

Unit 7

Decrease and Conquer

Structure:

- 7.1 Introduction
 - Objectives
- 7.2 Concepts of Decrease and Conquer
- 7.3 Insertion Sort
 - Algorithm
 - Best, worst and average cases
 - Advantages of insertion sort
- 7.4 Depth-First search
 - Efficiency of Depth-first search
 - Application of Depth-first search
- 7.5 Breadth-First Search
 - Application of Breadth-first search
- 7.6 Topological Sorting
 - Algorithm
 - Uniqueness
- 7.7 Algorithm for Generating Combinatorial Objects
 - Generating permutations
- 7.8 Summary
- 7.9 Glossary
- 7.10 Terminal Questions
- 7.11 Answers

7.1 Introduction

By now, you should be familiar with the divide and conquer algorithm. This unit explains the concepts of 'decrease and conquer' and the methodology it uses in various algorithms.

Decrease and conquer is a method by which we find a solution to a given problem based upon the solution of a number of problems. The principle idea of decrease and conquer algorithm is to solve a problem by reducing its instance to a smaller one and then extending the obtained solution to get a solution to the original instance. Here, the problem instance is reduced by decreasing its size to a constant. So, the difference between decrease-and-conquer and divide-and-conquer is that it makes the problem instance

smaller by decreasing its size by a constant, while divide-and-conquer usually divides the problem into a number of smaller but similar sub-problems.

Objectives:

After studying this unit you should be able to:

- define decrease and conquer methodologies
- explain the algorithm used for insertion sort, Depth-first search, Breadth-first search and topological sorting
- analyze the algorithm for generating combinatorial objects

7.2 Concepts of Decrease and Conquer

Let us study the basics of decrease and conquer.

Decrease and conquer is a concept wherein larger solutions for problems are broken down based upon the solution to a number of smaller problems. The solution to the original instance is found out by extending the solution of the smaller instance. Decrease and conquer can be implemented by a top-down or a bottom-up approach. It is also known as incremental approach.

The three main distinctions in decrease and conquer are:

- 1) **Decrease by a constant** – In this kind of method, the size of the instance is made smaller by the same constant in all iterations. The constant is generally equal to one. For example, x^{10} can be computed as:

$$x^{10} = x^9 \cdot x$$

The general formula for this is:

$$x^p = x^{p-1} \cdot x$$

If the function $a(p) = b^p$, then we can use a top-down recursive approach and express this as:

$$\begin{aligned} a(p) &= a(p-1) \cdot b \text{ if } p > 1 \\ \text{and } a(p) &= b \text{ if } p = 1 \end{aligned}$$

Using bottom up approach we multiply b by p-1 times of b. We can say that:

$$a(p) = b(p-1) \text{ times } b$$

A few applications of decrease by constant are:

- Graph traversal algorithms (DFS and BFS)
- Topological sorting
- Algorithms for generating permutations, subsets

2) Decrease by a constant factor – The size of a problem instance is reduced by some constant factor in all iterations of the algorithm.

For example: we can compute $x^8 = x^4 \cdot x^4$

If we consider this formula as an example:

$$x(n) = (x^{p/2}) * (x^{p/2}) \quad \text{if } p \text{ is even eg: } 2^8 = 2^{8/2} * 2^{8/2}$$

$$x(n) = (x^{(p+1/2)}) * (x^{(p-1/2)}) \quad \text{if } p \text{ is odd eg: } 2^9 = 2^{10/2} * 2^{8/2} = 2^5 * 2^4$$

$$x(n) = x \quad \text{if } p = 1$$

The application of decrease by a constant factor is found in binary search where an element is divided into two sub-arrays and only one sub-array would be considered to sort the numbers. Hence, the problem instance is reduced to half the size here.

3) Variable size decrease method - In variable size decrease method, the outline of size reduction will vary from one algorithm to another. Euclid's algorithm is one of the examples for this method.

Self Assessment Questions

1. Decrease and conquer can be implemented by a _____ or _____ approach.
2. Decrease and conquer is also known as _____ approach.
3. Decrease and conquer is a method by which we find a solution to a given problem based upon the _____ of a number of problems.

7.3 Insertion Sort

In the previous section, we studied about the concepts of decrease and conquer. In this section we will study about insertion sort, its working and also the advantages of insertion sort.

