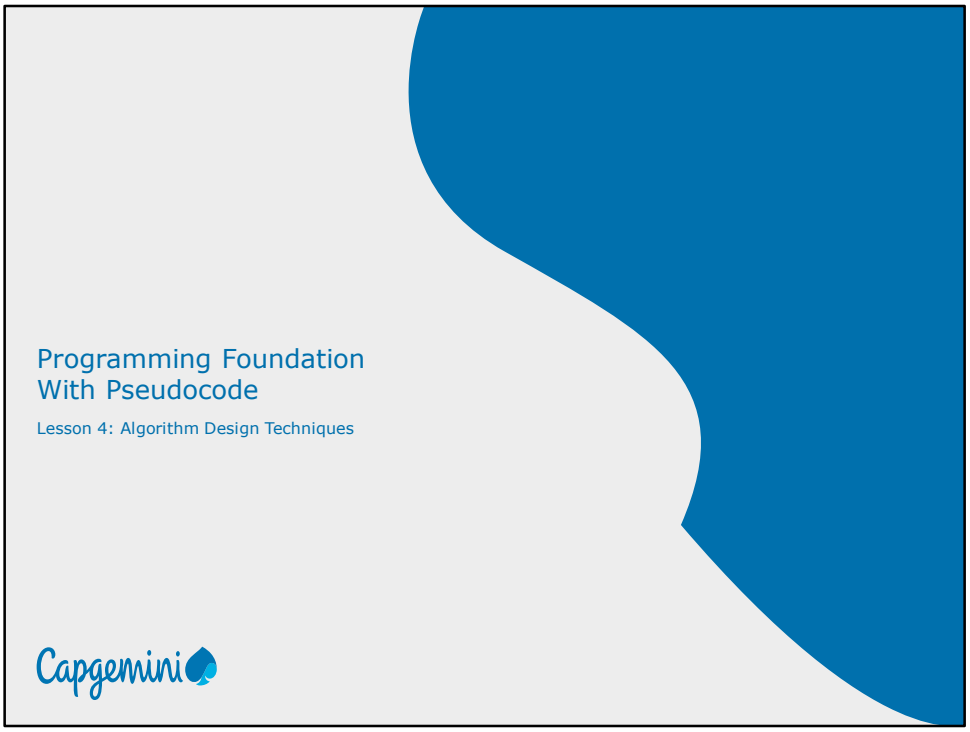
Instructor Notes:



Instructor Notes:

Lesson Objectives

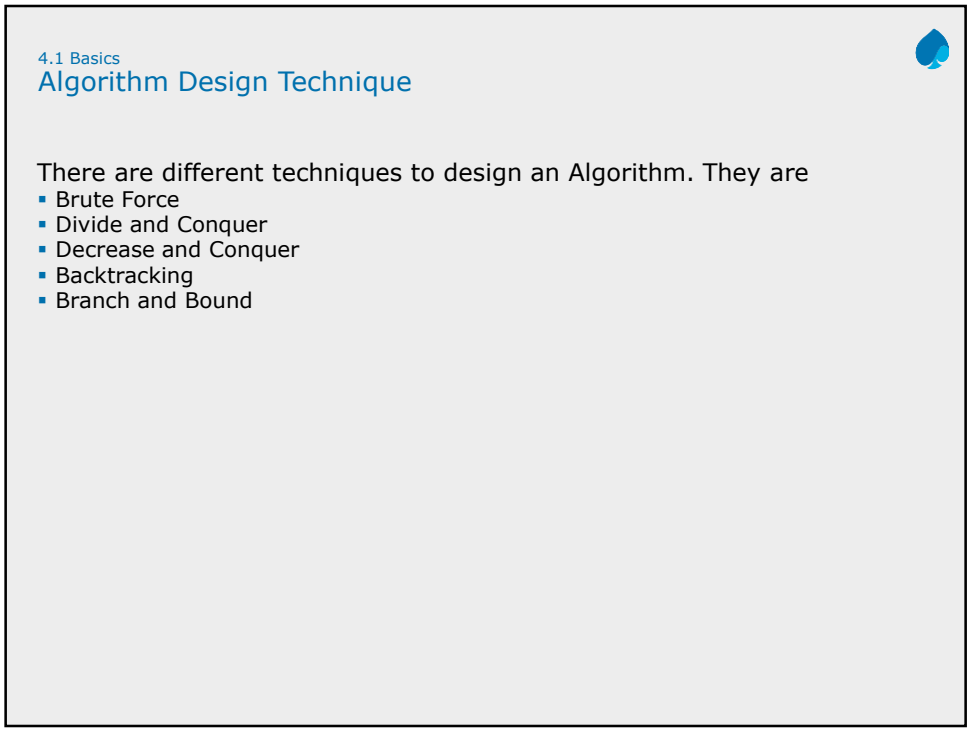
To enhance the logic building by designing algorithms efficiently for the given problem.

