Question 1

Question 1.1 : Answer

→ To access any random element the following formula can be used.

Address of Array [i] = B + S * (i - L.B)

Where :

i = index of an element we want to access

B = address of the first element of arrays also called/refers as base address of the array

S = size of stored data type in an array

L.B = lower bound of subscript

Given base address of an array;

Question => A[1300 ..... 1900] is 1022 and the size of each element is 2 bytes in the memory. Find the address of an array of A[1704].

Given :

B = 1022

S = 2 Byte

i = 1704

LB = 1300

Address of Array = B + S * (i - L.B)

Address of A[1704] = 1022 + 2 * (1704 - 1300)

Address of A[1704] = 1,830

Question 1.2

Solution : The following steps can be taken to traverse the whole array and finds the largest number among them.

steps

① Initialize largest Number as a variable to set the first element of the array as the largest number.

② Go throug each element of the array.

③ Compare each element ;

if current element > largest Number then we set large_Number to be the current element / as the new large_Number

④ After we have gone throug the whole array, then we return the largest_Number.

Question-2

Push Operation

Q.2-1 1, Before inserting any data / element, we need to check the status of the stack, wheater the stack is full.

2, When pushing an element into the stack and the stack is full, then the overflow Condition occurs.

→ ex: display an error message and exit.

3, If the stack is not full, then we set the top value as -1 to check that the stack is empty.

④ When adding/pushing a new data/element to the stack, the value of the top gets incremented.

⑤ We can keep pushing/adding until we reach the maximum size of the stack.

Pop Operation

① Before poping any data/element, we need to check if the stack is empty.

② If the stack is empty, we will not be able to delet any element, then the Underflow Condition occurs. (display error message and exit)

③ If the stack is not empty, we can access the element which is at the top of the stack and pop/remove the element from the stack

④ Each time this operation is preformed, the top of the stack is being decremented by 1, ⇒ top=top-1

Question 2-2 (look at the attached file) Code Solution →

Question 3 : Searching Algorithms

Solution 3.1 :

Algorithm

① Locate key x in an array of given size n in non-decreasing order.

② Compare the target value x with the middle element. using / calculating as follow (low + high) / 2 - if x and middle element are equal, then we are done and we can quit, else

③ we divide the array into two sub-arrays
* If x is smaller than middle element, we select the left sub-array
* If x is larger than middle element, we select the right sub-array.

④ Conquer the sub-array: Is x in the sub-array using recursion until the sub-array is sufficiently small.

⑤ finally the solution to the array is found in the subarray. (either in the left or right sub-array).

