Algorithms

8. Algorithm Design Techniques

R K Shyamasundar

R K Shyamasundar is Professor of Computer Science at TIFR, Mumbai and has done extensive research in various foundation areas of computer science.

Previous articles of this series were:
1. Introduction to algorithms, January 1996.
2. The while–construct, March 1996.
3. Procedures and recursion, June 1996.
4. Turtle graphics, September 1996.
5. Data types and their representation in memory, December 1996.
6. Algorithms for Sorting and Searching, March 1997.
7. Data structures: lists, queues, stacks and arrays, June 1997.

This article continues with the exploration of common algorithm design techniques. In particular, we discuss balancing, greedy strategy, backtracking or traversal of trees and the dynamic programming strategy.

Introduction

In the previous articles, we have discussed various common data-structures such as arrays, lists, queues and trees and illustrated the widely used algorithm design paradigm referred to as 'divide-and-conquer'. Although there has been a large effort in realizing efficient algorithms, there are not many universally accepted algorithm design paradigms. In this article, we illustrate algorithm design techniques such as balancing, greedy strategy, dynamic programming strategy, and backtracking or traversal of trees that are commonly used in the efficient design of algorithms.

Algorithm Design Techniques

In the previous article, we demonstrated how the divide-and-conquer strategy can be used to design efficient algorithms. In this section, we discuss refinements of the divide-and conquer technique and other strategies that will be useful in the design of algorithms.

In the illustration of the application of the divide-and-conquer method for arriving at efficient algorithms, we had seen that the problems were partitioned into sub-problems of *equal* size. Maintaining the balance among the sub-problems is a general guideline in the design of efficient algorithms. This principle

referred to as *balancing* is illustrated through an example on sorting.

Maintaining the balance among the sub-problems is a general guideline in the design of efficient algorithms. This principle is referred to as *balancing*.

Let us consider the problem of sorting a sequence of numbers in a non-decreasing order. The basic *insertion sort* algorithm discussed earlier provides a solution to the problem. The method consists of locating the minimum element, say m, in the sequence and exchanging m with the first element, say f, in the sequence. The process is repeated by finding the second smallest element, and so on. We have already shown that the time-complexity of the algorithm is $O(n^2)$ where n is the number of elements in the sequence.

The *insertion sort* algorithm can indeed be treated as a recursive algorithm that divides the problem into two subproblems of size 1 and $n-1$ (i.e., unequal size). Now, let us consider a method of dividing the problem into nearly equal sizes and analyze its efficiency.

For simplicity, let the sequence of integers to be sorted be a_1, ..., a_n and for the sake of convenience, let us assume that n is a power of 2 (i.e., $n = 2^k$ for some k). Intuitively, the algorithm corresponds to partitioning the sequence into two halves a_1, ..., $a_{n/2}$ $a_{(n/2)+1}$, ..., a_n, sorting each of the sequences and *merging* the two sorted sub-sequences. The process of *merging* corresponds to combining the two sorted sub-sequences into one sorted sequence. It is for this reason that the method is referred to as the *mergesort*. The program for mergesort is described in *Table 1*.

The program for mergesort uses procedure MERGE for merging any two sorted arrays; the first two parameters refer to the first array and the last two parameters refer to the second array. Basically, the smallest of the first elements of the two arrays are found and placed in the array with the merged array. The process of placing the smallest of the first two elements of the arrays is repeated till all the elements of the two arrays are

Table 1 Mergesort.

```
program MAIN;
    var A: array [1..N] of integer; (* Given list to be sorted *)
      ********** Description of SORT Procedure **********
        (* For simplicity, we use array indices as parameters rather than the array *)
      procedure SORT(i,j: integer);
        var mid: integer;
        begin
          if i = j then halt (* No need to sort *)
            else
              begin
                mid := (i+j - 1)/2 (* Get the mid point *)
                SORT( i, mid);
                SORT( mid+1, j);
                MERGE( i,mid, mid+1, j); (* merge the sorted lists *)
              end
          endif
        end
        ********** End of procedure SORT **********
  SORT(1,N); (* sort the given array *)
end (* end of the program *)
```

exhausted. If one of the arrays gets exhausted before the other, then the merging essentially corresponds to extending the merged array with the elements of the array that is not yet exhausted. The program for merging two arrays is shown in *Table* 2 . For the sake of clarity, the actual program code shown is simplified. The reader should be able to derive the procedure MERGE required for the SORT procedure described in *Table 1* by suitable modification of the parameters and declarations of the procedure MERGE_TWO_ARRAYS.