To understand and compare different Sorting methods under different design techniques:

- Bubble Sort
- Merge Sort
- Insertion Sort

To understand and compare different algorithms designed for a given problem under different design techniques:

Instructor Notes:



Algorithm Design Techniques provides various different approaches to solve a given problem. Various design techniques are available as listed in the above slide.
Brute Force: This is “Just do it” type. Easiest solution is found for a problem without any concern of the efficiency parameter.
Divide and Conquer: Here the task is divided, solution is found for the smallest unit and the solutions are conquered to derive the final result.
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.
Backtracking: Backtracking is a technique used to solve problems with a large search space, by systematically trying and eliminating possibilities.

Instructor Notes:

4.2.1 Bubble Sort

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

Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved. "Just do it!" would be another way to describe the prescription of the brute-force approach. Brute Force solutions will not have concerns for the efficiency parameters.

Example for Bubble sort is explained in detail in the above slides.

Instructor Notes:

4.2.1 Bubble Sort

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(n2)

ALGORITHM *BubbleSort(List[0..n-1])*
//Sorts a given array using bubble sort
//Input: An array list[0..n-1] of orderable elements
//Output: Array list[0..n-1] sorted in ascending order
for itr← 0 to n-2 do
for index←0 to n-2-itr do
if list[index+1] < list[index] swap list[index] and list[index+1]

Efficiency of the given algorithm is $\Theta(n^2)$.

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

$$= \sum_{itr=0}^{n-2} (n-1-itr) = n(n-1)/2 \in \mathbf{O(n^2)}$$

Instructor Notes:

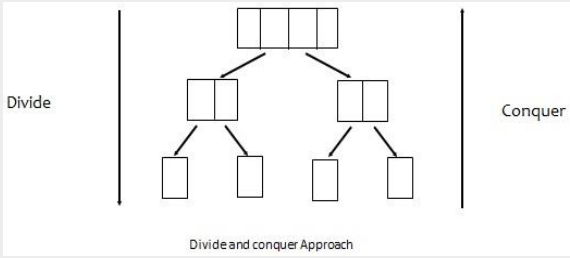
4.2.2 Merge Sort

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.

Ex:



Divide and conquer Approach

The *divide-and-conquer* technique involves solving a particular computational problem by dividing it into one or more subproblems of smaller size, recursively solving each subproblem, and then "merging" or "marrying" the solutions to the subproblem(s) to produce a solution to the original problem.

Quite a few very efficient algorithms are specific implementations of this general strategy. Divide and conquer technique work according to the following steps,

- 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.

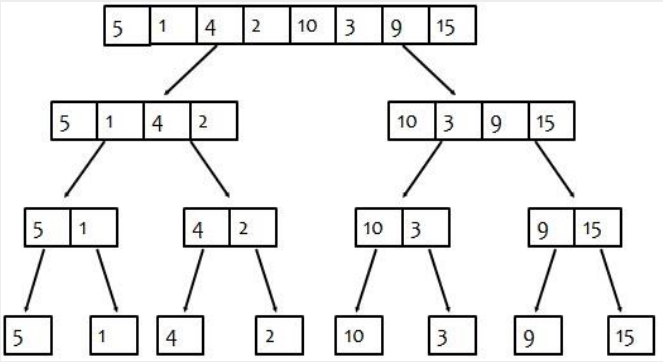
Note: Not every divide-and-conquer algorithm is necessarily more efficient than even a brute-force solution.

Instructor Notes:

4.2.2 Merge Sort

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.



