Problem Set 1

Both theory and programming questions are due **Monday October 12** 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.

We will provide the solutions to the problem set 10 hours after the problem set is due, which you will use to find any errors in the proof that you submitted. You will need to submit a critique of your solutions by **Tuesday, September 20th, 11:59PM**. Your grade will be based on both your solutions and your critique of the solutions.

Collaborators: Charlie Griffin

Problem 1-1. [15 points] **Asymptotic Practice**

For each group of functions, sort the functions in increasing order of asymptotic (big-O) complexity:

(a) [5 points] **Group 1:**

$$\begin{aligned} f_1(n) &= n^{0.999999} \log n \\ f_2(n) &= 10000000n \\ f_3(n) &= 1.000001^n \\ f_4(n) &= n^2 \end{aligned}$$

Your Solution: 1, 2, 4, 3
f_1: logs grow very slowly asymptotically, so we can approximate this as being $O(n)$ (it should be a little slower).
f_2: $O(n)$ with a higher coefficient than f_1
f_4: $O(n^2)$
f_3: Asymptotically exponential functions grow faster than power functions.

(b) [5 points] **Group 2:**

$$\begin{aligned} f_1(n) &= 2^{2^{1000000}} \\ f_2(n) &= 2^{100000n} \\ f_3(n) &= \binom{n}{2} \\ f_4(n) &= n\sqrt{n} \end{aligned}$$

Your Solution: 1, 4, 3, 2
f_1: $O(1)$, does not grow with n
f_4: $O(n) < O(f_4) < O(n^2)$
f_2: $nC2 = \frac{n!}{2(n-2)!} = \frac{n \times (n-2)}{2} = O(n^2)$
f_3: Exponential

(c) [5 points] **Group 3:**

$$\begin{aligned} f_1(n) &= n^{\sqrt{n}} \\ f_2(n) &= 2^n \\ f_3(n) &= n^{10} \cdot 2^{n/2} \\ f_4(n) &= \sum_{i=1}^{n} (i+1) \end{aligned}$$

Your Solution: 4, 1, 3, 2 This is really simple if we put functions 2,3 and 4 in the same form.
f_4: $\sum_{i=1}^{n} (i+1) = \frac{n(n+3)}{2} = O(n^2)$
f_1: $n^{\sqrt{n}} = e^{\ln n \sqrt{n}} \approx 2^{\ln n \sqrt{n}}$
f_3: For $n >> 1$, $O(f_3) = O(2^{\frac{n}{2}})$
f_2: n grows faster than $\frac{n}{2}$, which grows faster than $\sqrt{n} \ln n$, so the exponentials fall in the same order

Problem 1-2. [15 points] **Recurrence Relation Resolution**

For each of the following recurrence relations, pick the correct asymptotic runtime:

(a) [5 points] Select the correct asymptotic complexity of an algorithm with runtime $T(n, n)$ where

$$\begin{aligned} T(x, c) &= \Theta(x) && \text{for } c \leq 2, \\ T(c, y) &= \Theta(y) && \text{for } c \leq 2, \text{ and} \\ T(x, y) &= \Theta(x + y) + T(x/2, y/2). \end{aligned}$$

1. $\Theta(\log n)$.
2. $\Theta(n)$.
3. $\Theta(n \log n)$.
4. $\Theta(n \log^2 n)$.
5. $\Theta(n^2)$.
6. $\Theta(2^n)$.

Your Solution: 2
Writing out the runtime recursively:
$T(n, n) = \Theta(2n) + \Theta(n) + \Theta(\frac{n}{2}) + \Theta(\frac{n}{4}) + \ldots$ This is a geometric sequence, which

is bounded above as described below:
$\Theta(2n) \geq T(n,n) \geq \Theta(4n)$
$\implies T(n,n)$ is $\Theta(n)$

(b) [5 points] Select the correct asymptotic complexity of an algorithm with runtime $T(n,n)$ where

$$\begin{array}{rcll} T(x,c) & = & \Theta(x) & \text{for } c \leq 2, \\ T(c,y) & = & \Theta(y) & \text{for } c \leq 2, \text{ and} \\ T(x,y) & = & \Theta(x) + T(x,y/2). & \end{array}$$

1. $\Theta(\log n)$.
2. $\Theta(n)$.
3. $\Theta(n \log n)$.
4. $\Theta(n \log^2 n)$.
5. $\Theta(n^2)$.
6. $\Theta(2^n)$.