Insertion sort is a simple algorithm that implements the decrease and conquer methodology. Insertion sort executes in $O(n^2)$ time, but it's about twice as fast as the bubble sort and somewhat faster than the selection sort in normal situations. It's also not too complex, although it's slightly more

complex than the bubble and selection sorts. It's often used as the final stage of more sophisticated sorts, such as quick sort.

There are numerous approaches to sort an unsorted array in insertion sort like, start sorting the array from:

- The left
- The right
- Any desired place where the marker is placed.

7.3.1 Algorithm

The basic operation of the insertion sort algorithm is the comparison $A[j] > v$ because the working of the algorithm will be much faster than the computed results.

Algorithm for Insertion Sort

```
ALGORITHM Insertion sort(A[0.....n-1])
//sorts a given array by insertion sort
//Input: An array A[0...n-1] of n orderable elements
//Output: Array A[0...n-1] sorted in non decreasing order
//Let the array contain 6 elements
For i ← 1 to n-1 do
  v ← A[i]
  j ← i-1
  While j ≥ 0 and A[j] > v do
    A[j+1] ← A[j]
    j ← j-1
  A[j+1] ← v
```

Let us now trace the algorithm.

Algorithm Tracing for Insertion Sort

```
//Let us consider n=6 and A[n] = {12, 16, 8, 4, 2, 9}
For i ← 1 to 6-1 do
  v ← 16 // A[1]
  j ← 1-1//j=0
  While 0 ≥ 0 and A[0] > 0 do // condition is true as j = 0 and A[0] = 12
    A[0+1] ← A[0] //A[1]=12
    j ← j-1// j = -1
  A[0] ← 16
```

The number of key comparisons in this algorithm depends on the type of input given. In the worst case, $A[j] > v$ is executed the most number of times. For every $j = i-1$ to 0. Since $v = A[i]$, it happens if and only if $A[j] > A[i]$ for $j = i-1$ to 0. For the worst case input, we get $A[0] > A[1]$ (for $i=1$), $A[1] > A[2]$ (for $i=2$), ..., $A[n-2] > A[n-1]$ (for $i=n-1$). Hence, we can conclude that worst case input array will consist of decreasing values.

Example

Let us consider an example of cricket players lined up in random order to understand the significance of marker. Each one has a jersey with the numbering from 1 to 11, and that they are required to stand according to the numbers. It's easier to think about the insertion sort if we begin in the middle of the process, when the team is half sorted.

We also call this marker technique as partial sorting in stages. At this point there's an imaginary marker somewhere in the middle of the line. Let us consider the players to the left of this marker are partially sorted. This means that they are sorted among themselves; each one has a jersey with smaller number than the person to his or her left. However, the players aren't necessarily in their final positions because they may still need to be moved when previously unsorted players are inserted between them. The player where the marker is whom we will call the marked player, and all the players on his or her right, is as yet unsorted. To have a clear picture let us consider the figure 7.1 which represents the jersey numbers of the players, that is, {1, 10, 2, 4, 6, 8, 5, 11, 3, 7, 9}.

