DAA

Algorithm

- It is a sequence of unambiguous (clear) instructions for solving a problem, such as, for obtaining a required output for any legit (acceptable) input in a finite amount of time.

- The problem can be solved by multiple algorithms. However, the goal is to choose an algorithm that takes the least time and space (memory) to compute the problem.

Euclid's Algorithm (to find GCD)

```
// Computes gcd(m, n)
while n ≠ 0 do
    r ← m (mod) n
    m ← n
    n ← r
return m
```

Ex: $gcd(60, 24)$ → remainder of $m \% n$
$= gcd(24, 12)$
$= gcd(12, 0)$
$= 12$

Consecutive Integer Checking Algorithm (GCD)

1. Assign least/minimum value among (m) and (n) to t.
2. Divide m by t. If remainder = 0, → (3)
   else → (4)
3. Divide n by t, If remainder = 0, answer = t
   else → (4)
4. Decrease (t-1), then → (2)

— **Middle-School Procedure (GCD)**

- Find prime factors of m and n
  Find common factors and compute product to obtain GCD.

60 → 2 × 2 × 3 × 5 → 12
24 → 2 × 2 × 3 × 2

— **Sieve of Eratosthenes Algo** (to find prime numbers ≤ n)

- Assign all values of from 2 to n to an array A.

→
```
for p ← 2 to n do
    A[p] ← p
for p ← 2 to ⌊√n⌋ do
    if A[p] ≠ 0          (if not rejected)
        j ← p * p
        while j ≤ n do
            A[j] ← 0 (or) X      (eliminate multiples)
            j ← j + p
i ← 0
for p ← 2 to n do
    if A[p] ≠ 0          (not rejected)
        L[i] ← A[p]
        i ← i+1;
return L    (contains list of primes ≤ n)
```

Ex: For $n = 25$, (we are eliminating multiples from 2 to $\sqrt{25}$)

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
2 3 X 5 X 7 X 9 X 11 X 13 X 15 X 17 X 19 X 21 X 23 X 25
2 3 5 7 X 11 13 X 17 19 X 23 25
2 3 5 7 11 13 17 19 23 X

Note Some fundamental data structures are linear Data Structures, Graphs, Trees, Sets and Dictionaries.

✱ Algorithm Efficiency

– Measuring Space Complexity

- It is the amount of storage space or memory required to execute,

$$S(P) = C + S_P$$

where $S(P)$ → space complexity of program 'P'.
C → constants for inputs and outputs.
S_P → instance characteristics of a program

Ex: For algorithm Add (x, n)

```
sum ← 0
for i ← 1 to n do
    sum ← sum + x[i]
return sum
```

(Here, space for variables (sum = 1, i = 1, n = 1) ⟹ 3
space for array, x = n)

$S(P) = C + S_P = n + 3$

– Measuring Time Complexity

- It is the amount of computer time required to completion. Constraints include system load, speed of underlying hardware and instruction set used.

Ex:

```
for (i=0; i<n; i++){
    sum = sum + a[i];
}
```

Statement	Frequency
i=0	1
i<n	n+1
i++	n
sum=sum+a[i]	n
	3n+2

$O(n) = 3n + 2$

Measuring Input Size

- Efficiency of an algorithm can be computed as a function with input size as a parameter (exact/approx)

Measuring Running Time

- From an algorithm, identify a basic operation, understand the concept and compute total time taken for it.

$$T(n) = C_{op}\, C(n)$$

where $T(n) \rightarrow$ running time of basic operation.
$C_{op} \rightarrow$ time taken by basic operation to execute
$C(n) \rightarrow$ no. of times operations needs to be exec.

Computing Order of Growth

n	$\log n$	$n \log n$	n^2	2^n
1	0	0	1	2
2	1	2	4	4
4	2	8	16	16

Computing Best, Worst & Average Case Efficiencies

① Best Case : Search(10) [10 | 20 | 30] ↑ at 10 — $O(1)$

② Worst Case : Search(30) [10 | 20 | 30] ↑ at 30 — $O(n)$

③ Average Case Search(20) ✓ (or) Search(40) ✗ [10 | 20 | 30] 40 ↑ at 20, ↑ at 40

Let $P \rightarrow$ probability of successful search
$(1-P) \rightarrow$ probability of unsuccessful search
$n \rightarrow$ total input size of list.
$\frac{P}{n} \rightarrow$ probability of first match for every i^{th} element.

$C_{avg}(n) \rightarrow$ probability of successful + unsuccessful search

$$= \left[1.\frac{P}{n} + 2.\frac{P}{n} + 3.\frac{P}{n}, \ldots i.\frac{P}{n}\right] + n(1-P)$$

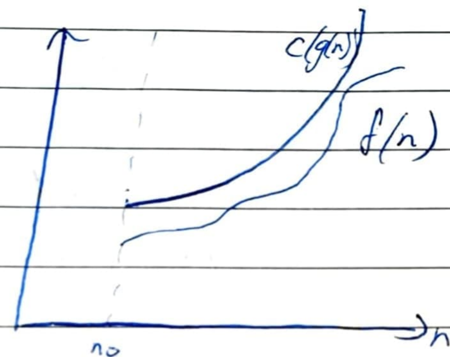
$$= \frac{P}{n}(1+2+\ldots+n) + n(1-P)$$

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

For search (20), $P = 1$ (successful), $C_{avg}(n) = \frac{n+1}{2}$
For search (40), $P = 0$, $C_{avg}(n) = n$

✶ Asymptotic Notations

- With increase in input size, performance of algorithm changes. Efficiency analysis framework concentrates on order of growth of an algorithms basic opcount as principal indicator of efficiency.

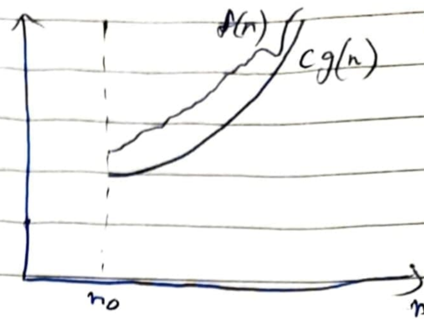


— O-notation

- Function $f(n)$ is said to be in $O(g(n))$; $f(n) \in O(g(n))$ if

$$f(n) \leq c\,g(n) \quad \forall\ n \geq n_0 \quad (n_0 > 0)(c > 0)$$

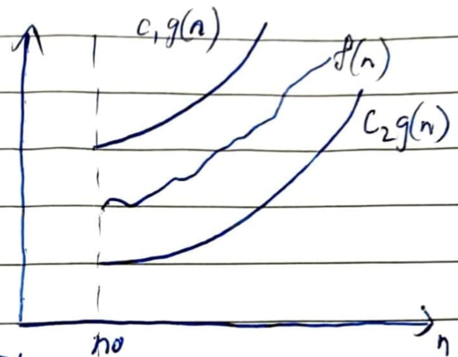
- **Ω-notation**



- Function $f(n)$ is said to be in $\Omega(g(n))$; $f(n) \in \Omega(g(n))$ if

$$f(n) \geq cg(n) \quad \forall n \geq n_0$$

- **θ-notation**



- Function $f(n)$ is said to be in $\Theta(g(n))$ if

$$c_2 g(n) \leq f(n) \leq c_1 g(n) \quad \forall n \geq n_0$$

Ex:-
```
def foo(lst):
    result1 = 1
    result2 = 6
    for i in range(len(lst)):
        for j in range(len(lst)):
            result1 += i*j
            result2 = lst[j] + i
    return
```

Here, let n be length of lst. Outer loop iterates n times and for each iteration, inner loop iterates n times. Each iteration of inner loop takes 2 steps; first two lines do some constant work. ∴ Overall runtime $= 2 + (n \times n \times 2)$

$= 2n^2 + 2 \approx O(n^2)$

Properties

- If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$ then,
  $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$

Proof: $t_1(n) \leq c_1 g_1(n) \quad \forall\, n \geq n_1$
$t_2(n) \leq c_2 g_2(n) \quad \forall\, n \geq n_2$
$t_1(n) + t_2(n) \leq c_1 g_1(n) + c_2 g_2(n)$
Let $c_3 = \max[c_1, c_2]$ and $n \geq \max[n_1, n_2]$.

$$t_1(n) + t_2(n) \leq c_3 g_1(n) + c_3 g_2(n)$$
$$\leq c_3 [g_1(n) + g_2(n)]$$
$$t_1(n) + t_2(n) \leq 2c_3 [\max\{g_1(n), g_2(n)\}]$$

where $c \to 2c_3 = 2\max[c_1, c_2]$
$n_0 \to \max[n_1, n_2]$.

- $$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0, & \text{implies } t(n) ~~\text{has smaller}~~ < O(g(n)) \\ c, & \text{implies } t(n) = c\,g(n) \\ \infty, & \text{implies } t(n) > c\,g(n) \end{cases}$$

Here first two cases $\to t(n) \in O(g(n))$
last two cases $\to t(n) \in \Omega(g(n))$
second case $\to t(n) \in \Theta(g(n))$

- L'Hospital's Rule: $\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$

- Stirling's Formula:
  $$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$ for larger values of n.

Ex- $t(n) = \frac{1}{2} n(n-1)$, $g(n) = n^2$

$$\frac{1}{2} \lim_{n\to\infty} \frac{n(n-1)}{n^2} = \frac{1}{2} \lim_{n\to\infty} \left(1 - \frac{1}{n}\right) = \frac{1}{2}$$

$\therefore$ Since limit = constant, $\frac{1}{2}n(n-1) \in \Theta(n^2)$

Ex: $t(n) = \log_2 n$, $g(n) = \sqrt{n}$

$$\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n\to\infty} \frac{\frac{1}{n}(\log_2 e)}{1/2\sqrt{n}}$$

$$= 2\log_2 e \lim_{n\to\infty} \left(\frac{1}{\sqrt{n}}\right) = 0$$

$\therefore$ Since limit = 0, $\log_2 n \in o(\sqrt{n})$ (little-oh)

Ex: $t(n) = n!$, $g(n) = 2^n$

$$\lim_{n\to\infty} \frac{n!}{2^n} = \lim_{n\to\infty} \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n} = \lim_{n\to\infty} \sqrt{2\pi n}\left(\frac{n}{2e}\right)^n$$

$$= \infty$$

$\therefore$ Since limit = ∞, $n! \in \Omega(2^n)$

- Basic Efficiency Classes

Class	Name	
1	constant	(best case efficiencies)
$\log n$	logarithmic	(Binary Search, GCD)
n	linear	(scanning a list of size n)
$n\log n$	linearithmic	(mergesort, quicksort)
n^2	quadratic	(nested loops)
n^3	cubic	
2^n	exponential	
$n!$	factorial	(all permutations of n-element set)

Ex: For the algorithm MaxElement(A[0.....n-1])

maxval ← A[0]

for i ← 1 to n-1 do

_/_/_

if A[i] > maxval
    maxval ← A[i]
return maxval

$$C(n) = \sum_{i=1}^{n-1} 1 = (n-1) - i + 1 = n-1-1+1 = n-1 \in \Theta(n)$$

Note $\sum_{i=I}^{U} 1 = (U - I + 1)$, $\sum_{i=1}^{n} = n - i + 1 = \Theta(n)$

$$\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$$

$$\sum_{i=0}^{n} a^i = a^0 + a^1 + \dots = \frac{a^{n+1} - 1}{a-1} \quad (a \neq 0)$$

Ex: An algorithm for uniqueElements (A[0:....n-1])

```
for i ← 0 to n-2 do
    for j ← i+1 to n-1 do
        if A[i] = A[j] return false (duplicates found)
return true.
```

Best Case: First two elements of array are same, O(1)
Worst Case: Either no equal elements, or, only last two elements are equal.

$$\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n - i - 1)$$

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

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

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

Ex: Matrix Multiplication (A[0.....n-1, 0...n-1], B[0...n-1, 0...n-1])

```
for i ← 0 to n-1 do
    for j ← 0 to n-1 do
        C[i,j] ← 0.0
        for k ← 0 to n-1 do
            C[i,j] ← C[i,j] + A[i,k] * B[k,j]
return C
```

__/__/__

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

$$= n\sum_{i=0}^{n-1}(n-1-0+1) = n^2\sum_{i=0}^{n-1} 1$$

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

$$T(n) = C_m(n^3) + C_a(n^3)$$

$$= (C_m + C_a)n^3$$

($C_m \to$ multiply
$C_a \to$ addition)

Ex: Tower of Hanoi (disk, source, dest, aux)

```
    if disk = 1 then
        move disk from source to dest
    else
        Hanoi (disk - 1, source, aux, dest)
        move disk from source to dest
        Hanoi (disk-1, aux, dest, source)
STOP
```

Here, $M(n) = M(n-1) + 1 + M(n-1)$ ($n \to$ no. of disks)

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

$$M(1) = 1$$

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

$$M(n) = 2^2 M(n-2) + 2 + 1 \ldots$$

$$M(n) = 2^i M(n-i) + 2^i - 1$$

For $i = n-1$,

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

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

Ex: Number of binary digits in n's binary representation

```
BinRec(n):
if (n = 1) return 1
else return BinRec(⌊n/2⌋) + 1
```

—1—

$A(n) = A(n/2) + 1$, $A(1) = 0$
$A(n/2) = A(n/2^2) + 1$
$A(n) = A(n/2^2) + 2 \ldots$ $\left(\frac{n}{2^i} = 1,\ n = 2^i\right.$
$A(n) = A(n/2^i) + i$ $\left.\log_2 n = i\right)$
$A(n) = A(1) + \log_2 n = \log_2 n$

—x—

Brute Force (✓)

- Brute force is a straightforward approach to solving a problem, usually directly based on problem statement.

Ex:- $a^n = \underbrace{a \times a \times a \times \ldots \times a}_{n \text{ times}}$ (This method suggests simply computing a^n by multiplying a n times)

— Selection Sort

- Selecting smallest/largest element from unsorted array and swapping with first element of unsorted array.

```
| 89   45   68   90   29   34  (17)
17 | 45   68   90  (29)  34   89
17   29 | 68   90   45  (34)  89
17   29   34 | 90  (45)  68   89
17   29   34   45 | 90  (68)  89
17   29   34   45   68 | 90  (89)
17   29   34   45   68   89 | 90
```

→
```
for i ← 0 to (n-2) do
  min ← i
  for j ← (i+1) to (n-1) do
     if A[j] < A[min], min ← j
  swap A[i] and A[min]
```

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} ((n-1)-(i+1)+1)$$

$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} = \frac{n^2-n}{2} \simeq n^2$$

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

— Bubble Sort

- Compare adjacent elements of list and exchange if they are out of order.

