CSCI 570 - Fall 2015 - HW 7 Solution

1. Reading Assignment: Kleinberg and Tardos, Chapter 6.1-6.4.

2. (*Chapter 5, Q 5*) Let $Y_{i,k}$ denote the substring $y_i, y_{i+1}, \cdots, y_k$. Denote the optimal segmentation for $Y_{1,k}$ as $X_1, X_2, \cdots, X_m$. A key observation to make in this problem is that $X_1, X_2, \cdots, X_{m-1}$ is an optimal segmentation for the prefix of $Y_{1,k}$ excluding X_m (because otherwise we could substitute the solution for the prefix by an optimal one and get a better solution).

   Given this observation, let $Opt(k)$ represent the quality of an optimal segmentation of the substring $Y_{1,k}$. We have the following recurrence:

   $$OPT(k) = \max_{1 \le i \le k} \{OPT(i-1) + Quality(Y_{i,k})\}. \tag{1}$$

   We can begin solving the above recurrence with the initial condition that $OPT(0) = 0$ and then go on to compute $OPT(k)$ for $k = 1, 2, \cdots, n$, i.e.,

```
$OPT(0) = 0$;
for $k = 1, \cdots, n$ do
  $OPT(k) = \max_{1 \le i \le k} \{OPT(i-1) + Quality(Y_{i,k})\}$;
  Record the index $i$ which achieves the maximum, denoted as $i^*(k)$;
end for
Return $OPT(n)$;
Track back through the array $OPT$ by checking from $i^*(n)$ to $i^*(1)$ to recover
the optimal segmentation.
```

   It takes $O(n)$ time to compute each $OPT(k)$;there are $O(n)$ iterations; it takes $O(n)$ time to trace back to recover the optimal segmentation. So the overall complexity is $O(n^2)$.

3. (*Chapter 6, Q 6*) Let $W = \{w_1, w_2, \cdots, w_n\}$ be the set of ordered words which we wish to print. In the optimal solution, if the last line contains p words, then the previous lines constitute an optimal solution for the sub problem with the set $\{w_1, \cdots, w_{n-p}\}$. Otherwise, by replacing with an optimal solution for the previous lines, we would get a better solution.

   For $i \le j$, let $S(i,j)$ represent the slack of a line containing the words $w_i, \cdots, w_j$. To facilitate later derivation, we set $S(i,j) = +\infty$ if these words exceed the length of a whole line L. Therefore, we have

$$S\left(i,j\right)=\left\{\begin{array}{cc} L-\sum\limits_{m=i}^{j-1}\left(c_m+1\right)-c_j, & \texttt{if } L\geq\sum\limits_{m=i}^{j-1}\left(c_m+1\right)-c_j \\ +\infty, & \texttt{otherwise}\end{array}\right., \tag{2}$$

where the summation $\sum_{m=i}^{j-1} c_m = 0$, if $i > j-1$. In order to reduce the complexity, $S(i,j)$ can be further computed based on the value of $S(i,j-1)$, i.e.,

$$S\left(i,i\right)=\left\{\begin{array}{c} L-c_i, L\geq c_i \\ +\infty,\ otherwise\end{array}\right., \tag{3}$$

$$S\left(i,j\right)=\left\{\begin{array}{cc} S\left(i,j-1\right)-c_j-1, & \text{if } S\left(i,j-1\right)<+\infty \text{ and } S\left(i,j-1\right)\geq c_j+1 \\ +\infty, & otherwise,\end{array}\right. \tag{4}$$

where $j \geq i+1$.

Define $OPT(k)$ as the sum of squares of slacks for the optimal solution with the words $w_1, \cdots, w_k$. As noted above, the optimal solution must have the following

$$OPT(k)=\min_{1\leq m\leq k}\left\{\left(S\left(m,k\right)\right)^2+OPT(m-1)\right\}, \tag{5}$$

and the base case is: $OPT(0) = 0$; Then the algorithm is as follows:

```
**for** $i = 1, \cdots, n$ **do**
  Compute $S(i,i)$ according to (3)
  **if** $i < n$ **then**
    **for** $j = i+1, \cdots, n$ **do**
      Compute $S(i,j)$ according to (4);
    **end for**
  **end if**
**end for**
$OPT(0) = 0$;
**for** $k = 1, \cdots, n$ **do**
  $OPT(k) = \min_{1\leq m\leq k}\left\{\left(S\left(m,k\right)\right)^2+OPT(m-1)\right\}$;
  Record the index $m$ which achieves the minimum, denoted as $m^*(k)$;
**end for**
Return $OPT(n)$;
Track back through the array $OPT$ by checking from $m^*(n)$ to $m^*(1)$ to
recover the optimal solution.
```

