Problem Set 1

Both theory and programming questions are due **Thursday, September 15** 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: None.

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

(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

(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

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

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

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

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: 3

(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

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

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

1. Yes.
2. No.

Your Solution: 1

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

1. Yes.
2. No.

Your Solution: 2

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

1. Yes.
2. No.

Your Solution: 1

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: 1

(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

(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

(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

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:

Proof for algorithm 2:

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 $m \times n$ and in fact we a lways face the same problem. We assume that the `algorithm1` have not return a position, but according to our algorithm, the current ckecking element must change to a bigger element to move forward, otherwise the current element will be a peak and was returned. However, since there are infinite elements, we must have biggest elements, so after enough moves, we will checking the biggest element, so it is of course a peak and was returned.

2. If `algorithm1` returns a location, it will be a peak in the original problem. Seeing the return condition of the `algorithm1`, it just check the element is the biggest element locally, if so, the element will be returned, so the it must a peak in the original problem.

Proof for algorithm 4:

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. 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 $1 \times n$, If the problem is of size $m \times 1$ or $1 \times n$, because if smallest num of cows or column must greater than 2, it can't lead to a empty subproblem, and it also can't be 2 since you will get a peak or a one-line subproblem,

then calculating the maximum of the central column or row 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. 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.

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 or the dividing col, say it is $(r_1, c_2$, and the values must satisfy the inequality $val(r_1, c_1) < val(r_1, c_2)$. However, we know that since (r_1, c_1) is returned, it must the biggest values that the algorithm ever saw, so there must have $(r_1, c_1) \geq (r_1, c_2)$. Hence, we have a contradiction.

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:

```
The counterexample of algorithm 3.
problemMatrix = [
[ 4,  5,  6,  7,  8,  7,  6,  5,  4,  3,  2],
[ 5,  6,  7,  8,  9,  8,  7,  6,  5,  4,  3],
[ 6,  7,  8,  9, 10,  14,  13,  7,  6,  5,  4],
[ 7,  8,  9, 10, 11, 10,  9,  8,  7,  6,  5],
[ 8,  9, 10, 11, 10, 11, 10,  16,  11,  7,  6],
[ 7,  8,  9, 10, 11, 10,  9,  15,  13,  6,  5],
[ 6,  7,  8,  9, 10,  9,  8,  7,  6,  5,  4],
[ 5,  6,  7,  8,  9,  8,  7,  6,  5,  4,  3],
[ 4,  5,  6,  7,  8,  7,  6,  5,  4,  3,  2],
[ 3,  4,  5,  6,  7,  6,  5,  4,  3,  2,  1],
[ 2,  3,  4,  5,  6,  5,  4,  3,  2,  1,  0]
]
```