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.

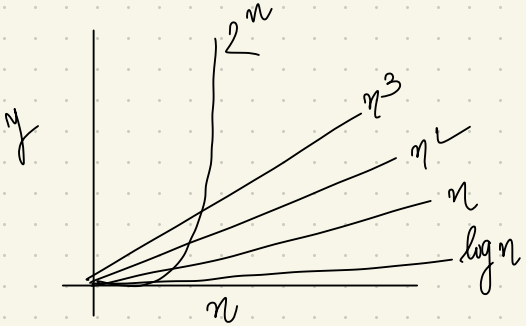
Types of time functions

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

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

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

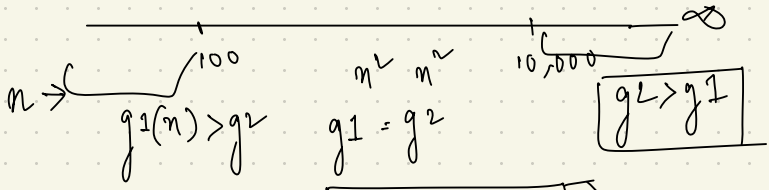
$\therefore \quad 2^n < 2^{2n}$

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

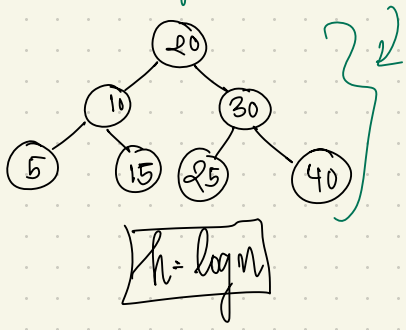
Best case – Search root element

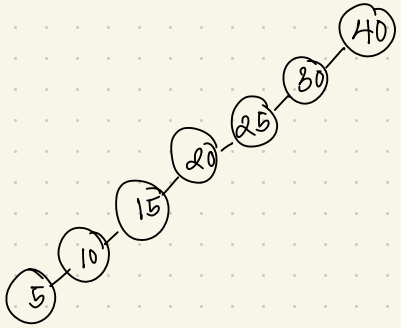
Best case time – $B(n) = 1$

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

Disjoint sets are useful for detecting cycle in non-directed graph

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

Disjoint Sets

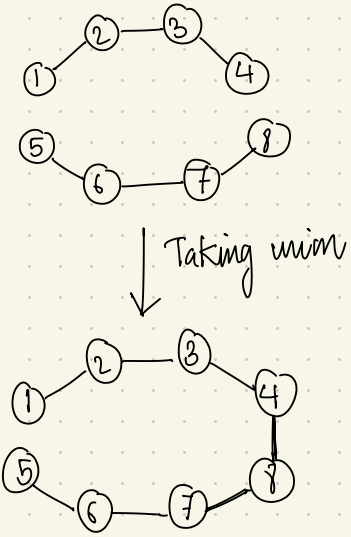
$S_1 = \{1, 2, 3, 4\}$

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

$S_1 \cap S_2 = \phi$

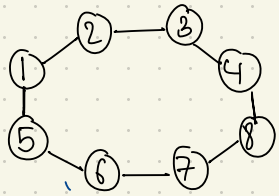
2 operations $\Rightarrow$ ① Find
② Union

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



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

$S_1 = \{1, 2\}$

$S_2 = \{3, 4\}$

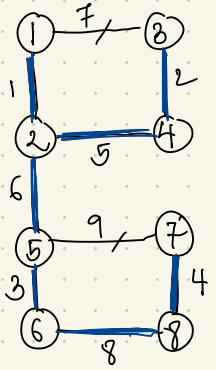
$S_3 = \{5, 6\}$

$S_4 = \{7, 8\}$

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

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

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



Steps:
1. Find
2. Same set? cycle detected
   else union

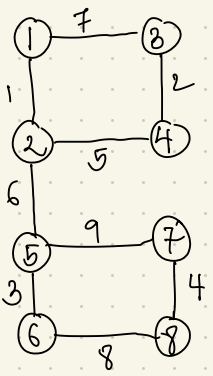
2 cycles

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

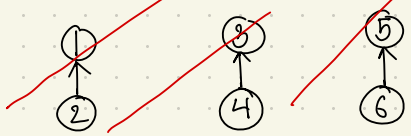
∵ (5,7) belongs to same set S_7. So, another cycle detected.

Graphical Representation

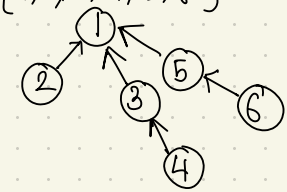
$U = \{1, 2, 3, 4, 5, 6, 7, 8\}$

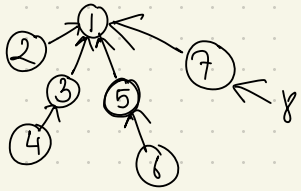


$S_1 = \{1, 2\}$ $S_2 = \{3, 4\}$ $S_3 = \{5, 6\}$ $S_4 = \{7, 8\}$



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





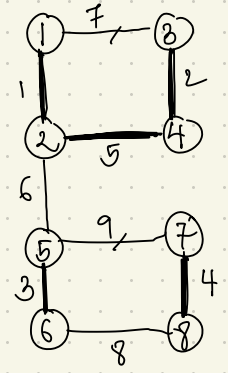
Array Representation:

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

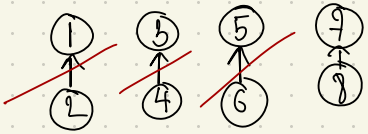
① ② ③ ④ ⑤ ⑥ ⑦ ⑧

(a) Parent

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



(b)

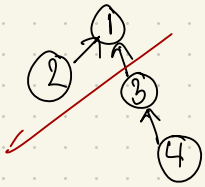


(1,2) (5,6)
(3,4) (7,8)

Parent

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

(c)



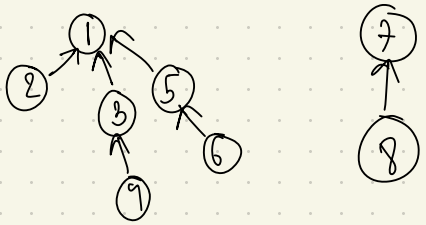
(2,4)

Parent

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

Union By Weight

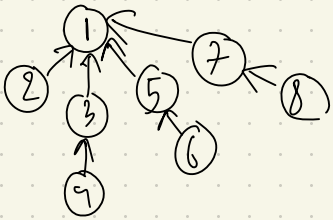
(d)



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

(e) (1,3) ⇒ Parent of both 1 and 3 is -6. If we include it, it could form a cycle. So, pass.

(f)



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

(g) (5,7) ⇒ Parent of both 5 and 7 is -8. Cycle detected.

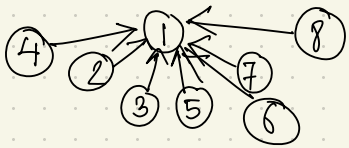
```
Kruskal(G):
    A = ϕ
    for each vertex v ∈ G.V:
        Make-set(v)
    for each edge (u,v) ∈ G.E ordered by increasing order by weight (u,v):
        if Find-set(u) ≠ Find-set(v):
            A = A ∪ {(u,v)}
            Union(u,v)

    return A
```

Collapsing Find :

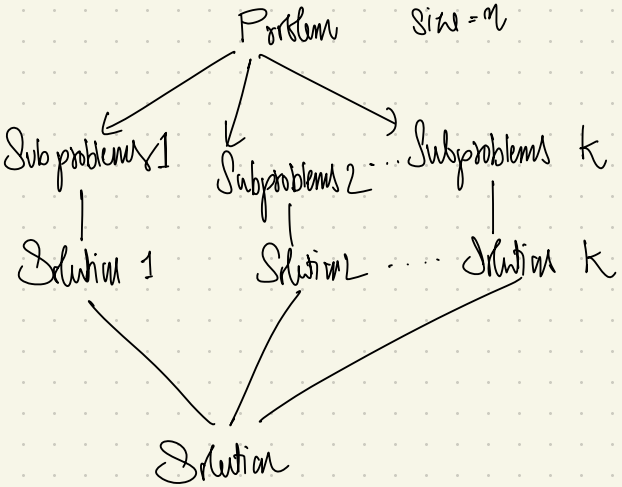
- The procedure of linking a node to the direct parent of a set is called collapsing find.
- So, the time taken to find a node can be reduced when we do it a second time.

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), ....)
    }
}
```

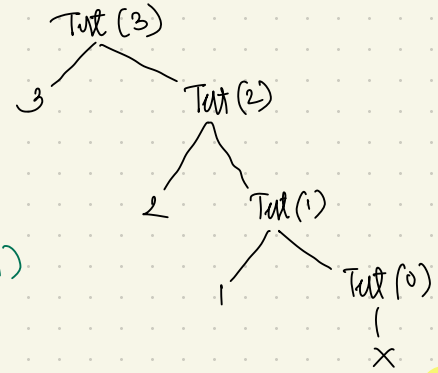
Divide & Conquer

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.

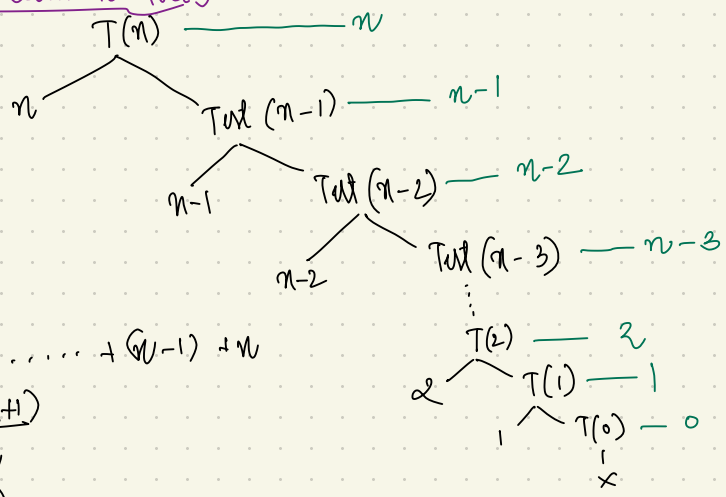
```
void Test (int n)          — T(n)
{
    if (n>0)               — 1
    {
        for (i=0; i<n; i++)
        {
            printf ("%d", n);   — n+1, n
        }
        Test (n-1);        — T(n-1)
    }
}
```

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

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

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

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 $n-k=0$

$\therefore n=k$

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

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

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

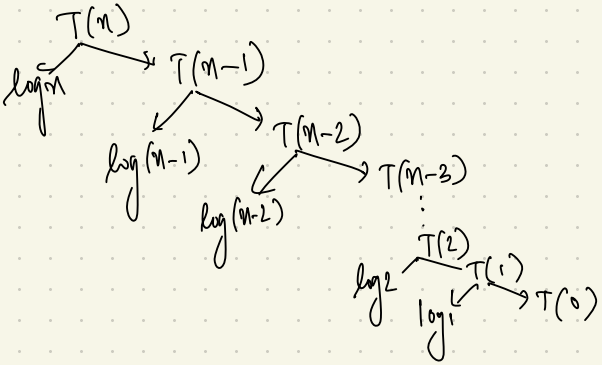
3. void Test (int n) — $T(n)$

```
{  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)

$T(n) = T(n-1) + 1$ — $O(n)$

$T(n) = T(n-1) + n$ — $O(n^2)$

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

$T(n) = T(n-1) + n^2$ — $O(n^3)$

$T(n) = T(n-2) + 1$ — $n/2$ $O(n)$

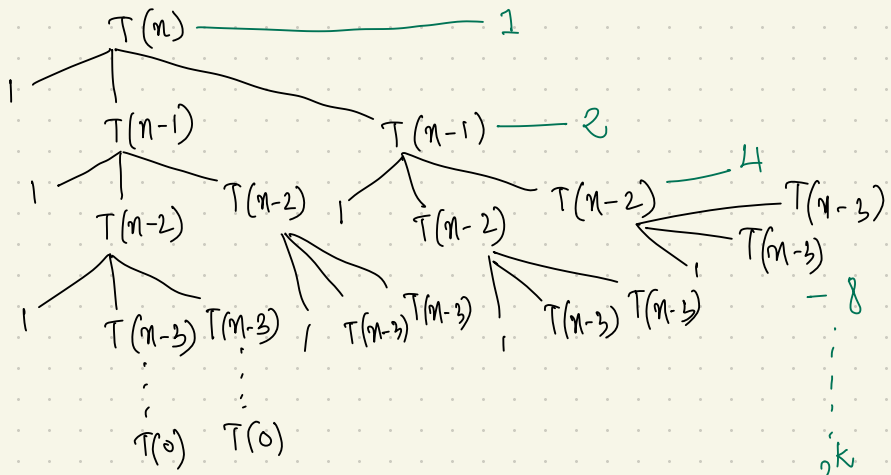
$T(n) = T(n-100) + n$ — $O(n^2)$

$T(n) = 2T(n-1) + 1$ — $O(?)$

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 $a > 1$ — $O(n^k a^{n/b})$, $O(f(n) * a^{n/b})$

