Lecture Notes for CSCI 311: Algorithms

Set 7-Introduction to Dynamic Programming

Professor Talmage

February 16, 2026

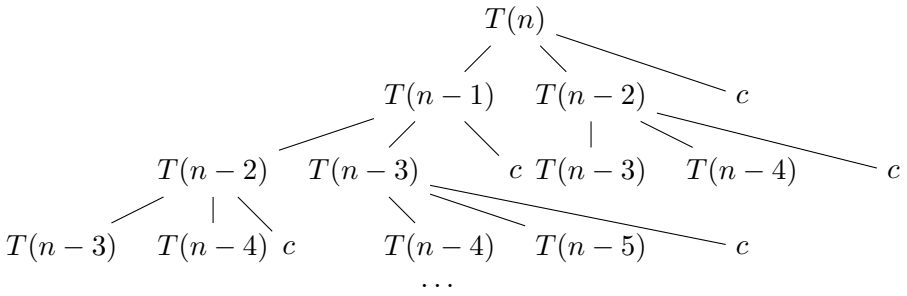
---

1 Fibonacci

Exercise: What is your best guess for the runtime of the following pseudocode?

```
1: function F(x)
2:     if n ≤ 2 then
3:         return 1
4:     end if
5:     return F(n − 1) + F(n − 2)
6: end function
```

First, note that the runtime is described by the recurrence $T(n) = T(n-1) + T(n-2) + O(1)$, and consider the recursion tree:



We cannot really combine the terms here, but we can give loose bounds.

- Find the shortest path down the tree to a base case: Descend to the $n-2$ term each time.
- That path has length $n/2$.
- Since each level doubles the number of nodes, this implies that there are at least $2^{n/2}$ base case nodes.
- This gives $T(n) = \Omega(2^{n/2}) + \sum_{j=0}^{n/2-1}(c2^j) = \Omega(2^{n/2}) = \Omega(\sqrt{2}^n)$.
- A similar analysis down the $n-1$ edges gives $T(n) = O(2^n)$.

So, we know that this code is exponentially expensive, with base between 1.414 and 2. Why is this simple code so expensive?

- We do the same work many, many times. Note that $T(n-2)$ appears twice in the recursion tree, $T(n-3)$ appears three times, $T(n-4)$ appears 5 times, and so on.

- Not every recursive solution does this. Binary search and Square Matrix Multiplication never ran into this problem, because their subproblems do not overlap.

How can we fix this code to not take so long?

- **Save your work!**
  - This is all there is to dynamic programming. Everything else is bookkeeping.
- There are two general ways we can structure our code:
  - Top-Down: Keep the basic recursive structure, but at every recursive call, check a database to see if we have already solved this subproblem. If so, use the stored solution to avoid any additional work.
  - Bottom-Up: Calculate the base cases, then compute each subproblem in order of increasing size, ending with the original problem.
  - Both approaches rely on understanding the space of possible subproblems, so we will spend a lot of time doing that.
- We will return to the formal details. For now, we will fix the Fibonacci code, then consider another example problem.

1.1 Efficient Fibonacci

First, we will build a bottom-up solution. The important step is to allocate space to solve all of the intermediate subproblems we might need to compute $f(n)$. Since the Fibonacci numbers are organized by a single, integer index, we just need an array from 1 to n.

Exercise: Give pseudocode to compute F_n non-recursively.

```
1: function BottomUpFibonacci(n)
2:     Create array F with n spaces (1-indexed)
3:     Set F[1] = F[2] = 1
4:     for j = 3 to n do
5:         F[j] = F[j − 1] + F[j − 2]
6:     end for
7:     return F[n]
8: end function
```

Correctness: By induction, $F[1] = F_1$ and $F[2] = F_2$. Assume for $n > 2$, all $F[k] = F_k$ for $1 \leq k < n$. Then $F[n] = F[n-1] + F[n-2] = F_{n-1} + F_{n-2} = F_n$. $\square$

Runtime: A single loop iterating n times, constant work per iteration, so $\Theta(n)$.

Notes:

- Bottom-up solutions sometimes require more space than recursive algorithms, as (1) recursion may skip some possible subproblems, and (2) recursion is depth-first, so it can sometimes reuse space across recursive calls. Same space in this example.
- It is very important that we have a clear order on the space of possible subproblems so that when we try to compute a particular subproblem, we have already completed everything on which it depends.

Next, we look at a top-down solution. This is the same as the original recursive algorithm, but saving our work so that we never have to solve the same subproblem more than once. We will use a wrapper function to set up the place to store our subsolutions:

```
1: function TopDownFibonacci(n)
2:     Create empty array F with n spaces (1-indexed)
3:     return recursiveTDFibo(n, F)
4: end function
```

Exercise: Write `recursiveTDFibo`, but only make a recursive call if the subproblem has not already been solved.

```
1: function recursiveTDFibo(n, F)
2:     if n ≤ 2 then
3:         F[n] = 1
4:         return F[n]
5:     end if
6:     f1 = F[n − 1]
7:     if f1 is empty then
8:         f1 = recursiveTDFibo(n − 1, F)
9:     end if
10:    f2 = F[n − 2]
11:    if f2 is empty then
12:        f2 = recursiveTDFibo(n − 2, F)
13:    end if
14:    F[n] = f1 + f2
15:    return F[n]
16: end function
```

Correctness: Same argument as recursive algorithm (matches inductive definition of Fibonacci numbers).

