Recitation 13 — Dynamic Programming

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

November 28, 2012

Today's Agenda:

- Announcements
- Dynamic Programming

1 Announcements

- HW 8 is due on Friday.
- HW 9 is a written homework and will be due next Friday.
- Questions about homework, class, life, universe?

2 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*.

So far, we haven't told you how to actually take advantage of overlapping solutions to get the efficiency gain. What we're doing right now is just steps 1 and 2 of DP, recognizing the inductive structure and the existence of the overlap. Let's review.

In class, we looked at the subset sum problem: given a set S and a number k, is there a subset of S whose elements sum to k?

Here's the code that produces a yes-or-no answer (remember, no actual DP yet):

```
(* SS : int list -> int -> bool *)

fun SS(S, k) =
  case (S, k) of
    (_, 0) => true
  | ([], _) => false
  | (s::S', k) =>
      if (s > k) then SS(S', k)
      else SS(S',k) orelse SS(S',k-s)
```

Let's tweak this a little to return the actual subsets rather than blind ourselves with booleans.

```
(* SS : int list -> int -> int list option *)
```

```
fun SS(S, k) =
  case (S, k) of
    (_, 0) => SOME []
  | ([], _) => NONE
  | (s::S', k) =>
      if (s > k) then SS(S', k)
      else case(SS(S',k), SS(S',k-s)) of
        (SOME L, _) => SOME L
      | (_, SOME L) => SOME (s::L)
      | _ => NONE
```

Another thing we might be interested in is how *many* solutions there are. This is just another minor tweak:

```
(* SS : int list -> int -> int *)

fun SS(S, k) =
  case (S, k) of
    (_, 0) => 1
  | ([], _) => 0
  | (s::S', k) =>
      if (s > k) then SS(S', k)
      else SS(S',k) + SS(S',k-s))
```

2.1 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 front and back). 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. Ugg.

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 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.

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

A: We can move our starting position or our ending position. Take the max of these two subcalls. In code:

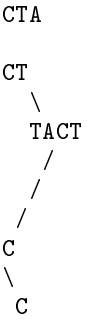
```
(* lp : 'a seq -> int *)
fun lp s =
  let fun lp' (i,j) =
      if (j-i <= 1) then j-i
      else (if s[i] = s[j-1] then 2 + lp'(i+1,j-1)
           else Int.max(lp'(i+1,j), lp'(i,j-1)))
      in lp'(0, |s|)
      end
```

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'`?

A: $n(n-1)/2$, choose two index for the start and end of the substring.

Since each call to `longest'` is constant work, the total work is $O(n^2)$.

Q: What is the span?

A: $O(n)$—since each time we invoke a subproblem, $j-i$ is at least one smaller and $0 \leq j-i \leq n$.

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 $A_{i_n} < A_{i_{n+1}}$ for $1 \leq n < k$.

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 to 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 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: Given $L[j]$ for $j < i$, how can we compute $L[i]$?
A: $L[i] = 1 + \max_{j<i, A[j]<A[i]} L[j]$. We can add $A[i]$ onto the subsequence ending at $A[j]$ if $A[i] > A[j]$.

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.

3.2 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$[i]$. We find the final element of the subsequence when extracting k (it is the i that maximizes $L[i]$). Then we can read off the subsequence we added $A[i]$ to (it ended at parent$[i]$, and the element before it was parent[parent$[i]]$, etc.) So our subsequence is 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. This often leads to code duplication and is slower than the first method ($O(n^2)$ instead of $O(n)$), but it usually doesn't cost more than the DP cost in the first place so it's OK.

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 subsequenc 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 experss 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.