

# **Design and Analysis of Algorithms: Homework #3**

Due in class on March 1, 2018

*Professor Kasturi Varadarajan*

**Alic Szecei**

## 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.

## Part B

---

**Algorithm 1** Elmo's Card Game
 

---

```

1: function ELMO(cards[1..n])
2:   memorized  $\leftarrow$  empty hash map
3:   memorized[the empty list]  $\leftarrow$  0
4:   for i  $\leftarrow$  1, n do
5:     for j  $\leftarrow$  1, n - i do
6:       subproblem  $\leftarrow$  cards[j..j + i]
7:       if i = 1 then
8:         memorized[subproblem]  $\leftarrow$  subproblem[1]
9:       else
10:        L  $\leftarrow$  subproblem[2..i] ▷ Try taking the first card
11:        if the first element of L is greater than the last element of L then
12:          remove the first element of L
13:        else
14:          remove the last element of L
15:        end if
16:        l  $\leftarrow$  subproblem[1] + memorized[L]
17:        R  $\leftarrow$  subproblem[1..i - 1] ▷ Try taking the last card
18:        if the first element of R is greater than the last element of R then
19:          remove the first element of R
20:        else
21:          remove the last element of R
22:        end if
23:        r  $\leftarrow$  subproblem[i] + memorized[R]
24:        memorized[subproblem]  $\leftarrow$  the maximum of l and r
25:      end if
26:    end for
27:  end for
28:  return memorized[cards]
29: 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  $\sum_{i=1}^n i$  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 AMANAPLANACAT-ACANALPANAMA.

- (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

---

```

1: function LPS(text[1..n])
2:   memorized ← empty hash map
3:   memorized[the empty list] ← 0
4:   for i ← 1, n do
5:     for j ← 1, n − i do
6:       subproblem ← text[j..j + i]
7:       if i = 1 then
8:         memorized[subproblem] ← 1           ▷ A single letter is always a palindrome of length one
9:       else
10:        r ← the maximum of memorized[subproblem[2..i]] and memorized[subproblem[1..i − 1]]
11:        if subproblem[1] = subproblem[i] then
12:          r ← the maximum of r and 2 + memorized[subproblem[2..i − 1]]
13:        end if
14:        memorized[subproblem] ← r
15:      end if
16:    end for
17:  end for
18:  return memorized[text]
19: 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  $O(n^2)$ .

**Part C**

**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**

## 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