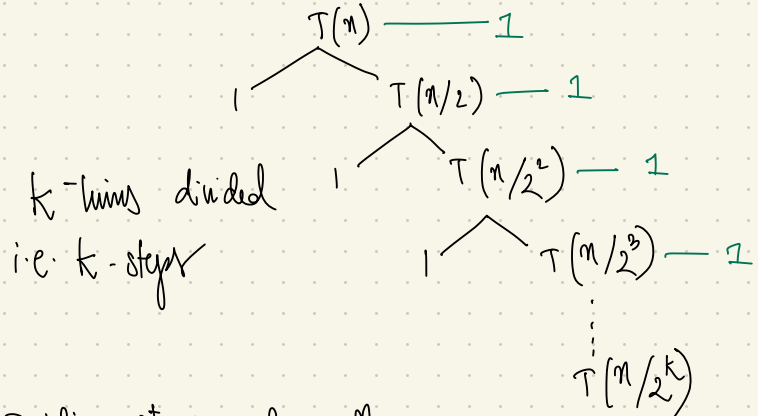
$f(n)$
$n * f(n)$
$f(n) * a^{n/b}$

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 $n/2^k = 1$

$n = 2^k$

$k = \log_2 n$

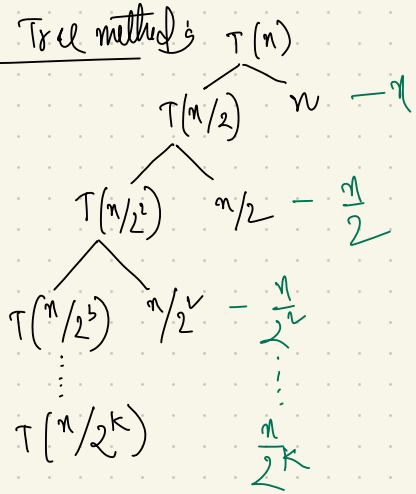
$\therefore T(n) = T(1) + \log(n)$

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

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

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

$\Rightarrow n = 2^k$

$\Rightarrow k = \log n$

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

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

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

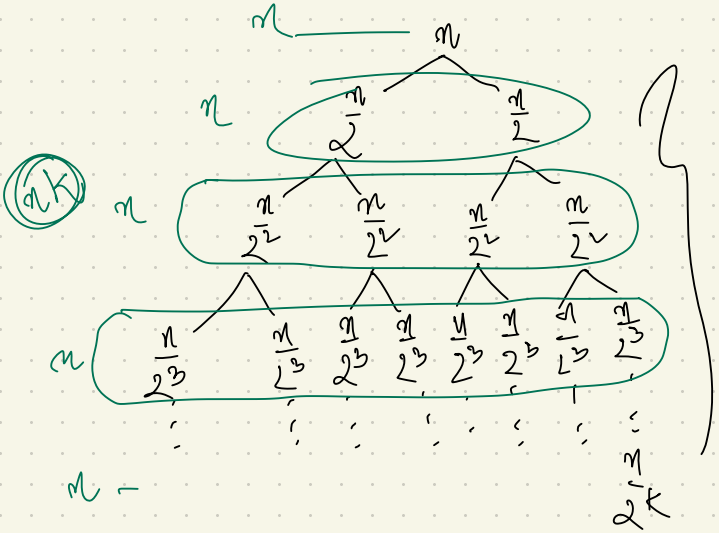
$O(n)$

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)

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

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

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

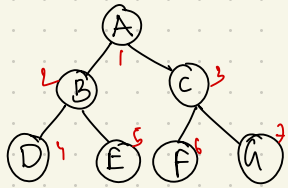
$p = 0 \Rightarrow p > -1$

$\Theta(\log n)$

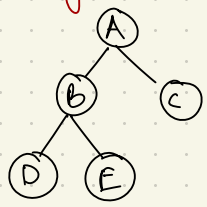
$p > -1$	$\Theta(n^k \log^{p+1} n)$
$p = -1$	$\Theta(n^k \log\log n)$
$p < -1$	$\Theta(n^k)$

HEAP

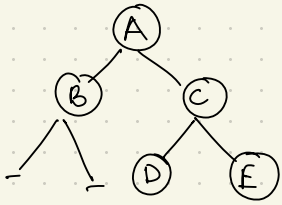
Representation of BT using array



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

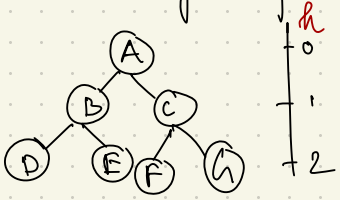
its left child is at — $2*i$

its right child is at — $2*i+1$

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

Full and Complete Binary Tree

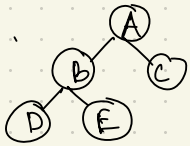
Full Binary Tree - No space for new node without introducing new level.



(*) 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.



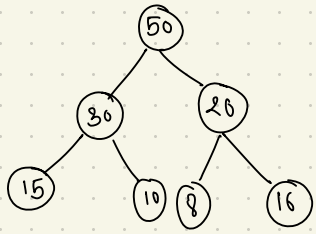
A	B	C	D	E	–	–

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 = $\lfloor \log n \rfloor$

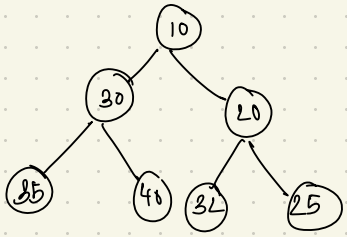
Heap

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

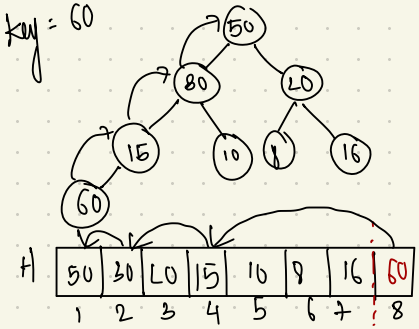
(1) Heap is a complete binary tree.

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

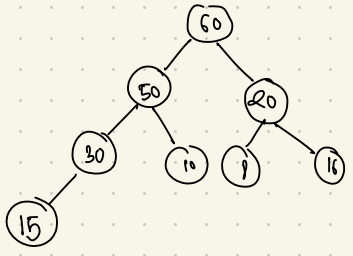
(2) Every node has a value smaller than or equal to all its descendents.

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.

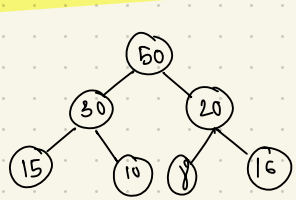
O(1) to O(log n)

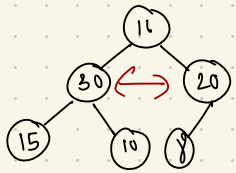
min. time — max. time

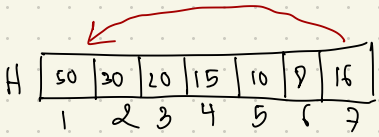
(*) Insertion → Upward Adjustment.

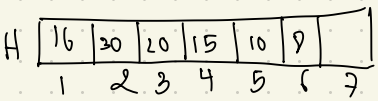
Delete opn. in max. heap

We cannot delete any other element but root element.

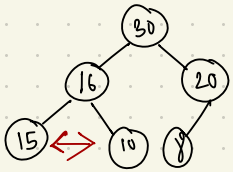






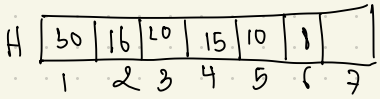


30 > 20
&
30 > 16



15 > 10
But, 16 > 15

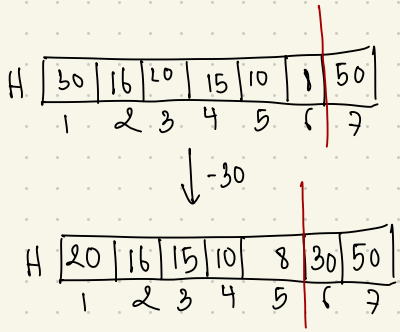
Same



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.

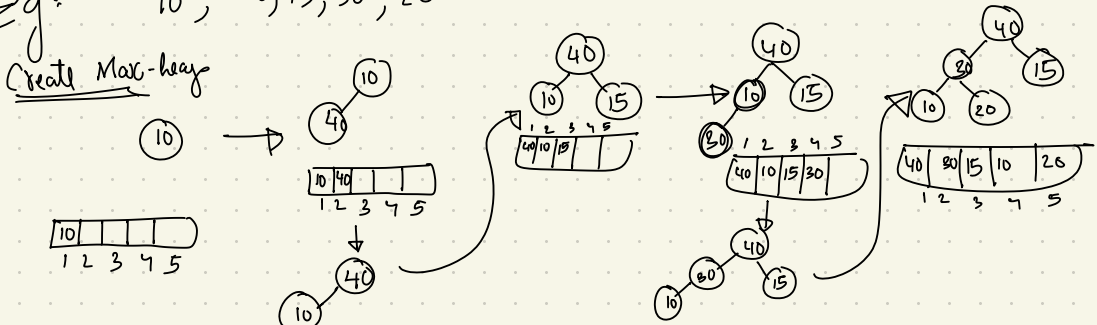


Heap Sort (increasing / decreasing)

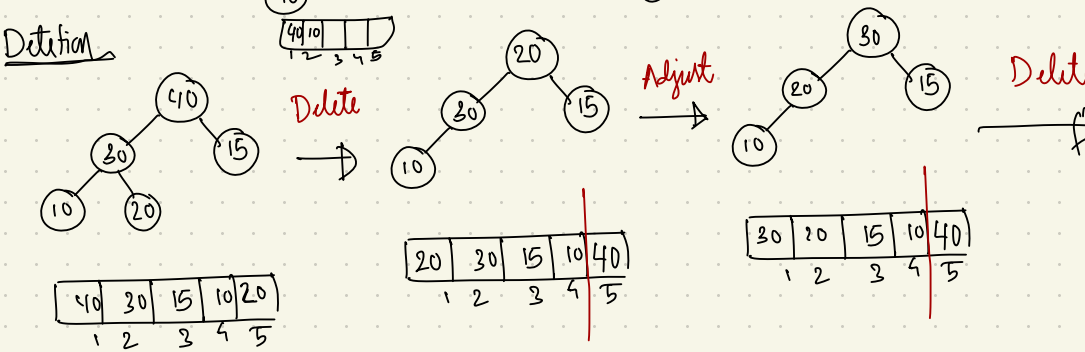
Steps: ① Create heap (min/max) from unordered array.
② Delete all elements one by one.

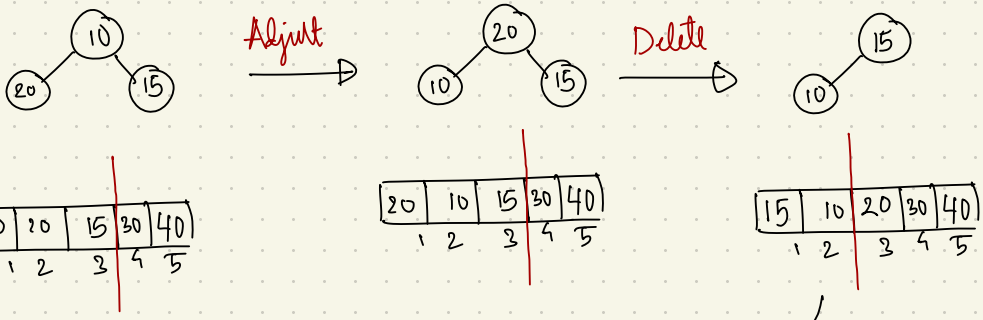
Eg: 10, 40, 15, 30, 20

Create Max-heap



Deletion

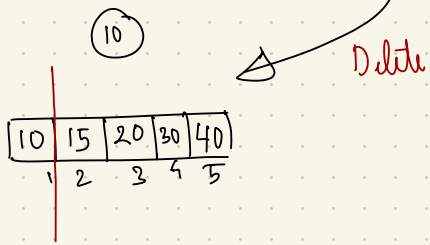




Heapify → O(n)

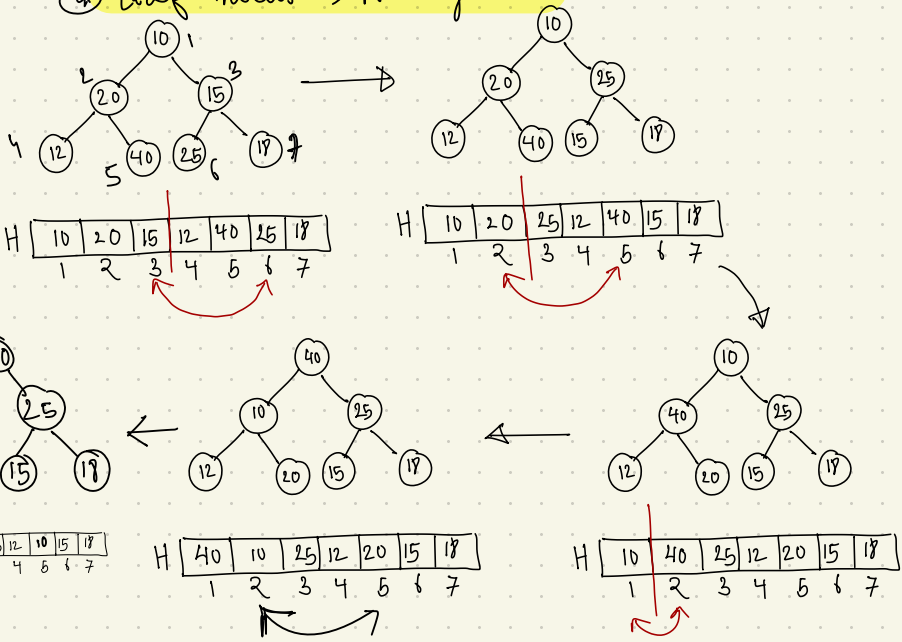
Adjust → O(log n)

Heapsort → O(n log n)



Heapify

- A procedure for creating a heap.
- Elements are inserted to the heap form right to left.
- Adjustment is done downwards.
- Leaf nodes → No adjustment.



