L. Hopital's Rule

$$= \lim_{n \to \infty} \frac{6n}{5}$$

$$= \frac{6}{5} \times \infty = \infty$$

Stirling's formula

$$n! \simeq \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

for large value for n

2) Brute Force

Brute force is a straightforward approach to solving a problem, directly based on problem's statement and definitions of the concepts involved.

Eg: Consider exponential problem, compute a^n for a given number 'a' and +ve integer 'n'.

By definition of exponentiation,

$$a^n = \underbrace{a \times a \times a \;..\; \times a}_{n \text{ times}}$$

Brute force is used for solving algorithmic tasks such as finding sum of 'n' nos, largest element in list, sorting, searching etc.

We consider the Brute force approach to solve sorting problem. i.e to sort the given list of elements in ascending order.

2.1)

Selection Sort

we start selection sort by scanning the entire given list to find its smallest element and exchange it with first element, putting the smallest element in its final position of the sorted list.

Then we scan the list, starting from second element to find smallest among last n-1 elements and exchange it with the second element, putting second smallest element in its final position.

At i^{th} pass through the list, we search for smallest element among last $n-i$ elements and exchange it with the A_i.

$$A_0 \le A_1 \le A_2 \ldots \le A_{i-1} \mid$$

$$\underbrace{A_i \ldots\ldots A_{min}}_{\text{last } n-i \text{ elements}} \ldots A_{n-1}$$

89 46 64 90 15

15 | 46 64 90 89

15 46 | 64 90 89

15 46 64 | 89 90

15 46 64 89 90 ∴ Sorted

Algorithm: Selection Sort

// I/P: A[0 .. n-1]
// O/P: Array sorted in ascending order

for $i \leftarrow 0$ to $n-2$ do
{
  min $\leftarrow i$
  for $j \leftarrow i+1$ to $n-1$ do
  {
    if $A[j] < A[min]$
    {
      min $\leftarrow j$
    }
  swap $A[i]$ and $A[min]$
  }
}

The input's size is given by no of elements n. The algorithm's basic operation is key comparison $A[j] < A[min]$. The no of times it is executed depends on array size and is given by foll sum.

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

$$\sum_{i=l}^{u} 1 = u - l + 1$$

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

$$\sum_{j=i+1}^{n-1 \to u} 1 = (n-1) - (i+1) + 1$$

$i+1 \to l$

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

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

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

Expand the Summation

for $i=0$ for $i=1$ for $i=2$

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

for $i = n-2$

$$\dots + 1$$

Apply Summation formula $\frac{n(n+1)}{2}$

write in reverse order

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

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

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

$i=0$, $= n-1-0$
$= n-1$

$i=1$, $n-1-1$
$= n-2$

$i=2$, $n-1-2$
$= n-3$

$\vdots$

$i = n-2$, $n-1-(n-2)$
$= \cancel{n}-1-\cancel{n}+2$
$= 1$

2.2) Bubble Sort

In Bubble Sort algorithm adjacent elements are compared and if it is out of order they are exchanged. by doing it repeatedly we bubble the largest element to last position on list. The next pass bubbles the second largest element to last position

89 $\leftrightarrow$	45	68	90	29	34	17
45	89 $\leftrightarrow$	68	90	29	34	17
45	68	89 $\leftrightarrow$	90	29	34	17
45	68	89	90 $\leftrightarrow$	29	34	17
45	68	89	29	90 $\leftrightarrow$	34	17
45	68	89	29	34	90 $\leftrightarrow$	17
45	68	89	29	34	17	90

Scanned by CamScanner

for $j \leftarrow 0$ to $n-2-i$ do

eg: 50, 30, 70, 40

if $A[j+1] < A[j]$

swap $A[j]$ and $A[j+1]$

30, 50, 70, 40

30, 50, 40, 70

30, 40, 50, 70

or $i = 0$ to $n-1$

$j = 0$ to $n-i-1$

No of key comparisons = $C(n)$

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

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

$$= n-2-i+1$$

$$= n-1-i$$

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

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

Note: Simplification is same as for selection sort

Note: $n-2-i$, in Pass 2, element at index 2 & 3 are not compared since last is bigger

$n-1-i$, in Pass 2 element at 2 & 3 index will be compared.

