Q.2 a. What is time and space complexity of algorithms? Provide the complexity of various searching and sorting techniques.

Answer:

The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the size of the input to the problem. The time complexity of an algorithm is commonly expressed using big O notation, which suppresses multiplicative constants and lower order terms. When expressed this way, the time complexity is said to be described asymptotically, i.e., as the input size goes to infinity. For example, if the time required by an algorithm on all inputs of size n is at most 5n3 + 3n, the asymptotic time complexity is O(n3).

Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, where an elementary operation takes a fixed amount of time to perform. Thus the amount of time taken and the number of elementary operations performed by the algorithm differ by at most a constant factor.

Since an algorithm's performance time may vary with different input sizes, the most commonly used measure of time complexity, the worst-case time complexity of an algorithm, denoted as T(n), is the maximum amount of time taken on any input of size n. Time complexities are classified by the nature of the function T(n). For instance, an algorithm with T(n) = O(n) is called a linear time algorithm, and an algorithm with T(n) = O(2n) is said to be an exponential time algorithm.

The way in which the amount of storage space required by an algorithm varies with the size of the problem it is solving. Space complexity is normally expressed as an order of magnitude, e.g. O(N^2) means that if the size of the problem (N) doubles then four times as much working storage will be needed.

- The worst-case time for a sequential search is always O(N).
- The worst-case time for binary search is proportional to $\log_2$ N: the number of times N can be divided in half before there is nothing left. Using big-O notation, this is O(log N)
- Selection sort and insertion sort have worst-case time O(N^2).
- Quick sort is also O(N^2) in the worst case, but its expected time is O(N log N).
- Merge sort is O(N log N) in the worst case.

Q.3 a. Write a recursive algorithm for solving Tower of Hanoi problem. What is the basic operation in the Tower of Hanoi problem and give the recurrence relation for the number of moves?

Answer:

```
#include <stdio.h>
void towers_of_hanoi (char source, char temp, char destination, int n)
  if (n == 0)
{
  return;
 }
// (S,D,T,n-1)
towers_of_hanoi (source, destination, temp, n-1);
```

```
// move the bottom most disk from `source' to `destination' of the current problem

printf ("\n%d disk (%c -> %c)", n, source, destination);

(T,S,D,n-1)

towers_of_hanoi (temp, source, destination, n-1);
 }
int main (void)

{
int n;

printf ("\nNumber of Disks \"n\": ");

scanf ("%d", &n);

towers_of_hanoi ('S', 'T', 'D', n);

printf ("\n");

 return 0;
```

From the above recursive algorithm we can see that the problem is broken down to 2 subproblems of size (n-1), and one operation to print the movement of the bottom most disk, which requires constant time, and can be seen as one operation.

Therefore the growth of time with respect to the input problem size is $T(n) = 2T(n-1) + 1$with $T(1) = 1$as the trivial case and base condition of the recurrence relation. This homogeneous recurrence relation can easily be solved to $T(n) = 2^n - 1$. Therefore total number of moves required to solve a Towers of Hanoi problem instance of size n is $2^n - 1$. Like for 2 disks we have 3 moves, 3 disks we have 7 moves, and 10 disks we have 1023 moves. Clearly the time growth of the problem is $\mathcal{O}(2^n)$. Therefore even for a computer is supplied with an enough large value its CPU time can be entirely devoted to solve the input problem instance.

b. What is the computational complexity of the Fibonacci sequence and how is it calculated?

Answer:

You model the time function to calculate Fib(n) as sum of time to calculate Fib(n-1) plus the time to calculate Fib(n-2) plus the time to add them together (O(1)).

T(n<=1) = O(1)

T(n) = T(n-1) + T(n-2) + O(1)

You solve this recurrence relation (using generating functions, for instance) and you'll end up with the answer.

Alternatively, you can draw the recursion tree, which will have depth n and intuitively figure out that this function is asymptotically $O(2^n)$. You can then prove your conjecture by induction.

Base: n = 1 is obvious

Assume $T(n-1) = O(2^{n-1})$, *therefore*

T(n) = T(n-1) + T(n-2) + O(1) *which is equal to*

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

However, as noted in a comment, this is not the tight bound. An interesting fact about this function is that the T(n) is asymptotically **the same** as the value of Fib(n) since both are defined as

f(n) = f(n-1) + f(n-2).

