Homework 2

Prateep Mukherjee

October 26, 2013

1. a) The maximum contiguous subarray problem can be solved using a dynamic programming algorithm as follows:

Input: A : array of numbers, which may be positive, negative, or zero
Output: maximum sum of elements in a contiguous subarray $A[i \dots j]$
//`Initialize global and local maximum sums`
$res \leftarrow 0$;
$current\_max \leftarrow 0$;
for $i = 1 \dots n$ **do**
    if $current\_max + A_i > current\_max$ **then**
        //`Add` A_i `to` $current\_max$ `if it improves later`
        $current\_max \leftarrow current\_max + A_i$;
    else
        //`Begin a new subarray at` i
        $current\_max \leftarrow A_i$;
    end
    //`Update global maximum if` $current\_max$ `improves former`
    if $current\_max > res$ **then**
        $res \leftarrow current\_max$;
    end
end
return res

Algorithm 1: Maximum contiguous subarray sum

Complexity of Algorithm 1 is $\Theta(n)$, as we are iterating over the array just once.

Proof of Correctness: Let, S be the current maximum in subarray ending at index j ($j < n$). Now, we add A_{j+1} to S, if it increases S. This means, we can get a larger sum, ending at $j + 1$, by including A_{j+1} to our current maximum. The above condition is possible only when $A_{j+1} > 0$. Hence, in the case when $A_{j+1} < 0$, we begin a new subarray starting at index $j + 1$.

A special case in this problem, is when all the elements of A are negative. In such a scenario, we argue that our algorithm produces the correct

result, which is the maximum of all the elements in A. In this case, each A_i will decrease the value of $current\_max$. So, for each element, the algorithm will go to else condition and update $current\_max$. The global maximum res, then will be equal to the maximum element in the array.
(b) The maximum contiguous array product problem is solved using a similar approach as in (a). One observation here, however, is that if an element of A is negative, a current minimum negative product can be multiplied with it to get the largest positive product. Hence, we need two running maximum positive and minimum negative products. The algorithm can be written as follows:

Input: A : array of integers, which may be both positive or negative or zero
Output: Maximum product in a contiguous subarray of A
$res \leftarrow 1$;
$maxpos \leftarrow 1$
//`Maximum running positive product` ≥ 1
$minneg \leftarrow 1$
//`Minimum running negative product` ≤ 1
for $i = 1 \ldots n$ **do**
  if $A_i > 0$ **then**
    //`Update` $maxpos$ `over here.`$minneg$ `will be updated if` $minneg < 0$`, in which case it will get smaller`
    $maxpos \leftarrow maxpos \times A_i$;
    $minneg \leftarrow \min(minneg \times A_i, 1)$;
  $A_i < 0$
  //`If` $minneg < 0$`,` $maxpos$ `will be updated, else` $maxpos = 1.minneg$ `will be updated to last` $maxpos \times A_i$`.`
  $prev \leftarrow maxpos$;
  $maxpos \leftarrow \max(minneg \times A_i, 1)$;
  $minneg \leftarrow prev \times A_i$;
  else
    //`Begin new subarray at` i
    $maxpos \leftarrow 1$;
    $minneg \leftarrow 1$;
  end
  //`Update global maximum if` $maxpos$ `improves former`
  if $maxpos > res$ **then**
    $res \leftarrow maxpos$;
  end
end
return res

Algorithm 2: Maximum contiguous array product

Complexity of algorithm 2 is $\Theta(n)$ as we are iterating over the array only once.

Proof of Correctness: As seen in Algorithm 2, we have two variables, storing maximum positive and minimum negative products. Now, if $A_i > 0$, we simply update the maximum positive product. This is done in the first *if* in Algorithm 2 However, if $A_i < 0$, we can get an even bigger product than *maxpos* by multiplying the minimum negative product and A_i. The minimum negative product, using similar analogy, will be the product of the previous *maxpos* and A_i. This is done in *else if* condition. Finally, we cannot include $A_i = 0$ elements in our result. Hence, we begin a new subarray if we see a $A_i = 0$. This is the *else* check.

