Recitation 13 — Dynamic Programming

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2013)

November 20, 2013

1 Dynamic Programming

Dynamic programming is a technique to avoid needless recomputation of answers to subproblems.

Q: When is it applicable?

A: When the computation DAG has a lot of sharing. Or when subproblems *overlap*.

1.1 Example: 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`, `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. Later, we can consider recovering the longest palindrome—or if you are feeling adventurous, count unique ones.

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

A: 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: 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. This is what allows for non-contiguous subsequences, since we'll be leaving off either the current start or end letter, but may be adding more letters on the outside of this result higher up in the call stack.

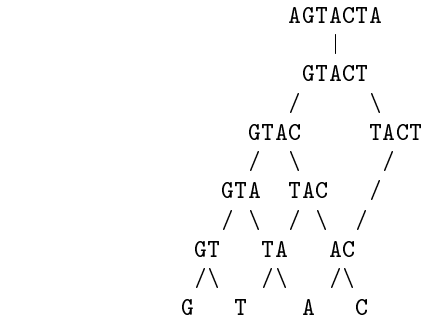
```
(* lp : 'a seq -> int *)
fun lp s =
  let fun lp' (i,j) =
      if (j-i <= 1) then j-i
      else (if (s[i] = s[j-1]) then 2 + lp'(i+1,j-1)
           else Int.max(lp'(i+1,j), lp'(i,j-1)))
  in lp'(0, |s|)
  end
```

Intuitively, when we memoize, we'll want our table to be indexed by i and j so we can easily store and look up the longest palindromic subsequence between those two indices.

Q: What's the sharing structure here?

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

Let's look at an example:



With proper memoization, we only need to consider the number of vertices in the DAG of recursive calls and the work at each vertex to find the total work.

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'` (ignoring the recursive calls) is constant work, the total work is $O(n^2)$.

Now we need to find out how to take advantage of this sharing. Recall that there are two ways of doing this: top-down and bottom-up. We looked at the top-down approach in lecture. 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. 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 2 + memo lp' MT (i+1,j-1)
           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 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!

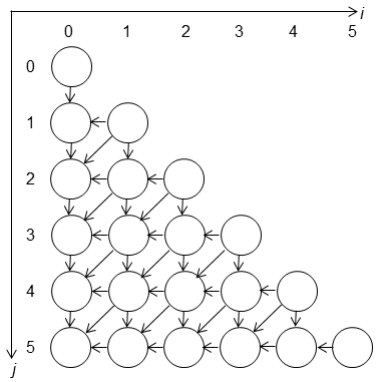
In practice, there are tricks we can pull to get memoization to work in parallel, but they're not pretty. Instead, let's look at a different way to achieve the goal of dynamic programming.

2 The Bottom-Up Approach

As its name suggests, the bottom-up approach is the opposite of the top-down approach. We know that attempting to run the root is going to result in recursive calls that it needs to complete, so we can start by computing the values at the leaves of the DAG and working up, computing nodes when the results from all of their dependencies are ready. This requires recognizing the structure of the DAG ahead of time, to know in what order to compute the values and exploit any available parallelism.

Q: What does the full DAG look like for the Longest Palindromic Subsequence problem?

A: The lower triangle of a matrix of 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 $(i, j-1)$.



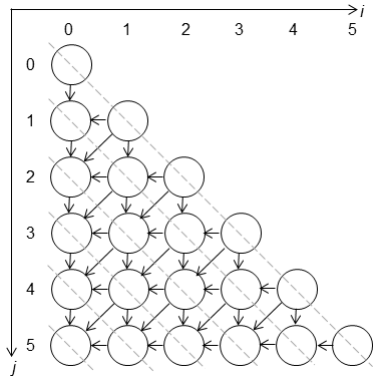
In the bottom-up approach, we will always run an instance of a function after the instances it may call, so if we store the results in a matrix `M`, the recursive calls simply become $O(1)$ lookups into M.

```
1  let function 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: We need to evaluate a node after all the nodes above and to the right. The nodes on each top-left-to-lower-right diagonal are independent and can 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  function diagonals(M, k) =
2    if (k ≥ n) then M
3    else let
4        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  function lp s
2    let
3      function lp'...
4      function 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 $O(n^2)$.

3 Longest Increasing Subsequence

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 $1 \le j < k$.

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.

3.1 $O(n^2)$

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).

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.

3.1.1 Example

Suppose we have $A = (2, 4, 1, 7, 3)$ where $n = 5$.

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.

3.2 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 subsequence one-by-one, we can just keep track of which subsequence we added $A[i]$ to. Call it parent$[i]$. The final element of the subsequence is the i that maximizes $L[i]$. Then we can 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.

3.3 $O(n \log n)$

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. That is, we want to keep track of "what is the smallest value that ends a subsequence of length 3? Oh, it's 5". Let `L[len]` be the smallest element that ends a length `len` subsequence (that we've seen so far). The key observation is that L is sorted: if $x < y, L[x] \leq L[y]$. 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` so that `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])`. 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.