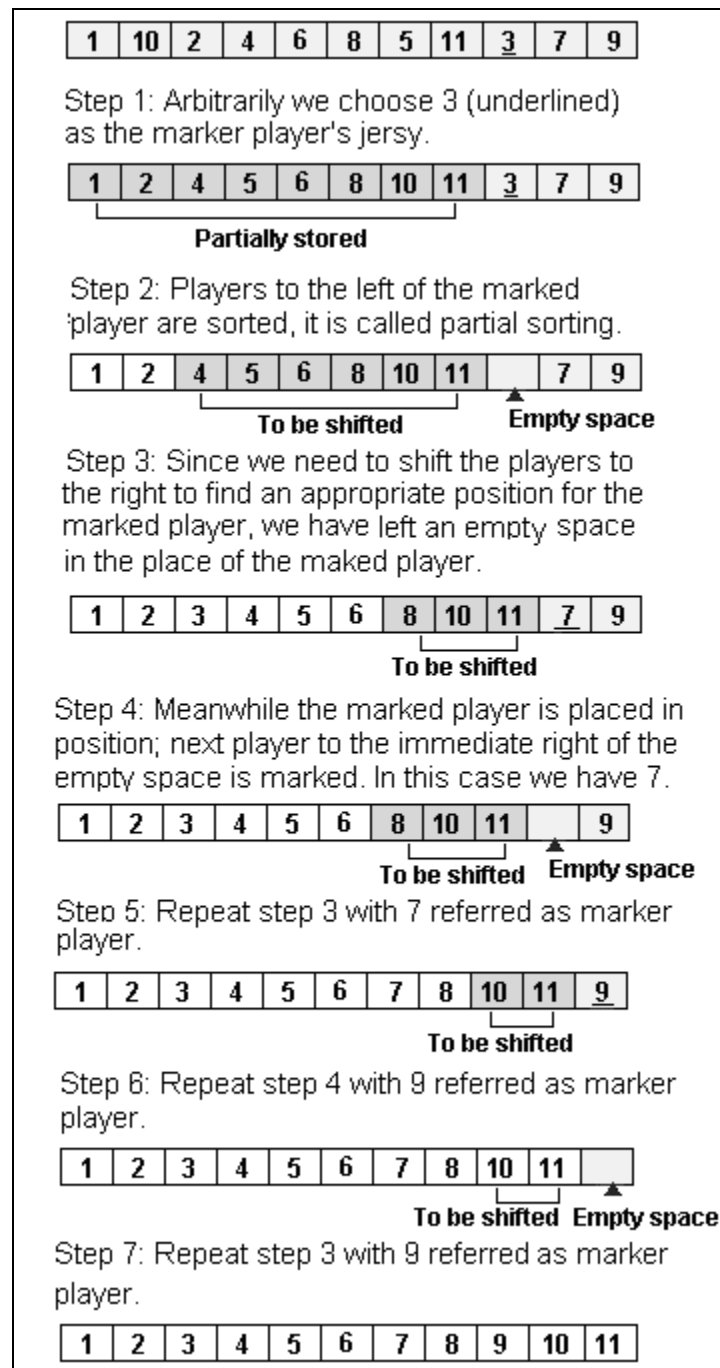


Figure 7.1: Insertion Sort

Similarly if we can choose to perform insertion sort either from the right or left end of the array, we will have to position the marker to the last player/last number.

7.3.2 Best, worst and average cases

The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., $O(n)$). Each cycle, the first remaining element of input is compared with the right-most element of the sorted subsection of the array.

The worst case input is an array sorted in reverse order.

In this case every cycle of the inner loop examines the entire sorted subsection of the array before inserting the next element. For this case insertion sort has a quadratic running time (i.e., $O(n^2)$).

The average case is also quadratic, which makes insertion sort unrealistic for sorting large arrays. However, insertion sort is the fastest algorithm for sorting arrays containing fewer than ten elements.

7.3.3 Advantages of insertion sort

The following are the advantages of insertion sort:

- It is simple to implement.
- The method is useful when sorting smaller number of elements.
- It is more efficient than other algorithms.
- It is a very stable algorithm.
- It is easy to understand.
- Hope you are clear about insertion sort and its working. Let us move on to the next section.

Self Assessment Questions

4. There are _____ major categories in insertion sort.
5. In insertion sort, the best case input is an array that is already _____.
6. To carry out an insertion sort, begin at the _____ most element of the array.

7.4 Depth-First Search

In this section we will study another decrease and conquer algorithm - Depth-first search. We will study the Depth-first search algorithm and its efficiency.

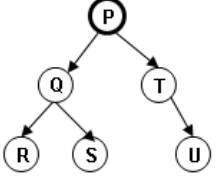
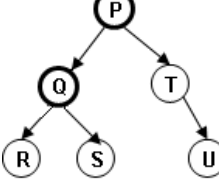
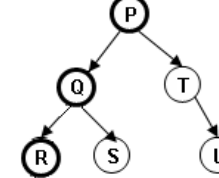
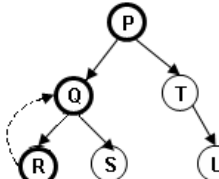
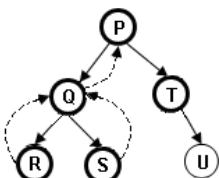
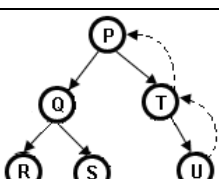
Depth-first search (DFS) is an algorithm for traversing graphs, and trees. One begins at the root (selecting some node as the root in the graph case) and then explore along each branch. Depth-first search works by considering vertices, checking its neighbors, extending the first vertex it finds across its neighbors, checking if that extended vertex is our destination, and if not, continue exploring more vertices.