89 $\overset{?}{\longleftrightarrow}$ 45   68   90
45   89 $\overset{?}{\longleftrightarrow}$ 68   90
45   68   89 $\overset{?}{\longleftrightarrow}$ 90

→
```
for i <- 0 to n-2 do
    for j <- 0 to (n-2-i) do
        if A[j+1] < A[j]
            swap A[j] and A[j+1]
```

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} ((n-2-i)-0+1)$$

$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{n(n-1)}{2} = \frac{n^2-n}{2} \simeq n^2$$

$\therefore O(n^2)$

— Linear / Sequential Search

- Searches given key element in list of n elements by checking successive elements in list until match is found / list is exhausted.

$n=5$

15	12	7	10	8

skibidi

n = 7, m = 3

__/__/____

→ i ← 0
while i<n and a[i] != k do
  i ← i+1
if i<n
  return i
else
  return -1

Best Case : O(1)
Avg Case : O(n/2)
Worst Case : O(n)

– **Brute Force String Matching**

• Given: Text (string of n characters)
  Pattern (string of m characters) (m≤n)
find substring of the text that matches the pattern.

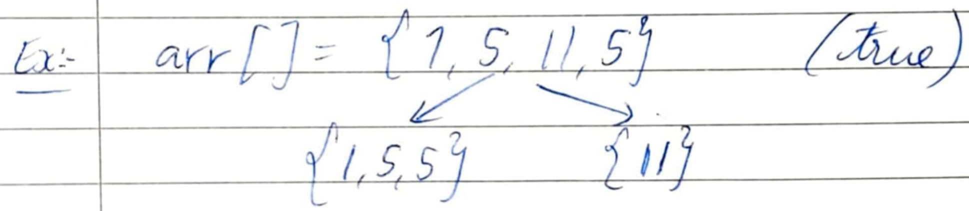
```
→ for i ← 0 to n-m do
    j ← 0
    while j<m and P[j] = T[i+j] do
      j ← j+1
    if j = m , return i
  return -1
```

O(nm) (Worst)
O(n) (Best)

– **Partition**

• To determine whether a given set can be partitioned into two subsets such that sum of elements in both subsets is same.

of $= \frac{\text{total sum}}{2}$



• For each recursion of the method, either (a) create a new subset of the array including last element if value does

not exceed S/2 and repeat for new subarray or (b) create a new subset of array excluding last element and repeat. Either (a) or (b) will return true.

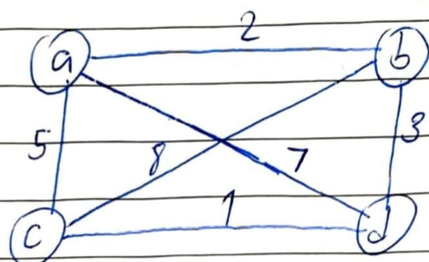
✱ Exhaustive Search

- Examining every possible combination of a set of choices.

– Traveling Salesman

- Find the shortest possible route that the salesman should consider to visit every city exactly once and return back. (Use Hamiltonian circuit) [(n-1)!]

Ex:-



Tours	Length
a→b→c→d→a	$l = 2+8+1+7 = 18$
a→b→d→c→a	$l = 2+3+1+5 = 11$ ✓ (optimal)
a→c→b→d→a	$l = 5+8+3+7 = 23$
a→c→d→b→a	$l = 5+1+3+2 = 11$ ✓
a→d→b→c→a	$l = 7+3+8+5 = 23$
a→d→c→b→a =	$l = 7+1+8+2 = 18$

– Knapsack Problem

- Find most valuable (subset) set of items without exceeding knapsack capacity.

- Number of subsets $= 2^n$ $\Omega(2^n)$

_/_/_

Ex:- Total capacity of knapsack = 10

item 1 ($w_1 = 7$) ($v_1 = \$42$)

item 2 ($w_2 = 3$) ($v_2 = \$12$)

item 3 ($w_3 = 4$) ($v_3 = \$40$)

item 4 ($w_4 = 5$) ($v_4 = \$25$)

Subset	Total weight	Total value
ϕ	0	\$0
{1}	7	\$42
{2}	3	\$12
{3}	4	\$40
{4}	5	\$25
{1,2}	10	\$54
{1,3}	11 >10	— (not feasible)
{1,4}	12 >10	—
{2,3}	7	\$52
{2,4}	8	\$37
{3,4}	9	\$65 ✓
{1,2,3}	14 >10	—
{1,2,3,4}	19 >10	—

– Assignment Problem

- Assign n-people n-jobs with minimum total cost.

Ex:-

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

Optimal ⇒ (Person 1 + Job 2) + (Person 2 + Job 1) + (Person 3 + Job 3)
= 9

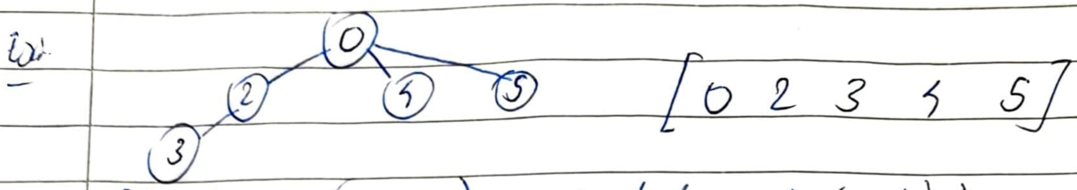
_/_/_

- DFS

- Starts traversing graph at arbitrary vertex by marking it as visited (1) and on each iteration, the algorithm proceeds to an unvisited vertex adjacent to current vertex.

- It is convenient to use a stack (LIFO) to trace operation.

Ex.



→ (Graph G (V, E) and starting node (root))

O(V+E)

```
count ← 0
for each vertex v in V do
    if v is marked with 0
        dfs(v)
dfs(v) {
    count ← count + 1 ; mark v with count
    for each vertex w in V adjacent to v do
        if w is marked with 0
            dfs(w)
}
```

- BFS

- On the other hand, BFS starts at root node and searches all nodes at current depth/level before moving on to nodes at next depth level.

- It is convenient to use a queue (FIFO) to trace operation.

- First, starting vertex is marked as visited. On each iteration,

algorithm identifies unvisited vertices adjacent to front vertex, marks them as visited and adds them to queue. Later, front vertex is removed from queue.

→ count ← 0 $O(V+E)$
for each vertex v in V do
  if v is marked with 0 (unvisited)
    bfs(v)
bfs(v) {
  count ← count + 1; mark v with count and init queue with v
  while queue is not empty do
    for each vertex w in V adjacent to front vertex do
      if w is marked with 0
        count ← count + 1 ; mark w with count.
        add w to queue.
    remove front vertex from queue
}

	DFS	BFS
Data structure	stack	queue
Edge types (undirected)	Tree and back edges	Tree and cross edges
Vertex orderings	2	1
Applications	Articulation points.	Minimum-edge paths
Efficiency (adj. matrix)	$O(V^2)$	$O(V^2)$
Efficiency (adj. list)	$O(V+E)$	$O(V+E)$

- Applications of DFS: Produces minimum spanning tree, detects cycle in graph, bipartite graphs, path between cities.

- Applications of BFS: Shortest path, peer-to-peer networks, mutuals in social media, broadcasting.

Decrease & Conquer

- This technique is a way of solving problems by making them smaller step-by-step instead of solving the big problem all at once. Once solved, we can use that solution to solve the bigger one.

- This can be done in two ways: (i) Top-Down (Recursive) (solve smaller version first, then build up) (binary search) (ii) Bottom-Up (Iterative) (iterative Fibonacci sequence).

- There are 3 major variations of decrease-and-conquer:

(i) Decrease by a constant (1) (size of an instance is reduced by the same constant on each iteration)

(sequential search / insertion sort)

Ex: $$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$$

(ii) Decrease by constant factor (half)

Ex: If instance of size n is to compute a^n, then the instance of half it's size will compute $a^{n/2}$ where

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

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and (+)ve} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd} \\ 1 & \text{if } n = 0 \end{cases}$$

Ex: Tree traversal, binary search.

(iii) Variable Size Decrease

- Size reduction pattern varies from one iteration of an algo to another.

Ex:- $GCD(m, n) = GCD(n, m \% n)$
DFS, BFS, Quicksort.

☆ Insertion Sort

Algo: InsertionSort (A[0 ..... n-1])
//Sort a given array by insertion sort.
//Input: An array A[0 .... n-1] of n orderable elements.
//Output: Array A sorted in ~~descend~~ ascending order.

```
for i ← 1 to n-1 do
    v ← A[i]
    j ← i-1
    while j ≥ 0 and A[j] > v do
        A[j+1] ← A[j]
        j ← j-1
    A[j+1] ← v
```

Ex:

A[j] v
B H A R A T I
B H A R A T I
B A H R A T I
A B H R A T I
A B H R A T I
A B H A R T I
A B A H R T I
A A B H R T I
A A B H R T I
A A B ~~H~~ ~~R~~ R T
H I

— <u>Time Complexity</u>

- <u>Worst Case</u> (descending order):

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

$$= \frac{n(n-1)}{2} \in \Theta(n^2)$$

- <u>Best Case</u> (ascending order):

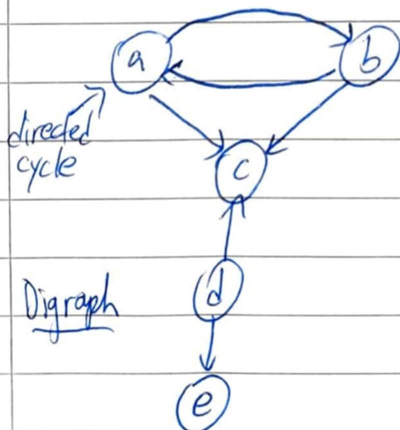
$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = (n-1) \in \Theta(n)$$

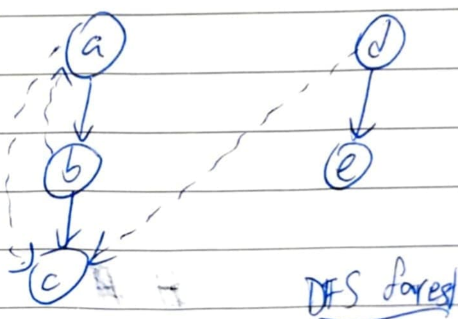
$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2)$$

☆ <u>Topological Sorting</u>

- It is a linear ordering of vertices such that there is an edge in the <u>DAG</u> (directed acyclic graph) from vertex $u \to v$, then u comes before v in ordering.

- Tree edge (ab, bc, de) (edge to new vertex in DFS tree)
  Back edge (ba) (edge to an ancestor)
  Forward edge (ac) (edge to a descendant)
  Cross edge (dc) (edge between different DFS tree in a disconnected graph)

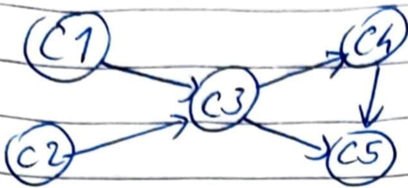




__/__/__

- **DAG** : If DFS forest of digraph has no back edge.

Ex:

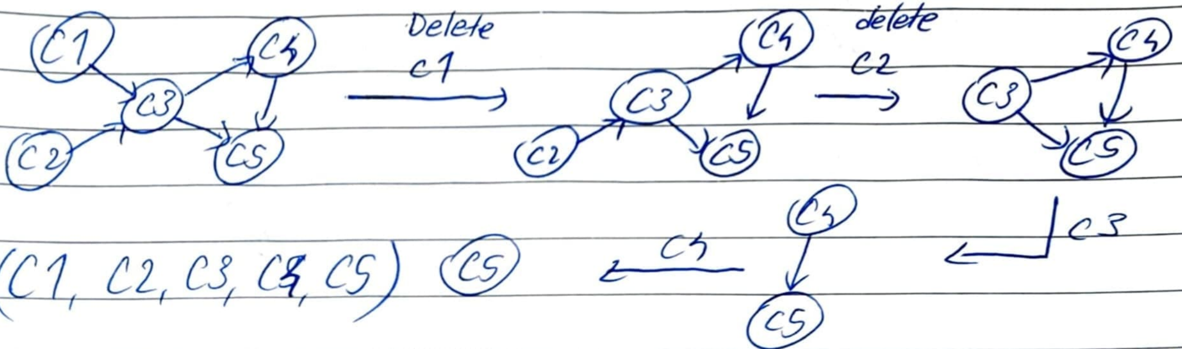


Popping off order :
C5, C4, C3, C1, C2

- Reversing the order of popped off traversal stack yields a solution to the topological sorting problem.

Topologically sorted list.

- Another method to obtain the solution is Source-Removal where you identify a source (vertex having no incoming edges) and delete it along with all outgoing edges from it. Order of deletion yields solution to topological sorting.



∴ (C1, C2, C3, C4, C5)

- Applications : (i) instruction scheduling in program compilation
(ii) Cell evaluation ordering in spreadsheet formulas

☆ **Binary Search** (decrease-by constant factor)

- It is an efficient algorithm for searching in a sorted array.