The leaves of the recursion tree will always return 1. The value of Fib(n) is sum of all values returned by the leaves in the recursion tree which is equal to the count of leaves. Since each leaf will take O(1) to compute, T(n) is equal to Fib(n) x O(1). Consequently, the tight bound for this function is the Fibonacci sequence itself ($\sim\theta(1.6^n)$)

Q.4 a. Write an algorithm used to partition an array for quicksort. Explain and calculate its complexity.

Answer:

Quicksort is one of the fastest and simplest sorting algorithms .It works recursively by a divide-and-conquer strategy.

First, the sequence to be sorted *a* is partitioned into two parts, such that all elements of the first part *b* are less than or equal to all elements of the second part *c* (divide). Then the two parts are sorted separately by recursive application of the same procedure (conquer). Recombination of the two parts yields the sorted sequence (combine). Figure 1 illustrates this approach.

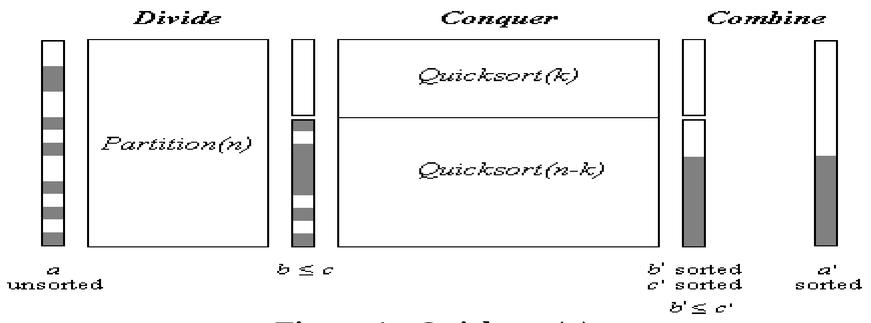


Figure 1: Quicksort(n)

The first step of the partition procedure is choosing a comparison element x. All elements of the sequence that are less than x are placed in the first part, all elements greater than x are placed in the second part. For elements equal to x it does not matter into which part they come. In the following algorithm it may also happen that an element equal to x remains between the two parts.

Algorithm *Partition*

Input: sequence $a_0, ..., a_{n-1}$ with n elements

Output: permutation of the sequence such that all elements $a_0, ..., a_j$ are less than or equal to all elements $a_i, ..., a_{n-1}$ $(i > j)$

Method:

```
1. choose the element in the middle of the sequence as comparison
   element $x$

   let $i = 0$ and $j = n-1$
   while $i \le j$

       1. search for the first element $a_i$ which is greater than or equal to
          $x$

          search for the last element $a_j$ which is less than or equal to $x$
          if $i \le j$

              1. exchange $a_i$ and $a_j$

                 let $i = i+1$ and $j = j-1$
```

After partitioning the sequence, quicksort treats the two parts recursively by the same procedure. The recursion ends whenever a part consists of one element only.

Analysis

The best-case behavior of the quicksort algorithm occurs when in each recursion step the partitioning produces two parts of equal length. In order to sort n elements, in this case the running time is in $\Theta(n \log(n))$. This is because the recursion depth is $\log(n)$ and on each level there are n elements to be treated (Figure 2 a).

The worst case occurs when in each recursion step an unbalanced partitioning is produced, namely that one part consists of only one element and the other part consists of the rest of the elements (Figure 2 c). Then the recursion depth is n-1 and quicksort runs in time $\Theta(n^2)$.

In the average case a partitioning as shown in Figure 2 b is to be expected.

