Programming Foundation With Pseudocode

Lesson 5: Algorithm Design Techniques

Lesson Objectives

- **To enhance the logic building by designing algorithms efficiently for the given problem.**
- **To understand and compare different Sorting methods under different design techniques:**
  - Bubble Sort
  - Insertion Sort
  - Merge Sort
- **To understand and compare different algorithms designed for a given problem under different design techniques:**

Algorithm Design Technique

- **There are different techniques to design an Algorithm. They are**
  - Brute Force
  - Divide and Conquer
  - Decrease and Conquer
  - Transform and Conquer
  - Space and Time Tradeoffs
  - Dynamic Programming
  - Greedy Technique
  - Backtracking
  - Branch and Bound

Brute Force – Bubble Sort

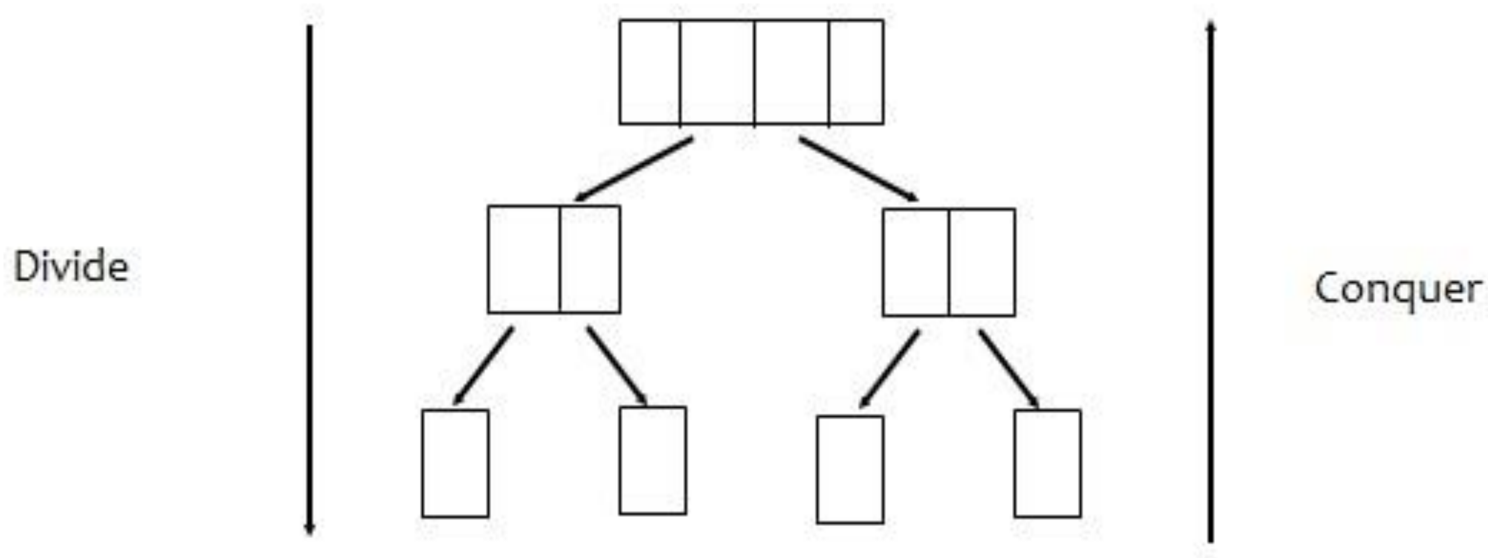
- **Brute Force: Just do it!!**
- **Easiest solution is found for a problem without any concern of the efficiency parameter.**
- **Ex: Bubble Sort and Selection Sort.**
- **Logic for Bubble Sort Algorithm**
  - Compare adjacent elements (n) and (n+1), starting with n=1.
    - If the first is greater than the second, swap them
  - Repeat this for each pair of adjacent elements, starting with the “first two elements”, and ending with the “last two elements”
    - At any point, the last element should be the largest
  - Repeat the steps for all elements except the last one
  - Keep repeating for one fewer element each time, until you have no more pairs to compare

Bubble Sort Algorithm

- **Write the Pseudo Code for the above logic**
- **Exchange your code with another participant**
  - Do a peer review of the pseudo code, and report defects that are found
- **How many passes, how many comparisons does this process involve for n=10?**
  - The number of passes are always n-1
- **If the data were mostly sorted, how can we do it faster?**
  - If there are no swaps in a particular iteration of the INNER loop, we can stop
- **Write a separate function SWAP to improve readability of the above code**
- **Efficiency is $O(n^2)$**

Divide and Conquer

- **Divide and Conquer:**
  - A problem is divided into several subproblems of the same type, ideally of about equal size.
  - The subproblems are solved.
  - If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

