Design and Analysis of Algorithms: Homework #3

Due in class on March 1, 2018

Professor Kasturi Varadarajan

Alic Szecsei

Problem 1

You and your eight-year-old nephew Elmo decide to play a simple card game. At the beginning of the game, the cards are dealt face up in a long row. Each card is worth a different number of points. After all the cards are dealt, you and Elmo take turns removing either the leftmost or rightmost card from the row, until all the cards are gone. At each turn, you can decide which of the two cards to take. The winner of the game is the player that has collected the most points when the game ends.

Having never taken an algorithms class, Elmo follows the obvious greedy strategy—when it's his turn, Elmo always takes the card with the higher point value. Your task is to find a strategy that will beat Elmo whenever possible. (It might seem mean to beat up on a little kid like this, but Elmo absolutely hates it when grown-ups let him win.

(a) Prove that you should not also use the greedy strategy. That is, show that there is a game that you can win, but only if you do *not* follow the same greedy strategy as Elmo.

(b) Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against Elmo.

Solution

Part A

Suppose there are a total of 8 cards, with point values ranging from 1 to 8. When dealt, they are arranged like so:

3	8	7	6	5	4	2	1

Table 1: an example game

If you were to play using the greedy strategy, you would first take card 3, while Elmo would take 8, and so on. At the end of the game, you would have $3 + 7 + 5 + 2 = 17$ points, while Elmo would have $8 + 6 + 4 + 1 = 19$ points.

However, if you first took card 1, Elmo would take card 3, leaving you to take card 8, and so on, now playing greedily. At the end of the game, you would have $1+8+6+4 = 19$ points, while Elmo would have $3+7+5+2 = 17$ points.

Thus, you can only win this game by making your first move take the smaller of the two available cards, as opposed to Elmo's greedy strategy.

Problem 1 continued on next page…

Part B

Algorithm 1 Elmo's Card Game

```
function Elmo(cards[1..n])
    memorized ← empty hash map
    memorized[the empty list] ← 0
    for i ← 1, n do
        for j ← 1, n − i do
            subproblem ← cards[j..j + i]
            if i = 1 then
                memorized[subproblem] ← subproblem[1]
            else
                L ← subproblem[2..i]                                      ▷ Try taking the first card
                if the first element of L is greater than the last element of L then
                    remove the first element of L
                else
                    remove the last element of L
                end if
                l ← subproblem[1] + memorized[L]
                R ← subproblem[1..i − 1]                                  ▷ Try taking the last card
                if the first element of R is greater than the last element of R then
                    remove the first element of R
                else
                    remove the last element of R
                end if
                r ← subproblem[i] + memorized[R]
                memorized[subproblem] ← the maximum of l and r
            end if
        end for
    end for
    return memorized[cards]
end function
```

This algorithm works from the bottom up to determine all sub-problems. Essentially, at any stage of the game, we may have some i cards, where $i < n$, and all i cards were originally neighbors (since players may only remove cards to the furthest left and furthest right). Therefore, all possible sub-problems are simply combinations of i neighboring cards. For each sub-problem, assuming it is not trivial (no cards remaining or one card remaining), we can perform two calculations: one assuming the player removes a card from the left, and one assuming she removes a card from the right. Whichever of these two routes produces the largest score is stored as the solution to the sub-problem. When all possible sub-problems have been calculated, the algorithm returns the solution to the n-sized sub-problem, which is the original problem.

The run-time here is straightforward to analyze; there are n "layers" of sub-problems, since we are analyzing games of 1 remaining card, 2 remaining cards, and so on until we have reached the original problem of n remaining cards. For each of these "layers", we find all possible collections of neighboring cards; so, in total, we have $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ sub-problems, which leaves us with a runtime of $O(n^2)$.

Problem 2

A palindrome is any string that is exactly the same as its reversal, like `I`, or `DEED`, or `RACECAR`, or `AMANAPLANACATACANALPANAMA`.

(a) Describe and analyze an algorithm to find the length of the *longest subsequence* of a given string that is also a palindrome. For example, the longest palindrome subsequence of `MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM` is `MHYMRORYHM`, so given that string as input, your algorithm should output the number 11.

