Lecture 17: Dynamic Programming

- Dynamic Programming: Definitions and Big Ideas
- Warmup: Longest palindromic sequence
- Optimal binary search tree
- Alternating coin game

"Dynamic" Programming

Recall:

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution based on optimal solutions of subproblems
3. Compute the value of an optimal solution in bottom-up fashion (recursion & memoization)
4. Construct an optimal solution from the computed information

Dynamic programming is an algorithmic technique that we can use to solve problems that look exponential in complexity, but actually aren't, if we are somewhow able to recurse and remember previous solutions. In short, it is "clever brute force"; think about the difference between this and greedy and divide and conquer algorithms. We usually have to guess something to start, so, dynamic programming is useful when:

1. Guessing part of a solution allows us to break the problem into subproblems (and this is recursively true)
2. # of guesses and # subproblems at each step is polynomial
3. Polynomial work is required to find the solution to a problem given solutions to subproblems.

(You might want to read up on the Bellman Equation to understand more about exactly when this works.)

In short, we try to find and characterize an optimum substructure associated with the problem and its connection to its subproblems. Once we've made this characterization, we can write a recurrence which relates the optimal value of a bigger problem

to the optimal values of subproblems. Then we compute the value of the optimal solution through a recursive memoization (memoization is really what gives us the efficient algorithm, because we don't redo subproblems).

Notice that when you're writing the program, you might be thinking about it, and writing it, in a top-down fashion, but when it executes, the solutions will "flow up" from the bottom. Usually, it's the second step, the recurisive definition, which is the hardest: how to relate the optimal solutions of subproblems to that of a larger problem.

Also recall that the running time is "the number of subproblems" $\times$ "time taken per subproblem *given that* you already have solutions to sub-subproblems" (that is, no further recursion, otherwise we'd be double counting, since these subproblems would be part of the first part of the multiplication). The second part can also be further broken down into "# sub-subproblem guesses per subproblem + work to find the solution to this subproblem from sub-subproblems".

Some History: The term "dynamic programming" was introduced by Richard Bellman; here's why he called it that, in his own words:

The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, programming. I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying – I thought, lets kill two birds with one stone. Lets take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is its impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. Its impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

Longest Palindromic Sequence

Definition: A palindrome is a string which remains unchanged when reversed (we usually ignore punctuation and whitespace). The word is derived from the Greek roots palin ($\pi\alpha\lambda\iota\nu$, meaning "again") and drom- / dromos ($\delta\rho o\mu o\zeta$, meaning "way" or "direction").

Finnish apparently has a 25-letter palindromic word: SOLUTOMAATTIMITTAAMOTULOS which means the result from a measurement laboratory for tomatoes, although technically it is a compound of four words. Weird Al Yankovic's song "Bob" is a palindrome-laden spoof of Bob Dylan's "Subterranean Homesick Blues".

Completely unnecessary extra notes

Unlike what wikipedia would have you believe, the word "palindrome" was not exactly *coined* by the English playwright Benjamin Johnson (who was very important nevertheless, but he was writing plays around the same time as Shakespeare and thus had to contend with some stiff competition in our cultural memory). The concept of palindromes has existed since time immemorial, and people have used other words to describe the idea. It just so happens that the particular connection made by Johnson, of the *already existing* word "palindrome" to the concept of a sequence of letters that reads the same in both directions, has survived to our time in English.

So, although the roots of the word "palindrome" are in Greek, the Greek themselves used karkinike epigrafe (crab inscription) or just karkinoi (crabs), alluding to the back-and-forth manner in which crabs ambulate to describe the concept.

You may have read Timon of Athens by Shakespeare (based on a legendary misanthrope) – Shakespeare was likely influenced by the works of Plutarch and Lucian (of Samosata). Lucian's work (titled simply "Timon" or "The Misanthrope") contained the word palindrome, but that was in the sense of someone going from A to B and coming back. Diogenes mentions the word when talking about Aristippus of Cyrene. (The latter is rather important – he was pretty much the first person in the Western world to charge for lectures. He was a student of Socrates, but ended up with a very different philosophy "to endeavour to adapt circumstances to myself, not myself to circumstances".) In short, the word has been in use, though not exactly common use, for a very long time.

Examples: racecar, civic, t, bb, "A man, a plan, a canal – Panama!", or even longer ones such as "Doc, note: I dissent. A fast never prevents a fatness. I diet on cod!"; the Sator Square (Sator Arepo Tenet Opera Rotas) is another famous example.

Given a string $X[1 \cdots n]$, $n \geq 1$, find the longest palindrome that is a subsequence

Example: Given "c h a r a c t e r" $\rightarrow$ output "c a r a c"

Answer will be ≥ 1 in length; try "unidentified"

Strategy

$L(i, j)$: length of longest palindromic subsequence of $X[x \cdots j]$ for $i \leq j$.

```
1  function L(i, j) :
2  if i == j: return 1
3  if X[i] == X[j]:
4        if i + 1 == j: return 2
5        else : return 2 + L(i + 1, j - 1)
6  else :
7        return max(L(i + 1, j), L(i, j - 1))
```

Exercise: compute the actual solution

Analysis

As written, program can run in exponential time: suppose all symbols $X[i]$ are distinct.

$T(n)$ = running time on input of length n

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n-1) & n > 1 \end{cases}$$
$$= 2^{n-1}$$

Subproblems

But there are only $\binom{n}{2} = \theta(n^2)$ distinct subproblems: each is an (i, j) pair with $i < j$. By solving each subproblem only once. running time reduces to

$$\theta(n^2) \cdot \theta(1) = \theta(n^2)$$

where $\theta(n^2)$ is the number of subproblems and $\theta(1)$ is the time to solve each subproblem, given that smaller ones are solved.

