ALGORITHMS

ALGORITHMS

Algorithm	Program
① Deals with design	① Implementation
② Anyone with domain knowledge can design.	② Programmer
③ any Lang	③ programming Lang.
④ H/W & OS independent	④ H/W & OS dependent
⑤ Analyze	⑤ Testing

PRIORI ANALYSIS	POSTERIORI TESTING
1. Deals with the algorithm	1. Deals with the program.
2. Independent of Language	2. Language dependent
3. Hardware independent	3. Hardware dependent
4. Time and Space complexity	4. Watch time and bytes

Characteristics of Algorithm

1. Input – 0 or more
2. Output – atleast 1 output
3. Definiteness
4. Finiteness
5. Effectiveness

How to write an Algorithm

```
Algorithm swap(a,b)
{
    temp ← a;   ——— 1
    a ← b;      ——— 1
    b ← temp;   ——— 1
}
```

$f(n) = 3$ | $O(1)$

Space

a — 1
b — 1
temp — 1

$S(n) = 3$ | $O(1)$

How to analyze an algorithm

1. Time
2. Space
3. N/w consumption
4. Power consumption
5. CPU Registers

Frequency Count Method

1.
```
Algorithm sum(A,n)
{
    s = 0;                          ——— 1
    for (i=0; i<n; i++)             ——— n+1
    {
        s = s + A[i];               ——— 1
    }
    return s;
}
```

$f(n) = n+3$

$O(n)$

A

8	3	9	7	2
0	1	2	3	4

$n = 5$

Space

A — n
n — 1
s — 1
i — 1

$S(n) = n+3$

$O(n)$

2. Algorithm Add(A, B, n)

```
{
    for (i=0; i<n; i++)          ——— n+1
    {
        for (j=0; j<n; j++)      ——— n x (n+1)
        {
            c[i,j] = A[i,j] + B[i,j];   ——— n x n
        }
    }
}
```

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

$O(n^2)$

Space

A	n^2
B	n^2
C	n^2
n	1
i	1
j	1

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

$O(n^2)$

Space

A	n^2
B	n^2
C	n^2
n	1
i	1
j	1
k	1

$S(n) = 3n^2 + 4$

$O(n^2)$

3. Algorithm Multiply(A, B, n)

```
(n+1)   {   for (i=0; i<n; i++)
n(n+1)      {   for (j=0; j<n; j++)
n x n           {   c[i,j] = 0
n x n x (n+1)       for (k=0; k<n; k++)
n x n x n           {   c[i,j] = c[i,j] + A[i,k] * B[k,j];
                    }
                }
            }
```

$f(n) = 2n^3 + 3n^2 + 2n + 1$

$O(n^3)$

① `for (i=0; i<n; i++)`
{ `statement;` — n
$O(n)$
}

`for (i=1; i<n; i=i+2)`
{ `stmt;` — $\frac{n}{2}$
}
$f(n) = \frac{n}{2}$ $O(n)$

② `for(i=0; i<n; i++)` — $n+1$
{ `for(j=0; j<n; j++)` — $n \times (n+1)$
{ `stmt;` — $n \times n$
}
$O(n^2)$
}

③ `for (i=0; i<n; i++)`
{ `for (j=0; j<i; j++)`
{ `stmt;`
}
}

i	j	no. of times
0	0 ✗	0
1	0 ✓	1
	1 ✗	
2	0 ✓	2
	1 ✓	
	2 ✗	
3	0 ✓	3
	1 ✓	
	2 ✓	
	3 ✗	

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

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

④ `p=0;`
`for (i=1; p<=n; i++)`
{ `p = p+i;`
}

i	p
1	0+1=1
2	1+2=3
3	1+2+3
4	1+2+3+4
⋮	
k	$1+2+3+4+\cdots+k$

stopping condition

assume $p > n$

$p = \frac{k(k+1)}{2}$

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

$k^2 > n$

$k > \sqrt{n}$

$O(\sqrt{n})$

⑤ `for (i=1; i<n; i= i*2)`
`{ stmt; }` —— $\log n$ times

i
1
$1\times2=2$
$2\times2=2^2$
$2^2\times2=2^3$
⋮
2^k

Assume $i \geq n$

$\therefore i = 2^k$

$\therefore 2^k \geq n$

$2^k = n$

$k = \log_2 n$

$O(\log_2 n)$

eg.

$n=8$

i	
1	✓
2	✓
4	✓
8	✗

③

$n=10$

i	
1	✓
2	✓
4	✓
8	✓
16	✗

④

No. of times $= \lceil \log n \rceil$

$\log_2 8 = 3$

$\log_2 10 = 3.2$

⑥ `for (i=n; i>=1; i= i/2)`
`{ stmt; }`

i
$n/2^0$
$n/2^1$
$n/2^2$
$n/2^3$
⋮
$n/2^k$

Assume $i < 1$

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

$\therefore \frac{n}{2^k} < 1$

$n < 2^k$

$2^k = n$

$k = \log_2 n$

$O(\log_2 n)$

⑦
```
for (i=0; i*i<n; i++)
{
    stmt;
}
```

$i * i < n$

$i * i >= n$ STOP

$i^2 = n$

$i = \sqrt{n}$

⑧
```
for (i=0; i<n; i++)
{
    stmt;        ——— n
}
for (j=0; j<n; j++)
{
    stmt;        ——— n
}
```
$2n$

$O(n)$

⑨
```
p=0;
for(i=1; i<n; i=i*2)
{
    p++;         ——— log n
}
for (j=1; j<p; j=j*2)
{
    stmt;        ——— log p
}
```
$p = \log n$

$\log(\log n)$

$\log n$

⑩
```
for (i=0; i<n; i++)          ——— n
{
    for (j=1; j<n; j=j*2)    ——— n × log n
    {
        stmt;                ——— n × log n
    }
}
```
$2n \log n + n$

$O(n \log n)$

Note:

```
for(i=0; i<n; i++)  ——  O(n)
for(i=0; i<n; i=i+2)  ——  n/2 O(n)
for(i=n; i>1; i--)  ——  O(n)
for(i=1; i<n; i=i*2)  ——  O(log2 n)
for(i=1; i<n; i=i*3)  ——  O(log3 n)
for(i=n; i>1; i=i/2)  ——  O(log2 n)
```

TIME COMPLEXITY OF IF AND WHILE

```
for i:=1 to n do step 1  ——  n+1
{
    stmt;  ——————  n
}
```

Time complexity is surely n.

```
while (condition)
{
    stmt;
}
```

We can't directly conclude that time complexity is n.

```
do
{
    stmt;
} while (condition)
```

executed atleast once

```
repeat
{
    stmt;
} until (condition);
```

repeated until condition is true.

Analysis of if & while

①
```
i=0;          ——— 1
while (i<n)   ——— n+1
{
    stmt;     ——— n
    i++;      ——— n
}
```
$f(n) = 3n+2$

$O(n)$

②
```
for (i=0; i<n; i++)   ——— n+1
{                      i=0: *   i<n: n+1   i++: n
    stmt;  ——— n
}
```
$f(n) = 2n+1$

$O(n)$

③
```
a=1;
while (a<b)
{   stmt;
    a=a*2;
}
```

a
1
$1\times2 = 2$
$2\times2 = 2^2$
$2^2\times2 = 2^3$
⋮
2^k

Terminate when

$a \geq b$

$\because a = 2^k$

$2^k \geq b$

$2^k = b$

$k = \log_2 b$

$O(\log n)$

④
```
i=n;
while (i>1)
{   stmt;
    i=i/2;
}
```
Same ↔
```
for (i=n; i>1; i=i/2)
{   stmt;
}
```

i
n
$n/2$
$n/2^2$
⋮
$n/2^k$

Terminate when

$i \leq 1$

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

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

$\Rightarrow n = 2^k$

$k = \log_2 n$

$O(\log n)$

⑤

```
i=1
k=1;
while (k<n)
{
    stmt;
    k=k+i
    i++;
}
```

i	k
1	1
2	1+1 = 2
3	2+2
4	2+2+3
5	2+2+3+4
⋮	⋮
m	2+2+3+4+.....+m

$\frac{m(m+1)}{2}$ roughly

Terminate at $k \geq n$

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

$\Rightarrow m^2 \geq n$

$\Rightarrow m = \sqrt{n}$

$O(\sqrt{n})$

```
for (k=1, i=1; k<n; i++)
{
    stmt;
    k=k+i;
}
```

---

⑥

```
while (m!=n)
{
    if (m>n)
        m=m-n;
    else
        n=n-m;
}
```

m=16	n=2
14	2
12	2
10	2
8	2
6	2
4	2
2	2

Executed almost $\frac{16}{2}$ times

i.e $\frac{n}{2}$ times

$O(n)$ max

$O(1)$ min

```
for ( ; m!=n; )
{
    if (m>n)
        m=m-n;
    else
        n=n-m;
}
```

⑦ Algorithm Test (n)

```
{
  if (n<5)
  {
    printf("%d", n);  ——— 1
  }
  else
  {
    for (i=0; i<n; i++)
    {
      printf("%d", i);  ——— n
    }
  }
}
```

Best – O(1)

Worst – O(n)

So, if there is a conditional statement in an algorithm, then it MAY take different amount of time depending on the condition.