Question 3.1

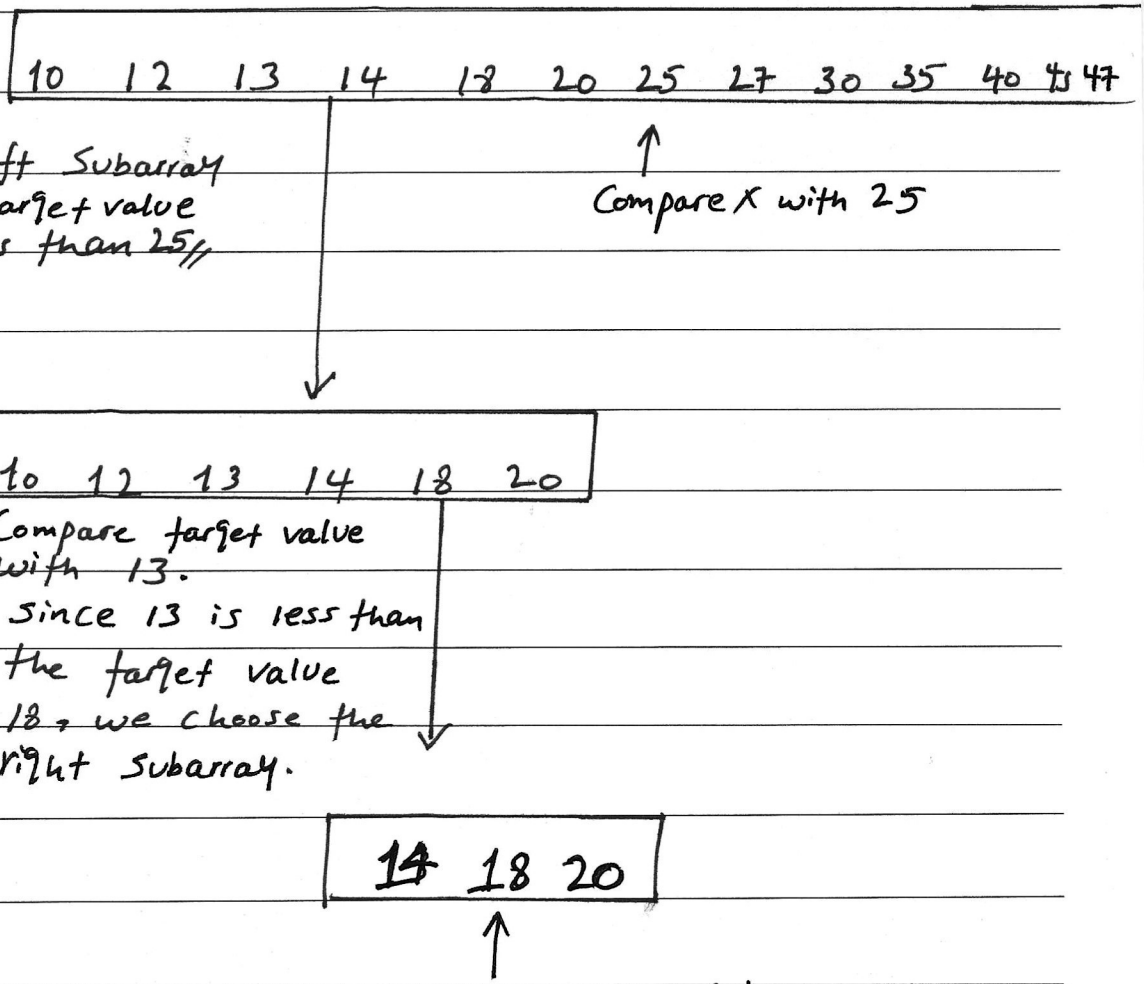
Solution for Binary search with a Recursive version is attached as a file with the name (Question_3_1_Solution)

Step-by-step human-represented Solution

Given

Suppose X = 18 and we are given the following array 10, 12, 13, 14, 18, 20, 25, 27, 30, 35, 40, 45, 47.

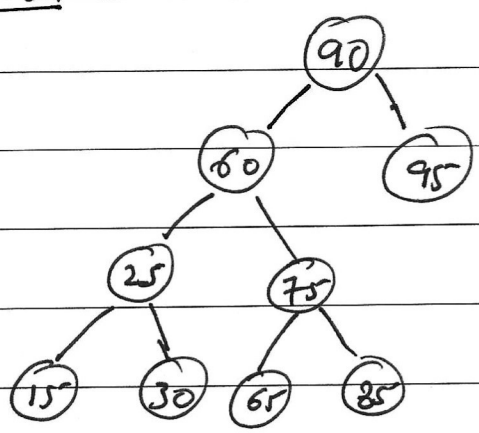
Solution :



Question 3, 2

Solution

Given



Question: Delete node 60.

Solution → First lets dive in to the Algorithms

NB! when deleting a node, three possibilites arise:

(1) If the node to be deleted has not child, simply remove it /delete it.

(2) If the node to be deleted has one child, replace ~~the child~~ it with its child.

(3) If the node to be deleted has two Children; Find in-order successor of the node (the smallest node in it's right subtree). Copy Contents of the in-order successor to the node and remove the in-order successor.

Setep - by - step - algorithms and human representation;

(1) we start with root node 90 and we then Compare its value with the node we would like to remove, in our case we would like to ~~Compare~~ delete 60.

(2) In step one we see that 60 is less than 90, therfore we move to the left Child, in this case is 60.

(3) Now we Compare the value we want to delete with the left child node. ➡ Now we see that both has value of 60.

(4) Since the value of the node we want to delete and the value of the left child node are the same, we can say that we have found the node we would like to delete.

