Problem set 1:

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:**

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

Order:

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

Explanation:

- $f_1(n) < f_2(n)$ because

$f_1(n) = n^{0.999999} \log n = O(n^{0.999999}) \times O(n^{0.000001})$

(for any $c > 0$, $\log n = O(n^c)$)

$= O(n^{0.999999} \times n^{0.000001}) = O(n) = O(f_2(n))$

- $f_2(n) < f_4(n)$ because $f_2(n)$ is linear and $f_4(n)$ is quadratic

- $f_4(n) < f_3(n)$ because $f_4(n)$ is quadratic, $f_3(n)$ is exponential

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

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

Order: $f_1(n) < f_4(n) < f_3(n) < f_2(n)$

- $f_1(n)$ is a constant $\rightarrow$ $f_1(n)$ is the least
- $f_4(n) = n\sqrt{n} = n^{1.5}$

  $f_3(n) = \binom{n}{2} = \frac{n(n-1)}{2} = \frac{n^2-n}{2} = \Theta(n^2)$

$\Rightarrow f_4(n) = n^{1.5} = O(n^2) = O(f_3(n))$

- $f_2(n) > f_3(n)$ because $f_2(n)$ is exponential, $f_3(n)$ is quadratic.

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

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

Order: $f_4(n) < f_1(n) < f_3(n) < f_2(n)$

- $f_4(n) = \sum_{i=1}^{n} (i+1) = \frac{n[(n+1)+2]}{2} = \frac{n(n+3)}{2}$

$= \Theta(n^2)$

$f_1(n) = n^{\sqrt{n}} = n^{n^{\frac{1}{2}}} = (2^{\log n})^{\sqrt{n}} = 2^{\sqrt{n}\log n}$

$f_3(n) = n^{10} \cdot 2^{n/2} = 2^{\log(n^{10})} \times 2^{n/2}$

$= 2^{n/2 + \log(n^{10})} = 2^{n/2 + 10\log(n)}$

while $\sqrt{n}\log n$ grows more slowly than linear time, $\frac{n}{2} + 10\log n$ grows linearly with n.

$\rightarrow f_1(n) = O(f_3(n)) \Rightarrow f_3(n) > f_1(n)$

- We have $O(n^2) < O(2^n) \Rightarrow f_4(n) < f_1(n)$
- $f_3(n)$ is $O(f_2(n))$, but $f_2(n)$ is not $O(f_3(n)) \Rightarrow f_2(n) > f_3(n)$

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. $\Theta(2^n)$.

Ans: $\Theta(n)$

Explanation:

$$T(x,y) = c(x+y) + T\left(\frac{x}{2}, \frac{y}{2}\right)$$

$$= c(x+y) + c\left(\frac{x}{2}, \frac{y}{2}\right) + T\left(\frac{x}{4}, \frac{y}{4}\right)$$

$$= c(x+y) + c\left(\frac{x}{2} + \frac{y}{2}\right) + c\left(\frac{x}{4}, \frac{y}{4}\right) + T\left(\frac{x}{8}, \frac{y}{8}\right) + \dots$$

Thus: $c(x+y) \leq T(x,y) \leq 2c(x+y)$

$\rightarrow T(x,y) = \Theta(x+y)$

$\rightarrow T(n,n) = \Theta(2n) = \Theta(n)$

(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. $\Theta(2^n)$.

Ans: $\Theta(n \log n)$

Explanation:

$$T(x,y) = c(x) + T\left(x, \frac{y}{2}\right)$$

$$= c(x) + c(x) + T\left(x, \frac{y}{4}\right)$$

$$= \dots$$

$$= \underbrace{c(x) + c(x) + \dots + c(x)}_{\log y \text{ times}}$$

→ $T(x,y) = \Theta(x \log y)$

→ $T(n,n) = \Theta(n \log n)$

(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. $\Theta(2^n)$.

Ans: $\Theta(n)$

Explanation:

$$T(x,y) = \Theta(x) + S\left(x, \frac{y}{2}\right)$$

$$= \Theta(x) + \Theta\left(\frac{y}{2}\right) + T\left(\frac{x}{2}, \frac{y}{2}\right)$$

(as $S\left(x, \frac{y}{2}\right) = \Theta\left(\frac{y}{2}\right) + T\left(\frac{x}{2}, \frac{y}{2}\right)$)

$$= \Theta(x) + \Theta\left(\frac{y}{2}\right) + \Theta\left(\frac{x}{2}\right) + \Theta\left(\frac{y}{4}\right) + T\left(\frac{x}{4}, \frac{y}{4}\right)$$

$$= \Theta(x) + \Theta\left(\frac{y}{2}\right) + \Theta\left(\frac{x}{2}\right) + \Theta\left(\frac{y}{4}\right) + \Theta\left(\frac{x}{4}\right) + \Theta\left(\frac{y}{8}\right)$$