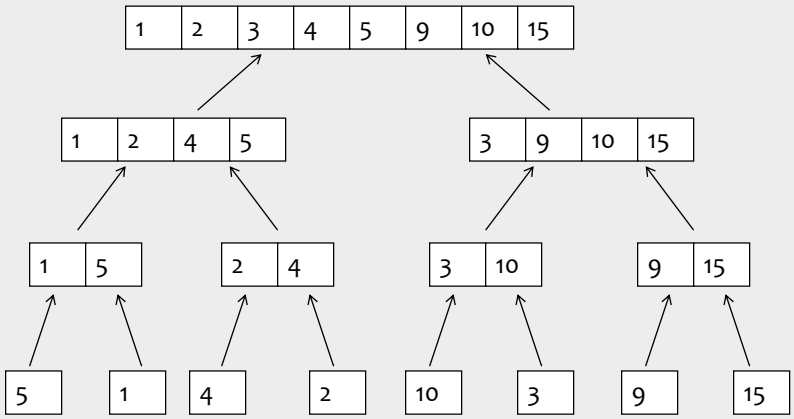
- Consider above given input, 5,1,4,2,10,3,9,15.
- First the task is divided in smallest unit as shown below.

Instructor Notes:

4.2.2 Merge Sort

ADT – Divide and Conquer

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



Efficiency of Merge Sort

- $C_{Best}(n)=O(nlogn)$
- $C_{Worst}(n)=O(nlogn)$
- $C_{Avg}(n)=O(logn)$

Efficiency of Quick Sort

- $C_{Best}(n)=O(nlogn)$
- $C_{Worst}(n)=O(n^2)$ [Depends on the pivot element chosen]
- $C_{Avg}(n)=O(1.38nlogn)$

Instructor Notes:

4.2.3 Insertion Sort

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)

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.

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

- decrease by a constant : In the ***decrease-by-a-constant*** variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one

- decrease by a constant factor: The ***decrease-by-a-constant-factor*** technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm.

- variable size decrease: The size-reduction pattern varies from one iteration of an algorithm to another.

Instructor Notes:

4.2.3 Insertion Sort

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]

Example: Consider the following array a.
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) – Initially only a[0] element is sorted

Consider element at a[1] 7 > 5 , So no change will be there and sorted part will increase from a[0] to a[1]
5 7 0 3 4 2 6 1 (0)

Consider element at a[2] we will copy it in index variable and check 7 >0, 5>0 hence 5 and 7 will get shifted to one location on right and put 0 at a[0] location Now the size of sorted part is 3
0 5 7 3 4 2 6 1 (2)

Similarly we will place element 3 from a[3] to a [1] position and array will be as follows
0 3 5 7 4 2 6 1 (2)

In further iterations array will get sorted as follows

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

Instructor Notes:

4.2.3 Insertion Sort

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

Instructor Notes:

4.2.3 Insertion Sort

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(n2).

Efficiency of insertion sort is $O(n^2)$.
More efficient sorting algorithms are merge sort, heap sort and quick sort. Their efficiency falls under nlogn class but they are restricted with their own conditions.
Hence it depends on the input type to decide which algorithm must be used for a given application.

Instructor Notes:

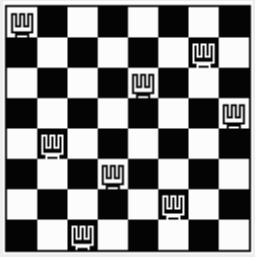
4.3 Basics

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.



Both backtracking and branch-and-bound are based on the construction of a ***state-space tree*** whose nodes reflect specific choices made for a solution's components. Both techniques terminate a node as soon as it can be guaranteed that no solution to the problem can be obtained by considering choices that correspond to the node's descendants. Difference is that Branch and Bound is only applied to the optimization problems where as Backtracking does not have any such constraints. The other difference is that the order in which nodes of the state-space tree are generated. Backtracking uses DFS and Branch and bound uses various rules, one of them is best-first rule.

Instructor Notes:

4.3 Example

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.

Instructor Notes:

4.3 Example

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.