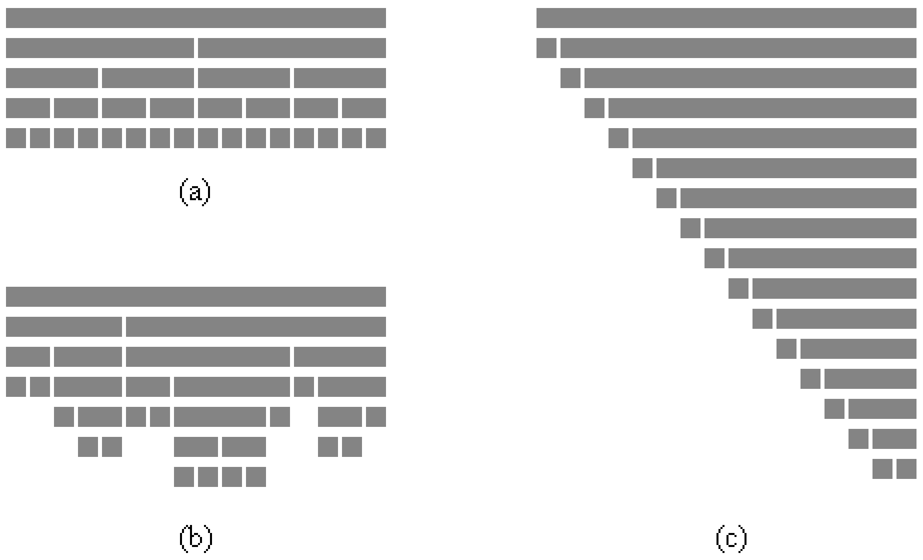


Figure 2: Recursion depth of quicksort: a) best case, b) average case, c) worst case

The choice of the comparison element x determines which partition is achieved. Suppose that the first element of the sequence is chosen as comparison element. This would lead to the worst case behavior of the algorithm when the sequence is initially sorted. Therefore, it is better to choose the element in the middle of the sequence as comparison element.

Even better would it be to take the $n/2$-th greatest element of the sequence (the median). Then the optimal partition is achieved. Actually, it is possible to compute the median in linear time . This variant of quicksort would run in time $O(n \log(n))$ even in the worst case.

However, the beauty of quicksort lies in its simplicity. And it turns out that even in its simple form quicksort runs in $O(n \log(n))$ on the average. Moreover, the constant hidden in the O-notation is small. Therefore, we trade this for the (rare) worst case behavior of $\Theta(n^2)$.

Proposition: The time complexity of quicksort is in

$\Theta(n \log(n))$	in the average case
$\Theta(n^2)$	in the worst case

Q.5 a. Write and explain heapsort algorithm and show that its worst case performance is O(n log n).

Answer:

The running time of heapsort is O(N*lgN) i.e. it achieves the lower bound for computational based sorting.

HEAP
The heap data structure is an array object which can be easily visualized as a complete binary tree.There is a one to one correspondence between elements of the array and nodes of the tree.The tree is completely filled on all levels except possibly the lowest,which is filled from the left upto a point.All nodes of heap also satisfy the relation that the key value at each node is at least as large as the value at its children.

Step I: The user inputs the size of the heap(within a specified limit).The program generates a corresponding binary tree with nodes having randomly generated key Values.

Step II: Build Heap Operation:Let n be the number of nodes in the tree and i be the key of a tree.For this,the program uses operation Heapify.when Heapify is called both the left and right subtree of the i are Heaps.The function of Heapify is to let i settle down to a position(by swapping itself with the larger of its children,whenever the heap property is not satisfied)till the heap property is satisfied in the tree which was rooted at (i).This operation calls

Heapsort (worst case complexity T(n) = O(n log n)) uses a data structure known as a max-heap which is a complete binary tree where the element at any node is the maximum of itself and all its children. A tree can be stored either with pointers as per the pictorial representation we are normally used to visualize, or by mapping it to a vector. Here if a node in the tree is mapped to the ith element of an array, then its left child is at 2i, its right child is at (2i+1) and its parent is at floor(i/2).
We can construct a heap by starting with a an existing heap, adding an element at the bottom of the heap, and allow it to migrate up the father chain till it finds its proper position.
Complexity T(n) = O(n log n)

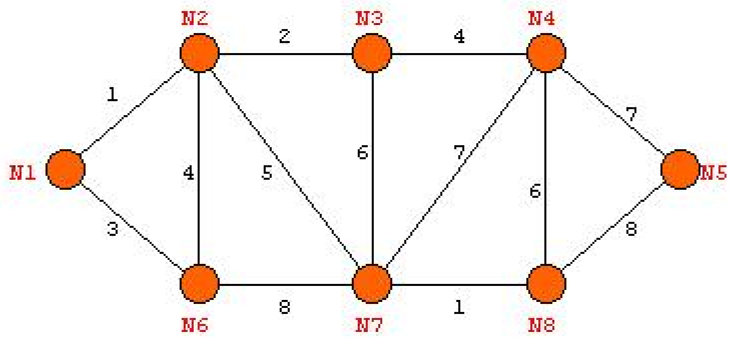
Step III: Remove maximum element:The program removes the largest element of the heap(the root) by swapping it with the last element.