Runtime: Only one recursive call for each $i, 1 \leq i \leq n$, each table cell is checked no more than twice, so $\Theta(n)$.

Notes:

- The method of explicitly saving subproblem solutions to avoid repeated work is called *memoization*, since you write memos to yourself.
- We could check the lookup table inside recursive calls instead of before making the call, as an extra base case, without changing the runtime. This may make runtime harder to argue, though, as we will have more recursive calls.
- In this example, we could optimize out the second recursive call, since we know recursively computing $F[n-1]$ will compute $F[n-2]$. Not all problems simplify that cleanly.

2 Linear Resource Allocation

Suppose we have a divisible resource where the value of each piece depends on its size. Imagine deciding how large cereal boxes should be: You have a large quantity of freshly-cooked breakfast cereal and need to decide how many boxes of what size to pack it in for sale. Prices and prices per unit are different for different size boxes, so how you divide it will affect how much money you make. In general, given a resource

and a list of values for different-size pieces of that resource, we want to decide how to divide that resource to optimize the total value we get for it.

> **Aside:** We are focused on learning about how to solve algorithmic problems, not what the problems are. As a computer scientist, though, you are responsible for thinking about the problems you are trying to solve and whether that is the right thing to do. As one example, the technique we are learning is exactly that used to gerrymander voting districts–to divide them in such a way as to suppress certain votes by clumping them so that they are not equally represented. That is, mathematically, an optimization problem, so solvable like any other. But it is your responsibility to think about how your solutions can be used.

Resource-Division Problem: Given a total amount n of a resource and a list of values for each possible subdivision size, determine how to divide your resource to maximize the total value you can achieve in dividing your resource.

- The book calls this the "Rod Cutting Problem", as one application is deciding how to cut up a metal rod into different lengths for sale. A foundry may forge rebar in 20-foot long rods, but many customers do not want 20-foot rods, or to have to cut them themselves. Thus, the store can cut the rods into different-length pieces and sell the pieces for different prices based on demand. Many other problems follow the same pattern, such as the cereal-boxing problem we mentioned, or deciding how to allocate your time to different jobs with different time requirements and income amounts to maximize income.

Input: $n \in \mathbb{Z}^+$, $v_1, \ldots, v_n \in \mathbb{R}^{\geq 0}$

Output: $i_1, \ldots, i_k \in \mathbb{Z}^+$ s.t. $i_1 + \cdots + i_k = n$ and $v_{i_1} + \cdots v_{i_k}$ is maximized.

- v_i is the value of a rod of length i: A 1-foot rod will earn v_1, a 2-foot rod will earn v_2, etc.
- This makes v_{i_1} the value of the first piece we choose to cut, etc., since i_1 is the length of the first piece and values are indexed by rod length.
- We assume that there are values for each integer length of rod, and that you can repeat lengths.

We can describe a straightforward recursive solution:

$$max\_value(n) = (v_{i_1}) + max\_value(n - i_1)$$

That is, the best we can do is the value from the first piece, plus the value we get from the rest of our rod. Of course, this (value) recurrence is incomplete, since we do not know the value of the first piece. We need to consider all the possibilities, so we maximize this recurrence over all possible choices of the first piece length:

$$max\_value(n) = \max_{i_1=1}^{n}(v_{i_1} + max\_value(n - i_1))$$

Example: Let $n = 5, V = [1, 2, 5, 7, 6, 10]$.

Division	(5)	(4, 1)	(3, 2)	(3, 1, 1)	(2, 3)	(2, 2, 1)	(2, 1, 2)	(2, 1, 1, 1)	(1, 1, 1, 1, 1)
Value	6	7+1	5+2	5+1+1	2+5	2+2+1	2+1+2	2+1+1+1	1+1+1+1+1

- Note that there are other permutations, but we assume order does not matter, so omit them for space.
- Best solution is to cut one 1-foot rod and one 4-food rod.

> **Exercise:** Solve the problem for $n = 5$, with $V = [1, 3, 4, 7, 10]$.

Exercise: Write pseudocode for the recursive (not dynamic programming) solution based on the maximization description above. What is your code's runtime?

