CS412 Algorithms: Design & Analysis

Spring 2024

Dhanani School of Science and Engineering

Habib University

Practice Problems

Week 10

1. Show, by means of a counterexample, that the following "greedy" strategy does not always determine an optimal way to cut rods. Define the density of a rod of length i to be $p_i = i$, that is, its value per inch. The greedy strategy for a rod of length n cuts off a first piece of length i, where $1 \leq i \leq n$, having maximum density. It then continues by applying the greedy strategy to the remaining piece of length $n - i$.

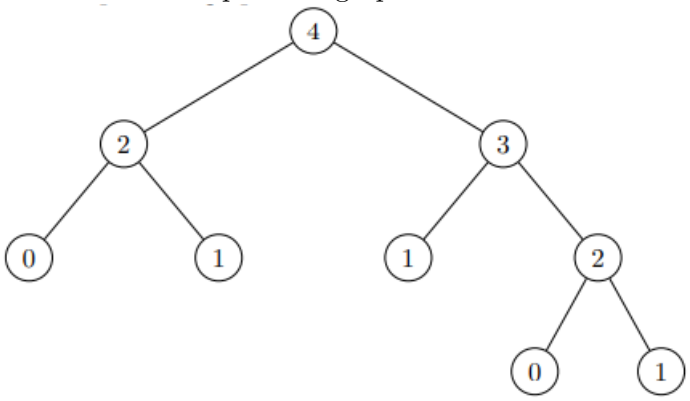
   Solution: Let $p_1 = 0$, $p_2 = 4$, $p_3 = 7$ and $n = 4$. The greedy strategy would first cut off a piece of length 3 since it has highest density. The remaining rod has length 1, so the total price would be 7. On the other hand, two rods of length 2 yield a price of 8.

2. The Fibonacci numbers are defined by recurrence:

$$F_i = F_{i-1} + F_{i-2} \text{ for } i \geq 2$$

   Give an $O(n)$-time dynamic-programming algorithm to compute the nth Fibonacci number. Draw the subproblem graph. How many vertices and edges are in the graph?

   Solution: The subproblem graph for $n = 4$ looks like:



   The number of vertices in the tree to compute the nth Fibonacci will follow the recurrence

$$V(n) = 1 + V(n-2) + V(n-1)$$

   And has initial condition $V(1) = V(0) = 1$. This has solution $V(n) = 2 * Fib(n) - 1$ which we will check by direct substitution. For the base cases, this is simple to check. Now, by induction, we have

$$V(n) = 1 + 2 * Fib(n-2) - 1 + 2 * Fib(n-1) - 1 = 2 * Fib(n) - 1$$

The number of edegs will satisfy the recurrence

$$E(n) = 2 + E(n-1) + E(n-2)$$

and having base cases $E(1) = E(0) = 0$. So, we show by induction that we have $E(n) = 2 * Fib(n) - 2$. For the base cases it clearly holds, and by induction, we have

$$E(n) = 2 + 2 * Fib(n-1) - 2 + 2 * Fib(n-2) - 2 = 2 * Fib(n) - 2$$

We will present a $O(n)$ bottom up solution that only keeps track of the the two largest subproblems so far, since a subproblem can only depend on the solution to subproblems at most two less for Fibonacci.

3. Which is a more efficient way to determine the optimal number of multiplications in a matrix-chain multiplication problem: enumerating all the ways of parenthesizing the product and computing the number of multiplications for each, or running RECURSIVE-MATRIX-CHAIN? Justify your answer.

Solution: The runtime of of enumerating is just $n * P(n)$, while is we were running RECURSIVE-MATRIX-CHAIN, it would also have to run on all of the internal nodes of the subproblem tree. Also, the enumeration approach wouldn't have as much overhead.

4. Draw the recursion tree for the MERGE-SORT procedure from Section 2.3.1 on an array of 16 elements. Explain why memoization fails to speed up a good divideand-conquer algorithm such as MERGE-SORT.

Solution: Let $[i..j]$ denote the call to Merge Sort to sort the elements in positions i through j of the original array. The recursion tree will have $[1..n]$ as its root, and at any node $[i..j]$ will have $[i..(j-i)/2]$ and $[(j-i)/2+1..j]$ as its left 4 and right children, respectively. If $j - i = 1$, there will be no children. The memoization approach fails to speed up Merge Sort because the subproblems aren't overlapping. Sorting one list of size n isn't the same as sorting another list of size n, so there is no savings in storing solutions to subproblems since each solution is used at most once.

5. Consider a variant of the matrix-chain multiplication problem in which the goal is to parenthesize the sequence of matrices so as to maximize, rather than minimize, the number of scalar multiplications. Does this problem exhibit optimal substructure?

Solution: This modification of the matrix-chain-multiplication problem does still exhibit the optimal substructure property. Suppose we split a maximal multiplication of $A_1, ..., A_n$ between $A_k and A_{k+1}$ then, we must have a maximal cost multiplication on either side, otherwise we could substitute in for that side a more expensive multiplication of $A_1, ..., A_n$.

6. As stated, in dynamic programming we first solve the subproblems and then choose which of them to use in an optimal solution to the problem. Professor Capulet claims that we do not always need to solve all the subproblems in order to find an optimal solution. She

suggests that we can find an optimal solution to the matrix-chain multiplication problem by always choosing the matrix A_k at which to split the subproduct $A_iA_{i+1}...A_j$ (by selecting k to minimize the quantity $p_{i-1}p_kp_j$) before solving the subproblems. Find an instance of the matrix-chain multiplication problem for which this greedy approach yields a suboptimal solution.

Solution: Suppose that we are given matrices A_1, A_2, A_3, and A_4 with dimensions such that p_0, p_1, p_2, p_3, $p_4 = 1000$, 100, 20, 10, 1000. Then $p_0p_kp_4$ is minimized when $k = 3$, so we need to solve the subproblem of multiplying $A_1A_2A_3$, and also A_4 which is solved automatically. By her algorithm, this is solved by splitting at $k = 2$. Thus, the full parenthesization is $(((A_1A_2)A_3)A_4)$. This requires $1000 \times 100 \times 20 + 1000 \times 20\mathring{u} \times 10 + 1000\mathring{u} \times 10 \times 1000 = 12, 200, 000$ scalar multiplications. On the other hand, suppose we had fully parenthesized the matrices to multiply as $((A_1(A_2A_3))A_4)$. Then we would only require $100 \times 20 \times 10 + 1000 \times 100 \times 10 + 1000\mathring{u} \times 10\mathring{u} \times 1000 = 11, 020, 000$ scalar multiplications, which is fewer than Professor Capulet's method. Therefore her greedy approach yields a suboptimal solution.