Step IV: The program executes Heapify(new root) so that the resulting tree satisfies the heap property.

Step V: Goto step III till heap is empty

Q.6 a. Describe in detail both Prim’s and Kruskal’s algorithms for finding a minimum cost spanning tree of an undirected graph with edges labelled with positive costs and explain why they are correct. Compare the relative merits of the two algorithms.

Answer:
The example graph below is used to show how Kruskal's Algorithm works for the determining of the minimum spanning tree (MST). It is highly recommended, in order for you to see the difference between Kruskal's Algorithm and Prim's Algorithm, that you draw the below graph for the Prim applet, and go through it step by step.



Kruskal's Algorithm

Using the above graph, here are the steps to the MST, using Kruskal's Algorithm:

N1 to N2 - cost is 1 - add to tree
N7 to N8 - cost is 1 - add to tree
N2 to N3 - cost is 2 - add to tree
N1 to N6 - cost is 3 - add to tree
N2 to N6 - cost is 4 - reject because it forms a circuit
N3 to N4 - cost is 4 - add to tree
N2 to N7 - cost is 5 - add to tree
N3 to N7 - cost is 6 - reject because it forms a circuit
N4 to N8 - cost is 6 - reject because it forms a circuit
N4 to N7 - cost is 7 - reject because it forms a circuit
N4 to N5 - cost is 7 - add to tree

We stop here, because n -1 edges have been added. We are left with the minimum spanning tree, with a total weight of 23.

Prim's Algorithm

This algorithm is directly based on the MST property. Assume that $V = \{1, 2,..., n\}$.

```
{
        T = $\phi$;
        U = { 1 };
        while ($U \neq V$)
        {
           let $(u, v)$ be the lowest cost edge
           such that $u \in U$ and $v \in V - U$;
T = $T \cup \{(u, v)\}$
U = $U \cup \{v\}$
        }
    }
```

Implementation of Prim's Algorithm

Use two arrays, **closest and lowcost.**

- For $i \in V - U$, closest[i] gives the vertex in U that is closest to i
- For $i \in V - U$, lowcost[i] gives the cost of the edge (i, closest(i))

b. Explain how the knapsack problem is solved with approximation algorithm.

Answer:

There are n items in a store. For i =1,2, . . . , n, item i has weight $w_i > 0$ and worth $v_i > 0$. Thief can carry a maximum weight of W pounds in a knapsack. In this version of a problem the items can be broken into smaller piece, so the thief may decide to carry only a fraction x_i of object i, where $0 \le x_i \le 1$. Item i contributes $x_i w_i$ to the total weight in the knapsack, and $x_i v_i$ to the value of the load.

In Symbol, the fraction knapsack problem can be stated as follows. maximize $^n\mathbf{S}_{i=1}\, x_i v_i$ subject to constraint $^n\mathbf{S}_{i=1}\, x_i w_i \le W$

It is clear that an optimal solution must fill the knapsack exactly, for otherwise we could add a fraction of one of the remaining objects and increase the value of the load. Thus in an optimal solution $^n\mathbf{S}_{i=1}\, x_i w_i = W$.

Greedy-fractional-knapsack (*w*, *v*, *W*)

```
FOR $i$ =1 to $n$
   do $x[i]$ =0
weight = 0
while weight < $W$
   do $i$ = best remaining item
      IF weight + $w[i] \le W$
         then $x[i]$ = 1
            weight = weight + $w[i]$
```

```
        else
            $x[i] = (w - \text{weight}) / w[i]$
            weight = $W$
return $x$
```

Analysis

If the items are already sorted into decreasing order of v_i / w_i, then
the while-loop takes a time in *O(n)*;
Therefore, the total time including the sort is in *O(n log n)*.

If we keep the items in heap with largest v_i/w_i at the root. Then

- creating the heap takes *O(n)* time
- while-loop now takes *O(log n)* time (since heap property must be restored after the removal of root)

  Although this data structure does not alter the worst-case, it may be faster if only a small number of items are need to fill the knapsack.

One variant of the 0-1 knapsack problem is when order of items are sorted by increasing weight is the same as their order when sorted by decreasing value.