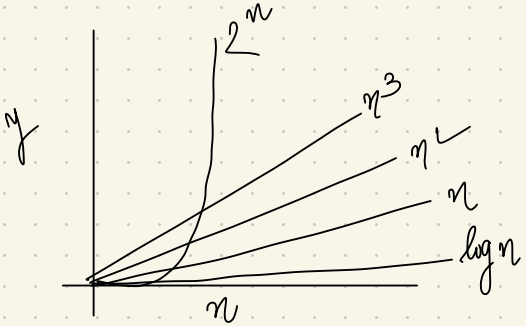
Types of time functions

- O(1) — constant
- O(log n) — Logarithmic
- O(n) — linear
- O(n²) — quadratic (Matrix addn.)
- O(n³) — Cubic (Matrix multiplication)
- O(2ⁿ), O(3ⁿ), O(nⁿ) — Exponentials

Weightage

$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \dots < 2^n < 3^n \dots < n^n$

log n	n	n^2	2^n
0	1	1	2
1	2	4	4
2	4	16	16
3	8	64	256
3.1	9	81	512



ASYMPTOTIC NOTATIONS

O	big-oh	upper bound
Ω	big-omega	lower bound
θ	theta	average bound

useful

Big-Oh

The function $f(n) = O(g(n))$ iff $\exists$ +ve constants c and n_0 such that $f(n) \leq c \cdot g(n)$ $\forall$ $n \geq n_0$

eg. $f(n) = 2n+3$

$2n+3 \leq 5n$ $\quad n \geq 1$ $\quad \therefore f(n) = O(n)$

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

OR

$2n+3 \leq 5n^2$ $\quad n \geq 1$

$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n^n$

Lower bounds — Average bound — Upper bounds

Omega

The function $f(n) = \Omega(g(n))$ iff $\exists$ +ve constants c and n_0 such that $f(n) \geq c \cdot g(n)$ $\forall$ $n \geq n_0$.

eg. $f(n) = 2n+3$

$2n+3 \geq 1 \times n$ $\forall$ $n \geq 1$

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

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

Theta Notation

The function $f(n) = \theta(g(n))$ iff $\exists$ +ve constants c_1, c_2 and n_0 such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

eg. $f(n) = 2n+3$

$1 \times n \leq 2n+3 \leq 5 \times n$ $n \geq 1$

$\therefore f(n) = \theta(n)$

Remarks: Do not attach the notations with the idea of best case or worst case. We can use any notation for best case and any notation for worst case.

eg. $f(n) = 2n^2 + 3n + 4$

$2n^2+3n+4 \leq 2n^2+3n^2+4n^2$

$2n^2+3n+4 \leq 9n^2$ $n \geq 1$

$9n^2$ → c

and $2n^2+3n+4 \geq 1 \times n^2$

and $1 \times n^2 \leq 2n^2+3n+4 \leq 9n^2$

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

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

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

eg. $f(n) = n^2 \log n + n$

$1 \times n^2 \log n \leq n^2 \log n + n \leq 10 n^2 \log n$

$O(n^2 \log n)$ $\Omega(n^2 \log n)$ $\theta(n^2 \log n)$

eg. $f(n) = n! = n \times (n-1) \times (n-2) \times \dots \times 3 \times 2 \times 1$

$1 \times 1 \times 1 \times \dots \times 1 \leq 1 \times 2 \times 3 \times \dots \times n \leq n \times n \times n \times \dots \times n$

$1 \leq n! \leq n^n$

$\Omega(1)$ $O(n^n)$

⇒ For small value of n, $n!$ is closer to 1 and for large value of n, $n!$ is closer to n^n. So, we can't fix a proper position for $n!$ in the weightage order.

⇒ Since we can't find θ-bound, upper & lower bounds become useful.

eg. $\log n!$

$\log(1 \times 1 \times 1 \times \dots \times 1) \leq \log(1 \times 2 \times 3 \times \dots \times n) \leq \log(n \times n \times \dots \times n)$

$1 \leq \log n! \leq \log n^n$

$n \log n$

$\Omega(1)$ $O(n \log n)$

Properties of Asymptotic Notations

1. General Properties

If $f(n)$ is $\Theta(g(n))$ / $O(g(n))$ / $\Omega(g(n))$, then $a * f(n)$ is $\Theta(g(n))$ / $O(g(n))$ / $\Omega(g(n))$

eg. $f(n) = 2n^2 + 5$ is $O(n^2)$

then $7 \cdot f(n) = 7(2n^2 + 5)$

$= 14n^2 + 35$ is $O(n^2)$

2. Reflexive properties

If $f(n)$ is given then $f(n)$ is $O(f(n))$

eg: $f(n) = n^2$ $f(n) = O(n^2)$

3. Transitive

If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$

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

eg: $f(n) = n$ $g(n) = n^2$ $h(n) = n^3$

n is $O(n^2)$ and n^2 is $O(n^3)$

then n is $O(n^3)$

4. Symmetric

If $f(n)$ is $\Theta(g(n))$ then $g(n)$ is $\Theta(f(n))$

eg. $f(n) = n^2$ $g(n) = n^2$

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

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

5. Transpose Symmetric

If $f(n) = O(g(n))$ then $g(n)$ is $\Omega(f(n))$

eg: $f(n) = n \quad g(n) = n^2$

then n is $O(n^2)$ and n^2 is $\Omega(n)$

6. If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$,

then $g(n) \leq f(n) \leq g(n)$

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

7. If $f(n) = O(g(n))$ and $d(n) = O(e(n))$,

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

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

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

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

8. If $f(n) = O(g(n))$ and $d(n) = O(e(n))$,

then $f(n) * d(n) = O(g(n) * e(n))$

COMPARISION OF FUNCTIONS

Method 1

n	n^2	$<$	n^3
2	$2^2 = 4$		$2^3 = 8$
3	$3^2 = 9$		$3^3 = 27$
4	$4^2 = 16$		$4^3 = 64$

Method 2

n^2 $\quad$ n^3

Apply log on both sides

$\log n^2 \quad \log n^3$

$2 \log n < 3 \log n$

Eg: $f(n) = n^2 \log n$ $\quad g(n) = n(\log n)^{10}$

Apply $\log_2$

$\log[n^2 \log n]$ $\quad \log[n(\log n)^{10}]$

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

$2\log n + \log\log n$ $\quad \log n + 10 \cdot \log\log n$

Biggn term

$\therefore n^2 \log n > n(\log n)^{10}$

$\log ab = \log a + \log b$

$\log \frac{a}{b} = \log a - \log b$

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

$a^{\log_c b} = b^{\log_c a}$

$a^b = n$ then $b = \log_a n$

Eg: $f(n) = 3n^{\sqrt{n}}$ $\quad g(n) = 2^{\sqrt{n}\log n}$

$a = 2 \quad b = n^{\sqrt{n}} \quad c = 2$

$3n^{\sqrt{n}}$ $\quad (2)^{\log_2(n^{\sqrt{n}})}$

$3n^{\sqrt{n}}$ $\quad (n^{\sqrt{n}})^{\log_2 2^1}$

$\left[\because a^{\log_c b} = b^{\log_c a}\right]$

$3n^{\sqrt{n}} > n^{\sqrt{n}}$ ← Asymptotically equal. Value-wise different

$\therefore 3n^{\sqrt{n}} > 2^{\sqrt{n}\log n}$

Eg: $f(n) = n^{\log n}$ $\quad g(n) = 2^{\sqrt{n}}$

Apply $\log$

$\log n^{\log n}$ $\quad \log 2^{\sqrt{n}}$

$\log n \times \log n$ $\quad \sqrt{n} \log_2 2^1$

$\log^2 n \qquad \sqrt{n} = n^{1/2}$

$2 \log \log n \quad < \quad \frac{1}{2} \log n$

$\therefore \quad n^{\log n} < 2^{\sqrt{n}}$

Eg: $f(n) = 2^{\log n} < g(n) = n^{\sqrt{n}}$

$\log n \times \log_n 2^1 \qquad \sqrt{n} \cdot \log n$

$\log n \quad < \quad \sqrt{n} \cdot \log n$

Eg: $f(n) = 2n \qquad g(n) = 3n$ Equal

Eg: $f(n) = 2^n \qquad g(n) = 2^{2n}$

$\log 2^n \qquad \log 2^{2n}$

$n \log_2 2^1 \quad < \quad 2n \cdot \log_2 2^1$

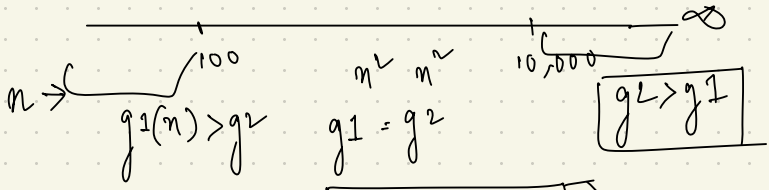
$\therefore \quad 2^n < 2^{2n}$

Remark: Once log is taken on both sides, coefficients begins to matter

Eg:

$$g1(n) = \begin{cases} n^3 & n < 100 \\ n^2 & n \geq 100 \end{cases}$$

$$g2(n) = \begin{cases} n^2 & n < 10,000 \\ n^3 & n \geq 10,000 \end{cases}$$



$\therefore$ $g2(n) > g1(n)$

Q. True or False

1. $(n+k)^m = \Theta(n^m)$ → $(n+3)^2 = \Theta(n^2)$
2. $2^{n+1} = O(2^n)$ → $2 \cdot 2^n$ ∴ $2^{n+1} = O(2^n)$, $2^{n+1} = \Omega(2^n)$, $2^{n+1} = \Theta(2^n)$
3. $2^{2n} = O(2^n)$ → $2^{2n} = 4^n$, $4^n > 2^n$
4. $\sqrt{\log n} = O(\log \log n)$ → $2^{\frac{1}{2}} \log \log n > \log \log \log n$
5. $n^{\log n} = O(2^n)$

$\log n \cdot \log n$ $\quad n \log \frac{n}{2}$

$2 \cdot \log n < n$

$\therefore n^{\log n} < 2^n$

Best, Worst and Average Case Analysis

1. Linear Search

Best case – Searching key element present at first index

$B(n) = O(1)$

Worst case – Searching a key element at last index

$W(n) = O(n)$

Average case – $\frac{\text{all possible case time}}{\text{no. of cases}}$

Avg. time $= \frac{1+2+3+\cdots+n}{n} = \frac{n(n+1)/2}{n} = \frac{n+1}{2}$

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

Asymptotic Notations :-

$B(n) = 1$	$W(n) = n$	$A(n) = \frac{n+1}{2}$
$B(n) = O(1)$	$W(n) = O(n)$	$A(n) = O(n)$
$B(n) = \Omega(1)$	$W(n) = \Omega(n)$	$A(n) = \Omega(n)$
$B(n) = \theta(1)$	$W(n) = \theta(n)$	$A(n) = \theta(n)$

2. Binary Search Tree

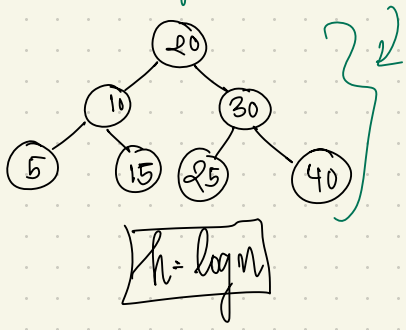
Best case – Search root element

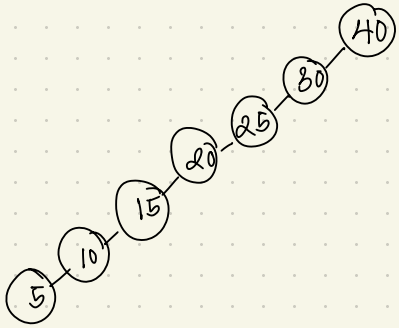
Best case time – $B(n) = 1$

Worst case – Searching for leaf element

Worst case time – $W(n) = h = \log n$

height balanced BST





Left skewed BST

$h = n$

$\therefore$ min $w(n) = \log n$

max $w(n) = n$

DISJOINT SETS

1. Disjoint Sets & operations
2. Detecting a cycle.
3. Graphical Representation
4. Array Representation.
5. Weighted Union and Collapsing Find.

