_{k-4} F_{k-5} F_{k-5} F_{k-6}

F_2

F_1 F_0

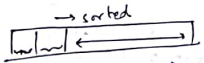
F_k

F_k

$TC = O(k \cdot F_k)$

$$T(F_k) = \begin{cases} T(F_{k-1}) + T(F_{k-2}) + F_k & ; k > 1 \\ O(1) & ; k \leq 1 \end{cases}$$

(39) → sorted

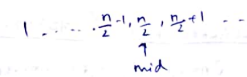


any two element sum < 1000 exist or not?

```
if (a[1] + a[2] < 1000)
    return (True)
else return (false)
```

40. ~~Take~~ cube root of n is largest m s.t. $m^3 \leq n$

Given	return
n = 25	2
n = 500	7



$\frac{n}{2}$ $\frac{n}{2}$ $\approx \frac{n}{2}$

$\log n$ $\log n$

$(\log n)^2$

$mid^3 \approx \log^2 n$

```
i = 1 j = n
while (i ≤ n)
{ mid = ⌊(i + j)/2⌋
  if (mid^3 ≤ n && (mid-1)^3 ≤ n)
      low = mid
  else if (mid^3 > n && (mid+1)^3 > n)
      high = mid
  else if (mid^3 > n && (mid-1)^3 ≤ n)
      return mid;
}
```

$TC = O(\log^3 n)$

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

(a) $= O(n)$ ✓ $\neq O(n^5)$ ✗

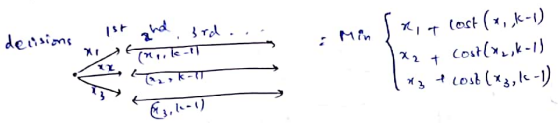
(b) $= O(n^{0.5})$ ✓ $\neq O(\log^k n)$ ✗

(c) $= O(\log^4 n)$ ✓ $\neq O(\log\log n)^5$ ✓

DYNAMIC PROGRAMMING: 03/09/2017:

→ Also called Principle of optimality design technique.

→ Whatever cost of first decision, cost of remaining decisions along with first decision must be optimal.



Greedy Algo	Dynamic Programming
→ Chooses one feasible solution using predefined method return optimal solution.	→ Computes all feasible solutions and returns optimal solution.
→ Runs in polynomial time complexity.	→ May require exponential time complexity. [almost polynomial TC.]
→ Fails to solve some optimization problems. Eg. 0/1 Knapsack problem.	→ Solves every optimization problem because uses principle of optimality.

Brute Force Algo	Dynamic Programming
→ Computes all feasible solution and returns optimal.	→ Computes all feasible solution such that avoids recomputation and returns optimal solution.
→ If # of solutions exponential [$2^n, n!$] Then TC of solution using brute force Algo also exponential.	→ Even exponential feasible soln problem may solve in polynomial time because of avoiding recomputation.
→ Less space complexity.	→ More space complexity.

Step to Design Dynamic Programming:

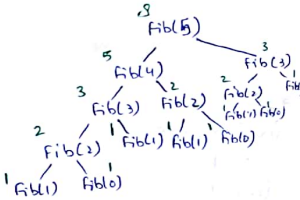
1. Solution of problem be formulated recursively. [To avoid recomputation]
2. Formulate Recurrence Relation for solution of problem.
3. Solve Recurrence Relation for given input in Bottom-Up approach.
   - return optimal solution.
   - return optimal sequence.

① n^{th} Fib Number:

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

Algo Fib(n)
{ if (n ≤ 1)
  return(1);
 else
  return (Fib(n-1) + Fib(n-2));
}

n levels

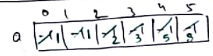


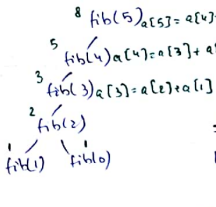
# of func^n calls = $1 + 2 + 2^2 + 2^3 + \ldots + 2^{n-3} + 2^{n-2} + 2^{n-1}$
$= 2^n - 1$

TC = $O(2^n)$ SC = $O(n)$ // Depth of recursion.

