Dalhousie University Faculty of Computer Science
Design and Analysis of Algorithms
Solutions to practice problems CSCI 3110 Practice Dec 2012

(1) **(a) 6.1:** A contiguous subsequence of a list S is a subsequence made up of consecutive elements of S. For instance, if S is $5, 15, -30, 10, -5, 40, 10$, then $15, -30, 10$ is a contiguous subsequence but $5, 15, 40$ is not. Give a linear-time algorithm for the following task:
Input: A list of numbers, $a_1, a_2, \ldots, a_n$.
Output: The contiguous subsequence of maximum sum (a subsequence of length zero has sum zero).
For the preceding example, the answer would be $10, -5, 40, 10$, with a sum of 55 .
(Hint: For each $j \in \{1, 2, \ldots, n\}$, consider contiguous subsequences ending exactly at position j.)

Let $s[j]$ be the sum of the maximum contiguous sequence that ends at position j. Then for $s[j+1]$ we can either start a new contiguous sequence with score a_{j+1} or continue the contiguous sequence with score $s[j] + a_{j+1}$. The recurrence is:

$$s[j] = \begin{cases} a_j, & \text{if } j = 0 \\ \max\{s[j-1] + a_j, a_j\}, & \text{if } j > 0 \end{cases}$$

We can implement this recurrence in linear time by computing $s[0], s[1], \ldots, s[n]$. To recover the sequence, we first scan through $s[]$ to find the location $s[j]$ which is maximum. We scan backwards from j to find the negative element $s[i]$ at the beginning of the maximum contiguous sequence, or have $i = -1$ (we could also directly store these values during the search). Then the maximum contiguous sequence is $a_{i+1}, a_{i+2}, \ldots A_j$ with score $s[j]$. The backtracking takes linear time as well.

(b) 6.2 : You are going on a long trip. You start on the road at mile post 0. Along the way there are n hotels, at mile posts $a_1 < a_2 < \ldots < a_n$, where each a_i is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance a_n), which is your destination.
You'd ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel x miles during a day, the penalty for that day is $(200 - x)^2$. You want to plan your trip so as to minimize the total penalty–that is, the sum, over all travel days, of the daily penalties.
Give an efficient algorithm that determines the optimal sequence of hotels at which to stop.

Let $p[j]$ be the minimum penalty required to travel to hotel j. To get the minimum penalty required to travel to hotel $j+1$, we could drive from any of the previous hotels i to j and have a penalty of $p[i] + (200 - (a_j - a_i))^2$. The recurrence is:

$$p[j] = \begin{cases} a_j, & \text{if } j = 1 \\ \min_{1 \le i < j}\{p[i] + (200 - (a_j - a_i))^2\}, & \text{if } j > 0 \end{cases}$$

We can implement this recurrence in $O(n^2)$ time by computing $p[0], p[1], \ldots, p[n]$. To determine the hotels that should be stopped at, we store the hotel i that we should travel from to reach each j that got the penalty $p[j]$. Starting with n, we can follow these hotel links backwards to get the sequence of hotels. The backtracking takes linear time, so this takes $O(n^2)$ time in total.

(2) **(a) 6.8 :** Given two strings $x = x_1 x_2 \ldots x_n$ and $y = y_1 y_2 \ldots y_m$, we wish to find the length of their longest common substring, that is, the largest k for which there are indices i and j with $x_i x_{i+1} \ldots x_{i+k-1} = y_j y_{j+1} \ldots y_{j+k-1}$. Show how to do this in time $O(mn)$.

Let $s[i][j]$ be the length of the longest common substring that ends with matching x_i and y_j. This is either 0, or, if $x_i = x_j$, then we can add x_i to the longest common substring that ends with matching x_{i-1} and y_{j-1}. The recurrence is:

$$s[i][j] = \begin{cases} 0, & \text{if } i = 0,\ j = 0, \text{ or } x_i \neq x_j \\ 1 + s[i-1][j-1], & \text{if } x_i = x_j \end{cases}$$

This can be implemented to run in $O(mn)$ time by filling the table in either row by row or column by column. To recover the longest common substring, we scan the table once to find the maximum value $s[i][j]$. The longest common substring is then $x_{i-s[i][j]-1} \ldots x_i$ (equivalently, $y_{j-s[i][j]-1} \ldots y_j$) or $\emptyset$ if the maximum is 0. Backtracking also takes $O(mn)$ time.

(b) 6.11 : Given two strings $x = x_1 x_2 \ldots x_n$ and $y = y_1 y_2 \ldots y_m$, we wish to find the length of their longest common subsequence, that is, the largest k for which there are indices $i_1 < i_2 <$

$\ldots < i_k$ and $j_1 < j_2 < \ldots < j_k$ with $x_{i_1} x_{i_2} \ldots x_{i_k} = y_{j_1} y_{j_2} \ldots y_{j_k}$. Show how to do this in time $O(mn)$.