Disjoint Sets

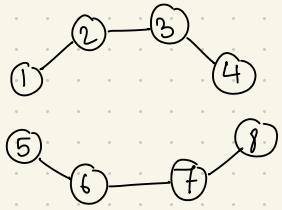
$S_1 = \{1, 2, 3, 4\}$

$S_2 = \{5, 6, 7, 8\}$

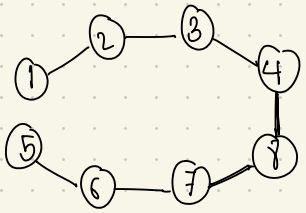
$S_1 \cap S_2 = \phi$

2 operations $\Rightarrow$ ① Find
② Union

$\therefore S_3 = S_1 \cup S_2 = \{1, 2, 3, 4, 5, 6, 7, 8\}$

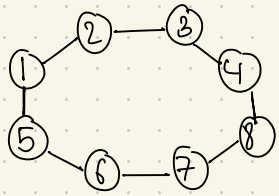


Taking union



Detecting cycle

$S_3 = \{1, 2, 3, 4, 5, 6, 7, 8\}$



- If we take an edge (v, v) and both $v, v \in S_3$, then a cycle is detected.

  eg. $(1, 5)$

- So, disjoint sets can be used to detect cycles in a graph.

Eg. $U = \{1, 2, 3, 4, 5, 6, 7, 8\}$

$S_1 = \{1, 2\}$

$S_2 = \{3, 4\}$

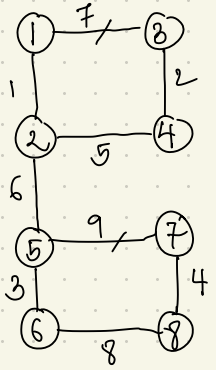
$S_3 = \{5, 6\}$

$S_4 = \{7, 8\}$

$S_5 = \{1, 2, 3, 4\}$

$S_6 = \{1, 2, 3, 4, 5, 6\}$

∵ (1,3) belongs to same set S_6. So, a cycle is detected.



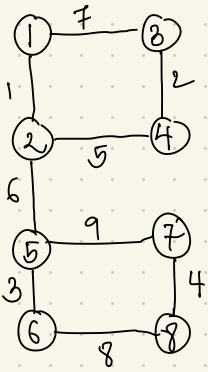
2 cycles

$S_7 = \{1, 2, 3, 4, 5, 6, 7, 8\}$

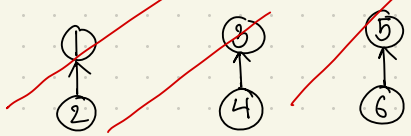
∵ (5,7) belongs to same set S_7. So, another cycle detected.

Graphical Representation

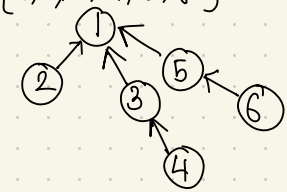
$U = \{1, 2, 3, 4, 5, 6, 7, 8\}$

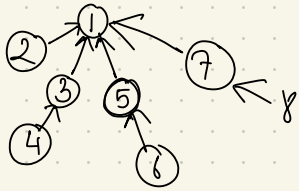


$S_1 = \{1, 2\}$ $S_2 = \{3, 4\}$ $S_3 = \{5, 6\}$ $S_4 = \{7, 8\}$



$S_5 = \{1, 2, 3, 4, 5, 6\}$

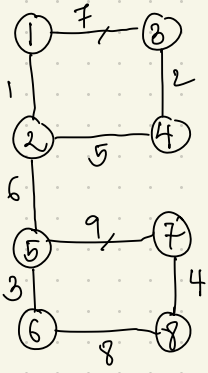


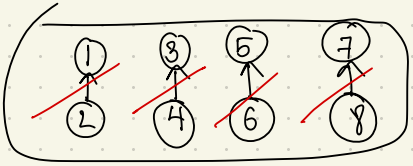


Array Representation:

$U = \{1, 2, 3, 4, 5, 6, 7, 8\}$

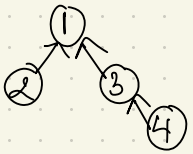
Parent	-8	1	1	3	1	5	1	7
	1	2	3	4	5	6	7	8



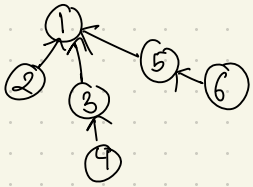


(⊖ sign shows its a parent. mag. represents no. of nodes)

(2,4)



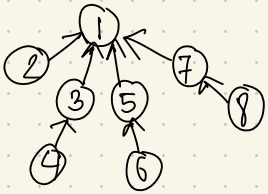
(2,5)



(1,3)

If (1,3) is included, it could form a cycle. So, pass.

(6,8)



(5,7)

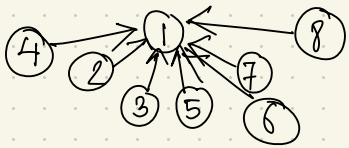
Could lead to cycle. PASS.

Collapsing Find :

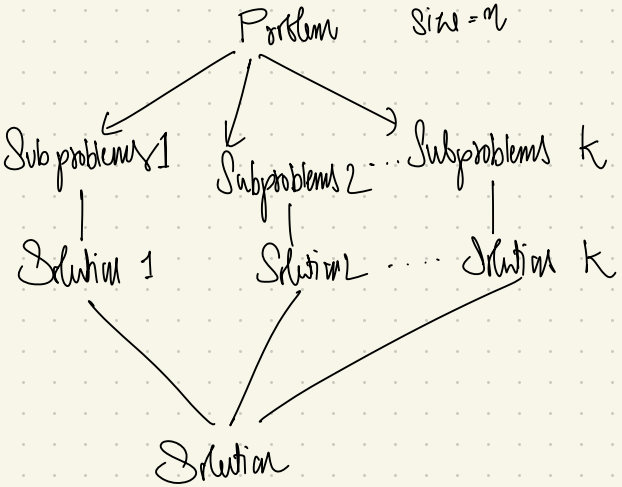
- The procedure of linking a node to the direct parent of a set is called collapsing find.
- So, the time taken to find a node can be reduced when we do it a second time.

Eg. Parent

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



DIVIDE AND CONQUER




```
DAC(P)
{   if (small(P))
    {   S(P);
    }
    else
    {   divide P into P1, P2, P3, ... Pk
        Apply DAC(P1), DAC(P2), ....
        Combine (DAC(P1), DAC(P2), ....)
    }
}
```

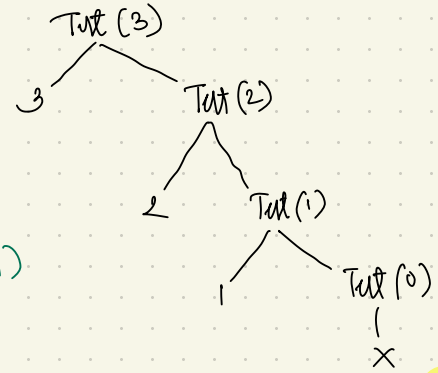
Divide & Conquer

1. Binary Search
2. Finding maximum and minimum
3. Merge Sort
4. Quick Sort
5. Strassen's Matrix Multiplication.

RECURRENCE RELATION

1.
```
void Test(int n) — T(n)
{ if (n>0)
  { printf("%d", n); — 1
    Test(n-1); ——— T(n-1)
  }
}
```

Test(3)



If n=3 ; 3 execution of printf() ; 3+1 calls

$\therefore f(n) = n+1$ (Approximating that the amount of work done depends on the no. of calls)

$O(n)$

Finding Recurrence Relation;

Let $T(n)$ = Time taken

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

- If we have some const. value, we can simply take 1/c/a.

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

Solving Recurrence Relation

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

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

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

Substitute $T(n-1)$,

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

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

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

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

⋮ continue for k times

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

Assume $n-k = 0$

$\therefore n = k$

$\therefore T(n) = T(n-n) + n$

$T(n) = T(0) + n$

$T(n) = 1 + n$

$\Theta(n)$

2.

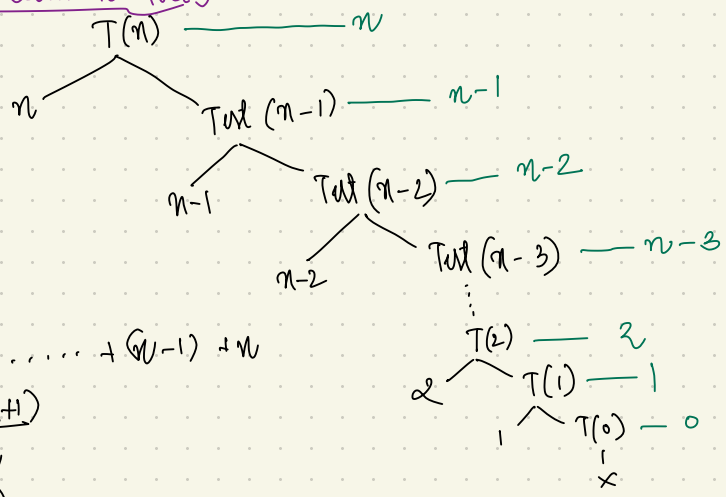
```
void Test (int n)          — T(n)
{
    if (n>0)               — 1
    {
        for (i=0; i<n; i++)
        {
            printf ("%d", n);   — n+1, n
        }
        Test (n-1);        — T(n-1)
    }
}
```

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

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

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

Solving using Recurrence tree



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

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

$\Theta(n^2)$

Solving using backsubstitution / induction method

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

$T(n) = T(n-1) + n$ — (1)

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

$T(n) = T(n-2) + (n-1) + n$ — (2)

