Recitation 13 — Longest Increasing Subsequence

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

April 16, 2012

Today's Agenda:
- Announcements
- Longest Increasing Subsequence
- Palindromes

1 Announcements

- HW 8 is due the Tuesday after Carnival. Start now!
- Questions about homework, class, life, universe?

2 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 $\underline{\text{increasing}}$ if $A_{i_n} < A_{i_{n+1}}$ for $1 \le n < k$.

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.

2.1 $O(n^2)$

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.

2.2 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.

2.3 $O(n\log(n))$

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.

2.4 Palindromic Decomposition

The problem: given a string `S`, how can you split into the minimum number of contiguous palindromes? Examples: "abc" becomes "a b c", "aaba" becomes "a aba" (NOT "aa b a"), "acabbada" becomes "aca bb ada" (NOT "a c abba d a"). Coming up with an optimal split seems hard; these examples show you can't just take the longest palindrome or some other easy trick. What should we do when finding the best option seems hard? Try them all with DP!

First, some notation: I write `S[i..j]` for the substring of `S` starting at `i` and ending at `j` (inclusive).

What is there shared substructure in this problem? It is very natural: we'll take out some palindrome and then recurse on either side. Which one? Maybe the longest one?

No! Why? Fisrt, the reason this sounds appealing is something like "to prune the search space faster", but DP doesn't care about pruning the search space: it always searces through all possible options. The goal is to decrease the number of subproblems. If you delete any old palindrome repeatedly, you could end up with any substring of S as a subproblem. But if you always take a palindrome that uses the first character of S, you only end up with prefixes of S as subproblems. That saves us a factor of n.

So our subproblems with be "what is the minimum number of contiguous palindromes you could split the last `i` characters of S" into? Call this `PAL[i]`. Then the answer is just `PAL[|S|]`. And it is easy to see that $PAL[i] = 1 + min_{j<i,\text{S[i..j-1] a palindrome}} PAL[j]$. (we are just enumerating over the length of the first palindrome).

What is the runtime of this algorithm? There are n subproblems; how much work do we have to do for each one? $O(n^2)$! Every j we check takes $O(n)$ because checking whether $S[i..j]$ is a palindrome takes $O(n)$ work. So the whole algorithm is $O(n^3)$.

If only we could check whether a substring of S was a palindrome faster. DP comes to rescue! This sort of DP-within-a-DP is common. This DP is easy; we already know the subproblems are "Is S[i..j] a palindrome?". Call this `is_pal[i][j]`. If the length of the substring to check is ≤ 1, the answer is always yes. Otherwise, the answer is "yes" if the end character match ($S[i] == S[j]$) and the rest of the string is a palindrome (i.e. `is_pal[i+1][j-1]`). The non-recursive work here is constant, so we can answer all of these subproblems in $O(n^2)$ (because there are $O(n^2)$ subproblems).

So our whole algorithm now runs in $O(n^2)$ work instead of $O(n^3)$.

One last detail: we now know *how many* palindromes we need, but we wanted to know what they actually are! Again, the method of parent pointers works well: for every i, keep track of the j that actually minimizes the expression (this is the palindrome we want to delete at this stage). This only requires $O(n)$ extra space.

If you want to try and code up the solution to this problem, this site will run tests on your code to see if it is right: (you have to adhere to a specific input/output format, and only Java, C, or C++ solutions can be tested)

```
http://acm.timus.ru/problem.aspx?space=1&num=1635
```