2.3) Sequential Search and Brute-force string matching

Sequential Search

General sequential search we know, here we alter the same algorithm by adding the key element at the end of the list. We can eliminate check for the list's end on each iteration of the algorithm

Algorithm

$A[n] \leftarrow k$

$i \leftarrow 0$

while $A[i] \neq k$ do

$\quad i \leftarrow i+1$

if $i < n$ return(i)

else return (-1)

Old Algorithm

$i \leftarrow 0$

while $(i < n)$ and

$\quad A[i] \neq k$ do

$i \leftarrow i+1$

if $i < n$ return i

else return (-1)

In old algorithm for each iteration $(i < n)$ condition is checked, whereas in new Algorithm the condition $(i < n)$ is checked only when key is found.

Brute-force String matching

Algorithm: // Input Text String and pattern String

// O/p: The position of first character in the text string that starts the first matching substring if search is successful else -1

for $i \leftarrow 0$ to $n-m$ do

{ $j \leftarrow 0$

while $j < m$ and $P[j] = T[i+j]$ do

{ $j \leftarrow j+1$

} }

if $j = m$ return i

return -1

Text String: T[i] = HELLO_WORLD.

P[j] = LO

$n = 11$, $n-m = 9$
$m = 2$

① $i = 0$, $0 < 9$ $j = 0$

$j < m$ && $P[j] = T[i+j]$

$0 < 2$ && $P[0] = T[0+0]$

$L \neq H$

if $j = m$
$0 \neq 2$ so won't return i.

for i = 0 to n-m
{ y = 0
while y < m & P[j] == T[i+j]
{
do
} y = y+1
if y = m return i
}
~~else~~ return -1

② $i = 1$, $1 < 9$ $j = 0$

$0 < 2$ && $P[j] == T[i+j]$

$L \neq E$

if $j = m$, $0 \neq 2$

③ $i = 2$, $2 < 9$ $j = 0$

$0 < 2$ & $P[j] == T[i+j]$

$L = L$

$j = j+1 = 1$

~~if j = m, 1 ≠ 2~~

$j = 1$, $1 < 2$, $P[1] = T[2+1]$
$O \neq L$
while loop is terminated
$j = m$, $1 \neq 2$

④ $i = 3$, $3 < 9$ $j = 0$

$0 < 2$ && $P[j] = T[i+j]$

$L == L$

$j = j+1 \Rightarrow j = 1$

~~if j = m i.e 1 ≠ 2~~

$1 < 2$ && $P[1] = T[3+1]$

$O = O$ $\therefore j = j+1 = 2$ → while (2 < 2) terminate.

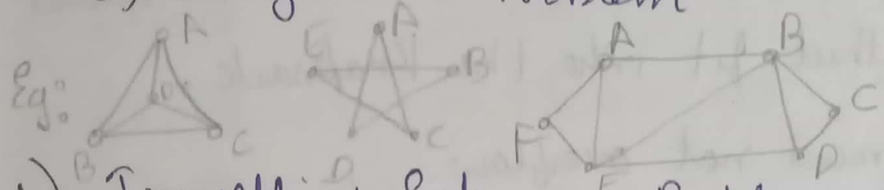
$j = m \Rightarrow$ yes $2 = 2$ return 2

2.4) Exhaustive Search

Exhaustive Search is simply a brute force approach to combinatorial problems.

1) Travelling Salesman Problem
2) Knapsack Problem
3) Assignment Problem

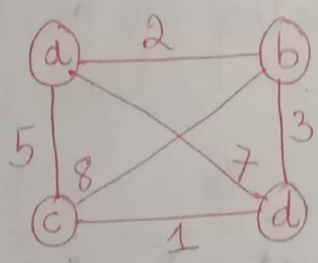
Eg:



1) Travelling Salesman Problem

The Problem asks to find the shortest tour through a given set of n cities that visits each city exactly once before reaching back the city where he started.

For this we use weighted graph, vertices represent the cities & edges represents the distances.