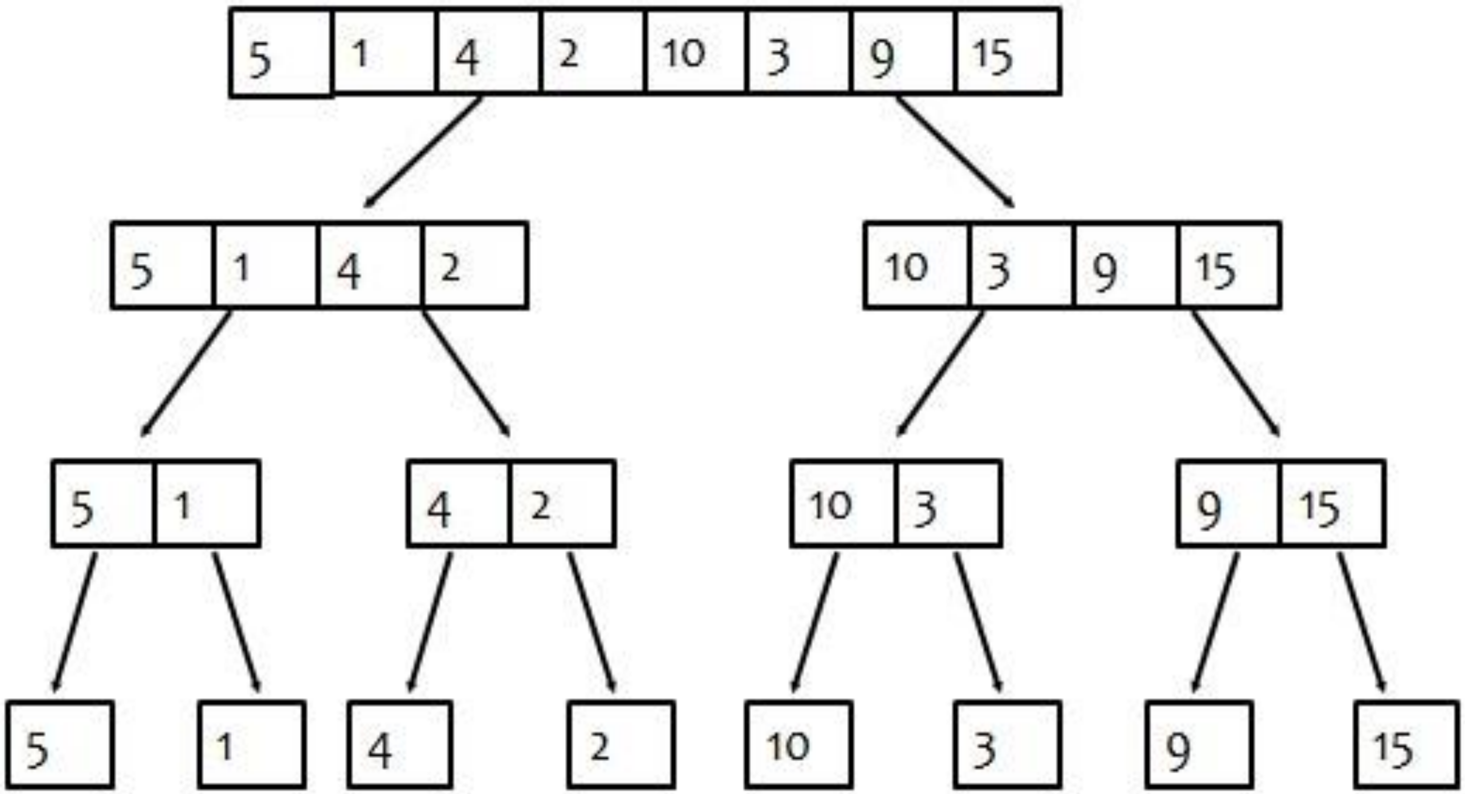


Divide and conquer Approach

- **Ex: Merge Sort, Quick Sort etc.**

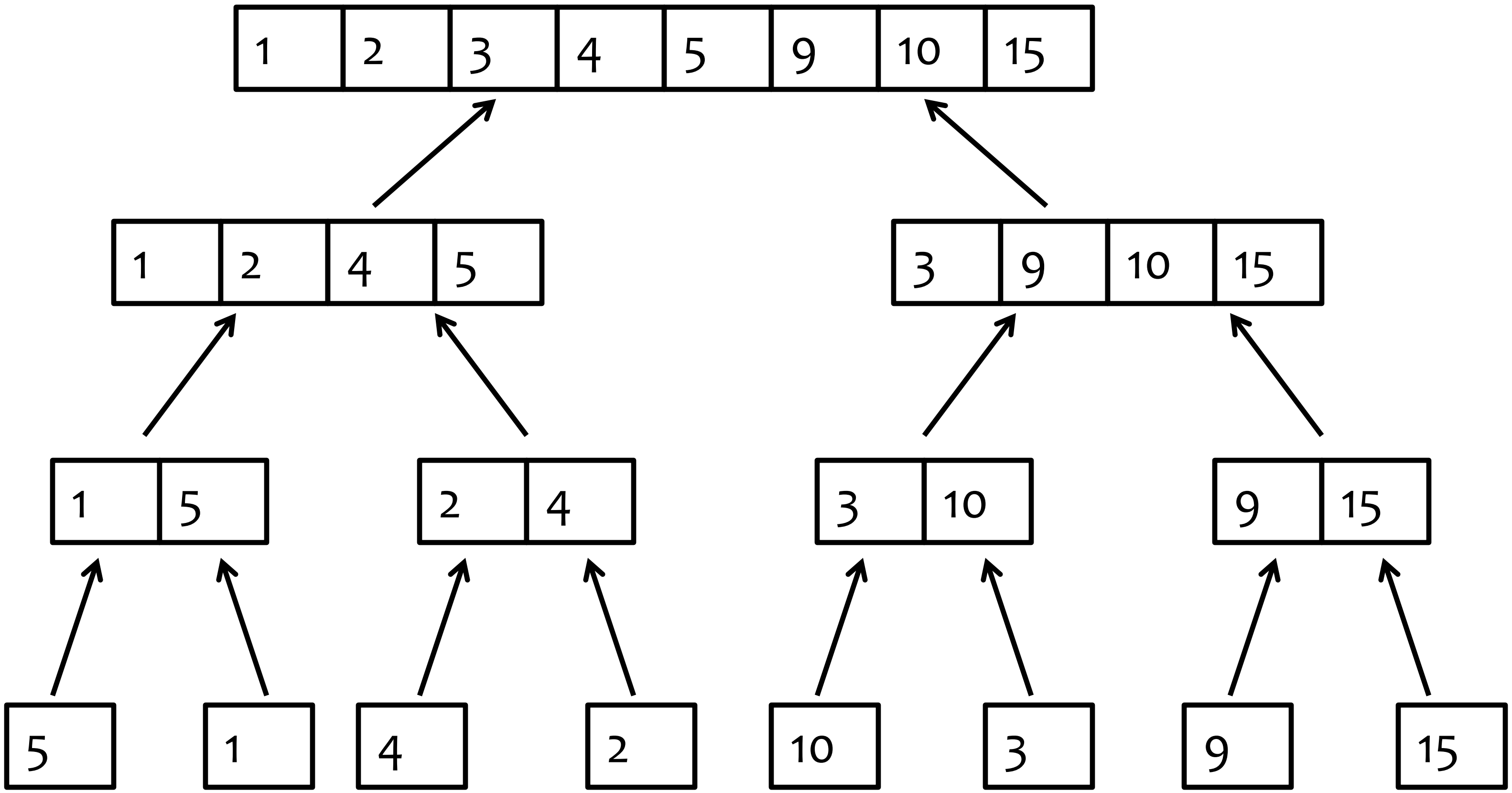
Divide and Conquer – Merge Sort

- **Merge sort sorts a given array *A*[0..*n* – 1] by dividing it into two halves *A*[0..*n*/2 – 1] and *A*[*n*/2..*n* – 1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.**



ADT – Divide and Conquer

- Below figure shows conquering steps. And Final list is sorted.



Decrease and Conquer – Insertion Sort

- **Decrease and Conquer:**
  - It is based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem.
- **Ex: Insertion Sort, Topological Sorting etc.**
- **Insertion Sort**
  - Implemented by inserting a particular element at the appropriate position
  - While inserting the element we need to find the position to insert the element
  - All other elements will be shifted one location on right to make place for new element and then the element will be inserted at the position
  - This is normally done in place (by using single array)

Insertion Sort - Example

- **Example: Consider the following array**
- **5 7 0 3 4 2 6 1**
- **On the left side the sorted part of the sequence is shown as underline. For each iteration, the number of positions the inserted element has moved is shown in brackets**
- **5 7 0 3 4 2 6 1 (0) – only a[0] is in sorted part**
- **5 7 0 3 4 2 6 1 (0) – array is sorted till a[1]**

Insertion Sort - Example

- 0 5 7 3 4 2 6 1 (2) - 0 will be inserted at a[0] location
- 0 3 5 7 4 2 6 1 (2) - 3 will be inserted at a[1] position
- 0 3 4 5 7 2 6 1 (2) - 4 will be inserted at a[2] position
- 0 2 3 4 5 7 6 1 (4) - 2 will be inserted at a[1] position
- 0 2 3 4 5 6 7 1 (1) - 6 will get inserted at a[5] position
- 0 1 2 3 4 5 6 7 (5) - 1 will be inserted at a[1] position

