Dan Ports
6.046 PS 4-1

12

Your 2Pm
Collab: ∅

a. We will build an array $B[1..n]$, where each $B[i]$ contains the length of the longest increasing subsequence that ends with $A[i]$, and the index of the previous element in the subsequence (or nil). We begin by initializing $B[1] = (1, nil)$, and proceed to iterate through the array. At each iteration, we compare $A[i]$ to all preceding entries of A, and find the k such that $A[k] < A[i]$ and $B[k].length$ is maximized, then set $B[i].length$ to $B[k].length + 1$ and $B[i].prev \leftarrow k$. Once we have done this for all elements of A, we find the longest increasing subsequence by finding the element of B whose length element is highest. We then reconstruct the indices for output by following the chain of prev pointers.

dynamic prog.

Proof of correctness: We note the recurrence relation where $f(n)$ is the length of the longest increasing subsequence ending in $A[x]$:

$f(n) = 1 + \max\{ f(i) \mid i < n \text{ and } A[i] < A[n] \}$

The given algorithm uses this recurrence to build up the elements of B, we can easily show the invariant that at the ith iteration of the loop, the first i values of B are correct. Thus at the end of the loop, B contains the correct values for the least increasing subsequences ending with each element, the algorithm returns the largest of these, which is the correct result.

Runtime analysis: The loop to build B requires $\Theta(n)$ iterations, each of which takes $\Theta(n)$ time because it must scan the previous elements. So this step is $\Theta(n^2)$. It also takes $\Theta(n)$ to find the maximum. The algorithm is $\Theta(n^2)$

b. From part a we can generate the array $B[1..n]$ of longest increasing subsequences that end on each element of A, with lengths and back-pointers. We can trivially modify the algorithm to generate a similar array $C[1..n]$ of longest decreasing subsequences that start with element $A[i]$, with lengths and back-pointers. Simply reverse array A, apply the aforementioned loop algorithm, and reverse the results to find the longest decreasing subsequences. It is

Next we note that a bitonic subsequence is an increasing sequence that ends in element A[t] followed by a decreasing sequence that begins with A[t]. This follows from the definition. Since we have the longest increasing and decreasing subsequences, we simply need to find the optimal choice for t. For a given t, the longest bitonic subsequence has length B[t].length + C[t].length − 1 (the length of the increasing sequence, the length of the decreasing sequence, minus 1 because A[t] should only be counted once). So we evaluate this quantity for all t between 1 and n, and find the maximum. We then generate the output by following the chain of pointers in B[t] and C[t]. This takes $\Theta(n^2)$ time, because generating the arrays B and C takes $\Theta(n^2)$ (from part a) and finding the maximum takes $\Theta(n)$.

c. To find the longest bitonic sequence, we iterate through the array, ignoring monotone sequences and adding new elements when the sequence switches between increasing and decreasing and vice-versa, as in this pseudocode:

Create a list L of output indices

Initialize L containing only index 1

```
for i = 2 to n
{
  if length(L) is even
  {
    if A[i] > A[i-1]
      add i to L
    else
      replace last element of L with i
  } else {
    if A[i] < A[i-1]
      add i to L
    else
      replace last element of L with i
  }
}
return L
```

This algorithm has time complexity $\Theta(n)$ because each iteration of the loop requires constant time, and the loop runs $\Theta(n)$ times.

To prove correctness, we show the invariant that on each iteration of the loop, L

last item of L is maximized if L has an odd number of items and minimized if L has an even number. This is trivially true for L=1 since we initialize L to 1. Suppose it holds for i-1. We show that the algorithm maintains the invariant for i. There are two cases: L has either an even or odd number of items. If even, then, since we have the longest pytonic subsequence from the i-1 elements, and the last element is minimized, the only way we can make the pytonic subsequence longer is to add a larger element. This is only possible if A[i] > A[i-1], in which case the invariant is maintained by adding i to the pytonic subsequence, giving the longest possible pytonic subsequence in A[1..i]. Otherwise A[i-1] ≥ A[i], in which case there is no way to make the pytonic subsequence longer (if there were, then the current pytonic subsequence would not be ideal, violating our assumption). So by replacing the last item i-1 with the smaller element i, we have produced a maximum-length pytonic subsequence with last element minimized (from our assumption and the inequality A[i-1] > A[i]). In the case that L has an odd number of elements, we can show that the invariant is maintained by using the same argument