PTO →

Algorithm: BinarySearch($A[0 \dots n-1]$, K)
// Input: An array A sorted in ascending order and search key K
// Output: Index of array element $= K$ or -1 if does not exist

$l \leftarrow 0$; $r \leftarrow n-1$
while $l \leq r$ do
    $m \leftarrow \lfloor (l+r)/2 \rfloor$
    if $K = A[m]$ return m
    else if $K < A[m]$: $r \leftarrow m-1$
    else $l \leftarrow m+1$
return -1

- Time Complexity:

- After iteration 1, length of array $= n$
  After iteration 2, length of array $= n/2$

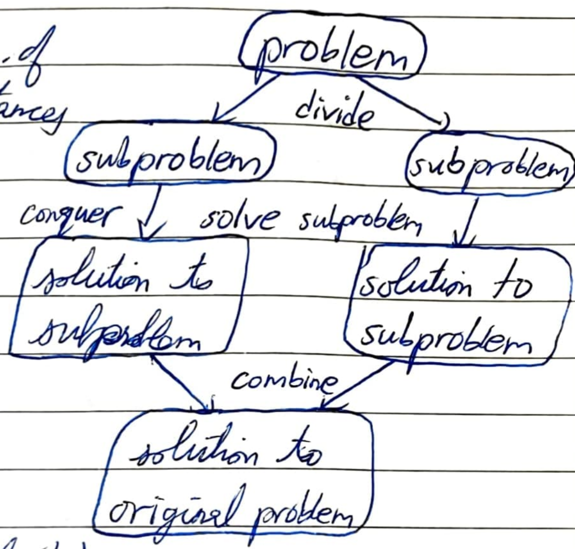
  After iteration k, length of array $= n/2^k = 1$

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

- Best Case ($O(1)$)
  Worst Case ($\Theta(\log n)$)

Divide & Conquer

1. **Divide** the problem into a no. of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively.
3. **Combine** the solutions to subproblems into solution for original problem.



- Master's Theorem (for time complexity)

$$T(n) = a\,T\left(\frac{n}{b}\right) + \Theta(n^k \log_p n)$$

$n \to$ size of problem
$a \to$ no. of subproblems
$n/b \to$ size of subproblem
$k >= 0$, p is real.

Case 1: If $a > b^k$, $T(n) = \Theta(n^{\log_b a})$

Case 2: If $a = b^k$, if $p < -1$, $T(n) = \Theta(n^{\log_b a})$
if $p = -1$, $T(n) = \Theta(n^{\log_b a} \cdot \log_2 n)$
if $p > -1$, $T(n) = \Theta(n^{\log_b a} \cdot \log_{p+1} n)$

Case 3: If $a < b^k$, if $p < 0$, $T(n) = O(n^k)$
if $p \geq 0$, $T(n) = \Theta(n^k \log_p n)$

Ex: $T(n) = 3T(n/2) + n^2$
Here $a = 3$, $b = 2$, $k = 2$, $p = 0$
Hence $a < b^k$, $(3 < 2^3)$

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

$(T(1) = 1)$

_ / _ / _

☆ Mergesort

- It sorts a given array A by dividing it into two halves B and C, sorting each of them recursively and then merging them into a single sorted array.

— **Algorithm**

→ Mergesort (A[0..... n-1])
// Sorts array A[0,.....n-1] by recursive mergesort
// Input : Array A[0...n-1] of orderable elements
// Output : Array A[0,... n-1] sorted in descending order

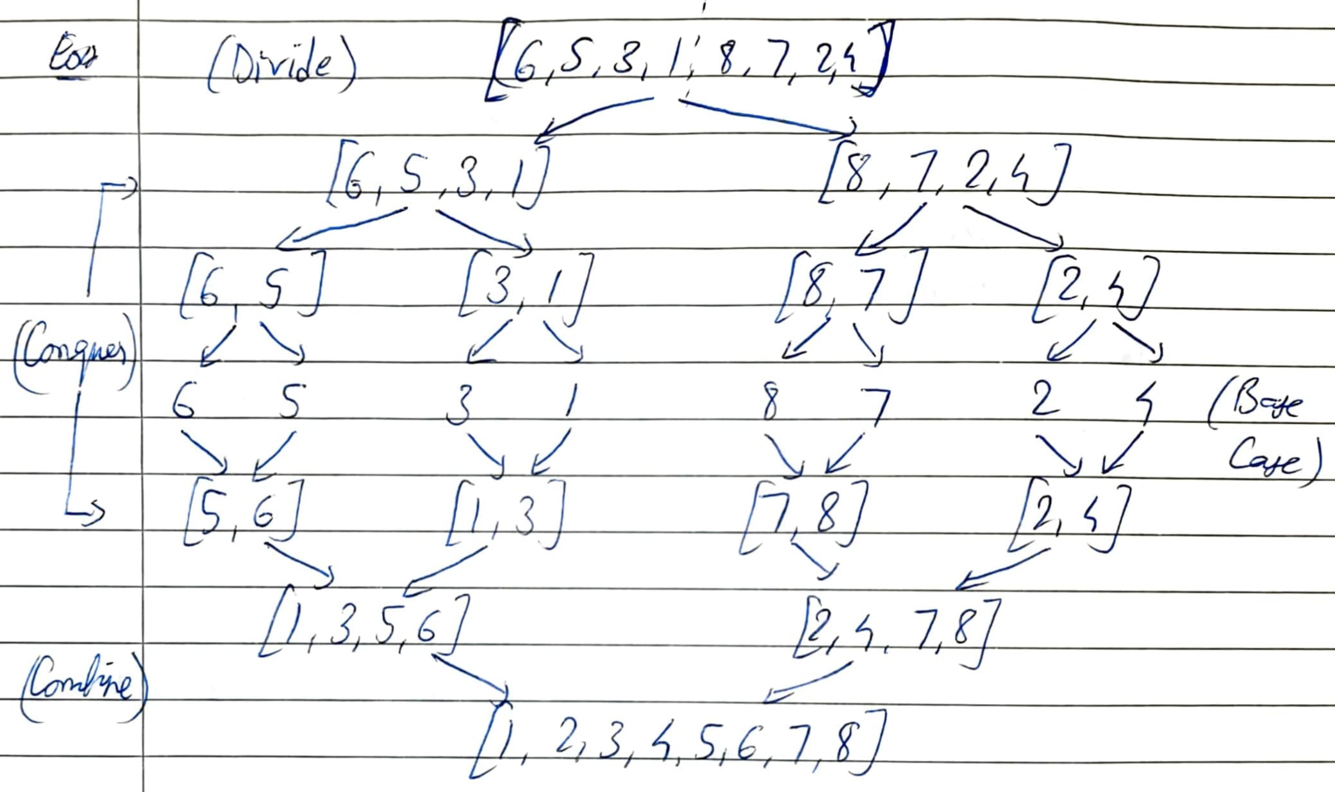
```
if n > 1
    copy A[0.....⌊n/2⌋ - 1]] to B
    copy A[⌊n/2⌋ ..... n-1] to C
    Mergesort (B)
    Mergesort (C)
    Merge (B, C, A)
```

→ Merge (B[0... p-1], C[0.....q-1], A[0.... p+q-1])
// Merges two sorted arrays into one sorted array
// Input : Arrays B[0....p-1] and C[0...q-1] both sorted
// Output : Sorted array A[0.....p+q-1]

```
while i<p and j<q do
    if B[i] ≤ C[j]
        A[k] ← B[i]
        i ← i+1
    else
        A[k] ← C[j]
        j ← j+1
    k ← k+1
```

if i = p (if all of B is inserted)
  copy C[j....q-1] to A[k...p+q-1] (remaining)
else (if all of C is inserted)
  copy B[i....p-1] to A[k...p+q-1]

Eg



– Time Complexity

- Divide part : $O(1)$ (calculating middle index takes constant time)
- Conquer part : We are recursively solving two subproblem so $2T(n/2)$
- Combine part : Worst case of merging is $O(n)$.

$$T(n) = O(1) + 2T(n/2) + O(n)$$
$$= 2T(n/2) + cn$$
$$(a = 2,\ b = 2,\ k = 1,\ p = 0)$$

By applying Master's theorem, $(a = b^k,\ p > -1)$

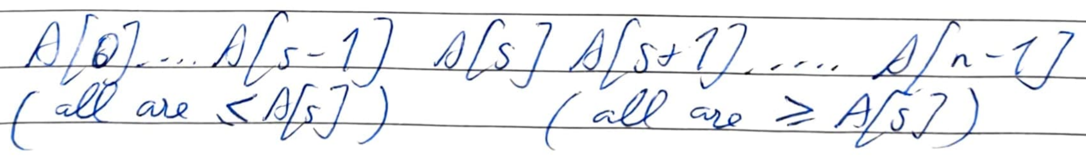
$$T(n) = \Theta(n \log n)$$

– Applications

- Database Sorting: Used in databases to efficiently sort large datasets like customer records and transactions
- Parallel Processing: Can be parallelized for better ~~processing~~ performance in multi-core and distributed systems.
- E-commerce: Sorts product listings by relevance, price or popularity for better user experience.
- External Sorting: Ideal for sorting data that doesn't fit in memory using external storage like hard drives.

★ Quicksort (Fastest)

- A partition is an arrangement of the array's elements so that all elements to the left of some pivot element A[s] is less than or equal to A[s] and all elements to the right of A[s] are greater than or equal to A[s].



① Select pivot element (for example, first element in subarray)
② Left → right scan (i) starts from second element. Since we want elements < pivot to be in left subarray, this scan skips over all elements < pivot and stops when element ≥ pivot
③ Right → left scan (j) starts from last element. Since we want elements > pivot to be in right subarray, this scan skips over all elements > pivot and stops when element ≤ pivot.

__/__/__

When scanning stops,

④ If $i < j$, we just exchange $A[i]$ with $A[j]$ and proceed.
If $i \geq j$, we exchange pivot with $A[j]$

— Algorithm

→ QuickSort(A[l....r])
// Sorts a subarray by quicksort
// Input : Subarray of array A[0....n-1] defined by left and right indices l and r.
// Output : Subarray A[l...r] sorted in ascending order.
if $l < r$
    s ← Partition(A[l...r])   (s is split position.)
    Quicksort(A[l... s-1])
    Quicksort(A[s+1....r])

→ Hoarse Partition(A[l....r])
// Partitions subarray by Hoare's algorithm, using first element as pivot.
// Input : Subarray of A[0....n-1] defined by left and right indices l and r ($l<r$)
// Output - Partition of A[l...r] with split position returned.
p ← A[l]
i ← l ; j ← r+1
repeat
    repeat i ← i+1 until A[i] ≥ p   ②
    repeat j ← j-1 until A[j] ≤ p   ③
    swap(A[i], B[j])   (exchange)
until i ≥ j   (they cross)
swap(A[i], A[j])   (undo previous swap)
swap(A[l], A[j])   (exchange pivot with A[j])
return j   (split position)

Ex:- (l at index 1, r at index 7)

	0	1	2	3	4	5	6	7
	(5)	3 i	1	9	8	2	4	7 j
	(5)	3	1	9 i	8	2	4 j	7
(Exchange)	(5)	3	1	4 i	8	2	9 j	7
	(5)	3	1	4	8 i	2 j	9	7
(Exchange)	(5)	3	1	4	2 i	8 j	9	7
(Cross)	(5)	3	1	4	2 j	8 i	9	7
(Undo swap)	(5)	3	1	4	2 j	8 i	9	7
(Change pivot)	[2	3	1	4]	[5]	[8	9	7]

($i \geq j$) (partition)

Left part:

	(2)	3 i	1	4 j
	(2)	3 i	1 j	4
(Exchange)	(2)	1 i	3 j	4
(Cross)	(2)	1 j	3 i	4
	[1]	[2]	[(3)	4 ij]
			[3]	[4]

Right part:

	(8)	9 i	7 j
(Exchange)	8	7 i	9 j
(Cross)	8	7 j	9 i
	[7]	[8]	[9]

- Time Complexity

- Best Case (Pivot divides array evenly)

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

$T(n) \leq 2T(n/2) + O(n)$

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

- Average Case (pivot is chosen randomly)

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

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

- Worst Case (pivot is always smallest / largest element)
(array is already sorted)
(array is partitioned into subarrays of size (n-1) and 0)

$$T(n) = T(n-1) + T(0) + \theta(n)$$
$$= c(n-1) + c(n-2) + \dots + c$$
$$= \sum_{i=1}^{n-1} c(n-i)$$
$$= c\sum_{k=1}^{n-1} k = c\left(\frac{1}{2}(n-1)n\right) = O(n^2) \quad (k = n-i)$$

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

Multiplication of Large Integers

- We multiply two n-digit integers 'a' and 'b' using divide and conquer method. We split them into two halves.

$$a = a_1 a_0 = a_1 10^{n/2} + a_0$$
$$b = b_1 b_0 = b_1 10^{n/2} + b_0$$

$$c = a \times b = (a_1 10^{n/2} + a_0)(b_1 10^{n/2} + b_0)$$
$$= (a_1 \times b_1) 10^n + (a_1 \times b_0 + a_0 \times b_1) 10^{n/2} + (a_0 \times b_0)$$
$$c = c_2 10^n + c_1 10^{n/2} + c_0$$

- **Algorithm**

```
DC_LI_MULTIPLICATION(A, B)
// Input: A = a_{n-1} ..... a_1 a_0  and  B = b_{n-1} ..... b_1 b_0
// Output: C = A x B
if |a| == 1 or |b| == 1 then
    return A * B
else: n <- max(|A|, |B|)
    c2 <- DC_LI_MULTIPLICATION(a1, b1)
    c0 <- DC_LI_MULTIPLICATION(a0, b0)
    c1 <- DC_LI_MULTIPLICATION(a1 + a0, b1 + b0)
    return c2 * 10^n + (c1 - c0 - c2) * 10^(n/2) + c0
end
```

– Time Complexity

- Since there are 3 multiplications of $(n/2)$ digit numbers, recurrence of no. of multiplications $M(n)$:

$$M(n) = 3M(n/2) \quad \text{for } n > 1,\ M(1) = 1.$$

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

$$M(2^k) = 3M(2^{k-1}) = 3(3M(2^{k-2}))$$
$$= 3^i M(2^{k-i}) = 3^k M(2^{k-k}) = 3^k$$

$$\left[M(n) = 3^{\log_2 n} = n^{\log_2 3} \right]$$

- Since there are 5 additions and 1 subtraction,

$$A(n) = 3A(n/2) + cn$$
$$A(n) \in \Theta(n^{\log_2 3})$$

Ex: $a = 123456$, $b = 786932$ $(n = 6)$

$a_1 = 123$, $a_0 = 456$, $b_1 = 786$, $b_0 = 932$

$$c = a \times b = c_2 10^n + c_1 10^{n/2} + c_0$$

~~$c_2 = a_1 * b_1 = 909\ 56088$~~
~~$c_0 = (a_0 \times b_1)\ a_0 * b_0 = 1388,$~~
~~$c_1 = (a_1 + a_0)(b_1 + b_0) - c_2 - c_0 =$~~

$c_2 = a_1 \times b_1 = 96678$
$c_0 = a_0 \times b_0 = 424992$
$c_1 = (a_1 + a_2)(b_1 + b_2) - c_0 - c_2 = 473058$

$c = 97151476992$ //

Strassen's Matrix Multiplication

- Used to multiply two 2×2 matrices A and B to get C.