* Dynamic Solution:

Algo nthFib(n)
{ for (i=0; i≤n; i++)
 {a[i] = -1;}
 Fib(n)
 { if (n ≤ 1)
  { a[n] = 1;
   return (1);
  }
  else
  { if (a[n-1] == -1)
   {a[n-1] = Fib(n-1);}
   if (a[n-2] == -1)
   {a[n-2] = Fib(n-2);}
   a[n] = a[n-1] + a[n-2];
  return (a[n]);





# of function calls = n+1
[Each func^n call required constant time]

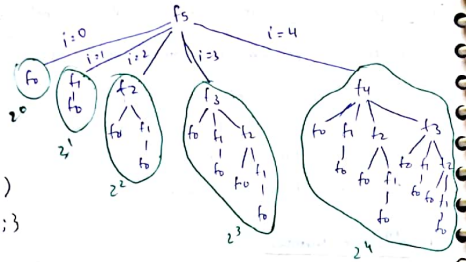
TC of algo = O(n) // polynomial

SC of Algo:
a[0..n] (n+1) entries array m/m units
Stack : n entries
O(n)

WB Pg 12

Q.61. double foo(int n)

```
{
  if(n==0) return 1;
  else
  {sum=0;
   for(i=0; i<n; i++)
   {sum = sum + foo(i);}
   return(sum);
  }
}
```



No. of func^n calls = $2\left[1+2+2^2+\ldots+2^{n-1}\right]+1$

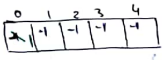
$= 2^n$

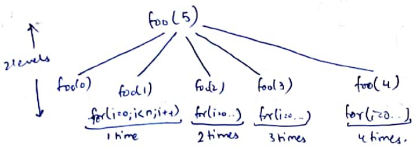
TC = $\theta(2^n)$

SC = $\theta(n)$

Q.62

```
for(i=0; i<n; i++)
  a[i] = -1;
foo(n)
{ if (n==0)
  {a[0]=1;
   return(1)
  }
  else
  { sum=0
    for(i=0; i<n; i++)
    { if(a[i]==-1)
      {a[i] = foo(i);}
      sum = sum + a[i];
    }
   return(sum)
}
```





# of func^n calls = $(n+1)$

TC of Algo = $[n+1]+\{1+2+\ldots+(n-2)+(n-1)\}$

$= \theta(n^2)$

SC of algo:

a[0 ... n-1] = $\theta(n)$

Stack space = $\theta(1)$ // 2 entries

$\theta(n)$

Matrix Chain Problem:

→ $A_{p\times q}$ $B_{q\times r}$ cost of multiplication = p·q·r (element multiplication required)

→ Matrix Multiplication Associativity: [A, B, C, Matrices]

$(A\times B)\times C \equiv A\times(B\times C)$

* But no. of multiplications differ.

$A_{100\times 50}$, $B_{50\times 75}$, $C_{75\times 10}$

① Cost of $(A*B)*C$:

$100*50*75 = 375000$

$100*75*10 = 75000$

450000 (Scalar mult.)

② Cost of $A*(B*C)$

$50*75*10 = 37500$

$100*50*10 = 50000$

87500 (Scalar mult.)

optimal Mult. of A, B, C matrix

Problem Statement:

Given n matrices:

$A_1\ A_2\ A_3\ \ldots\ A_n$ and $<P_0\ P_1\ P_2\ P_3\ \ldots\ P_n>$ order set such that cost of multiplication of n-matrices is minimum.

$P_0\ P_1\ P_2\ P_3\ P_4$

$A_1\ A_2\ A_3\ A_4$ $\quad <10, 5, 3, 6, 7>$

$A_1*(A_2*A_3)*A_4$

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

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

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

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

5 different parenthesizations required.

