

1.

Yes, we can use a similar dynamic programming algorithm to find the sequence of matrix multiplications which maximizes the number of scalar multiplies, rather than the sequence that minimizes it. Let A_1, \dots, A_n be the input sequence of matrices, with dimensions $p_0 \times p_1, \dots, p_{n-1} \times p_n$, and let $c(i, j)$ be the maximum number of scalar multiplies needed to multiply together the subsequence A_i, \dots, A_j . Then we can express $c(i, j)$ recursively:

$$c(i, j) = \max_{k \in \{i, \dots, j-1\}} c(i, k) + c(k+1, j) + p_i p_k p_j$$

This recursive formulation leads to a similar pyramid of subproblems, where the solutions for the larger subproblems near the top of the pyramid can be calculated given the solutions for the smaller subproblems near the bottom. The running time is again $O(n^3)$.

2.

We want to show by substitution that the solution to the recurrence below is $\Omega(2^n)$, with base case $P(1) = 1$.

$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)$$

Like all proofs by substitution, this is a proof by induction. We'll do the induction step first, since it will determine what n_0 we use for the base case.

Induction step: Assume inductively that $P(i) \geq c2^i$, for all $1 \leq i < n$. Then we can write $P(n)$ as:

$$P(n) \geq \sum_{k=1}^{n-1} (c2^k)(c2^{n-k}) = (n-1)c^2 2^n$$

We want to prove a *lower bound* on $P(n)$, so we need to show that $P(n) \geq c2^n$, for some c . This will be true for $c \geq 1$.

Base case: We now need to pick an n_0 so that $P(n_0) \geq c2^{n_0}$, for any $c \geq 1$. $P(1) = 1$, so this does not work. $P(2) = 4$, which does work, so we set $n_0 = 2$.

3.

The idea of this problem is that we want to describe how different a string x is from a string y . We express the difference by counting the number of editing operations required to transform x into y . We consider moving a pointer i from left to right across string x , and performing operations to transform x into y . As we go along, we are producing a copy of y , also from right to left, with a pointer j pointing to the current character in y .

If $x(i) = y(j)$, we do not need to change the current character. We call this a COPY operation (although we could also have called it a NOP) and we give it cost 0. The other operations are INSERT, REPLACE and DELETE.

We want to find the cost $c(m, n)$ of the minimum cost sequence of operations, where m is the length of x and n is the length of y . To do this using dynamic programming, we consider subproblems of the form $c(i, j)$, the minimum cost required to transform the string $x(1), \dots, x(i)$ to $y(1), \dots, y(j)$. We express this recursively by thinking about the last operation which will be performed in the sequence of operations. If $x(i) = y(j)$, we could first transform $x(1), \dots, x(i-1)$ to $y(1), \dots, y(j-1)$ and then COPY $x(i)$ to $y(j)$. Other options would be to first transform $x(1), \dots, x(i-1)$ to $y(1), \dots, y(j)$ and then DELETE $x(i)$, or to first transform $x(1), \dots, x(i)$ to $y(1), \dots, y(j-1)$ and then INSERT $y(j)$. There would be no point in using a REPLACE operation. If $x(i) \neq y(j)$, then we might transform $x(1), \dots, x(i-1)$ to $y(1), \dots, y(j-1)$ and

then REPLACE $x(i)$ with $y(j)$. Or we might use a DELETE or INSERT, as above. A COPY would not be possible. So we get this recursive formulation for the cost:

$$c(i, j) = \begin{cases} \min\{c(i-1, j-1), 1 + c(i-1, j), 1 + c(i, j-1)\} & \text{if } x(i) = y(j) \\ \min\{1 + c(i-1, j-1), 1 + c(i-1, j), 1 + c(i, j-1)\} & \text{if } x(i) \neq y(j) \end{cases}$$

To compute $c(i, j)$ using this formulation, we need to have already solved the smaller subproblems $c(i-1, j-1), c(i, j-1), c(i-1, j)$. So we need to solve the subproblems in some order that will ensure this. We can consider the subproblems organized into a table as follows:

$$\begin{array}{cccc} c(n, 0) & c(n, 1) & \cdots & c(n, m) \\ & \cdots & & \\ & & \cdots & \\ c(1, 0) & c(1, 2) & \cdots & c(1, m) \\ c(0, 0) & c(0, 2) & \cdots & c(0, m) \end{array}$$

The only solutions for the bottom row require INSERTING j characters, so $c(0, j) = j$. Similarly the only solutions for the left column are to DELETE i characters, so $c(0, i) = i$. If we solve the next-lowest row first, from left to right, then the next-lowest row, and so on, we will always have the three subproblem solutions we need to compute $c(i, j)$. We can compute $c(i, j)$ from the subproblem solutions $c(i-1, j-1), c(i, j-1), c(i-1, j)$ in constant time. The total number of subproblems is $(n+1)(m+1)$, so the running time of the dynamic programming solution is $O(nm)$.