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

where
$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$
$m_2 = (a_{10} + a_{11}) * b_{00}$
$m_3 = a_{00} \times (b_{01} - b_{11})$
$m_4 = a_{11} \times (b_{10} - b_{00})$
$m_5 = (a_{00} + a_{01}) \times b_{11}$
$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$
$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$

- Algorithm

```
Strassen(n, a, b, c)
if n <= 2
    return brute_force(a, b)        (a*b)
else
    Partition a into 4 submatrices (a00, a01, a10, a11)
    Partition b into 4 submatrices (b00, b01, b10, b11)
    ~~Rotation~~
    Strassen(n/2, a00+a11, b00+b11, m1)
    Strassen(n/2, a10+a11, b00, m2)
    Strassen(n/2, a00, b00-b11, m3)
    Strassen(n/2, a11, b10-b00, m4)
    Strassen(n/2, a00+a01, b11, m5)
    Strassen(n/2, a10-a00, b00+b01, m6)
```

Strassen (n/2, a01 - a11, b10 + b11, m7)

$$C = \begin{bmatrix} m1 + m4 - m5 + m7 & m3 + m5 \\ m_2 + m_4 & m_1 + m3 - m2 + m6 \end{bmatrix}$$

end if
return (C)

Time Complexity

- Since 7 multiplications and 18 additions/substractions take place,

$$M(n) = 7M(n/2) = n^{\log_2 7}$$

$$A(n) = 7A(n/2) + 18(n/2)^2$$

$$A(n) \in \Theta(n^{\log_2 7})$$

Eg:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 8 & 5 \\ 20 & 13 \end{bmatrix}$$

$m_1 = 5 \times 5 = 25$

$m_2 = 7 \times 4 = 28$

$m_3 = 1 \times 2 = 2$

$m_4 = 4 \times -2 = -8$

$m_5 = 3 \times 1 = 3$

$m_6 = 2 \times 7 = 14$

$m_7 = -2 \times 3 = -6$

Transform & Conquer

- In transformation stage, problem's instance is modified / transformed into simpler or more convenient instance, or to different representation of same instance, or to instance of different problem whose algorithm is already available. Then it is solved.

Presorting

— Checking element uniqueness in an array.

→ PresortElementUniqueness (A[0.... n-1])
```
sort the array A
for i <- 0 to n-2 do
    if A[i] = A[i+1] return false
return true.
```

- $T(n) = T_{sort}(n) + T_{scan}(n) = \Theta(n \log n) + \Theta(n)$
  $T(n) = \Theta(n \log n)$

- Applications : (i) Database Operations (in database, uniqueness constraints are enforced on columns to ensure no duplicates are found)
  (ii) Input Validation (in user registration form, it is essential to verify that chosen username is unique among existing users)

— Computing Mode

$T(n) = T_{sort}(n) + T_{scan}(n) = \Theta(n \log n) + \Theta(n)$
$= \Theta(n \log n)$

→ PresortMode (A[0....n-1])
sort the array A
i ← 0
modefrequency ← 0 (highest frequency)
while i ≤ n-1 do
  runlength ← 1 ; runvalue ← A[i]
  while i + runlength ≤ n-1 and A[i + runlength] = runvalue
    runlength ← runlength + 1
  if runlength > modefrequency
    modefrequency ← runlength
    modevalue ← runvalue
  i ← i + runlength
return modevalue

- Applications: Data Analysis and Educational Systems

— Searching Problem

- Consider a problem of searching given value v in array of n sortable items. Brute-force solution is sequential search which has $\Theta(n)$ time complexity.

- If array is sorted first, we can apply binary search which requires only $(\log_2 n + 1)$ comparisons.

$$T(n) = T_{sort}(n) + T_{search}(n) = \Theta(n\log n) + \Theta(\log n)$$
$$= \Theta(n\log n)$$

—x—

Balanced Search Trees

– Red-Black Tree

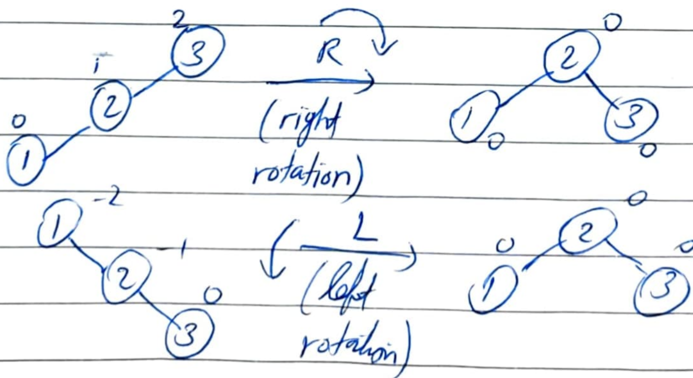
- It is a binary search tree in which each node is colored red / black such that:

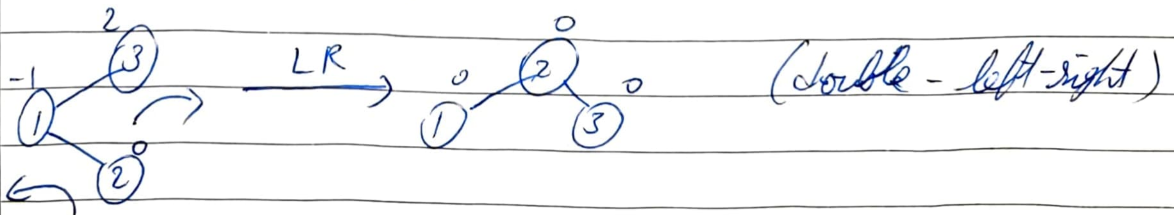
(a) Root is black, children of red node are black.
(b) No two consecutive red nodes
(c) Paths from root → leaves (NULL) must have same no. of black nodes
(d) New nodes inserted are always red.

– AVL Trees

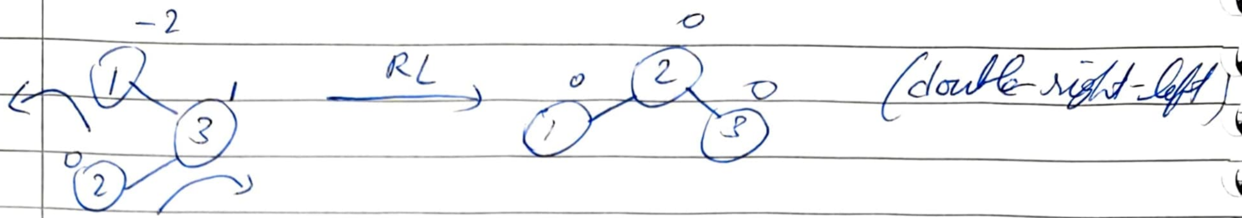
- Balance factor = Height (left) – Height (right) (subtree)
  (or diff in no. of levels)
- It is a BST in which balance factor of every node is either 0, 1 or –1

- If tree is unbalanced, we transform the tree by rotation.





__/__/____

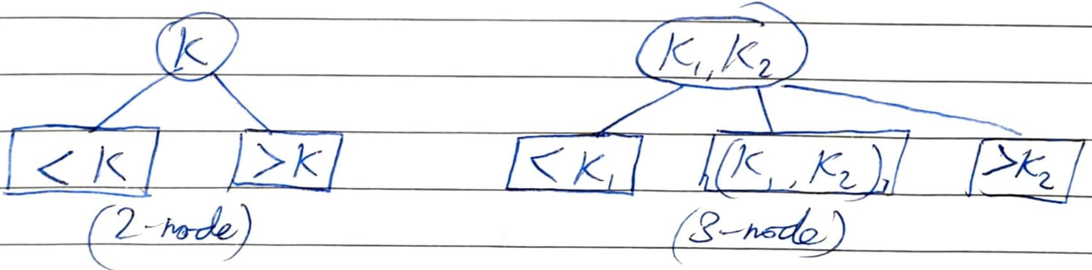


- $T(n) = \Theta(\log n)$
- To calculate height of AVL tree,

$$\lfloor \log_2 n \rfloor \leq h < (1.4405 \log_2(n+2) - 1.3277)$$

2-3 trees

- Nodes with two children are called 2-nodes (one data value and 2 children). Nodes with three children are called 3-nodes (two data values and 3 children)



- $(\log_3(n+1) - 1) \leq h \leq (\log_2(n+1) - 1)$

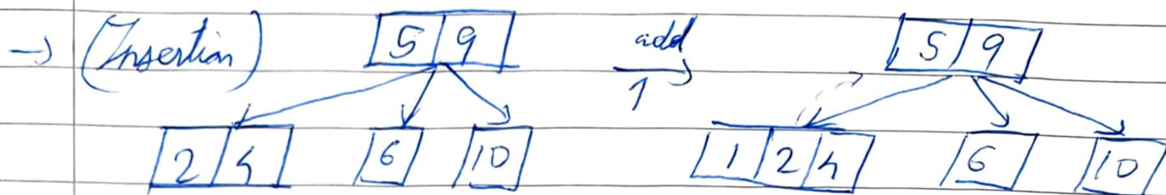
→ To search for key K in given 2-3 tree T,

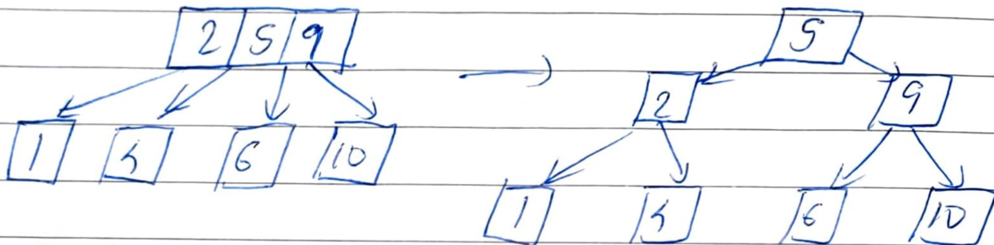
Base Cases:
- If T is empty, return False
- If current node contains data value = K, return True
- If we reach leaf node, and it doesn't contain K, return F.

Recursive Calls:
- If K < currentNode.leftVal, explore left subtree
- If currentNode.leftVal < K < currentNode.rightVal, explore middle subtree.

• If k > currentNode.rightVal, explore right subtree.

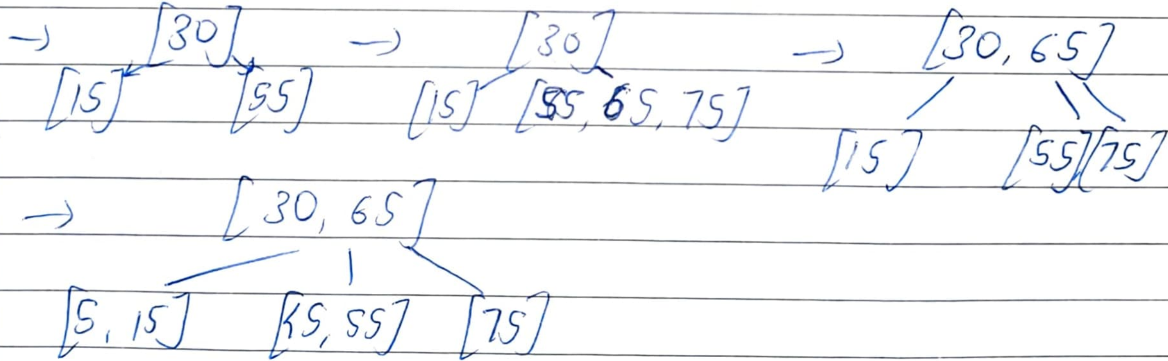
→ (Insertion)



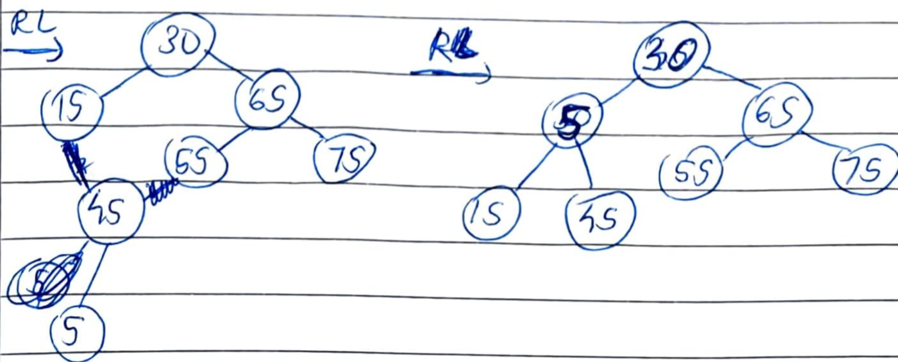


Ex:- Construct AVL and 2-3 tree for the following data:
55, 30, 15, 75, 65, 45, 5.