(5) In this step we see that the nod we would like to delete has Children node 25 and node 75.

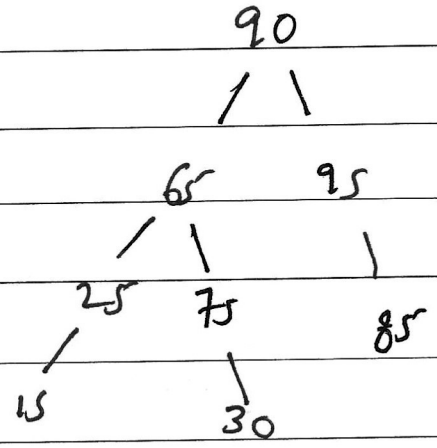
(6) At this step we will ~~be~~ find the in-order successor of the node we would like to delete, in this case the smallest value in the right Subtree.

(7) The ~~node~~ in-order successor of the node we want to delete is 65.

(8) Now we will Copy the Content of 65 to 60. In another word we will replace node 60 with the in-order

(9) Then we will delete 65 (right subtree)

(10) The result would be :



Question 4: Sorting Algorithms

Given: Array = {70, 50, 30, 10, 20, 40, 60}

Question 4.1: Solution

Algorithm - Merge Sort

⇒ Merge Sort also known as "divide and Conquer" algorithm is used to sort arrays.

⇒ To solve the given array the following algorithm can be used.

(1) Divide the array into 2-sub arrays each with n/2 items. (left and right).

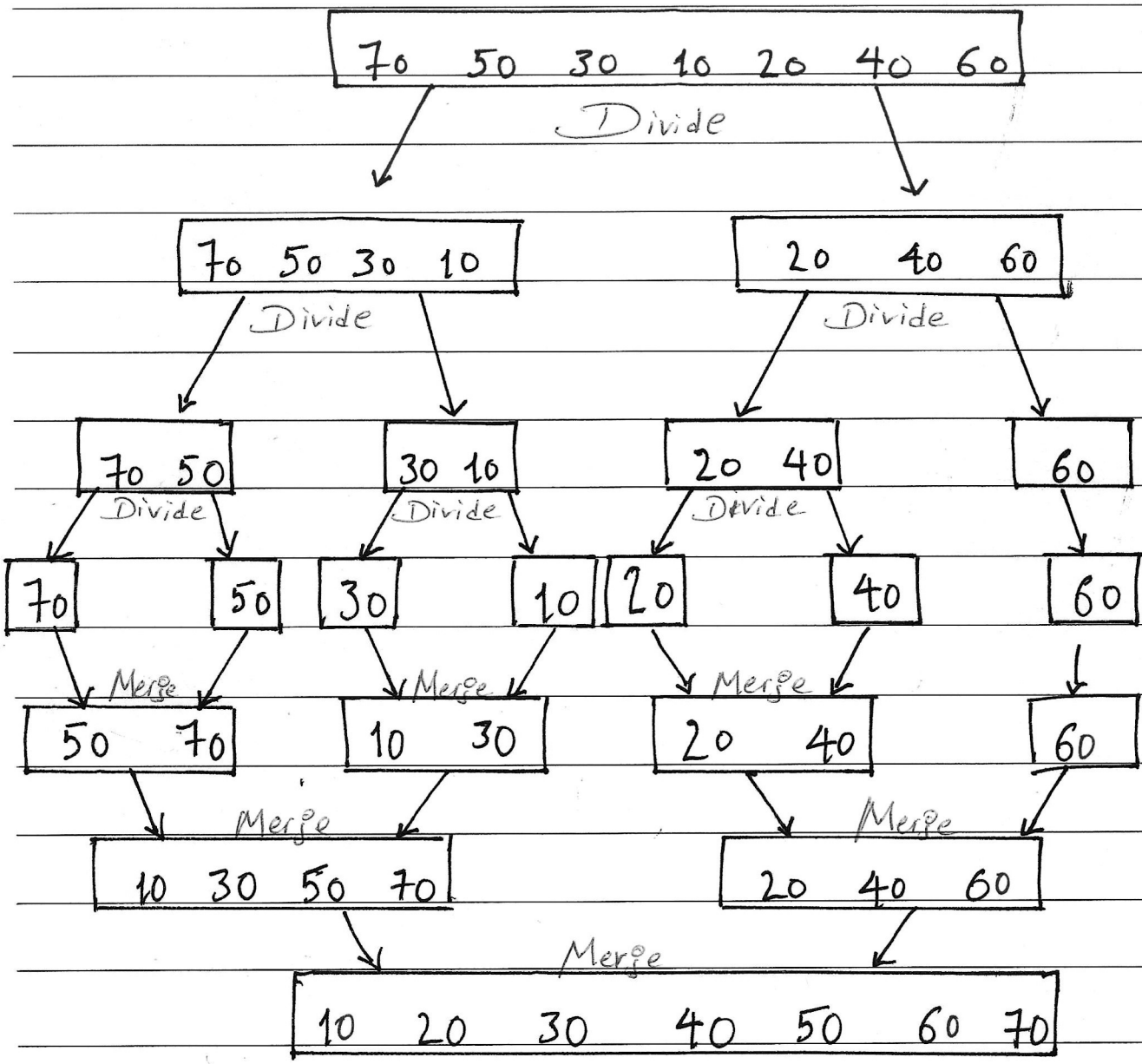
(2) Conquer (solve) recursively sort each sub-array the left and right half.