Your Solution: 3
Writing out the runtime recursively:
$T(n,n) = \Theta(n) + \Theta(n) + \ldots + \Theta(n)$
The algorithm will repeat until $n \leq 2$, and n is divided by two in each recursive call. Thus there will be $\log_2 n$ terms in the sum. Using this information to simplify the sum we find:
$T(n,n) = \Theta(n) * \log_2 n = \Theta(n \log n)$

(c) [5 points] Select the correct asymptotic complexity of an algorithm with runtime $T(n,n)$ where

$$\begin{array}{rcll} T(x,c) & = & \Theta(x) & \text{for } c \leq 2, \\ T(x,y) & = & \Theta(x) + S(x,y/2), & \\ S(c,y) & = & \Theta(y) & \text{for } c \leq 2, \text{ and} \\ S(x,y) & = & \Theta(y) + T(x/2,y). & \end{array}$$

1. $\Theta(\log n)$.
2. $\Theta(n)$.
3. $\Theta(n \log n)$.
4. $\Theta(n \log^2 n)$.
5. $\Theta(n^2)$.
6. $\Theta(2^n)$.

Your Solution: 2
When the runtime is written recursively it becomes a geometric series, similar to part a.

$T(n,n) = \Theta(n) + \Theta(\frac{n}{2}) + \Theta(\frac{n}{2}) + \Theta(\frac{n}{4}) + \ldots$
$T(n,n) = \Theta(n) + 2\Theta(\frac{n}{2}) + 2\Theta(\frac{n}{4}) + \ldots$
$T(n,n) \leq \Theta(n) + 2\Theta(n)$
so $3\Theta(n) \geq T(n,n) \geq \Theta(n) \implies T(n,n)$ is $\Theta(n)$

Peak-Finding

In Lecture 1, you saw the peak-finding problem. As a reminder, a *peak* in a matrix is a location with the property that its four neighbors (north, south, east, and west) have value less than or equal to the value of the peak. We have posted Python code for solving this problem to the website in a file called `ps1.zip`. In the file `algorithms.py`, there are four different algorithms which have been written to solve the peak-finding problem, only some of which are correct. Your goal is to figure out which of these algorithms are correct and which are efficient.

Problem 1-3. [16 points] **Peak-Finding Correctness**

(a) [4 points] Is `algorithm1` correct?

1. Yes.
2. No.

Your Solution: 1
This was argued in lecture 1, algorithm1 is the 2d version of the 1d bisection search, which was presented and shown to be correct. Also the problem statement of 1-5 is a proof that this algorithm is correct.

(b) [4 points] Is `algorithm2` correct?

1. Yes.
2. No.

Your Solution: 1
This is the greedy ascent algorithm that was shown to be correct in lecture 1. It was guaranteed to work since in the worst case scenario it will check if every position is a peak (and return once one is found).

(c) [4 points] Is `algorithm3` correct?

1. Yes.
2. No.

Your Solution: 2
This will not work in all cases. A counter-example is provided in 1-6. To verify that this is a counter example run the following command in the ps1/ directory:
>>python main.py
>>Enter a file name to load from (default: problem.py): counterexample.py

A basic description of why the algorithm fails is that it can find a peak in an iterative subproblem that is not a peak in the original problem.

(d) [4 points] Is `algorithm4` correct?

1. Yes.
2. No.

Your Solution: 1
Yes, algorithm 4 is a more efficient version of algorithm 1. It works following the same procedure but rotating the problem between steps (so it alternates between rows and columns.) For a more detailed explanation see 1-5.

Problem 1-4. [16 points] **Peak-Finding Efficiency**

(a) [4 points] What is the worst-case runtime of `algorithm1` on a problem of size $n \times n$?

1. $\Theta(\log n)$.
2. $\Theta(n)$.
3. $\Theta(n \log n)$.
4. $\Theta(n \log^2 n)$.
5. $\Theta(n^2)$.
6. $\Theta(2^n)$.

Your Solution: 3
$T(n, n) = T(n, \frac{m}{2} + \Theta(n) = \log_2 m \times \Theta(n) = \Theta(n \log_2(n))$ The first term in the recurrence relations is the subproblem (half of the matrix). The second term corresponds to the cost of finding the global max along the dividing column. This was shown in lecture, and is very similar to the argument presented in 1-2 b.

(b) [4 points] What is the worst-case runtime of `algorithm2` on a problem of size $n \times n$?

1. $\Theta(\log n)$.
2. $\Theta(n)$.
3. $\Theta(n \log n)$.
4. $\Theta(n \log^2 n)$.
5. $\Theta(n^2)$.
6. $\Theta(2^n)$.