(a) 2-3 tree: [55] → [30, 55] → [15, 30, 55]

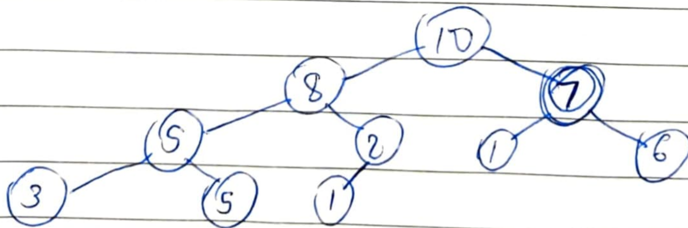


(b) AVL Tree:



☆ Heaps

- It is a binary tree with keys assigned to each node
- Heap / binary tree is essentially complete, all levels are full except possibly the last level.
- Key in each node $\geq$ keys in it's children (Parental dominance)
- Root of heap always contains largest element. Height of heap $= \log_2 n$ (n nodes)
- Heap can be implemented as an array $H[1 \ldots n]$ where parental nodes are in first $n/2$ positions of array and leaf keys will occupy last $n/2$ positions
- Children of key i are in positions $(2i)$ and $(2i+1)$. Parent of key i $(2 \leq i \leq n)$ will be in position $i/2$.
- $H[i] \geq \max(H[2i], H[2i+1])$



1	2	3	4	5	6	7	8	9	10
10	8	7	5	2	1	6	3	5	1

(parents) (children) (leaves)

- __/__/__

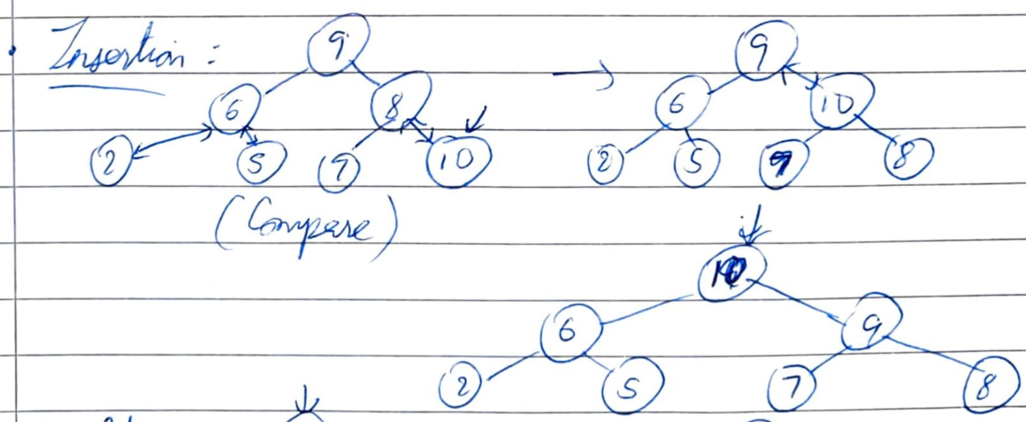
Construction (Bottom Up)

→ HeapBottomUp (H[1....n])

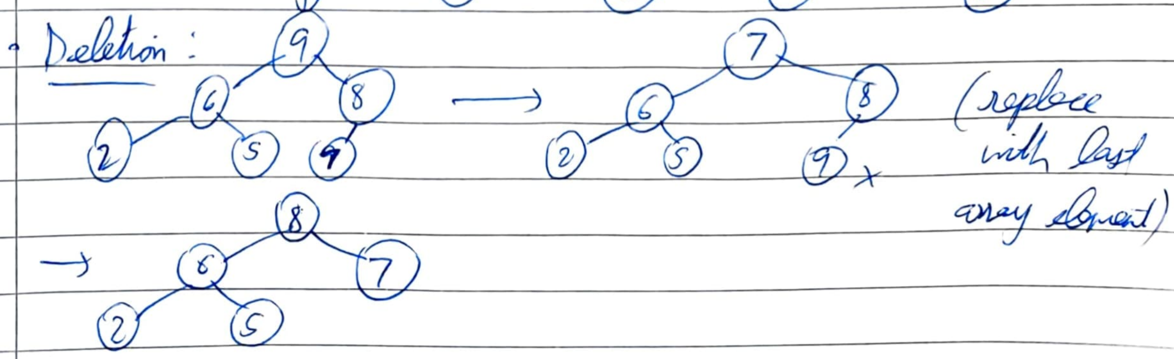
```
// Input: An array H of orderable items
// Output: A heap H[1...n]
for i ← ⌊n/2⌋ downto 1 do
    k ← i ; v ← H[k]
    heap ← false
    while not heap and 2*k ≤ n do
        j ← 2*k
        if j < n
            if H[j] < H[j+1], j ← j+1
        if v ≥ H[j]
            heap ← true
        else H[k] ← H[j]; k ← j
    H[k] ← v
```

Construction (Top-Down)

• Insertion:



• Deletion:



• Heapsort involves heap construction from given array and applying root-deletion operation $(n-1)$ times to remaining heap. As a result, element are eliminated in descending order but result is given in ascending order (sorted)

__/__

SPACE AND TIME TRADE-OFFS

- Algorithm trades increased space usage with decreased time and vice-versa.

- Input enhancement is the approach where we preprocess the problem's input, in whole/part and store additional info obtained to accelerate solving problem later.

Ex:- Counting method for Sorting, Boyer-Moore and Horspool algo for string matching.

- Prestructuring simply uses extra space to facilitate faster and more flexible access to data.

Ex: Hashing, Indexing with B-trees.

✱ Sorting by Counting

- Idea: Count no. of elements < this element, which gives the position of this element in the final sorted array.

→ Algorithm : ComparisonCountingSort(A[0.....n-1])

```
// Input : A[0....n-1] , Output : S[0....n-1].
for i ← 0 to n-1 do Count[i] ← 0
for i ← 0 to n-2 do
    for j ← i+1 to n-1 do
        if A[i] < A[j]
            Count[j] ← Count[j] + 1
        else Count[i] ← Count[i] + 1
for i ← 0 to n-1 do S[Count[i]] ← A[i]
return S
```

_/_/_

Ex: Array A : [62, 31, 84, 96, 19, 47]

Initially,	0	0	0	0	0	0
i=0	3	0	1	1	0	0
i=1		1	2	2	0	1
i=2			4	3	0	1
i=3				5	0	1
i=4					0	2
Finally;	3	1	4	5	0	2

→ [19, 31, 47, 62, 84, 96]

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} ((n-1)-(i+1)+1) = \sum_{i=0}^{n-2}(n-i-1)$$

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

- <u>Time Complexity</u> : $O(n+k)$ where n → no. of elements in input array
  k → range of input

☆ <u>Distribution Counting</u>

- To avoid overwriting, if duplicates found.

→ <u>Algorithm</u> : DistributionCountingSort(A[0....n-1], l, u)
//Input: A[0....n-1] of integers between l and u. ($l \leq u$)
//Output: S[0...n-1] in ascending order.
(init) for $j \leftarrow 0$ to $(u-l)$ do $D[j] \leftarrow 0$
(freq) for $i \leftarrow 0$ to $(n-1)$ do $D[A[i]-l] \leftarrow D[A[i]-l]+1$
(cummu) for $j \leftarrow 1$ to $(u-l)$ do $D[j] \leftarrow D[j-1]+D[j]$
for $i \leftarrow n-1$ down to 0 do
  $j \leftarrow A[i]-l$
  $S[D[j]-1] \leftarrow A[i]$
  $D[j] \leftarrow D[j]-1$
return S

Here $l \to$ lowest integer, & $u \to$ largest integer in A
D contains frequencies (cummulative) of each integer.

Ex:- $A = [13, 11, 12, 13, 12, 12]$ (Here, $l = 11$, $u = 13$) ($n = 6$)

Array values :	11	12	13
Frequency :	1	3	2
Cummulative / Distribution values :	1	4	6

	D[0...2]			S[0....5]					
A[5] = 12	1	4	6				12		
A[4] = 12	1	3	6			12			
A[3] = 13	1	2	6						13
A[2] = 12	1	2	5		12				
A[1] = 11	1	1	5	11					
A[0] = 13	0	1	5					13	

[11, 12, 12, 12, 13, 13]

Note: Time Complexity : $O(n+k)$

★ Horspool Algo (String Matching)

→ Algorithm : HorspoolMatching ($P[0 \ldots m-1]$, $T[0 \ldots n-1]$)

```
// Input : pattern P, text T. Output : Index of left end of substring
ShiftTable (P[0... m-1])
i ← m-1
while i ≤ n-1 do
    k ← 0
    while k ≤ m-1 and P[m-1-k] = T[i-k] do   (match)
        k ← k+1                              (← ← ←)
    if k = m return i-m+1                    (start of substring)
    else i ← i + Table[T[i]]                 (shift → →)
return -1;
```

__/__/__

Note: Time Complexity: O(nm) (worst case) or O(n).

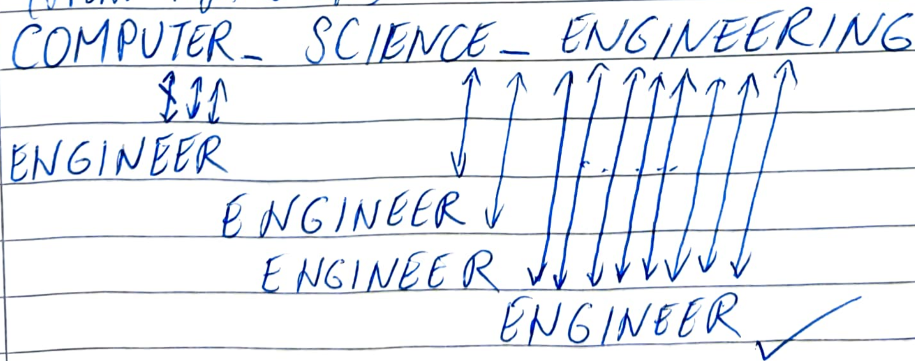
Ex: Text = COMPUTER SCIENCE ENGINEERING (n=28)
Pattern = ENGINEER ~~ING~~ (m=8)

To construct shift table for pattern, we mention the order in which each letter in the pattern appears in decreasing order from m and use that order number on the top of the stack for shifting the pattern through the text. (except last character). Shift value for character not present in pattern will be m.

Here,

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	_
8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
				7		5		4					6													
				2									3													
				1																						

Shifting is done based on the shift value of the first character on the text that does not match the adjacent character on pattern. (from right to left)



- Once shift value is used, it is popped from the stack and next value is used.

__/__/____

✱ Boyer - Moore

- Here, pattern is shifted based on d_1 (Bad Character Table) or d_2 values (Good Suffix) tables values.

$$\text{Shift}, d = \begin{cases} d_1, & \text{if } k=0 \\ \max(d_1, d_2), & \text{if } k>0 \end{cases}$$

where $d_1 = \max(t_1(c) - k, 1)$

- Good Suffix Table: Here d_2 values are determined for certain values of k (index of mismatch in text T), by checking any other occurrences of the selected suffix in pattern and it's position is obtained (from right to left) (else (m-1))

- Bad Character Table: This just gives the index of the first occurrence of the symbol/character in the pattern, ~~from~~ by taking last symbol as index 0 and reading from right to left, and setting it as the $t_1(c)$ value. (else m)

$d_1 = t_1(c) - k$

Exc: Searching for Pattern = BAOBAB in
Text = BESS_KNEW_ABOUT_BAOBABS

Bad Character Table

c	A	B	C	D	.....	O	.....	Z	_
$t_1(c)$	1	2	6	6	.....	3	.....	6	6

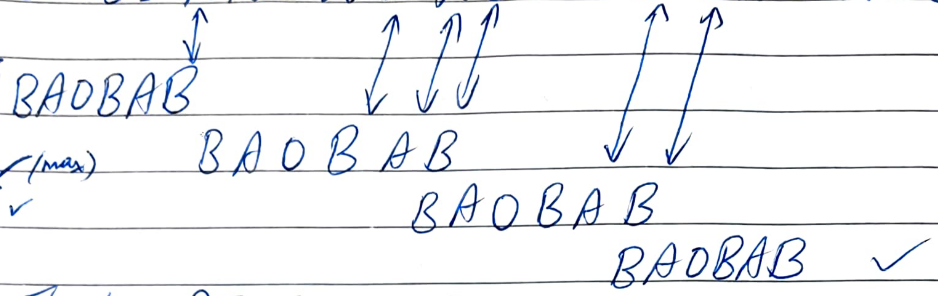
(Here, B (first character) is ignored)

Suffix Table:

k	Pattern	d_2
1	BAOBAB	2
2	BAOBAB	5
3	BAOBAB	5
4	BAOBAB	5
5	BAOBAB	5

(no previous occurrence so $d_2 = m-1$)

_/_/_



$d_1 = t_1(K) - 0 = 6$

$d_1 = t_1(_) - 2 = 4$

$d_2 = 5$ (AB) ✓ (max)

$d_1 = t_1(_) - 1 = 5$ ✓

$d_2 = 2$

Ex: Text: STOU_TOR_STOR
Pattern: STOR

k	Pattern	d_2
1	STOR	4
2	STOR	4
3	STOR	4

GS → (row 1)

c	A	B	....	O	R	O	R	S	T
$t_1(c)$	4	4	....	1	4	4	4	3	2

↑ BS

Ex: STOU_TOR_STOR

$d_1 = t_1(U) - 0 = 4$ STOR

$d_1 = t_1(_) - 3 = 1$ STOR

$d_2 = 4$ ✓ STOR

$d_1 = t_1(O) - 0 = 1$ STOR ✓

Hashing

- Technique used for efficient storing and retrieving data using hash function and hash table.
- Hash function transforms a key into an index in a hash table enabling fast lookup. It determines the hash address where key is stored in hash table.
- Here, the key is said to uniquely identify a record.

Ex: h(key) = (sum of ASCII values / order in alphabet of characters) % table size

__/__/____

- This technique helps in directly being able to search, insert or delete a key in $O(1)$ time, compared to $O(n)$ using traditional methods.

Ex: key = FOOL
table size = 13,
$h(FOOL) = (6+15+15+12) \% 13 = 9$ (index in hash table)

- There are two types of hashing:

(a) Open Hashing (Separate Chaining)

- Each slot in Hash table stores a linked list. If two or more keys hash to same slot, they are inserted into that list.

- Efficiency of searching depends on length of linked list, which depend on table size and quality of hash function.

- Load Factor: $\alpha = \frac{n}{m}$ (n - keys) (m - cells of hash table)
Each list will be about (n/m) keys long.

Note: Avg. no. of pointers (chain links) inspected in
Successful searches $(S) \approx \left(1 + \frac{\alpha}{2}\right)$

Unsuccessful searches $(U) = \alpha$.

(b) Closed Hashing (Open Addressing)

- All keys are stored inside the table, no linked list.

_/_/_

- On collision, either check next slot (linear probing) or use a second hash function to jump slots (double hashing)
- The issue with linear probing is it can lead to clustering (long sequences of filled slots) which can be bad for efficiency.

Note: Upon deletion, use lazy deletion technique to mark slots that are deleted instead of making them empty to avoid ~~deletion~~ search failures.

- For implementing double hashing, table size must be prime and $h_2(key)$ should be relatively prime to table size.
- Another solution is rehashing, where we create a bigger table and reinsert all keys.