Tour	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 optimal
a→d→b→c→a	l = 7+3+8+5 = 23
a→d→c→b→a	l = 7+1+8+2 = 18

The no of permutations needed $= \frac{(n-1)!}{2}$

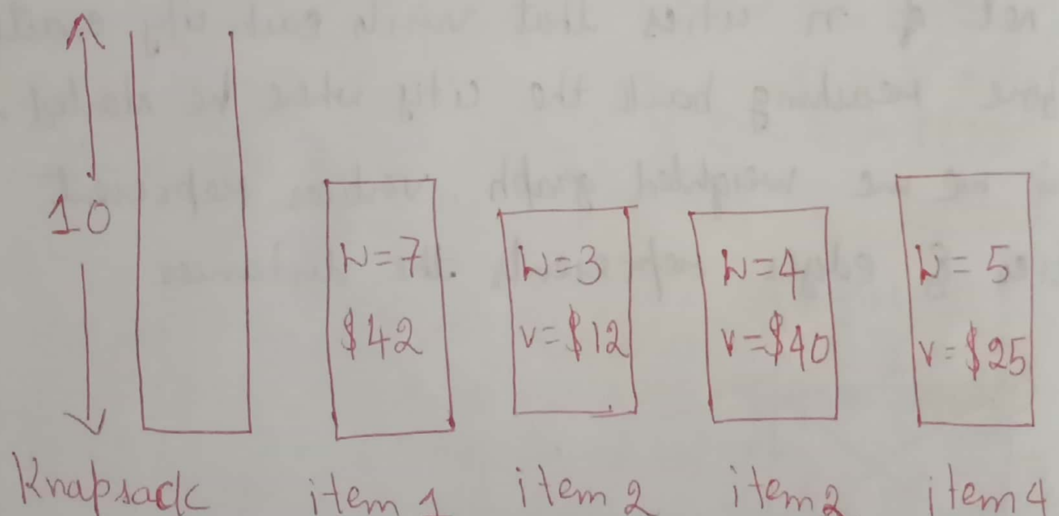
Note: $(n-1)!$ = no of hamiltonian cycle in graph, $\frac{(n-1)!}{2}$ = dist edg

2) Knapsack Problem

Given 'n' items of known weights $w_1, w_2 \ldots w_n$ and values $v_1, v_2, v_3 \ldots v_n$ and a knapsack of capacity W, find the most valuable subset of items that fit into the knapsack.

Knapsack must not overflow.

Example:



Note: for 'n' elements, there will be 2^n subsets

$\therefore n=4,\ 2^n \Rightarrow 2^4 = 16$ subsets

set of all subsets are called Power sets

We have to take all the subsets and then add the respective weights and value. If value > 10, it is not feasible.

Subset	Total weight	Total value
0	0	$0
{1}	7	42
{2}	3	12
{3}	4	40
{4}	5	25
{1,2}	10	~~36~~ 54
{1,3}	11	Not feasible
{1,4}	12	NF
{2,3}	7	52
{2,4}	8	37
{3,4}	9	65
{1,2,3}	14	NF
{1,2,4}	15	NF
{1,3,4}	16	NF
{2,3,4}	12	NF
{1,2,3,4}	19	NF

3) Assignment Problem

There are 'n' people who need to be assigned to execute 'n' jobs, one person per job. If i^{th} person is assigned j^{th} job in a known as $C[i, j]$ for each pair $i, j = 1 \ldots n$. The problem is to find an assignment with smallest total cost.

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

In terms of this matrix, the problem calls for a selection of one element in each row, so that all selected elements are in different columns and total sum of the selected element is smallest possible.

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

For this matrix, we can select any element from each row.

Example: Take 2 from first row, Then we can not take 4 from second row. because the element 2 and 4 both are in same column i.e in second column.