Your Solution: 5
As shown and argued in lecture 1, the algorithm could visit every matrix element and check the neighbors (cost $\Theta(1)$ to check neighbors), in a worst case scenario. There are $n \times n$ elements, so $T(n, n)$ is $\Theta(n^2)$.

(c) [4 points] What is the worst-case runtime of `algorithm3` on a problem of size $n \times n$?

1. $\Theta(\log n)$.
2. $\Theta(n)$.
3. $\Theta(n \log n)$.
4. $\Theta(n \log^2 n)$.
5. $\Theta(n^2)$.
6. $\Theta(2^n)$.

Your Solution: 2
$T(n, n) = T(\frac{n}{2}, \frac{n}{2}) + \Theta(m + n)$
The first term comes from the new subproblem (one quadrant of the matrix), and the second term is the cost to find the global max across the cross. This is another geometric series.
$T(n, n) = \Theta(2n) + \Theta(n) + \Theta(\frac{n}{2}) + \ldots < \Theta(4n)$
$\Theta(4n) > T(n, n) > \Theta(2n) \implies T(n, n)$ is $\Theta(n)$

(d) [4 points] What is the worst-case runtime of `algorithm4` on a problem of size $n \times n$?

1. $\Theta(\log n)$.
2. $\Theta(n)$.
3. $\Theta(n \log n)$.
4. $\Theta(n \log^2 n)$.
5. $\Theta(n^2)$.
6. $\Theta(2^n)$.

Your Solution: 2
$T(m, n) = \Theta(n) + T(\frac{n}{2}, m)$
The first term is from finding the global max along the dividing row/column, the second term is the new subproblem, which has half the columns/rows, and it searches rows if columns were removed last time, as well as the converse, which is why the ordering of m and n are switched here. Writing this out it also reveals itself to be a geometric series.
$T(n, n) = \Theta(n) + \Theta(n) + \Theta(n2) + \Theta(n2) + \ldots$
$T(n, n) = 2\left(\Theta(n) + \Theta(\frac{n}{2}) + \ldots\right) \leq 2\left(2\Theta(n)\right)$ This argument has been presented enough times in this solution for it to be clear $T(n, n)$ is $\Theta(n)$.

Problem 1-5. [19 points] **Peak-Finding Proof**

Please modify the proof below to construct a proof of correctness for the *most efficient correct algorithm* among `algorithm2`, `algorithm3`, and `algorithm4`.

The following is the proof of correctness for `algorithm1`, which was sketched in Lecture 1.

We wish to show that `algorithm1` will always return a peak, as long as the problem is not empty. To that end, we wish to prove the following two statements:

1. If the peak problem is not empty, then `algorithm1` will always return a location. Say that we start with a problem of size $m \times n$. The recursive subproblem examined by `algorithm1` will have dimensions $m \times \lfloor n/2 \rfloor$ or $m \times (n - \lfloor n/2 \rfloor - 1)$. Therefore, the number of columns in the problem strictly decreases with each recursive call as long as $n > 0$. So `algorithm1` either returns a location at some point, or eventually examines a subproblem with a non-positive number of columns. The only way for the number of columns to become strictly negative, according to the formulas that determine the size of the subproblem, is to have $n = 0$ at some point. So if `algorithm1` doesn't return a location, it must eventually examine an empty subproblem.

We wish to show that there is no way that this can occur. Assume, to the contrary, that `algorithm1` does examine an empty subproblem. Just prior to this, it must examine a subproblem of size $m \times 1$ or $m \times 2$. If the problem is of size $m \times 1$, then calculating the maximum of the central column is equivalent to calculating the maximum of the entire problem. Hence, the maximum that the algorithm finds must be a peak, and it will halt and return the location. If the problem has dimensions $m \times 2$, then there are two possibilities: either the maximum of the central column is a peak (in which case the algorithm will halt and return the location), or it has a strictly better neighbor in the other column (in which case the algorithm will recurse on the non-empty subproblem with dimensions $m \times 1$, thus reducing to the previous case). So `algorithm1` can never recurse into an empty subproblem, and therefore `algorithm1` must eventually return a location.

2. If `algorithm1` returns a location, it will be a peak in the original problem. If `algorithm1` returns a location (r_1, c_1), then that location must have the best value in column c_1, and must have been a peak within some recursive subproblem. Assume, for the sake of contradiction, that (r_1, c_1) is not also a peak within the original problem. Then as the location (r_1, c_1) is passed up the chain of recursive calls, it must eventually reach a level where it stops being a peak. At that level, the location (r_1, c_1) must be adjacent to the dividing column c_2 (where $|c_1 - c_2| = 1$), and the values must satisfy the inequality $val(r_1, c_1) < val(r_1, c_2)$.

