Introduction to Algorithms: 6.006
Massachusetts Institute of Technology
Professors Erik Demaine and Srini Devadas

September 16, 2011
Problem Set 1 Solutions

Problem Set 1 Solutions

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:**

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

Solution: The correct order of these functions is $f_1(n), f_2(n), f_4(n), f_3(n)$. To see why $f_1(n)$ grows asymptotically slower than $f_2(n)$, recall that for any $c > 0$, $\log n$ is $O(n^c)$. Therefore we have:

$$f_1(n) = n^{0.999999} \log n = O(n^{0.999999} \cdot n^{0.000001}) = O(n) = O(f_2(n))$$

The function $f_2(n)$ is linear, while the function $f_4(n)$ is quadratic, so $f_2(n)$ is $O(f_4(n))$. Finally, we know that $f_3(n)$ is exponential, which grows much faster than quadratic, so $f_4(n)$ is $O(f_3(n))$.

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

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

Solution: The correct order of these functions is $f_1(n), f_4(n), f_3(n), f_2(n)$. The variable n never appears in the formula for $f_1(n)$, so despite the multiple exponentials, $f_1(n)$ is constant. Hence, it is asymptotically smaller than $f_4(n)$, which does grow with n. We may rewrite the formula for $f_4(n)$ to be $f_4(n) = n\sqrt{n} = n^{1.5}$. The value of $f_3(n) = \binom{n}{2}$ is given by the formula $n(n-1)/2$, which is $\Theta(n^2)$. Hence, $f_4(n) = n^{1.5} = O(n^2) = O(f_3(n))$. Finally, $f_2(n)$ is exponential, while $f_3(n)$ is quadratic, meaning that $f_3(n)$ is $O(f_2(n))$.

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

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

Solution: The correct ordering of these functions is $f_4(n), f_1(n), f_3(n), f_2(n)$. To see why, we first use the rules of arithmetic series to derive a simpler formula for $f_4(n)$:

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

This is clearly asymptotically smaller than $f_1(n) = n^{\sqrt{n}}$. Next, we want to compare $f_1(n)$, $f_2(n)$, and $f_3(n)$. To do so, we transform both $f_1(n)$ and $f_3(n)$ so that they look more like $f_3(n)$:

$$\begin{aligned} f_1(n) &= n^{\sqrt{n}} = (2^{\lg n})^{\sqrt{n}} = 2^{\sqrt{n}\cdot\lg n} \\ f_3(n) &= n^{10} \cdot 2^{n/2} = 2^{\lg(n^{10})} \cdot 2^{n/2} = 2^{n/2+10\lg n} \end{aligned}$$

The exponent of the 2 in $f_1(n)$ is a function that grows more slowly than linear time; the exponent of the 2 in $f_3(n)$ is a function that grows linearly with n. Therefore, $f_1(n) = O(f_3(n))$. Finally, we wish to compare $f_3(n)$ with $f_2(n)$. Both have a linear function of n in their exponent, so it's tempting to say that they behave the same asymptotically, but they do not. If c is any constant and $g(x)$ is a function, then $2^{cg(x)} = (2^c)^{g(x)}$. Hence, changing the constant of the function in the exponent is the same as changing the base of the exponent, which does affect the asymptotic running time. Hence, $f_3(n)$ is $O(f_2(n))$, but $f_2(n)$ is *not* $O(f_3(n))$.

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. $\Theta(2^n)$.

Solution: The correct answer is $\Theta(n)$. To see why, we rewrite the recurrence relation to avoid Θ notation as follows:

$$T(x, y) = c(x + y) + T(x/2, y/2).$$

We may then begin to replace $T(x/2, y/2)$ with the recursive formula containing it:

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

This geometric sequence is bounded from above by $2c(x + y)$, and is obviously bounded from below by $c(x + y)$. Therefore, $T(x, y)$ is $\Theta(x + y)$, and so $T(n, n)$ 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. $\Theta(2^n)$.

Solution: The correct answer is $\Theta(n \log n)$. To see why, we rewrite the recurrence relation to avoid Θ notation as follows:

$$T(x, y) = cx + T(x, y/2).$$

We may then begin to replace $T(x, y/2)$ with the recursive formula containing it:

$$T(x, y) = \underbrace{cx + cx + cx + \ldots + cx}_{\Theta(\log y) \text{ times}}.$$

As a result, $T(x, y)$ is $\Theta(x \log y)$. When we substitute n for x and y, we get that $T(n, n)$ is $\Theta(n \log n)$.

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

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

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

Solution: The correct answer here is $\Theta(n)$. To see why, we want to first eliminate the mutually recursive recurrence relations. To do so, we will replace all references to the function $S(x, y)$ with the definition of $S(x, y)$. This yields the following recurrence relation for $T(x, y)$:

$$T(x, y) = \Theta(x) + \Theta(y/2) + T(x/2, y/2).$$

We can rewrite this to eliminate the constants and get the recurrence $T(x, y) = \Theta(x + y) + T(x/2, y/2)$. This is precisely the same recurrence relation as seen in part (a) of this problem, so it must have the same complexity.

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.