1) <1, 2, 3, 4> this refers we are take 9 from 1^{st} row
2 means second element from 2^{nd} row
3 means third element from 3^{rd} row
4 means fourth element from 4^{th} row

$= 9 + 4 + 1 + 4$

$= 18$

2) $\langle 1,2,4,3 \rangle = 9+4+8+9$
$= 30$

3) $\langle 1,3,2,4 \rangle = 9+3+8+4 = 24$

4) $\langle 1,3,4,2 \rangle = 9+3+8+6 = 26$

if $n=4$
$4! = 24$

5) $\langle 1,4,2,3 \rangle = 9+7+8+9 = 33$

6) $\langle 1,4,3,2 \rangle = 9+7+1+6 = 23$ etc

Note: No of permutations for general case is $n!$

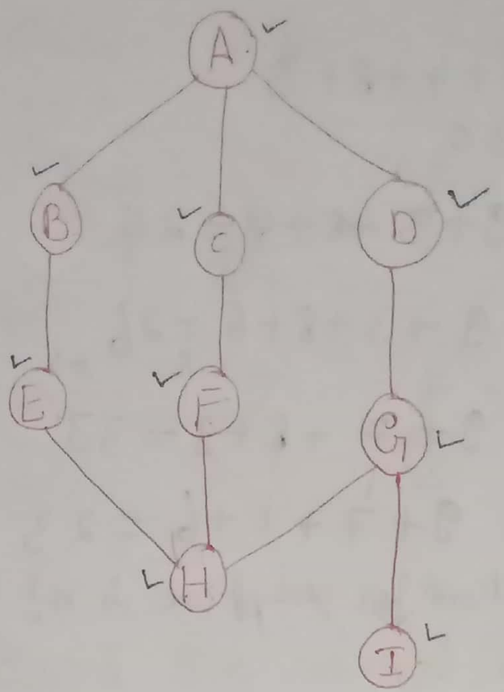
2.5) Depth-First Search and Breadth-First Search

1) DFS (Depth-first Search): DFS is algorithm used for doing traversals in graph. In DFS we select an arbitrary vertex by marking it as visited. On each iteration the algorithm proceeds to an unvisited vertex. we visit the adjacent vertices of arbitrary vertex.

we can order the vertices by alphabets, and then based on alphabetical order we can do traversal

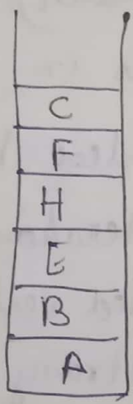
We make use of stack to trace the operation of DFS. We push a vertex into the stack when the vertex is reached for first time (i.e visit of vertex starts).

when we reach the dead end, the vertex is popped out from stack.

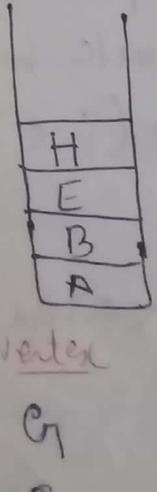


dfs(A)
dfs(B)
dfs(E)
dfs(H)
dfs(F)
dfs(C)
dfs(G)
dfs(I)

Vertex visited		Path
A initial visit	Stack	
B (Adjacent of A)		AB
E (Adjacent of B)		BE
H		EH
F		HF
C → dead end, because next adjacent vertex to C is A. 'A' is already visited		FC

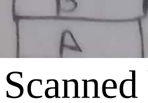


So, now we have to pop C, F from stack



Vertex	Path
G	HG
I	GI

Push, G, I to stack



Now 'I' is dead end, so pop it from the stack.
next 'D' is visited, so push 'D' into stack

Vertex	Path
D	G D

D
G
H
E
B
A

dfs(a)

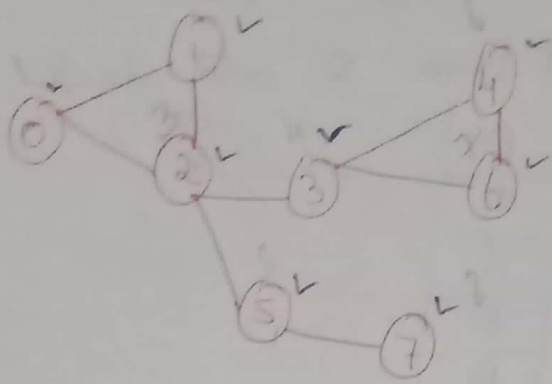
Algorithm: // Inp: G = {V, E}
DFS
// O/p: G with its vertices marked with consecutive integers.
0 means not visited

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)

2) Breadth First Search

BFS is one more method of doing traversals on graph. here the source node is selected and all the adjacent vertices to source node is visited and it is enqueued. once the adjacent vertices are dequeued.