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Insertion Sort - Features

- **Less efficient on large lists than more advanced algorithms such as quick sort, heap sort, or merge sort**
- **Advantages**
  - simple implementation
  - efficient for (quite) small data sets
  - efficient for data sets that are already substantially sorted: the time complexity is O (n + d), where d is the number of inversions
- **Efficiency is $O(n^2)$.**

Transform and Conquer – Pre-Sorting

- **Two stages in Transform and Conquer.**
  - Problem's instance is modified and then conquered.
- **Example: Pre-sorting, AVL trees, 2-3 trees etc.**
- **Presorting:**
  - Many questions about a list are easier to answer if the list is sorted.
  - For Ex: Checking element uniqueness in an array.
  - Brute Force method for this problem is $O(n^2)$.
  - In Transform and Conquer, first, Sort the list (Transform) and then check for uniqueness (Conquer).
- **Efficiency: Depends on Sorting algorithm chosen.**
- **If Merge sort is chosen then,**
  - $T(n)=T_{sort}(n)+T_{scan}(n) \in O(n \log n) + O(n) \in O(n \log n)$

Space and Time Tradeoffs– Sorting

- **In this technique, Problem's input is preprocessed and the additional information is stored, used while solving the problem.**
- **Ex: Sorting by counting, Boyer-Moore etc**
- **Sorting by Counting:**
  - Idea to count, for each element of a list to be sorted, the total number of elements smaller than an element and record the results in a table.
  - These numbers will indicate the positions of the elements in the sorted list
- **Consider the array,**

78	12	45	67	23	37

- **After applying the algorithm as said above the Count_Array [] would be,**

5	0	3	4	1	2

- **Final sorted list would be,**

0	1	2	3	4	5
12	23	37	45	67	78

Space and Time Tradeoffs– Sorting

- **Efficiency: It should be quadratic because the algorithm considers all the different pairs of an *n*-element array.**
- **Same as Selection sort.**
- **On the positive note, the algorithm makes the minimum number of key moves possible, placing each of them directly in their final position in a sorted array.**

Dynamic Programming – Knapsack Problem

- **Used to solve overlapping sub problems.**
- **Solves sub problems only once, store it in a table and will be used in future to obtain the solution.**
- **Lets consider Knapsack problem as an example.**
  - Given *n* items of known weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$ and a knapsack of capacity W, Find the most valuable subset of the items that fit into the knapsack.
- **Consider instance defined by first *i* items and capacity *j* ($j \le W$).**
- **Let $V[i, j]$ be optimal value of such an instance. Then**

$$V[i,j] = \begin{cases} \max\{V[i-1,j],\ v_i + V[i-1,j-w_i]\} & \text{if } j-w_i \ge 0 \\ V[i-1,j] & \text{if } j-w_i < 0 \end{cases}$$

- **Initial conditions:** $V[0,j] = 0$ and $V[i,0] = 0$

Dynamic Programming – Knapsack Problem

- **Example: Knapsack of capacity *W* = 5.**
- **Consider the below given table,**

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

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

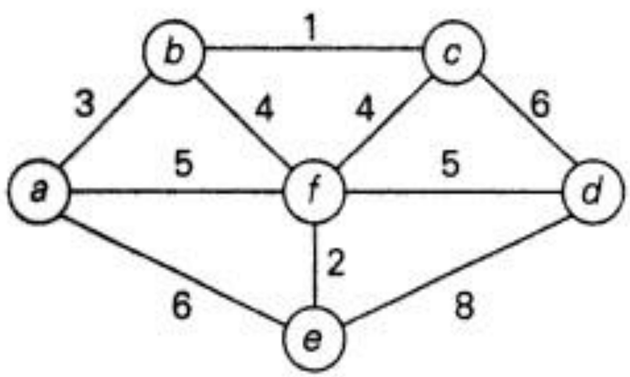
$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

	0	1	2	3	4	5
0	0	0	0			
1	0	0	12			
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	**37**

Greedy Technique – Kruskal's Algorithm

- **It constructs the solution through a sequence of steps.**
- **Example - Kruskal's Algorithm.**
  - Algorithm begins by sorting the graph's edges in non decreasing order of their weights.
  - Start with empty sub graph.
  - Scan the sorted list and add next edge on the list to the current sub graph. If the edge is creating a cycle then simply skip the edge.
- **Consider the following graph,**