It takes $O(n)$ time to compute all $S(i,j)$ with fixed i by considering values of j in increasing order; thus, we can compute all $S(i,j)$ in $O(n^2)$ time. Each iteration of computing $OPT(k)$ takes $O(n)$ time, and there are $O(n)$ iterations. The tracking back operation takes $O(n)$ time. Therefore, the overall time complexity is $O(n^2)$.

4. (*Chapter 6, Q 10*)

   a) Consider the following example: there are totally 4 minutes, the numbers of steps that can be done respectively on the two machines in the 4 minutes are listed as follows (in time order):

      - Machine A: 2, 1, 1, 200
      - Machine B: 1, 1, 20, 100

      The given algorithm will choose A then move, then stay on B for the final two steps. The optimal solution will stay on A for the four steps.

   b) An observation is that, in the optimal solution for the time interval from minute 1 to minute i, you should not move in minute i, because otherwise, you can keep staying on the machine where you are and get a better solution ($a_i > 0$ and $b_i > 0$). For the time interval from minute 1 to minute i, consider that if you are on machine A in minute i, you either (i) stay on machine A in minute $i-1$ or (ii) are in the process of moving from machine B to A in minute $i-1$.

      Now let $OPT_A(i)$ represent the maximum value of a plan in miniute 1 through i that ends on machine A, and define $OPT_B(i)$ analogously for B. If case (i) is the best action to make for minute $i-1$, we have $OPT_A(i) = a_i + OPT_A(i-1)$; otherwise, we have $OPT_A(i) = a_i + OPT_B(i-2)$. In sum, we have

$$OPT_A\left(i\right) = a_i + \max\left\{OPT_A\left(i-1\right), OPT_B\left(i-2\right)\right\}. \tag{6}$$

      Similarly, we get the recursive relation for $OPT_B(i)$:

$$OPT_B\left(i\right) = b_i + \max\left\{OPT_B\left(i-1\right), OPT_A\left(i-2\right)\right\}. \tag{7}$$

   The algorithm initializes $OPT_A(0) = 0$, $OPT_B(0) = 0$, $OPT_A(1) = a_1$ and $OPT_B(1) = b_1$. Then the algorithm can be written as follows:

```
$OPT_A(0) = 0$; $OPT_B(0) = 0$;
$OPT_A(1) = a_1$; $OPT_B(1) = b_1$;
for $i = 2, \cdots, n$ do
  $OPT_A(i) = a_i + \max\{OPT_A(i-1), OPT_B(i-2)\}$;
  Record the action (either stay or move) in minute $i-1$ that achieves the maximum.
  $OPT_B(i) = b_i + \max\{OPT_B(i-1), OPT_A(i-2)\}$;
  Record the action in minute $i-1$ that achieves the maximum.
end for
Return $\max\{OPT_A(n), OPT_B(n)\}$;
Track back through the arrays $OPT_A$ and $OPT_B$ by checking the action records from minute $n-1$ to minute 1 to recover the optimal solution.
```

   It takes $O(1)$ time to complete the operations in each iteration; there are $O(n)$ iterations; the tracing backs takes $O(n)$ time. Thus, the overall complexity is $O(n)$.

5. (*Chapter 6, Q 16*) For each subtree T' of T, we define $OPT(T')$ to be the number of rounds it takes for everyone in T' to be notified, once the root has the message. Suppose T' has child sub-trees $T_1^{(T')}, T_2^{(T')}, \cdots, T_{d_{T'}}^{(T')}$, and we label them so that $OPT(T_1^{(T')}) \geq OPT(T_2^{(T')}) \geq \cdots \geq OPT(T_{d_{T'}}^{(T')})$.

   For tree T', here the root node should talk to its child sub-tree in decreasing order of the time it takes for their sub-trees (recursively) to be notified. This greedy strategy must be optimal for T' (You can use the "swapping method" you learned in class for greedy algorithm to show its optimality, which is left to you as an exercise). Then the recurrence can be expressed as

$$OPT(T') = \max_{1 \leq j \leq d_{T'}} \left\{ j + OPT(T_j^{(T')}) \right\}. \tag{8}$$

   Base case: if T' is simply a leaf node, then $OPT(T') = 0$.

