# Problem Set 1

**All parts are due on September 13, 2018 at 11PM**. Please write your solutions in the LaTeX and Python templates provided. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct. Solutions should be submitted on the course website, and any code should be submitted for automated checking on `alg.mit.edu`.

---

**Problem 1-1.** [20 points] **Asymptotic behavior of functions**

For each of the following sets of five functions, order them so that if $f_a$ appears before $f_b$ in your sequence, then $f_a = O(f_b)$. If $f_a = O(f_b)$ and $f_b = O(f_a)$ (meaning $f_a$ and $f_b$ could appear in either order), indicate this by enclosing $f_a$ and $f_b$ in a set with curly braces. For example, if the functions are:

$$f_1 = n, \qquad f_2 = \sqrt{n}, \qquad f_3 = n + \sqrt{n},$$

the correct answers are $(f_2, \{f_1, f_3\})$ or $(f_2, \{f_3, f_1\})$.

**Note:** Recall that $a^{b^c}$ means $a^{(b^c)}$, not $(a^b)^c$, and that $\log$ means $\log_2$ unless a different base is specified explicitly. Stirling's approximation may help for part **d)**.

| **a)** | **b)** | **c)** | **d)** |
|---|---|---|---|
| $f_1 = 20n + 18$ | $f_1 = n^{2\log n}$ | $f_1 = 2^{n^3}$ | $f_1 = (2n)!$ |
| $f_2 = 20n \cdot 18$ | $f_2 = 2^{2^{\log n}}$ | $f_2 = 2^{(n+1)^3}$ | $f_2 = \binom{2n}{n}$ |
| $f_3 = 20n^{18}$ | $f_3 = 2^{(\log n)^2}$ | $f_3 = n^{n^2}$ | $f_3 = 2^{2n}$ |
| $f_4 = \log_{20}(n^{18})$ | $f_4 = n^{\log n}$ | $f_4 = 2^{2^{n+1}}$ | $f_4 = (2n)^n$ |
| $f_5 = (\log_{18}(n))^{20}$ | $f_5 = 2^{\log(\log n)}$ | $f_5 = 3^{2^n}$ | $f_5 = n^{2n}$ |

**Solution:**

a. $(f_4, f_5, \{f_1, f_2\}, f_3)$. This order follows from knowing that $\log_a n = O(\log n)$ for any $a > 1$, $(\log n)^a$ grows slower than $n^b$ for all $0 < a$ and $0 < b$, and $n^a$ grows slower than $n^b$ for all $0 < a < b$.

b. $(f_5, \{f_3, f_4\}, f_1, f_2)$. This order follows from elementary exponentiation and logarithm rules. Observe that $f_3$ and $f_4$ are the same function, while $f_1 = (f_3)^2$.

c. $(f_3, f_1, f_2, f_5, f_4)$. This order follows after converting all the exponent bases to 2 and remembering that big-$O$ is much more sensitive to changes in exponents. For example, the ratio $f_2/f_1 = 2^{3n^2+3n+1}$ approaches $\infty$ as $n \to \infty$, so $f_2 \neq O(f_1)$. Similarly, $f_4 = 2^{2 \cdot 2^n}$ and $f_5 = 2^{(\log 3) \cdot 2^n}$, and constant factors matter inside exponents, since $f_4/f_5 = 2^{(2-\log 3)2^n} \to \infty$.

d. $(f_2, f_3, f_4, f_1, f_5)$. This order follows from the definition of the binomial coefficient and Stirling's approximation. The trickiest one is $f_2 = \Theta(2^{2n}/\sqrt{n})$ (by repeated use of Stirling), which grows slower than $f_3 = 2^{2n}$. Similarly, $f_1 = \Theta(\left(\frac{2n}{e}\right)^{2n} \cdot \sqrt{n})$ (by Stirling), which lies between $f_4 = (2n)^n$ and $f_5 = n^{2n}$.

**Rubric:**

- 5 points per set for a correct order
- $-1$ point per inversion
- $-1$ point per grouping mistake, e.g., $(\{f_1, f_2, f_3\})$ instead of $(f_2, f_1, f_3)$ is $-2$ points because they differ by two splits.
- 0 points minimum

**Problem 1-2.** [30 points] **Recurrences**

**(a) 3-Way Max Subarray Sum:** In Lecture 2 we saw Divide-and-Conquer algorithm for the Max Subarray Sum problem based on the insight that, for an input array $A$ of length $n$, the best (**consecutive**) subarray will either:

- lie within $A$'s first half,
- lie within $A$'s second half, or
- use at least one element from each half.

Recall that we have an $O(n)$ strategy for the last case: scan forwards to find the largest subarray beginning in the middle, and scan backward to find the largest subarray ending in the middle.

