Recitation 14 – *Dynamic Programming!*

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2014)

April 15th, 2014

1 Announce Dynamically

- *RangeLab* is due on Wednesday! *DPLab* is coming out on Wednesday!
- Questions about homework or lecture?
- As a heads up, we're planning to have *two* labs after RangeLab instead of the usual one. More on this later.

2 Dynamic Review!

Q: What is DP?

A: Dynamic programming is an algorithmic technique to avoid needless recomputation of answers to subproblems. DP problems have two identifying characteristics :

- *Inductive/recursive*. The solution to the larger problem instance is composed from solutions to smaller instances of the same problem.
- *Sharing*. The solution to each smaller problem instance can be used by multiple larger instances. This is what differentiates DP from other inductive techniques like divide-and-conquer. Avoiding needless recomputation means that we reduce the overall work of our algorithm.

3 Dynamic Approach to a Dynamic Problem

3.1 Problem : Longest Palindromic Subsequence

Given a string *s*, we want to find the longest subsequence *ss* of *s* that is a palindrome (reads the same in both directions). The letters don't have to be consecutive.

Example: `QRAECDETCAURP` has inside it palindromes `RR`, `RADAR`, `RAEDEAR`, `RACECAR`, etc.

Q: How many palindromes could there be?

A: An exponential number.

Q: How do we keep track of all of them?

A: We don't. Instead, we simplify the problem space, and find the *length* of the longest palindrome. Doing this will help us identify the inductive structure of the problem and sharing between subproblems. Later, we can consider recovering the longest palindrome (or if you are feeling adventurous, count unique ones).

3.2 Solution

In this section we describe a general approach to DP problems. As we go along, we'll also write up the solution according to the structure we'd like you to follow for assignments in this course (like *DPLab!*). Solutions should have 4 components:

1. Define the subproblems you are working with.
2. The recurrences for the problem and their base case(s).
3. The final answer.
4. Runtime analysis.

Q: What's step one of coming up with a DP solution?

A: We want to be able to define our main problem instance in terms of smaller problem instances. This requires us to recognize the inductive structure of the problem.

Q: What are the base cases of being a palindrome?

A: A 1- or 0-length string.

Q: How do you get bigger palindromes from smaller ones?

A: Add the same letter to both ends.

If you are familiar with the BNF grammar, one way to express palindrome is

$$\mathsf{pal} := \emptyset \mid \ell \mid \ell\,\mathsf{pal}\,\ell,$$

where ℓ is a "letter" and $\emptyset$ denotes the empty string.

From the top-down approach, this translates into having two pointers into the string, a start point and an end point and checking whether the outer letters are the same.

Q: If they are, how do we proceed?

A: We're looking for the longest palindrome subsequence between the two pointers (inclusive). So we add 2 to the recursive call on the string between them. We can add these two letters to the outside of whatever palindromic subsequence was found on the inner string.

Q: What if they're not – how can we proceed?

A: One of the letters may still be part of a palindromic subsequence. We split into two branches. One moves the end point inwards and keeps the same start point, the other moves the start point inwards and keeps the same end point. Take the max of the results of these two calls.

At this point, we've pretty much determined the recursive structure of the problem. We can start to attempt to write up our solution. Formalizing our thinking will help us check that we're on the right track.

> Let $S[i,j]$ denote the substring between position i and j (inclusive) of the input string S, and $S[i]$ the character at position i.
>
> **Subproblems:**
> `LP[i,j]` is the length of the longest palindrome subsequence of the substring $S[i,j]$.
>
> **Recurrences:**
>
> $$\texttt{LP}[i,j] = \begin{cases} 2 + \texttt{LP[i+1,j-1]}, \text{ if } i < j \text{ and } S[i] = S[j] \\ \max(\texttt{LP[i,j-1]}, \texttt{LP[i+1,j]})), \text{ if } i < j \text{ and } S[i] \neq S[j] \\ 1, \text{ if } i = j \\ 0, \text{ otherwise} \end{cases}$$
>
> **Final answer:**
> $\texttt{LP}[0, n-1]$

Notice how the recurrences and final answer naturally fall off from the sub-problem definition. It's important to define your sub-problems correctly, clearly, and concisely, and you'll always want to spend some time on this.