Note: ① Creating heap by applying Insert 'n' times $\Rightarrow O(n \log n)$

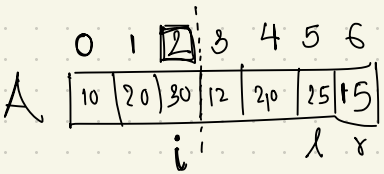
② Creating heap using Heapify $\Rightarrow O(n)$

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) {

largest = i;

l = LEFT(i);

r = RIGHT(i);

i=2, l=5, r=6, largest = K, Y

if (l < A.heap-size and A[l] > A[largest])

largest = l;

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)
    }
}
```

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

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

$f(n) = n^2$

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

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

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

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

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

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

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

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

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

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

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

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

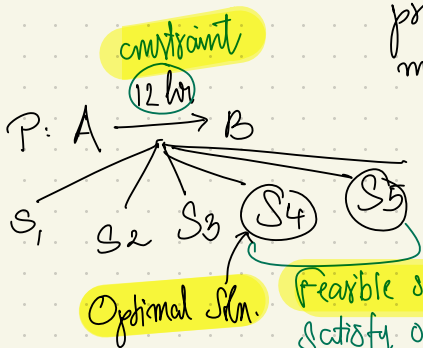
$$T(n) = \begin{cases} 1 & n = 1 \\ 7\,T(n/2) + n^2 & n > 1 \end{cases}$$

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:

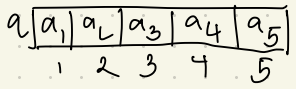
1. Greedy method
2. Dynamic programming
3. Branch and Bound

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



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 + P_i (M / w_i);
        }
    }
}
```

* Time complexity after sorting → $O(n)$
* Time complexity including sorting → $O(n \log n)$

Job Sequencing with deadlines

→ n - jobs / programs / process

→ Each job arrives at time 0 and needs one unit time.

→ Every job is associated with a deadline d_i & profit p_i. The profit is obtained only if the job is completed within its deadline.

Problem statement:
Select a subset of given n-jobs such that the jobs in the subset are completable within their deadlines and generate maximum profit.

→ Here the trivial case is when all the jobs can be scheduled.

→ Size of soln. space is 2^n
It means working out this problem using brute force, the time complexity is $O(2^n)$.

→ Feasible solution for this problem are those subsets having all the jobs which can be scheduled within the deadline. (Implicit constraint)

eg. $n=4$; $(J_1 - J_4)$;

$(P_1 - P_4) = \langle 100, 15, 10, 40 \rangle$

$\langle d_1, d_2, d_3, d_4 \rangle = \langle 2, 1, 2, 1 \rangle$

Let J be soln. subset. Initially $J = \phi$

I. $|J| = 0$

feasible soln. profit, $p = 0$

II. $|J| = 1$

$J = \{J_1\}$ feasible soln. profit, $p = 100$

$J = \{J_2\}$ feasible soln. profit, $p = 15$

III. $|J| = 2$

$J = \{J_1, J_2\}$

feasible soln.

profit, $p = 100 + 15 = 115$

schedule: J_2, J_1

$J = \{J_1, J_3\}$

feasible soln.

profit, $p = 100 + 10 = 110$

schedule: J_1, J_3 (or) J_3, J_1

$J = \{J_2, J_4\}$

not feasible soln.

∴ we cannot schedule J_2 and J_3 within their deadlines.

IV. $|J|=3$

no three jobs can be scheduled.

∴ there is no feasible soln. with $|J|=3$

Note: The maximum no. of jobs that a feasible soln. can have = maximum deadline.

(This rule applies only when each job takes 1 unit of time).

GREEDY APPROACH for JSD

$n=4$; $\langle J_1 - J_4 \rangle$; $\langle P_1 - P_4 \rangle = \langle 100, 15, 10, 40 \rangle$; $\langle d_1 - d_4 \rangle = \langle 2, 1, 2, 1 \rangle$

Let J be the soln. subset.

→ The job with highest profit will always be part of optimal soln.

Step (i) Arrange the jobs in decreasing order of profit.

i.e. J_1, J_4, J_2, J_3

(ii) One by one schedule the jobs of sorted order such that the job gets into maximum feasible slot before its deadline.

∴ Pick J_1

deadline is 2.

slot 2 is free.

∴

	J_1

Pick J_4

deadline is 1

∴ slot 1 is free.

J_4	J_1

Pick J_2:

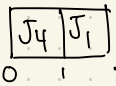
Slot 1 is not empty.

Pick J_3:

Slot 2 is not empty. Now check slot 1.

Slot 1 is not empty.

∴ final schedule



∴ profit = 100 + 40 = 140

eg. $n = 7$

$\langle J_1 - J_7 \rangle$

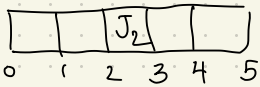
$\langle d_1 - d_7 \rangle = \langle 2, 3, 4, 5, 2, 4, 4 \rangle$

$\langle p_1 - p_7 \rangle = \langle 54, 92, 30, 10, 15, 60, 45 \rangle$

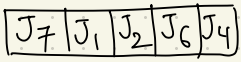
decreasing order of profits $J_2, J_6, J_1, J_7, J_3, J_5, J_4$

J_3 ✗ J_5 ✗

Pick J_2:



Pick J_6:



eg. $n = 6$ $\langle J_1 - J_6 \rangle$

$\langle d_1 - d_6 \rangle = \langle 4, 4, 3, 4, 2, 3 \rangle$

$\langle p_1 - p_6 \rangle = \langle 105, 39, 46, 150, 20, 14 \rangle$

1 2 3 4 5 6

$J_4, J_1, J_3, J_2, J_5, J_6$ $\quad p = 105 + 39 + 46 + 150$

0 $\underline{J_2}$ 1 $\underline{J_3}$ 2 $\underline{J_1}$ 3 $\underline{J_4}$ 4 $\quad = 339$

Control Abstraction of Greedy Method:

```
Procedure Greedy (A, n)        → i/p size
{
  1. solution ← ϕ
  2. for i ← 1 to n
     {
        x ← select (A);
        if (FEASIBLE (x, solution)) // checks whether including
                                    // x in soln. is feasible
           ADD (x, solution);       // or not
     }
     return (solution);
}
```

From this control abstraction, we say minimum time complexity is $O(n)$. i.e. when SELECT(), FEASIBLE(), ADD() are $O(1)$.

High-level Pseudocode of JSD

```
Algorithm JSD (d, p, J, n)
// we assume that the jobs are arranged in decreasing
   order of profit  p1 ≥ p2 ≥ p3 ..... ≥ pn
{
  1. J ← {1}
  2. for i ← 2 to n ⟶ O(n)
     { if (all jobs in J ∪ {i} can be completed by their
           deadlines) then                    ↳ O(n)
           J ← J ∪ {i};
     }
}
```

Time complexity of JSD : $O(n^2)$

→ The working of JSD is similar to the worst case behaviour of insertion sort.

Algorithm JS(d, j, n)

// d[i] = deadline $1 \le i \le n$

// jobs are ordered such that $p[1] \ge p[2] \ge \dots \ge p[n]$

// J[i] = ith job in the optimal solution. $1 \le i \le k$

// At termination $d[J[1]] \le d[J[2]] \le \dots \le d[J[k]]$

```
{
    d[0] = J[0] = 0;
    J[1] = 1;          // first job is always in optimal solution.
    k ← 1;             // Holds index of last job in the solution.
    for i ← 2 to n
    {
        r ← k;         // Finding the position of newly inserted job. i.e. r+1
        while ( d[J[r]] > d[i] and (d[J[r]] ≠ r) )
            r = r - 1;
        // Checking position of new job should begin from right.
        // New job always comes on RHS.
        // Making sure deadline is in increasing order. We are calculating r. (r+1) will be the index of our new job.
        // We don't need a job from original index if its deadline equals its index
    }
}
```

// Checking feasibility

```
if((d[J[r]] <= d[i]) and (d[i] > r))
{
```

Deadline of the new job must be greater than the deadline of the job that is going to be on its left.

Deadline of the New job must be greater than the index on the left.

```
    for q <- k to (r+1) step -1
    {
        J[q+1] <- J[q];
    }
    J[r+1] <- i; k <- k+1;
}
}
return (k);
}
```

← No. of jobs selected.

OPTIMAL MERGE PATTERN

Merge A & C

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

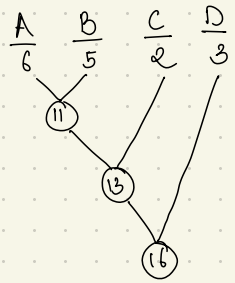
$\theta(m+n)$

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

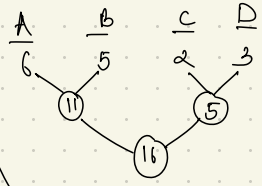
Eg.

List →	A	B	C	D
Sizes →	6	5	2	3

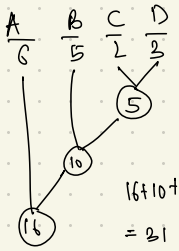
* We have to merge the four lists using 2-way merging.



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



11+5+16 = 32

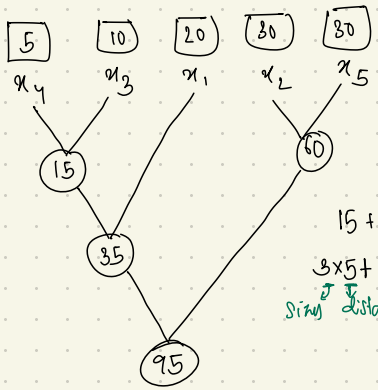


16+10+5 = 31

Eg.

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

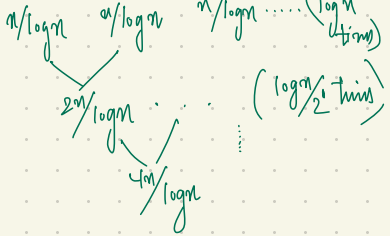
Ans.



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

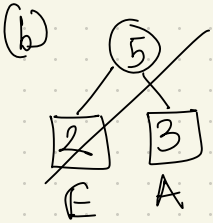
115 bits

Variable Size Code :

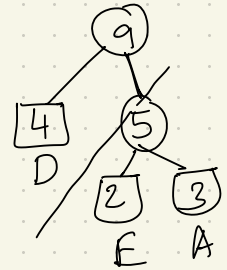
Message → BCCABB DDAE CC BB AE DD CC

(a) E, A, D, B, C
2, 3, 4, 5, 6

(b)

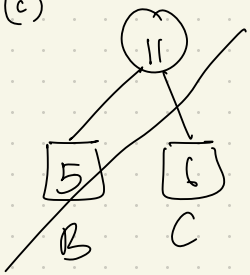


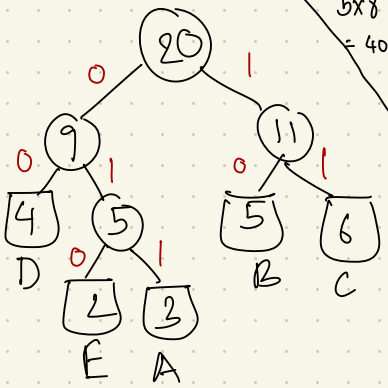
(c)



Char	Count	Code	
A	3	011	3×3 = 9 bits
B	5	10	2×5 = 10 ,
C	6	11	2×6 = 12
D	4	00	2×4 = 8
E	2	010	3×2 = 6
5×8 bits = 40 bits	20	12 bit	45 bits

(c)





Left → 0
Right → 1

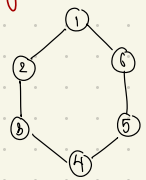
(*) Msg - 45 bits
Tree/Table - 52 bits
97 bits

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

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

Spanning Tree :

Eg:



$G = (V, E)$

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

$E = \{(1,2), (2,3), (3,4), \ldots\}$

$|V| = 6$

$\Rightarrow$

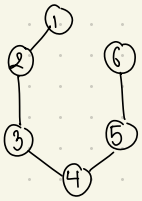
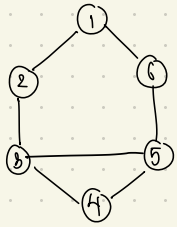


Fig: Spanning Tree (S)

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

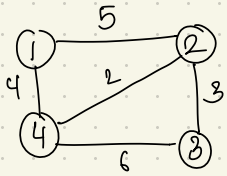
Eg.



$^7C_5 - 2$

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

Weighted Graph



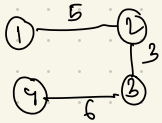
No. of possible spanning trees $= {}^{|E|}C_{|V|-1} -$ no. of cycles

$= {}^5C_3 - 2$

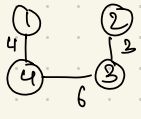
$= 8$

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

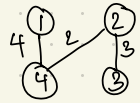
Possible Sp. trees :-



cost = 14



cost = 13



cost = 9

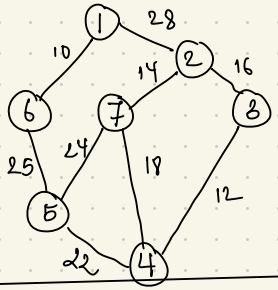
Minimum Cost Spanning Tree

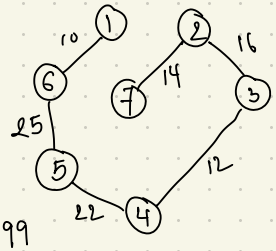
2 methods of finding M.C.S.T

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.



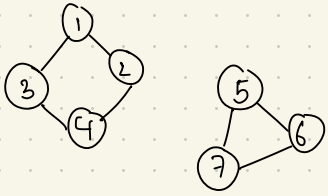


Cost = 99

Pseudocode: U = visited vertices
V-U = unvisited vertices
T = selected edges

```
T = ϕ
U = {1};
while (U ≠ V)
    let (u,v) be the lowest cost edge
    such that u ∈ U and v ∈ V - U
    T = T ∪ {(u,v)}
    U = U ∪ {v}
```

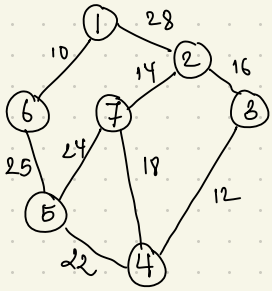
Eg:

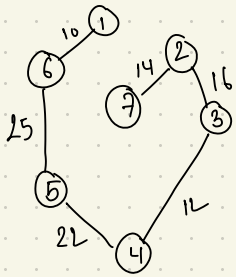


No algorithm can find min. cost sp. tree for unconnected graph.

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 = $\log n$.

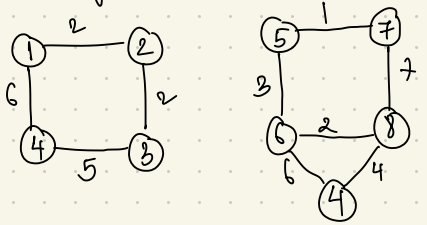
If it is repeated 'n' times,

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

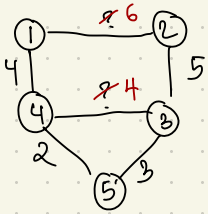
Kruskal's Algorithm on Non-connected Graph

It may find the spanning tree of each component.

