Problem Set 1

This problem set is divided into two parts: Part A problems are theory questions, and Part B problems are programming tasks.

Part A questions are due **Tuesday, September 21st** at **11:59PM**.

Part B questions are due **Thursday, September 23rd** at **11:59PM**.

Solutions should be turned in through the course website. Your solution to Part A should be in PDF form using LaTeX or scanned handwritten solutions. Your solution to Part B should be a valid Python file which runs from the command line.

A template for writing up solutions in LaTeX is available on the course website.

Remember, your goal is to communicate. Full credit will be given only to the 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.

Part A: Due Tuesday, September 21st

1. **(18 points)** Asymptotic Growth

   For each group of four functions below, rank the functions by increasing order of growth; that is, find an arrangement g_1, g_2, g_3, g_4 of the functions satisfying $g_1 = O(g_2)$, $g_2 = O(g_3)$, $g_3 = O(g_4)$. (For example, the correct ordering of n, n^2, n^3, n^4 is n, n^2, n^3, n^4.)

   (a) (6 points) Group 1:

   $$f_1(n) = n2^n \qquad f_2(n) = 2^{n/2} \qquad f_3(n) = n! \qquad f_4(n) = (n/e)^n$$

   (b) (6 points) Group 2:

   $$f_1 = (\log n)^{1000} \qquad f_2 = n^{1/2} \qquad f_3 = 1/n \qquad f_4 = n^{6.006}$$

   (c) (6 points) Group 3:

   $$f_1(n) = n^{100} \quad f_2(n) = n \log n \quad f_3(n) = \frac{1}{n}\binom{n}{100} \quad f_4(n) = \log(\log(n))$$

2. **(12 points)** Binary Search

   Binary search is a fast algorithm used for finding membership of an element in a sorted list. The recursive version of the algorithm is given below. The function takes a sorted list of numbers, `alist`, and a query, `item`, and returns true if and only if `item` $\in$ `alist`. Let n denote the length of the list `alist`.

```
def binarySearch(alist, item):
    if len(alist) == 0:
        return False
    else:
        midpoint = len(alist)/2
        if alist[midpoint]==item:
            return True
        else:
            if item<alist[midpoint]:
                return binarySearch(alist[:midpoint],item)
            else:
                return binarySearch(alist[midpoint+1:],item)
```

(a) **(5 points)** What is the runtime of the recursive version in terms of n, and why?

(b) **(7 points)** Write a concise proof of correctness for the algorithm. (Note: you may wish to use a loop invariant for your proof. See section 2.1 of CLRS for an example.)

3. **(20 points)** Uncoordinated Peak Finding

Recall the 2-dimensional Peak finding problem discussed in lecture. Consider the following algorithm for solving the problem, given a 2-dimensional integer matrix B of size nxn:

1 Let n = len(row(B)). Find the maximum element of the $(n/2)^{th}$ column and call it $c_{max} = B[i][n/2]$

2 **If** $c_{max} \geq B[i][n/2-1]$ and $c_{max} \geq B[i][n/2+1]$ **return** c_{max}

3 **If** $c_{max} < B[i][n/2-1]$ **then** B = B[0..(n-1)][0..n/2-1] **else** B=B[0..(n-1)][n/2+1..(n-1)]

4 Find the maximum element of the $(n/2)^{th}$ row of B and call it $r_{max} = B[n/2][j]$

5 **If** $r_{max} \geq B[n/2-1][j]$ and $r_{max} \geq B[n/2+1][j]$ **return** r_{max}

6 **If** $r_{max} < B[n/2-1][j]$ **then** B = B[0..n/2-1][0..n/2-1] **else** B=B[n/2+1..(n-1)][0..n/2-1]

7 **goto** Step 1.

(a) **(10 points)** Give a counterexample (an instance of B of size no larger than 7×7) where the above algorithm fails to find a peak in B even though it exists. (If your example is smaller than 7×7 and does not have a well-defined middle row or middle column, state which rows/columns you use as the middle rows/columns.)