   The full algorithm builds up the values $OPT(T')$ using the recurrence, beginning at the leaves and moving up to the root to obtain $OPT(T)$.

   We can write the algorithm in a recursive way:

```
Compute $Schedule(T)$;
Track back through the sorted orders at every sub-tree from the root to the
leafs, we can also reconstruct the sequence of phone calls that should be made.
$Schedule(T')${
if $T'$ is a leaf node then
  $OPT(T') = 0$;
  Return $OPT(T')$;
else
  for $j = 1, \cdots, d_{T'}$ do
    Compute $Schedule$($j$th child sub-tree);
  end for
  Sort the child sub-trees to get $T_1^{(T')}, T_2^{(T')}, \cdots, T_{d_{T'}}^{(T')}$ in descending order
  of $OPT(T_1^{(T')}), OPT(T_2^{(T')}), \cdots, OPT(T_{d_{T'}}^{(T')})$;
  Record this order for $T'$;
  $OPT(T') = \max_{1 \leq j \leq d_{T'}} \left\{ j + OPT(T_j^{(T')}) \right\}$;
  Return $OPT(T')$;
end if
}
```

   The time taken for sorting for T' is $O(d_{T'} \log(d_{T'}))$, if the optimal solutions of the child sub-trees of T' have been obtained. There are n sub-trees, so the aggregate sorting complexity is analyzed as follows

$$\sum_{T'} d_{T'} \log\left(d_{T'}\right) \leq \log(n-1) \cdot \sum_{T'} d_{T'} = (n-1)\log(n-1) = O\left(n \log(n)\right) \tag{9}$$

The aggregate complexity to compute all $OPT(T')$ after sorting is $O(n)$. Thus, the overall complexity is $O(n \log(n))$.

6. (*Chapter 6, Q 20*) Let the (i, h)-subproblem be the problem in which one wants to maximize one's grade on the first i courses, using at most h hours.

   Let $OPT(i, h)$ be the maximum total grade that can be achieved for this subproblem. Then $OPT(0, h) = 0$ for all h, and $OPT(i, 0) = \sum_{j=1}^{i} f_j(0)$. Now, in the optimal solution to the (i, h) subproblem, one spends k hours on course i for some value of $k \in \{0, 1, \cdots, h\}$; thus

$$OPT(i, h) = \max_{0 \leq k \leq h} \left\{ f_i(k) + OPT(i-1, h-k) \right\}. \tag{10}$$

   The algorithm can be summarized as follows:

```
**for** $h = 1, \cdots, H$ **do**
  $OPT(0, h) = 0$;
**end for**
$OPT(1, 0) = f_1(0)$;
**for** $i = 2, \cdots, n$ **do**
  $OPT(i, 0) = f_i(0) + OPT(i-1, 0)$;
**end for**
**for** $i = 1, \cdots, n$ **do**
  **for** $h = 1, \cdots, H$ **do**
    $OPT(i, h) = \max_{0 \leq k \leq h} \{f_i(k) + OPT(i-1, h-k)\}$;
    Record the $k$ that achieves the maximum, denoted as $k^*(i, h)$;
  **end for**
**end for**
Return $OPT(n, H)$;
Having obtained the $(n+1) \times (H+1)$ table with each entry filled with $OPT(i, h)$, track back from the $(n+1, H+1)$th entry using $\{k^*(i, h)\}$ until a boundary entry of the table is reached, in order to produce the optimal distribution of time.
```

   The total time to fill in each entry $OPT(i, h)$ is $O(H)$, and there are nH entries, for a total time of $O(nH^2)$. Tracking back according to the records takes $O(n)$ time. Thus, the overall time is $O(nH^2)$.