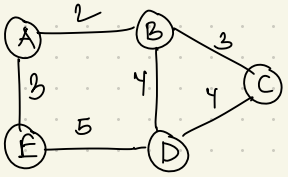
Eg:



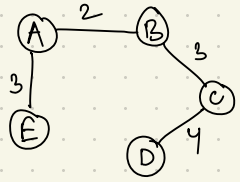
Missing Edges



Q. Find min cost sp. tree.



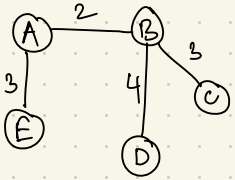
Ans. By Kruskal's



$^5C_4 - 2$ sp. trees.

3+2+3+4 = 12

OR

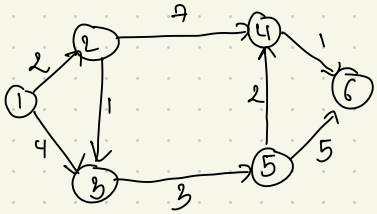


3+2+3+4 = 12

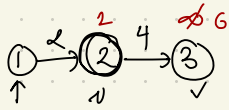
Only one optimal answer.

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

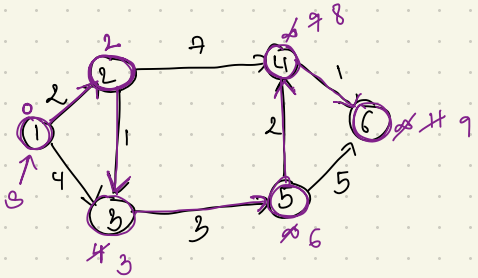
$d[v] = d[u] + c(u,v)$

$d[u]$ = distance of vertex u

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

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

Eg.



	1	2	3	4	5	6
1	0	2	4	9	8	9
2	0	0	1	7	6	8
3	0	0	0	8	3	9
4	0	0	0	0	5	1
5	0	0	0	2	0	3
6	0	0	0	0	0	0

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

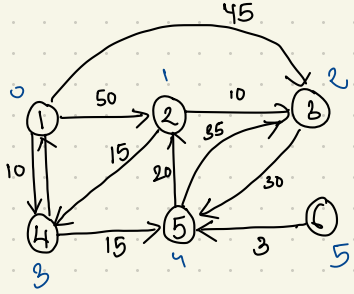
$n = |V|$

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

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

$= \theta(n^2)$

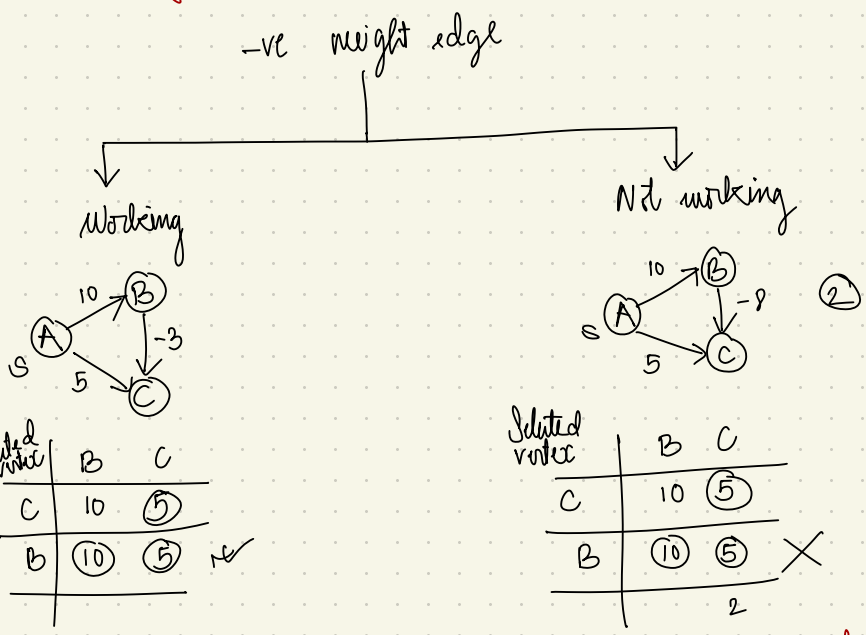
Eg: starting vertex 1



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

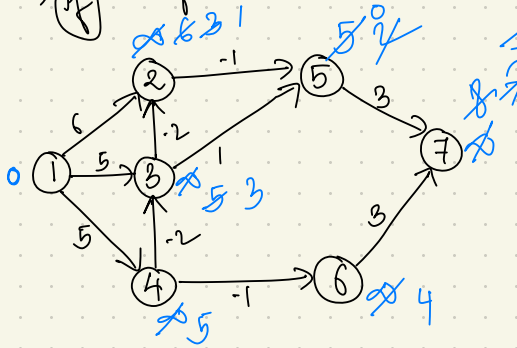
Selected vertex	B	C
C	10	5
B	10	5
		2

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 = $|V| - 1$

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

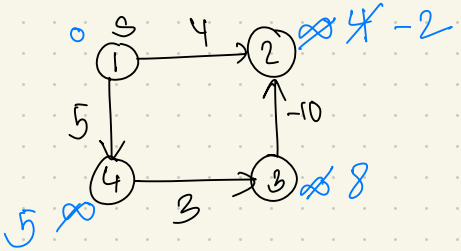
$n C_2 = \frac{n!}{(n-2)! \, 2!}$

→ 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.

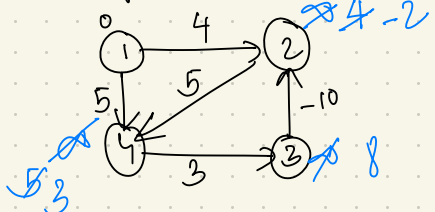


$n = 4$

$n-1 = 3$ times relaxation

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

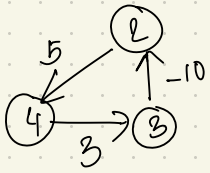
Drawback of B-F. algorithm



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

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

}

DYNAMIC PROGRAMMING

→ DP is an algorithm design technique used for solving problems which solutions are viewed as a result of a sequence of decisions.

→ One way of solving these problems is by making decisions at every step based on local information available at that step. This is true for problems solved by greedy method

→ But for many problems, like multistage graphs, optimal solution cannot be achieved by making decisions based on local information in a stepwise manner.

→ One way of solving the problems for which optimal solutions cannot be made based on local information, is to enumerate all possible decision sequences and pick up the best. This is known as brute force (enumeration strategy) approach. However, for this strategy, the drawback is that it has very large time and space complexity.

→ DP is based on enumeration, but it often tries to curtail the amount of enumeration by removing/avoiding those sequences that are not feasible or sub-optimal. Due to this, there may be a significant improvement in time complexities.

→ DP makes the sequence of decisions by applying principle of optimality / Global optimality / optimality substructure.

→ Principle of optimality states that whatever the initial state and decision are, the remaining sequence of decisions must be optimal with regard to the state resulting from the current decision.

→ Any problem that says principle of optimality and overlapping subproblems can be solved by DP.

→ The fundamental difference between GM and DP is that GM generates only one decision sequence whereas DP may generate multiple decision sequences.

→ In DP, the results of subproblems can be tabulated & reused. (Memoization)

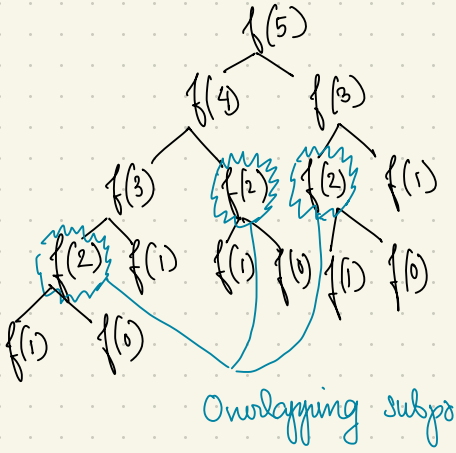
→ Thus DP optimizes time by curtailing decisions that lead to non-feasible or sub-optimal solution and by tabulating results of subproblems.

Eg. Consider computing nth fibonacci number.

$fib(0) = 0$

$fib(1) = 1$

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



Overlapping subproblems.

→ Regular recursive implementation

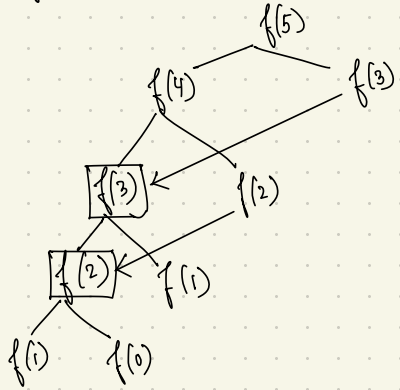
```
int fib(n)
{
    if (n <= 1)  return n;
    else
        return fib(n-1) + fib(n-2);
}
```

TC: $O(2^n)$ Space comp: $O(n)$

Properties of problems that can be solved by DP:

1. Optimal substructure
2. Overlapping subproblems.

→ Top down DP with memoization:



Rather than tree, a closed graph is formed without cycle.

→ DP problems can be solved in 2 ways:

- Top-down approach (Memoization)
  - Caches the result of each function call.
  - Recursive implementation.
  - Well suited for problems with relatively small set of inputs.
  - Used when problems have overlapping subproblems.
- Bottom-up approach (Tabulation)
  - Stores the result of subproblems in a table.
  - Iterative implementation.
  - Well suited for problems with a large set of inputs.
  - Used when problems do not overlap.
  - More efficient in terms of T.C.

Note:

Greedy: Builds up a solution incrementally by optimizing at each step some local criteria.

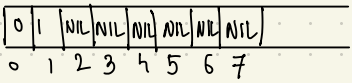
D&C: Break down a problem into separate subproblems hierarchically & solving each subproblem independently and combining solutions to subproblems to get solution of original problem.

DP: Breaks up a problem into a series of overlapping subproblems and build up solutions to larger & larger subproblems.

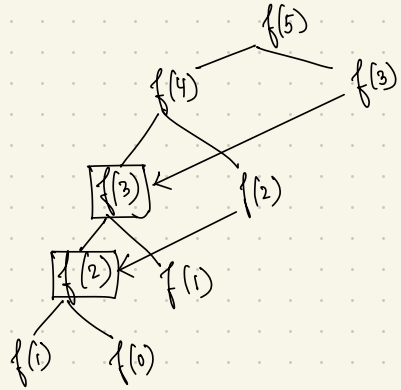
- Unlike D&C, DP involves solving all sub-problems
- DP solves each subproblem only once & then stores the result so that it can be reused when required.

Top-down approach (Memoization) for nth fibonacci

initialize array f[] as shown below




```
int fib(n)
{
    if(f[n] == NIL)
        f[n] = fib(n-1) + fib(n-2)
    return (f[n]);
}
```



T.C: O(n) ∵ for every number fib(n) is computed only once.

S.C: O(n) i.e. size (array) + size (stack)

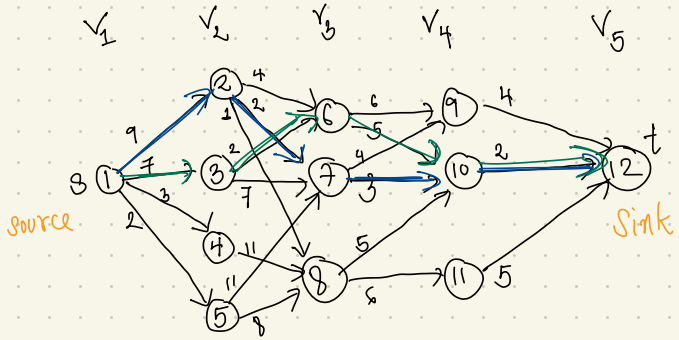
Bottom-up approach (Tabulation) for nth fibonacci number

```
f[1000] : int array
f[0] = 0 ; f[1] = 1 ;
int fib(n)
{   for i = 2 to n
        f[i] = f[i-1] + f[i-2] ;
    return (f[i]) ;
}
```

T.C: O(n)

S.C: O(n) [∵ size of array]

MULTI STAGE GRAPH



V	1	2	3	4	5	6	7	8	9	10	11	12
Cost	16	7	9	18	15	7	5	7	4	2	5	0
d	2/3	7	6	7	8	10	10	10	12	12	12	12

destination of newly selected edge

cost(5, 12) = 0

stage, vertex

cost(4, 9) = 4
cost(4, 10) = 2
cost(4, 11) = 5

cost(3, 6) = min { c(6, 9) + cost(4, 9), c(6, 10) + cost(4, 10) }
= min { 6+4, 5+2 }
= 7

start table filling from last stage

$$cost(3,7) = \min\{ c(7,9) + cost(4,9),\ c(7,10) + cost(4,10) \}$$

cost of the edge — cost of the vertex

$$= \min\{4+4,\ 3+2\} = 5$$

$$cost(3,8) = \min\{ c(8,10) + cost(4,10),\ c(8,11) + cost(4,11) \}$$

$$= \min\{5+2,\ 6+5\} = 7$$

$$cost(2,2) = \min\{ c(2,6) + cost(3,6),\ c(2,7) + cost(3,7),\ c(2,8) + cost(3,8) \}$$

$$= \min\{4+7,\ 2+5,\ 1+7\}$$

$$= \min\{11, 7, 8\}$$

$$= 7$$

$$cost(1,1) = \min\{ c(1,2) + cost(2,2),\ c(1,3) + cost(2,3),\ c(1,4) + cost(2,4),\ c(1,5) + cost(2,5) \}$$

$$= \min\{\underbrace{9+7}_{16},\ \underbrace{7+9}_{16},\ \underbrace{3+18}_{21},\ \underbrace{2+15}_{17}\}$$

$$= 16$$

$$Cost(i,j) = \min_{\substack{<j,l> \in E \\ l \in V_{i+1}}} \{ c(j,l) + Cost(i+1, l) \}$$

stage — vertex — destination

$d(1,1) = 2$

$d(2,2) = 7$

$d(3,7) = 10$

$d(4,10) = 12$

$d(1,1) = 3$

$d(2,3) = 6$

$d(3,6) = 10$

$d(4,10) = 12$

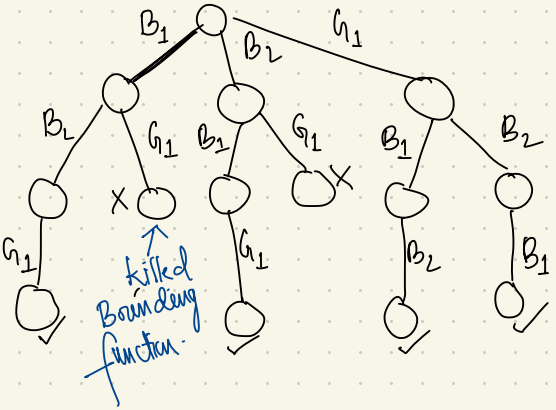
Backtracking (Brute - Force Approach)