2. The given problem can be solved using a dynamic programming algorithm. However, first we need to pre-process the input string to generate the number of palindromes at each element of the string. Let, $L[i][j]$ denote the number of palindromes of length j ending at S_i. This can be generated in $\Theta(n^2)$ time complexity by checking the number of palindromes of length 1, 2 and so on, with the end character being S_i.
The subproblem of the DP solution is as follows. $L[i][j]$ is updated only when there is at least one palindrome of length $j-2$ ending at S_{i-1}. Thus, pseudo-code for computing $L[i][j]$ for the entire string S can be written as:

```
//Each character in S is a palindrome in itself.
L[i][0] = 1;
for i ← 0 ... |S| do
    for j ← 2 ... |S| do
        //Check if there is at least one palindrome of length
          j − 2 ending at S_{i−1}
        if j == 2 then
            //Two characters, hence a palindrome
            L[i][j] = 1;
        else
            //Check maximum length palindrome ending at S_{i−1}
            len ← |L[i − 1]|;
            //If there exists palindrome of length j − 2 ending
              at S_{i−1}, we add to our list a palindrome of
              length j ending at S_i
            if L[i − 1][len] == j − 2 then
                L[i][j] = 1;
            end
        end
    end
end
```

Algorithm 3: PREPROCESS: Compute L

Having generated L, we can now use dynamic programming to generate an

optimal result. Let, $A[i]$ denote the minimum number of palindromes in $S[0 \dots i-1]$. Then,

$$A[i+1] = \min\left(A[i - L[i][j]] + 1, \forall j \in [0 \dots |L[i]|]\right) \quad (1)$$

Solving eq. 1 is also $\Theta(n^2)$ time complexity. Overall time complexity of this algorithm is thus $\Theta(n^2)$.

Proof of correctness: L_{ij} stores the number of palindromes of length j with the last character in the palindrome being S_i. Now, A_i will have the minimum number of palindromes in the substring $S[0 \cdots i-1]$. A_i is computed taking the minimum of all palindromes ending at S_{i-1}. Therefore, A_i must have the minimum number of palindromes in the substring $S[0 \cdots i-1]$. Finally, $A_{|S|}$ gives us the required result.

3. The first obvious algorithm that one might use to solve this problem is greedy-based - choose the exit nearest to the current location with the maximum capacity to drop-off students, reach that exit traveling forward or backward, drop students off at that exit and repeat the algorithm until there is at least one student left in the bus. However, this algorithm does not give us the optimal solution.

In Fig.1, we see an example of the greedy algorithm. Let the initial number of students in the bus be 20. Using the greedy algorithm, we will choose the exit on the right, drop 5 students over there and come back to the original location. Thus, the total amount of soda that will be consumed is $20 \times 1 + 15 \times 1 = 35$. However, if we chose the right direction, the maximum amount of soda consumed will be $20 \times 1.5 = 30$. This simple example proves that a greedy algorithm will not give the optimal strategy. The other option is to use a dynamic programming method. The first observation we make is that visited exits on the highway should be contiguous. This means that we can go to an exit not yet visited, only from either its immediate left visited or right visited exit.

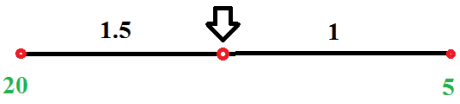

Figure 1: The red circles show the exits, with the one pointed by an arrow being the current location. The numbers in black are the time required to travel between the exits, and those in green represent the maximum number of students that can be dropped at each exit.

So, given a state of visited exits, we can only make two possible transitions - go to the immediate left unvisited exit, or the immediate right unvisited one. The optimal solution solution, of course, is the minimum of them. It is to be noted here that at **each transition we are decreasing the number of unvisited exits by one**. Hence, our algorithm is guarranted to converge.

Now, let us define the state of our DP solution. Let, $< lhs, rhs, dir >$ denote the left and right-hand side exits which are yet not visited, given that every exit $\{e | lhs < e < rhs\}$ has been visited before, and dir is a boolean variable denoting which direction the bus is travelling. The DP recurrence relation can now be written as follows:

$$< lhs, rhs, dir >= \min \left\{ \begin{array}{l} < lhs + 1, rhs, dir = true >, \\ < lhs, rhs - 1, dir = false > \end{array} \right\} \quad (2)$$

The base case of our algorithm is when $lhs + 1 == rhs$, which we can compute using the time it takes to travel from exit lhs to rhs. Solving eq.2 naively will lead us to an exponential time algorithm, as to solve each problem we are computing twice as many subproblems. This is solved by using *memoization* technique, or caching the previously computed interval results. Eq. 2 can be implemented using a 2D array of all exits, where each cell (i, j) denote the state $< i, j, dir >$. The array is computed using $\Theta(n^2)$, where n is the total number of exits.