(3) Merge / combine the two sorted halves into a single array. This array is sorted as well.

Question 4.1 : Solution

Graphical representation of given array ; using Merge-Sort.

Array = { 70, 50, 30, 10, 20, 40, 60 }



Question 4.2: Solution

Quick-Sort Algorithm:

⇒ Quicksort is a divide-and-Conquer method for Sorting. It is Complementary to mergesort: for merge sort we divide/brake the array into two subarrays to be sorted and merge the ~~array~~ Sub-arrays to make a whole ordered array: for quicksort we rearrange the array in a way that, when the sub-arrays are sorted, then the hole array is ordered. The algorithm looks as follow;

(1) Array recursively brake into two partitions and recursively sorted

(2) The braking of the array happens based on a pivot item. The pivot item can be any item, but typically Chosen to be the first item.

(3) pivot item breaks or divides the array into two sub-array.

(4) All items less than the pivot will be placed in sub-array before the pivot.

(5) All items greater than the pivot will be placed in sub-array after the pivot.

(6) Recursively apply steps 2, 3, 4 and 5 to the left and right sub-arrays of the pivot until the whole array is sorted.

Question 4.2 : Solution Continues...

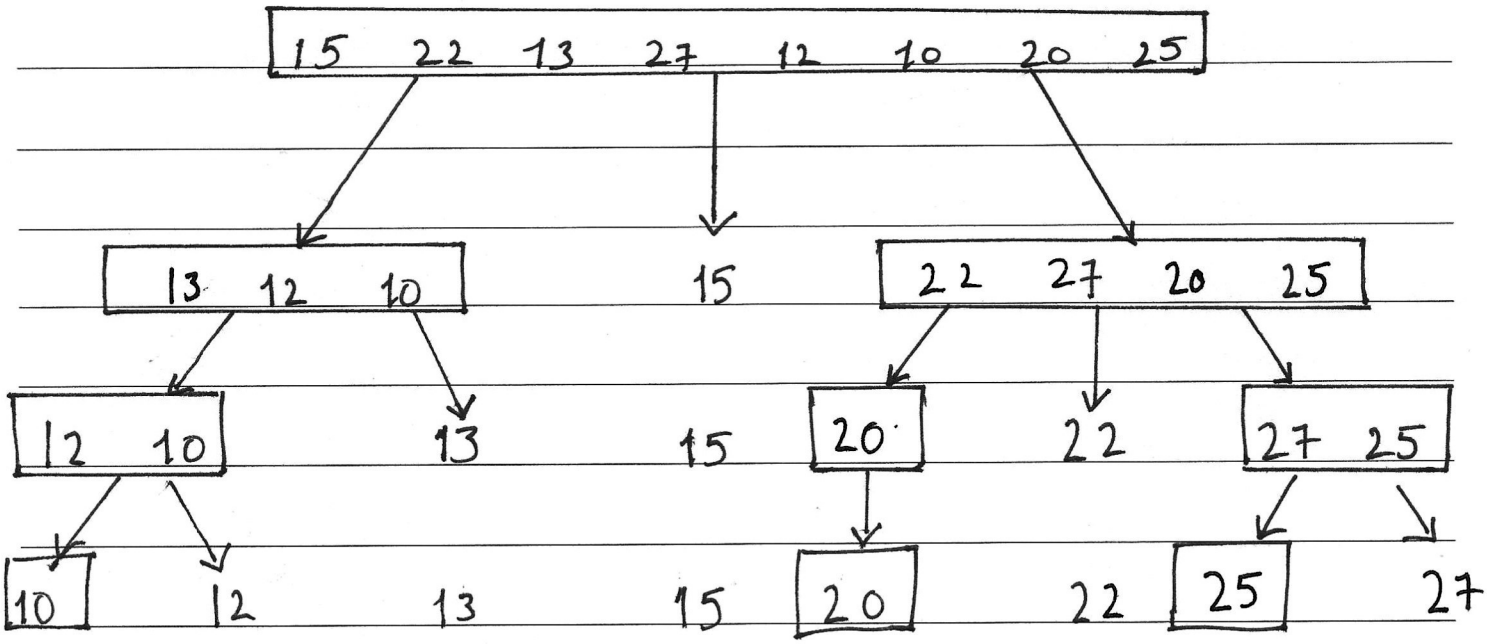
⇒ Graphical representation of given array using quick sort.