→ Backtracking & (B&B) are algorithm design strategies that use DFS & BFS to find feasible/optimal soln. in the solution space (State space tree)

→ Backtracking is used when we have multiple soln. & we want all those soln.

eg. B_1 B_2 G_1

State Spaced Tree



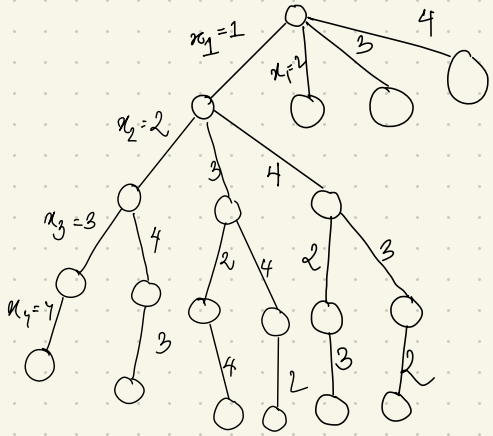
Constraint : Girl should not sit in the middle.

Problems

1. N-Queens Problem
2. Sum of the subsets problem
3. Graph Coloring Problem

Note: Brute Force (DFS) Approach is also used in Branch & Bound (BFS) Strategy.

N- Queens Problem



q_1, q_2, q_3, q_4

	1	2	3	4
1		q_1		
2				q_2
3	q_3			
4			q_4	

x_i	2	4	1	3
	1	2	3	4

Column No.

→ The possibilities of two queens being in the same row or same column is ruled out.

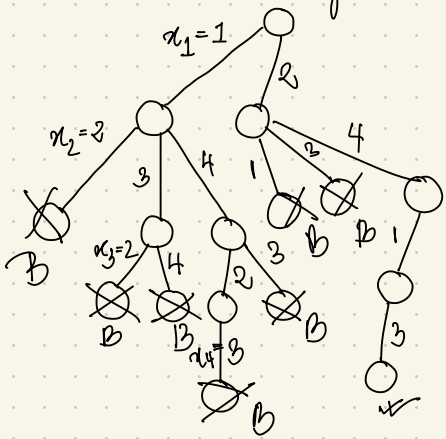
∴ No. of rows when we avoid same row & same column. But, we are allowing diagonals $= (1+4+4\times3+4\times3\times2+4\times3\times2\times1)$

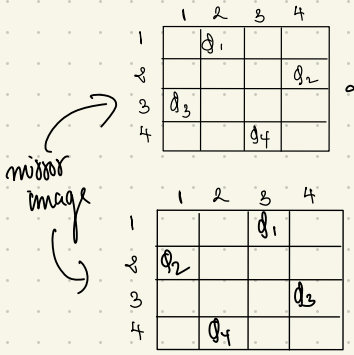
$= 1+\sum_{i=0}^{3}\left[\prod_{j=0}^{i}(4-j)\right] = 65$ (4×7)

→ In general, for N×N chessboard

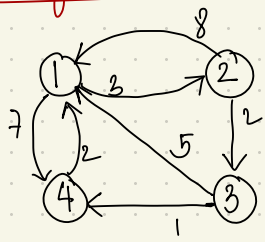
No. of nodes $= 1+\sum_{i=0}^{N-1}\left[\prod_{j=0}^{i}(N-j)\right]$

→ Bounding Function: Nil in same row, same column or same diagonal.





All pairs shortest path (Floyd - Warshall)

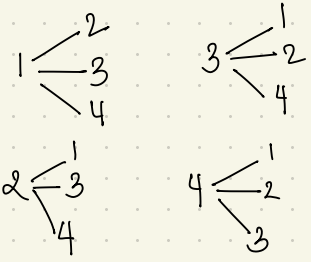


$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

Original matrix

Using Dijkstra algorithm on all 'n' vertices →

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



$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & \infty & 0 \end{bmatrix} \end{array}$$

Intermediate vertex → (1)

1st row & 1st col. remains as it is.

$A^0[2,3] \quad A^0[2,1] + A^0[1,3]$

$2 \quad < \quad 8 + \infty$

$A^0[2,4] \quad A^0[2,1] + A^0[1,4]$

$\infty \quad > \quad 8 + 7$

$A^0[3,2] \quad A^0[3,1] + A^0[1,2]$

$\infty \quad > \quad 5 + 3$

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{array}$$

Intermediate vertex → (2)

2nd row & 2nd column remains as it is.

$A^1[1,3] \quad A^1[1,2] + A^1[2,3]$

$\infty \quad > \quad 3 + 2$

$A^1[1,4] \quad A^1[1,2] + A^1[2,4]$

$7 \quad < \quad 3 + 15$

$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 5 & 6 \\ 7 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ 2 & & 7 & 0 \end{bmatrix} \end{array}$$

$$A^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 3 \\ 3 & 6 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{array}$$

$$A^k[i,j] = \min\{A^{k-1}[i,j],\ A^{k-1}[i,k] + A^{k-1}[k,j]\}$$

```
for (k=1 ; k<=n; k++)
{   for (i=1 ; i<=n ; i++)
    {   for (j=1 ; j<=n ; j++)
        {   A[i,j] = min ( A[i,j] , A[i,k]+ A[k,j])
        }
    }
}
```

$\Theta(n^3)$

Matrix Chain Multiplication

$c[1,3]$ $A_1 \times A_2 \times A_3$

$2\ 3 \quad 3\ 4 \quad 4\ 2$

$d_0\ d_1 \quad d_1\ d_2 \quad d_2\ d_3$

We want to find out the no. of multiplications we have to perform.

$(A_1 \times A_2) \times A_3$

$2\times3 \quad 3\times4 \quad 4\times2$

$2 \quad 4 \quad 4\ 2$

$24 + 0 + 2\times4\times2 = \boxed{40}$

$A_1 \times (A_2 \times A_3)$

$2\times3 \quad 3\times4 \quad 4\times2$

$2\ 3 \quad 3 \quad 2$

$3\times4\times2 = 24$

$0 + 24 + 2\times3\times2 = \boxed{36}$

$(A_1 \times A_2) \times A_3$

$2\times3 \quad 3\times4 \quad 4\times2$

$c[1,2] = 24 \quad c[3,3] = 0$

$2 \quad 4 \quad 4 \quad 2$

$d_0\ d_2 \quad d_2\ d_3$

$c[1,2] + c[3,3] + d_0 \times d_2 \times d_3 = 40$

$i\ k \quad k+1\ j \quad d_{i-1}\ k\ d_j$

$A_1 \times (A_2 \times A_3)$

$2\times3 \quad 3\times4 \quad 4\times2$

$c[1,1] = 0 \quad c[2,3] = 24$

$2\ 3 \quad 3 \quad 2$

$d_0\ d_1 \quad d_1\ d_3$

$c[1,1] + c[2,3] + d_0 \times d_1 \times d_3 = 36$

$i\ k \quad k+1\ j \quad d_{i-1}\ k\ d_j$

Cost of multiplying matrix A_i to A_j

$$c[i,j] = \min_{i \le k < j} \{ c[i,k] + c[k+1, j] + d_{i-1} \times d_k \times d_j \}$$

eg. $A_1 \times A_2 \times A_3 \times A_4$

$d_0\ d_1 \quad d_1\ d_2 \quad d_2\ d_3 \quad d_3\ d_4$

Catalan No. $= \frac{2nC_n}{n+1}$

Catalan number $= \frac{2(n-1)C_{n-1}}{n}$

n = no. of matrices

1. $A_1 (A_2 (A_3 A_4))$
2. $A_1 ((A_2 A_3) A_4)$
3. $(A_1 A_2)(A_3 A_4)$
4. $(A_1 (A_2 A_3)) A_4$
5. $((A_1 A_2) A_3) A_4$

$$c[1,4] = \min_{1 \le k < 4} \begin{cases} k=1 & c[1,1] + c[2,4] + d_0 \times d_1 \times d_4 \\ k=2 & c[1,2] + c[3,4] + d_0 \times d_2 \times d_4 \\ k=3 & c[1,3] + c[4,4] + d_0 \times d_3 \times d_4 \end{cases}$$

(i) (ii)

$k=1$: $A_1 (A_2 A_3 A_4)$

$k=2$: $(A_1 A_2)(A_3 A_4)$

$k=3$: $(A_1 A_2 A_3) A_4$

$$c[2,4] = \min_{2 \le k < 4} \begin{cases} k=2 & c[2,2] + c[3,4] + d_1 \times d_2 \times d_4 \\ k=3 & c[2,3] + c[4,4] + d_1 \times d_3 \times d_4 \end{cases}$$

C

i ↓ \ j →	1	2	3	4
1	0			
2		0		
3			0	
4				0

K

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

q.

$A_1 \times A_2 \times A_3 \times A_4$

$\underset{d_0}{3} \quad \underset{d_1}{2 \quad 4} \quad \underset{d_2}{4 \quad 2} \quad \underset{d_3}{2 \quad 5} \quad d_4$

C	1	2	3	4	
1	0	24	28	58	4
2		0	16	36	3
3			0	40	2
4				0	1

$1+2+3+4 = \frac{4(5)}{2}$

k	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

Start with Diagonal with difference $j-i=1$.

$c[1,1] = \min_{1 \le k < 1} = 0$

$c[1,2] = \min_{1 \le k < 2}$ $k=1$ { $c[1,1] + c[2,2] + d_0 \times d_1 \times d_2$ }

$0 + 0 + 3 \times 2 \times 4$

$c[2,3] = \min_{2 \le k < 3}$ $k=2$ { $c[2,2] + c[3,3] + d_1 \times d_2 \times d_3$ }

$0 + 0 + 2 \times 4 \times 2$

$= 16$

$c[3,4] = \min_{3 \le k < 4}$ $k=3$ { $c[3,3] + c[4,4] + d_2 \times d_3 \times d_4$ }

$0 + 0 + 4 \times 2 \times 5$

$= 40$

Difference $j-i=2$

$c[1,3] = \min_{1 \le k < 3}$

$k=1$: $c[1,1] + c[2,3] + d_0 \times d_1 \times d_3 = 28$ ✓

$0 + 16 + 3 \times 2 \times 2$

$k=2$: $c[1,2] + c[3,3] + d_0 \times d_2 \times d_3 = 44$

$24 + 0 + 3 \times 4 \times 2$

$c[2,4] = \min_{2 \le k < 4}$

$k=2$: $c[2,2] + c[3,4] + d_1 \times d_2 \times d_4 = 80$

$k=3$: $c[2,3] + c[4,4] + d_1 \times d_3 \times d_4 = 36$ ✓

$j-i=3$

$c[1,4] = \min_{1 \le k < 4}$

$k=1$: $c[1,1] + c[2,4] + d_0 \times d_1 \times d_4 = 96$

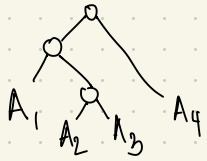
$k=2$: $c[1,2] + c[3,4] + d_0 \times d_2 \times d_4 = 124$

$k=3$: $c[1,3] + c[4,4] + d_0 \times d_3 \times d_4 = 58$ ✓

Parenthesization based on k table

$(A_1\ A_2\ A_3)\ (A_4)$

$((A_1).(A_2\ A_3)).(A_4)$



Time taken to find parenthesization

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

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

∴ Since computation is done using different k values.

$A_1 \times A_2 \times A_3 \times A_4$

$5\times4 \quad 4\times6 \quad 6\times2 \quad 2\times7$

```
main()
{
  int n=5;
  int p[]={5,4,6,2,7};
  int m[5][5]={0};
  int s[5][5]={0};
  int j, min, q;
  for(int d=1; d<n-1; d++)
  {
    for(int i=1; i<n-d; i++)
    {
      j=i+d;
      min=32767;
      for(int k=i; k<=j-1; k++)
      {
        q=m[i][k]+m[k+1][j]+p[i-1]*p[k]*p[j];
        if(q<min)
        {
          min=q;
          s[i][j]=k;
        }
      }
      m[i][j]=min;
    }
  }
  cout<<m[1][n-1];
}
```

n=5

d<4

1-3

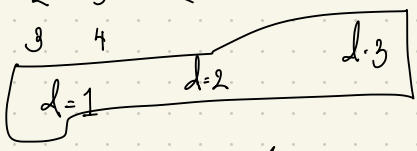
p

0	1	2	3	4
5	4	6	2	7

i	j
1	2
2	3
3	4

i	j
1	3
2	4

i	j
1	4



M

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	120	88	
2	0		0	48	
3	0			0	84
4	0				0

d=2 i=1 j=3

d=2 i=2 j=4

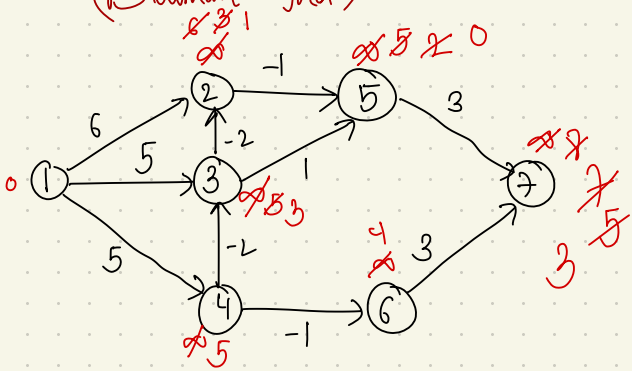
k=2

k=3

s

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

Single Source Shortest Path (Bellman - Ford)



1 - 0
2 - 1
3 - 3
4 - 5
5 - 0
6 - 4
7 - 3

Relaxation

(u, v)

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

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

$|V| = n = 7$

* Relaxation is done $|V|-1$ times.