We have thus proven the maintenance of this invariant in all cases. When the loop terminates $i = n$ and thus L contains the maximum-length pytonic subsequence in $A[1..n]$ by the invariant, which is the correct result.

d. We can optimize the solution to part a by noting that we do not need to maintain the full array of longest increasing subsequences in the first i elements of A. In particular, if we have two subsequences a and b, and length(a) $\geq$ length(b) and last-element(a) $\leq$ last-element(b), we can discard b because any subsequence beginning with b can be transformed to one of equal or longer length by replacing b with a. So we can place the subsequences in order based on the value of the last element, and there will only be one subsequence for any value of the last element. We can store these subproblem results (the ones not discarded) in a balanced tree. This changes the algorithm only slightly: On each iteration of the loop, rather than comparing the new element $A[i]$ to all previous elements and finding the one that leads to the longest subsequence, we find the subsequence

value less than A[i] and create a new subsequence by adding A[i] to it. This search takes $\Theta(\lg n)$ time. We then check whether the subsequence in the tree with last element value less than or equal to A[i] has length less than the new subsequence we just created. If so, then we remove that old subsequence and add the new one; otherwise the new subsequence is not as good and we discard it. This is a sequence of balanced-tree operations that require $\Theta(\lg n)$ time, so each iteration of the loop requires $\Theta(\lg n)$ time, and there are n iterations, so the algorithm is $\Theta(n \lg n)$.

The proof of correctness is essentially the same as in part a, with the use of the pruning property mentioned above.

Since this algorithm solves part a in $\Theta(n \lg n)$ time, it can also solve part b in $\Theta(n \lg n)$ time since part b uses the same algorithm for generating its arrays.

Dan Ports
6.046 PS4-2

8/10

Your 2Pn
Collab: ∅

We will solve this problem with a greedy algorithm. First, sort the socks according to the number of holes, using mergesort or quicksort. Call the resulting sorted array (in increasing order) $h'[1..2n]$. Then iterate through this array, matching $h'[1]$ with $h'[2n]$, $h'[2]$ with $h'[2n-1]$, $h'[i]$ with $h'[2n-i+1]$.

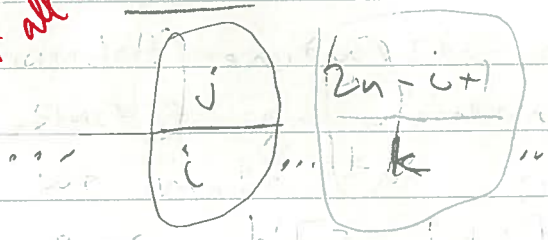
This requires $\Theta(n \log n)$ runtime to sort and $\Theta(n)$ to match, so $\Theta(n \log n)$ overall. ✓

Proof of correctness: Suppose the greedy algorithm does not give the correct result, that is, it must differ from the optimal solution at some point. Suppose that, with the larger socks in decreasing order, the first difference is in the pair with larger sock $h'[i]$. The greedy algorithm matches this with sock $h'[2n-i+1]$; suppose the optimal solution matches it with some $h'[j]$. We will show that the optimal solution can be rearranged

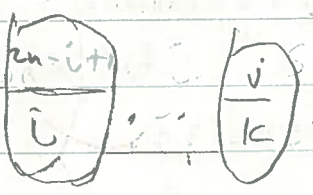
smaller socks were already used in earlier pairs by the greedy algorithm, and this is the first location where they differ. Similarly, $h'[2n-i+1] \leq h'[k]$, where $h'[k]$ is the sock currently matched with sock $h'[j]$ in the optimal solution. By the same reasoning, $h'[i] \geq h'[j]$ and $h'[i] \geq h'[k]$ because all socks larger than i have already been used. So we swap $h'[j]$ and $h'[2n-i+1]$ in the optimal solution:

Wasn't $h'[i]$ paired with $h'[j]$? so what is k after all?

Before swap



After



We show that this swap does not increase the maximum pairing M. By definition, the two original pairings were not greater than M:

$$h'[j] + h'[i] \leq M \; ; \; h'[2n-i+1] + h'[k] \leq M$$

Using these and the inequalities above, we find

$$h'[2n-i+1] + h'[i] \leq h'[j] + h'[i] \leq M$$

and $h'[j] + h'[k] \leq h'[j] + h'[i] \leq M$.