(c) Any string can be decomposed into a sequence of palindromes. For example, the string `BUBBASEESABANANA` ("Bubba sees a banana.") can be broken into palindromes in the following ways (and many others):

`BUB • BASEESAB • ANANA`
`B • U • BB • A • SEES • ABA • NAN • A`
`B • U • BB • A • SEES • A • B • ANANA`
`B • U • B • B • A • S • E • E • S • A • B • A • N • ANA`

Describe and analyze an efficient algorithm to find the smallest number of palindromes that make up a given input string. For example, given the input string `BUBBASEESABANANA`, your algorithm would return the integer 3.

Solution

Part A

Algorithm 2 Longest Palindrome Subsequence

```
function LPS(text[1..n])
    memorized ← empty hash map
    memorized[the empty list] ← 0
    for i ← 1, n do
        for j ← 1, n − i do
            subproblem ← text[j..j + i]
            if i = 1 then
                memorized[subproblem] ← 1        ▷ A single letter is always a palindrome of length one
            else
                r ← the maximum of memorized[subproblem[2..i]] and memorized[subproblem[1..i − 1]]
                if subproblem[1] = subproblem[i] then
                    r ← the maximum of r and 2 + memorized[subproblem[2..i − 1]]
                end if
                memorized[subproblem] ← r
            end if
        end for
    end for
    return memorized[text]
end function
```

This solution is similar to the previous algorithm. Each subproblem is a set of neighboring characters in the string, with i "layers" of substring lengths and $1 + n - i$ substrings of length i. Using the previous summation, we see that the runtime is $O(n^2)$.

Problem 2 continued on next page…

Part C

Algorithm 3 Smallest Number of Palindromes

```
function SmallestNumPalindromes(text[1..n])
    memorized ← empty hash map
    palindromes ← empty hash map
    memorized[the empty list] ← 0
    palindromes[the empty list] ← True
    for i ← 1, n do                                   ▷ First, calculate all palindromes
        for j ← 1, n − i do
            subproblem ← text[j..j + i]
            if i = 1 then
                palindromes[subproblem] ← True
            else if subproblem[0] ≠ subproblem[j + i] then
                palindromes[subproblem] ← False
            else
                palindromes[subproblem] ← palindromes[subproblem[2..j + i − 1]]
            end if
        end for
    end for
    for i ← 1, n do              ▷ Now we calculate the actual number of sub-palindromes
        for j ← 1, n − i do
            subproblem ← text[j..j + i]
            if i = 1 then
                memorized[subproblem] ← 1
            else
                min ← i         ▷ The largest value here assumes there are no sub-palindromes
                for k ← 1, i do
                    if palindromes[subproblem[1..k]] then
                        min ← the minimum of min and 1 + memorized[subproblem[k..i]]
                    end if
                end for
                memorized[subproblem] ← min
            end if
        end for
    end for
    return memorized[text]
end function
```

This solution has a few notable differences from the previous algorithm. First, we make sure to store whether or not any sub-string is a palindrome, which can be done in $O(n^2)$ time, using a similar strategy as the last few algorithms.

However, the second half of the solution is where things get more complex. For each substring, we wish to determine the smallest number of palindromes that can make up that sub-string; to do this, we attempt to find palindromes at the beginning of the sub-string, and then find how many palindromes it takes to fill in the *rest* of the sub-string. As we have $O(n^2)$ sub-strings and we use an additional loop to determine beginning palindromes, the final running time for this second half of the solution is $O(n^3)$. Therefore, our final running time is $O(n^3)$.

Problem 3

Suppose we are given a set L of n line segments in the plane, where each segment has one endpoint on the line $y = 0$ and one endpoint on the line $y = 1$, and all $2n$ endpoints are distinct. Describe and analyze an algorithm to compute the largest subset of L in which no pair of segments intersects.

