CS 180 - Algorithms and Complexity
Homework 5

Jayant Mehra
Discussion 1D

November 15, 2018

Solution 1. The closest pair algorithm runs in $O(n \log n)$ time. The initial sorting of the points by their x and y coordinates takes $O(n \log n)$ time.

Then, at each step we divide the input into 2 parts recursively and find the closest pair of points in both sides. Thus, there are $\log n$ levels of division. The combine/merge part of the algorithm is tricky and is performed in linear time.

Let δ be the smaller of the two minimum distances from the two sides. While merging we need to find two points, one on either side whose distance is less than δ.

Let L denote a vertical line equal to the rightmost x coordinate of the left side of the division. If there exists two points, one on either sides, whose distance is less than δ, then they must lie within δ of this line, L. Thus, this is how we manage points in the x-dimension.

If there exist two such points whose distance is less than δ then they must lie within 15 positions of each other in the list sorted according to the y-coordinates of points in the region described above. Finding these points is a linear time operation. Finding the distance is nothing but comparing all these points which is a constant time operation. This is how we manage and search points in the y-dimension.

As noted earlier the sort takes $O(n \log n)$ time. Then the algorithm is similar to merge sort wherein at each step we divide our search space into two, thus giving us $\log n$ levels. The combining part is linear. Thus, $O(n \log n)$.

Solution 2. The following is the solution to problem 4 on page 315 of the textbook.

(a)

City	*Month 1*	*Month 2*
NY	*1*	*30*
SF	*50*	*2*

M = 100

The optimal solution would be NY NY with a cost of 1 + 30 = 40. However, the algorithm given would produce NY SF which has a cost of 1 + 2 + 100 = 103. Thus, algorithm **does not** produce the optimal plan.

(b)

City	*Month 1*	*Month 2*	*Month 3*	*Month 4*
NY	*5*	*50*	*5*	*50*
SF	*50*	*5*	*50*	*5*

M = 1

The optimal solution is NY SF NY SF with a total cost of 5 + 1 + 5 + 1 + 5 + 1 + 5 = 23. Any other plan would be greater than 50. Thus, the optimal plan requires us to move thrice.

(c) The following algorithm returns the optimal cost:

```
1.  optimalCost(N[1...n], S[1...n], M)
2.      cache[1...n + 1]
3.      cache[1] ← 0        // base case
4.
5.      for i ← 1 to n
6.          tempNY ← cache[i] + N[i]
7.          tempSF ← cache[i] + S[i]
8.          if previous cache entry was SF
9.              tempNY ← tempNY + M
10.         if previous cache entry was NY
11.             tempSF ← tempSF + M
12.
13.         cache[i + 1] ← min(tempNY, tempSF)
14.     return cache[n + 1]
```

Algorithm Explanation The algorithm follows the dynamic programming paradigm wherein it caches all recurrences in the cleverly named cache array. At each point we can make two choices - remain where we are or move to the other city. The algorithm makes this choice based on the previous choice. If it decides to stay, it adds the city's cost to the previously calculated amount. If it decides to move, it adds the new city's cost plus the moving cost to the previously calculated amount. Note all arrays are 1-indexed and the cache has $n + 1$ elements because the first element is the base case for a 0 sized input. The recurrence is the following:

$$\text{OPT}(i) = \min(\text{OPT}(i-1) + N_i + \text{If Moving, then M else 0}, \text{OPT}(i-1) + S_i + \text{If Moving, then M else 0})$$

Algorithm Analysis The algorithm is linear in the size of the input i.e. O(n). There is just one for loop which goes over all elements of N and S.

Solution 3. The following is the solution to problem 6 on page 317 of the textbook.

```
1.  prettyPrint(W[1...n], L)
2.      cache[1...n + 1]
3.      cache[n + 1] ← 0
4.
5.      for i ← n to 1
6.          temp ← ∞
7.          for j ← i to the word position which fits limit L ∧ is less than n
8.              if temp^2 > (cache[j + 1] + L − (Σ_{k=i}^{j−1} c_k + 1) − c_j)^2
9.                  temp ← cache[j + 1] + L − (Σ_{k=i}^{j−1} c_k + 1) − c_j
10.                 store the current j to mark the new end of line
11.
12.         cache[i] ← temp
13.     return cache[1] and follow the stored j values for line endings
```

Algorithm Explanation The algorithm follows the dynamic programming paradigm and the cache stores the error value. The cache is filled in the bottom up manner. The outer loop goes over each word and the inner loop determines the optimal line break given that the i^{th} word is the first word. As the inner loop goes over words from $i + 1$ to n, it tries to minimize the slack. We also keep track of the word at which a line ends so as to pretty print the text. Thus, we build our solution by solving smaller subproblems first.