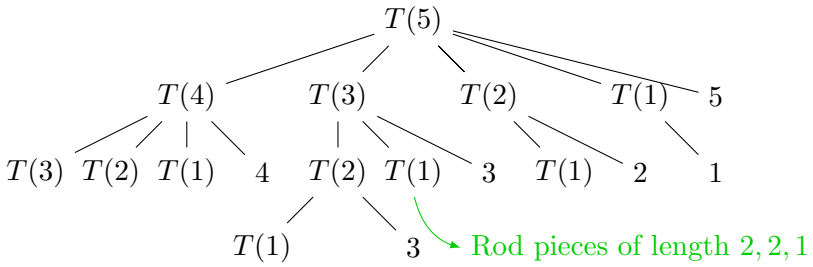
$$T(n) = \begin{cases} 1 & n = 0 \\ \sum_{i=0}^{n-1} T(i) + \Theta(n) & n > 0 \end{cases}$$

This is not obvious to solve.

Claim 1. $T(n) = \Omega(2^n)$

Proof. Consider the recursion tree. Each step from a node to a child represents a choice of the length of the next piece we cut. Thus, each path from the root to a leaf represents a complete division of our rod. The number of leaves in the tree will then be the number of ways to divide our rod into pieces of integer length.

Since we could choose to cut or not cut at each 1-foot mark, in a total of n feet, there are $n-1$ different places we could split our rod. Two choices (cut or not) for each of those places gives 2^{n-1} possible divisions. We do at least a constant amount of work on each path to a leaf (sometimes n work!), so we have between $c * 2^n$ and $n2^n$ total work, giving a lower bound of $\Omega(2^n)$.



□

Observe that the same subproblems appear repeatedly in our recursion tree. This suggests that we can improve our runtime by saving subsolutions. Create an array $opt[0..n]$ that stores the optimal value we can earn by dividing each total rod length from 1 foot to n feet. In other words, $opt[i]$ will save the maximum value achievable with an i foot starting rod.

```
1: function MAX-VALUE(n, V)
2:     create opt[0..n], all elements initially −∞ except opt[0] = 0
3:     return TD-RodCut(n, V, opt)
4: end function

5: function TD-RodCut(n, V, opt[])
6:     if opt[n] ≥ 0 then return opt[n]
7:     else if n == 0 then val = 0
8:     else
9:         val = max_{1≤i≤n}(v_i + TD-RodCut(n − i, V, opt))
10:    end if
11:    opt[n] = val
12:    return opt[n]
13: end function
```

Runtime: We only call `TD-RodCut` once for each $1 \leq i \leq n$, not counting calls which find $opt[n]$ already filled and thus return on line 6. Each of those n calls does up to n work, looping across the lengths of the first piece we cut, each with either $O(1)$ time for a recursive call that is just a lookup or cost counted in a separate first call with argument $n - i$. This gives a total time of $O(n^2)$.

Correctness: Same argument as the simple recursive algorithm.

Note that while the algorithm finds the optimal value you can earn, it does not tell you how to do this, since it does not store which i led to the maximum value. This is a common feature of dynamic programming algorithms, and can be easily solved. We will come back to this, but first, let us consider how to remove the recursion from our algorithm. The previous solution was top-down with memoization. Consider a bottom-up approach.

> **Exercise:** In what order should we calculate values for a bottom-up solution? What was the base case, what is the goal value? In what direction do the dependencies lie?
> Write pseudocode for a bottom-up solution to the rod-cutting problem.

```
1: function BU-RodCut(n, V)
2:     Let opt[0..n] be an array
3:     opt[0] = 0
4:     for j = 1 to n do
5:         opt[j] = max_{1≤i≤j}(v_i + opt[j − i])
6:     end for
7:     return opt[n]
8: end function
```

Runtime: $O(n^2)$, since we have a for loop running $n-1$ times, each time running an up-to-n element max. (Precise analysis gives triangular numbers, like `InsertionSort`.)

Correctness: After cutting one piece, we can only have a smaller amount of rod left than we started with. Thus, every dependency will be a subproblem we have already solved, and each solution is based off our original recurrence, so our overall solution is correct.

> **Exercise:** Fill out the *opt* table, given $n = 5$ and $V = [30, 65, 150, 170, 185]$:

i	0	1	2	3	4	5
v_i	0	30	65	150	170	185
opt_i	0	30	65	150	180	215

Now, for a final step, we can track the actual choices of piece lengths which led to our optimal solution.

```
1: function BU-RodCut(n, V)
2:     Let opt[0..n], sol[0..n] be arrays
3:     opt[0] = sol[0] = 0
4:     for j = 1 to n do
5:         opt[j] = −∞
6:         for i = 1 to j do
7:             if opt[j] < v_i + opt[j − i] then
8:                 sol[j] = i
9:                 opt[j] = (v_i + opt[j − i])
10:            end if
11:        end for
12:    end for
13:    return opt, sol
14: end function
```

Exercise: Complete the following table to determine the optimal values, then reconstruct the piece lengths that give those optimal values.

i	0	1	2	3	4	5	6	7	8	9	10
v_i	0	1	5	8	9	10	17	17	20	24	30
opt_i											
sol_i											

i	0	1	2	3	4	5	6	7	8	9	10
v_i	0	1	5	8	9	10	17	17	20	24	30
opt_i	0	1	5	8	10	13	17	18	22	25	30
sol_i	0	1	2	3	2	2	6	1	2	3	10