The steps executed by the mergesort program on a sample input are shown in *Figure 1*. The merged sequence from the two branches are shown enclosed in boxes.

Let us analyze the time complexity of the mergesort algorithm.

Table 2 Merging two sorted arrays.

```
procedure MERGE_TWO_ARRAYS(A[1,p], B[1,q], C[1,p+q]:integer);
   (* A[1,p], B[1,q] are the sorted arrays to be merged and placed in array C. *)
   (* Note that array C will be of length p+q; in the program we use parameters *)
   (* p and q explicitly *)
  var i, j, k: integer;
  begin
    i:=1; j:=1;
    for k:= 1 to p+q do
        if i ≤ p and j ≤ q then
          (* both arrays still have elements *)
            if (A[i] < B[j]) then
                begin
                        C[k]:= A[i];
                        i:=i+1;
                end
            else
                begin
                        C[k]:= B[j];
                        j:=j+1
                end
            endif
            else (* Array A still has elements *)
            if i ≤ p then
                begin
                        C[k]:= A[i];
                        i:=i+1;
                end
          else (* Array B still has elements and j ≤ q holds *)
                begin
                        C[k]:= B[j];
                        j:=j+1
                end
          endif
      endif
    endfor
  end
```

The arrays are partitioned into two equal halves till each partition is a single element. Partitions (i.e., an array) of single elements require no computation. For merging two arrays of sizes m and

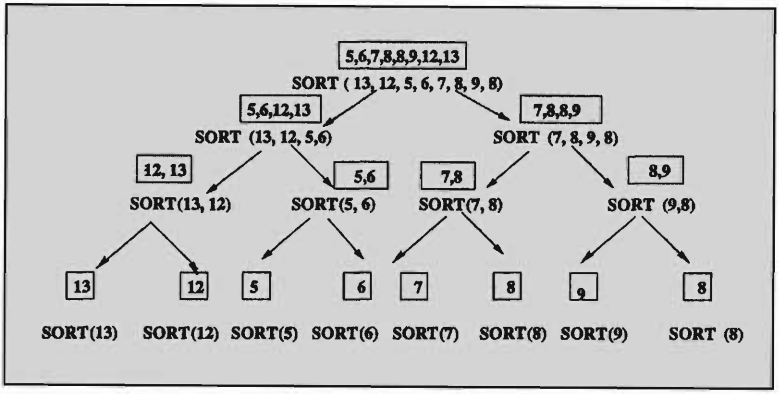


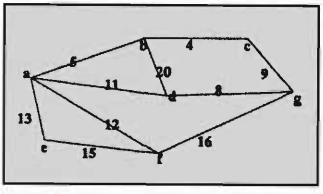
Figure 1 Algorithmic steps for the mergesort .

n, one requires a computation proportional to $m+n$ as we need to scan each of the arrays. Since an array of size n (in the case given above, n is a power of two) can be recursively divided by two, $\log_2 n$ times, it can be easily seen that the worst case complexity is given by $O(n \log_2 n)$. Thus, we have gained considerably in efficiency by balancing the partitions. In other words, *balancing* the subproblems pays off handsomely (more so, for large values of n).

Greedy Strategy

Consider the problem of laying roads between various cities. Abstractly, let us consider an undirected graph where nodes or vertices represent cities; and undirected edges between cities denote roads; and the number over the edges denote the cost of laying the road between the two cities. Our main task is to reduce the total cost and maintain the link between the cities. There is no need to have a direct road between all cities; however, all the cities should be connected among themselves. The graph shown in *Figure 2* can be interpreted to denote such a network.

Figure 2 Network of cities.