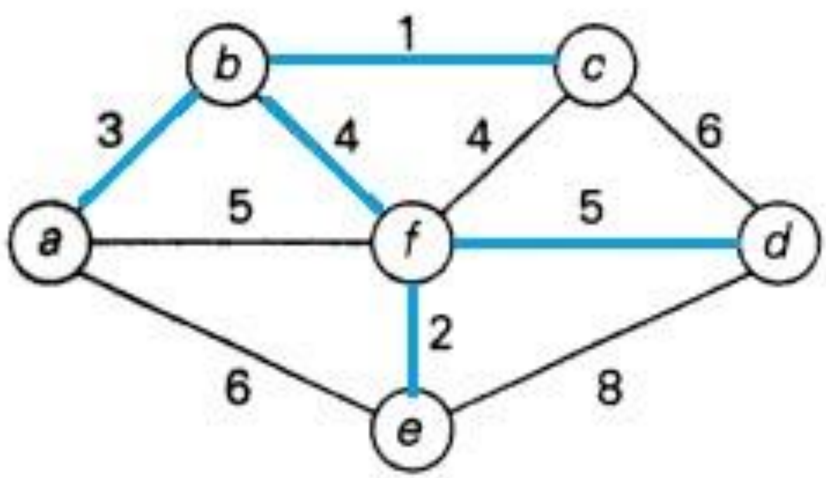


- **Sorted Edge List: (bc,1) (ef,2) (ab,3) (bf,4) (cf,4) (af,5) (df,5) (ae,6) (cd,6) (de, 8)**

Greedy Technique – Kruskal's Algorithm

- **Sub tree which forms minimum spanning tree is according to Kruskal's algorithm is**
  - (bc,1) (ef,2) (ab,3) (bf,4) (df,5)

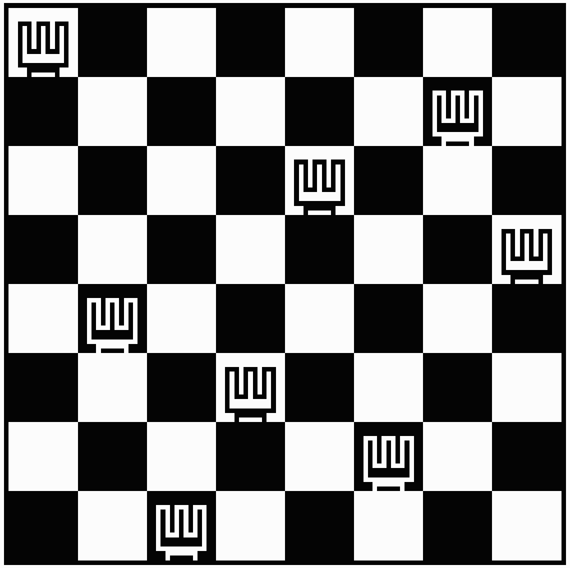
- **Below figure shows the subgraph which is minimum spanning tree.**



Backtracking – n-Queens problem

- **Backtracking is a technique used to solve problems with a large search space, by systematically trying and eliminating possibilities.**

- **Standard example of Backtracking is n-Queens Problem.**
  - Find an arrangement of 8 queens on a single chess board such that no two queens are attacking one another.
  - Due to the first two restrictions, it's clear that each row and column of the board will have exactly one queen.



