ESO211 (Data Structures and Algorithms) Lectures 25 to 27

Shashank K Mehta

1 Divide and Conquer: A review

We have discussed *Divide and Conquer* technique earlier. It is based on reducing a problem instance into one or more instances of the same problem with smaller size, i.e., smaller instances. If we can compute the solution of the given instance using the solutions of the smaller instances, then Divide and Conquer technique is to write a recursive algorithm in which in first we check if the instance is a *base case*. If so, then directly solve it. Else generate all the relevant smaller instances, make recursive calls for each of these instances, and finally use their solutions and build the solution of the given instance.

If we unroll the entire execution of the program, we see several smaller instances are instantiated. One can visualize it as a tree. See such an imaginary tree below. At the first stage the input instance of size parameter 8 is split into three problems if sizes $6, 4$, and 1. In the recursive call for problem of size 6 two problems of size 1 and one problem of size 4 are generated, so on.

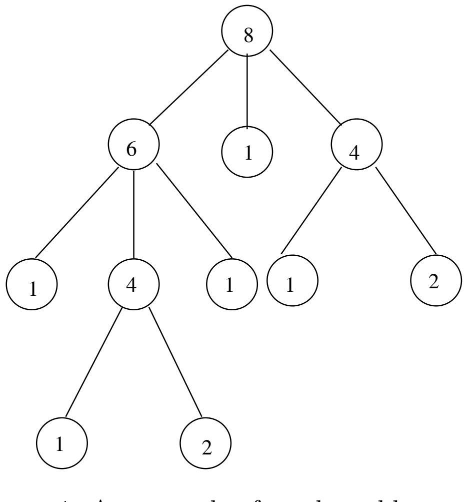

Figure 1: An example of a sub-problem tree.

Although each instance plays a role in the computation of the solution of the input instance, it is very much possible that same instance (instances with the same size and parameters) may have been generated more than once. In the imaginary tree of the figure problem of size 4 has two occurrences. In such a situation we may adopt the technique of *memoization*. In this technique we proceed just as in divide and conquer, but after solving each instance we save the solution. When a new instance is created, we first check if its solution already exists. If yes, then directly pick it from the memory. Otherwise we make a recursive call to solve it.

At the first look it appears that the memoization is a winner. But that is not quite true. It carries its own costs. One is the space (memory) cost. We need to save the results. The second is the time cost of finding from the memory whether the result exists or not.

2 Dynamic Programming

Divide and conquer or divide and conquer with memoization are top-down techniques. But we can adopt a bottom-up approach in the same class of problems if the number of possible instances grow polynomially with the size parameter. In this case we proceed by solving all the smaller instances in order such that every time we pick an instance to solve, all the relevant smaller instances are already solved. This bottom-up approach is known as *Dynamic Programming*. It can give a polynomial time algorithm only if the number of all the smaller problems is polynomial in the size parameter. Clearly we may have solved some instances which have no roll in solving the given instance, i.e., these instances may not occur in the sub-problem tree. But on the positive side we get the advantage of memoization. In the following sections we will discuss some examples of the application of dynamic programming technique.

Optimal Matrix Multiplication Order

Given n matrices $M_1, M_2, \ldots, M_n$ of sizes $a_1 \times a_2, a_2 \times a_3, \ldots, a_n \times a_{n+1}$ respectively, we want to compute the product $M_1 \cdot M_2 \cdots M_n$. The time complexity of multiplying an $p \times q$ matrix with and $q \times r$ matrix using the standard method is $O(p.q.r)$. To be precise, it involves $p.q.r$ multiplications and $p.(q-1).r$ additions. It is easy to verify that the total cost of computing the product of n matrices depends on the order in which we multiply them. If we decide to compute the product as $(M_1 \cdots M_i) \cdot (M_{i+1} \cdots M_n)$ then its will cost $C(1:i) + C(i+1:n) + a_1 a_{i+1} a_{n+1}$ where $C(1:i)$ denotes the cost of computing the product of the first i matrices and $C(i+1:n)$ is that of the last $n-i$ matrices. Since we do not know what will be the value of i for the multiplication order that costs the minimum, we have the following relation: $C(1:n) = \min_{i=1}^{n-1}\{C(1:i) + C(i+1:n) + a_1 a_{i+1} a_{n+1}\}$. Clearly, if $C(1:n)$ must be optimum, then so should $C(1:i)$ and $C(i+1:n)$. In stead of finding the optimal order, let us first design an algorithm to compute the optimal cost. We will show how to find the order later.

This discussion indicates that the problem instances that we should address are the optimal cost of computation $M_i \cdot M_{i+1} \cdots M_j$ for any $j \geq i$. Let $C(i:j)$ denote this cost. The problem size can be taken as $j-i+1$, the length of the chain. The smallest instances are those when $i = j$. Since it is the product of a "chain" of one matrix, it involves no computation. Hence $C(i:i) = 0$ for all i. The algorithm is as follows. The time complexity of the algorithm is $O(n^2)$ where n is the number of matrices in the chain.