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

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

⋮ k times

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

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

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

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

Assume $n-k=0$

$\therefore n=k$

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

$T(n) = T(0) + [1 + 2 + 3 + \dots + n]$

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

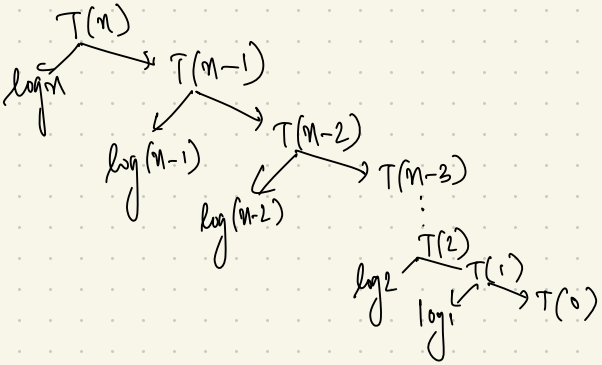
3. void Test (int n) — $T(n)$

```
{  if (n>0)
   {  if (i=1 ; i<n ; i=i*2)
      {   printf ("%d", i);   ——— logn
      }
      Test (n-1);   ——————————— T(n-1)
   }
}
```

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

$$\therefore T(n) = \begin{cases} 1 & n=0 \\ T(n-1) + \log n & n>0 \end{cases}$$

Tree method



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

$= \log\{n(n-1)(n-2)\cdots 1\}$

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

Substitution method

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

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

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

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

$\vdots$

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

$\therefore n - k = 0$

$n = k$

$\therefore T(n) = T(0) + \log(n!)$

$\therefore T(n) = 1 + \log(n!)$ $\quad O(n \log n)$

Recurrence Relation for Decreasing Function (Direct Answer)

$T(n) = T(n-1) + 1$ — $O(n)$

$T(n) = T(n-1) + n$ — $O(n^2)$

$T(n) = T(n-1) + \log n$ — $O(n \log n)$

$T(n) = T(n-1) + n^2$ — $O(n^3)$

$T(n) = T(n-2) + 1$ — $n/2$ $O(n)$

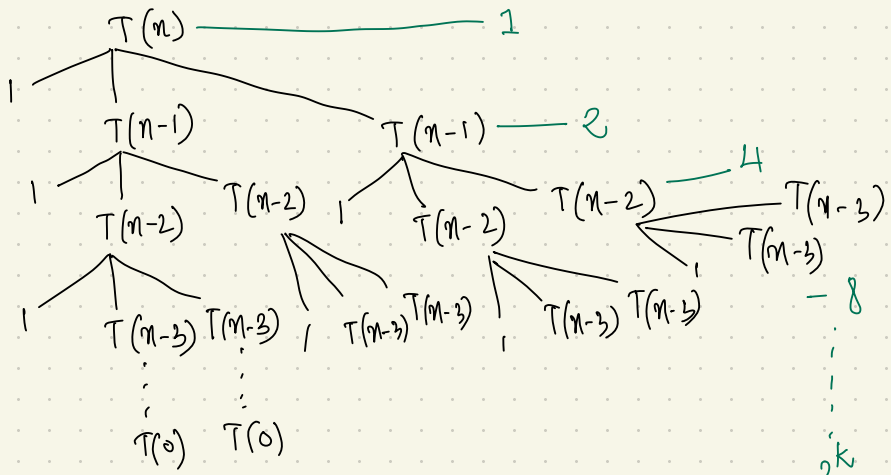
$T(n) = T(n-100) + n$ — $O(n^2)$

$T(n) = 2T(n-1) + 1$ — $O(?)$

4.

```
Algorithm Test (int n)  ——— T(n)
{
    if (n>0)
    {
        printf("%d", n) ;— 1
        Test (n-1) ; ——— T(n-1)
    }
    Test (n-1) ; ——— T(n-1)
}
```

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



$$1 + 2 + 2^2 + 2^3 + \cdots + 2^k = 2^{k+1} - 1$$

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

Assume $n - k = 0$

$\therefore n = k$

$$a + ar^1 + ar^2 + \cdots + ar^k = \frac{a(r^{k+1} - 1)}{r - 1}$$

Back substitution method

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

$T(n) = 2T(n-1)+1$ — (1)

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

$T(n) = 2^2 T(n-2)+2+1$ — (2)

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

$T(n) = 2^3 T(n-3)+2^2+2+1$ — (3)

$\vdots$

$T(n) = 2^k T(n-k) + 2^{k-1} + 2^{k-2} + \cdots + 2^2 + 2 + 1$ — (4)

Assume $n-k=0$,

$n = k$

$\therefore T(n) = 2^k T(0) + 1 + 2 + 2^2 + \cdots + 2^{k-1}$

$= 2^k \times 1 + 2^k - 1$

$= 2^n + 2^n - 1$

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

$O(2^n)$

MASTER THEOREM FOR DECREASING FUNCTION

$T(n) = T(n-1)+1$ — $O(n)$

$T(n) = T(n-1)+n$ — $O(n^2)$

$T(n) = T(n-1)+\log n$ — $O(n \log n)$

$T(n) = 2T(n-1)+1$ — $O(2^n)$

$T(n) = 3T(n-1)+1$ — $O(3^n)$

$T(n) = 2T(n-1)+n$ — $O(n \cdot 2^n)$

General form of Recurrence Relation:

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

$a > 0 \quad b > 0$ and $f(n) = O(n^k)$ where $k \geq 0$

Case

1. if $a < 1$ — $O(n^k)$, $O(f(n))$
2. if $a = 1$ — $O(n^{k+1})$, $O(n * f(n))$
3. if $a > 1$ — $O(n^k a^{n/b})$, $O(f(n) * a^{n/b})$

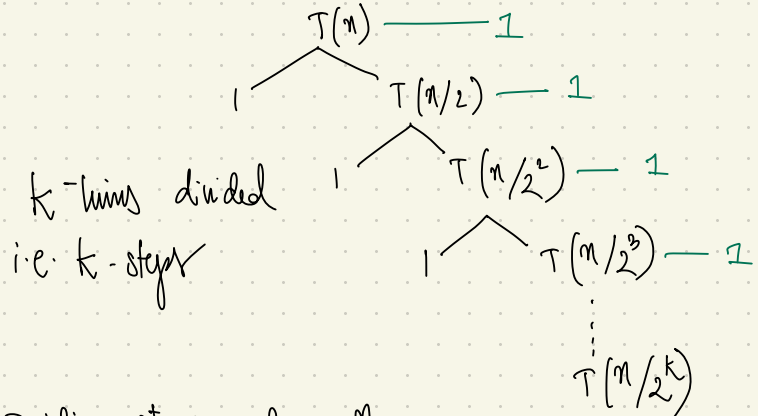
$f(n)$
$n * f(n)$
$f(n) * a^{n/b}$

DIVIDING FUNCTIONS

①
```
Algorithm Test (int n)  ——— T(n)
{
    if (n > 1)
    {
        printf ("%d", n);  ——— 1
        Test (n/2);  ——— T(n/2)
    }
}
```

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

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



k - times divided
i.e. k - steps

Dividing steps when $\frac{n}{2^k} = 1$

$\therefore n = 2^k$

$k = \log_2 n$

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

$2^5 = 32$

$5 = \log_2 32$

Substitution method :

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

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

$T(n) = T\left(n/2^2\right) + 2$ —— (2)

$T(n) = T\left(n/2^3\right) + 3$ —— (3)

$\vdots$

$T(n) = T\left(n/2^k\right) + k$ —— (4)

Assume $n/2^k = 1$

$n = 2^k$

$k = \log_2 n$

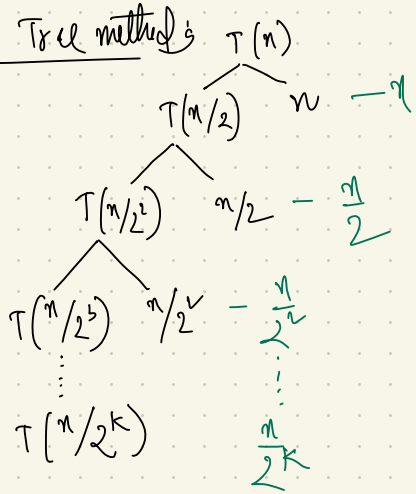
$\therefore T(n) = T(1) + \log(n)$

$T(n) = 1 + \log(n)$

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

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

Recurrence Tree method:



$$T(n) = n + \frac{n}{2} + \frac{n}{2^2} + \frac{n}{2^3} + \cdots + \frac{n}{2^k}$$

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

$$= n\left(\sum_{i=0}^{k} \frac{1}{2^i}\right) = 1$$

$$= n \times 1$$

$$T(n) = n$$

$O(n)$

Substitution method:

$$T(n) = T(n/2) + n \quad \text{——} (1)$$

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

$$T(n) = T\left(n/2^2\right) + \frac{n}{2} + n \quad \text{——} (2)$$

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

$$\vdots$$

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

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

$\Rightarrow n = 2^k$

$\Rightarrow k = \log n$

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

$$T(n) = 1 + n \cdot [1+1]$$

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

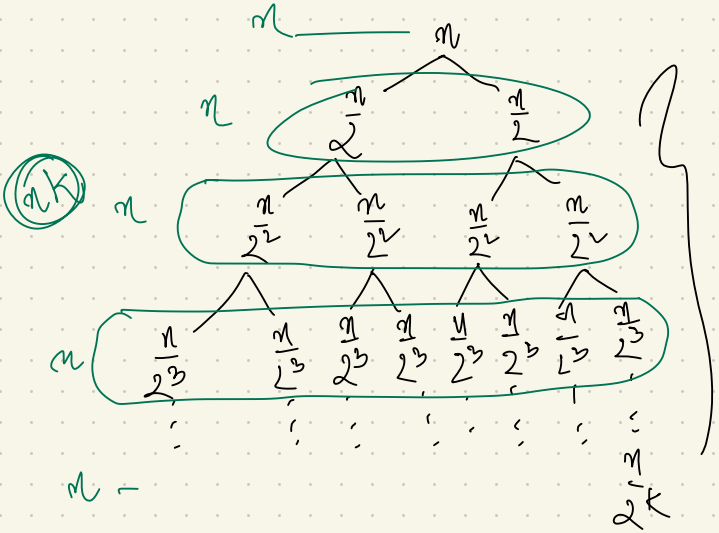
$O(n)$

③ void Test (int n) — T(n)

```
{ if (n>1)
  { for (i=0 ; i<n; i++)        } — n
    { stmt ;                    }
  }
  Test (n/2) ;  — T(n/2)
  Test (n/2) ;  — T(n/2)
}
```

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

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



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

$n = 2^k$

$k = \log n$

$\therefore T(n) = O(nk)$

$= O(n \log n)$

Substitution method:

MASTER THEOREM FOR DIVIDING FUNCTIONS

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

① $\log_b a$

② k

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

$$f(n) = \theta(n^k \log^p n)$$

Case 1: if $\log_b a > k$, then $\theta(n^{\log_b a})$

Case 2: if $\log_b a = k$, then

- if $p > -1$, then $\theta(n^k \cdot \log^{p+1} n)$
- if $p = -1$, then $\theta(n^k \cdot \log \log n)$
- if $p < -1$, then $\theta(n^k)$

Case 3: if $\log_b a < k$, then

if $p \geq 0$, then $\theta(n^k \log^p n)$.

if $p < 0$, then $O(n^k)$.