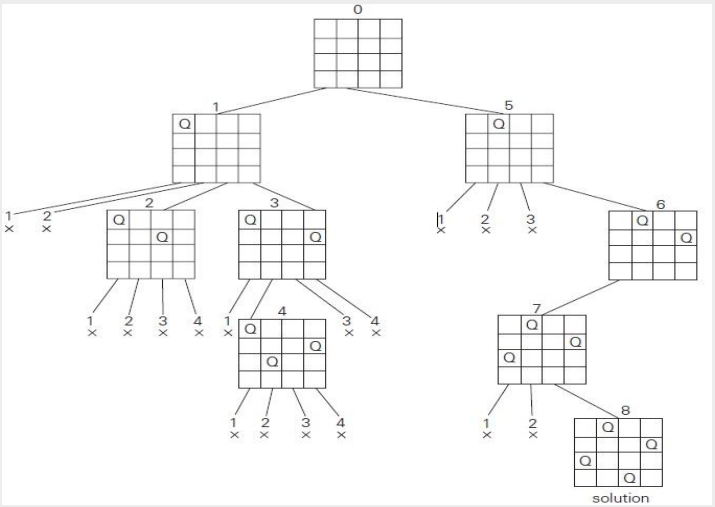
The backtracking algorithm, is a slight improvement on the permutation method,

- 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.

Instructor Notes:

4.3 State space tree for 4-Queens Problem

Backtracking - n-Queens Problem



Lets place queen 1 in the first possible position of its row, which is in column 1 of row 1. Then we place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3. This proves to be a dead end because there is no acceptable position for queen 3. So, the algorithm backtracks and puts queen 2 in the next possible position at (2, 4). Then queen 3 is placed at (3, 2), which proves to be another dead end. The algorithm then backtracks all the way to queen 1 and moves it to (1, 2). Queen 2 then goes to (2, 4), queen 3 to (3, 1), and queen 4 to (4, 3), which is a solution to the problem. This is depicted in the state space tree on the above slide.

Instructor Notes:

4.4 Basics

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 is applicable for the optimization problem. ***Optimal solution*** is a feasible solution with the best value of the objective function. we can compare a node's bound value with the value of the best solution seen so far. If the bound value is not better than the value of the best solution seen so far—i.e., not smaller for a minimization problem and not larger for a maximization problem—the node is nonpromising and can be terminated. Indeed, no solution obtained from it can yield a better solution than the one already available. This is the principal idea of the branch-and-bound technique.

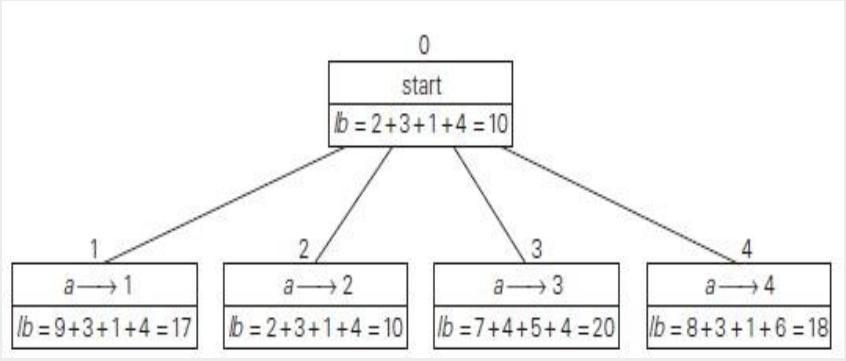
Let us illustrate the branch-and-bound approach by applying it to the problem of assigning n people to n jobs so that the total cost of the assignment is as small as possible. Here each person is assigned to exactly one job and each job is assigned to exactly one person. The cost that would accrue if the ith person is assigned to the jth job is a known quantity $C[i, j]$ for each pair $i, j = 1, 2, \ldots, n$. The problem is to find an assignment with the minimum total cost.Consider the above given instance,

The cost of any solution, including an optimal one, cannot be smaller than the sum of the smallest elements in each of the matrix's rows. For ex: 2 + 3+ 1+ 4 = 10. It is the lower bound.

Instructor Notes:

4.4 Example

Branch and Bound – Assignment Problem



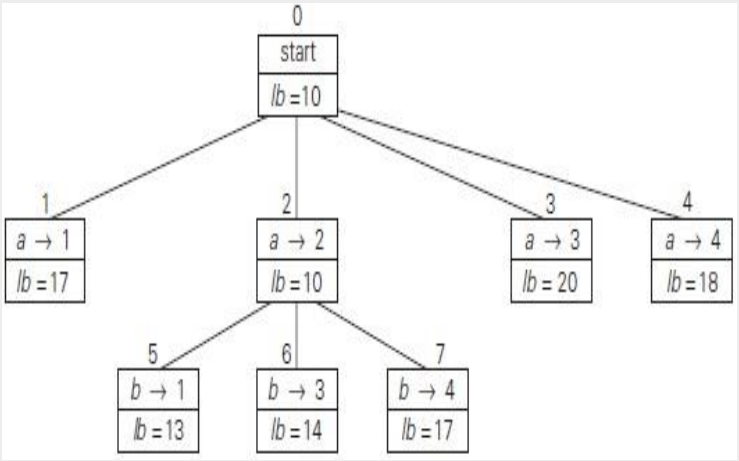
The nodes on the first level of the tree correspond to selections of an element in the first row of the matrix, i.e., a job for person *a*. So we have four live leaves—nodes 1 through 4—that may contain an optimal solution. The most promising of them is node 2 because it has the smallest lowerbound value.