# of Matrices	# of parenthesizations
1. A_1	1
2 $A_1\ A_2$	1
3 $A_1\ A_2\ A_3$	2
4 $A_1\ A_2\ A_3\ A_4$ $2+1+1+2+1$	5
5 $A_1\ A_2\ A_3\ A_4\ A_5$ $5+1+2+1+5+1$	14
n	$\frac{1}{n}\left[\frac{(2n-2)!}{(n-1)!\,(n-1)!}\right]$

$T(n)$ = # of parenthesizations for n matrices

$$T(n) = \begin{cases} 1 & ; n \le 2 \\ \sum_{k=1}^{n-1} T(k) \cdot T(n-k) & ; n > 2 \end{cases}$$

$A_1 A_2 A_3 A_4 \ldots A_{n-1} A_n$

$$T(n) = \underset{k=1}{T(1)} \cdot T(n-1) + \underset{k=2}{T(2)} \cdot T(n-2) + \ldots + \underset{k=n-1}{T(n-1)} \cdot T(n)$$

$$T(n) = \frac{1}{n}\left[\frac{(2n-2)!}{(n-1)!\,(n-1)!}\right]$$ → # of feasible solution for matrix chain problem.

Brute Force Algo ⇒ $TC = \Omega(2^n)$

DP Solution [recursive solution] of matrix chain problem: $TC = \Theta(n^3)$

$A_1 \; A_2 \; A_3 \ldots A_n$ are n matrices

$\langle P_0 \; P_1 \; P_2 \ldots P_n \rangle$ are order set

Assume $c(1,n)$ is MinCost of Matrices.

$A_1 \; A_2 \; \ldots \; A_n$

first parenthesization at kth matrix

$(A_1 \; A_2 \ldots A_k) * (A_{k+1} \ldots A_n)$

$P_0 \times P_k$; $P_k \times P_n$; $A_1 \ldots A_k$: $P_0 \times P_k$; $A_{k+1} \ldots A_n$: $P_k \times P_n$

MinCost of Mul of $A_1 \ldots A_k$ is $C(1,k)$

Cost of Mul $A_{k+1} \ldots A_n$ matrices is $C(k+1, n)$

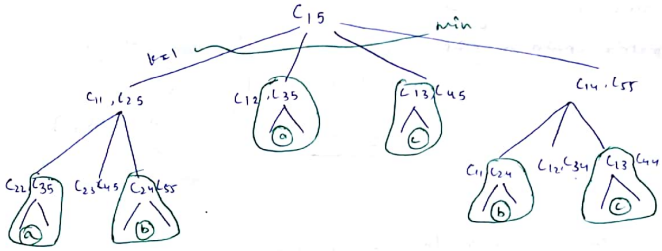
$$c(1,n) = \min_{1 \le k < n} \left\{ c(1,k) + c(k+1,n) + P_0 \cdot P_k \cdot P_n \right\}$$

* General Recurrence Relation of Matrix chain:

$A_i \; A_{i+1} \ldots A_j$ Matrices

$\langle P_{i-1} \; P_i \; P_{i+1} \ldots P_j \rangle$ Order set

$$C(i,j) = \begin{cases} \min_{i \le k < j} \left\{ c(i,k) + c(k+1,j) + P_{i-1} \cdot P_k \cdot P_j \right\} & ; i < j \\ 0 & ; i = j \end{cases}$$



e.g. $A_1 \; A_2 \; A_3 \; A_4$

$\langle P_0 \; P_1 \; P_2 \; P_3 \; P_4 \rangle$

$\langle 40 \; 50 \; 25 \; 60 \; 20 \rangle$

$C(1,4)$ = Final Value ?