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

$a = 2$

$b = 2$

$f(n) = \theta(1) = \theta(n^0 \cdot \log^0 n)$

$k = 0 \quad p = 0$

$\log_b a = 1 \quad k = 0$

$\therefore$ Case 1. $\theta(n^1)$

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

$a = 4$

$b = 2$

$f(n) = \theta(n) = \theta(n^1 \cdot \log^0 n)$

$k = 1 \quad p = 0$

$\log_b a = 2 \quad k = 1$

$\therefore$ Case 1. $\theta(n^2)$

Eg. $T(n) = 8T(n/2) + n^2$

$\log_2 8 = 3 > k = 2 \quad p = 0$

$\theta(n^3)$

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

$\log_2 2 = 1 \quad k = 1 \quad p = 0$

$\theta(n^k \cdot \log^{p+1} n) = \theta(n \cdot \log n)$

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

$\log_2 4 = 2 \quad k = 2$

$\theta(n^2 \cdot \log^3 n)$

Eg. $T(n) = 2T(n/2) + \frac{n}{\log n}$

$\log_2 2 = 1 \quad k = 1 \quad p = -1$

$\Theta(n \cdot \log\log n)$

Eg. $T(n) = 2T(n/2) + n \times \log^2 n$

$\log_2 2 = 1 \quad k = 1 \quad p = -2$

$\Theta(n)$

ROOT FUNCTION

```
void Test(int n) ---- T(n)
{
    if (n>2)
    {
        stmt; ---------- 1
        Test(√n); ------ T(√n)
    }
}
```

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

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

$\therefore \quad T(n) = T(n^{1/2}) + 1 \quad —(1)$

$T(n) = T(n^{1/2^2}) + 2 \quad —(2)$

$T(n) = T(n^{1/2^3}) + 3 \quad —(3)$

$\vdots$

$T(n) = T(n^{1/2^k}) + k \quad —(4)$

Assume $n = 2^m$

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

Assume $T(2^{m/2^k}) = T(2)$

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

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

BINARY SEARCH (RECURSIVE)

```
Algorithm RBinSearch (l, h, key) ---- T(n)
{
  if (l == h)
  {
    if (A[l] == key)
      return l;
    else
      return 0;
  }
  else
  {
    mid = (l+h)/2;
    if (key == A[mid])
      return mid;
    if (key < A[mid])
      return RBinSearch (l, mid-1, key);
    else
      return RBinSearch (mid+1, h, key);
  }
}
```

1 — 1 — 1 — 1 — T(n/2)

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

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

$a = 1 \quad b = 2 \quad \log_b a = 0 \quad k = 0$

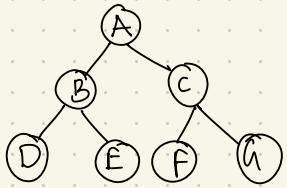
$p = 0 \Rightarrow p > -1$

$\Theta(\log n)$

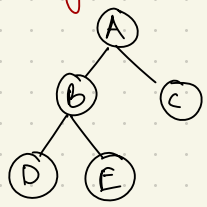
$p > -1$	$\Theta(n^k \log^{p+1} n)$
$p = -1$	$\Theta(n^k \log\log n)$
$p < -1$	$\Theta(n^k)$

HEAP

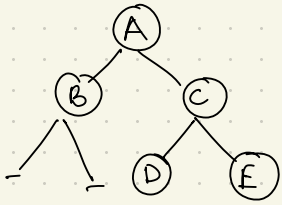
Representation of BT using array



T	A	B	C	D	E	F	G
	1	2	3	4	5	6	7



T	A	B	C	D	E
	1	2	3	4	5



T	A	B	C	–	–	D	E
	1	2	3	4	5	6	7

if a node is at index — i

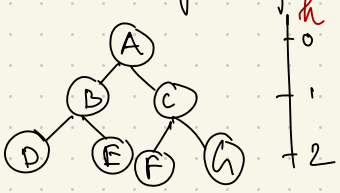
its left child is at — $2*i$

its right child is at — $2*i+1$

its parent is at — $\left\lfloor \frac{i}{2} \right\rfloor$

Full and Complete Binary Tree

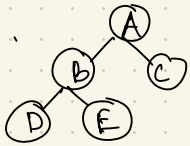
Full Binary Tree - No space for new node



(*) If height of binary tree = h, then

no. of nodes in full binary tree = $2^{h+1} - 1$

Complete Binary Tree - No empty space or gaps when represented as an array. eg.



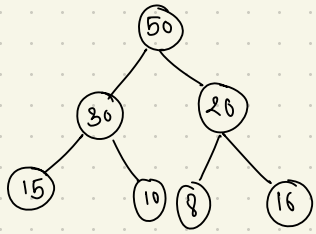
A	B	C	D	E	–	–

A complete binary tree is a full binary tree upto height h-1, and in the last level, elements are filled from left to right.

(*) If no. of nodes = n, then height of complete binary tree = $\lfloor \log n \rfloor$

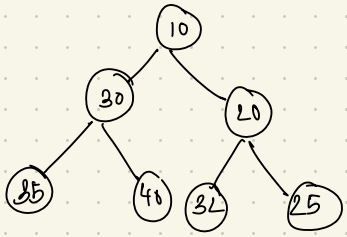
Heap

Max heap



H	1	2	3	4	5	6	7
	50	30	20	15	10	8	16

Min heap



H	1	2	3	4	5	6	7
	10	30	20	35	40	32	25

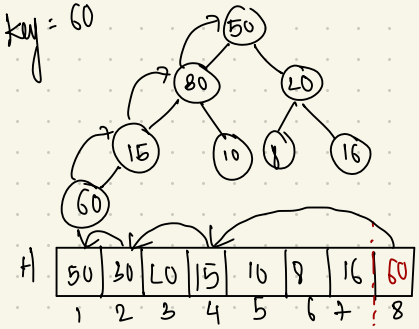
(1) Heap is a complete binary tree.

(2) Every node has a value greater than or equal to all its descendents.

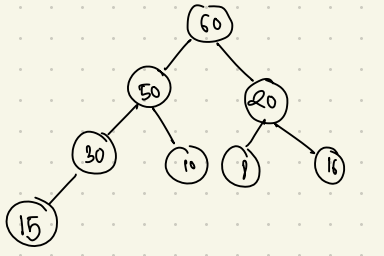
(2) Every node has a value smaller than or equal to all its descendents.

Insert opn. in max heap

key = 60



H	1	2	3	4	5	6	7	8
	50	30	20	15	10	8	16	60



H	1	2	3	4	5	6	7	8
	60	50	20	30	10	8	16	15

(*) Time taken = no. of swaps ⇒ depends on height of comp. binary tree.

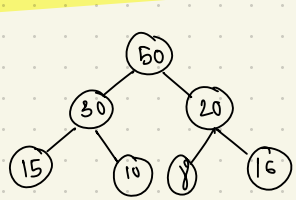
O(1) to O(log n)

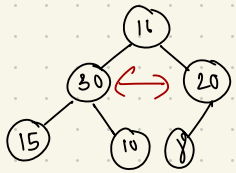
min. time — max. time

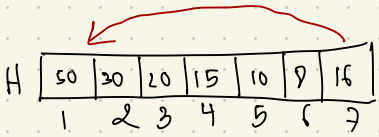
(*) Insertion → Upward Adjustment.

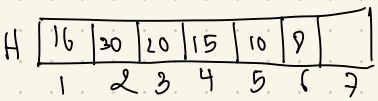
Delete opn. in max. heap

We cannot delete any other element but root element.

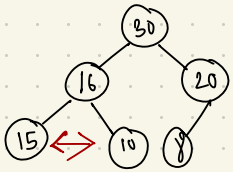






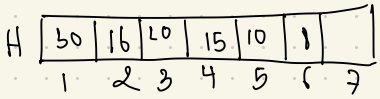


30 > 20
&
30 > 16



15 > 10
But, 16 > 15

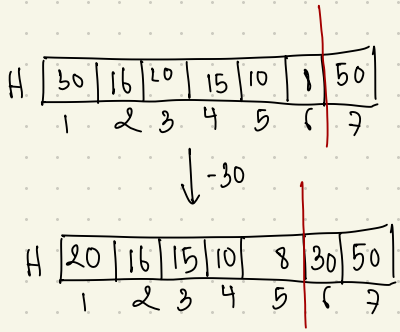
Same



Note. ① Deletion → Downward Adjustment

② Time complexity → O(log n)

(3) If we want to maintain a copy of the deleted elements, we can keep it in the empty space of the array. Heap size still remains 6.