Lines 8-12 in the algorithm above neatly describe what the recurrence looks like.

Algorithm Analysis The algorithm runs in O(n^2) time since there are nested loops both of which go up to n.

Solution 4. The following is the solution to problem 9 on page 320 of the textbook.

An efficient algorithm that caches the overlapping subproblems is:

```
1.  totalProcessedData(x[1...n], s[1...n])
2.      cache[1...n + 1][1...n + 1]
3.      fill the (n + 1)^th row and column with 0s          //Base Case
4.      for i ← n to 1
5.          for j ← n to 1
6.              cache[i][j] = max(cache[i + 1][1], cache[i + 1][j + 1] + min(x_i, s_j))
7.      return cache[1][1]
```

Algorithm Explanation At each step we have two choices - process the data at the capacity allowed or don't process any data and start from full capacity from the next day. The recurrence looks like the following:

$$\text{OPT}(i, j) = \max(\text{OPT}(i+1,1),\ \text{OPT}(i+1, j+1) + \min(x_i, s_j))$$

where i and j are pointers into array x and s respectively. We cache the recurrence in a 2D array and build it bottom up since the i^{th} value depends on the $(i+1)^{th}$ value.

Algorithm Analysis The algorithm is responsible for filling up a table of size $n \times n$. No cell is visited more than once. Thus, the runtime is $\text{O}(n^2)$.

Solution 5. The algorithm to find the maximal price of the wood is:

```
1. cutRod(Prices[1...n], n)
2.     cache[1...n + 1]
3.     cache[1] ← 0          //Base Case
4.     for i ← 1 to n
5.         current ← −∞
6.         for j ← 1 to i
7.             current ← max(current, prices[j] + caches[i − j + 1])
8.         cache[i + 1] ← current
9.     return cache[n + 1]
```

Algorithm Explanation The recurrence for the algorithm is maximizing the price by cutting it into all sizes smaller than n and then recursing over the remaining piece. This, however, introduces a lot of overlapping subproblems which can be cached. The algorithm presented above does that. It calculates the maximum price for each length less than n and then just reads off of the cache instead of calculating them each time.
Note that insertions into cache have been offset by 1 because the first element is the base case and is for a rod of length 0.

Algorithm Analysis The algorithm runs in $\text{O}(n^2)$ time. The recursive algorithm on the other hand is $\text{O}(2^n)$ since at each step we have a choice of whether to cut there or not and we try all possibilities. Thus, our dynamic programming algorithm is way more efficient.

Solution 6. Let i represent the front of the input row of coins and j represent the back of the input of row of coins.

The recurrence is of the form:

$$\text{OPT}(i, j) = \max(v_i + \min(\text{OPT}(i+1, j-1),\ \text{OPT}(i+2, j),\ v_j + \text{OPT}(i+1, j-1),\ \text{OPT}(i, j-2))$$

At each step, the first player has two choices - pick the first coin or the last coin. We add that to the minimum of the recursive results because the other player tries to maximize his amount of money too.

Therefore, we try both choices and pick the maximum of the two.

There are a lot of overlapping subproblems and thus an efficient dynamic programming algorithm is:

```
1.  maximumMoney(v[1...n])
2.      cache[1...n][1...n]
3.
4.      for i ← n to 1
5.          for j ← 1 to n
6.              if i > j
7.                  cache[i][j] = 0
8.              else if i = j
9.                  cache[i][j] = v_i
10.             else if i = j − 1
11.                 cache[i][j] = max(v_i, v_j)
12.             else
13.                 temp1 = min(cache[i − 2][j], cache[i − 1][j − 1])
14.                 temp2 = min(cache[i − 1][j − 1], cache[i][j − 2])
15.                 cache[i][j] = max(v_i + temp1, v_j + temp2)
16.
17.     return cache[1][n]
```

Algorithm Explanation The algorithm just caches the recursions (the recurrence is stated above) in a 2D array. Since the recurrence for i, j depends on $i + 1$ or $i + 2$ and $j - 1$ or $j - 2$, we move from n to 1 for i and from 1 to n for j.

Algorithm Analysis The algorithm fills an $n \times n$ array. Thus, the run time is just $O(n^2)$.