Depth-first search starts exploring vertices of a graph at a random vertex and marks the vertices explored as visited. In each step, the algorithm moves to an unvisited vertex that is adjacent to the one recently visited. If there are many adjacent unvisited vertices, a tie would be solved randomly. The data structure which depicts the graph indicates which unvisited vertices are chosen. In our example we will always cut off ties in alphabetical order. This procedure will continue until a vertex which has no adjacent vertices is found. After this, the algorithm comes back to one edge of vertex that it came from and tries to go to other unvisited vertices from there. The algorithm finally stops after going back to the starting vertex. By now, all vertices in the same connected section as the starting vertex have been visited. If any unvisited vertices are found, Depth-first search must be begun again at any one of these vertices.

DFS takes out a vertex from the stack when it reaches the end. Figure 7.4 shows a simple Depth-first search traversal with vertices and their respective edges.

Example: Let us consider a sub-tree for performing Depth-first search as given in the table 7.1.

Table 7.1: Depth-First Search

	<p><i>Step 1:</i> Here we can see that vertex P will be visited first and marked. Now our list contains only P.</p>
	<p><i>Step 2:</i> The tree will be searched in depth that is the sub-trees corresponding to vertex P are searched. Hence, vertex Q is visited and marked. Now our list contains P, Q</p>
	<p><i>Step 3:</i> Here we visit vertex R and mark it. After this, sub-tree for vertex R is checked. Since there is no sub-tree present for vertex R, the algorithm comes back to vertex Q. Here, our list would remain as P, Q, and R</p>
	<p><i>Step 4:</i> Here the vertex S is visited, since it is a sub-tree of Q. Now, our list contains P, Q, R, S.</p>
	<p><i>Step 5:</i> As there is no sub-tree present for vertex S, algorithm traces back to vertex Q, and then back to vertex P. The algorithm then visits vertex T and marks it as visited. Now, our list contains P, Q, R, S, T.</p>
	<p><i>Step 6:</i> Here we can see that vertex U is visited since it is a sub-tree of T and the final list contains P, Q, R, S, T, and U. Then, the algorithm traces back to vertex T, and then to P again.</p>

The following pseudo code explains the Depth-first search algorithm.

Pseudo code for Depth-first Search

DFS (G)

// Implements a Depth-first search traversal of a given graph

// Input: Graph(G) = V(E)

// Output: Graph G with vertices marked with consecutive integers

// In the order they have been found by the DFS traversal

Mark each vertex in V as 0 as an indication of being unvisited

Count \leftarrow 0

For each vertex v in V do

If v is marked with 0

dfs(v)

// visits recursively all the unvisited vertices connected to vertex v by a path

// and numbers them in the order they are met

// via global variable count

Count \leftarrow count+1; mark v with count

For each vertex w in v adjacent to v do

If w is marked with 0

dfs(w)

7.4.1 Efficiency of Depth-first search

The Depth-first search algorithm is quite efficient because it just takes time proportional to the size of the data structure which is used for depicting the data structure for the question here. The traversal time is $O(v^2)$ for a tree; a depth-first search may take an excessively long time to find even a very nearby goal node. There are two loops in the algorithm; the first one loops through all vertices and the second loops through all neighbors of a particular vertex. All other operations performed within these loops, such as changing times, are considered $O(1)$. The time spent on assignment statements in the outer loop is $O(|V|)$. The loop through all neighbors of vertices takes a check at other sub-trees. A useful aspect of the DFS algorithm is that it traverses through the connected components visiting one at a time, and thus it can be used to identify the connected components in a given graph.

$$O\left(\sum_u \# \text{neighbors}(u) = 2 |E| \right)$$

Therefore the total complexity is $O(|V| + |E|)$, which is optimal, as mentioned in the introduction. V and E are the number of graph vertices, and edges respectively.