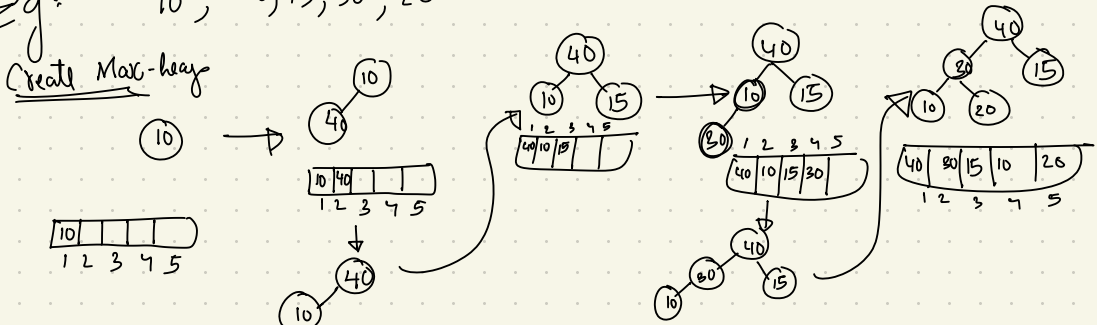


Heap Sort (increasing / decreasing)

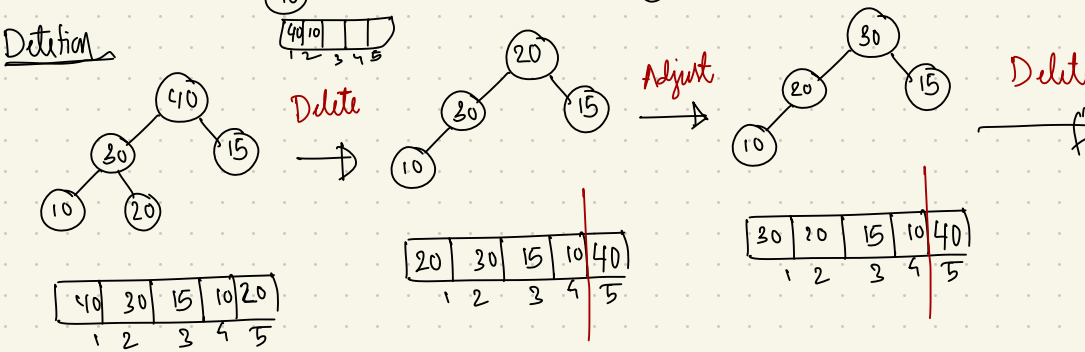
Steps: ① Create heap (min/max) from unordered array.
② Delete all elements one by one.

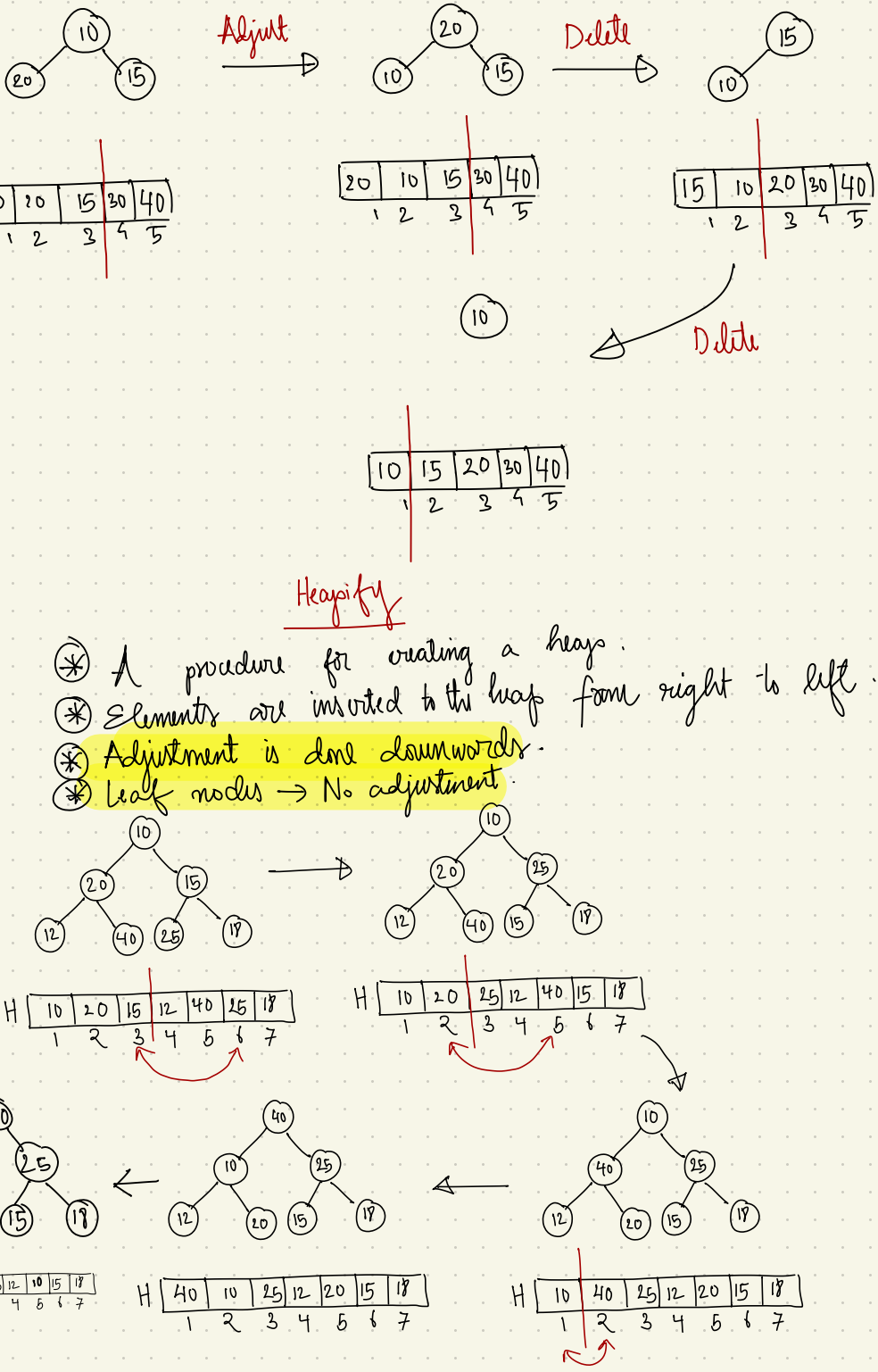
Eg: 10, 40, 15, 30, 20

Create Max-heap



Deletion




Adjust
Delete
Delete
Heapify
A procedure for creating a heap.
Elements are inserted to the heap from right to left.
Adjustment is done downwards.
Leaf nodes → No adjustment

* Creating a heap by performing insertion operation for each element → $O(n \log n)$
* Heapify → $O(n)$

MAX-HEAPIFY (A, i)

→ For maintaining the max-heap property.
→ Sub trees rooted at i are made to obey max-heap property.
→ $O(\log n)$

```
MAX-HEAPIFY(A, i) {
    l = LEFT(i)
    r = RIGHT(i)
    if l ≤ A.heap-size and A[l] > A[i]
        largest = l
    else largest = i
    if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
    if largest ≠ i
        exchange A[i] and A[largest]
        MAX-HEAPIFY(A, largest)
}
```

Strassen's Matrix Multiplication

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

Muggle Method

$$c_{ij} = \sum_{k=1}^{n} a_{ik} * b_{kj}$$

```
for (i=0; i<n; i++)
{   for (j=0; j<n; j++)
    {   c[i,j] = 0;
        for(k=0; k<n; k++){
        }  c[i,j] += A[i,k] * B[k,j];   ⟶ O(n^3)
    }
}
```

Divide & Conquer Strategy

$$\left.\begin{aligned} c_{11} &= a_{11} * b_{11} + a_{12} * b_{21} \\ c_{12} &= a_{11} * b_{12} + a_{12} * b_{22} \\ c_{21} &= a_{21} * b_{11} + a_{22} * b_{21} \\ c_{22} &= a_{21} * b_{12} + a_{22} * b_{22} \end{aligned}\right\} 2\times 2$$

$$\left.\begin{aligned} A &= [a_{11}] \quad B = [b_{11}] \\ C &= [a_{11} * b_{11}] \end{aligned}\right\} 1\times 1$$

$$A = \left[\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right] \quad \begin{matrix} A_{11} \\ A_{12} \\ A_{21} \\ A_{22} \end{matrix}$$

$$\frac{4}{2}\times\frac{4}{2}$$

$$B = \left[\begin{array}{cc|cc} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ \hline b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{array}\right] \quad \begin{matrix} B_{11} \\ B_{12} \\ B_{21} \\ B_{22} \end{matrix}$$

$$\frac{4}{2}\times\frac{4}{2}$$

Algorithm MM(A, B, n)

```
{
    if (n <= 2)
    {
        c = 4 formulas
        ⋮
    }
    else
    {
        mid = n/2
        MM(A11, B11, n/2) + MM(A12, B21, n/2)
        MM(A11, B12, n/2) + MM(A12, B22, n/2)
        MM(A21, B11, n/2) + MM(A22, B21, n/2)
        MM(A21, B12, n/2) + MM(A22, B22, n/2)
    }
}
```

$$T(n) = \begin{cases} 1 & n = 1 \\ 8T(n/2) + n^2 & n > 1 \end{cases}$$

$a = 8$ $\quad b = 2$ $\quad \log_b a = 3 \quad k = 2$

$f(n) = n^2$

$$\boxed{\Theta(n^3)}$$

$$P = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$$

$$Q = (A_{21} + A_{22}) \cdot B_{11}$$

$$R = A_{11} \cdot (B_{12} - B_{22})$$

$$S = A_{22} \cdot (B_{21} - B_{11})$$

$$T = (A_{11} + A_{12}) \cdot B_{22}$$

$$U = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$$

$$V = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$$

$$C_{11} = P + S - T + V$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

$$C_{22} = P + R - Q + U$$

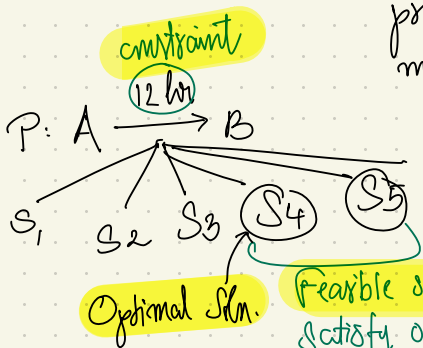
$$T(n) = \begin{cases} 1 & n = 1 \\ 7\,T(n/2) + n^2 & n > 1 \end{cases}$$

GREEDY METHOD

This method is used for solving Optimization problems

problems that demand either minimum or maximum result.

Eg:



minimum cost — MINIMIZATION PROBLEM

(*) For any problem, there can be only one optimal solution.