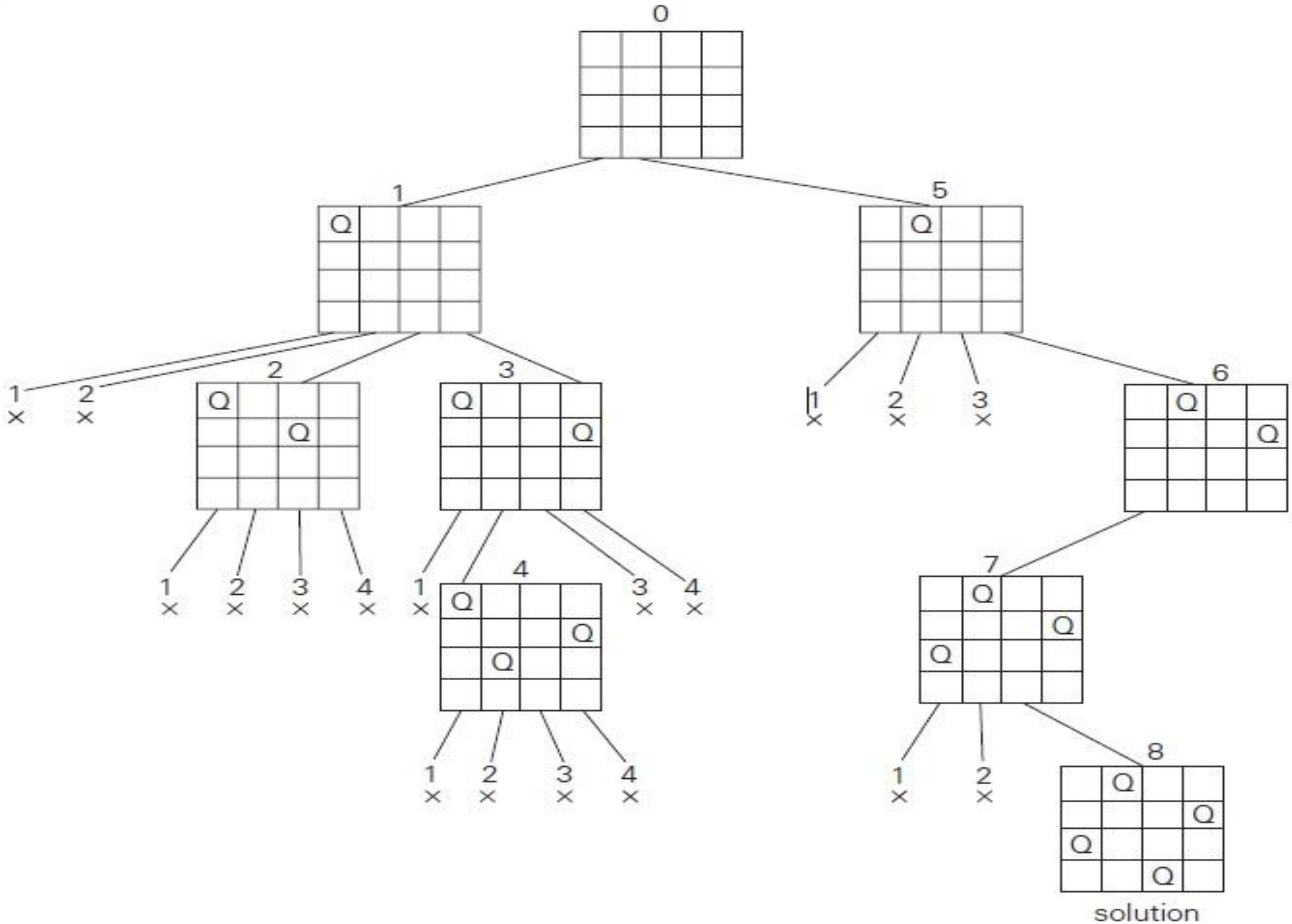
Backtracking - n-Queens Problem

- **The backtracking strategy is as follows:**
  - Place a queen on the first available square in row.
  - Move onto the next row, placing a queen on the first available square there (that doesn't conflict with the previously placed queens).
  - Continue in this fashion until either:
    - you have solved the problem, or
    - you get stuck.
  - When you get stuck, remove the queens that got you there, until you get to a row where there is another valid square to try.

Backtracking - n-Queens Problem

- **Another possible brute-force algorithm is generate the permutations of the numbers 1 through 8 (of which there are 8! = 40,320), and uses the elements of each permutation as indices to place a queen on each row. Then it rejects those boards with diagonal attacking positions.**
- **The backtracking algorithm, is a slight improvement on the permutation method,**
  - Constructs the search tree by considering one row of the board at a time, eliminating most non-solution board positions at a very early stage in their construction.
  - Because it rejects row and diagonal attacks even on incomplete boards, it examines only 15,720 possible queen placements.

Backtracking - n-Queens Problem



Branch and Bound – Assignment Problem

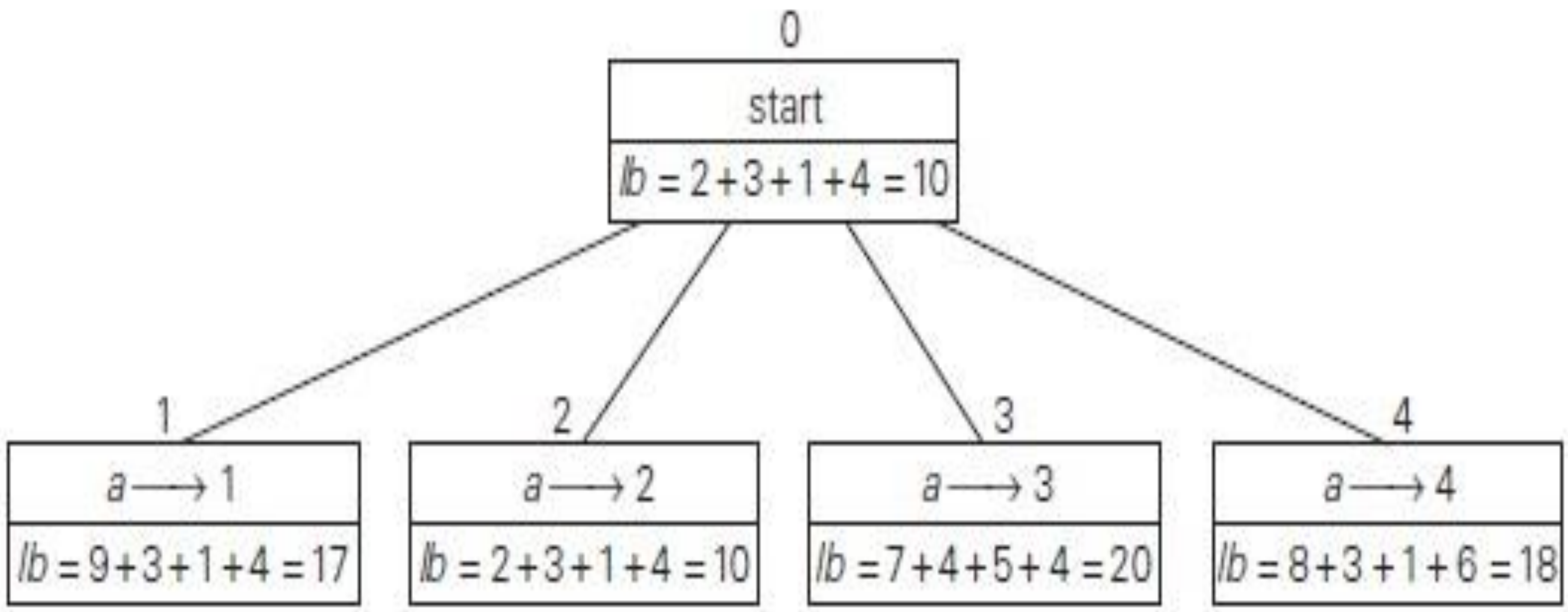
- **Enhancement of Backtracking.**
- **Applicable to optimization problems.**
- **Example: Assignment Problem**
  - **Select one element in each row of the cost matrix C so that:**
    - **no two selected elements are in the same column**
    - **the sum is minimized**