Note: In double hashing, we use another hash function $h_2(key)$ to determine a fixed increment for probing; after collision at $h_1(key)$.

$$(h_1(key) + h_2(key)) \% m \ldots$$

—x—

DYNAMIC PROGRAMMING

- Optimization technique used to improve efficiency of algorithms that use recursion to solve problems.

- In plain recursion, same subproblems are solved multiple times, leading to redundant work.

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

- Dynamic Programming solves this inefficiency by storing (memoizing) results of subproblems the first time they are computed.

- As a result, time complexity drops from exponential to polynomial/linear.

Ex:
```
f[0] = 0;
f[1] = 1;
for (i=2; i<=n; i++){
    f[i] = f[i-1] + f[i-2];
}
return f[n];
```

☆ **Knapsack Problem**

– Using Bottom-Up DP

- Objective: Put objects in the bag such that total weight does not exceed capacity of bag and profit/value is maximum.

_/_/_

Here $V[i,j] \rightarrow$ value of ith element/item for capacity j.
$v_i \rightarrow$ value of item i
$w_i \rightarrow$ weight of item i

$$V[i,j] = \begin{cases} V[0,j] = 0 \text{ for } j \geq 0 \\ V[i,0] = 0 \text{ for } i \geq 0 \\ V[i-1,j] \text{ if } w_i > j \\ \max(V[i-1,j], V[i-1,j-w_i]+v_i) \text{ if } w_i \leq j \end{cases}$$

Ex:-

item	weight	value
1	2	12
2	1	10
3	3	20
4	2	15

Knapsack capacity, $W = 5$
$n = 4$
(Build a table with $N+1$ rows and $W+1$ columns)

(capacity j)

	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1=2, v_1=12$	1	0	0	12	12	12	12
$w_2=1, v_2=10$	2	0	10	12	22	22	22
$w_3=3, v_3=20$	3	0	10	12	22	30	32
$w_4=2, v_4=15$	4	0	10	15	25	30	37

37 → solution (item 1, 2, 4)

Ex:-

item	weight	value
1	3	25
2	2	20
3	1	15
4	4	40
5	5	50

Capacity $W = 6$
$n = 5$
(Build at table with $N+1$ rows and $W+1$ columns

(Set first row and first column values all to 0)
(No item / no capacity → no value)

__/__/____

	i	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1=3, v_1=25$	1	0	0	0	25	25	25	25
$w_2=2, v_2=20$	2	0	0	20	25	25	45	45
$w_3=1, v_3=15$	3	0	15	20	35	40	45	60
$w_4=4, v_4=40$	4	0	15	20	35	40	55	60
$w_5=5, v_5=50$	5	0	15	20	35	40	55	(65) (item 3 + item 5)

– Using Memory Function

- While bottom-up approach (tabulation) builds the solutions iteratively from smallest subproblems upward, top-down approach (memoization) solves problems recursively but remembers already solved subproblems.

- Here, only necessary subproblems are solved, and stored.

Ex: For the first Knapsack example;

	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1=2, v_1=12$	1	0	0	12	12	12	12
$w_2=1, v_2=10$	2	0	–	12	22	–	22
$w_3=3, v_3=20$	3	0	–	–	22	–	32
$w_4=2, v_4=15$	4	0	–	–	–	–	37

$n=4$, $W=5$ (refer to previous table)
Here, we try to obtain the final largest value (bottom right corner) with the already saved/stored results and substitute backwards.

__/__/__

$V[4,5] = \max(V[3,5], V[3,3] + 15)$

$V[3,5] = \max(V[2,5], V[2,2] + 20)$

$V[3,3] = \max(V[2,3], V[2,0] + 20)$

$V[2,5] = \max(V[1,5], V[1,4] + 10)$

$V[2,2] = \max(V[1,2], V[1,1] + 10)$

$V[2,3] = \max(V[1,3], V[1,2] + 10)$

$V[2,0] = ~~\max~~ (V[1,0]) = 0$

$V[1,5] = \max(V[0,5], V[0,3] + 12) = 12$

$V[1,4] = \max(V[0,4], V[0,2] + 12) = 12$

$V[1,3] = \max(V[0,3], V[0,1] + 12) = 12$

$V[1,2] = \max(V[0,2], V[0,0] + 12) = 12$

$V[1,1] =$ ~~max(V[0,1], V[0,~~ $V[0,1] = 0$

Substitute backwards,

$V[2,3] = 22$ / $V[2,5] = 22$ / $V[3,5] = 32$

$V[2,2] = 12$ / $V[3,3] = 22$ / $V[4,5] = 37$

→ Algorithm: MFKnapsack (i, j)

// Input: i ← items, j ← knapsack capacity

// Output: Optimal feasible subset value.

// Initialize: Set first row and first column values to 0.
and rest as −1.

```
if F[i, j] < 0            (-1)
    if j < Weights[i]
        value ← MFKnapsack(i-1, j)
    else
        value ← max (MFKnapsack (i-1, j),
                Values[i] + MFKnapsack(i-1, j-Weights[i]))
    F[i,j] ← value
return F[i,j]
```

_/_/__

☆ Binomial Coefficient

- Binomial coeff $C(n,k)$ or ^nC_k defines coeff of term x^n in expansion of $(1+x)^n$. ^nC_k also defines no. of ways to select any k items out of n items.

$$^nC_k = \frac{n!}{(n-k)!\,k!} \quad (0 \leq k \leq n)$$

- Divide and conquer divides problem of size $C(n,k)$ into two subproblems each of size $C(n-1,k-1)$ and $C(n-1,k)$

$$^nC_k = {}^{n-1}C_{k-1} + {}^{n-1}C_k$$

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

$$C(n,0) = C(n,n) = 1$$

$$C[i,j] = \begin{cases} 1 & \text{if } i=j \text{ or } j=0 \\ C[i-1,j-1] + C[i-1,j] & \text{otherwise} \end{cases}$$

→ **Algorithm** Binomial (n, k)

```
if k==0 or k==n then return 1
else return Binomial(n-1, k-1) + Binomial(n-1, k)
End
```

(Time Complexity : $O(nk)$)

_/_/_

✯ Warshall Algorithm

- Helps determine the transitive closure of a directed graph with n vertices as n×n boolean matrix.

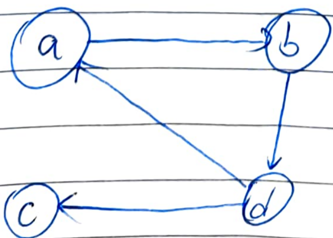
- It tells us ~~iff~~ that a path exists between two vertices i and j if:

(a) there is a direct edge from i to j (from adjacency matrix)
(b) there is an indirect path from i to j passing through any subset of vertices as intermediate vertices.

→ Algorithm: Warshall(A[1...n, 1...n])
// Input: Adjacency matrix ~~of~~ A of digraph having n vertices
// Output: transitive closure of digraph.

$R^{(0)} \leftarrow A$
for $k \leftarrow 1$ to n do
  for $i \leftarrow 1$ to n do
    for $j \leftarrow 1$ to n do
      $R^{(k)}[i,j] \leftarrow R^{(k-1)}[i,j]$ or $(R^{(k-1)}[i,k]$ and $R^{(k-1)}[k,j])$
return $R^{(n)}$

Ex:-



$R^{(0)} =$

	ⓐ	b	c	d
ⓐ	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

$R^{(1)} =$

	ⓐ	ⓑ	c	d
ⓐ	0	1	0	0
ⓑ	0	0	0	1
c	0	0	0	0
d	1	1	1	0

$R^{(2)} =$

	ⓐ	ⓑ	ⓒ	d
ⓐ	0	1	0	1
ⓑ	0	0	0	1
ⓒ	0	0	0	0
d	1	1	1	1

$R^{(3)} =$

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

$R^{(4)} =$

	a	b	c	d
a	1	1	1	1
b	1	1	1	1
c	0	0	0	0
d	1	1	1	1

- Here $R^{(0)}$ represents adjacency matrix with no intermediate vertices. Boxed row and column are used for getting next $R^{(k)}$ table. In this example, for each table $R^{(k)}$, 1's reflect existence of path with intermediate vertices no. $\leq k$.

- Here, in $R^{(1)}$, A acts as intermediate ($D \to A$, $A \to B \Rightarrow D \to B$ (1)) (→ k in algo)
  in $R^{(2)}$, A, B acts as intermediate vertices
  in $R^{(3)}$, A, B, C and in $R^{(3)}$, A, B, C, D act as intermediate

- Time Complexity = $\Theta(n^3)$

☆ Floyd's Algorithm

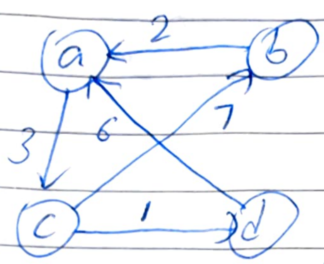
- Given a weighted directed/undirected graph, this algo helps find distances (length of shortest path) from each vertex to all other vertices, using a distance matrix D_{ij} (where D_{ij} indicates length of shortest path from vertex i to vertex j)

→ Algorithm : Floyd (W[1....n, 1...n])

```
// Weight matrix W of graph (input)
// Output: Distance matrix of shortest path length.
D ← W
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i, j] ← min(D[i, j], D[i, k] + D[k, j])
return D
```

$\Theta(n^3)$

Ex:-



$D^{(0)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

(A) $D^{(1)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	∞	7	0	1
d	6	∞	9	0

(A,B) $D^{(2)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	9	7	0	1
d	6	∞	9	0

(A,B,C) $D^{(3)} =$

	a	b	c	d
a	0	10	3	5
b	2	0	5	6
c	9	7	0	1
d	6	16	9	0

(A,B,C,D) $D^{(4)} =$

	a	b	c	d
a	0	10	3	5
b	2	0	5	6
c	7	7	0	1
d	6	16	9	0

—x—

GREEDY TECHNIQUE

- Solves optimization problems by making best local choice at each step to find global optimal solution. Works well if the problem has:

(a) Greedy Choice Property: A locally optimal choice leads to globally optimal solution.

(b) Optimal Substructure: An optimal solution contains optimal solutions to it's subproblems.

Prim's Algorithm

- To find, or obtain minimum spanning tree (connected, acyclic subgraph) from undirected connected graph (with smallest weight on edges)

→ Algorithm: Prims(G)
// Input: Weighted connected graph $G = \langle V, E \rangle$
// Output: E_T (set of edges already included in MST)
$V_T \leftarrow \{v_0\}$
$E_T \leftarrow \phi$

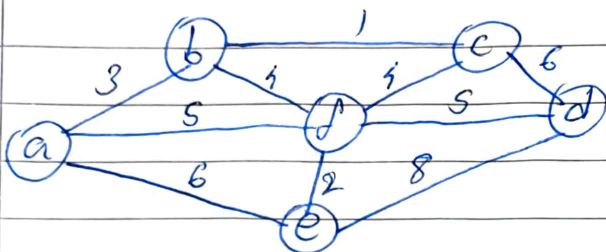
for $i \leftarrow 1$ to $|V| - 1$ do
 find minimum-weight edge $e^* = (v^*, u^*)$ among all edges (v, u) such that v is in V_T and u is in $V - V_T$
 $V_T \leftarrow V_T \cup \{u^*\}$
 $E_T \leftarrow E_T \cup \{e^*\}$
return E_T

Note: Here V_T → set of vertices already included in MST
V_0 → starting vertex

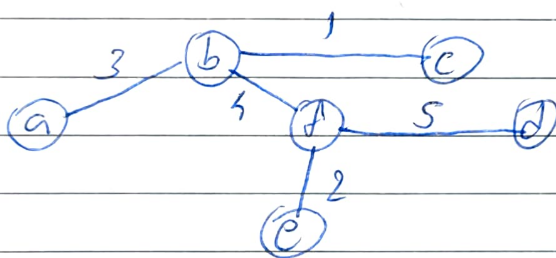
v^*, u^* → ~~vertex~~ vertex already in V_T (tree)
vertex not in V_T $(V - V_T)$

(v^*, u^*) → minimum-weight edge connecting tree & new vertex.

Ex:



Tree vertices	Remaining
a(-,-)	b(a,3), c(-,∞), d(-,∞), e(a,6), f(a,5)
b(a,3)	c(b,1), d(-,∞), e(a,6), f(b,4)
c(b,1)	d(c,6), e(a,6), f(b,4)
f(b,4)	d(f,5), e(f,2)
e(f,2)	d(f,5)
d(f,5)	



- Here, we first push starting vertex v_0 (A) into V_T and set E_T as null. Then, we find an edge (v^*, u^*) with the smallest weight such that $v^* \in V_T$ and $u^* \in (V - V_T)$ and then we add u^* to V_T and (v^*, u^*) weight to E_T. This is repeated until V_T contains all vertices.

- Time Complexity = $O(|E| \log |V|)$

Kruskal's Algorithm

- In contrast to the Prim's algo, Kruskal's algorithm finds the MST of a connected, weighted graph by sorting all edges in ascending order of weight and adding edges one-by-one to an initially empty subgraph, edges are skipped if they form a cycle. Repeated until MST has $|V| - 1$ edges.