⇒ Given array;

Array = { 15, 22, 13, 27, 12, 10, 20, 25 }

Solution :

⇒ I will select my pivot = 15



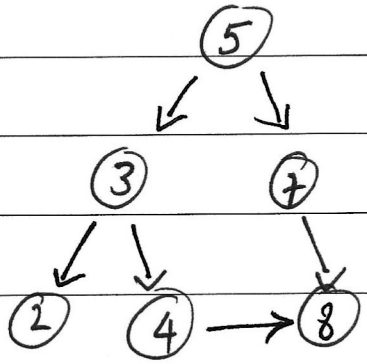
Question 5

Solution 5.1 (look at the attached file) Code

Question 5.2 : Solution

The algorithm to solve the given graph is as follow:

Given Graph:



Algorithm

(1) Start by putting any one of the graph's vertics on top of a stack.

(2) Take the top item of the stack and add it to the visited list.

(3) Create a list of that vertex's ~~edges~~ adjecent nodes. Add the ones which aren't in the visited list to the top of the stack.

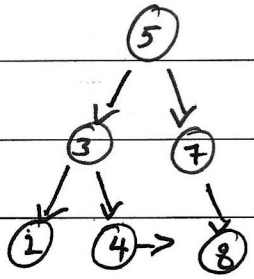
(4) Keep ~~a~~ applying steps 2 and 3 until the stack is empty

Source from lecture 4

Question 5.2 : Solution Continues.

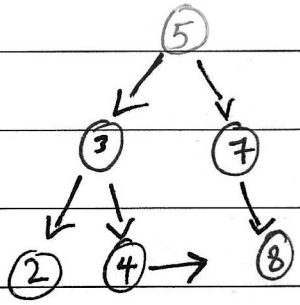
Illustration / human representation

Initially both stack & visited are empty.



