Recitation 14 – *Dynamic Programming!*

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

November 18th, 2014

1 Announce Dynamically

- *H*ow was RangeLab? *DPLab* just came out!
- Questions about homework or lecture?
- Just *CilkLab* after *DPLab*, then you're done!

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.

How do we use these subproblems?

- We model our computation scheme as a **Directed Acyclic Graph**, a **DAG**, where each of our subproblems is a node, and we have a directed edge from problem A to B if the result of A depends on first computing B.
- Afterwards, we can solve a given problem by starting at the node representing our problem, S, and visiting all nodes in our DAG reachable from S. We call this the Top-Down approach.
- Alternatively, we can start from the bottom of the graph and work our way back up towards subproblems that depend on the current subproblem. We call this the Bottom-Up approach.
- Which method you use often does not make a difference except for space efficiency; however, there are times when one is better than the other due to data structure limitations, real-world implementations, or other limitations.

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:

Q: How do we keep track of all of them? (Trick question!

A:

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:

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

A:

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

A:

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:

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

A:

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:

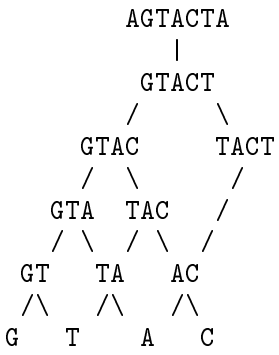
Q: How can we analyze the work of this DP solution?

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?

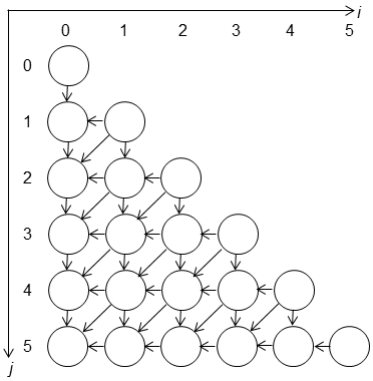
A:

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



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:



Q: What is the span of `LP`?

A:

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? You can describe what is going on in the solution code for help.

A:

```
(* 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:

Q: What's the span of this code?

A:

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:

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

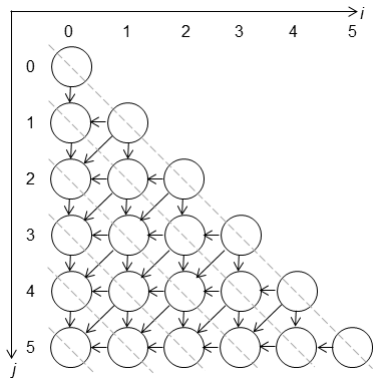
A:

```
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:



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:

Q: What about the work? Did it change?

A:

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

A:

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:

Q: What is the base case?
A:

Q: Given $L[j]$ for all $j < i$, how can we compute $L[i]$?
A:

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:

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:**

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.

5 Bonus

How could you implement parallelism for the top-down approach? Memoization in parallel gets tricky. How will you handle fetching unknown results? What type of system level locks would you have to use to synchronize that fetching? Do you notice any inefficiencies in the naive approach? If you're interested, look up *futures and promises*.