7.4.2 Applications of Depth-first search

We can use the Depth-first search algorithm for the path finding application. In the path finding application we can find the path between vertices a and b . Here we use a stack S to keep track of the visited nodes. While we traverse and reach b , the stack returns all its elements to form a path from a to b .

Depth-first search algorithm is commonly used for the following:

- To find the path from one vertex to another
- To solve connected or unconnected graphs
- To compute a spanning tree for a connected graph by using a backtracking technique
- To check connectivity and acyclicity of graphs
- To find the articulation point

Self Assessment Questions

7. DFS uses the _____ technique.
8. It is easier to use a _____ to trace the working of a depth-first search.
9. Depth-first search starts exploring vertices of a graph at a _____ vertex.

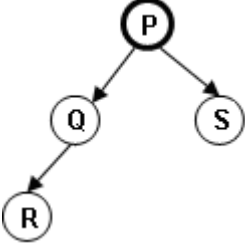
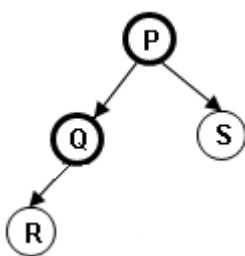
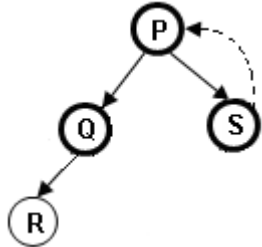
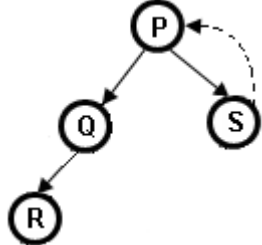
7.5 Breadth-First Search

In the previous section we studied about the Depth-First Search algorithm. In this unit we will study about the Breadth-first search algorithm and its working.

Breadth-first search (BFS) is an algorithm which travels in such a way that all vertices are visited along every sub-tree and which are adjacent to the starting vertex, then all the unvisited vertices will be visited which are a part of it. This process continues until all vertices in the same section as the starting vertex are visited. If unvisited vertices remain, the algorithm will begin considering any random vertex which is connected to the graph.

The data structure which is used to track the working of Breadth-first search is a queue. The table 7.2 depicts the working of Breadth-first search. Here, Breadth-first search is performed level by level.

Table 7.2: Breadth-First Search

	<p><i>Step 1:</i> Here the vertex P would be marked first, here vertex P is considered to be at the first level. Our list now has only P.</p>
	<p><i>Step 2:</i> The vertex Q would be visited and marked. Here, vertex Q is at second level. Now our list contains P, Q.</p>
	<p><i>Step 3:</i> Here vertex S would be visited and marked. Here, vertex S is at second level of the tree. After marking vertex S, the algorithm returns back to the initial vertex</p>
	<p><i>Step 4:</i> The vertex R is visited which is at the third level. Our final list would consist of P, Q, R, and S.</p>

The total running time of Breadth-first search algorithm is given as $O(V+E)$. Where V is the set vertices and E is the set of edges.

Let us now discuss the applications of Breadth-first search algorithm.

7.5.1 Application of Breadth-first search

Breadth-first search is commonly used for the following:

- To test the connectivity of a graph
- To compute the spanning forest of a graph
- To check for and compute cycles of a graph
- To check for a path with minimum number of edges between the start and the end vertex
- To check connectivity and acyclicity of graphs

Activity 1

Draw a Breadth-first search graph traversal for the Depth-first search figure given.

Self Assessment Questions

10. The data structure which is used to track the working of Breadth-first search is a _____.
11. Breadth-first search is an algorithm which travels in such a way that all vertices are finished along every _____.
12. The data structure which is used to track the working of Breadth-first search is a _____.

7.6 Topological Sorting

In the previous section, we studied about the Breadth-first search algorithm and its working. In this section we will study about topological sorting, its examples and also the uniqueness which it possesses.

Topological sort is done using a directed acyclic graph (DAG), which is a linear ordering of all vertices $G = (V, E)$ is an ordering of all vertices such that if G contains an edge (u, v) , then u appears before v in the ordering. A topological sort of a particular graph can be looked upon as a horizontal line where all directed edges travel from left to right. Thus, topological sort differs from the usual sorting technique. Linear ordering cannot be performed for cyclic graphs. Topological sorting can be used for scheduling a sequence of jobs or tasks. The jobs are denoted by vertices, and there is an edge from a to b , if job a must be completed before job b can be started (for example, when starting a car, the ignition must be started first). Therefore a topological sort gives an order in which to perform the jobs.