... source node



① '0' is arbitrary vertex, taken as source node
adjacent vertices to '0' is 1, 2, enqueue 1, 2

queue: 1, 2

②

Source node	adjacent node
1	—

queue: ~~1~~, 2

③

Source Node	adjacent node
2	3, 5

queue: ~~1~~, ~~2~~, 3, 5

④

3	4, 6

queue: ~~1~~, ~~2~~, ~~3~~, 5, 4, 6

⑤

5	7

queue: ~~1~~, ~~2~~, ~~3~~, ~~5~~, 4, 6, 7

Result: 0, 1, 2, 3, 5, 4, 6, 7

Algorithm: BFS

count ← 0

for each vertex v in V do

if v is marked with 0

bfs(v)

→ bfs(v)

count = count + 1; mark v with count & initialize a queue with v

while the queue is not empty do

for each vertex w in V adjacent to front vertex v do

if w is marked as 0

count ← count + 1;

add w to the queue

remove vertex v from front of queue

END OF UNIT 1

~~0, 1, 2, 3~~, 5, 4, 6, 7

2.1) Decrease and Conquer

The decrease and conquer technique is based on exploiting the relationship between a solution to a given instance of problem and solution to a smaller instance of same problem.

There are 3 types : 1) decrease by constant
2) decrease by a constant factor
3) Variable size decrease

1) **Decrease by constant**: In this method, the size of an instance is reduced by the same constant on each iteration of the algorithm.

Example: exponentiation problem to compute a^n for positive integer.

The solution for size n and solution for size n-1 is obtained by formula $a^n = a^{n-1} . a$

$f(n) = a^n$

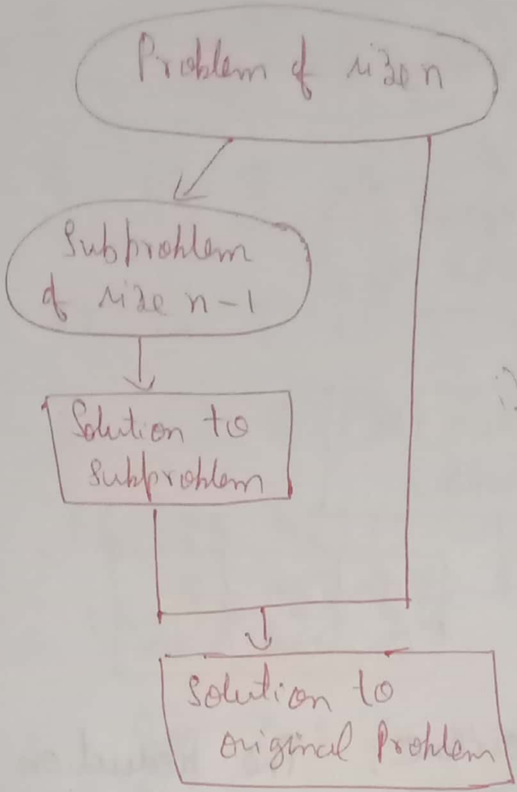
$$f(n) = \begin{cases} f(n-1) . a & \text{if } n > 1 \\ a & \text{if } n = 1 \end{cases}$$

$n = 2$

$a^2 = a^{2-1} . a$
$= a . a$

$n = 1$

$a^1 = a^{1-1} . a$
$= a^0 a$
$a^1 = a$



2) <u>Decrease by constant factor</u>: It suggests reducing the problem's instance by same constant factor on each iteration of algorithm.