$+ \cdots$

$$\Rightarrow \Theta(x) + \Theta\left(\frac{y}{2}\right) \leq T(x) \leq 2\left[\Theta(x) + \Theta\left(\frac{y}{2}\right)\right]$$

$$\Rightarrow T(x,y) = \Theta\left(x + \frac{y}{2}\right)$$

$$\Rightarrow T(n,n) = \Theta\left(\frac{3n}{2}\right) = \Theta(n)$$

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.

a) Yes. This is a $\Theta(\log \Theta)$ algorithm

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

1. Yes.
2. No.

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

1. Yes.
2. No.

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

1. Yes.
2. No.

b). Yes. Algorithm 2 always return a location because the value of location that it stores strictly increases with each recursive call. There are only a finite number values in the grid. Thus, it eventually returns a value, which is a peak

c) No.

Counter example: a matrix for which the algorithm not returns a peak

problemMatrix =

$$\begin{bmatrix} 0 & 0 & 9 & 8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

d) yes.

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 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.

- Some modifications can be made to the proof of correctness for algo 1 :

1. More variance in the sizes of

the recursive subproblems as we can split by rows or columns

2. An ability to split by rows or columns
→ the number of columns is not neccessary to be strictly decreasing. With every step, either number of rows or columns strictly decrease as long as they >0. However, it does not mean that the algo must return a location or excamine an empty subproblem.

3. If the algo splits on rows, it is no longer true that the size of the problem is $m \times 1$ or $m \times 2$, but $1 \times n$ or $2 \times n$.

4. We do not know that the value returned by the algo is the maximum in

some row or column of the original problem. Instead, it is about the use of bestSeen variable, which contains the location of the best value in the matrix.

Adjustment to the proof:

1) If the peak problem is not empty then the algo will always return a location.

Proof:

We start the problem of size $m \times n$. Depending on whether the algo is splitting rows or columns, the recursive subproblem examined by the algo will have dimensions $m \times \left(n - \left[\frac{n}{2}\right] - 1\right)$, $\left[\frac{m}{2}\right] \times n$, $\left(m - \left[\frac{m}{2}\right] - 1\right)$ $\times n$ or $m \times \left[\frac{n}{2}\right]$. Therefore, with each recursive call, either the number of rows or columns strictly decreases as long as both are greater

than 0. The algorithm either halts and return at some point, or eventually examines a sub-problem with a non-positive number of rows or columns. The only way for the number of rows or columns to become strictly negative is to have $m = 0$ or $n = 0$ at some point. Thus, if the algo does not 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, the algo does not examine an empty subproble Without loss of gener-ality, the algorithm splits the columns just prior to this. At that point of the algorithm, it must have examined a subproblem of size $m \times 1$ or $m \times 2$

• Size $m \times 1$: maximum of the central column is equivalent to the maximum of the entire problem ⟶ the maximum must be a peak

→ the algo will halt and return a location

• Size $m \times 2$: 2 possibilities:

+ Maximum of the central column is a peak

+ It has a strictly better neighbor in the other column (in which case the algo will recurse on the subproblem with dimension $m \times 1$, thereby ensuring that the algo will always recurse into the non-empty subproblem). So the algorithm can never recurse into an empty subproblem, and therefore the algorithm must return a location.

2. If the algo returns a location, it will be a peak in the original problem. If the algo

returns a location (r_1, c_1), then that location must have been a peak within some recursive problem. Additionally, if (r_2, c_2) is the location of the best location seen during the execution of the algorithm (the location stored in the variable bestSeen), it must be that $val(r_1, c_1) \geq val(r_2, c_2)$

In the contradiction, that (r_1, c_1) is not a peak within the original problem. Then 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. Hence, it must be that the subproblem considered at that level includes some neighbor (r_3, c_3) of (r_1, c_1) with $val(r_1, c_1) < val(r_3, c_3)$.

In order for (r_3, c_3) to be adjacent to the recursive subproblem, but not included, it has to have been in the dividing row or dividing column. Thus, (r_3, c_3) must have been examined during the progression of the algo. As a result, it must be that $val(r_3, c_3) \leq val(r_2, c_2)$. Hence, we have the following chain of inequalities:

$$val(r_1, c_1) < val(r_3, c_3) \leq val(r_2, c_2) \leq val(r_1, c_1) \rightarrow \text{Contradiction}$$

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).

Algorithm 1, 2, and 4 are correct

Counterexample for algorithm 3:

problemMatrix =

$$\begin{bmatrix} 0 & 0 & 9 & 8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$