Directed acyclic graphs are used in many instances. They usually indicate precedence among events. Topological sorting can be used to arrange tasks under precedence constraints. Suppose we have a set of tasks to do, but particular tasks have to be performed before other tasks, we can use these precedence conditions to form a directed acyclic graph (DAG). Any topological sort (also known as a linear extension) describes an order to perform these tasks such that each task is performed only after all of its conditions are satisfied.

Example

Let us consider a set of five required assignments that a student has to complete which are named as S1, S2, S3, S4, and S5. The assignments should be completed within some time as long as certain conditions are met. S1 and S2 have no conditions to be met. S3 requires S1 and S2 to be completed. S4 requires S3 to be completed and S5 needs S3 to be completed. The student can complete only one assignment in a day. The order in which the student takes up a course is the main concern here. Figure 7.2 illustrates the assignments of the students represented as a DAG.

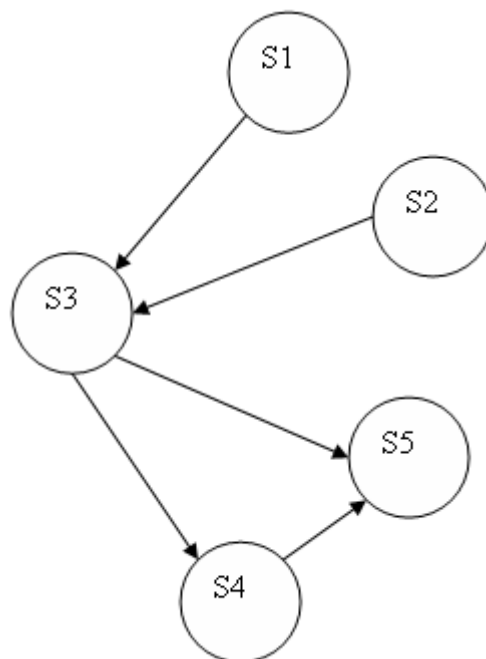


Figure 7.2: The Five Assignments of Students Represented by a DAG

In the above figure, the vertices represent the assignments to be completed by the student and the directed edges show the pre-required conditions to be met. The main concern here would be whether ordering of vertices should be possible in such a way that every vertex where the edge begins must be listed before a vertex where the edge closes. This concept is called as topological sorting. If the di-graph has a directed cycle present, the problem will not have a solution.

7.6.1 Algorithm

One of the algorithms for topological sort was first described by Kahn (1962). It proceeds by selecting vertices in the same order as the eventual topological sort. For topological sorting to be performed a graph should be a DAG.

We can see that a DAG also need not necessarily perform topological sorting. If a graph has no directed cycles, topological sorting for that case will have a solution. Depth-first search algorithm can be used to apply topological sorting. We can note how the traversal is performed here. We have to note how the traversal is performed until it reaches the dead end.

An algorithm which can be applied here uses the decrease by one technique. Here; we identify a vertex which has no incoming edges, and remove all the edges which are coming out of it. These steps have to be checked visually in order to ensure that topological sorting is correctly performed.

Small examples of topological sorting might prove to be wrong sometimes. But, if we consider situations where big projects are involved, the pre-specified conditions must be taken care such that they occur in sequence and the project is successfully completed. For this to happen, topological sorting must be performed using a di-graph. After this step, we can decide about the situations that occur while performing topological sorting.

First, find a list of start nodes which doesn't have incoming edges and insert them into a set A; at least one such node must exist if graph is acyclic.

Let us now discuss the pseudo code for topological sorting.

Pseudo code for topological sorting

```
//L ← Empty list that will contain the sorted elements
//S ← Set of all nodes with no incoming edges
While S is non-empty do
    remove a node n from S
    insert n into L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has edges then
    output error message (graph has at least one cycle)
else
    output message (proposed topologically sorted order:L)
```

If the graph was a DAG, a solution is contained in list L, else, the graph has at least one cycle and therefore a topological sorting is not possible.