Strategies used for solving Optimization problems:

1. Greedy method
2. Dynamic programming
3. Branch and Bound

GREEDY METHOD

It says that a problem must be solved in stages.

```
Algorithm Greedy (a, n)
{
    for i = 1 to n do
    {
        x = Select (a);
        if Feasible (x) then
        {
            Solution = Solution + x;
        }
    }
}
```

n = 5

a	a_1	a_2	a_3	a_4	a_5
	1	2	3	4	5

FRACTIONAL KNAPSACK PROBLEM

$n = 7$

$m = 15$ → Bag capacity

Objects: O	1	2	3	4	5	6	7
Profits: P	10	5	15	7	6	18	3
Weights: W	2	3	5	7	1	4	1
P/W	5 ✓	1.3	3 ✓	1	6 ✓	4.5 ✓	3 ✓

① Greedy about P_i
② Greedy about W_i
③ Greedy about P/W (optimal)

✱ We have to fill the bag with the objects such that the profit is maximized (OPTIMIZATION PROBLEM) → $\max \sum x_i p_i$

✱ Total weight should be less than or equal to 15 kg. (CONSTRAINT) → $\sum x_i w_i \leq m$

✱ x ⇒ selected objects & $0 \leq x \leq 1$.
i.e. fraction of an object can also be selected

15 − 1 = 14
14 − 2 = 12
12 − 4 = 8
8 − 5 = 3
3 − 1 = 2
2 − 2 = 0

$x = (1, 2/3, 1, 0, 1, 1, 1)$
$\quad\quad x_1, x_2, x_3, x_4, x_5, x_6, x_7$

Weight $= \sum x_i w_i = 1\times2 + \frac{2}{3}\times3 + 1\times5 + 0\times7 + 1\times1 + 1\times4 + 1\times1$

$= 2 + 2 + 5 + 0 + 1 + 4 + 1 = 15$

Profit $= \sum x_i p_i = 1\times10 + \frac{2}{3}\times5 + 1\times15 + 0\times7 + 1\times6 + 1\times18 + 1\times3$

$= 10 + 2\times1.3 + 15 + 6 + 18 + 3 = 54.6$

M = Knapsack Capacity

```
Greedy_Knapsack()
{
  → for i = 1 to n
      calculate profit / weight
  → sort objects in decreasing order of P/W Ratio
  → for i = 1 to n
    {
      if (M > 0 and w_i ≤ M){
        M = M − w_i
        P = P + i
      }
```

```
        else break;
        if (M>0){
            P = P + Pi (M/Wi);
        }
    }
}
```

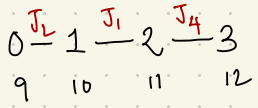
* Time complexity after sorting → O(n)
* Time complexity including sorting → O(nlogn)

JOB SEQUENCING WITH DEADLINES

n = 5

Jobs	J_1	J_2	J_3	J_4	J_5
Profits	20	15	10	5	1
deadlines	2 ✓	2 ✓	1 ✗	3 ✓	3 ✗

* Profit has to be maximised
* No one is willing to wait more than 3 hrs
* 1 unit time to complete 1 Job.



* Selected Jobs → $\{J_1, J_2, J_4\}$
* Sequence → $J_1 \rightarrow J_2 \rightarrow J_4$

  or

  $J_2 \rightarrow J_1 \rightarrow J_4$
* Total profit = 20 + 15 + 5 = 4

Job consider	Slot assign	solution	Profit
—	—	ϕ	0
J_1	[1,2]	J_1	20
J_2	[0,1][1,2]	J_1, J_2	20+15
J_3 ×	[0,1][1,2]	J_1, J_2	20+15
J_4	[0,1][1,2][2,3]	J_1, J_2, J_4	20+15+5
J_5 ×	"	"	40

J_1 J_4
$0 \to 1 \to 2 \to 3$

Q. n=7

Jobs	J_1	J_2	J_3	J_4	J_5	J_6	J_7
Profits	35	30	25	20	15	12	5
Deadlines	3	4	4	2	3	1	2

(110)

65
45

Ans.

J_4 J_3 J_1 J_2
$0 — 1 — 2 — 3 — 4$

35 + 30 + 25 + 20
= 110

OPTIMAL MERGE PATTERN

Merge in C

A	B	C
3	5	3
8	9	5
12	11	8
20	16	9
(m)	(n)	11
		12
		16
		20

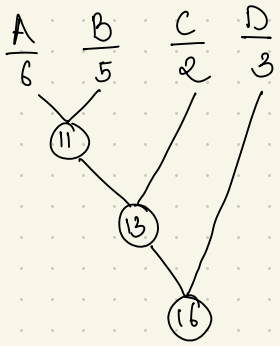
$\theta(m+n)$

* Merging can be done only on sorted list.
* A → m elements → m unit time
  B → n elements → n " "
  C → (m+n) elements → $\theta(m+n)$

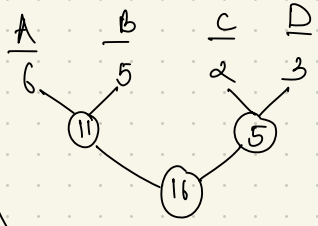
Eg.

List →	A	B	C	D
Size →	6	5	2	3

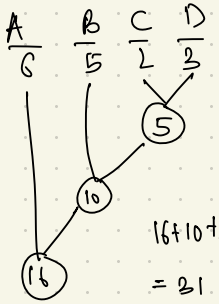
(*) We have to merge the four lists using 2-way merging.



11 + 13 + 16 = 40 = total amount of merging.
= total cost of merging.



11 + 5 + 16 = 32

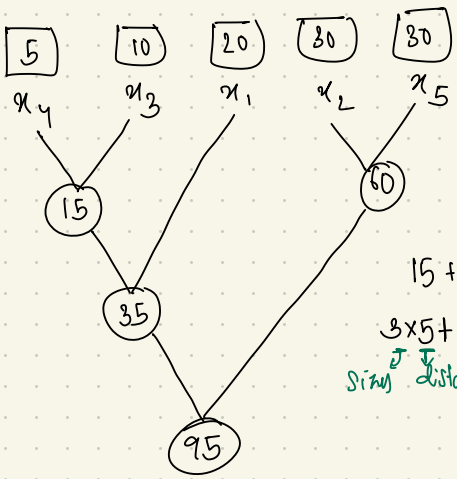


16 + 10 + 5 = 31

Q.

lists →	x_1	x_2	x_3	x_4	x_5
Sizes →	20	30	10	5	30

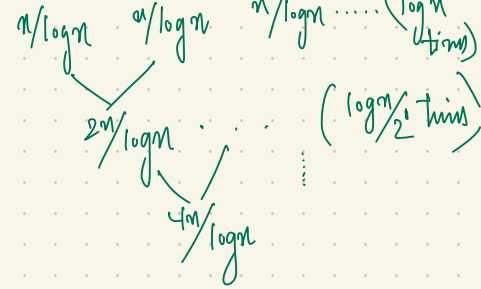
Ans.



15 + 35 + 60 + 95 = 205

3×5 + 3×10 + 2×20 + 2×30 + 2×30 = 205

Sizes * Distances



Huffman Coding

Message → BCCABB DDAE CC BB AE DD CC

length = 20

ASCII – 8 bit

8 × 20 = 160 bits

A	65	01000001
B	66	01000010
C	67	⋮
D	68	
E	69	

* We don't need 8-bit codes to represent just 5 alphabets
* We can use our own code with less amount of bits to represent the message.

Fixed Size Code:

Message → BCCABB DDAE CC BB AE DD CC

Character	Count / Frequency		Code
A	3	3/20	000
B	5	5/20	001
C	6	6/20	010
D	4	4/20	011
E	2	2/20	100
	20		

20 × 3 = 60 bits

5 × 8 bit + 5 × 3

↑ Character ↑ Code

= 40 + 15 = 55

Msg – 60 bits

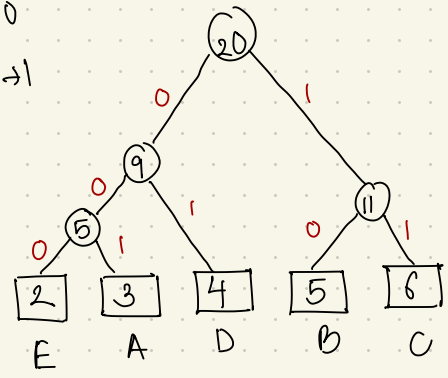
Table – 55 bits

115 bits

Variable Size Code :

Message → BCCABB DDAE CC BB AE DD CC

Left → 0
Right → 1



Char	Count	Code	
A	3	001	9 = 3×3
B	5	10	10 = 5×2
C	6	11	12 = 6×2
D	4	01	8 = 4×2
E	2	000	6 = 2×3
5×8 bits = 40 bits	20	12 bit	45 bits

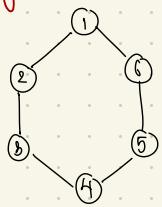
(*) Msg - 45 bits
Tree/Table - 52 bits
97 bits

(*) We can know about size of the msg. from the tree also.

$$\sum l_i * f_i = 3\times2 + 3\times3 + 2\times4 + 2\times5 + 2\times6$$
$$= 45 \text{ bit}$$

Spanning Tree :

Eg:



$G = (V, E)$

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

$E = \{(1,2),(2,3),(3,4),\ldots\}$ $|V|-1 = 5$

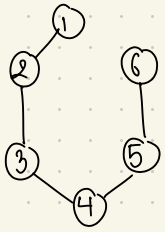
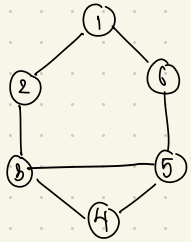


Fig: Spanning Tree (S)

$S \subseteq G$ where $S = (V', E')$ and $V' = V$ and $|E'| = |V|-1$.

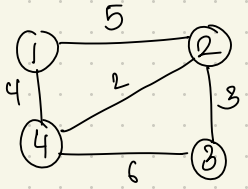
Eg.



$^7C_5 - 2$

∴ No. of possible spanning trees = $^{|E|}C_{|V|-1}$ − no. of cycles

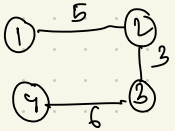
Weighted Graph



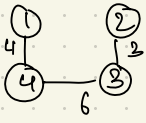
No. of possible spanning Trees
= $^{|E|}C_{|V|-1}$ − no. of cycles
= $^5C_3$ − 2
= 8

$\frac{5 \times 4}{2} = 10$

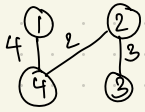
Possible sp. trees :-



cost = 14



cost = 13



cost = 9

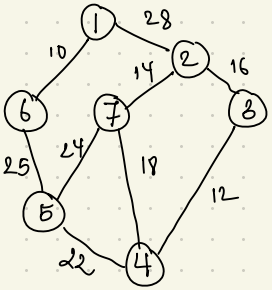
Minimum Cost Spanning Tree

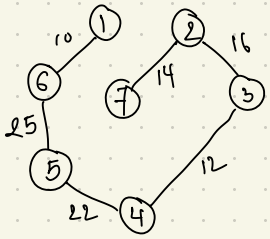
2 methods of finding M.C.S.T

1. Prim's
2. Kruskal's

Prim's Algorithm

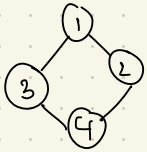
1. First select a min. cost edge.
2. For rest of the procedure, always select min. cost edge from the graph. But, make sure it is connected to already selected vertices.