Memoize $L(i, j)$, hash inputs to get output value, and lookup hash table to see if the subproblem is already solved, else recurse. (So we can fix the program above with a single line of code: the first line in the program should be "if L(i,j) has been computed in the past, return that value".)

Memoizing Vs. Iterating

1. Memoizing uses a dictionary for $L(i, j)$ where value of L is looked up by using i, j as a key. Could just use a 2-D array here where null entries signify that the problem has not yet been solved.

2. Can solve subproblems in order of increasing $j - i$ so smaller ones are solved first.

Optimal Binary Search Trees: CLRS 15.5

Given: keys $K_1, K_2, \cdots, K_n, K_1 < K_2 < \cdots < K_n$, WLOG $K_i = i$
weights $W_1, W_2, \cdots, W_n$
Find: BST T that minimizes:

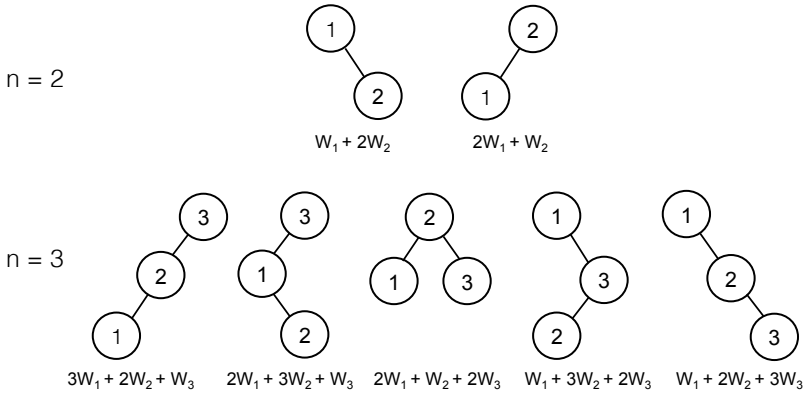
$$\sum_{i=1}^{n} W_i \cdot (depth_T(K_i) + 1)$$

Example: $W_i = p_i =$ probability of searching for K_i

Then, we are minimizing expected search cost. (This kind of thing is also useful in compression.)

Enumeration

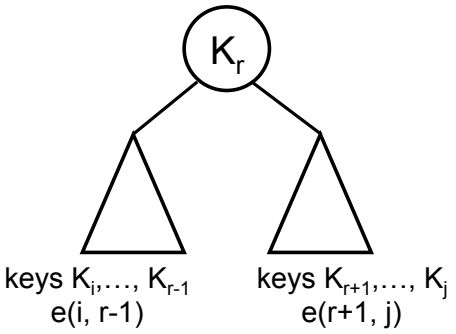
Exponentially many trees



Strategy

$W(i, j) = W_i + W_{i+1} + \cdots + W_j$

$e(i, j)$ = cost of optimal BST on $K_i, K_{i+1}, \cdots, K_j$
Want $e(1, n)$
Greedy solution?
Pick K_r in some greedy fashion, e.g., W_r is maximum.



greedy doesn't work.

DP Strategy: Guess all roots

$$e(i,j) = \begin{cases} W_i & \text{if } i = j \\ \min\limits_{i \le r \le j} (e(i, r-1) + e(r+1, j) + w(i,j)) & \text{else} \end{cases}$$

$+W(i, j)$ accounts for W_r of root K_r as well as the increase in depth by 1 of all the other keys in the subtrees of K_r (DP tries all ways of making local choices and takes advantage of overlapping subproblems)

Complexity: $\theta(n^2) \cdot \theta(n) = \theta(n^3)$

where $\theta(n^2)$ is the number of subproblems and $\theta(n)$ is the time per subproblem.

Alternating Coin Game

Row of n coins of values $V_1, \cdots, V_n$, n is even. In each turn, a player selects either the first or last coin from the row, removes it permanently, and receives the value of the coin.

Question

Can the first player always win?

Try: 4 42 39 28 25 6 17 8

Simple Strategy to Not Lose

Given coins with values $V_1, V_2, \cdots, V_{n-1}, V_n$:

1. Compare $V_1 + V_3 + \cdots + V_{n-1}$ against $V_2 + V_4 + \cdots + V_n$ and pick whichever is greater.
2. During the game we only pick from the chosen subset (we will always be able to!)

But how can we maximize the amount of money won assuming we move first?

Optimal Strategy

$V(i, j)$: max value we can definitely win if it is our turn and only coins $V_i, \cdots, V_j$ remain.

$V(i, i)$: just pick i.

$V(i, i+1)$: pick the maximum of the two.

$V(i, i+2)$, $V(i, i+3), \cdots$ this is a bit more difficult, but we can say:

$$V(i,j) = max\{\langle \text{range becomes } (i+1, j)\rangle + V_i, \langle \text{range becomes } (i, j-1)\rangle + V_j\}$$

Solution

We can't actually use $V(i_1, j)$ or $V(i, j-1)$, becasue we never see that step; the opponent plays. So, in the $V(i+1, j)$ subproblem with the opponent picking, we are guaranteed $min\{V(i+1, j-1), V(i+2, j)\}$ (because the less we have, the more the opponent has). Where $V(i+1, j-1)$ corresponds to opponent picking V_j and $V(i+2, j)$ corresponds to opponent picking V_{i+1}.

Thus, we have:

$$V(i,j) = max\left\{min\left\{\begin{array}{c} V(i+1, j-1), \\ V(i+2, j) \end{array}\right\} + V_i, min\left\{\begin{array}{c} V(i, j-2), \\ V(i+1, j-1) \end{array}\right\} + V_j\right\}$$

Complexity: $\Theta(n^2) \cdot \Theta(1) = \Theta(n^2)$