To arrive at an optimal cost is to arrive at a tree that spans all the cities such that the sum of the cost of all the edges is the minimum. An algorithm for arriving at such a solution is reflected by the method usually referred to as the *Greedy Method*. Basically, the method starts with the *best* choice and recursively goes

on finding the *new best candidates* till the given domain is completely covered (i.e., exhausted). In a sense, the approach has the attitude *enjoy today as there may not be a tomorrow*. In other words, the best-looking choice from among the set of possibilities at each stage leads to the required solution. The important feature is that one chooses the best candidate locally at each stage. Let us apply this intuition for arriving at a solution for the above problem. The solution has basically the following steps:

The Greedy Method starts with the *best* choice and recursively goes on finding the *new best candidates* till the given domain is completely covered.

- Start with the simplest tree having the minimum cost.
- At each stage extend the tree by adding the cheapest edge not yet considered with the restriction that only one end lies in the tree considered already (and hence, no cycle will be formed). *That is, we choose the best option given the local situation.*
- Repeat the second step till we get a tree that has all the nodes.

Figure 3 illustrates the procedure applied on the graph shown in *Figure 2*. It can be shown easily that the above procedure results in the *minimum spanning tree*.

The greedy strategy is applicable for a large spectrum of

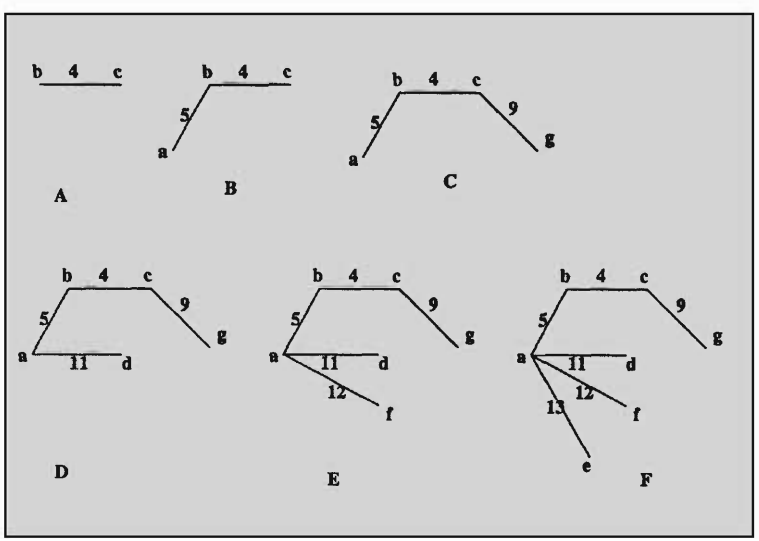


Figure 3 Successive steps in getting a minimal spanning Tree.

combinatorial problems. One of the most important aspects of this technique is that one can often arrive at an algorithm quite easily. But it is difficult to establish that the algorithm derived indeed works.

Dynamic Programming Approach

We have already seen that recursive techniques divide the problem into subproblems recursively till we reach a subproblem of reasonable size. The method leads to an efficient algorithm, provided the size of the original problem and the sum of the sizes of the subproblems in some sense compare reasonably (in computer science, we take polynomial relations to be reasonable and exponential relations to be unreasonable). For instance, suppose we divide a problem of size n into n problems of size $n-1$, it can be easily seen that the recursive technique results in an exponential growth and hence, becomes inefficient. Under such circumstances, we resort to an iterative method or a tabular method referred to as the *dynamic programming* strategy. Note that the approach is a non-greedy strategy. In fact, the approach can be considered as a refinement of the *local best choice* of the greedy method. Obviously, in arriving at the optimal solution, one needs to go through a sequence of choices. Even though simply looking for the best local choice (as in the simple greedy case) may not lead to the optimal sequence of choices, it is very likely that an optimal sequence of choices can indeed be obtained by considering all possible combinations of making a choice; and then finding an optimal subsequence of the remaining choice. This is the guiding principle of this strategy. This strategy has the following computation structure.

The *dynamic programming* strategy can be considered as a refinement of the *local best choice* of the greedy method.

- The method is bottom-up, in the sense, that it starts from solutions of small subproblems and builds up solutions to large problems.
- The answers to the subproblems are stored in a table.
- Whenever a subproblem is encountered, one checks whether it has already been solved. If so, one uses the

solution already computed rather than recomputing it. This is one of the distinct advantages of this approach.