The optimal solution to this problem is to sort by the value of the item in decreasing order. Then pick up the most valuable item which also has a least weight. First, if its weight is less than the total weight that can be carried. Then deduct the total weight that can be carried by the weight of the item just pick. The second item to pick is the most valuable item among those remaining. Keep follow the same strategy until thief cannot carry more item (due to weight).

Q.8 a. Write a note on the challenges of numerical algorithms.
Answer: Page Number 382 of Text Book I

Q.9 a. Apply the branch-and-bound technique in solving the travelling Salesman Problem.
Answer:
Suppose it is required to minimize an objective function. Suppose that we have a method for getting a lower bound on the cost of any solution among those in the set of solutions represented by some subset. If the best solution found so far costs less than the lower bound for this subset, we need not explore this subset at all.

Let S be some subset of solutions. Let

$L(S) =$ a lower bound on the cost of
any solution belonging to S

Let $C =$ cost of the best solution
found so far

If $C \leq L(S)$, there is no need to explore S because it does not contain any better solution.

If $C > L(S)$, then we need to explore S because it may contain a better solution.

A Lower Bound for a TSP

Note that:

Cost of any tour

$$= \frac{1}{2} \sum_{v \in V} \text{(Sum of the costs of the two tour edges adjacent to } v\text{)}$$

Now:

The sum of the two tour edges adjacent to a given vertex v

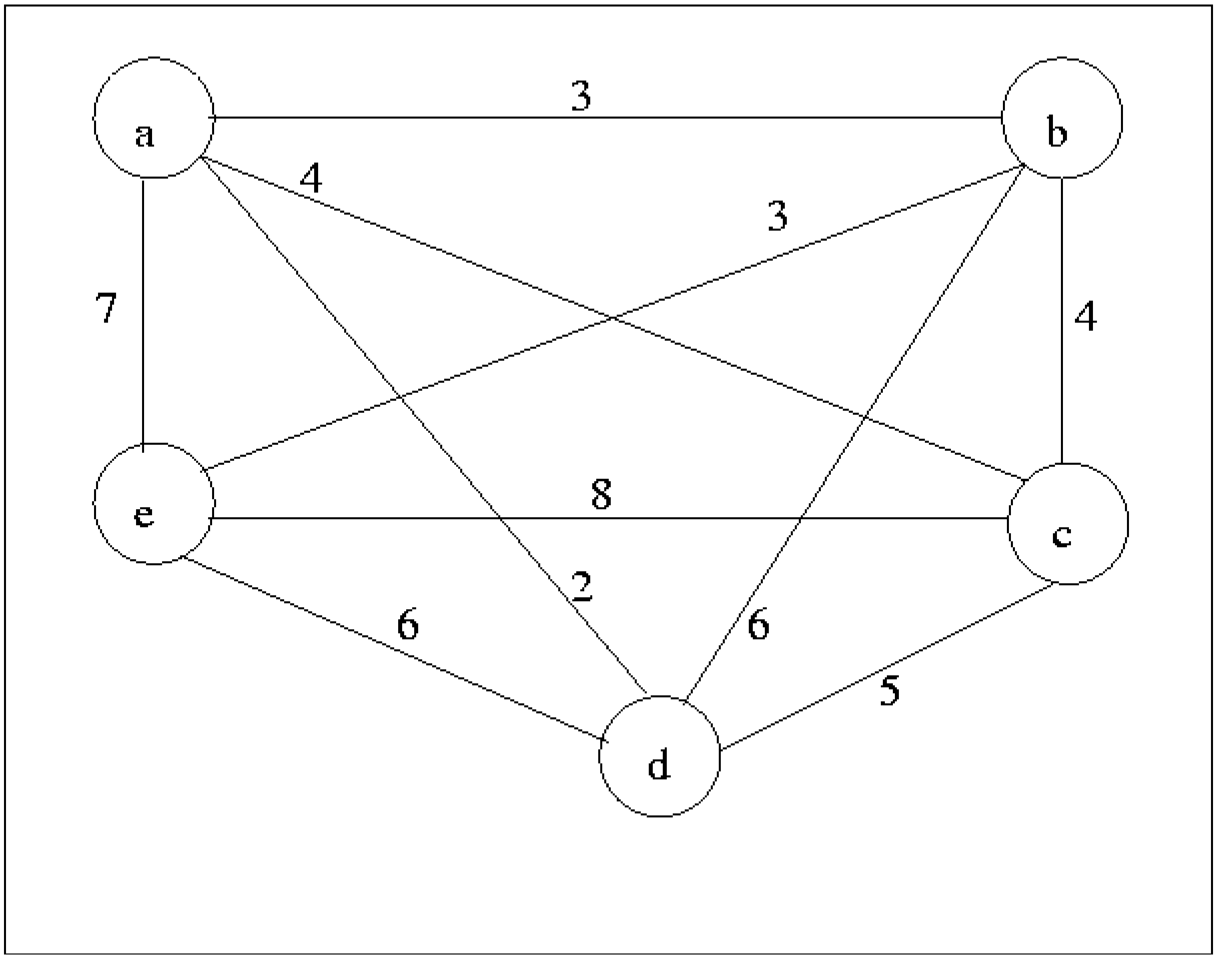
$\geq$ sum of the two edges of least cost adjacent to v