(b) **(10 points)** We can fix the above algorithm by keeping track of the running maximum run_{max} as below. Explain how it solves the problem in the previous counterexample.

1 $run_{max} = -\infty$

2 Let n = len(row(B)). Find the maximum element of the $(n/2)^{th}$ column and call it $c_{max} = B[i][n/2]$

3 **If** $c_{max} \geq run_{max}$ **then**
4 **If** $c_{max} \geq B[i][n/2-1]$ and $c_{max} \geq B[i][n/2+1]$ **then return** c_{max}
5 **If** $c_{max} < B[i][n/2-1]$ **then** $run_{max} = B[i][n/2-1]$ **else** $run_{max} = B[i][n/2+1]$
6 **If** $c_{max} < B[i][n/2-1]$ **then** B=B[0..(n-1)][0..n/2-1] **else** B=B[0..(n-1)][n/2+1..(n-1)]
7 **Else** /* Update B to be the partition of B containing run_{max} */
8 **If** $run_{max} \in$ B[0..(n-1)][0..n/2-1] **then** B=B[0..(n-1)][0..n/2-1] **else** B=B[0..(n-1)][n/2+1..(n-1)]
9 Find the maximum element of the $(n/2)^{th}$ row of B and call it $r_{max} = B[n/2][j]$
10 **If** $r_{max} \geq run_{max}$ **then**
11 **If** $r_{max} \geq B[n/2-1][j]$ and $r_{max} \geq B[n/2+1][j]$ **return** r_{max}
12 **If** $r_{max} < B[n/2-1][j]$ **then** $run_{max} = B[n/2-1][j]$ **else** $run_{max} = B[n/2+1][j]$
13 **If** $r_{max} < B[n/2-1][j]$ **then** B = B[0..n/2-1][0..n/2] **else** B=B[n/2+1..(n-1)][0..n/2]
14 **Else** /* Update B to be the partition of B containing run_{max} */
15 **If** $run_{max} \in B[0..n/2-1][0..n/2-1]$ **then** $B = B[0..n/2-1][0..n/2-1]$ **else** $B = B[n/2+1..(n-1)][0..n/2-1]$
16 **goto** Step 2.

Part B: Due Thursday, September 23rd

1. **(50 points)** Peak Finding

   Consider an array A containing n integers. We define a *peak* of A to be an x such that $x = A[i]$, for some $0 \leq i < n$, with $A[i-1] \leq A[i]$ and $A[i] \geq A[i+1]$. In other words, a peak x is greater than or equal to its neighbors in A (for boundary elements, there is only one neighbor). Note that A might have multiple peaks.

   As an example, suppose $A = [10, 6, 4, 3, 12, 19, 18]$. Then A has two peaks: 10 and 19.

   Note that the absolute maximum of A is always a peak, but it requires $\Omega(n)$ time to compute.

   - **(20 points)** Write `quick_find_1d_peak` to compute any peak of array A in $O(\log(n))$ time using the algorithm described in the lecture.

   Now consider a three dimensional matrix B of integers of size $n \times n \times n$. We define the neighborhood of an element $x = B[i][j][k]$ as $B[i+1][j][k]$, $B[i-1][j][k]$, $B[i][j+1][k]$, $B[i][j-1][k]$, $B[i][j][k+1]$ and $B[i][j][k-1]$. For elements on a face we consider only five neigbors, for elements on an edge we consider only four neighbors, and for elements on the eight corners, only three neighbors are considered. x is defined to be a peak of B if and only

if it is greater than or equal to all of its neighbours. Note that the maximum element of B is a possible solution for x but finding it requires $\Omega(n^3)$ time.

For python coding help, the $O(n \log(n))$ algorithm described in the lecture is provided as `medium_find_2d_peak`.

- **(10 points)** Describe an algorithm to find a peak of a three dimensional matrix B in $O(n^2)$ time, and explain why the running time of your algorithm is $O(n^2)$.

- **(20 points)** Write `quick_find_3d_peak` to compute any peak of array B in $O(n^2)$ time using your algorithm.