We're now left with the runtime analysis.

Q: Argue for why we do not need to proceed to the second branch if we enter the first branch.
A: Take the longest palindromic subsequence found by the call `L[i,j-1]`. Replace the last letter of that subsequence with $S[j]$. The new subsequence has the same length as the original, and would have been found in the first branch. Likewise for `L[i+1,j]`.

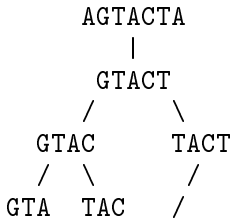
Q: How can we analyze the work of this DP solution?

To analyze work, we look at the number of sub-problem instances in a call to `LP[0,n-1]`, and the non-recursive work at each sub-problem. If we model this is as a DAG, this is the number of vertices in the DAG, and the work at each vertex.

Q: What's the sharing structure here?

A: The recursive calls share the **entire middle of the string**!

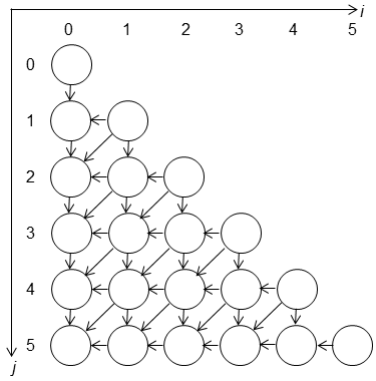
Let's look at the DAG for `AGTACTA`:



```
    / \  / \  /
  GT   TA   AC
  /\   /\   /\
 G   T    A   C
```

Q: How many vertices could there be in the worst case? That is, how many different arguments could there be to LP**? What does this mean for the work of** LP**?**

A: $\frac{n(n-1)}{2}$, choose two indices, one for the start and one for the end of the substring. Since each call to LP has constant non-recursive work, the total work is $O(n^2)$. The full DAG for this problem is the lower triangle of a matrix of sub-problem nodes for each (i, j) with $0 \leq i \leq j < n$, since we could end up making a call to LP with each such pair of arguments. The node (i, j) depends on (may call) $(i+1, j-1)$, $(i+1, j)$ and $(i, j-1)$.



Q: What is the span of LP**?**

A: If our implementation traverses the DAG in the most efficient way possible, then the span is simply the longest path in the DAG : $O(n)$.

3.3 Dynamic Implementations

In lecture, we covered two ways to implement DP solutions : top-down and bottom-up.

Q: What's a top-down DP solution to the LP **problem?**

A: In this approach, we write the recursive solution but use memoization to avoid recomputing shared function calls. For this, we use a function `memo` which takes a function to memoize, a memo table and an argument to the function. If the argument matches in the memo table, the result is returned. If not, the function is run and the result is stored in the memo table to be reused. Regardless, the memo table and the result are returned as a pair. This gives the code below.

```
(* lp : 'a seq -> int *)
let fun lp s =
  let fun lp' MT (i,j) =
      if (j-i <= 1) then j-i
      else (if (s[i] = s[j-1]) then let val (MT', move) = memo lp' MT (i+1, j-1)
                                    in (MT', 2+move)
```

```
                                    end
          else let val (MT', movestart) = memo lp' MT (i+1, j)
                   val (MT'', moveend) = memo lp' MT' (i, j-1)
               in
                 (MT'', Int.max (movestart, moveend)
               end
in
  lp' {} (0, |s|)
end
```

Q: What's the work of this code?

A: Since this mirrors the recursive definition of LP, this is as we argued above : $O(n^2)$ because there are $O(n^2)$ possible arguments to LP.

Q: What's the span of this code?

A: $O(n^2)$. We need to wait for the result of one branch before computing the next, since they share a memo table, so we can't parallelize them!

In practice, there are tricks we can pull to get memoization to work in parallel, but they're not pretty.

Q: What about a bottom-up solution?