Derek Amayn thinks that splitting $A$ into three pieces instead of two will speed things up! Derek's modified Divide-and-Conquer algorithm observes correctly that the best subarray will either:

- lie within the first two-thirds of $A$,
- lie within the last two-thirds of $A$, or
- use at least one element from each third.

You may assume $n$ is a power of 3, for simplicity.

i. [5 points] **Write** a recurrence for the amount of work Derek's strategy will take.
**Solution:**

$$T(n) = 2T\left(\frac{2n}{3}\right) + O(n).$$

**Rubric:**

- 5 points for correct recurrence

ii. [5 points] **Solve** the recurrence using the Master Theorem. Is it $O(n \log n)$?
**Solution:** With $a = 2$ and $b = 3/2$, we have $\log_b a = \log_{3/2} 2 \approx 1.7095$. Because $n^1 = O(n^{(\log_b a)-0.7})$, Case 1 of the Master Theorem shows that $T(n) = O(n^{\log_{3/2} 2})$. This is not $O(n \log n)$, so Derek's algorithm is asymptotically worse.
**Rubric:**
- 4 points for correct solution to recurrence in part i, even if part i is incorrect
- 1 point for correct comparison to $O(n \log n)$

**(b) Solving recurrences:** Derive solutions to the following recurrences in two ways: draw a recursion tree **and** apply the Master Theorem. Assume $T(1) = O(1)$.

i. [5 points] $T(n) = 2\,T(\frac{n}{4}) + O(\log n)$
**Solution:** $T(n) = \Theta(\sqrt{n})$ by Case 1 of the Master Theorem, since $f(n) = O(n^{\frac{1}{2}-\varepsilon})$ for any $\varepsilon < \frac{1}{2}$.

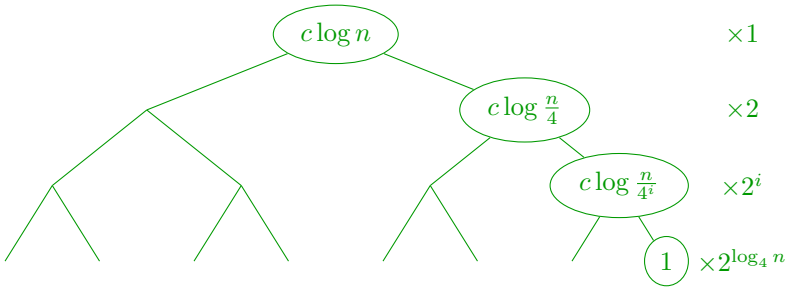
Drawing a tree, there are $2^i$ nodes at depth $i$ each doing $\log(n/4^i)$ work, so the total work (up to constant factors) is at most

$$\sum_{i=0}^{\log_4 n} 2^i(\log n - 2i).$$

It's tempting to split this into $(\sum 2^i \log n) - (\sum 2i \cdot 2^i)$, but this gets tricky: both sums are $\Theta(2^{\log_4 n} \log n) = \Theta(\sqrt{n} \log n)$, and we'd have to compute these sums rather accurately to verify that their higher-order terms cancel when they're subtracted. This can indeed be done, with the formula $\sum_{i=0}^{k} i \cdot 2^i = (k-1)2^{k+1} + 2$, but here's a simpler method.
Because $\log x < c \cdot x^{1/4}$ for every $x > 1$ (and some constant $c$), the amount of work may be upper bounded (up to constant factors) by

$$\sum_{i=0}^{\log_4 n} 2^i \cdot \left(\frac{n}{4^i}\right)^{1/4} = \sum_{i=0}^{\log_4 n} 2^i \cdot n^{1/4} \cdot 2^{-i/2} = n^{1/4} \cdot \sum_{i=0}^{\log_4 n} 2^{i/2}.$$

This geometric series is $O(n^{1/4} \cdot 2^{(\log_4 n)/2}) = O(\sqrt{n})$. (Note: Why use $n^{1/4}$? Because $n^{\log_4 n} = n^{1/2}$ is the critical point where Case 2 of the Master theorem kicks in, so anything with smaller exponent works.)
For the lower bound, there are $\Theta(\sqrt{n})$ nodes in the tree, so the work done is at least $\Omega(\sqrt{n})$.