$$C_{12} = \min_{1 \le k < 2} \{ k=1 \; C_{11} + C_{22} + P_0 P_1 P_2 = 0 + 0 + 50000 \}$$

$$C_{13} = \min_{1 \le k < 3} \begin{cases} k=1 & C_{11} + C_{23} + P_0 P_1 P_3 = 0 + 75000 + 120000 \\ k=2 & C_{12} + C_{33} + P_0 P_2 P_3 = 50000 + 0 + 60000 \end{cases}$$

$$C_{24} = \min_{2 \le k < 4} \begin{cases} k=2 & C_{22} + C_{34} + P_1 P_2 P_4 = 0 + 30000 + 25000 = 55000 \\ k=3 & C_{23} + C_{44} + P_1 P_3 P_4 = 75000 + 0 + 60000 = 135000 \end{cases}$$

$$C_{14} = \min_{1 \le k < 4} \begin{cases} k=1 & C_{11} + C_{24} + P_0 P_1 P_4 = 0 + 55000 + 40000 = 95000 \\ k=2 & C_{12} + C_{34} + P_0 P_2 P_4 = 50000 + 30000 + 20000 = 100000 \\ k=3 & C_{13} + C_{44} + P_0 P_3 P_4 = 110000 + 0 + 48000 = 158000 \end{cases}$$

	1	2	3	4
1	0	50000	110000	95000
2	X	0	75000	55000
3	X	X	0	20000
4	X	X	X	0

≃ $\frac{n^2}{2}$; $\Theta(n^2)$

Final val ; $\Theta(n)$; Top

$|i-j| = 3$

$|j-i| = 2$

$|j-i| = 1$

→ Initial Val

Bottom

→ Optimal cost of Mul. of $A_1 A_2 A_3 A_4$ matrices = 95000 (Scalar multiplications)

→ Optimal parenthesization of matrix mul of $A_1 A_2 A_3 A_4$:

$A_1 (A_2 \; A_3 \; A_4)$ → k value of C_{14}

$(A_2 \; (A_3 \; A_4))$ → k value of C_{24}

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

TC of matrix chain using DP = $\Theta(n^3)$

SC of matrix chain using DP = c[1..n][1..] // to store c(i,j)
k[1..n][1..] // to store k^{th} value of min. cost of c[i,j]

$\Theta(n^2)$

$$T(n) = \begin{cases} 1 & n \le 1 \quad ; n \le 1 \\ \sum_{k=1}^{n} T(k-1) * T(n-k) & ; n > 1 \end{cases}$$

$$T(n) = \frac{1}{n+1}\left[\frac{(2n)!}{n!.n!}\right]$$

# of BSTs possible for n distinct keys. $\frac{2^n C_n}{n+1}$

→ No. of parenthesizations for (n+1) matrices = # of BST for n distinct keys.

① # No. of Binary Search Tree for n distinct keys:

# of distinct keys	# of BSTs
0	1
1	1
2	2
3	5
4	14

3. $1 \times 2 + 1 \times 1 + 2 * 1$

4. $k_1 < k_2 < k_3 < k_4$

$1 * 5 + 1 \times 2 + 2 * 1 + 5 \times 1$

$T(n)$ = # of BST's for n distinct key.

$k_1\ k_2\ k_3\ \ldots\ k_{n-1}\ k_n$

$T(0) = 1$, $T(1) = 1$ — Initial value

$T(n) = T(0).T(n-1) + T(1)T(n-2) + T(2)T(n-3) + \ldots + T(n-1).T(0)$

② No. of unlabled Binary Trees for n nodes:

# of Nodes	# of unlabled BTs
1	1
2	2
3	5
:	:
n	$\frac{1}{n+1}\left[\frac{(2n)!}{n!.n!}\right]$

# of unlabled BTs for 'n' nodes = # of BSTs for 'n' distinct keys.

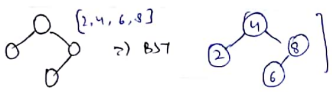
③ No. of labled BTs for n keys:

No. of nodes	# of labled BTs
1.	1
2.	$2 * 2! = 4$
3.	$5 \times 3! = 30$
n	$\frac{n!}{n+1}\left[\frac{(2n)!}{n!.n!}\right]$

No. of labled BTs for n keys = n!. [No. of unlabled BTs]

Q. How many ways can label given unlabled binary tree of n-nodes && n distinct keys into Binary search tree.

(a) $n!$ (b) $\frac{1}{n+1}\left[\frac{(2n)!}{n!}\right]$ (c) $\frac{1}{n+1}\left[\frac{(2n)!}{n!.n!}\right]$ ~~(d)~~ 1



only 1 way.

