Coursework 1
Data Structures and Algorithms

Andreas Georgiou - ag14774

November 26, 2015

Question 2

Part a

Let $P[i,x]$ to be the maximum profit that can be obtained if you only run the shop for the first i day(s) and you start the i-th day with item x in stock. $U[i]$ is defined to be the profit you will make on the i-th day if you have umbrellas in stock and $S[i]$ is defined to be the profit you will make on the i-th day if you have suncream in stock.

Define $X_x[i]$ as a function that returns the profit of the shop on the i-th day with stock x:

$$X_x[i] = \begin{cases} U[i] & x = u \\ S[i] & x = s \end{cases}$$

Define $t(x)$ as a function that toggles x between umbrellas(u) and suncream(s). Formally:

$$t(x) = \begin{cases} u & x = s \\ s & x = u \end{cases}$$

The recursive formula is formally defined as (where m is the cost of switching between products):

$$P[i,x] = \begin{cases} X_x[i] & i = 1 \\ X_x[i] + max \begin{cases} P[i-1,x] \\ P[i-1,t(x)-m \end{cases} & otherwise \end{cases}$$

Part b

Consider the solution of $P[i,x]$. During day i the shop will make $X_x[i]$ profit. During the previous day $(i-1)$ the shop either sold the same product x as day i or product $t(x)$.

In the first case, since the shop sold the same product, it did not incur any costs going from day $i-1$ to day i. Therefore the total profit for the first i days will be the sum of the profit made on day i and the profit made during the first $i-1$ days, where the product sold during day $i-1$ was x.

Alternatively, the shop sold a different product on day $i-1$. In this case, the switching cost the shop m pounds. The total profit here, is the sum of the profit made on day i and the profit made during the first $i-1$ days, where the product sold during day $i-1$ was $t(x)$ minus the cost m of going from day $i-1$ to day i.

Finally, the shop can sell either umbrellas or suncream during the last day n. Therefore the solution to the problem is $max\{P[n,u], P[n,s]\}$.

Part f

```
(0)void cornershopI(){
(1)   for(int i=0;i<n;i++){
(2)      for(int j=0;j<2;j++){
(3)         if(i==0)
(4)            mem[j][i] = input[j][i];
(5)         else
(6)            mem[j][i] = Math.max(mem[j][i-1], mem[j^1][i-1]-m) + input[j][i];
(7)      }
(8)   }
(9)}
```

The time complexity of the above algorithm is $\Theta(n)$. The array $mem[j][i]$ holds the result of $P[i,j]$ where $j = 0$ represents u and $j = 1$ represents s. The inner for-loop(2) is only executed twice and therefore takes constant time. The time complexity of lines (3)-(6) is constant since the max operation also takes constant time. The function $t(x)$ here is implemented as a XOR bitwise operation on j. The total time complexity is only determined by the outer for-loop(1) which is executed n times.

$$T(n) = \sum 2_{i=1}^{n} = 2n = \Theta(n)$$

Question 3

Part a

Let v be the magician at the root of the tree(with $id = 0$). For each magician we define the following attributes/methods:

- $v.ability$: The ability of magician v.
- $v.children$: Returns a set with all the apprentices of magician v. (i.e. all nodes that are directly below of v in the tree hierarchy.)
- $v.grandchildren$: Returns a set with all the apprentices of all the apprentices of magician v (i.e. those that are 2 levels below of v in the tree hierarchy.)

Formally the recursive formula is:

$$magic(v) = max \begin{cases} \left[\sum_{\forall x \in v.grandchildren} magic\,(x)\right] + v.ability \\ \sum_{\forall x \in v.children} magic\,(x) \end{cases}$$

The above formula can be simplified by adding two additional attributes:

- $v.includeMe$: This stores the best magical ability if we include v in our solution. Initially set to $v.ability$
- $v.excludeMe$: This stores the best magical ability if we exclude v in our solution. Initially set to 0.

Then the formula can be broken into smaller sub-formulas:

$$v.excludeMe = \begin{cases} 0 & v.children = \emptyset \\ v.excludeMe + magic\,(x) & \forall x \in v.children \end{cases}$$

$$v.includeMe = \begin{cases} v.ability & v.children = \emptyset \\ v.ability + v.includeMe + x.excludeMe & \forall x \in v.children \end{cases}$$

$$magic(v) = max \begin{cases} v.excludeMe \\ v.includeMe \end{cases}$$

Part b

The problem of $magic(v)$ can be thought of as having only two cases. The first case is when node v itself will be included in the optimal set. Because of that, we skip the children directly below of v and we add to the total ability of the tree with root v the maximum ability $magic(x)$ for each one of its grandchildren x. This is only possible because of the fact that there are no cycles in the graph (i.e. there are no nodes in the same level directly connected with one another).

The second case is when node v is excluded from the optimal set. The total ability of the tree with root v will be the same as the sum of the magic abilities $magic(x)$ of each child $x \in v.children$.

The MAGIC-TOURNAMENT problem is similar to the problem of Weighted Scheduling Interval in the sense that we are trying to find compatible magicians. In the WIS problem, a function $p(i)$ returns the next compatible interval assuming that i will be in the optimal set. However, in this case we already know that the next compatible magician will be the set of all its grandchildren.