A: A bottom-up DP solution does the opposite of what a top-down solution does; it starts at the leaves of the DAG and works up, computing nodes when the results from all of their dependencies are ready. In this way, we can compute independent nodes in parallel (eg., all leaves can be computed in parallel). This makes use of our advance knowledge of the DAG, to know in what order to compute the values and can exploit any available parallelism.

Q: In a bottom-up approach, what is the cost of a recursive lookup?

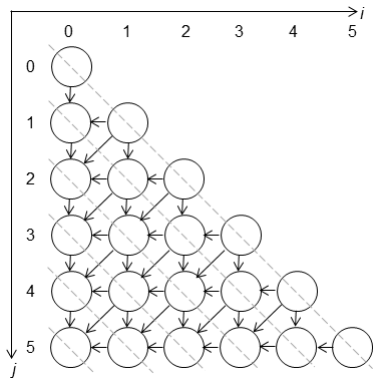
A: In the bottom-up approach, we will always run an instance of a function after the instances it may call, so if we store the results in a matrix M, the recursive calls simply become $O(1)$ lookups into M.

```
1  let fun lp' M (i, j) =
2    if (j − i ≤ 1) then j − i
3    else (if (s[i] = s[j − 1]) then 2 + M_{i+1,j−1}
4      else max(M_{i+1,j}, M_{i,j−1})
```

Now, we need to handle calling `lp'` with the right arguments in the right order.

Q: In what order should we compute the nodes to get correct results with the most parallelism?

A: Each node depends on/can only be evaluated after all the nodes above and to the right of it. The nodes on each top-left-to-lower-right diagonal are independent and can thus be evaluated in parallel.



The k^{th} diagonal consists of the $n-k$ nodes $(0,k),(1,k+1),...,(n-k-1,n-1)$. The following function computes the k^{th} diagonal in parallel, and then calls itself recursively to compute the next diagonal.

```
1  fun diagonals(M, k) =
2    if (k ≥ n) then M
3    else let
4      val M' = M ∪ {(i, k+i) ↦ lp'(M, (i, k+i)) : i ∈ {0...|s| − k − 1}}
5    in
6      diagonals(M', k+1)
7    end
```

Putting these functions together gives the bottom-up solution to the problem.

```
1  fun lp s
2    let
3      fun lp'...
4      fun diagonals...
5    in
6      diagonals({}, 0)
7    end
```

Q: What's the span now?

A: $O(n)$. The work and span to compute each node is $O(1)$, each diagonal is computed in parallel and there are $O(n)$ diagonals.

Q: What about the work? Did it change?

A: The work is still $O(n^2)$, since it's $O(1)$ times the number of nodes in the DAG, which is $O(n^2)$.

Q: What's the best case work of the top-down solution? What about for the bottom-up solution?

A: The best case for the top-down solution would be a string like `RADAR`, where we always enter the first branch of the recurrence. Work for the top-down solution would be linear here. However, because the bottom-up solution always computes all sub-problem instances, work of the bottom-up solution would stil be $O(n^2)$. We call top-down DP a lazy solution, and bottom-up an eager one.

4 More Dynamic Problems

In this section, we tackle the classic Longest Increasing Subsequence problem with our newfound DP chops. As usual, a subsequence of $A = (a_1, a_2, ..., a_n)$ is $(a_{i_1}, a_{i_2}, ..., a_{i_k})$ where $i_1 < i_2 < ... < i_k$. A subsequence is increasing if $A_{i_j} < A_{i_{j+1}}$ for $1 \leq j < k$.

Given the sequence A, we want to find the longest increasing subsequence (the one with maximum k). We write $n =$ `(length A)` for simplicity.

More intuitively, we will cross out some numbers from A. We want the remaining numbers to be increasing. What is the fewest number of numbers we need to cross out?

One idea is a brute force solution: generate all possible subsequences, filter out the increasing ones, and select the longest one. This is prohibitively slow and painful to code; we can do better.