- **Example**

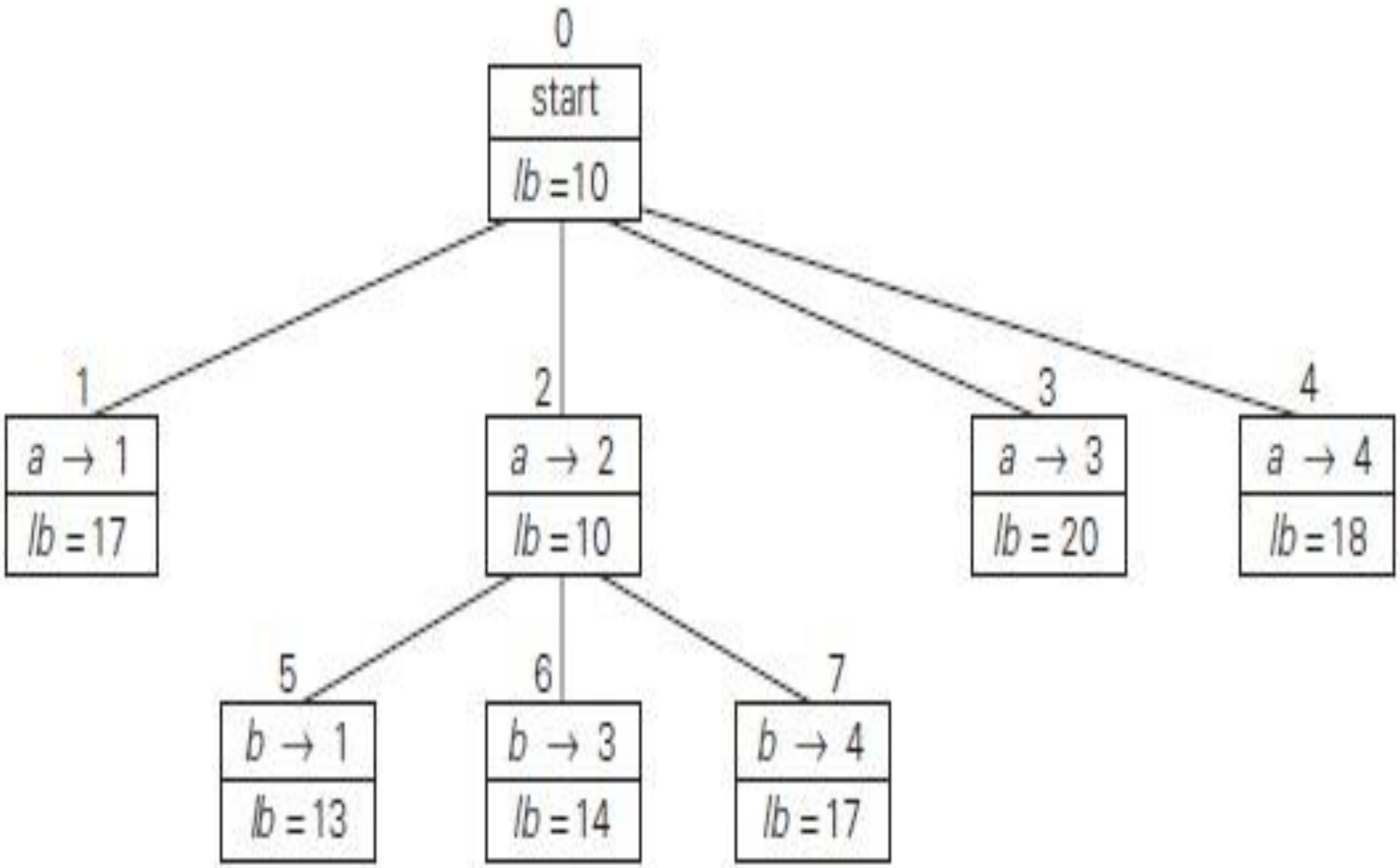
	Job 1	Job 2	Job 3	Job 4
Person *a*	9	2	7	8
Person *b*	6	4	3	7
Person *c*	5	8	1	8
Person *d*	7	6	9	4