Therefore:

Cost of any tour

$$\geq \frac{1}{2} \sum_{v \in V} \text{(Sum of the costs of the two least cost edges adjacent to } v\text{)}$$

Figure 8.17: Example of a complete graph with five vertices



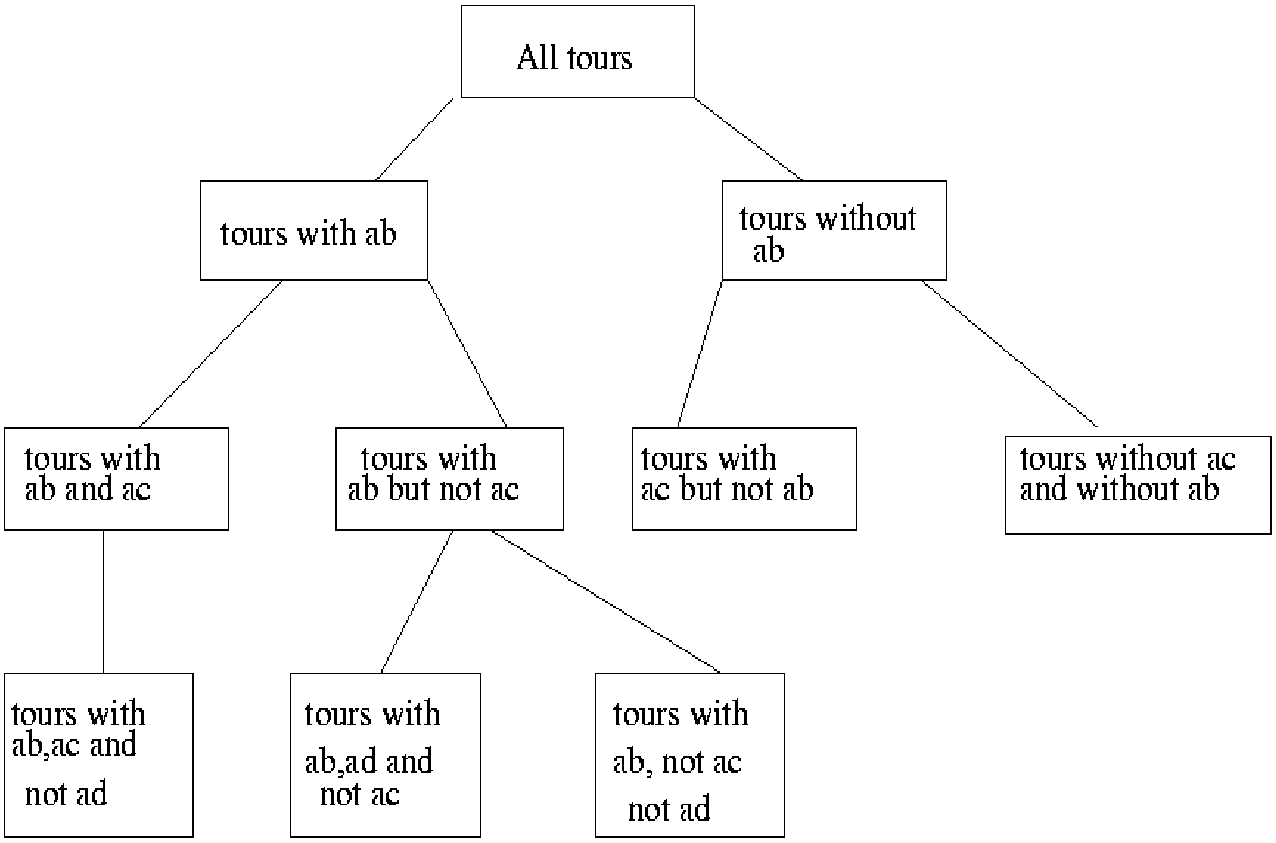
Node	**Least cost edges**	**Total cost**
a	(a, d), (a, b)	5
b	(a, b), (b, e)	6
c	(c, b), (c, a)	8
d	(d, a), (d, c)	7
e	(e, b), (e, f)	9