<u>Example</u>: Take same exponentiation problem

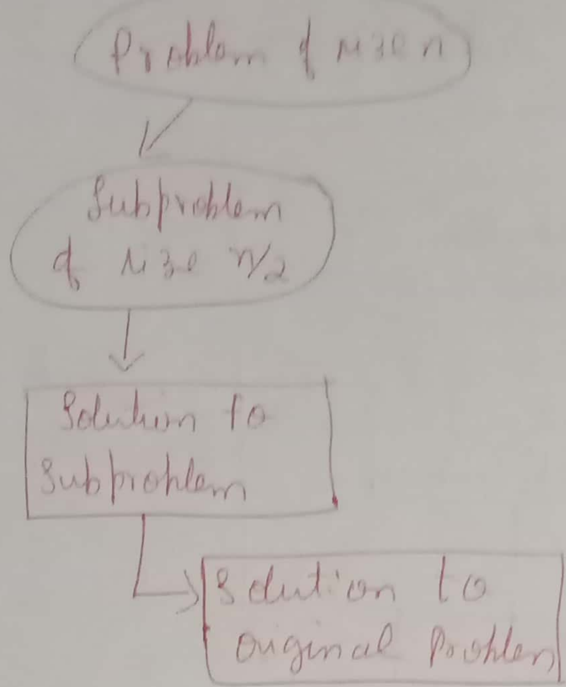
solution for size n is to compute a^n

The instance of half its size is to compute $a^{n/2}$

a^n solution works only for ~~odd~~ even values of 'n'.

if n is odd value, compute a^{n-1}

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd} \quad \text{Eg: } a^5 = a^{5-1} \cdot a = a^4 \cdot a \\ a & \text{if } n = 1 \end{cases}$$



3) <u>Variable-size-decrease</u>: The reduction size varies from one iteration to another iteration.

<u>Eg</u>: gcd algorithm using Euclid's method

while (n≠0) {r = m%n, m=n, n=r}

2.1.1) <u>Insertion Sort</u>: It is a sorting algorithm used to sort the array elements in an order. how, this algorithm works means, The entire array is divided as sorted array and unsorted array. The first element of ~~elem~~ array is considered as sorted, then the remaining all elements except first element is put into unsorted portion. Second element is compared with first element and then sorted portion has 2 elements. like this it continues.

Decrease by one technique

23 | 42 4 16 8 15
sorted unsorted

23 42 | 4 16 8 15

4 23 42 | 16 8 15

4 16 23 42 | 8 15

4 8 16 23 42 | 15

4 8 15 16 23 42

Algorithm:

```
for i=1 to n-1
{
    element = arr[i]
    j = i-1
    while (j >= 0 & arr[j] > element)
    {
        arr[j+1] = arr[j]
        j = j-1
    }
    arr[j+1] = element
}
```

Number of comparisons in worst case = $C^{(n)}_{worst}$

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

$$= \sum_{i=1}^{n-1} i \quad \text{Apply: } \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

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

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

$$\sum_{i=l}^{u} 1 = u - l + 1$$

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

$$\cancel{= 0 - (i-1) + 1}$$

$$\cancel{= -0 - i + 1}$$

$$= i - 1 - 0 + 1$$

$$= i$$

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 \rightarrow \sum_{i=l}^{u} 1 = u - l + 1$$

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

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

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

no of comparisons in avg case = half of the comparisons of worst case

$$\therefore C_{avg}(n) = \frac{\frac{n(n-1)}{2}}{2}$$

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

$$C_{avg}(n) = \frac{n(n-1)}{4} \approx \frac{n^2}{4}$$

2.1.2) Topological Sorting:

Topological Sort is used for digraphs. Topological sort of graph $G = \{V, E\}$ is linear ordering of all vertices such that an edge (u, v) in G appears such that u appears before v.

Topological sort is nothing but the ordering of its vertices along a horizontal line.

There are two possible solutions for this

1) DFS method
2) Source Removal method

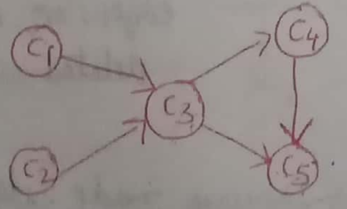
1) **Problem:** A student has to take courses $\{C_1, C_2, C_3, C_4, C_5\}$ in some degree program.

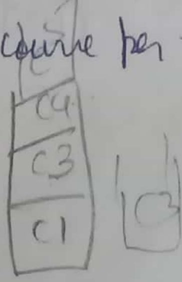
Courses can be taken in any order as long as the following prerequisites are met.