Largest Common Subsequence Problem [LCS Problem]:

→ Sub Sequence: Sequence of char from string which may not be consecutive.

X = "a b a d e"

Some subsequence of X: [a b d, a b d e, a a e, a b a d e]

String x with n characters.
# of possible subsequence = 2^n
X => "a b c d"
$2^4 = 16$ possible subsequences of X

→ Common Subsequence: Subsequence which is common for two strings X & Y.

X = a b a d e
Y = b a a d a

Some common subsequences of strings X & Y: [a a d, b a d, b d, a d]
Largest Common subsequences of X & Y = Length(3)

→ Largest Common Subsequence: Common subsequence of X & Y, which is maximum length.

Recursive Solution of LCS:

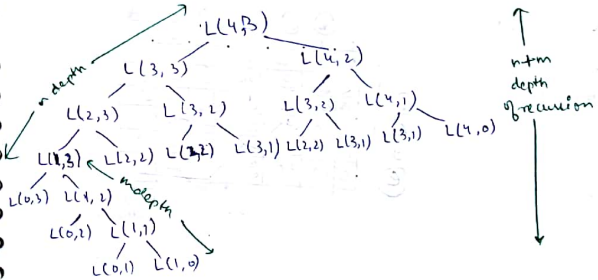
LCS(n,m): Largest common subsequence length of two strings. n & m char. respectively.

$X = [X_1\ X_2\ X_3\ \ldots\ X_{n-1}\ X_n]$
$Y = [Y_1\ Y_2\ Y_3\ \ldots\ Y_{m-1}\ Y_m]$

$$LCS(n,m) = \begin{cases} 0 & , n=0 \text{ || } m=0 \\ 1+LCS(n-1,m-1) & , n>0 \text{ \&\& } m>0 \text{ \&\& } X_n == Y_m \\ \max\{LCS(n-1,m),\ LCS(n,m-1)\} & , n>0 \text{ \&\& } m>0 \text{ \&\& } X_n \neq Y_m \end{cases}$$

$$L(i,j) = \begin{cases} 0 & ; i=0 \text{ || } j=0 \\ 1+L(i-1,j-1) & ; i>0 \text{ \&\& } j>0 \text{ \&\& } X_i == Y_j \\ \max\{L(i-1,j),\ L(i,j-1)\} & ; i>0 \text{ \&\& } j>0 \text{ \&\& } X_i \neq X_j \end{cases}$$

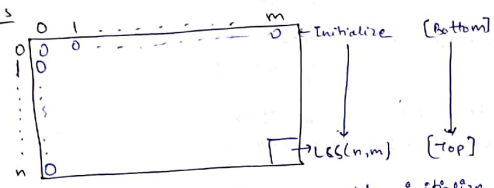
↳ General Recurrence Relation for LCS.
L(n,m) = ? //Final value.



of f^n calls $\approx 2^{n+m}$
TC of LCS using rec soln = $O(2^n \cdot 2^m)$
SC = $O(n+m)$

DP implementation of LCS:

X & Y are two strings of n & m characters.



⇒ 1st row & 1st column of LCS should initialize 0's

	X_{j-1}	Y_j
	$j-1$	j
X_{i-1} $i-1$	$L(i-1,j-1)$	$L(i-1,j)$
Y_i i	$L(i,j-1)$	$L(i,j)$

```
if (Xi == Yj)
    L(i,j) = 1 + L(i-1, j-1)
else if (L(i,j-1) ≥ L(i-1,j))
    L(i,j) = L(i,j-1)
else
    L(i,j) = L(i-1,j)
```

→ LCS can compute either in row major order or column major order.