So we have four live leaves—nodes 1 through 4—that may contain an optimal solution. The most promising of them is node 2 because it has the smallest lowerbound value.

Instructor Notes:

4.4 Example

Branch and Bound – Assignment Problem

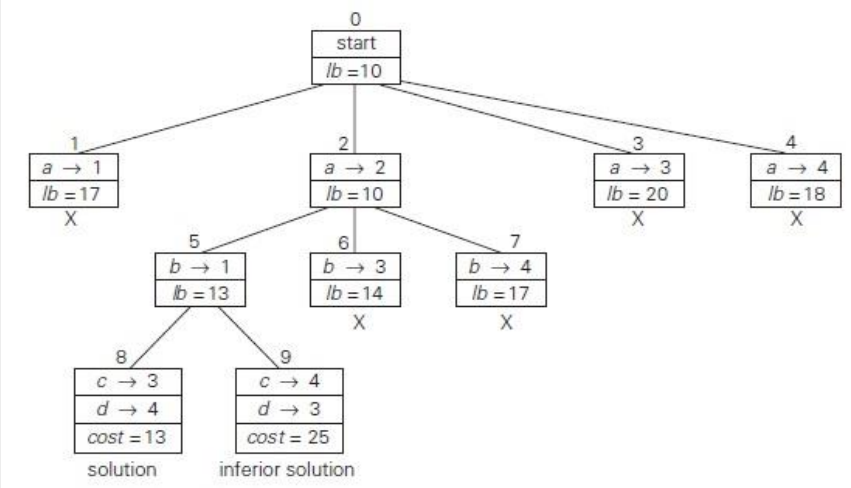


Following our best-first search strategy, we branch out from that node first by considering the three different ways of selecting an element from the
second row and not in the second column—the three different jobs that can be assigned to person *b*

Instructor Notes:

4.4 Example

Branch and Bound – Assignment Problem



From the six live leaves—nodes 1, 3, 4, 5, 6, and 7 (which may contain an optimal solution) , again node with the smallest lower bound is chosen, node 5. First, we consider selecting the third column's element from *c*'s row (i.e., assigning person *c* to job 3); this leaves us with no choice but to select the element from the fourth column of *d*'s row (assigning person *d* to job 4). This yields leaf 8 which corresponds to the feasible solution {*a*→2, *b*→1, *c*→3, *d* →4} with the total cost of 13. Its sibling, node 9, corresponds to the feasible solution {*a*→2, *b*→1, *c*→4, *d* →3} with the total cost of 25. Since its cost is larger than the cost of the solution represented by leaf 8, node 9 is simply terminated. (Of course, if its cost were smaller than 13, we would have to replace the information about the best solution seen so far with the data provided by this node.) Now, as we inspect each of the live leaves of the last state-space tree—nodes 1, 3, 4, 6, and 7 we discover that their lower-bound values are not smaller than 13, the value of the best selection seen so far (leaf 8). Hence, we terminate all of them and recognize the solution represented by leaf 8 as the optimal solution to the problem.

Instructor Notes:

Lab

Lab Exercises 3

Write a pseudocode to test the concept of all sorting and searching problems.

Instructor Notes:

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.

Instructor Notes:

Answers:
Question1: A
Question 2: B
Question 3 : Sorting

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?

- A. O(n)
- B. O(n2)
- C. O(nlogn)
- D. O(logn)

Question 3: Arranging elements in an ascending or descending order is called as _____.

Instructor Notes:

Answers

Question 4:

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

Review Questions: Match the Following

Question 4:

1. Bubble sort	a. Best case is finding element at the fist position
2. Sequential search	b. Require to use nested loops
3. Binary search	c. Find position before inserting element
4. Insertion sort	d. Best case is finding the element at the middle
	e. Collision