Illustrative Example: Let us consider computing the product of r matrices:

$$A = A_1 \times \ldots \times A_r ,$$

where A_i is of dimension $n_{i-1} \times n_i$ The complexity of the computations depends on the order of evaluating the product of the matrices. This can be seen by considering the evaluation of

$$B = B_1[10, 20] \times B_2[20, 30] \times B_3[30, 1] \times B_4[1, 10],$$

where $B_i\ [n_1, n_2]$ denotes matrix B_i of dimension $n_1 \times n_2$. From the earlier article, it can be easily seen that multiplication of two matrices $M_1[p,q] \times M_2[q,r]$ requires $p \times q \times r$ operations. Thus, evaluating $(B_1[10, 20] \times (B_2[20, 30] \times (B_3[30, 1] \times B_4[1, 10])))$ requires $(30 \times 1 \times 10) + (20 \times 30 \times 10) + (10 \times 20 \times 10) = 8300$ operations whereas evaluating $((B_1\ [10, 20] \times (B_2\ [20, 30] \times B_3[30, 1])) \times B_4[1, 10])$ requires $(20 \times 30 \times 1) + (10 \times 20 \times 1) + (10 \times 1 \times 10) = 900$ operations. Note that the multiplication of matrices is associative (and not commutative) and hence, we can choose any order.

If we want to select the best ordering that leads to the minimum number of operations, a naive way would be to enumerate all the orderings and choose the best. However, the number of choices are exponentially many and hence, it is certainly impractical. Using the dynamic programming approach, it is possible to obtain a polynomial time ($O(n^3)$) algorithm. We shall describe the approach in the sequel.

Let a_{ij} be the minimum cost of computing the product

$$A' = A_i \times A_{i+1} \times \ldots \times A_j,\ 1 \le i \le j \le r,$$

where the dimension of A_i is denoted by $n_{(i-1)} \times n_i$. The task of evaluating A', can be split into two parts: evaluate $(A_i \times A_{i+1} \times \ldots \times A_k)$ and $(A_{(k+1)} \times A_{(k+2)} \times \ldots \times A_j)$ where $i \le k \le j$ and

then compute the product of the resultant matrices. Now, we can see that

$$a_{ij} = min \ \{a_{ik} \ + a_{(k+1)j} + n_{(i-1)} \times n_k \times n_j) \mid i \leq k < j \},$$

where the cost of multiplying the two resultant matrices is $n_{i-1} \times n_k \times n_j$ Thus, we have to design an algorithm for computing a_{ij}'s. It can be easily seen that a_{ii} is zero for all i. The dynamic programming algorithm essentially computes a_{ij}'s starting from a_{ii}'s in order of increasing difference in the subscripts. The algorithm is described in *Table 3*. We can compute the values of the matrix A starting from the diagonal

Table 3 Algorithm for choosing the optimal multiplication order of matrices.

```
(* N is the number of matrices to be multiplied *)
   (*  A_i  is of dimension  d_{i-1} × d_i .
 type twotuple: array[1.. 2] of integer;
 var A: array [1.. N,  1.. N] of integer;
             (* twotuple is a two-tuple of integers *) (* d [i] stores the dimension of the a^th matrix *)
    d: array [1.. N] of twotuple;
    i, j, k, m, p, q: integer;
 begin
             (* Initialization *)
    for i := 1 to N do
       for j:= 1 to N do
           A[i,j] := 0
       endfor
    endfor;
    for m := 1 to N do
       for i := 1 to N-m do
          begin
             j:= i + m;
             for k:= i to j-1 do
                A[i, j] := min (A[i, j], (A[i, k] + A[k, j] + d_{i-1} [1] * d_k [1] * d_j [2]))
             endfor
          end
       endfor
    endfor
 end
```

value only (i.e., $A[i,i] = 0$ for all i); however, we have initialized all the elements of the array as we are using the operation *min* (X) (i.e., the minimum of the set of values X) in the algorithm.

From the nested 'for-loops', it can be observed that the number of operations is of the order of $O(N^3)$. The application of the above algorithm to the example discussed earlier leads to the cost table shown in *Table 4*.

A variation of the problem discussed earlier is the *minimum path* or the *shortest path* problem i.e., find the shortest path from a given *source* to all *destinations*. In finding a solution for this problem, the greedy strategy of finding the locally best solution does not work. This can be seen by applying the greedy method to the graph shown in *Figure 2* after changing the label between d and g to 2 instead of 8.