Visited

Stack

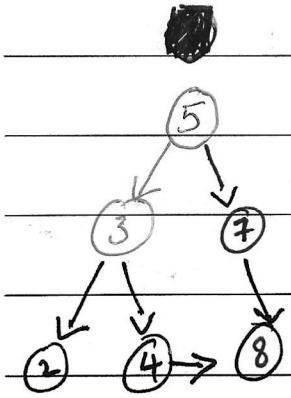


5 | | | | | |

Visited

3 | 7 | | | | |

stack

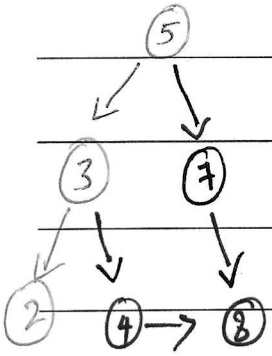


5 | 3 | | | | |

Visited

2 | 4 | 7 | | | |

stack

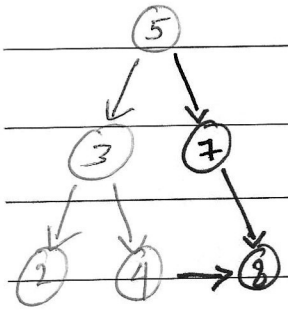


5 | 3 | 2 | | | |

Visited

4 | 7 | | | | |

stack

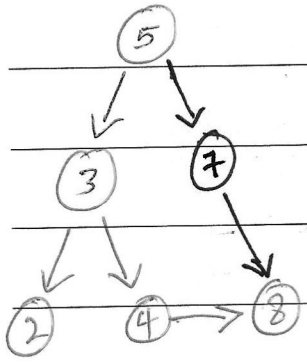


5	3	2	4		

Visited

8	7				

stack

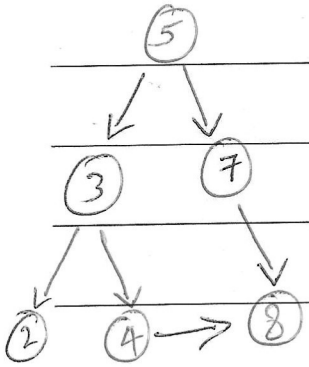


5	3	2	4	8	

Visited

7					

Stack



5	3	2	4	8	7

visited

stack

↑

Now the stack becomes empty. this means that we have visited all the nodes and DFS traversal ends.

Output will be: 5 3 2 4 8 7

Question 6: Solution

Solution 6.1

Complexity class: There are some unsolved problems in computer science; with out any solution yet. These problems are divided into Complexity Classes.

Why is it known as complexity classes in data structure?

Answer:

The Complexity class in data structure helps scientists to catagorize problems based on how much time and space they repuire to Solve a problem and verify the solution The Complexity classes are also used to analized how efficient an algorithm is for preforming different operations on the data structure.

Solution 6.2

Answer: Because it describes/analizes a lower bound on the worst case, and provides also a way to express growth of running time for an algorithm or usage of memory as the size of the input increases.

Solution 6.3

What is P Class: P class stands for Polynomial Time.

⇒ Here we have a set of decision problems that can be solved by a deterministic machine in polynomail time. The answer to the problems are a "yes" or "no" answers.

Features:

① problems in P are solvable and the solutions are easy to find.

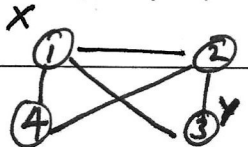
② problems in P class are tractable as well solvable

③ Tractable - the problems can be solve both in theory and practice.

⇒ problems that can be solved in theory but not in practice are called intractable.

Real-life example:

The shortest xy-path in a given graph 2, with vertices X, Y. and we have to find the shortest path from x to y.

Instance. 

Solution 1-3

Solution 6.4

NP Class: Stands for - Non - Deterministic polynomail time. It is a set of decision problems that can be solved by a non - deterministic machine in polynomail time.

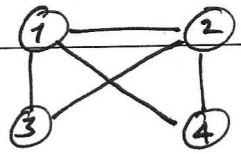
Features:

⇒ Solutions are hard to find but easy to verify.

⇒ To verify problems of NP in polynomail time, we need to use f.example a TM or Turing Test.

Real - life examples:

To mention one; The Hamiltonian path problem. Where we have a input of graph X? and we have to find a simple path that visits every vertex.

Instance 

Solution: 1 - 3 - 2 - 4

Solution 6.5

NP - Complete Class : NP - Complet classes refers to problems that are hard both in NP and NP-hard.

Features :

→ problems in NP-complete can be transformed to other NP-complete Problems in polynomial time.

→ If we can solve an NP-Complete problem in polynomial time, then we can also slove any NP problems in polynomial time.

Real-life-example:

To mention : Vertex Cover, where we have a given graph and a integer C, and we need to find a set of C vertices such that each edge of the graph is incident to at least one vertex of the set.