1) C_1 and C_2 has no prerequisites
2) C_3 requires C_1 and C_2
3) C_4 requires C_3
4) C_5 requires C_3 and C_4

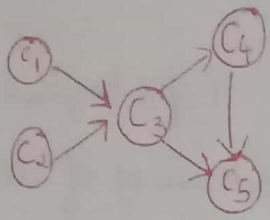
The student can take only one course per term

C5
C4
C3
C1
C2





Digraph representing the prerequisite structure of 5 courses



DFS Traversal: Nodes visited

Push the stack

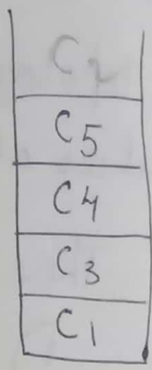
C_1 (Root)

C_3 (Adj to C_1)

C_4 (Adj to C_3)

C_5 (Adj to C_4)
Dead End

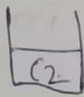
C_5 adjacent C_4 is already visited



Now pop elements from stack. we can see the pop order :

C_5 , C_4 , C_3 , C_1

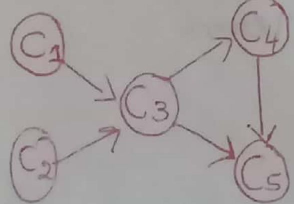
Push C_2 into stack



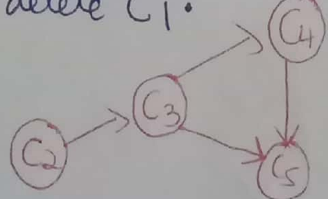
Adjacent to C_2 is C_3 already visited. so pop C_2

∴ Topologically sorted list is C_2 $C_1 \to C_3 \to C_4 \to C_5$

2) Source Removal Method

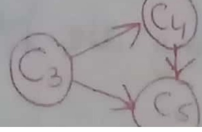


① Take C_1 as source node delete C_1.



C_1 has no incoming edges. so it is deleted

② Take C_2 as source node, delete C_2



(3) Take C_3 as source node, delete it

(4) Take C_4 as source node, as it has no incoming edges. delete it also.

(5) Delete C_5

<u>Solution obtained</u> : C_1, C_2, C_3, C_4, C_5

2.1.3) <u>Decrease by a Constant-factor Algorithms</u>

Here Consider Binary Search algorithm, because in binary search for each iteration the array size is reduced to half. Binary Search is remarkably efficient algorithm for searching in a sorted array.

This algorithm works by comparing a search key 'K' with the array's middle element $A[m]$. If they match, the algorithm stops.

otherwise, if $K < A[m]$ the same operation is repeatedly done in first half of the array.

if $K > A[m]$ then search is done in second half of the array.

K

$$\underbrace{A[0] \ldots A[m-1]}_{\substack{\text{search here if} \\ K < A[m]}} \quad A[m] \quad \underbrace{A[m+1] \ldots A[n-1]}_{\substack{\text{search here if} \\ K > A[m]}}$$

As an example, let us apply binary search to search for $K = 70$ in the below array.

3	14	27	31	39	42	55	70	74	81	85	93	98

The iterations are given in the following table:

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	l						m						h
iteration 2								l		m			h
iteration 3								l	m	h			
iteration 4								l	h				

$\left(\frac{0+12}{2}\right) = 6$ 70 > 55 key > mid low = mid + 1 = 7

mid $= \frac{l+h}{2} = \frac{7+12}{2} = \frac{19}{2} = 9.5 = 9$

i.e (70 < 81) high = mid − 1 high = 8

$m = \frac{7+8}{2} = \frac{15}{2} = 7$

Now key = mid

70 = 70 stops.

<u>Algorithm</u>: // Input: Search key in sorted array

// Output: Index of the element which is = key or else −1

$l \leftarrow 0;\ h \leftarrow n-1$

while $l \leq h$ do

$m \leftarrow \lfloor (l+h)/2 \rfloor$

if $K == A[m]$ return m

else if $K < A[m]$

$h \leftarrow m-1$

else $l \leftarrow m+1$

return −1

The standard way to analyze the efficiency of binary search is to count the number of times the search key is compared with an array element.

We will count the so-called 3-way comparisons i.e $K = A[m]$, $K \le A[m]$ and $K \ge A[m]$

How many such comparisons does the algorithm make on a array of n elements. It depends on 'n' and instance of problem.