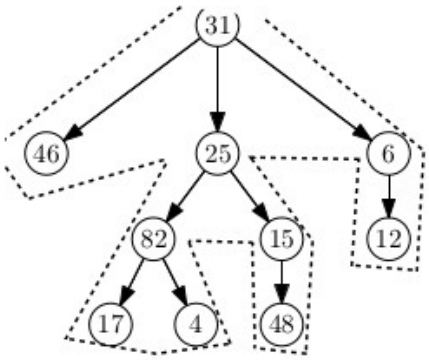
Part d[1]

```
(0)int magicTournament(Magician root){
(1)   root.includingMe = root.ability;
(2)   root.excludingMe = 0;
(3)   Iterator<Magician> it = root.getChildrenIterator();
(4)   while(it.hasNext()){
(5)      Magician child = it.next();
(6)      root.excludingMe += magicTournament(child);
(7)      root.includingMe += child.excludingMe;
(8)   }
(9)   return Math.max(root.includingMe, root.excludingMe);
(10)}
```

[1]The solution was inspired by Jeff Erickson's Maximum Independent Set algorithm.

The time complexity of the above algorithm is $\Theta(n)$. We are visiting each node only once in a depth first manner as shown in the picture below. For a node v, the while loop(4) is only executed k times where $k = v.children < n$. The total amount of times the loop is executed after the algorithm has finished running, will be $n - 1$ which is equal to the edges of the tree.

The algorithm shown here, is memoized because of the fact that we are storing the results in the tree itself. We are storing two values for each node: the value of the maximum ability that includes that node and the value of the maximum ability that excludes the root. This algorithm benefits from memoization, because in line (7) we are reusing the results calculated in the recursion call in line (6). Specifically, $child.excludingMe$ will already be calculated by the time that line gets executed because *child* was recursively passed as a parameter to the function during the execution of the previous line.



The tree is traversed in a depth first manner

Question 4

Part a

Let $D[i,j]$ be the maximum profit possible if you work only for the last i days(from day i to day n) and the last day-off was j days ago. Then the recursive formula is as follows:

$$D[i,j] = \begin{cases} min\{B[i], M[j]\} & i = n \\ max \begin{cases} min\{B[i], M[j]\} + D[i+1, j+1] \\ D[i+1, 1] \end{cases} & otherwise \end{cases}$$

Part b

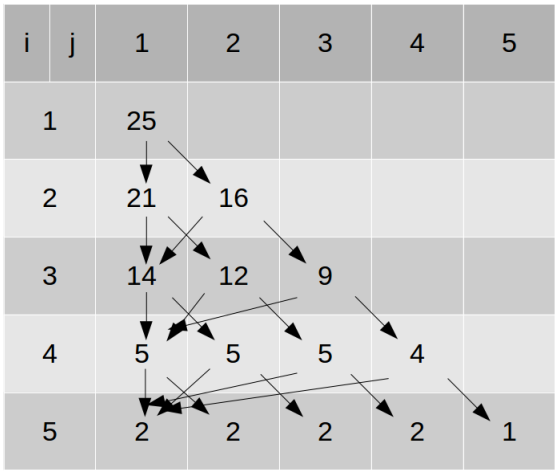
The problem consists of two cases. The first case, which is also the case when the recursion stops, is when you take into account only the last day of work(i.e. day n). Since there is no profit in taking the last day off, you can either deliver the maximum amount of sandwiches as allowed by regulations based on the last day off you have taken or deliver all orders for day i if the number of orders is within the allowance(i.e. $min\{B[i], M[j]\}$).

Lastly, the second case consists of two subcases, of which we want to choose the one that returns the most profits. The first subcase is that you work during that day and sell the maximum amount of sandwiches as allowed by law and the second is that you take that day off. In the first one, the profits would be equal to the amount of sandwiches sold during day i(always taking the minimum of $B[i]$ and $M[j]$ to make sure that no regulations are violated) plus the maximum amount of sandwiches sold during the days $i+1$ to n. The second subcase, since you take that day off, no sandwiches will be delivered, but the allowance can be reset. That is the profits in this case would be equal to the maximum amount of sandwiches sold during the day $i+1$ to n(taking into account the new allowance i.e. resetting the index of the array $M[j]$ back to 1).

Part c

The recursive formula directly outputs the correct solution to the Day-Off problem. Since we want the result taking into account all days and initially we have taken the previous day before taking control of the deliveries off, we start with $D[1,1]$.

Part f



The time complexity of the memoized algorithm is $\Theta(n^2)$. As we see in the table above, at worst, half of the array would have to be filled. That is, an isosceles triangle with side length n. Total cells are equal to $\frac{n^2}{2}$. Each cell in the picture chooses the maximum between the value pointed by the left arrow, and the value pointed by the right arrow plus $min\{B[i], M[j]\}$.

Part h

```
(0)int dayOffI(){
(1)   for(int i=n-1;i>=0;i--){
(2)      for(int j=0;j<=i;j++){
(3)         if(i==n-1)
(4)            mem[i][j] = Math.min(input[0][i],input[1][j]);
(5)         else
(6)            mem[i][j] = Math.max(Math.min(input[0][i],input[1][j])+mem[i+1][j+1],
    mem[i+1][0]);
(7)      }
(8)   }
(9)   return mem[0][0];
(10)}
```

The time complexity of the iterative algorithm is $\Theta\left(n^2\right)$. Here the two arrays $B[i]$ and $M[j]$ are combined in a single array called *input*. All lines will be executed in constant time $\Theta\left(1\right)$ except the two for-loops in lines (1) and (2). As it is shown in the picture in the previous part, we can see that the triangle has to be filled from the bottom up to satisfy all the dependencies. Specifically, the inner loop will be executed $n + (n-1) + \ldots + 1$ times:

$$T\left(n\right) = 1 + 2 + ... + n$$

$$T\left(n\right) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} = \Theta\left(n^2\right)$$

It might be possible to improve the time complexity to $\Theta\left(n \log n\right)$ using the fact that the array $M\left[j\right]$ is sorted with $M\left[j\right] \leq M\left[j-1\right]$ for all j.