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

Time complexity : $O(|E|(|V|-1))$

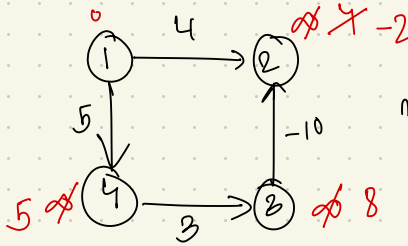
$\approx O(|V||E|)$

$\approx O(n^2)$ min

For a complete graph, $|E| = \frac{n(n-1)}{2}$

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

eg.



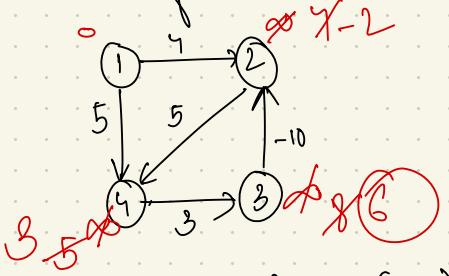
$n = 4 - 1 = 3$

1 - 0
2 - (-2)
3 - 8
4 - 5

edges → (3,2) (4,3) (1,4) (1,2)

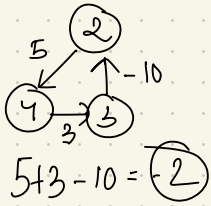
i.e. Distance of all vertices should remain same even in 4th relaxation.

Drawback of Bellman Ford:



$|V| = 4$

3 relaxations



edges → (3,2) (4,3) (1,4) (1,2) (2,4)

i.e. One more vertex is relaxed in 4th relaxation & so on.

∴ BF fails in presence of ⊖ve weight cycle.

→ BF can be used to detect ⊖ve weight cycle.

0/1 Knapsack Problem → Tabulation method / Sets method

$m = 8$, $n = 4$

$p = \{1, 2, 5, 6\}$

$w = \{2, 3, 4, 5\}$

$\max \sum p_i x_i$

$\sum w_i x_i \leq m$

DP

* optimization problem.
* Sol. ⁿ decisions. (Include/Not)
* Try all possible sol's & pick up the best.

Tabulation Method:

p_i	w_i	i ↓ / w →	0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
6	5	4	0	0	1	2	5	6	6	7	8

$$V[i,w] = \max\{V[i-1,w], V[i-1, w-w[i]] + p[i]\}$$

$$V[4,1] = \max\{V[3,1], V[3,1-5] + 6\}$$

$V[3,-4]$ → 0

We will explore all possibilities without having to spend this much of time.

$$V[4,5] = \max\{V[3,5], V[3,5-5]+6\} = \max\{5, 0+6\} = 6$$

$$V[4,6] = \max\{V[3,6], V[3,6-5]+6\} = \max\{6, 0+6\} = 6$$

$$V[4,7] = \max\{V[3,7], V[3,7-5]+6\} = \max\{7, 1+6\} = 7$$

$$V[4,8] = \max\{V[3,8], V[3,8-5]+6\} = \max\{7, 2+6\} = 8$$

$$\{x_1\ x_2\ x_3\ x_4\} = \{0\ 1\ 0\ 1\} \qquad 8-6=2,\ 2-2=0$$

T.C: O(n×W)

∵ T.C. depends on size of the table.

Sets method:

n=4, m=9

$P = \{1, 2, 5, 6\}$

$W = \{2, 3, 4, 5\}$

(P, W)

$S^0 = \{(0,0)\}$ — add (1,2) to each pair

$S_1^0 = \{(1,2)\}$

After merging: $S^1 = \{(0,0), (1,2)\}$ — add (2,3)

$S_1^1 = \{(2,3), (3,5)\}$

$S^2 = \{(0,0), (1,2), (2,3), (3,5)\}$ — add (5,4)

$S_1^2 = \{(5,4), (6,6), (7,7), (8,9)\}$ → exceeded bag capacity

$S^3 = \{(0,0), (1,2), (2,3), (3,5), (5,4), (6,6), (7,7)\}$

(3,5): profit inc. but weight increased. ∴ Discard one with less profit. [Dominance rule]

$S_1^3 = \{(6,5), (7,7), (9,8), \cancel{(11,9)}, \cancel{(12,11)}, \cancel{(13,12)}\}$

$S^4 = \{(0,0), (1,2), (2,3), (5,4), \cancel{(6,5)}, (6,5), (7,7), (9,8)\}$

① $(9,8) \in S^4$
but $(9,8) \notin S^3$ $\therefore x_4 = 1$
$(9-6, 8-5) = (2,3)$

② $(2,3) \in S^3$
and $(2,3) \in S^2$ $\therefore x_3 = 0$

③ $(2,3) \in S^2$
but $(2,3) \notin S^1$ $\therefore x_2 = 1$
$(2-2, 3-3) = (0,0)$

④ $(0,0) \in S^1$
and $(0,0) \in S^0$ $\therefore x_1 = 0$

```
main()
{ int p[5] = {0,1,2,5,6};
  int wt[5] = {0,2,3,4,5};
  int m=8, n=4;
  int k[5][9];
  for (int i=0; i<=n; i++)
  { for (int w=0; w<=m; w++)
    { ① if (i==0 || w==0)
          k[i][w] = 0;
      ② else if (wt[i] <= w)
          k[i][w] = max(p[i] + k[i-1][w-wt[i]],
                        k[i-1][w]);
      ③ else
          k[i][w] = k[i-1][w];
    }
  }
  cout << k[n][w];
}
```

	0	1	2	3	4
p	0	1	2	5	6

	0	1	2	3	4
wt	0	2	3	4	5

$n = 4$

$m = 8$

$k[i][w] = \max\left(p[i] + k[i-1][w-wt[i]],\ k[i-1][w]\right)$

1 0 0

k

i \ w	0	1	2	3	4	5	6	7	8
p wt 0	0	0	0	0	0	0	0	0	0
1 2 1	0	0	1	1	1	1	1	1	1
2 3 i→2	0	0	1	2	2	3	3	3	3
5 4 3	0	0	1	2	5	5	6	7	7
6 5 4	0	0	1	2	5	6	6	7	(8)

w ↓ 2

$8 - 6 = 2$

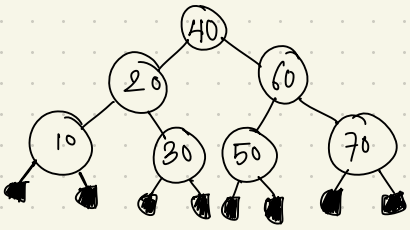
$2 - 2 = 0$

x_1	x_2	x_3	x_4
0	1	0	1

```
i = n; j = m;
while(i>0 && j>0)
{
  if (k[i][j] == k[i-1][j])
  { cout << i << "=0" << endl;
    i--; }
  else { cout << i << "=1" << endl;
    i--;
    j = j - wt[i]; }
}
```

Optimal Binary Search Tree

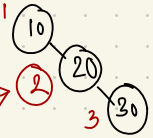
Keys → 10, 20, 30, 40, 50, 60, 70



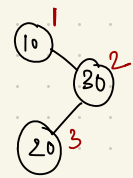
levels
1
2
3

eg. Keys → 10, 20, 30 $T(n) = \frac{2^n C_n}{n+1}$ $T(3) = \frac{2^3 C_3}{3+1} = \frac{{}^6C_3}{4} = 5$

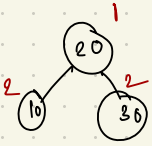
Number of comp. while Searching →



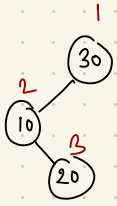
Avg = $\frac{1+2+3}{3} = \frac{6}{3} = 2$



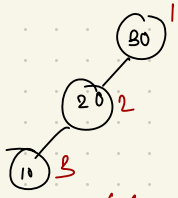
$\frac{1+2+3}{3} = \frac{6}{3} = 2$



$\frac{1+2+2}{3} = \frac{5}{3}$



6/3



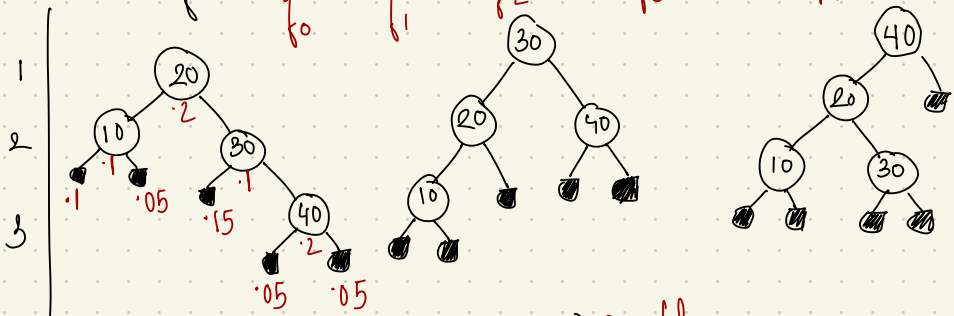
6/3

Probability of successful & unsuccessful search

Keys : 10 20 30 40

p_i : .1 (p_1) .2 (p_2) .1 (p_3) .2 (p_4)

q_i : .1 (q_0) .05 (q_1) .15 (q_2) .05 (q_3) .05 (q_4)



Cost of searching = $(1 \times .2 + 2 \times .1 + 2 \times .1 + 3 \times .2)$ Successful

$+ (2 \times .1 + 2 \times .05 + 2 \times .15 + 3 \times .05 + 3 \times .05)$ Unsuccessful (take level − 1)

$= 2.1$

$$Cost[0,n] = \sum_{0 \le i \le n} p_i * level(a_i) + \sum_{0 \le i \le n} q_i * (level(E_i) - 1)$$

* Cost of searching should be minimum out of all possible trees.
* We will try to find the best tree without generating every tree.

$$c[i,j] = \min_{i<k\le j} \{ c[i,k-1] + c[k,j] \} + w[i,j]$$

$$c[0,3] = \min_{0<k\le 3} \{ c[0,0] + c[1,3],\ c[0,1] + c[2,3],\ c[0,2] + c[3,3] \} + w[0,3]$$

$$c[1,3] = \min_{1<k\le 3} \{ c[1,1] + c[2,3],\ c[1,2] + c[3,3] \} + w[1,3]$$

	i			j
j-i=0	c[0,0]	c[1,1]	c[2,2]	c[3,3]
j-i=1	c[0,1]	c[1,2]	c[2,3]	
j-i=2	c[0,2]	c[1,3]		
j-i=3	c[0,3]			

$$w[0,2] = q_0 + p_1 + q_1 + p_2 + q_2$$

$$w[0,3] = q_0 + p_1 + q_1 + p_2 + q_2 + p_3 + q_3$$

$$w[0,3] = w[0,2] + p_3 + q_3$$

$$w[i,j] = w[i,j-1] + p_j + q_j$$

eg. $Keys = \{ 10, 20, 30, 40 \}$

$p_i = \{ 3, 3, 1, 1 \}$

$q_i = \{ 2, 3, 1, 1, 1 \}$

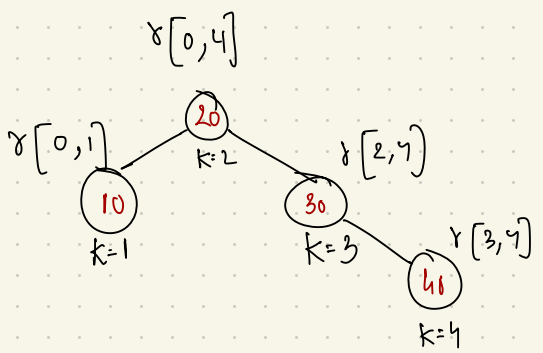
$w[0,1] = w[0,0] + p_1 + q_1$

$= 2 + 3 + 3$

$= 8$

j →	0	1	2	3	4
j-i=0	$w_{00}=2$ $c_{00}=0$ $r_{00}=0$	$w_{11}=3$ $c_{11}=0$ $r_{11}=0$	$w_{22}=1$ $c_{22}=0$ $r_{22}=0$	$w_{33}=1$ $c_{33}=0$ $r_{33}=0$	$w_{44}=1$ $c_{44}=0$ $r_{44}=0$
j-i=1	$w_{01}=8$ $c_{01}=8$ $r_{01}=1$	$w_{12}=7$ $c_{12}=7$ $r_{12}=2$	$w_{23}=3$ $c_{23}=3$ $r_{23}=3$	$w_{34}=3$ $c_{34}=3$ $r_{34}=4$	
j-i=2	$w_{02}=12$ $c_{02}=19$ $r_{02}=1$	$w_{13}=9$ $c_{13}=12$ $r_{13}=2$	$w_{24}=5$ $c_{24}=8$ $r_{24}=3$		
j-i=3	$w_{03}=14$ $c_{03}=25$ $r_{03}=2$	$w_{14}=11$ $c_{14}=19$ $r_{14}=2$			
j-i=4	$w_{04}=16$ $c_{04}=32$ $r_{04}=2$				

22/16 = 2



∴ This is the optimal BST based on successful & unsuccessful search probabilities which takes minimum cost.