Eg X = [a b c d e] n=5

Y = [b c a e c d] m=6

LCS(5,6) = 3

[b c e]

		0	1 (b)	2 (c)	3 a	4 (e)	5 c	6 d
	0	0	0	0	0	0	0	0
a	1	0	0 ←	0 ←	1 ↖	1 ←	1 ←	1 ←
(b)	2	0	1 ↖	1 ←	1 ←	1 ←	1 ←	1 ←
(c)	3	0	1 ↑	2 ↖	2 ←	2 ←	2 ↖	2 ←
d	4	0	1 ↑	2 ↑	2 ←	3 ↖	3 ←	3 ←
(e)	5	0	1 ↑	2 ↑	2 ←	3 ↑	4 ↖	4 ←

Final value

```
Algo Lcs(X[], Y[], n, m)
{
  for (i=0; i≤m; i++)
  { L[0][i] = 0; }
  for (i=0; i≤n; i++)
  { L[i][0] = 0; }
  for (i=1; i≤n; i++)
  {  for (j=1; j≤m; j++)
     {
        if (X[i] == Y[j])
        { L[i][j] = 1 + L[i-1][j-1]
          Path[i][j] = "↖"
        }
        else if (L[i][j-1] ≥ L[i-1][j])
        { L[i][j] = L[i][j-1];
          Path[i][j] = "←";
        }
        else
        { L[i][j] = L[i-1][j];
          Path[i][j] = "↑"
        }
     }
  }
  return (L(n,m), Path[][])
}
```

TC of LCS using DP = Θ(n×m)

SC of LCS using DP = Θ(n×m)

L[0...n, 0...m] = (n+1)(m+1)

Path[1...n, 1...m] = n×m

Θ(n×m)

```
Algo Print(n,m) //to print LCS
  //path[][], X[] or Y[]
{ if (n==0 || m==0)
    return;
  else
  { if (path[n][m] == "↖")
    { Print (n-1, m-1);
      printf ("X(n)");
    }
    else if (path[n][m] == "←")
    { Print (n, m-1); }
    else
    { Print (n-1, m);
    }
  }
}
```

→ TC = Θ(n+m)

SC = Θ(n+m)

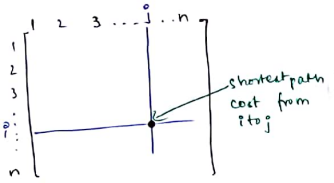
* TC to print LCS using path array is Θ(n+m).

All pair shortest path Algo:

Graph G
cost [1..n][1..n] edge costs
cost(i,j) : 0 , if i=j
Edge cost , if (i,j) ∈ Edge
∞ , if (i,j) ∉ Edge
} Given

* Compute A[1..n][1..n) such that A[i][j]: shortest path cost from vertex i to vertex j.

~~Greedy Algo~~



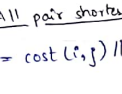
Greedy Algo [Dijkstra]:

```
for (i=1; i≤n; i++)
{A[i][1..n] Dijkstra(G,n,e,i)}
                          source
```

TC = (n² + n.elogn)

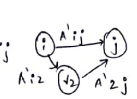
DP All pair shortest path Algo: [Floyd Warshal's Algo]

A^0_{ij} = cost(i,j) // Edge cost
Initial value.

① A^1_{ij} 

$A^1_{ij} = \min\{$ i→j or i→1→j $\}$

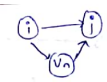
$A^1_{ij} = \min\{A^0_{ij}, A^0_{i1} + A^0_{1j}\}$ // shortest path from i to j going through 1.

② A^2_{ij} 

$A^2_{ij} = \min\{A^1_{ij}, A^1_{i2} + A^1_{2j}\}$

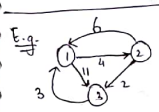
$A^2_{ij} = \min\{$ i→j , i→1→j , i→2→j , i→1→2→j , i→2→1→j $\}$

// shortest path cost from i to j going through 1 and/or 2.