Thus a lower bound on the cost of any tour

$$= \frac{1}{2}(5 + 6 + 8 + 7 + 9) = 17.5$$

A solution Tree for a TSP instance: (edges are considered in lexicographic order): See Figure 8.18

Figure 8.18: A solution tree for a TSP instance


All tours
tours with ab
tours without ab
tours with ab and ac
tours with ab but not ac
tours with ac but not ab
tours without ac and without ab
tours with ab,ac and not ad
tours with ab,ad and not ac
tours with ab, not ac not ad

- Suppose we want a lower bound on the cost of a subset of tours defined by some node in the search tree.
  In the above solution tree, each node represents tours defined by a set of edges that must be in the tour and a set of edges that may not be in the tour.
- These constraints alter our choices for the two lowest cost edges at each node. e.g., if we are constrained to include edge (a, e), and exclude (b, c), then we will have to select the two lowest cost edges as follows:

a	(a, d), (a, e)	9
b	(a, b), (b, e)	6
c	(a, c), (c, d)	9
d	(a, d), (c, d)	7
e	(a, e), (b, e)	10

Therefore lower bound with the above constraints = 20.5

- Each time we **branch**, by considering the two children of a node, we try to infer additional decisions regarding which edges must be included or excluded from tours represented by those nodes. The rules we use for these inferences are:

  1. If excluding (*x*, *y*) would make it impossible for *x* or *y* to have as many as two adjacent edges in the tour, then (*x*, *y*) must be included.
  2. If including (*x*, *y*) would cause *x* or *y* to have more than two edges adjacent in the tour, or would complete a non-tour cycle with edges already included, then (*x*, *y*) must be excluded.

- When we branch, after making what inferences we can, we compute lower bounds for both children. If the lower bound for a child is as high or higher than the lowest cost found so far, we can ``prune" that child and need not consider or construct its descendants.
  Interestingly, there are situations where the lower bound for a node *n* is lower than the best solution so far, yet both children of *n* can be pruned because their lower bounds exceed the cost of the best solution so far.
- If neither child can be pruned, we shall, as a heuristic, consider first the child with the smaller lower bound. After considering one child, we must consider again whether its sibling can be pruned, since a new best solution may have been found.

b. Give a template for a generic backtracking algorithm. What are the additional features required in branch-and-bound when compared to backtracking?

Answer:

```
ALGORITHM Backtrack(X[1..i])
  //Gives a template of a generic backtracking algorithm
  //Input: X[1..i] specifies first i promising components of a solution.
  //Output: Alll the tuples representing the problem's solutions
  If X[1..i] is a solution write X[1..i]
  else
    for each element x belongs to Si+1 consistent with X[1..i] and constraints do
      X[i+1] <- x
      Backtrack(X[1..i+1]
```

Backtracking
[1] It is used to find all possible solutions available to the problem.
[2] It traverse tree by DFS(Depth First Search).
[3] It realizes that it has made a bad choice & undoes the last choice by backing up.
[4] It search the state space tree until it found a solution.
[5] It involves feasibility function.

Branch-and-Bound (BB)
[1] It is used to solve optimization problem.
[2] It may traverse the tree in any manner, DFS or BFS.
[3] It realizes that it already has a better optimal solution that the pre-solution leads to so it abandons that pre-solution.

[4] It completely searches the state space tree to get optimal solution.
[5] It involves bounding function.

<u>Text Book</u>

Introduction to The Design & Analysis of Algorithms, Anany Levitin, Second Edition, Pearson Education, 2007