Brief review of course to date

This course is about the design and analysis of algorithms.

- Iterative algorithms
- Divide-and-conquer algorithms
- Randomized algorithms
- Analysis of running time of each type
- Analysis of correctness of each type

Dynamic programming (chapter 15)

I briefly reviewed the Fibonacci problem from last week and the two solutions we came up with. One used memoization and the other computed the values bottom-up.

Usually, dynamic program is used to solve optimization problems, that is, problems where we are trying to maximize or minimize some value.

The rod-cutting problem

We started with the rod-cutting problem: This is presented in the text in section 15.1 so see the description there. Recall that there are two inputs: a natural number \( n \), the length of the rod to be cut, and an array \( p[1..n] \), where \( p[i] \) is the payoff for a rod of length \( i \). The goal is to maximize the profit one can earn by cutting the rod into one or more pieces.

A good way to view the problem is that you need to determine where the first cut should be made to get the maximum profit. Without loss of generality we can assume that the first cut is also the leftmost cut.

If we somehow knew that to get the optimal solution the first cut should be at position \( i \), our problem would now reduce to solving the problem on the right-hand piece (a rod of length \( n - i \)). The profit from the left-hand piece is \( p[i] \). Let \( r \) denote the profit from the right-hand side (which may result from further cuts on that right-hand piece). Then the maximum profit for the original rod is \( p[i] + r \).

One thing to note here is that because \( p[i] + r \) is an optimal solution to the length \( n \) problem, \( r \) must be an optimal solution to the length \( n - i \) problem. If it weren’t, there would be a solution to it with value \( r' > r \). But then we could use that solution in the original problem and so get a larger profit \( p[i] + r' > p[i] + r \), contradicting our assumption that we have the optimal solution for \( n \).
Of course, we don’t know where the optimal place for the first cut is so we will have to try all $n$ positions and take the maximum profit from among them. Let $r[n]$ denote that profit. We can express its value in the following recurrence relation:

$$r[n] = \begin{cases} 0 & \text{if } n = 0 \\ \max_{1 \leq i \leq n} (p[i] + r[n - i]) & \text{if } n > 0 \end{cases}$$

The **CUT-ROD** algorithm on page 363 simply computes this value in a top-down recursive manner. And it works…eventually.

The problem is the running time. Let $T(n)$ denote the number of calls made to **CUT-ROD** on input $n$. As the book explains $T(n) = 1 + \sum_{i=1}^{n-1} T(i)$. Solving this (with techniques we have not seen) results in finding that $T(n) = 2^n$. In other words, this algorithm runs in exponential time and so is useful only for very small inputs. This happens because, for most values of $i$, the value of $r[i]$ is computed many times.

One quick way to improve its performance is to use memoization. Use essentially the same algorithm but modify it so that, for each $i$, the first time it computes $r[i]$, it stores the value in an array. After that, whenever the value $r[i]$ is needed, it is simply retrieved from the array.

This is expressed in the **MEMOIZED-CUT-ROD** and **MEMOIZED-CUT-ROD-AUX** algorithms in the text. The first one initializes the entries in the $r$ array and makes the first call to the second. The second is a modified version of the **CUT-ROD**. It’s a little difficult to see but the first time $r[i]$ is computed requires $i$ recursive calls to **MEMOIZED-CUT-ROD-AUX**. So the total number of recursive calls is:

$$\sum_{i=0}^{n} i = \frac{n(n + 1)}{2} = \Theta(n^2)$$

This version is still computing top-down but it’s not recomputing values already known.

But look at the definition of $r[n]$ more closely. You can see that computing it only relies on values $r[k]$ where $k < n$ and the value of $r[0]$ is given outright. A dynamic programming approach suggests computing from the bottom-up and that is how it’s done in the algorithm **BOTTOM-UP-CUT-ROD**.

Inside the outer loop of that algorithm the index variable $j$ indicates which entry in $r$ is being computed. Note that computing $r[i]$ only relies on values in. Because $j$ starts at 1, this is exactly what happens, computing $r[j]$ relies only on values in $r[k], 0 \leq k < j$.

Analyzing it is easy. We will count the number of times the $\max()$ method is called in line 6. It is inside nested for-loops and using the translation from loops to sums we’ve seen previously:
\[ \sum_{j=1}^{n} \sum_{i=1}^{j} 1 = \sum_{j=1}^{n} j = \frac{n(n + 1)}{2} = \Theta(n^2) \]

Asymptotically the algorithms MEMOIZED-CUT-ROD and BOTTOM-UP-CUT-ROD have the same running time.

The section in the book goes on to explain sub-problem graphs and how to reconstruct the cuts needed in the rod cutting problem. We will discuss these next week.