Now to determine the order of multiplication which gives the optimal cost $C(1:n)$ we only need to save one piece of additional information during the above computation. When we compute the minimum in the second last step we should also save the value of k which gives the minimum value of $C(i:j)$. So compute in that step of the program $C'(i:j)$ which is the 2-tuple $(C(i:j), k)$. This indicates that in the computation of $M_i \cdots M_j$ we must perform first $M_i \cdots M_k$ and $M_{k+1} \cdots M_j$ products and then perform the multiplication of these two matrices. Define a method to denote the multiplication order as follows: If we have $P = M_1 \cdot M_2 \cdot M_3 \cdot M_4$ then sequence $3, 2, 1$ represents $((M_1 \cdot M_2) \cdot M_3) \cdot M_4$; sequence $2, 1, 3$ denotes $(M_1 \cdot M_2) \cdot (M_3 \cdot M_4)$; and sequence $3, 1, 2$ denotes $(M_1 \cdot (M_2 \cdot M_3)) \cdot M_4$. In this notation, if the sequences for $M_i \cdots M_k$ and $M_{k+1} \cdots M_j$ are $p_1, p_2, \ldots$ and $q_1, q_2, \ldots$ respectively, then that of $M_i \cdots M_j$ is $k, p_1, p_2, \ldots, q_1, q_2, \ldots$.

```
for $i := 1$ to $n$ do
    $C(i : i) := 0$;
end
for $l := 2$ to $n$ do
    for $i := 1$ to $n - l + 1$ do
        $j := i + l - 1$;
        $C(i : j) := \min_{k=i}^{j-1}\{C(i : k) + C(k + 1 : j) + a_i a_{k+1} a_{j+1}\}$
    end
end
Return $C(1 : n)$;
```

Algorithm 1: Optimal Matrix Multiplication

2.1 Longest Common Subsequence (LCS)

For an sequence $A = a_1a_2 \dots a_n$, sequence $a_{j_1}a_{j_2} \dots a_{j_k}$ is a subsequence of A if $j_1 < j_2 < j_3 \dots$.

Suppose we are given two sequences $A = a_1a_2 \dots a_n$ and $B = b_1b_2 \dots b_m$. In this example of dynamic programming, our goal is to compute a longest possible sequence C which a subsequence of A as well as of B. We will denote one such longest common subsequence by $LCS(A, B)$.

Once again our first task is to find a problem-subproblem relationship. Let us denote the initial contiguous subsequence $a_1a_2a_3 \dots a_i$ of A by $A[1 : i]$. Similarly for B. Now we make following trivial observation: (i) if $a_i = b_j$, then the $LCS(A[1 : i], B[1 : j]) = LCS(A[1 : i - 1], B[1 : j - 1]) \cdot a_i$ (which means the longest common subsequence of $A[1 : i-1]$ and $B[1 : j-1]$ followed by a_i); (ii) If $a_i \neq b_j$, then $LCS(A[1 : i], B[1 : j])$ is either $LCS(A[1 : i], B[1 : j-1])$ or $LCS(A[1 : i-1], B[1 : j])$, which ever is longer. These observations give us the problem-subproblem relationship which can be used to define dynamic programming based algorithm.

From the above observations we identify the relevant subproblems to be the $LCS(A[1 : i], B[1 : j])$ for each $i = 1, 2, \dots, n$ and each $j = 1, 2, \dots, m$. Let $C[i, j]$ denote the length of the longest common subsequence of $A[1 : i]$ and $B[1 : j]$. The total number of subproblems is $n.m$ which is small. The base cases are $C[0, j] = C[i, 0] = 0$. The following algorithm computes the length of the longest common subsequence. It take $O(1)$ (constant) time to compute each $C(i, j)$. Hence the time complexity is $O(n.m)$.

Once again we can recover the actual subsequence by saving an additional piece of information while computing all $C[i, j]$. Suppose we define $C'[i, j] = (C[i, j], x)$ where $x \in \{0, 1, 2\}$. In case $a_i \neq b_j$ if $C[i, j] = C[i-1, j]$, then we save $x = 1$. Otherwise we save $x = 2$. If $a_i = b_j$, then we save $x = 0$. After computing all $C'[i, j]$, we can start from $C'[n, m]$ and construct the longest common subsequence.

2.2 Rod Cutting Problem

Given an iron rod of n inches length (n is an integer). We are also give a table of prices of the iron rod pieces of length 1-inch, 2-inches, so on. The prices may not be proportional to the size of the pieces. Our objective is to find how to cut the given rod into pieces (each piece of integral number of inches) such that their total value based on the price table is maximum.

For example, let the price table be: $p_1 = 3$ Rs, $p_2 = 5$ Rs, $p_3 = 11$ Rs, $p_4 = 13$ Rs, $p_5 = 16$ Rs etc. We are given a rod of length $n = 5$. Then cutting it into sizes $3, 1, 1$ inches will give a value $3 + 3 + 11 = 17$ Rs, which is maximum.

Let us denote the maximum value achievable by optimally cutting a rod of n inches by $P(n)$.

```
for $i := 1$ to $n$ do
    $C[i,0] := 0$;
end
for $j := 1$ to $m$ do
    $C[0,j] := 0$;
end
for $i := 1$ to $n$ do
    for $j := 1$ to $m$ do
        if $a_i = b_j$ then
            $C[i,j] := C[i-1,j-1] + 1$;
        end
        else
            $C[i,j] := \max\{C[i,j-1], C[i-1,j]\}$;
        end
    end
end
return $C[n,m]$;
```

Algorithm 2: $LCS(A,B)$

Then $P(n) = \max_{i=1}^{n}\{p_i + P(n-i)\}$, where $p_0 = 0$ Also assume that if for any i, p_i is undefined, then we will take it to be zero. Also $P(0) = 0$. The justification of the above recurrence relation is that if the left most piece is of length i inches, then we get a value of $p_i + P(n-i)$. We then determine what value of i gives the maximum value. Note that the case $i = n$ corresponds to not cutting the rod and sell it as it is.

The time complexity of this solution is $O(n)$ since we need constant time to compute the values of $P(i)$ given the values of $P(j)$ for all $j < i$.