7.6.2 Uniqueness

If a topological sort has the property that every pair of consecutive vertices in the sorted order is linked by edges, then these edges form a directed Hamiltonian path. A Hamiltonian path, also called a Hamilton path, is a path between the vertices of a DAG that visits each vertex exactly once. If a Hamiltonian path is present, the topological sort order is unique. On the other hand, if a topological sort does not form a Hamiltonian path, the DAG will have two or more valid topological orderings, in this case it is always possible to form a second valid ordering by swapping two consecutive vertices that are not connected by an edge to each other. Therefore, it is possible to examine in polynomial time whether a unique ordering exists, and whether a Hamiltonian path exists.

Self Assessment Questions

13. Topological ordering of a _____ is a linear ordering of its nodes.
14. In topological sorting the jobs are denoted by _____.
15. Being a DAG is also a necessary condition for _____ sorting to be possible.

7.7 Algorithms for Generating Combinatorial Objects

In the previous section, we studied about topological sorting and its uniqueness. In this section we will explain the algorithms for generating combinatorial objects.

The most essential types of combinatorial objects are permutations and combinations and subsets of a particular set. Combinatorial objects are given importance in a mathematical subject called as combinatorics.

They usually arise in problems that require a consideration of different choices such as Travelling Salesman problem (TSP), Knapsack problem and so on.

To solve these problems we need to generate combinatorial objects.

7.7.1 Generating permutations

The set of numbers to be permuted are the number of integers which range from 1 to n .

Let us presume the set whose elements need to be permuted is the set of integers from 1 to n . They can be interpreted as indices of elements in an n -element set $\{a_1 \dots a_n\}$

Now, let us see how a decrease-by-one technique solves the problem of generating all $n!$ Permutations:

Approach:

1. The smaller-by-one problem is to generate all $(n-1)!$ Permutations.
2. Assume that the smaller problem is solved.
3. We can get a solution to the larger problem by inserting n in each of the n possible positions among elements of every permutation of $n-1$ elements. The order of insertions depends on the total number of all permutations which is $n \cdot (n-1)! = n!$

We can insert n in the previously generated permutations:

- left to right
- right to left

<i>start</i>	1					
<i>insert 2</i>	12	21				
	<i>right to left</i>					
<i>insert 3</i>	123	132	312	321	231	213
	<i>right to left</i>			<i>left to right</i>		

Example: Traveling salesman problem

A difficulty which is closely connected to the topic of Hamiltonian circuits is the Traveling-Salesman problem stated as follows: A sales person is required to go to a certain number of cities in a trip. The distances between cities are provided. The order in which he has to travel to every city and return home with minimum distance covered to each city is the main concern here.

The cities are represented by vertices and the roads between them are shown as edges. A graph is obtained by doing this. In this graph, with every edge s_i , there is a real number associated with it. The distance is measured in miles here. This graph is called as a weighted graph; $w(s_i)$ being the weight of edge s_i .

If each of the cities is connected to another city by a road a complete weighted graph is obtained. The graph has many Hamiltonian circuits, and we must choose the one that has the smallest sum of distances or weights. The total number of different Hamiltonian circuits excluding the disjoint ones in a complete graph of x vertices can be shown to be $(x-1)! / 2$. This shows from that starting from any vertex we have $x-1$ edges to pick from the first vertex, $x-2$ from the second, $x-3$ from the third, and so on. These being individual, results with $(x-1)!$ choices. However, this number is, divided by 2, because each Hamiltonian circuit is counted twice.

Theoretically, the traveling salesman problem can always be solved by specifying all $((x-1)!)/2$ Hamiltonian circuits, finding out the distance traveled in each, and then choosing the shortest one. However, for a large value of x , the cost and time taken is very high even for a digital computer.