Travelling Salesman Problem

Problem Defn.:

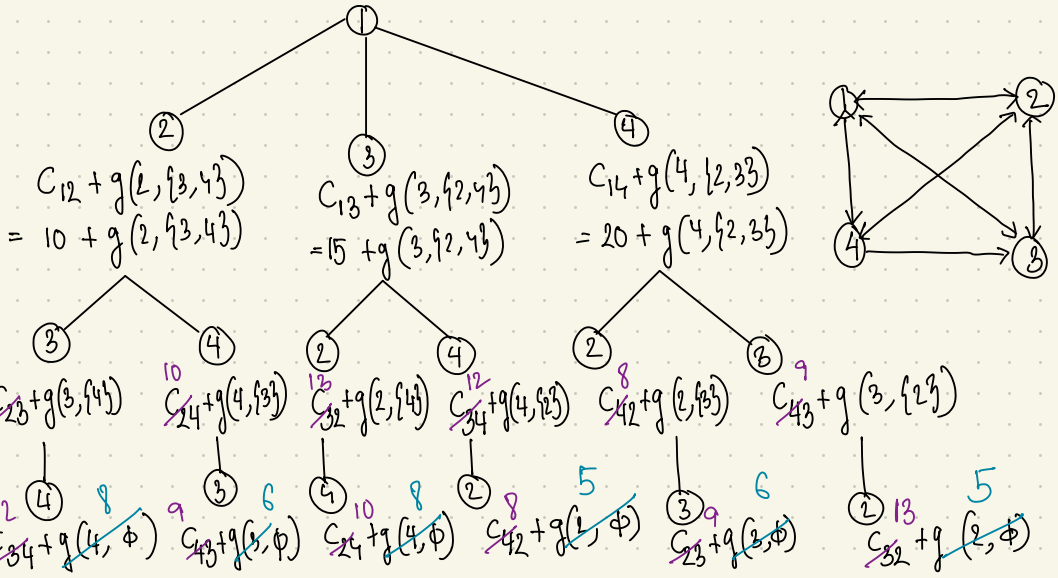
We have to start from some vertex and travel all the other vertices once & return back to same starting vertex. The cost of travelling must be minimum.

Recurrence Formula

$$g(i,S) = \min_{k \in S} \{ c_{ik} + g(k, S-\{k\}) \}$$

$$g(1,\{2,3,4\}) = \min_{k \in \{2,3,4\}} \{ c_{1k} + g(k, \{2,3,4\} - \{k\}) \}$$

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0



→ If we use recursive tree, the above expansion occurs. To avoid this, we will make a table where we get the values of the last level first. (bottom-up).

→ Now let us calculate $g(1, \{2,3,4\})$

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

$|S| = 0$:

$g(2, \phi) = 5$
$g(3, \phi) = 6$
$g(4, \phi) = 8$

$|S| = 1$:

$g(2, \{3\}) = 15$ $\quad \min\{c_{23} + g(3, \phi)\} = 9 + 6 = 15$ $\quad J(2, \{3\}) = 3$
$g(2, \{4\}) = 18$ $\quad \min\{c_{24} + g(4, \phi)\} = 10 + 8 = 18$ $\quad J(2, \{4\}) = 4$
$g(3, \{2\}) = 18$ $\quad \min\{c_{32} + g(2, \phi)\} = 13 + 5 = 18$ $\quad J(3, \{2\}) = 2$
$g(3, \{4\}) = 20$
$g(4, \{2\}) = 13$
$g(4, \{3\}) = 15$

$|S| = 2$:

$g(2, \{3,4\}) = 25$ $\quad \min\{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\} = \min\{29, 25\} = 25$
$g(3, \{2,4\}) = 25$ $\quad J\{2, \{3,4\}\} = 4$
$g(4, \{2,3\}) = 23$ $\quad J(3, \{2,4\}) = 4$
$J(4, \{2,3\}) = 2$

$|S| = 3$:

$g(1, \{2,3,4\}) = 35$ $\quad J(1, \{2,3,4\}) = 2$

Path construction:

If $J(i, s) = x$, then from x we go to $J(x, s - x)$

$J(1, \{2,3,4\}) = 2$

$J(2, \{3, 4\}) = 4$

$J(4, \{3\}) = 3$

$\therefore$ path: 1-2-4-3-1

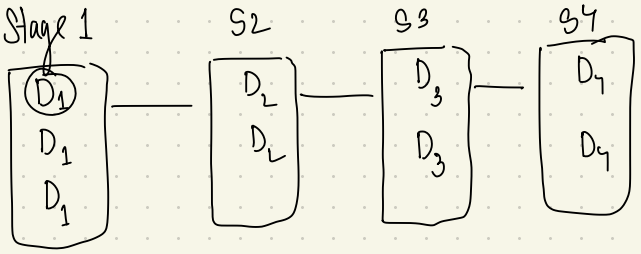
Reliability Design

Setup a system

D_1	D_2	D_3	D_4
C_1	C_2	C_3	C_4
r_1	r_2	r_3	r_4
.9	.9	.9	.9

Reliability of the system

$= \pi r_i = (0.9)^4 = 0.6561$



→ At each stage, we will have multiple copies of the same device. If first copy isn't working, the next will continue working in its place. [Devices are in parallel]

Calculating reliability of a stage

$r_1 = 0.9$ — success

$1 - r_1 = 1 - 0.9 = 0.1$ — failure

$(1 - r_1)^3 = (0.1)^3 = 0.001$

$1 - (1 - r_1)^3 = 0.999$

So, reliability of the stage increases if we have multiple copies of the same device.

Problem Statement :- If "C" is cost we have in hand & we have to set up a system. How many copies of each device should be bought for maximum reliability of the system.

Upper Bound

eg. $\sum c_i = c_1 + c_2 + c_3$

$= 30 + 15 + 20$

$= 65$

$\frac{40}{30} = 1$

D_i	C_i	r_i	v_i
D_1	30	0.9	2
D_2	15	0.8	3
D_3	20	0.5	3

$C = 105$

remaining, $C - \sum c_i = 105 - 65 = 40$

No. of copies for each device, $v_i = \left\lfloor \frac{C - \sum c_i}{c_i} \right\rfloor + 1$

$v_1 = \frac{105 - 65}{30} + 1 = \frac{40}{30} + 1 = 2$

$v_2 = \frac{40}{15} + 1 = 2 + 1 = 3$

$v_3 = \frac{40}{20} + 1 = 3$

$(R, C) \Rightarrow$ (Reliability, Cost)

$S^0 = \{(1, 0)\}$ // Initially amount spent (cost) is 0 & therefore reliability is 1.

Consider D_1,

$S_1^1 = \{(0.9, 30)\}$ // one copy

$S_2^1 = \{(0.99, 60)\}$ // two copies

$S^1 = \{(0.9, 30), (0.99, 60)\}$

$1 - (1-R_1)^2 = 1 - (1-0.9)^2$
$= 1 - (0.1)^2$
$= 1 - 0.01$
$= 0.99$

Consider D_2,

$S_1^2 = \{(0.72, 45), (0.792, 75)\}$

// Reliability is multiplied & Cost is added to S^1.

$S_2^2 = \{(0.864, 60), (0.9504, 90)\}$

Infeasible bcoz we cannot buy 3rd device from remaining amount.

$1 - (1-R_2)^2$
$= 1 - (1-0.8)^2$
$= 1 - (0.2)^2$
$= 1 - 0.04$
$= 0.96$ 30

$S_3^2 = \{(0.8928, 75), (\_, 105)\}$

$1 - (1-R_2)^3$
$= 1 - (1-0.8)^3$
$= 1 - (0.2)^3$
$= 1 - 0.008$
$= 0.992$ 45

$S_2 = \{(0.72, 45), (0.792, 75), (0.864, 60), (0.8928, 75)\}$

Reliability inc. But cost ↓.
So, delete one ordered pair by Dominance Rule. One with higher cost is deleted.

Consider D_3,

$S_1^3 = \{(0.36, 65), (0.432, 80), (0.4464, 95)\}$

$S_2^3 = \{(0.54, 85), (0.648, 100), (\_, 115)\}$

$S_3^3 = \{(0.63, 105), (\_, 120), (\_, 135)\}$

$1 - (1-R_3)^2$
$= 1 - (1-0.5)^2$
$= 1 - (0.5)^2$
$= 1 - 0.25$
$= 0.75$ 40

0.875 60

$S^3 = \{(0.36, 65), (0.432, 80), (0.~~4464~~, 95), (0.54, 85), (0.648, 100), (0.~~65~~, 105)\}$

∴ Maximum reliability, $\Pi r_i = 0.648$ & Cost, $\Sigma c_i = 100$

$D_3 \rightarrow$ 2 copies
$D_2 \rightarrow$ 2 copies
$D_1 \rightarrow$ 1 copy

Longest Common Subsequence (LCS)

1. What is LCS
2. LCS using Recursion
3. LCS using Memoization
4. LCS using DP

eg. String 1 : a b c d e f g h i j
String 2 : c d g i

cdgi ✓
dgi
gi

∴ LCS = cdgi

→ LCS is the longest set of characters that comes in the same sequence in both strings.
→ Intersection of matching not allowed!

eg. String 1 : a b c d e f g h i j
String 2 : e c d g i

Subsequences are ⇒ e g i, cdgi ✓

eg. String 1 : a b d a c e
String 2 : b a b c e

Subsequences are ⇒ bace, a b c e

LCS using Recursion

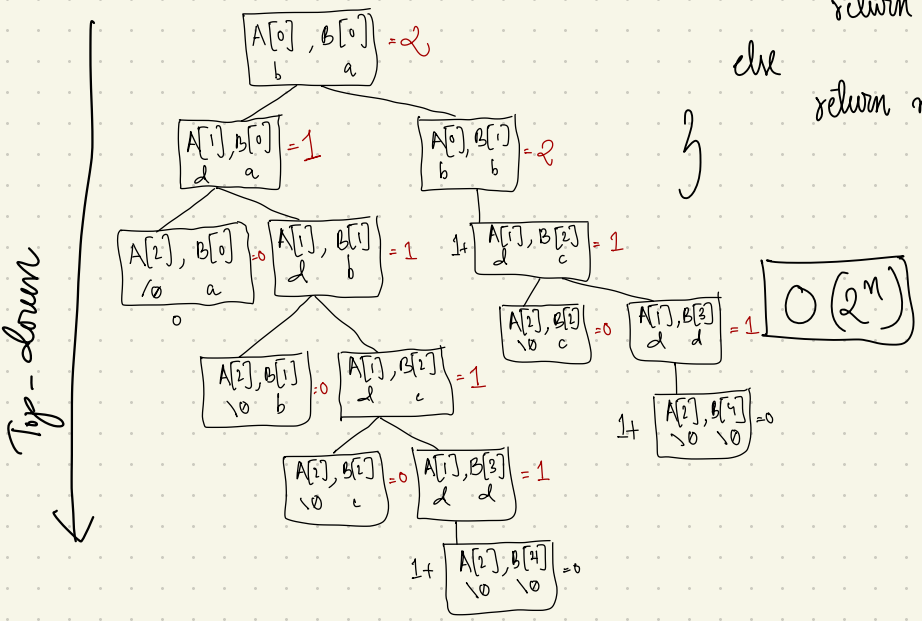
A	b	d	\0
	0	1	2

m

B	a	b	c	d	\0
	0	1	2	3	4

n

```
int LCS(i, j)
{
  if (A[i] == '\0' || B[j] == '\0')
      return 0;
  else if (A[i] == B[j])
      return 1 + LCS(i+1, j+1)
  else
      return max(LCS(i+1, j),
                 LCS(i, j+1));
}
```



LCS By Memoization :

No need to make unnecessary function calls if we already found the results

	a	b	c	d	\0
	0	1	2	3	4
b 0	2	2			
d 1	1	1	1	1	
\0 2	0	0	0		0

Bottom-up

$O(m \times n)$

Thus, memoization helps in reducing time complexity.

*LCS By DP

A | b | d | m

1 2

B | a | b | c | d | n

1 2 3 4

		a	b	c	d
	0	1	2	3	4
0	0	0	0	0	0
b 1	0	0	1	1	1
d 2	0	0	1	1	2

b d

```
for (i ← 1 to m) {
  for (j ← 1 to n) {
    if (A[i] == B[j])
        LCS[i,j] = 1 + LCS[i-1, j-i]
    else
        LCS[i,j] = max (LCS[i-1,j], LCS[i,j-1])
  }
}
```

∴ Length of LCS = 2

LCS = bd

T.C : $\theta(m \times n)$

Eg. Str1 : s t o n e

Str2 : l o n g e s t

		l	o	n	g	e	s	t
	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
s 1	0	0	0	0	0	0	1	1
t 2	0	0	0	0	0	0	1	2
o 3	0	0	1	1	1	1	1	2
n 4	0	0	1	2	2	2	2	2
e 5	0	0	1	2	2	3	3	3

o n e

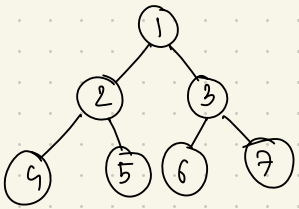
∴ Length of LCS = 3

LCS = one

BFS and DFS

1. Visiting a vertex.
2. Exploration of each vertex.

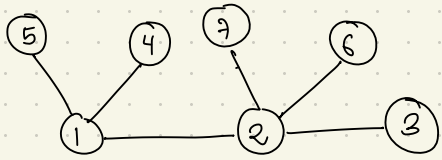
eg.



(level-wise) BFS: 1, 2, 3, 4, 5, 6, 7

(preorder) DFS: 1, 2, 4, 5, 3, 6, 7

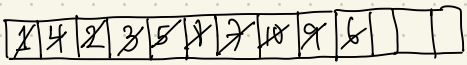
eg.

