1 Announcements

- HW ?? will be handed back today
- HW 8 will go out tomorrow (Thursday) and will be due Tue after Carnival.
- Questions about homework, class, life, universe?

2 Exam II Debriefing

Refer to the exam :)

3 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 yesterday, 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.

```ml
(* 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:

```ml
(* 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))
```

3.1 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: \( \text{QRAECETCAURP} \) has inside it palindromes \( \text{RR, RADEAR, RACECAR, RADEAR} \), 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

\[ \text{pal} := \emptyset \mid \ell \mid \ell \text{pal} \ell, \]

where \( \ell \) 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:

```ml
(* 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:

```
AGTACTA
    |
   GTAC
    / \  
   GTAC TACT
    / \  
   GTA TAC
    / \  
   GT TA AC
    / \  
   G T A C
```

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 `longest'?
A: \( n(n-1)/2 \), since \( i \) and \( j \) can range between 0 and \( n-1 \), and \( i \leq j \).

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 \( 0 \leq j - i \leq n \).

**Exercise**: how could we tweak this code to return the actual palindrome?