Let $s[i][j]$ be the length of the longest common subsequence of $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$. We have three options. The longest common subsequence of $s[i][j]$ could be the longest common subsequence of $s[i-1][j]$, or of $s[i][j-1]$, or, if $x_i = x_j$, then we could add x_i to the longest common subsequence of $s[i-1][j-1]$. The recurrence is:

$$s[i][j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ \max\{s[i-1][j], s[i][j-1], s[i][j]+1\}, & \text{if } x_i = x_j \\ \max\{s[i-1][j], s[i][j-1], & \text{if } x_i \neq x_j \end{cases}$$

This can be implemented to run in $O(mn)$ time by filling the table in either row by row or column by column. Similar to question 6.2, we can store the locations of the last matched values a and b for each $s[i][j]$. The length of the longest common subsequence is $s[n][m]$ and we backtrack using the stored values to recover the actual sequence. Backtracking takes linear time, so this algorithm runs in $O(mn)$ time in total.

(3) **Moving on a checkerboard :**

Suppose that you are given an $n \times n$ checkerboard and a checker. You must move the checker from the bottom edge of the board to the top edge of the board according to the following rule. At each step you may move the checker to one of three squares:
(a) the square immediately above,
(b) the square one up and one to the left (but only if checker is not already in leftmost column),
(c) the square one up and one to the right (but only if checker not already in the rightmost column),
Each time you move from square x to square y, you receive $p(x, y)$ dollars. You are given $p(x, y)$ dollars for all pairs (x, y) for which a move from x to y is legal. Do not assume that $p(x, y)$ is positive.

Let $g[i, j]$ be the gain (in dollars) in the most profitable way to square (i, j), from row 1.For this problem - the sub-problems to consider are: going from some square in row 1 to a particular square in row i. To obtain most profitable way to sq. (i, j) - we need to most profitable way to get to get to sqs. $(i-1, j-1)$; $(i-1, j)$; $(i-1, j+1)$ (This shows the sub-problem optimality property) We want to calculate $g[1, n]$ (Please see dev. of recurrence notes posted on web-page)

$$\begin{aligned} g[i,j] &= max(g[i-1,j-1] + p([i-1,j-1], \nearrow) \\ &\quad , \quad g[i-1,j] + p([i-1,j], \uparrow) \\ &\quad , \quad g[i,j+1] + p([i-1,j+1], \nwarrow)) \\ g[i,j] &= 0, \ when \ i = 1, j = 1, 2 \ldots n \end{aligned}$$

Notice, to get the most profitable way to get to sqs. $(i+1, j-1)$ OR sq. $(i+1, j)$ OR sq. $(i+1, j)$ we will need to check sq. (i, j). Hence we have overlapping sub-problems. We can make a table, and search from bottom to up, storing how we get to sq (i, j) in $w[i, j]$ Below: `R` is $\nearrow$ etc..

```
CHECKER_PROFIT(n, p)
 1. for j = 1 to n
 2.        g[1,j] = 0
 3. for i = 2 to n
 4.     for j = 1 to n
 5.         g[i,j] = -infinity
 6.         if j > 1
 7.             g[i,j] = g[i-1,j-1] + p(i-1,j-1,"R")
 8.             w[i,j] = j-1
 9.         if g[i-1,j] +  p(i-1,j,"UP") > g[i,j]
10.             g[i,j] = g[i-1,j] +  p(i-1,j,"UP")
11.             w[i,j] = j
12.         if j < n && g[i-1,j+1] +  p(i-1,j+1,"L") > g[i,j]
13.             g[i,j] = g[i-1,j+1] +  p(i-1,j+1,"L")
14.             w[i,j] = j+1
15.  return g and w
```

Once the table is filled all can be looked up. The maximum profit is at some $g[n, j]$, where $1 \leq j \leq n$. We can then backtrack through the board by finding, for a square (i, j), a square $(i-1, j-1)$, $(i-1, j)$, or $(i-1, j+1)$ such that the moving from that square to (i, j) results in the penalty $g[i, j]$. First loop through $g[n, j]$ to find the maximum profit. Then, the following recursive procedure will output the visited squares using CHECKER_SQUARES(n, p, n, j):

```
CHECKER_SQUARES(n, p, i, j)
1. print (i,j)
2. if i == 1
3.     return
4. if j > 1 && g[i,j] == g[i-1,j-1] + p(i-1,j-1,"R")
5.     CHECKER_SQUARES(n, p, i-1, j-1)
6. if g[i,j] == g[i-1,j] + p(i-1,j,"UP")
7.     CHECKER_SQUARES(n, p, i-1, j)
8. if j < n && g[i,j] == g[i-1,j+1] + p(i-1,j+1,"L")
9.     CHECKER_SQUARES(n, p, i-1, j+1)
```

The running time of the algorithm is $O(n^2)$ -clear from pseudocode.