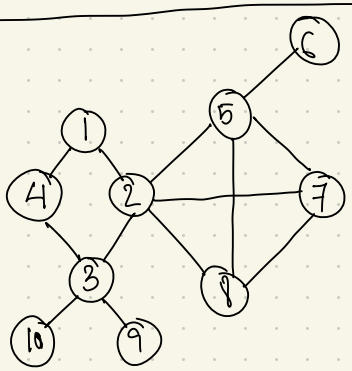


BFS: 1, 2, 4, 5, 7, 3, 6

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

eg. (a) BFS: 1, 4, 2, 3, 5, 8, 7, 10, 9, 6





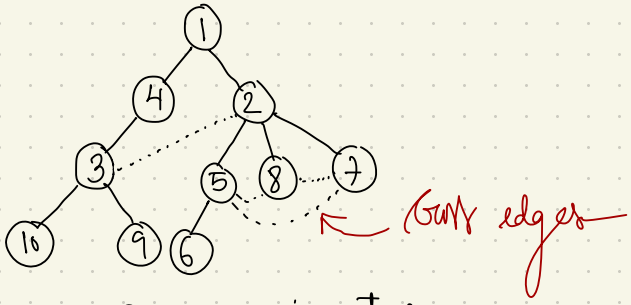


fig: BFS spanning tree

Note: 1. We can start BFS from any vertex.

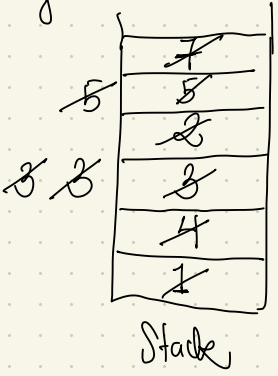
2. When exploring any vertex, its neighbours can be visited in any order.

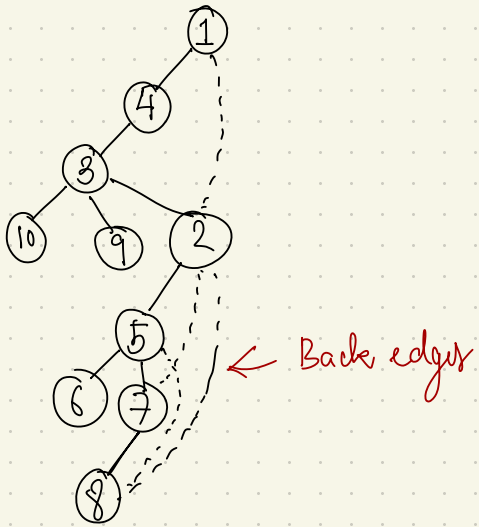
Other Valid BFS :-

(b) 1, 2, 4, 8, 5, 7, 3, 6, 10, 9

(c) 5, 2, 8, 7, 6, 3, 1, 9, 10, 4

eg. (a) DFS :- 1, 4, 3, 10, 9, 2, 5, 6, 7, 8





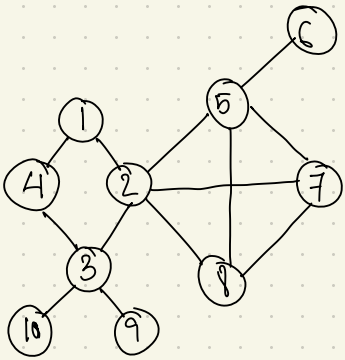


fig: DFS Spanning Tree

Other valid DFS :-

(b) 1, 2, 8, 7, 5, 6, 3, 10, 9, 4

(c) 3, 4, 1, 2, 5, 6, 7, 8, 10, 9

Time complexities

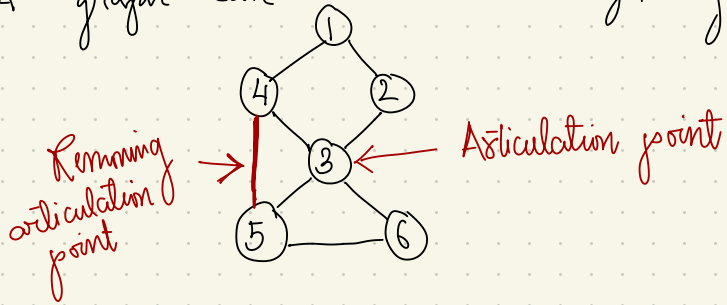
BFS → O(n)

DFS → O(n)

ARTICULATION POINT

→ Any vertex whose removal will disconnect the graph into multiple components is called articulation point.

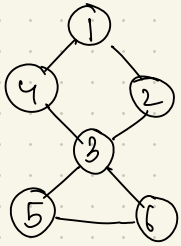
→ A graph can be made stronger by removing articulation pt.



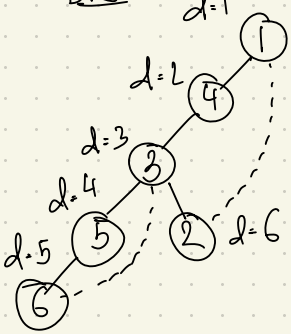
Finding articulation point :

Step 1: Find DFS Spanning Tree

Step 2: Mention DFS No. in the order they are visited.



DFS:



	1	2	3	4	5	6
d =	1	6	3	2	4	5
L =	1	1	1	1	3	3

Step 3: Find out lowest discovery no. reachable from any vertex by taking one backarc.

Step 4: parent u, child v

If $L[v] \geq d[u]$, then u is an articulation pt.

→ This condition works for all vertices except root.

L[5] ≥ d[3]

3 ≥ 3

∴ 3 is an articulation point.

Note: If root is having multiple children, then root is an articulation point.

Branch and Bound Strategy

→ It is similar to Backtracking in that it also uses state space tree to solve a problem.

→ It is useful for solving only minimization problems.

Eg. Job Sequencing

Jobs = {J1, J2, J3, J4}

p = {10, 5, 8, 3}

d = {1, 2, 1, 2}

Two ways of representing the soln.

S1 = {J1, J4} Variable Size solution

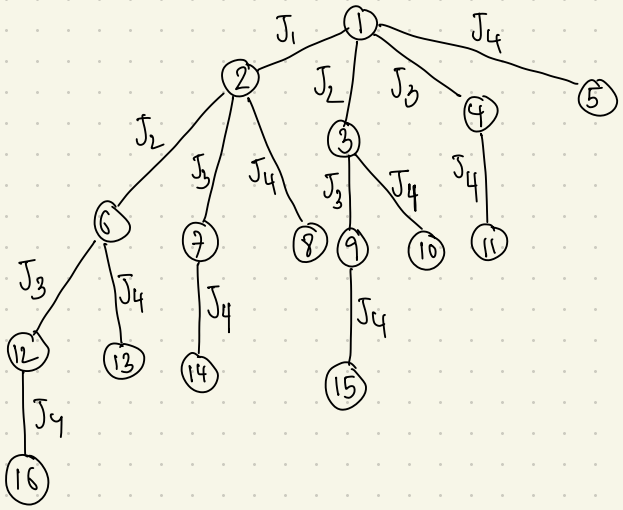
S2 = {1, 0, 0, 1} Fixed Size solution

→ Branch & Bound — BFS

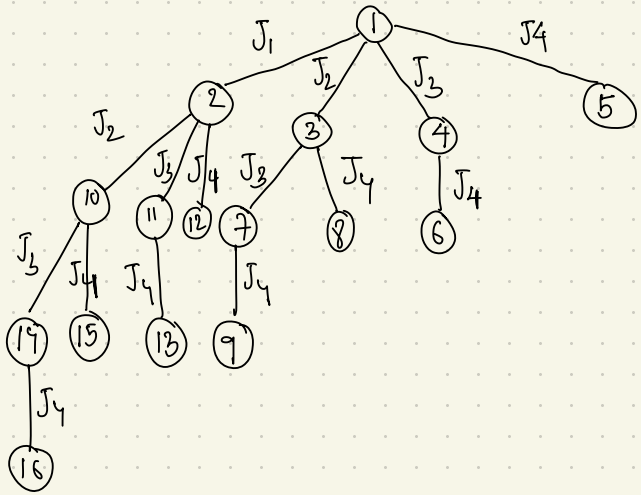
Backtracking — DFS

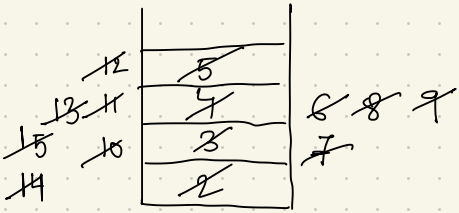
Variable size Soln.

Method ① : Using queue

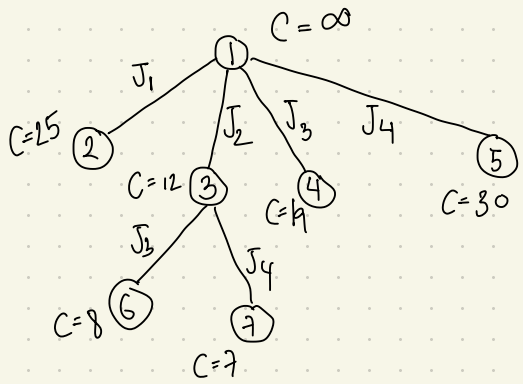


Method ② : Using stack



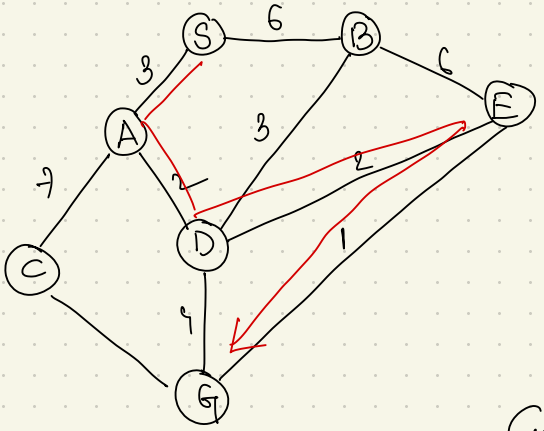


Method ③ : Least Cost - BB



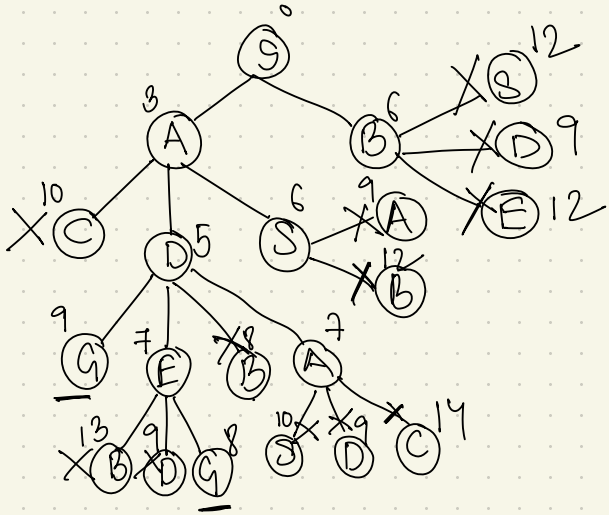
Node with minimum cost must be explored next.

Eg.



S = start state
G = goal state

Current Best soln. = 9 / 8



KMP String Matching Algorithm

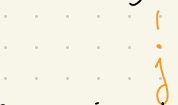
Naive Algorithm

Eg. String: a b c d e f g (1 2 3 4 5 6 7)

pattern: d e f

Eg. String: a b c d a b c a b c d f (1 2 3 4 5 6 7 8 9 10 11 12)

pattern: a b c d f (1 2 3 4 5)



Once there is a mismatch, i gets shifted back to a position 1 ahead of where we started.

Worst case of Naive Algorithm

Eg. String: a a a a a a a b (1 2 3 4 5 6 7 8) — n

pattern: a a a b (1 2 3 4) — m

T.C: O(mn)

KMP

pattern: a b c d a b c

prefix: a, ab, abc, abcd,

suffix: c, bc, abc, dabc,

→ Observation is done on the pattern to reduce the no. of comparisim. We check if the beginning part of a pattern reappears anywhere else or not.

π or lps table:

p_1 : a b c d a b e a b f
1 2
0 0 0 0 1 2 0 1 2 0

p_2 : a b c d e a b f a b c
0 0 0 0 0 1 2 0 1 2 3

p_3 : a a b c a d a a b c
0 1 0 0 1 0 1 2 0 0

p_4 : a a a a b a a c d
0 1 2 3 0 1 2 0 0

Eg. Step ①:

String : a b a b c a b c a b a b a b d
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

pattern : 0 1 2 3 4 5

a	b	a	b	d
0	0	1	2	0

Compare i with j+1.

If matched, forward both i & j by 1 step.

If there is a mismatch, bring j to corresponding index of π-table. i.e. 2.

Step ②:

String : a b a b c a b c a b a b a b d
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

pattern : 0 1 2 3 4 5

a	b	a	b	d
0	0	1	2	0

Step ③:

String: a b a b c a b c a b a b a b d
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

pattern: 0 a b a b d

1	2	3	4	5
a	b	a	b	d
0	0	1	2	0

We cannot move j further. So, move i.

Step ④:

String: a b a b c a b c a b a b a b d
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

pattern: 0 a b a b d

1	2	3	4	5
a	b	a	b	d
0	0	1	2	0

Step ⑤:

String: a b a b c a b c a b a b a b d
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

pattern: 0 a b a b d

1	2	3	4	5
a	b	a	b	d
0	0	1	2	0

Step ⑥:

String: a b a b c a b c a b a b a b d
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

pattern: 0 a b a b d

1	2	3	4	5
a	b	a	b	d
0	0	1	2	0

Step ③:

String: a b a b c a b c a b a b a b d
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

pattern: 0

1	2	3	4	5
a	b	a	b	d
0	0	1	2	0

T.C: $O(m+n)$

→ KMP string matching algorithm uses degenerating property (pattern having the same sub-pattern appearing more than once in the pattern) of the pattern & improves the worst case complexity to $O(n+m)$.

→ Basic Idea: Whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will match anyway.

eg. String: A A A A A B A A A B A
1 2 3 4 5 6 7 8 9 10 11

pattern: 0

1	2	3	4
A	A	A	A
0	1	2	3

lps[i] = longest proper prefix of pat[0...i] which is also a suffix of pat[0....i]

NP-Hard & NP-Complete

Polynomial Time

Linear Search – n

Binary Search – $\log n$

Insertion Sort – n^2

Merge Sort – $n \log n$

Matrix multiplication – n^3

Exponential Time

0/1 Knapsack – 2^n

Travelling SP – 2^n

Sum of subsets – 2^n

Graph coloring – 2^n

Hamiltonian Cycle – 2^n