Solution

Algorithm 4 Largest subset of non-intersecting line segments

```
function LargestSubsetNonIntersecting(set[1..n])
    Sort(set[1..n])                          ▷ sort by the x-value of the first point
    set[0] ← (−∞, −∞)                        ▷ sentinel value
    for i ← 0, n do                          ▷ Base cases
        memorized[i, n + 1] ← an empty list
    end for
    for j ← n, 1 do                          ▷ Now we calculate the longest increasing subsequence
        for i ← 0, j − 1 do
            if set[i].b > set[j].b then
                memorized[i, j] ← memorized[i, j + 1]
            else
                memorized[i, j] ← whichever of memorized[i, j + 1] and [set[j]] + memorized[j, j + 1] has
the longer length
            end if
        end for
    end for
    return memorized[0, 1]
end function
```

Here we attempt to find the largest subset of non-intersecting line segments. To do this, we first sort the list of non-intersecting line segments by the x-value of the first point. Now, we need only examine the x-value of the second point to determine if two line segments intersect; we are guaranteed that, for some line segments X_1 and X_2, if X_2 is sorted after X_1, then the only way X_1 and X_2 can intersect is if the second point of X_1 is greater than the second point of X_2.

Now, the problem transforms into a modified longest increasing subsequence; we simply must test that the second point of each line segment is increasing, and we are guaranteed that no line segments will intersect.

As sorting can be done in $n \log n$ time, and the longest increasing subsequence has $O(n^2)$ runtime (as seen in the notes), our algorithm has a runtime of $O(n^2)$.

Problem 4

Oh, no! You have been appointed as the organizer of Giggle, Inc.'s annual mandatory holiday party! The employees at Giggle are organized into a strict hierarchy, that is, a tree with the company president at the root. The all-knowing oracles in Human Resources have assigned a real number to each employee measuring how "fun" the employee is. In order to keep things social, there is one restriction on the guest list: an employee cannot attend the party if their immediate supervisor is also present. On the other hand, the president of the company *must* attend the party, even though she has a negative fun rating; it's her company, after all. Give an algorithm that makes a guest list for the party that maximizes the sum of the "fun" ratings of the guests.

Solution

First, we perform a breadth-first traversal of the tree. Then, we reverse the resulting list of nodes. This gives us a list which starts with leaf nodes, and then reaches the root at the very end of the list.

Next, we can go through the list and define sub-problems. Each sub-problem has a root node, $current$, and two choices: either $current$ should be included in the party, or not. If $current$ is not included in the party, we can examine its children (if there are any) to see the resulting maximum fun value; otherwise, we must look at $current$'s grandchildren to determine the maximum possible fun value. Either way, we can determine which strategy would lead to the most fun list of attendees.

At the conclusion of our algorithm, since the boss *must* attend the party, and our algorithm so far supposed that the $current$ root might not attend the party, we include her in a separate $returnList$ and then append the most fun guest lists we have for each of her supervisee's supervisees (as her direct supervisees may never attend the party, according to Giggle, Inc.'s stringent attendance policies).

The breadth-first traversal portion of the algorithm is relatively minor, having a linear-time complexity. The bulk of our algorithm's complexity lies in the subsequent calculations.

Since there are n sub-problems, and the algorithm requires *at most* n calculations per sub-problem (in the case that all n nodes are either children or grandchildren of the node), we can safely say that the algorithm has a worst-case runtime of $O(n^2)$.

Algorithm 5 Most fun party attendance

```
function MostFunParty(root)
    queue ← [root]                                  ▷ Do a breadth-first traversal
    i ← 1
    while i ≤ the length of queue do
        current ← queue[i]
        for all child in current.children do
            push child onto queue
        end for
        i ← i + 1
    end while
    reverse queue
    for i ← 1, the length of queue do
        current ← queue[i]
        maxA ← current.fun
        listA ← [current]                           ▷ route A: if the root is included in the party
        maxB ← 0
        listB ← []                                  ▷ route B: if the root is not included in the party
        for all child in current.children do
            maxB ← maxB + fun[child]
            listB ← listB + list[child]             ▷ the children can only be included if the root is not
            for all grandchild in child.children do
                maxA ← maxA + fun[grandchild]
                listA ← listA + list[grandchild]
            end for
        end for
        if maxA > maxB then                         ▷ use whichever route, A or B, provides the most fun
            fun[current] ← maxA
            list[current] ← listA
        else
            fun[current] ← maxB
            list[current] ← listB
        end if
    end for
    returnList ← [root]                             ▷ returnList contains the final guest list
    for all child in root.children do
        for all grandchild in child.children do
            returnList ← returnList + list[grandchild]
        end for
    end for
    return returnList
end function
```