4.1 Attempt 1 : $O(n^2)$

The key observation is that if we take off the last element from an increasing subsequence, the result is still an increasing subsequence. For each i, we will keep track of the length of the longest increasing subsequence that *ends* with $A[i]$; call this quantity $L[i]$.

Q: Suppose we knew $L[i]$ for every i. How could we compute the answer?
A: $k = \max_i L[i]$ (NOT $L[n-1]$ or similar). Notice that we need the recursive results for every i to compute the answer. **Q: What is the base case?**
A: $L[0] = 1$ because the longest increasing subsequence ending with the first element contains just the element itself.

Q: Given $L[j]$ for all $j < i$, how can we compute $L[i]$?
A: $L[i] = 1 + \max_{j<i, A[j]<A[i]} L[j]$. We quantify over all $L[j]$ where $A[j] < A[i]$ because we can only add $A[i]$ onto the subsequence ending at $A[j]$ if $A[i] > A[j]$. We take the max, and add one to indicate that $A[i]$ is added.

What is the runtime of this solution? As always, the work is the sum of the non-recursive work to calculate each cell in the table. This is $\sum_{i=0}^{n} i = O(n^2)$. We also use $O(n)$ space.

4.1.1 Example

Q: Suppose we have $A = (2, 4, 1, 7, 3)$ where $n = 5$. Compute L for each i

A: By definition of L we have, $L = (1, 2, 1, 3, 2)$, which can be computed by applying the above formula starting from $i = 0$ to $n - 1$.

The maximum value of L is the length of the longest increasing subsequence. In this case it is 3.

4.2 Extracting the Answer

This gives us the length of the longest subsequence; if we want the subsequence itself, we need to do more work. This is a very common problem in DP; it is a pain. There are two general solutions:

Since we are building up the subsequences one-by-one, and we don't change a subsequence after it's found, we can separately keep track of an array of *parents*, where parent$[i]$ is the last index of the subsequence we added $A[i]$ to. Once we have the i that maximizes $L[i]$, we can start at that index in *parents* and follow the parent pointers backward to read off the subsequence. We added $A[i]$ to the subsequence ending at parent$[i]$, which was added to the subsequence ending at parent[parent$[i]$], etc. So our subsequence is the reverse of i, parent$[i]$, parent[parent$[i]$], parent[parent[parent[i]]], etc.

The other idea is to run the DP table "in reverse". You know you must have gotten to a length $L[i]$ subsequence at i from a length $L[i]-1$ subsequence at j with $A[j] < A[i]$ and $j < i$, so just look for one! Repeat until you get to the start.

4.3 Faster! $O(n \log n)$

We can do even better than $O(n^2)$ for this problem. Recall that we are looking for the longest subsequence so far that uses an element smaller than us. Two requirements: "longest" and "smaller". We can take care of one of them by sorting the list of options so far: for example, we could sort our list from longest subsequences so far to smallest. This would do better on average, but still might be $O(n^2)$ in the worst case (e.g. a list sorted in decreasing order).

To really save time, we need some insight: we just need to keep track of the smallest element that can end a subsequence of a given length, for each possible subsequence length. Let `L[len]` be the smallest element that ends a length `len` subsequence (that we've seen so far). **Q: The key observation is that L is sorted; if $x < y, L[x] \leq L[y]$. Why is this so? A:** This is because any number that ends a subsequence of length y also ends a subsequence of length x (just take the last x elements of the longer subsequence).

So we can turn our linear scan for which subsequence $A[i]$ should update into a binary search: what is the largest `len` (out of the subsequences that end before position i) for which `A[i] > L[len]`? Once we find that value of len, we see if the subsequence we can make with `A[i]` is better: that is, set `L[len+1] = min(A[i], L[len+1])`. The information in L only ever corresponds to subsequences we've seen already, so if we traverse A from left-to-right we are always allowed to append $A[i]$. You should check that this leaves `L` sorted. To finish the algorithm, we want to initialize every element of of `L` to infinity (some constant larger than any element of `A`) to indicate that we don't have any subsequences yet.

This is tough; give it a moment to sink in. Think about how we could extract the actual subsequence here; it is harder than it was before.