__/__/____

Algorithm: Kruskal (G)

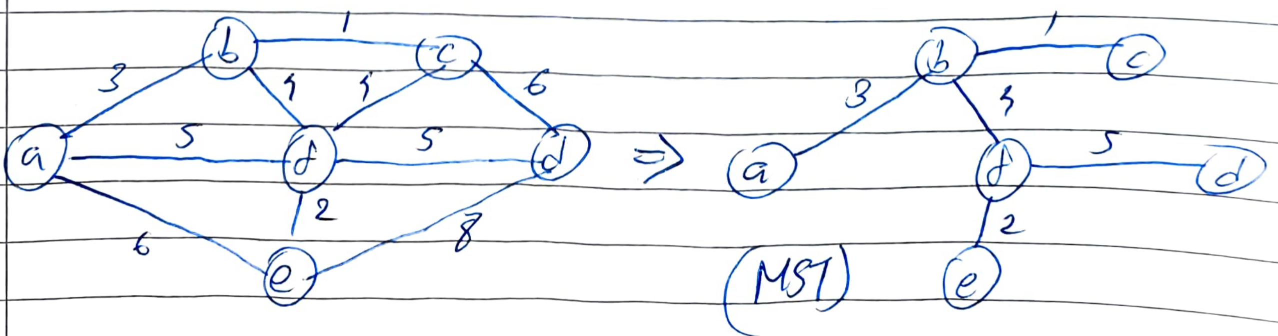
Sort E (edges) in ascending order of edge weights.
$E_T \leftarrow \phi$ (set of edges) (in MST)
ecounter $\leftarrow 0$
$k \leftarrow 0$
while ecounter $< |V| - 1$ do
    $k \leftarrow k + 1$ (more like an index in E)
    if $E_T \cup \{e_{ik}\}$ is acyclic (while adding edge)
        $E_T \leftarrow E_T \cup \{e_{ik}\}$;
        ecounter $\leftarrow$ ecounter $+ 1$;
return E_T

$TC = O(|E| \log |E|)$

Ex:

Tree edges	Sorted list of edges									
	(bc)	ef	ab	bf	cf	af	df	ae	cd	de
	(1)	2	3	4	4	5	5	6	6	8
bc	bc	(ef)	ab	bf	cf	af	df	ae	cd	de
1	1	(2)	3	4	4	5	5	6	6	8
ef	bc	ef	(ab)	bf	cf	af	df	ae	cd	de
2	1	2	(3)	4	4	5	5	6	6	8
ab	bc	ef	ab	(bf)	cf	af	df	ae	cd	de
3	1	2	3	(4)	4	5	5	6	6	8
bf	bc	ef	ab	bf	cf	af	(df)	ae	cd	de
4	1	2	3	4	4	5	(5)	6	6	8
df										
5										

(cf, af) will cause cycle



__/__/__

— **Disjoint Set Union / Union Find**

- Used to efficiently manage a partition of a set into disjoint subsets. (applied in Kruskal)

- makeSet(x) creates a singleton set $\{x\}$
  find(x) returns a representative / root of subsets containing x
  union(x, y) combines set containing x and y.

Ex: $S = \{1, 2, 3, 4, 5, 6\}$

makeset(i) $\forall i \in S \Rightarrow \{1\}, \{2\}, \{3\}, \{4\}, \{5\}$
union(1, 4) → $\{1, 4\}$
union(5, 2) → $\{5, 2\}$
union(4, 5) → $\{1, 4, 5, 2\}$

☆ **Dijkstra's Algorithm** $O(|E| \log |V|)$

→ Algorithm: Dijkstra(G, s) → (start vertex)
//Output: Length d_v: shortest path from $s \to v$, predecessor p_v
Initialize(Q) (empty queue)
for every vertex v in V
    $d_v \leftarrow \infty$; $p_v \leftarrow$ null
    Insert(Q, v, d_v)
$d_s \leftarrow 0$; Update/Decrease(Q, s, d_s)
$V_T \leftarrow \emptyset$
for $i \leftarrow 0$ to $|V| - 1$ do
    $u^* \leftarrow$ DeleteMin(Q)
    $V_T \leftarrow V_T \cup \{u^*\}$
    for every vertex u in $(V - V_T)$ adjacent to u^* do
        if $d_{u^*} + w(u^*, u) < d_u$
            $d_u \leftarrow d_{u^*} + w(u^*, u)$; $p_u \leftarrow u^*$
            Decrease/Update(Q, u, d_u)

__/__/__

- Here Q → priority queue which hold vertices not yet finalized, priority set by current shortest distance from s
  - d_v → Current shortest distance from s to vertex v. Except source, all d_v are initialized to ∞.
  - p_v → predecessor of v in shortest path from s.
  - V_T → set of already visited vertices.

Ex:

Tree vertices	Remaining vertices
a(−, 0)	b(a, 3), c(−, ∞), d(a, 7), e(−, ∞)
b(a, 3)	c(b, 3+4), d(b, 3+2), e(−, ∞)
d(b, 5)	c(b, 7), e(d, 5+4)
c(b, 7)	e(d, 9)
e(d, 9)	

Shortest paths: From a to b, a → b (3)
From a to d; a → b → d (5)
From a to c, a → b → c (7)
From a to e, a → b → d → e (9)
(d = p_v of e (predecessor))

- Here, after setting all distances from $s \to v = \infty$ and predecessors to null, ~~and~~ each vertex is inserted into queue with distance as priority (shorter distance = higher priority)

- Distance to source s is set to 0 and priority updated (highest)

- Throughout the loop, u^* is set as highest priority, shortest distance vertex and added to V_T. For every neighbour of u^* not yet finalized, if distance to u through u^* is less ~~that~~ than currently known distance d_u, then

update d_u and set predecessor to u^*. Update the priority of u in Q to reflect new distance

☆ Huffman Encoding / Decoding   O(n logn)

1. Start by treating each symbol as a separate tree
2. Combine two trees with smallest frequencies into one.
3. Repeat ~~up~~ until all symbols are combined into a single binary tree.

- Here, the tree assigns shorter binary codes to more frequent symbols and longer binary codes to less frequent symbols.

- During construction of Huffman tree, when two trees are combined, left edge from new root → 0
  right edge from new root → 1.

Example (given: symbol, frequency)

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

→ codewords to be obtained from Huffman tree.

[Refer to ~~table~~ trees in Math Section (Green)] :(

- Average number of bits per symbol =
$(2 \times 0.35) + (3 \times 0.1) + (2 \times 0.2) + (2 \times 0.2) + (3 \times 0.15) = 2.25$

-x-

(To be continued in MATH Section)

__/__/__

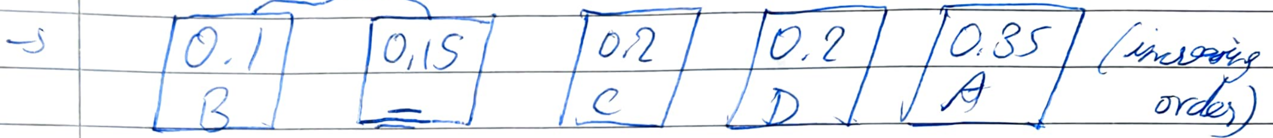
DAA (Greedy Technique)

☆ Huffman Encoding / Decoding

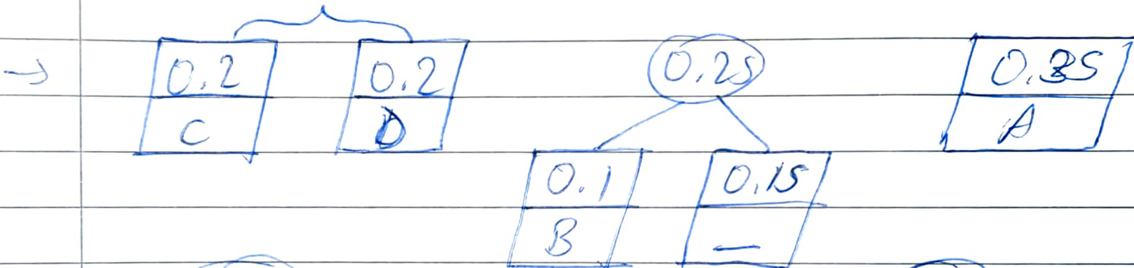
Ex:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

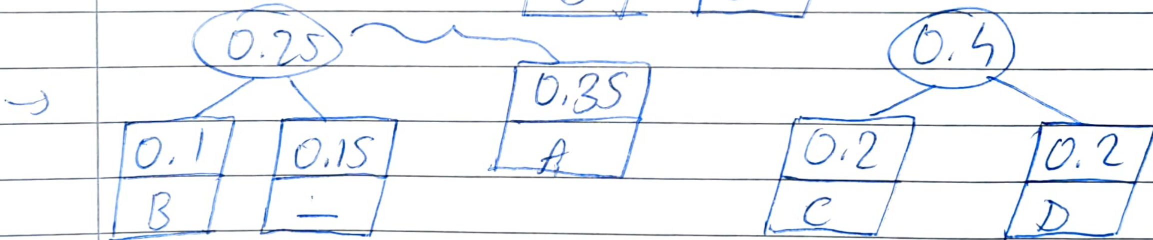
→



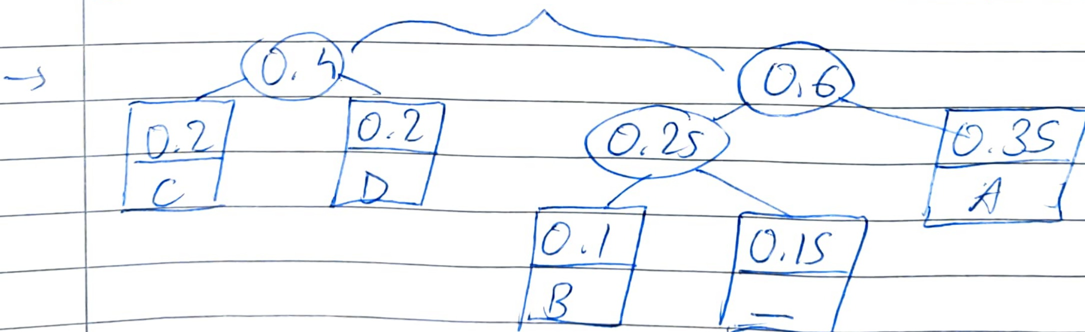
→



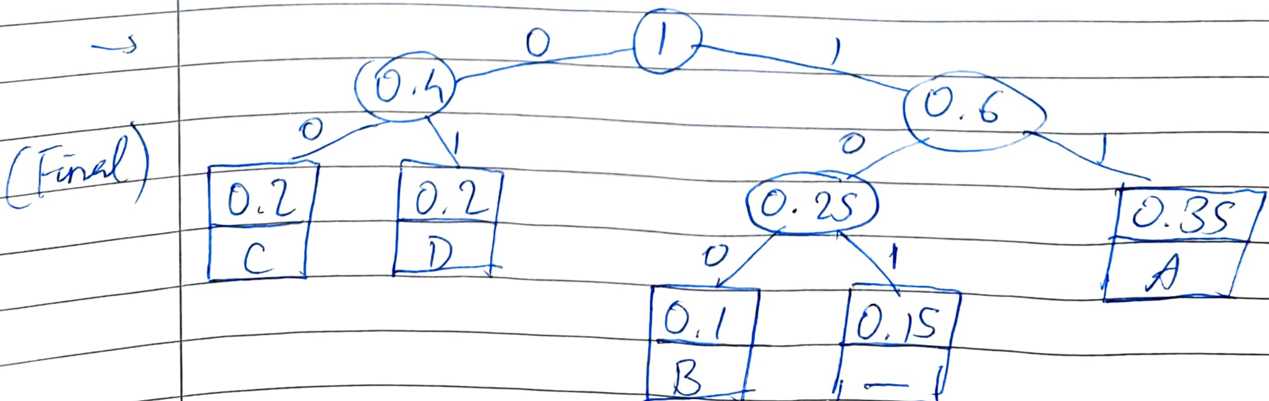
→



→



→ (Final)