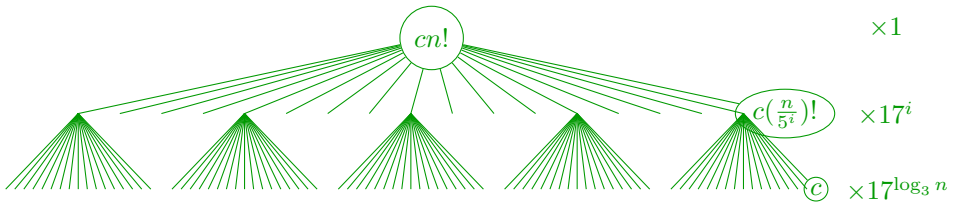
ii. [5 points] $T(n) = 17\,T(\frac{n}{3}) + \Theta(n!)$

**Solution:** $T(n) = \Theta(n!)$ by Case 3 of the Master Theorem, since $f(n) = \Theta(\sqrt{n} \times (\frac{n}{e})^n) = \Omega(n^{\log_3 17+\varepsilon})$

Drawing a tree, the total work at depth $i$ is (# nodes $\times$ work) $= O(17^i \times (\frac{n}{3^i})!)$. Observe that this quantity decreases very quickly as $i$ increases (faster than a geometric sequence, for example), so the running time,

$$O\left(\sum_{i=0}^{\log_3 n} 17^i \times \left(\frac{n}{3^i}\right)!\right),$$

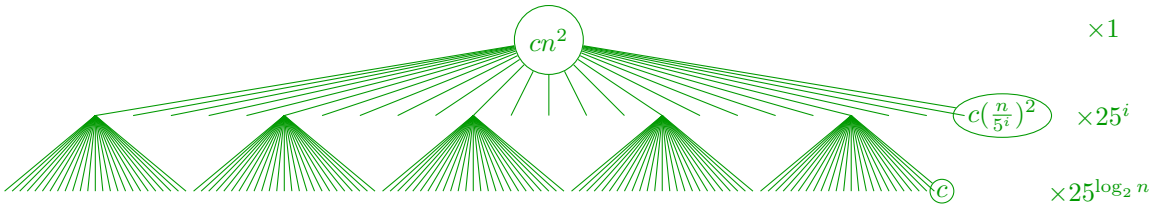
is dominated by its first term, $O(n!)$.




iii. [5 points] $T(n) = 25\,T(\frac{n}{5}) + n^2 + 2n + \log n$

**Solution:** Notice that $n^2 + 2n + \log n = \Theta(n^2)$, so $T(n) = \Theta(n^2 \log n)$ by Case 2 of the Master Theorem, since $f(n) = O(n^{\log_5 25}) = \Theta(n^2)$.

Drawing a tree, the total work at depth $i$ is (# nodes $\times$ work) = $25^i \times (\frac{n}{5^i})^2 = n^2$, so the total work is $\sum_{i=0}^{\log_5 n} n^2 = \Theta(n^2 \log n)$. (In the figure, the branching factor of the tree is 25.)



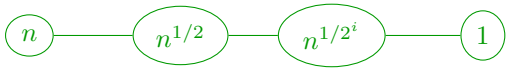

iv. [5 points] $T(n) = T(\sqrt{n}) + O(1)$ (Tree only)

**Solution:** $T(n) = O(\log \log n)$.

The 'tree' is simply a chain where the node at depth $i$ does $O(1)$ work, so the running time is simply $O(d)$, where $d$ is the depth of the tree. How do we find the depth? The input at depth $i$ is $\sqrt{\sqrt{\cdots\sqrt{n}}} = n^{1/2^i}$, and the tree bottoms out when this quantity becomes constant—say, less than 2:

$$n^{1/2^i} < 2 \iff 2^{2^i} > n \iff 2^i > \log n \iff i > \log \log n.$$

So the tree has depth $\log\log n$, and $T(n) = O(\log\log n)$.
A different strategy is to substitute $n = 2^m$ and $S(m) = T(2^m)$; the recurrence becomes $T(2^m) = T(2^{m/2}) + O(1)$, i.e., $S(m) = S(m/2) + O(1)$. This has solution $S(m) = O(\log m)$, meaning $T(n) = S(\log n) = O(\log\log n)$.




**Rubric:**

- 1 point per tree drawing
- 2 points per correct solution via tree, 4 points for part (iv)
- 2 points per correct solution via Master Theorem
- $-1$ point globally (on first instance only) of writing $O(\cdots)$ in every node in a recurrence tree, instead of using $c\cdots$ (need to use a single constant for all nodes)

**Problem 1-3.** [50 points] **Searching a Sorted 2D Array**

A two-dimensional $n \times m$ matrix $A$ is **two-dimensionally sorted** if every row and column is sorted in increasing order rightward and downward: in other words,

$$A[y][x] \geq A[y][x-1] \qquad \text{and} \qquad A[y][x] \geq A[y-1][x]$$

whenever those indices are in range. For example, the matrix below is two-dimensionally sorted.

```
A = [[  1,  3,  7,  8, 16 ],
     [  2,  3,  8, 13, 17 ],
     [  4,  5, 14, 15, 21 ],
     [  8, 17, 17, 21, 23 ],
     [ 11, 17, 24, 26, 30 ]]
```

The SORTED2DSEARCH problem asks, given a two-dimensionally sorted array $A$ and integer $v$, does the value $v$ appear as an element of $A$?