- **Lower bound: Any solution to this problem will have total cost at least: 2 + 3 + 1 + 4**

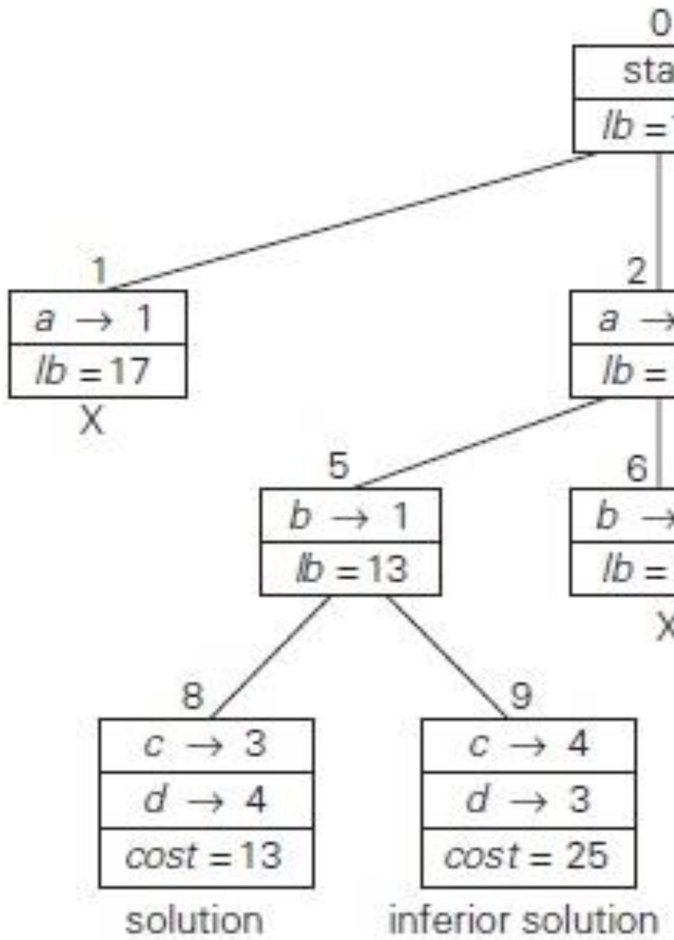
Branch and Bound – Assignment Problem



Branch and Bound – Assignment Problem



Branch and Bound – Assignment Problem



Lab

- **Lab Exercises 4**

Lesson Summary

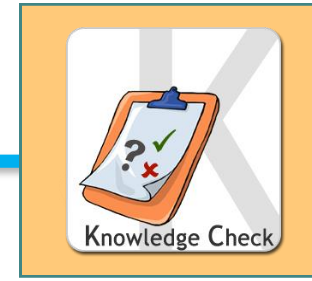
- **To understand and compare different Sorting techniques:**
  - Bubble Sort
  - Insertion Sort
- **To understand and compare different design techniques**
- **To identify proper design technique for the given problem and design an efficient algorithm accordingly.**

Review Questions

- **Question 1: Which of the following sorting techniques uses swapping of two elements to sort the array:**
  - A: Bubble sort
  - B: Quick sort
  - C: Insertion sort
- **Question 2: What is the efficiency of bubble sort algorithm?**
  - O(n)
  - $O(n^2)$
  - O(nlogn)
  - O(logn)
- **Question 3: Arranging elements in an ascending or descending order is called as ___**

- **Question 4: ____________ techniques are mainly used to solve difficult combinatorial problems.**

Review Questions: Match the Following



- **Question 3:**

1.	Bubble sort
2.	Sequential search
3.	Binary search
4.	Insertion sort

a. Best case is finding element at the fist position
b. Require to use nested loops
c. Find position before inserting element
d. Best case is finding the element at the middle
e. Collision