**The matrix-chain multiplication problem**

Recall that there are four steps in solving a problem with a dynamic programming algorithm:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up manner.
4. Construct an optimal solution from computed information.

As we did in the rod-cutting problem, we will solve the matrix-chain multiplication problem by following these four steps.

First, recall how matrix multiplication works. To multiply two matrices \( A \) and \( B \), the number of columns in \( A \) must equal the number of rows in \( B \). The critical operation is scalar multiplication and we want to count how many must be done. If matrix \( A \) has size \( m \times n \) and matrix \( B \) has size \( n \times p \) then the number of scalar multiplications performed is \( m \cdot n \cdot p \).

You are given a sequence of matrices \( A_1, A_2, \ldots, A_n \) to be multiplied together and the problem is to minimize the number of scalar multiplications. Let \( p_i \) denote the number of rows in matrix \( A_i \). Because the number of columns of one matrix must be equal to the number of rows of the next, matrix \( A_i \) must have \( p_i \) columns. In other words, the sequence of the sizes of the matrices is \( p_0 \times p_1, p_1 \times p_2, p_2 \times p_3, \ldots, p_{n-1} \times p_n \).

We can rephrase the problem by saying that we need to decide where to split the matrix sequence and so decide what will be the last matrix multiplication. Where we do that split, we put parentheses around the matrices to the left and around the matrices to the right. For example, if decide to split after \( A_k \) that’s the same as putting in parentheses to get \((A_1A_2 \ldots A_k)(A_{k+1}A_{k+2} \ldots A_n)\). In other words compute the matrix products \( B = A_1A_2 \ldots A_k \) and \( C = A_{k+1}A_{k+2} \ldots A_n \) and then compute the matrix product \( BC \).

Where the parentheses are placed does make a difference in the number of scalar multiplications. Let \( A_1A_2A_3A_4 \) be the sequence with sizes \( n \times 1, 1 \times n, n \times 1, 1 \times n \). If we parenthesize the sequence as \((A_1A_2)(A_3A_4)\), we will perform \( n^2 \) scalar multiplications.
forming the \( n \times n \) product \( A_1A_2 \), \( n^2 \) forming the product \( n \times n \) product \( A_3A_4 \), and \( n^3 \) multiplying those products together, for a total of \( n^3 + 2n^2 = \Theta(n^3) \) scalar multiplications.

But if we parenthesize as \((A_1(A_2A_3))A_4\), we perform \( n \) scalar multiplications computing the product \( B = A_2A_3 \), \( n \) computing \( C = A_1B \), and \( n^2 \) computing \( CA_4 \) for a total of \( n^2 + 2n = \Theta(n^2) \) scalar multiplications, a very large improvement.

For a moment, we consider the brute-force algorithm that tries all possible ways of placing parentheses. As the book points out, there are \( \Theta(4^n) \) ways and that’s too many.

So back to deciding where to place the first two pairs of parentheses or, viewed another way, where to split the original sequence. Of course, we don’t know ahead of time where the best split should be so we will have to try every possible position. Here is the recurrence that defines the value of \( m[i, j] \), which is the minimum number of scalar multiplications needed to compute the matrix product \( A_iA_{i+1}\cdots A_j \):

\[
m[i, j] = \begin{cases} 
0 & \text{if } i = j \\
\min_{i \leq k < j} \{ p_{i-1}p_kp_j + m[i, k] + m[k + 1, j]\} & \text{if } i < j
\end{cases}
\]

To find the optimal solution for the original sequence, we compute \( m[1, n] \). Doing so top-down results in an algorithm requiring \( \Theta(2^n) \) steps.

Now we turn to the DP approach. We will store the values of \( m[i, j] \) in an \( n \times n \) table. In fact we don’t need the entire table because we only fill the entries where the row index is less than or equal to the column index.

The trick is to fill the entries in the table in an efficient manner. As described above, \( m[i, j] \) is the optimal solution for a sequence of \( j - i + 1 \) matrices. Because this problem has the optimal sub-problem property, this means we should first compute optimal solutions for shorter sequences. The shortest sequence contains one matrix and computing that requires 0 scalar multiplications so we initialize the table with zeroes along its diagonal. Computing optimal solutions for sequences of two matrices is also straightforward and computing solutions for longer sequences means looking up solutions for shorter sequences.

Imagine the table drawn so that the rows are numbered 1 to \( n \), top to bottom, and the columns are numbers 1 to \( n \), left to right. We first fill the main diagonal, from \( m[1,1] \) to \( m[n,n] \). Next we fill the diagonal above that, from \( m[1,2] \) to \( m[n-1,n] \). Next is the diagonal above that, and so on. The last entry we fill is \( m[1,n] \) and that’s the solution to the original problem.

Because this recap is so late in coming I’m going to stop here and get it out. I will discuss all of this during the next lecture.