Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — February 6, 2014
Professors Srini Devadas, Nancy Lynch and Vinod Vaikuntanathan — Problem Set 1

Problem Set 1

All parts are due Thursday, February 20 at 11:59PM. Please download the .zip archive for this problem set, and refer to the `README.txt` file for instructions on preparing your solutions. Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Your Name: Eunice Wu

Collaborators: Rebekah Cha, Jordan Powell

Part A

Problem 1-1.

(a) $f_3(n) < f_1(n) < f_2(n) < f_4(n) < f_5(n)$

(b) $f_4(n) < f_5(n) < f_3(n) < f_1(n) < f_2(n)$

(c) $f_3(n) < f_2(n) < f_1(n) < f_5(n) \sim f_4(n)$

Problem 1-2.

(a) $\theta(x)$

(b) $\theta(x \log(x))$

(c) $\theta(x \log(y))$

(d) $\theta(\log x \log y)$

(e) $\theta(logx + logy)$

Problem 1-3.

(a) The recurrence can be represented by a binary tree and therefore has a height O(log(number of columns)). Supposing that we have an array with dimensions n rows x m columns. Each step can be represented by

$$T(n, m) = \theta(n) + T(n, m/2)$$

where $f(n)$, the time it takes to divide/combine and execute each step, is $\theta(n)$ because to find the peak, each step involves finding the maximum of each column which runs through all n rows of the current midpoint column and therefore is O(n). From this we can conclude, using the Master's Theorem that an upperbound would be $\theta(n) * \log(m)$

(b) *Assume*: Recursive call returns peak for subproblem (given by problem)

Base Case: Number of Columns (m) = 1
For termination, the algorithm will always terminate when passed the base case because each step is strictly smaller than the previous step.
At the base case case, it is also guaranteed to return the correct answer because, assuming that the method peakProblem.getMaximum() is correct, finding the maximum is the only step needed for the base case Peak Problem. In the case where there is no dividing line, the "getBetterNeighbor" function just returns the previous "bestLoc," so at m = 1, the algorithm is correct.
Inductive Step: Let us assume that the algorithm is correct for an m-column array and therefore we know that it works for $\frac{m}{2}$ -column arrays as well. Each step of the algorithm halves the current problem, so we also know that $2m$ column arrays will return the correct answer because after one step, it will become an m-column problem.
Now, if we add another column, we will be at 2m+1 columns. There will be two cases upon finding the midpoint column's max:

Case 1: A neighbor is the better peak
If this is the case, then we will move on to apply the algorithm on that neighbor's corresponding $\frac{(2m+1)}{2}$ column array. We can see that $2m + 1$ will be an odd number, and thus $\frac{(2m+1)}{2}$ will round down to a m-column array problem, and therefore would return the correct answer with algorithm 3 and also terminate

Case 2: The current place is a peak and, therefore, the previous stored bestLoc is returned
If this is the case, then the program terminates and has returned a correct answer.

Problem 1-4.

(a) Algorithm 4 is almost exactly the same as algorithm 3, except that it incorporates an algorithm search when finding maximum in the rows of the divider column, not just for splitting the main problem into subproblems. So the only thing that changes is the time it takes per step, $f(n)$. Previously, the find maximum iterated through all n rows; now it continuously divides the n rows into subproblems, like a binary tree. This means that $f(n)$ is now $\theta(\log(n))$. It can be represented by:

$$T(n, m) = \theta(log(n)) + T(n, m/2)$$

Using the Master Theorem, we would get an upper bound of $\theta(\log(n) * \log(m))$

(b)

$$\begin{array}{ccccccc} 1 & 2 & 3 & 5 & 6 & 1 & 1 \\ 3 & 3 & 2 & 2 & 7 & 1 & 1 \\ 1 & 1 & 1 & 4 & 8 & 1 & 1 \\ 1 & 1 & 1 & 3 & 9 & 1 & 1 \\ 1 & 1 & 1 & 4 & 10 & 1 & 1 \\ 1 & 1 & 1 & 100 & 11 & 1 & 1 \end{array} \quad \rightarrow \quad \begin{array}{ccc} 6 & 1 & 1 \\ 7 & 1 & 1 \\ 8 & 1 & 1 \\ 9 & 1 & 1 \\ 10 & 1 & 1 \\ 11 & 1 & 1 \end{array}$$

Threw away the 100 by accident, because started searching in the direction of the 4 within the middle column! So this problem will return 11 as a peak, because it does not know that 100 is right beside it.

Problem 1-5.

(a) This can be defined by two recurrence equations, one for rows and one for columns, that alternate between each other:

$$T_r(n, m) = \theta(m) + T_c(n/2, m)$$

$$T_c(n, m) = \theta(n) + T_r(n, m/2)$$

They can also be expressed as one equation, or one step that decreases the array to one fourth its size:

$$T(n, m) = \theta(n + m) + T(n/2, m/2)$$

Using the Master Theorem, we can see that the upper bound for this is $\theta(m)$. This can also be found by drawing a tree, and summing up the work over each of the steps. We see we get:

$$\sum_{i}^{\infty} (n + m) * (1/2)^i = (n + m) * (\frac{1}{1 - 1/2})$$

(b) This algorithm seems to be correct. We know that we are guaranteed to find the maximum of each column/row, but like algorithm4, the problem comes when we throw away half of the problem at each step. An example of a counter example would be:

$$\begin{array}{cccccccc} 1 & 1 & 1 & 2 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 3 & 2 & 1 & 1 & 1 \\ 1 & 1 & 1 & 4 & 3 & 2 & 2 & 9 \\ 1 & 1 & 1 & 5 & 2 & 3 & 2 & 8 \\ 1 & 1 & 1 & 6 & 5 & 4 & 3 & 7 \\ 1 & 1 & 1 & 7 & 10 & 1 & 1 & 6 \\ 1 & 1 & 1 & 8 & 9 & 8 & 1 & 5 \end{array} \quad \text{or} \quad \begin{array}{ccccccccc} 1 & 1 & 1 & 1 & 2 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 3 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 4 & 3 & 2 & 2 & 2 \\ 1 & 1 & 1 & 1 & 5 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 6 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 7 & 10 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 8 & 9 & 8 & 1 & 1 \end{array}$$

Because the array ends up switching the dimension it is evaluating for maxes on, it will overlook certain parts of the array and will not return a correct answer.

Problem 1-6.

(a) Greedy Algorithm Pseudocode:

```
def greedy(loc = (0,0)):
     curr = Problem.get(loc)
     (nr,nc) = Problem.getBetterNeighbor((r,c))
     if (nr,nc) == curr
          return curr
     else
          return greedy((nr,nc))
```

(b) A good upperbound would be $\theta(n * m)$ because at worst, the algorithm would have to go through almost all cells of the array.

(c) The greedy algorithm is guaranteed to terminate because at each step we are either finding a peak or going to a larger number, meaning that it will not be possible to re-visit an already visited cell. This means that the problem is in a way "getting smaller" at each step, and thus we will eventually reach a state where the greedy algorithm will end.
The greedy algorithm is correct because at each step there are two cases:

Case 1: The current cell value has no larger neighbor
By definition, this cell is a peak and when returned would give us a correct answer.

Case 2: The current cell has a larger neighbor
We would move on to the that cell and the problem would get smaller by at least one cell. We would then call greedy again and eventually we would reach the state of Case 1, from which we would get a correct answer and also termination

Part B

Submit your implemented python script.