Proof of Correctness: To solve each subproblem, we are taking the minimum time of two transitions, traversing left and right. This has to be the optimal way of traversal, as we have to travel contiguous exits.

4. a) The naive approach to compute the minimum number of bills to make k Dream Dollars is to try all possible combinations of the given denominations. The algorithm proceeds as follows:

Input: coin denominations $C = \{c_1, c_2, c_3, \ldots c_n\}$, K
Output: Minimum number of coins, from given denominations, to generate K Dream Dollars

```
best ← −1;
numCoinsSoFar ← 0;
solve(C, K)
for c ∈ C do
    change ← K − c;
    if change > 0 || numCoinsSoFar + 1 ≥ best then
        if change = 0 then
            best ← numCoinsSoFar + 1;
        else
            numCoinsSoFar = numCoinsSoFar + 1;
            solve(C, change);
            numCoinsSoFar = numCoinsSoFar − 1;
        end
    end
end
return best
```

Algorithm 4: Solve_Naive

Algorithm 4 includes each denomination once and recomputes the entire procedure using the change(> 0). The complexity of the algorithm is $\Theta\left(2^K\right)$.

Proof of Correctness: The naive approach covers the entire search space of possible solutions, and chooses the minimum one. Hence, it always generates the optimal solution.

b) The dynamic programming algorithm proceeds as follows. Let, S be an array, where every element S_j represents the minimum number of coins to generate j Dream Dollars. S is initialized to 0.

The algorithm can be stated simply as follows. For each coin c, $c \leq k$, look at the minimum number of coins found for the $k - c$ Dream Dollars(which we have found previously). Let, this number be p. If $p + 1$ is less than the minimum number of coins already found for the current sum k, then we update S_k.

The Pseudo-code for this algorithm can be written as:

Input: Dream Dollar K
Output: Minimum number of coins, from given denominations, to generate K Dream Dollars
$S[0 \dots MAX] \leftarrow \infty$
//`MAX is the maximum possible Dream Dollar amount`
for $k = 1 \dots K$ **do**
  for $c \in C$ **do**
    if $c \leq k \quad \& \quad S[k-c] + 1 < S[k]$ **then**
      $S[k] \leftarrow S[k-c] + 1$;
    end
  end
end
return S[K]

Algorithm 5: Solve_DP

Algorithm 5 computes current value in S based on previously computed numbers. Complexity of the above algorithm is $\Theta(K|C|)$.

Proof Of Correctness At each step of algorithm 5 we update our current answer for denomination k if and only if we find a better answer using a coin of lesser denomination. Therefore, at each step we are computing the optimal result for denomination k.

(c) (i) Table 1 shows the runtimes of naive and DP algorithms for a set of dream dollars. The denominations used are $\{1, 4, 7, 13, 28, 52, 91, 365\}$. The results for naive and DP methods matched in each of the cases.

(ii) In this section, we report the difference in results from greedy and dynamic programming algorithms.

It is obvious from Table 2 that the greedy algorithm fails to report the optimal result in many cases. Let, *proc* denote the procedure name for coin exchange problem. For example, for $K = 416$, the greedy algorithm works as follows.

$$proc(416) \equiv \{1(\$\ 365), 1(\$\ 28), 1(\$\ 13), 1(\$\ 7), 3(\$\ 1)\} \equiv 7$$

The DP method, however, finds an optimal solution as follows.

Table 1: Computation time(in seconds) for different denominations.

Denomination	Naive	DP
10	0.18	0.10
15	0.04	0.12
20	0.12	0.12
25	0.21	0.13
30	0.26	0.14
35	0.33	0.15
40	0.47	0.19
45	0.70	0.20
50	1.00	0.19
55	0.74	0.21

Table 2: Minimum number of coins as reported by Greedy & DP algorithms

K	Greedy	DP
416	7	5
455	7	5
507	8	6
546	8	6
598	9	7

$$proc(416) \equiv \{4(\$\ 91), 1(\$\ 52)\} \equiv 5$$