
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:**

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

Your Solution:

With relatively simple functions, we can recognize the functions as members of certain asymptotic classes, and automatically order them according to class. For example, f_2 is linear in n , while f_4 is polynomial, so $f_2 = O(f_4)$. Any exponential function with base greater than one grows faster than any polynomial function, so $f_4 = O(f_3)$. Function f_1 is a logarithm times a function that asymptotically approaches zero. It therefore grows more slowly than a logarithm. Logarithms grow more slowly than linear functions, so $f_1 = O(f_2)$. The asymptotic ordering is therefore f_1, f_2, f_4, f_3 .

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

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

Your Solution:

We'll continue to map functions to asymptotic classes. Function f_1 is an elaborately disguised constant. Its growth rate is zero. Function f_2 is exponential. To see what class f_3 belongs to, we can transform it, as $\binom{n}{2} = \frac{N(N-1)}{2} = O(N^2)$. So f_3 is a second-degree polynomial. For f_4 we have $n\sqrt{n} = n^{\frac{3}{2}}$, so f_4 is a polynomial of order $\frac{3}{2}$. The asymptotic ordering of these functions is therefore f_1, f_4, f_3, f_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:

If we cannot easily map functions to classes, we can calculate $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$. If this limit is 0, then $g = O(f)$. To order f_1 and f_2 , write $\frac{f_1}{f_2} = \frac{n^{\sqrt{n}}}{2^n} = \frac{n^{\sqrt{n}}}{(2^{\sqrt{n}})^{\sqrt{n}}} = \left(\frac{n}{2^{\sqrt{n}}}\right)^{\sqrt{n}} \rightarrow 0$. So $f_1 = O(f_2)$. Comparing f_2 and f_3 , we get $\frac{f_2}{f_3} = \frac{2^n}{n^{10} \cdot 2^{n/2}} = \frac{2^{n/2}}{n^{10}} \rightarrow \infty$, so $f_3 = O(f_2)$. We have shown that both f_1 and f_3 grow more slowly than f_2 .

The limit method is harder to apply to f_2 and f_3 , so we will instead rewrite them into a common form that will make them easier to compare. Specifically, we will exponentiate both of them base 2. For f_3 we get $n^{10} \cdot 2^{n/2} = 2^{\log_2(n^{10} \cdot 2^{n/2})} = 2^{10 \cdot \log_2 n + \frac{n}{2}}$. For f_1 we get $2^{\log_2(n^{\sqrt{n}})} = 2^{\sqrt{n} \cdot \log_2 n}$. The exponent of f_1 is $O(\sqrt{n} \cdot \log_2 n)$. The exponent for f_3 is $O(n)$. Using the limit method on these functions, we get $\frac{n}{\sqrt{n} \cdot \log_2 n} = \frac{\sqrt{n}}{\log_2 n} \rightarrow \infty$, so $f_3 = O(f_1)$. Finally, $f_4(n) = \sum_{i=1}^n (i+1)$, which is bounded above by n^2 , so $f_4 = O(n^2)$. The asymptotic order is therefore f_4, f_1, f_3, f_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:

So far in lectures and the text, recurrence relations have been strictly univariate. Here, and in subsequent parts of this problem, we deal with multivariate recurrences. This introduces an additional degree of freedom that allows, for example, one input size to

halve at each iteration, but another to remain constant. The first recurrence does not utilize this additional freedom at all. It is in fact identical to $T(n) = \theta(n) + T(n/2)$. Total cost of this recurrence is $\sum_1^{\log_2 n} \frac{n}{2^i} = \theta(n)$.

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

Your Solution:

This recurrence is almost the same as part a, except here x is not halved at each iteration. Total cost of this recurrence is $\sum_1^{\log_2 n} n = \theta(n \log_2 n)$.

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

Your Solution:

This is a two-step recurrence, but by substituting the second step into the first, we can reduce it to a single step. Substituting S into the expression for $T(x, y)$ in line 2, we get $T(x, y) = \theta(x) + \theta(y/2) + T(x/2, y/2)$. So $T(n, n) = \theta(n) + T(n/2, n/2)$, the same recurrence as in part a. Therefore, this recurrence is also $\theta(n)$.

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?

Your Solution: Yes. This algorithm was covered in Lecture.

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

Your Solution: Yes. Every iteration selects a grid location with a higher value than the previous location, unless there is none, in which case the current location is a peak, and is returned. The grid is finite, so this process must terminate at a grid element with a higher value than any previously visited. This element is a peak.

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

Your Solution: No. See counter-example file, `counterexample.py`.

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

Your Solution: Yes. See proof in next problem.

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

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

Your Solution: Worst-case runtime is $\theta(n \log n)$. If n is the number of rows and columns, and $T(r, c)$ is the running time on a matrix of r rows and c columns, respectively, then the recurrence for this algorithm is $T(n, n) = \theta(n) + T(n, n/2)$. We solved this recurrence in Problem 2.

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

Your Solution: Worst-case runtime is $\theta(n^2)$. In the worst-case, the algorithm visits every cell, and there are n^2 cells in an n by n matrix.

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

Your Solution: Worst-case runtime is $\theta(n)$. On a problem with n rows and n columns, the recurrence for this problem is $T(n, n) = \theta(n/2, n/2) + T(n/2, n/2)$. We solved this recurrent in problem 2.

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

Your Solution: Worst-case runtime is $\theta(n)$. Reasoning is same as previous problem.

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 `algorithm4` will have dimensions $m \times \lfloor n/2 \rfloor$ or $m \times (n - \lfloor n/2 \rfloor - 1)$, if the split is made row-wise, or $\lfloor m/2 \rfloor \times \lfloor n \rfloor$ or $(m - \lfloor m/2 \rfloor - 1) \times n$ if the split is made column-wise. Therefore, the number of columns or rows in the problem strictly decreases with each recursive call as long as $n > 0$ or $m > 0$. So `algorithm4` either returns a location at some point, or eventually examines a subproblem with a non-positive number of columns or rows. The only way for the number of columns or rows to become strictly negative, according to the formulas that determine the size of the subproblem, is to have $n = 0$ or $m = 0$ at some point. So if `algorithm4` 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 `algorithm4` does examine an empty subproblem. Just prior to this, it must examine a subproblem of size $m \times 1$ or $m \times 2$ or $1 \times n$ or $2 \times n$. 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). The argument is exactly the same for the cases $1 \times n$ or $2 \times n$. So `algorithm4` can never recurse into an empty subproblem, and therefore `algorithm4` 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 or row r_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 or dividing row r_2 (where $|c_1 - c_2| = 1$ or $|r_1 - r_2| = 1$).

$|r_1 - r_2| = 1$), and the values must satisfy the inequality $val(r_1, c_1) < val(r_1, c_2)$, and similarly for a dividing row.

Let (r_2, c_2) be the location of the maximum value found by `algorithm4` in the dividing column or row. Assume for the rest of this argument, that it is a 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: Write your proof here.

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:

```
problemMatrix = [
    [0, 0, 0, 0, 0],
    [0, 0, 9, 10, 0],
    [0, 0, 8, 0, 0],
    [0, 0, 7, 0, 0],
    [0, 0, 0, 0, 0]
]
```