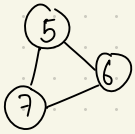




Cost = 99

Eg:

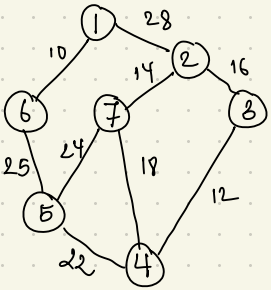


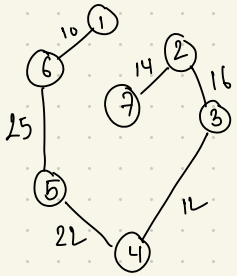


No algorithm can find min. cost sp. tree for unconnected graph.

Kruskal's Algorithm

Always select a minimum cost edge.
But, if it forms an circle, discard it.





Cost = 99

Kruskal's

① Without using min heap DS

Time complexity ⇒ $|E|$ = No. of edges
$|V|-1$ = Selected edges

$\Theta(|V||E|)$
$\Theta(n.e) = \Theta(n^2)$

② Using min heap

- Min heap always give min value when we delete.
- For min heap, time taken for deletion = $\log n$

∴ Time taken for finding min. cost edge = $\log n$.

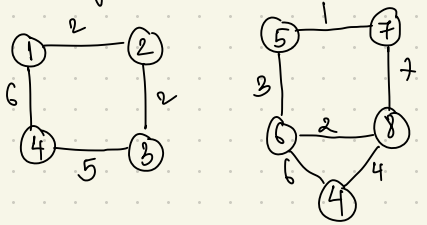
If it is repeated 'n' times,

Time complexity = $\Theta(n \log n)$

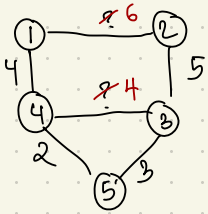
Kruskal's Algorithm on Non-connected Graph

It may find the spanning tree of each component.

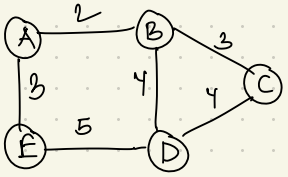
Eg:



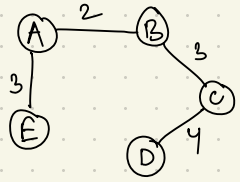
Missing Edges



Q. Find min cost sp. tree.



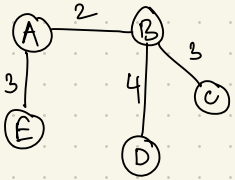
Ans. By Kruskal's



$^5C_4 - 2$ sp. trees.

3+2+3+4 = 12

OR

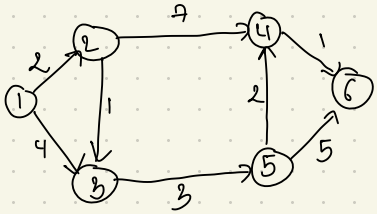


3+2+3+4 = 12

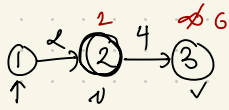
Only one optimal answer.

Dijkstra Algorithm

- For single source shortest path problems.
- Minimization Problem
- Works on both directed and non-directed graphs.



Eg.



$2+4 < \infty$

(*) This updation is called relaxation

(*) Relaxation

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

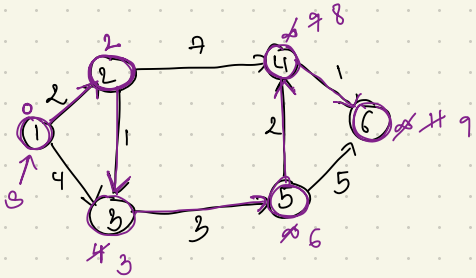
$d[v] = d[u] + c(u,v)$

$d[u]$ = distance of vertex u

$d[v]$ = " " " v.

$c(u,v)$ = cost of the edge (u,v)

Eg.



v	d[v]
2	2
3	3
4	8
5	6
6	9

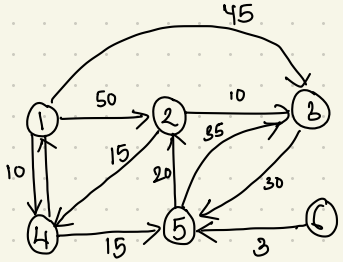
$n = |V|$

(*) Maximum no. of vertex relaxed $= n \times n$

(*) Worst case time $\Rightarrow \theta(|V|^2)$

$= \theta(n^2)$

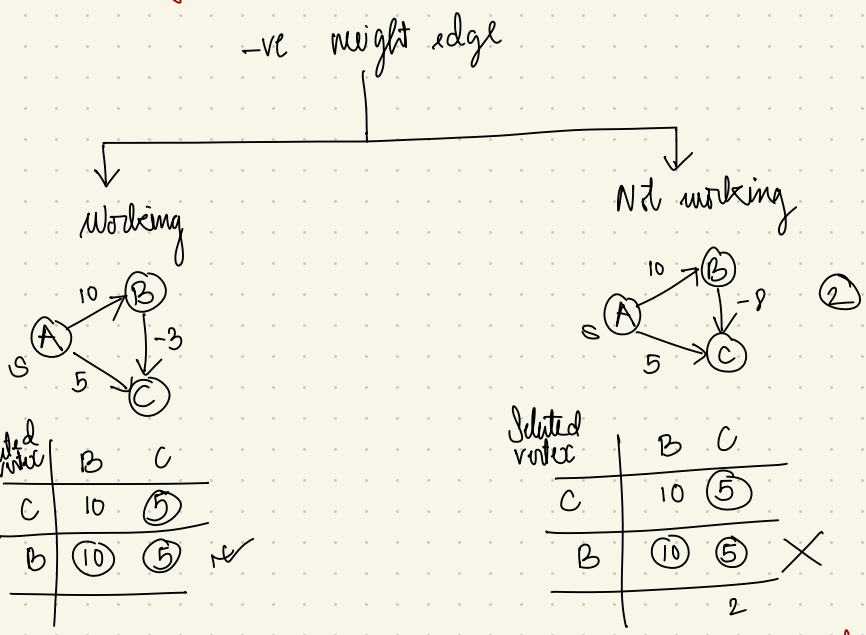
Eg: starting vertex 1



Selected vertex	2	3	4	5	6
4	50	45	(10)	∞	∞
5	50	45	(10)	(25)	∞
2	(45)	45	(10)	(25)	∞
3	(45)	(45)	(10)	(25)	∞
	(45)	(45)	(10)	(25)	(∞)

Drawbacks of Dijkstra Algorithm

-ve weight edge



Selected vertex	B	C
C	10	5
B	10	5

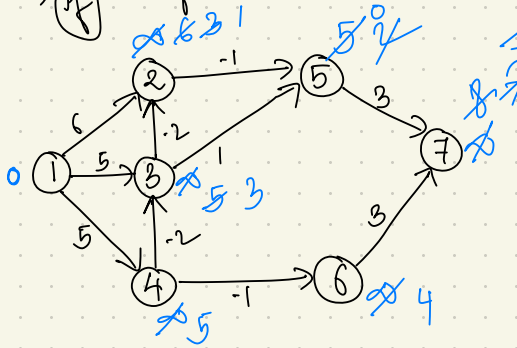
Selected vertex	B	C
C	10	5
B	10	5
		2

BELLMAN FORD ALGORITHM - Single Source Shortest Path

→ Repeated relaxation.

→ If # of vertex = $|V|$, then relax $|V|-1$ times

Eg:



$|V| = n = 7$

→ Relax $|V|-1$ times i.e. 6 times

→ Relaxation

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

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

edgeList → (1,2), (1,3), (1,4), (2,5), (3,2), (3,5), (4,3), (4,6), (5,7), (6,7)

Time complexity of Bellman-Ford

→ BF algorithm is relaxing all edges repeatedly.

→ No. of edges = $|E|$

No. of relaxations = $|V| - 1$

$\therefore O\{|E|(|V|-1)\} = O(|V||E|) = O(n^2)$

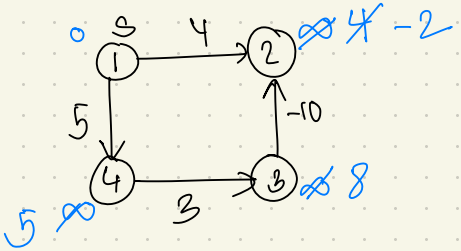
$n C_2 = \frac{n!}{(n-2)! \, 2!}$

→ If complete graph is given,

$|V| = n \qquad |E| = {}^nC_2 = \frac{n(n-1)}{2}$

$\therefore \frac{n(n-1)}{2} \times (n-1) = O(n^3)$

Q.

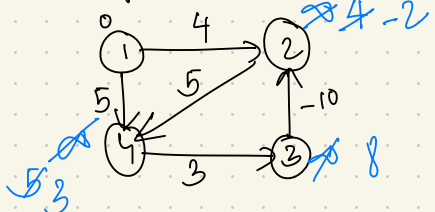


$n = 4$

$n-1 = 3$ times relaxation

Ans. edges → (3,2), (4,3), (1,4), (1,2)

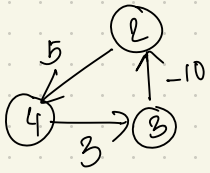
Drawback of B-F. algorithm



edges → (3,2), (4,3), (1,4), (1,2), (2,4)

→ If relaxation is done 4th time, one more vertex can still be relaxed That should not happen.

→



$5+3-10=-2$

∴ B-F. algorithm fails if the graph has a negative weight cycle.

→ B-F. can detect if there is a ⊖ve weight cycle. After $n-1$ relaxations, we can relax one more time and check if the vertices are still getting changed.

Bellman-Ford (G, w, s)

{

Initialize-single-source (G, s)

for $i \leftarrow 1$ to $|V|-1$

for each $(u,v) \in E$

Relax (v)

for each $(u,v) \in E$

if $d[v] > d[u] + c(u,v)$

return False

return True

}

$G(V, E)$

Relaxation

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

{ $d[v] = d[u] + c(u,v)$

}