So the two new pairings do not exceed the old total value of M, and

the optimal solution is still an optimal solution. We have now shown we can transform an optimal solution that matches the greedy solution for the first $i-1$ elements into one that matches for i elements. Inductively, we can continue this process until we have transformed the optimal solution into the greedy solution. Thus the greedy solution is optimal, and we have proven correctness.

But k and j may be in the elements already settled - how do you consider this case?

(If the number of holes in each sock satisfies the appropriate conditions, we might be able to use radix sort and solve the problem in $\Theta(n)$ time, but it isn't given in the problem description that this is the case)

Dan Ports

6.046 PS4-3

Yoav ZPM

Collab: ∅

We are given a graph of vertices and edges (V, E) with all edges distinct. Suppose this graph can have two minimum spanning trees, A and B. We show that this leads to a contradiction. Since A and B are distinct MSTs, there must be some edge e in A that is not in B. (We know that $A \neq B$, so either there is an edge in A that is not in B, or there is an edge in B that is not in A. If the latter then there must also be an edge in A that is not in B, or one of them would not be a MST).

So take $B+e$. This graph has a cycle. Choose some edge f that is a part of the cycle and not in A (there must be one because A is acyclic). f must have a higher weight than e because all edge weights are distinct and e is in the MST A while f is not. So the weight of $B+e-f$ is less than that of B.

huh?

But $B+e-f$ is a spanning tree, since adding e creates a cycle and removing f breaks it. So $B+e-f$ is a spanning tree

∴ This is a contradiction, so there can only be one MST.

Dan Ports
6.046 PS4-4

10/10

4 hours 2 pm
collab: Ø

a. First we show that this problem has optimal substructure. Suppose that we have an optimal solution for making change for n cents. If we remove one coin, then we have an optimal solution for $n - x$ cents, where x is the denomination of the removed coin. If this were not an optimal solution, then there is a better solution for making $n-x$ cents change, and by adding the removed coin to the better solution, we would have a better solution for n cents than the one we supposed was optimal, a contradiction.

Next we show that the problem has the greedy-choice property. Consider an optimal solution to the problem. Suppose the greedy solution differs from the optimal one. Then consider the highest-denomination coin at which the optimal and greedy solutions differ. Call the denomination of this coin d_i, and let X_i be the number of this coin in the greedy and y_i the number in the optimal soln.

$X_i > y_i$, since the greedy solution takes as

We can first note that the optimal solution can contain no more than 4 pennies (if there were 5 or more, we could replace them with a nickel and have a better solution), and similarly no more than one nickel and two dimes. Since the greedy and optimal solutions first differ at the coin of denomination d, and the greedy solution has at least one more coin, we must make up at least d cents change using smaller denominations. Consider the cases for d:

$d = 25¢$: From above, we cannot have more than 2 dimes and 1 nickel. This allows us to make 25¢ change, but the solution is non-optimal because using a quarter would have all be 2 less coins. This is a contradiction.

$d = 10¢$: We cannot have more than 1 nickel and 4 pennies, so we cannot make 10¢ or more change, and the solution does not work. This is a contradiction.

$d = 5¢$: We can only have 4 pennies in an optimal solution, so we reach another contradiction.

✓ $d = 1¢$: If all higher-denomination coins are the same and the greedy and optimal solutions differ only in number of pennies, then they do not add to the same amount, a contradiction.

15

b. From part a, this problem has optimal substructure (that proof did not depend on any specific set of coin denominations). We note the following recurrence, where $c(n)$ is the optimal number of coins required to make n cents change, and $d_1 \dots d_k$ are the denominations.

$$c(x) = \begin{cases} 0, & x = 0 \\ \min \{ c(x - d_i) \mid d_i \le k, \; x \ge d_i \} + 1 \end{cases}$$

The proof of this recurrence is obvious.

So we can compute the optimal change for n cents using a dynamic programming approach: we create an array A and initialize $A[0] = 0$. Then, for each i from 1 to n, we try all values j from 1 to k to find the j that minimizes $A[i - d_j]$ (and satisfies $i - d_j \ge 0$, of course). We then set $A[i] = 1 + A[i - d_j]$ and also include a backpointer to $A[i - d_j]$ and the value of the denomination d_j. When we are done filling the array $A[0..n]$, we follow the chain of backpointers from $A[n]$, outputting the denomination of each coin at each step in the chain.

The correctness of this algorithm can

follows the recurrence above.

The runtime of this algorithm is $\Theta(kn)$. There are n iterations of the loop, and at each one it must find the minimum of a set of $\Theta(k)$ elements. ✓

+5