The problem is to lay down a convenient algorithm for computing the shortest route. Although many attempts have been made, there is no

efficient algorithm for problems of random size that has been found till date. Since this problem has applications in operations research, some specific large-scale examples have been worked out. There are also few methods available that are suggested by expert mathematicians, but the shortest possible path which is perfect has yet to be invented

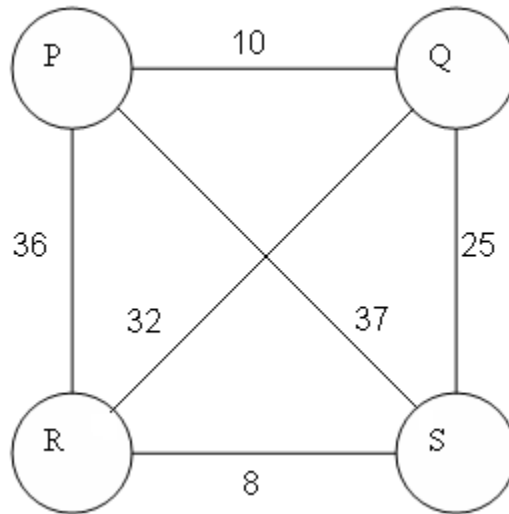


Figure 7.3: Travelling Salesman Problem with 4 Cities

Figure 7.3 shows how travelling salesman problem is solved between four cities – P, Q, R, and S. The number above the edges indicates the distance between the cities. Here, distance between two cities is same in each direction which forms an undirected graph. The minimum cost for the cities travelled would be $P \rightarrow R \rightarrow S \rightarrow Q \rightarrow P = 79$.

Hope you are clear about algorithms for generating combinatorial objects.

Self Assessment Questions

16. Theoretically, the traveling salesman problem can always be solved by specifying all _____ Hamiltonian circuits.
17. The cities are represented by _____ and the roads between them are shown as edges.
18. Each of the cities is connected to another city by a road a complete _____ is obtained.

7.8 Summary

Let us summarize what we have studied in this unit.

Decrease and conquer is a method by which we find a solution to a given problem based upon the solution of a number of smaller problems. The principle idea of decrease and conquer algorithm is to solve a problem by reducing its instance to a smaller one and then extending the obtained solution to get a solution to the original instance.

Insertion sort is one of the simplest algorithms that is implemented under the decrease and conquer methodology. In this technique the element is inserted at the suitable position.

Depth-first search (DFS) is an algorithm for searching a tree, tree structure, or graph. We begin at the root and then explore along each branch.

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, it explores their unexplored neighbour nodes, and so on, until it finds the goal.

A topological sort of a particular graph can be looked upon as a horizontal line where all directed edges travel from left to right. Thus, topological sort differs from the usual sorting technique. Topological sorting can be used for scheduling a sequence of jobs or tasks.

Algorithms for generating combinatorial objects define most important types of combinatorial objects such as permutations and combinations.

7.9 Glossary

Term	Description
Decrease-by-one technique	Type of decrease and conquer, which can also be termed as Decrease by constant.
Combinatorics	Mathematical subject which involves solving problems using figures.

7.10 Terminal Questions

1. Define the concept of decrease and conquer algorithm?
2. What are the best, worst and average cases in an insertion sort?

3. What are the major advantages of insertion sort?
4. Explain the working of DFS algorithm with example
5. Describe BFS algorithm.
6. Explain the travelling salesman problem in brief.

7.11 Answers

Self Assessment Questions

1. Top down or Bottom-up
2. Incremental
3. Solution
4. Two
5. Sorted
6. Left
7. Backtracking
8. Time and space
9. Backtracks
10. Graph
11. Uninformed
12. Goal
13. Directed acyclic graph
14. Vertices
15. Topological
16. Combinatorial
17. Subset
18. Decrease-by-one

Terminal Questions

1. Refer section 7.2 – Concepts of Decrease and Conquer
2. Refer section 7.3.2 – Best, worst and average cases
3. Refer section 7.3.3 – Advantages of insertion sort
4. Refer section 7.4- – Depth-first search
5. Refer section 7.5 – Breadth-first search
6. Refer section 7.7.1 – Generating permutations

References

- Puntambekar, A. A. (2008). *Design and Analysis of Algorithms*, First edition, Technical publications, Pune.
- Anany Levitin (2008). *The Design & Analysis of Algorithms*, Second edition, Pearson.
- Pandey, Hari Mohan (2008). *Design Analysis and Algorithm*, First edition. University science press, New Delhi.
- Thomas H. Cormen (2001). *Introduction to Algorithms*, Second edition. Mc Graw Hill.

E-References

- <http://www.cs.sunysb.edu/~algorithm/files/topological-sorting.shtml>