Let (r_2, c_2) be the location of the maximum value found by `algorithm1` in the dividing column. As a result, it must be that $val(r_1, c_2) \leq val(r_2, c_2)$. Because the algorithm chose to recurse on the half containing (r_1, c_1), we know that $val(r_2, c_2) < val(r_2, c_1)$. Hence, we have the following chain of inequalities:

$$val(r_1, c_1) < val(r_1, c_2) \leq val(r_2, c_2) < val(r_2, c_1)$$

But in order for `algorithm1` to return (r_1, c_1) as a peak, the value at (r_1, c_1) must have been the greatest in its column, making $val(r_1, c_1) \geq val(r_2, c_1)$. Hence, we have a contradiction.

Your Solution: 1. If the peak problem is not empty, then algorithm 4 will always return a location.

Say that we start with a problem of size $m \times n$. The recursive subproblem examined by algorithm 4 will have dimensions $m \times \lfloor \frac{n}{2} \rfloor$ or $m \times (n - \lfloor \frac{n}{2} \rfloor - 1)$ if the columns are split and $\lfloor \frac{m}{2} \rfloor \times n$ or $(m - \lfloor \frac{m}{2} \rfloor - 1) \times n$ if the rows are split on the recursive function call. Therefore the number of columns and rows strictly decrease as long as $n, m > 0$. So algorithm 4 either returns a location at some point, or eventually examines a subproblem with a non-positive number of columns or rows. For the number of columns or rows to become strictly negative, n or m must becomes 0 at some point, so if algorithm 4 does not return a location, it must eventually examine an empty subproblem. We want to show that this cannot occur.
Assume algorithm 4 examines an empty subproblem. Before this it must examine a problem which has 1 or 2 rows or columns and is dividing along the dimension (rows if number of rows is 1 or 2, columns if columns meet this criteria) that is headed towards zero. If there is one row or column, the max along that row or column is the max of the problem, meaning that it is a peak, which is returned. The remaining case is 2 rows or columns. For this scenario there are two possibilities.

1.the max is in the searched (dividing) row/column

2.there is a strictly better neighbor in the remaining/adjacent row/column.

In case 1 the max is a peak and is returned. In case 2 the higher neighbor is identified, and a recursive call is made (after a call cutting the other dimension in half) reducing the number of rows/columns to 1, which was discussed previously in this proof.
Thus algorithm 4 will never recurse into an empty subproblem $\implies$ algorithm 4 always returns a location.

2. If algorithm 4 returns a location it will be a peak in the original problem.
Suppose algorithm 4 returns a peak (r, c) which is not a peak in the original problem.
(r, c) must have been a peak in a recursive subproblem if it is returned. If it is not a peak in the original subproblem, then we can undo the recursive subproblems (move out of the recursive depth) to a level where (r, c) is no longer a peak. Notice the variable bestSeen records the highest value recorded thus far. Call the location stored by bestSeen (rb, cb). For a return call to be made val$(r, c) \geq$ val(bestSeen).
To not be a peak in a previous subproblem, one of the neighbors to (r, b) must be greater than it. For it to be a peak of a smaller subproblem, the larger value must have lied on an edge (column/row) that was previously used to create the smaller subproblem (through division). Call this better neighbor (rt, ct). In the process of creating the smaller subproblem, that column or row was checked for maxima with the call
bestLoc = problem.getMaximum(divider,trace).
There exist two cases for recursing into a subproblem with (r, c).
Case 1: There was a better neighbor along the dividing row. We know this to be valse since val$(r, c) >$ val(rt, ct) must be true for (r, c) to be left in the subproblem.
Case 2: There are no better neighbors, but the max value, which is $\geq (rt, ct) < (rb, cb)$ Then val$(r, c) <$val$(rt, ct) \leq$val$(rb, cb) \leq$ val(r, c). This is a contradiction so neither case is valid.
These cases are exhaustive from the line:
if neighbor == bestLoc and problem.get(bestLoc) $\geq=$ problem.get(bestSeen) . . . return bestLoc.

So there is no scenario in which the location returned is not a peak in the original problem, nor is there a case in which no location is returned. Thus a peak is always returned by algorithm 4.

Problem 1-6. [19 points] **Peak-Finding Counterexamples**

For each incorrect algorithm, upload a Python file giving a counterexample (i.e. a matrix for which the algorithm returns a location that is not a peak).

Your Solution: see counterexample.py