7. Solution:

Based on the assumption that you have to make at least one cut, an observation can be made: for a rope with length i, if the optimal cut includes a cut at length j, $1 \le j \le i-1$, then at least one of the following 4 cases must happen for the left part (with length j) and right part (with length $i-j$) after the cutting:

- no cut on the left part, optimal cut on the right part;
- optimal cut on the left part, no cut on the right part;
- optimal cut on the left part, optimal cut on the right part;
- no cut on the left part, no cut on the right part;

Define $OPT(i)$ as the maximum product of the lengths of smaller ropes after cutting the rope of length i. Based on above observation, we have the following recursive relation for length $i > 2$:

$$\begin{aligned} OPT\left(i\right) = \max_{1 \le j \le \left\lfloor \frac{i}{2} \right\rfloor} &\left\{ \max\left\{ j \times OPT\left(i-j\right), OPT\left(j\right) \times \left(i-j\right), \right.\right. \\ &\left.\left. OPT\left(j\right) \times OPT\left(i-j\right), j \times \left(i-j\right) \right\} \right\}, \end{aligned} \tag{11}$$

where j only need to takes values up to $\left\lfloor \frac{i}{2} \right\rfloor$ because of the symmetry of the rope; if $j = 1$, there is no possible cut on the part with length 1, we simply define $OPT(1) = 1$.

Base case: $OPT(2) = 1$, because it has to be cut at least once and this is the only one way to cut.

Then the algorithm can be summarized as

```
$OPT(1) = 1$; $OPT(2) = 1$;
for $i = 3, \cdots, n$ do
    Compute $OPT(i)$ according to (11);
    Record the $j$ and the $Action$ ("cut" or "no cut") on the corresponding left part and right part that achieve the maximum, denoted as $\{j^*(i), Action(i, left), Action(i, right)\}$;
end for
Return $OPT(n)$;
Track back through the array $OPT$, by recursively checking the records starting from $\{j^*(n), Action(n, left), Action(n, right)\}$ to get the optimal set of cutting positions.
```

Each iteration takes $O(n)$ time to compute; there are $O(n)$ iterations; tracking back with the help of records takes $O(n)$ time. Thus, the overall complexity is $O(n^2)$.

8. Solution:

   First index the bills from the leftmost one to the rightmost one as $1, \cdots, n$. Their corresponding values are $a_1, \cdots, a_n$. Define $OPT(i,j)$ as the maximum amount of money that Alice can get from the remaining bill sequence $a_i, \cdots, a_j$ if she gets the first turn to pick a bill, where $1 \le i \le n-1$; $2 \le j \le n$; $i < j$.

   Since n is even, whenever it is Alice's turn to pick the bill, the remaining bill sequence must have even number of bills, i.e., $j-i+1$ is positive and even.

   Thus, here is the recursive relation:for $j \ge i+3$, and $j-i+1$ is even

$$OPT\left(i,j\right) = \max\left\{Left\left(i,j\right), Right\left(i,j\right)\right\}, \tag{12}$$

   where

$$Left\left(i,j\right) = \left\{ \begin{array}{c} OPT\left(i+2,j\right)+a_i, \text{ if } a_{i+1} \ge a_j \\ OPT\left(i+1,j-1\right)+a_i, \text{ if } a_{i+1} < a_j \end{array} \right., \tag{13}$$

$$Right\left(i,j\right) = \left\{ \begin{array}{c} OPT\left(i+1,j-1\right)+a_j, \text{ if } a_i \ge a_{j-1} \\ OPT\left(i,j-2\right)+a_j, \text{ if } a_i < a_{j-1} \end{array} \right.. \tag{14}$$

   Here $Left(i,j)$ represents the maximum amount of money that Alice can get if she picks the leftmost bill of $a_i, \cdots, a_j$; $Right(i,j)$ represents the maximum amount of money that Alice can get if she picks the rightmost bill of $a_i, \cdots, a_j$.

   Base case: $OPT(i,i+1) = \max\{a_i, a_{i+1}\}$, for $i = 1, \cdots, n-1$.

   With the above recursive relation, we give the following algorithm:

```
for $k = 1, 3, \cdots, n-3, n-1$ do
  for $i = 1, 2, \cdots, n-k$ do
    if $k == 1$ then
      $OPT(i, i+k) = \max\{a_i, a_{i+1}\}$;
    else
      Compute $OPT(i, i+k)$ according to (12);
    end if
    Record the decisions made by Alice and the corresponding decision made
    by Bob;
  end for
end for
Return $OPT(1, n)$;
Track back through the table with entry $(i, j)$ filled with value $OPT(i, j)$, by
checking the records starting from the $(1, n)$th entry until a boundary entry
is reached.
```

   The running time in each iteration is $O(1)$; there are $O(n) \times O(n)$ iterations; tracking back with the records takes time $O(n)$. Thus, the overall running time is $O(n^2)$.