A solution to the above problem can be obtained as follows:

> Start with a set of vertices S whose shortest distances from the given source are known. At each step, add to S the vertex whose distance from the source is the shortest among the remaining vertices.

The reader is encouraged to write an algorithm for this problem. The attentive reader can see that this strategy is in fact a refinement of the divide-and-conquer strategy.

The above strategy is very useful in the multiplication of a chain of matrices (two dimensional matrices). Since this strategy considers all the possible solutions in a sense, it becomes complex for some classes of problems.

$a_{11}=0$	$a_{12}=6000$	$a_{13}=800$	$a_{14}=900$
	$a_{22}=0$	$a_{23}=600$	$a_{24}=800$
		$a_{33}=0$	$a_{34}=300$
			$a_{44}=0$

Table 4 Cost table for evaluating the product of matrices.

Searching Data Structures

From the various examples discussed so far, it must be clear that there is a need for searching exhaustively through the data which has been represented in some form. An explicit example we have considered is searching through the tree abstraction.

There are a variety of searching mechanisms for solutions in a tree. One of the most prominent is called the *Depth First Search* (DFS). The recursive steps of DFS for a binary tree are:

- Visit the root of the binary tree.
- Traverse the left subtree in DFS.
- Traverse the right subtree in DFS.

The DFS is also referred to as *preorder*. In DFS, we go down the level in the tree to the extent possible and then backtrack to the previous level. As an example consider searching in a maze that has: two possibilities at each branch, or no further branch at all with following markings: an exit door labeled SUCCESS (have come out of the Maze) or a label FAIL (that means one cannot come out of the maze). Let us assume that there is atleast one label SUCCESS at some leaf. A possible search is reflected in the binary tree shown in *Figure 4*; the dotted directed lines indicate the order of traversal. It may also be seen that the algorithm searches until either SUCCESS is reached or the tree is exhausted (in which case it means there was no solution). Note

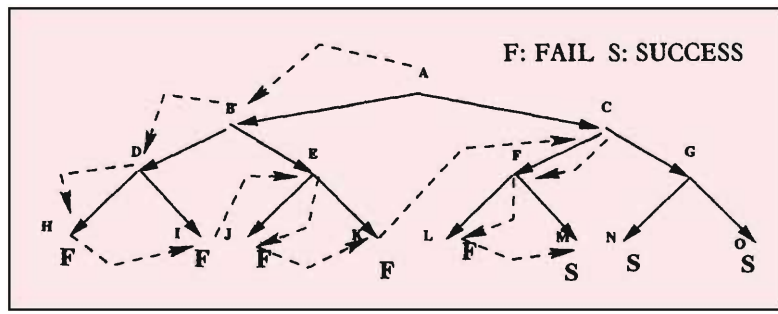


Figure 4 Traversal: depth first search.

that there could be several answers. While searching we could stop at the first SUCCESS or collect all the solutions as required. These searches are very handy in solving problems.

As one can see there are various ways of searching the tree exhaustively; the order of traversal depends upon the problem and the representation. Another method of traversal is referred to as the *Breadth-First Search* (BFS) wherein one does an exhaustive search for solutions at each level before proceeding to the next level. Further variations can be seen in the way one visits the root and the left or the right subtrees.

Discussion

In this article, we have seen various common data-structure abstractions, and their representations. Further, we have discussed various algorithm design techniques such as balancing, greedy technique, dynamic programming strategy and backtracking. In the next article, we shall explore correctness issues of program design.

Address for Correspondence
R K Shyamasundar
Computer Science Group
Tata Institute of Fundamental Research
Homi Bhabha Road
Mumbai 400 005, India
email:shyam@tcs.tifr.res.in
Fax:022-215 2110

Suggested Reading

- N Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall. Englewood Cliffs. New Jersey, 1976.
- A V Aho, J E Hopcroft and J D Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley Publ. Co., 1980.
- D Harel. *Algorithmics: The Spirit of Computing*. Addison-Wesley Publishing Co. Inc, 1987.

When lead is bombarded by neutrons for a longtime, it rearranges itself internally and becomes so elastic that a bell made of it might chime as resonantly as bells cast from the best bronze.

Quantum Kaleidoscope, Sept.-Oct.1995.