Let us compute for worst case. Worst case includes all arrays that do not have key element, as well as some successful search.

$$C_w(n) = C_w(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1 \longrightarrow (1)$$

$$C_w(1) = 1$$

already proved in binary previous

Std approach

for $n = 2^k$, $C_w(2^k) = k+1$

$C_w(2^k) = \log_2 n + 1$

$C_w(2^k) = C_w\left(\frac{2^k}{2}\right) + 1$

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

$\therefore C_w(n) = \lfloor \log_2 n \rfloor + 1 \Longrightarrow C_w(n) = \lfloor \log_2 n \rfloor + \log_2 2$

$C_w(n) = \lceil \log_2 (n+1) \rceil$

$C_w(n) = \log_2 (n+2)$

Note: $\lceil x \rceil \le x$

$C_{avg}(n) \approx \log_2 n$ i.e for Average case efficiency the no of key comparisons is smaller than worst case

More accurate formula for successful & failure search in average case

$C_{avg}^{yes} \approx \log_2 (n-1)$, $C_{avg}^{no} \approx \log_2 (n+1)$

Scanned by CamScanner

2.1.4) Interpolation Search

This is an example for variable - size decrease algorithm. Interpolation search is an algorithm for searching key in a sorted array.

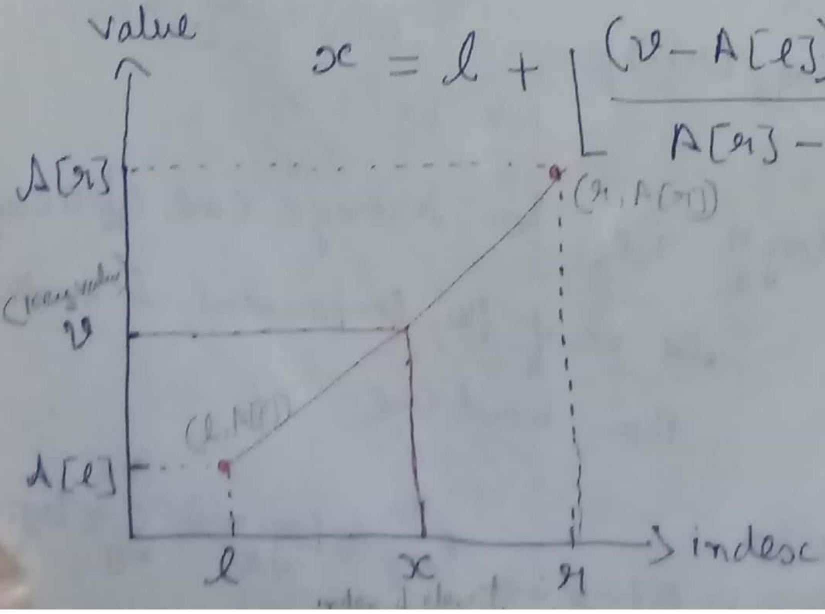
The Concept in Interpolation Search is, it concentrates on the value of the search key to find the element in array which has to be compared with key.

It mimics the way we search for a name in the phonebook.

If A is an array, the leftmost element of A is A[l] and Rightmost element is A[r]. The algorithm assumes that the array's value increase linearly. Linearly means along the straight line through the points (l, A[l]) and (r, A[r]).

The array's element index which has to be compared with key value 'v' is calculated by formula

$$x = l + \left\lfloor \frac{(v - A[l])(r - l)}{A[r] - A[l]} \right\rfloor$$



Example:

10	20	30	40
0	1	2	3
l			r

$l = 0 \quad v = 40$

$r = 3$

$A[0] = 10$

$A[3] = 40$

$$x = l + \frac{(v - A[l])(r-l)}{A[r] - A[l]}$$

$$x = 0 + \frac{(40 - A[0])(3-0)}{A[3] - A[0]}$$

$$= 0 + \frac{(40-10)(3)}{40-10}$$

$$x = \frac{30 \times 3}{\cancel{30}}$$

$$\boxed{x = 3}$$

Compare the element in index 3 in A

$$A[3] = v$$

The interpolation search uses less than $\log_2^n + 1$ comparisons. The function grows so slowly that the number of comparisons will be a very small constant for all feasible inputs

We can conclude that binary search is suitable for small files, but for large files Interpolation search is useful.