Solution: Yes. This is the $\Theta(n \log n)$ algorithm whose proof was sketched in Lecture 1. A more rigorous version of the proof of correctness was included in this homework.

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

1. Yes.
2. No.

Solution: Yes. This is the same as the greedy ascent algorithm presented in Lecture 1. The algorithm will always eventually return a location, because the value of location that it stores strictly increases with each recursive call, and there are only a finite number of values in the grid. Hence, it will eventually return a value, which is always a peak.

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

1. Yes.
2. No.

Solution: No. To see that this is true, try running the algorithm on the counterexample given in the solutions to Problem 1-6.

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

1. Yes.
2. No.

Solution: Yes. See Problem 1-5 for a proof of correctness.

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. $\Theta(2^n)$.

Solution: The worst-case runtime of `algorithm1` is $\Theta(n \log n)$, as explained in Lecture 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. $\Theta(2^n)$.

Solution: The worst-case runtime of `algorithm2` is $\Theta(n^2)$, as explained in Lecture 1.

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

Solution: The worst-case runtime of `algorithm3` is $\Theta(n)$. To see why, note that a single call of the function (not counting the cost of the recursive call) does work proportional $m+n$, where m is the number of rows and n is the number of columns. The recursive subproblem examined involves roughly half the number of rows and half the number of columns. Hence, the recurrence relation for this algorithm is approximately $T(m,n) = \Theta(m+n) + T(m/2, n/2)$. This is precisely the recurrence relation we solved in Problem 1-2(a).

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

Solution: The total runtime of the algorithm in the worst case is $\Theta(n)$. The algorithm alternates between splitting

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.

Solution: To prove that `algorithm4` is correct, we make the following modifications to the proof of correctness for `algorithm1`:

1. There is more variation in the sizes of the recursive subproblems, because we can split by rows as well as by columns.

2. Because we can split by rows or columns, it is no longer true that the number of columns is always strictly decreasing. With every step, either the number of rows or the number of columns strictly decreases, as long as both are greater than zero. However, this does still mean that the algorithm must either return a location, or examine an empty subproblem.
3. It is no longer true that the size of the problem just prior to examining an empty subproblem is $m \times 1$ or $m \times 2$. If the algorithm splits on rows instead of columns just prior to examining an empty subproblem, then the size of the problem must be $1 \times n$ or $2 \times n$. Even so, the analyses of the two cases (splitting on rows versus splitting on columns) are nearly identical.
4. The second part of the proof diverges more from the proof given for `algorithm1`. We no longer know that the value returned by the algorithm is the maximum in some row or column of the original problem. Instead, we base our argument about the correctness of the algorithm on the use of the `bestSeen` variable, which contains the location of the best value seen so far in the matrix. This lets us know that we will never choose to return a location that looks like a peak within the current subproblem, but is adjacent to some greater value just outside the subproblem.

The result of these changes, written more rigorously, is the following proof:

> We wish to show that the algorithm will always return a peak. To that end, we wish to prove the following two statements:
>
> **1. If the peak problem is not empty, then the algorithm will always return a location.** Say that we start with a problem of size $m \times n$. Depending on whether the algorithm is splitting the rows or the columns, the recursive subproblem examined by the algorithm will have dimensions $\lfloor m/2 \rfloor \times n$, $(m - \lfloor m/2 \rfloor - 1) \times n$, $m \times \lfloor n/2 \rfloor$, or $m \times (n - \lfloor n/2 \rfloor - 1)$. Therefore, with each recursive call, either the number of rows or the number of columns strictly decreases, as long as both are greater than zero. So the algorithm either halts and returns at some point, or eventually examines a subproblem with a non-positive number of rows or columns. The only way for the number of rows or columns to become strictly negative, according to the formulas that determine the size of the subproblem, is to have $m = 0$ or $n = 0$ at some point. So if the algorithm does not returns 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 the algorithm does examine an empty subproblem. Without loss of generality, say that the algorithm split the columns just prior to this. Then at that point in the algorithm it must have examined 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 return the location), or it has a strictly better neighbor in the other column (in which case the algorithm will recurse on the subproblem with

dimensions $m \times 1$, thereby ensuring that the algorithm 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 algorithm returns a location, it will be a peak in the original problem. If the algorithm returns a location (r_1, c_1), then that location must have been a peak within some recursive subproblem. In addition, if (r_2, c_2) is the location of the best location seen during the execution of the algorithm (that is, the location stored in the variable `bestSeen`), it must be that $val(r_1, c_1) \geq val(r_2, c_2)$.

Assume, for the sake of contradiction, that (r_1, c_1) is not 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. Hence, it must be that the subproblem considered at that level includes some neighbor (r_3, c_3) of (r_1, c_1) with value $val(r_1, c_1) < val(r_3, c_3)$. 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. Therefore, (r_3, c_3) must have been examined during the progression of the algorithm. 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:

$$val(r_1, c_1) < val(r_3, c_3) \leq val(r_2, c_2) \leq val(r_1, c_1).$$

This results in 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).

Solution: The following is a counterexample for `algorithm3`:

```
problemMatrix = [
    [ 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]
]
```