$n \rightarrow A^n_{ij} = \min\{A^{n-1}_{ij}, A^{n-1}_{in} + A^{n-1}_{nj}\}$ 

* Recurrence Relation:

$$A^n_{ij} = \begin{cases} \min\limits_{1 \le k \le n} A^{k-1}_{ij}, A^{k-1}_{ik} + A^{k-1}_{kj} \\ A^0_{ij} = \text{cost}_{ij} \text{ // Edge costs.} \end{cases}$$

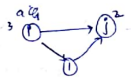
E.g. 

cost
	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

Soln:

$A^0 =$
	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

$A^1 =$
	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0



$A^2 =$
	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

$A^3 =$
	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

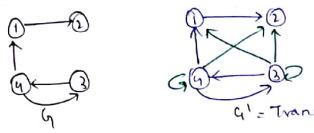
Result of All pair shortest path.

Algo APSP_FloydWarshal's(n, cost[ ][ ])

```
{ for (i=1; i≤n; i++)                    } θ(n²)
  { for (j=1; j≤n; j++)                  }
    A[i][j] = cost[i][j];
}
for (k=1; k≤n; k++)                      } θ(n³)
{ for (i=1; i≤n; i++)                    }
  { for (j=1; j≤n; j++)                  }
    { if (A[i][j] > A[i][k] + A[k][j])   }
      { A[i][j] = A[i][k] + A[k][j];     }
    }
  }
}
```

TC = θ(n³)
SC = θ(n²)

Transitive closure of Graph:



G^1 = Transitive Closure

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

→

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

$A^0_{ij} = Adj[i][j]$

① $A^1_{ij} = A^0_{ij} \vee (A^0_{i1} \wedge A^0_{1j})$

② $A^1_{ij} = A^0_{ij} \vee (A^0_{i1} \wedge A^0_{1j})$

A^1_{ij} is 1 ⇒ if there exist path from i to j going through ①

② $A^2_{ij} = A^1_{ij} \vee (A^1_{i2} \wedge A^1_{2j})$

$n \Rightarrow A^n_{ij} = A^{n-1}_{ij} \vee (A^{n-1}_{in} \wedge A^{n-1}_{nj})$

$A^1 =$

	1	2	3	4
1				
2				
3				
4				

```
Algo TC4 - FloydWarshal's(n, Adj[][])
{
  for(i=0; i≤n; i++)
  { for(j=1; j≤n; j++)
      Aij = Adj[i][j];
  }
  for(k=1; k≤n; k++)
  { for(i=1; i≤n; i++)
    { for(j=1; j≤n; j++)
      { Aij = Aij ∨ (Aik ∧ Akj)
      }
    }
  }
  return (A[][])
}
```

0/1 Knapsack Problem:

→ Given 'n' object and m capacity knapsack && $P_1 .. P_n$ profits $W_1 .. W_n$ weights

Maximize $\sum_{i=1}^{n} P_i * W_i$

subjected to $\sum_{i=1}^{n} W_i * X_i \leq m$; where $X_i = 1$ or 0.

* 0/1 knapsack problem failed to solve using greedy method.
* 0/1 knapsack problem TC using Brute force Algo $O(2^n)$

DP solution of 0/1 knapsack Problem:

V(n,m) = Max profit for n objects and m capacity knapsack.

$V(n,m) = \max\{V(n-1,m), V(n-1, m-w_n)+P_n\}$

$\leftarrow X_n = 0 \rightarrow$ $\leftarrow X_n = 1 \rightarrow$

V(i,j) = Max profit for i objects & j knapsack capacity.

$$V(i,j) = \begin{cases} 0 & ; i=0 \,\&\&\, j \geq 0 \\ V(i-1,j) & ; i>0 \,\&\&\, w_i > j \;\; // \text{Weight of object more than knapsack free space.} \\ \max\{V(i-1,j),\ V(i-1,j-w_i)+P_i\} & ; i>0 \,\&\&\, w_i \leq j \end{cases}$$

V(n,m) = final (val)

Eg: n=3 m=6

$\langle P_1\ P_2\ P_3 \rangle = \langle 20, 10, 15 \rangle$

$\langle W_1, W_2, W_3 \rangle = \langle 2, 1, 4 \rangle$

V	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	20	20	20	20	20
2	0	10	20	-	-	-	-
3	0						

- define V[0..n, 0..m]
- Initialize 1st row 0's.