Here, A → 11
B → 100
C → 00
D → 01
_ → 101

Ex: Encode DAD ⇒ 01 11 01
Decode 100110110111101 ⇒ BAD_AD

-x-

__/__/__

CHAPTER - 12

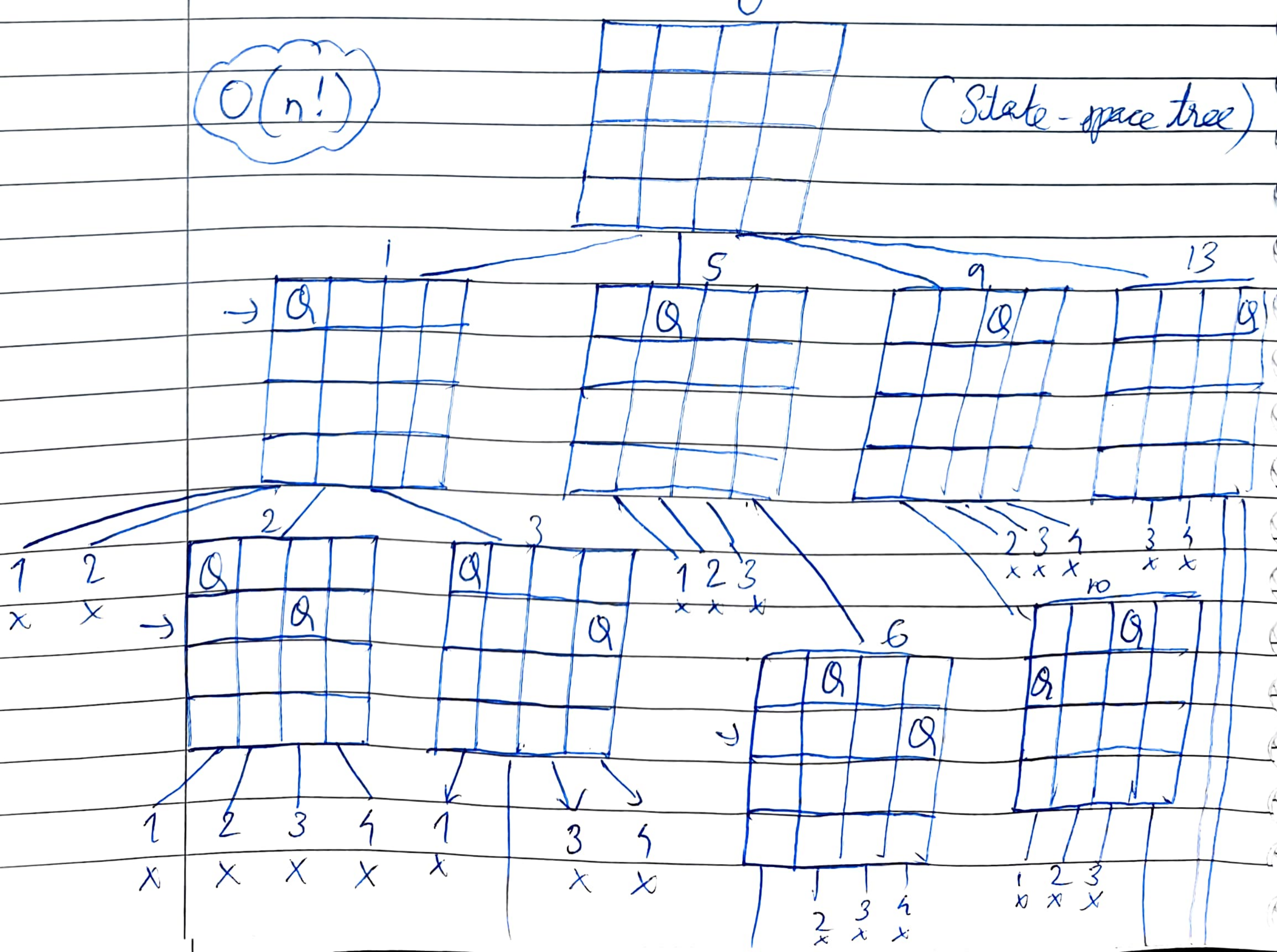
Backtracking

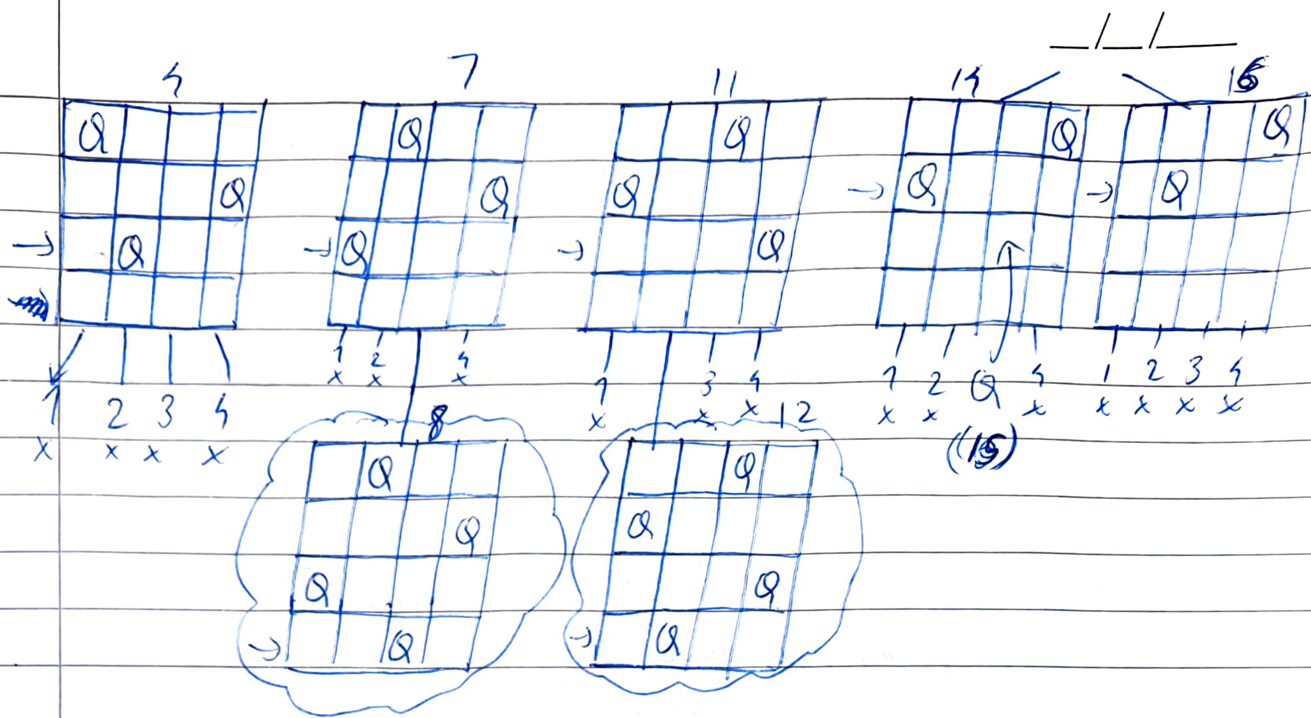
- Problem-solving technique that builds solutions step-by-step using recursion. If a steps leads to a dead end, it backtracks and tries a different path

n-Queen's Problem

- Given NxN chess board, arrange N queens such that no two queens attack each other horizontally, vertically or diagonally.

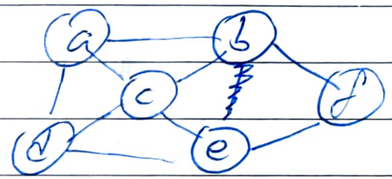
- N=1 (trivial case), N=2, 3 (solution not possible)

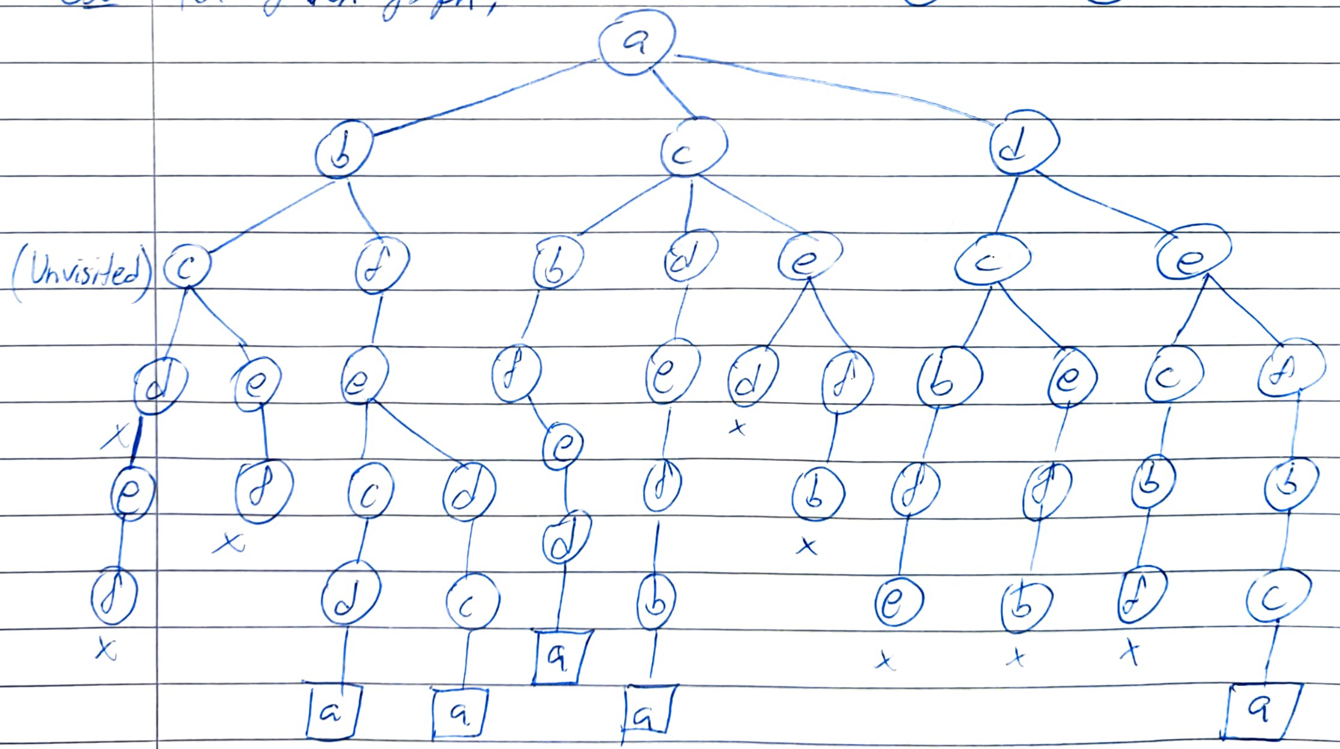




— *Travelling Salesman*

Eg: For given graph,





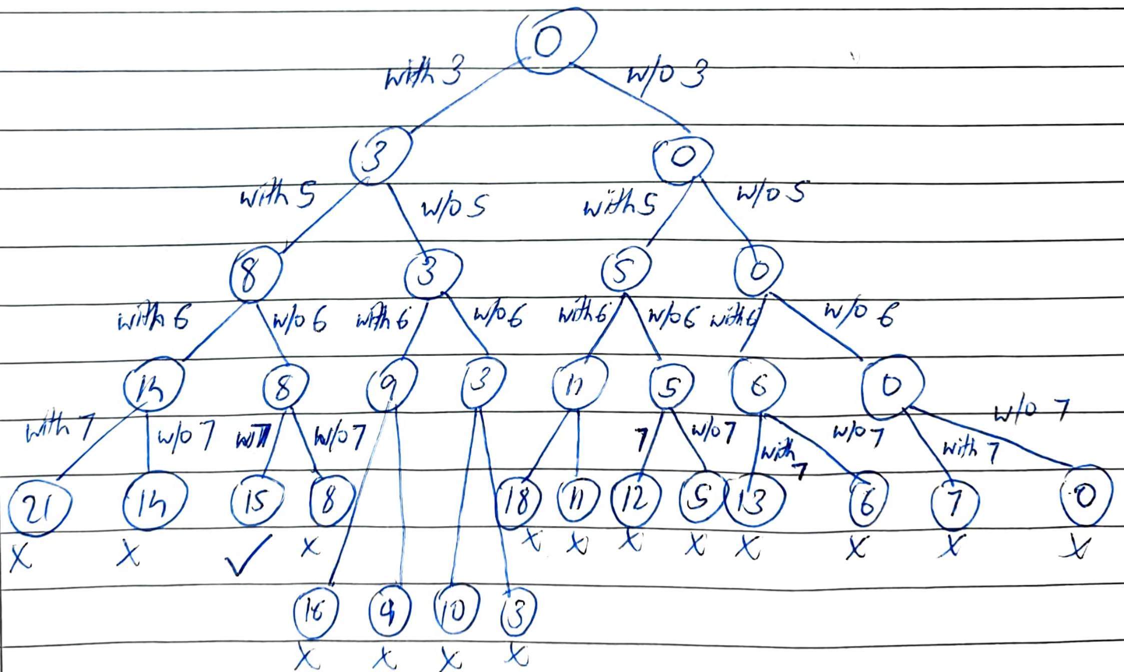
$O(n!)$

Subset-Sum $O(2^n)$

$A = \{3, 5, 6, 7\}$, $d = 15$ (max size)

- Here root of tree represents starting point. Value of sum is on the node. Left and right children of node represent inclusion and exclusion of a_i.

- If $s == d$, we have a solution. If $s != d$, we can prune / terminate the node if
  - (a) $S + a_{i+1} > d$ (sum is too large)
  - (b) $S + \sum_{j=i+1} a_j < d$ (sum is too small)



-x-

Branch-and-Bound

- Another optimization technique, which explores the state-space tree and prunes branches that can't yield better solutions based on calculated bounds.

_/_/_

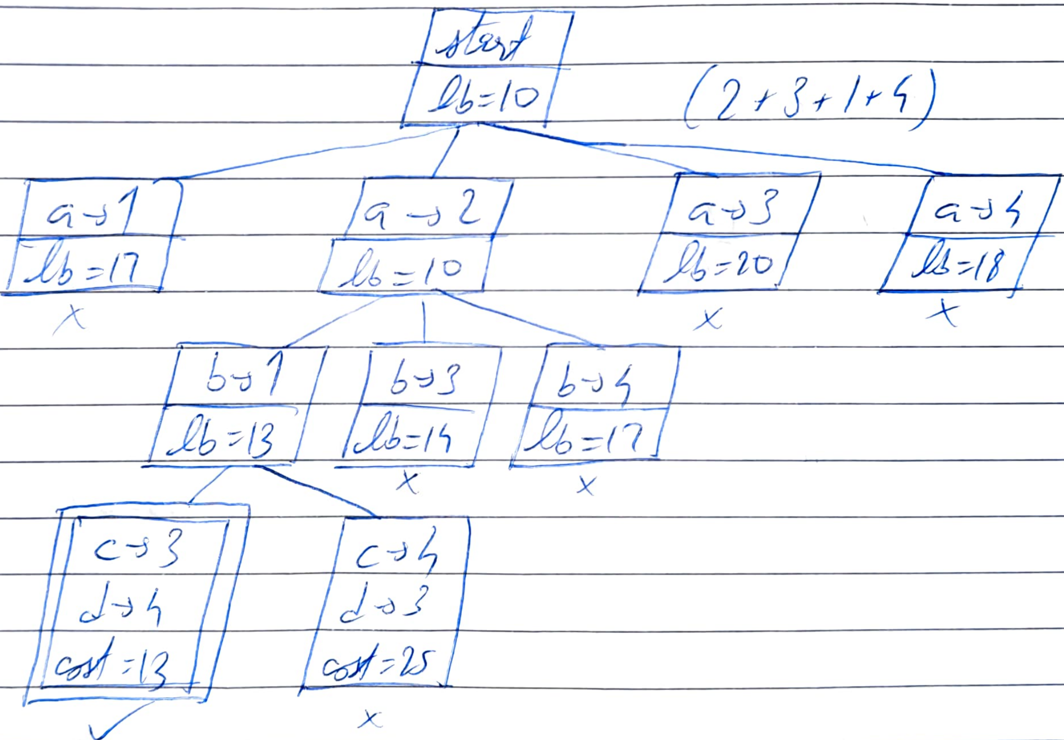
Assignment Problem O(M×N)

- Assigning n people to n jobs to obtain minimum cost.

Ex: Cost Matrix, C =

	job1	job2	job3	job4
person A	9	2	7	8
person B	6	4	3	7
person C	5	8	1	8
person D	7	6	9	4

Here lower bound lb = lowest cost possible during assignment.



- Here, lb of root = ~~lowest~~ sum of lowest value in each row
  (a→1) means assigning job1 to person A, find minimum cost from columns other than job1 for each row (repetition in each other column is allowed sometimes)

  (lb ← sum)

After obtaining the lowest lb value in each level of the tree, the rest of the nodes are pruned. Hence more efficient than backtracking.

Knapsack

- Items are ordered in descending order of by value-to-weight ratio. (v/w)

$$ub = v + (W - w)\left(\frac{v_{i+1}}{w_{i+1}}\right)$$

where $v \rightarrow$ total value of items already selected.

$(W-w) \rightarrow$ remaining capacity of knapsack

$\left(\frac{v_{i+1}}{w_{i+1}}\right) \rightarrow$ best / highest value-to-weight ratio among remaining items.

Eg: Knapsack capacity $W = 10$.

item	weight	value	value / weight
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4