V(1,0) = V(0,0) $w_1 > 1$

V(1,2) = max{V(0,2), V(0,0)+20}

V(2,1) = max{V(1,1), V(1,0)+10}

V(2,2) = max{V(1,2), V(1,1)+10}

Algo 01knapsack (n, m, w[1..n], p[1..n])

```
{
  //v[0..n, 0..m] define array
  for (i=0; i≤m; i++)                    O(m)
    { v[0][i]=0 }
  for (i=1; i≤n; i++)                    O(n)
    for (j=0; j≤m; j++)                  O(m+1)
      if (wi>j)
        v[i][j]=v[i-1][j] ; x[i]=0
      else
        v[i][j]=max{v[i-1][j], v[i-1][j-wi]+pi};
                    xi=0       xi=1
  return (x[ ], v[n][m])
}
```

x[ ] → object keep. v[n][m] → Max Profit

* TC of 01 knapsack using DP = ~~O(n)~~ $\theta(n*m)$
* SC of 01 knapsack using DP = $\theta(n*m)$

TC : 01 knapsack

	Brute Force $O(2^n)$	DP $\theta(n*m)$
n=10 & m=1000	1024 comp.	10000
n=20 m=10000	2^{20}	200000 ✓

Sum of Subset Problem : (SOS :

$[a_1, a_2, a_3 \ldots a_n]$: n Integers.
Test any subset of given n Integers sum equal to "m" exists/not exists.

E.g. n=4 $\{a_1, a_2, a_3, a_4\} = \{5, 4, 8, 6\}$ & m=16

sos(n,m) = False

m=15 ⇒ sos(n,m) = True

→ Greedy method fails to solve SOS Problem.

→ Brute Force Algo to solve SOS problem ⇒ $O(2^n)$ TC.

→ DP soln of SOS :

sos(n,m) = True / False
any subset of n elements sum equal to m exists/not exists.

$sos(n,m) = sos(n-1,m) \vee (sos(n-1, m-a_n) \wedge m \geq a_n)$

$$sos(i,j) = \begin{cases} \text{True} & ; j=0 \ \&\& \ i \geq 0 \\ \text{False} & ; j>0 \ \&\& \ i=0 \\ sos(i-1,j) & ; j < a_i \\ sos(i-1,j) \vee (sos(i-1, j-a_i) \wedge j \geq a_i) \end{cases}$$

E.g- n=3 m=7

$[a_1, a_2, a_3] = [3\ 2\ 5]$

	0	1	2	3	4	5	6	7
0	T	F	F	F	F	F	F	F
1	T	F	F	T	F	F	F	F
2	T	F	T	T	F	T	F	F
3	T	F	T	T	F	T	F	(T)

- First column initialize True.
- First row S[0][1..n] false.

sos(1,2) = sos(0,2) j<ai

sos(2,2) = sos(1,2) ∨ sos(1,0)
True = False ∨ True

```
Algo sos(n, m, a[1...n])
{ //sos [0...n, 0...m] define array.
  for(i=0; i≤n; i++)
      sos[i][0] = True
  for(i=1; i≤m; i++)
  { sos[0][i] = False}
  for(i=1; i≤n; i++)
  { for(j=0; j≤m; j++)
    { if(ai>j)
        sos[i][j] = sos[i-1][j];
      else
        sos[i][j] = sos[i-1][j] v sos[i-1][j-ai]
    }
  }
  return(sos[n][m])
}
```

TC of sos = $\Theta(n*m)$

SC of sos = $\Theta(n*m)$