**(a)** [5 points] Checking every element of the array against $v$ solves this problem in $O(nm)$ time. Describe an algorithm that uses binary search to solve SORTED2DSEARCH in $O(n \log m)$ time. (In the following parts, you'll design and implement a linear time algorithm.)

**Solution:** Use **binary search** to look for $v$ in each **row** of $A$. Each row is sorted (by assumption), so binary search will correctly locate $v$ if it is present, meaning our algorithm is correct. Each binary search takes $O(\log m)$ time to search a row of length $m$, and there are $n$ rows, so the overall running time is $O(n \log m)$.

**Rubric:**

- 4 points for a correct algorithm
- 1 point for arguing running time
- Partial credit may be awarded

**(b)** [5 points] Consider element $s = A[n-1][0]$ in the bottom left corner of $A$. If $v > s$, how does that limit possible locations of $v$? What if $v < s$? Show that if $v \neq s$, the number of elements from $A$ that could be equal to $v$ reduces by at least $\min(n, m)$.

**Solution:** Because column 0 of $A$ is sorted in increasing order, element $s = A[n-1][0]$ is the largest element in this column (possibly with ties). When $v > s$, it follows that $v$ is strictly greater than **every** element in this column and therefore cannot appear in this column. Similarly, when $v < s$, $v$ is strictly smaller than every element in row $n-1$, because this row is sorted in increasing order. So when $v \neq s$, we can eliminate either the $n$ elements of column 0 (if $v > s$) or the $m$ elements of row $n-1$ (if $v < s$), and both $n$ and $m$ are at least $\min(n, m)$, as desired.

**Rubric:**

- 5 points for a correct argument
- Partial credit may be awarded

**(c)** [15 points] Describe an $O(n+m)$ time algorithm to solve SORTED2DSEARCH. (For any algorithm you describe in this class, you should **argue that it is correct**, and **argue its running time**.)

**Solution:** Start at the bottom-left corner, $(x, y) = (0, n-1)$, and iteratively move across the grid as follows: if $A[y][x] < v$ then move to $(x+1, y)$; if $A[y][x] > v$ then move to $(x, y-1)$; and if $A[y][x] = v$ then return True (because $v$ is at indices $(x, y)$). If we ever move outside the bounds of the grid (i.e., $x \geq m$ or $y < 0$), conclude that $v$ is not present in $A$ and return False.

To prove correctness, we'll argue by induction on the number of iterations that at each stage, if $v$ lies in $A$ then it must lie in the rectangular subarray with lower-left corner $(x, y)$ (inclusive) and the same upper-right corner as $A$ itself; call this rectangle $R(x, y)$, and note that its dimensions are $(m-x) \times (y+1)$. Initially, $R(0, n-1)$ is the full array $A$, proving the base case. Part (b) is the heart of the inductive step: if $A[y][x] < v$ then $v$ cannot appear in $R(x, y)$ in its leftmost column, and by the inductive hypothesis $v$ does not occur outside $R(x, y)$, so incrementing $x$ to $x+1$ preserves our claimed property. The situation is analogous when $A[y][x] > v$. If we ever arrive at $x \geq m$ or $y < 0$ then $R(x, y)$ is empty, so $v$ cannot appear in the array, as claimed.

To analyze the running time, let's look at the **sum** of the dimensions of $R(x, y)$, namely $M = (m-x) + (y+1)$. Each iteration returns immediately or decreases $M$ by exactly 1. The value $M$ starts at $m+n$, and because $(m-x)$ and $(y+1)$ stay nonnegative throughout the algorithm, $M$ never becomes negative. Thus, monovari-

ant $M$ shows that our loop has at most $m + n$ iterations. Each iteration does $O(1)$ work, so the running time is $O(m + n)$, as claimed. **Rubric:**

- 7 points for a correct algorithm
- 5 points for arguing correctness
- 3 points for arguing running time
- Partial credit may be awarded

**(d)** [25 points] Write a Python function `search_sorted_2D_array` that implements your algorithm. You can download a code template containing some test cases from the website. Submit your code online at `alg.mit.edu`.

**Solution:**

In this implementation, region $R(x, y)$ is encoded by upper-left corner $(x_0, y_0)$ and lower-right corner $(x_1, y_1)$.

```
1  def search_sorted_2D_array(A, v):
2
3      def search_subarray(A, v, x0, y0, x1, y1):
4          if (x1 < x0) or (y1 < y0):
5              return None              # Base Case, no elements
6          a = A[y1][x0]
7          if a > v:                    # All of last row > v
8              return search_subarray(A, v, x0, y0, x1, y1 - 1)
9          if a < v:                    # All of first column < v
10             return search_subarray(A, v, x0 + 1, y0, x1, y1)
11         return (x0, y1)              # Found!
12
13     return search_subarray(A, v, 0, 0, len(A[0]) - 1, len(A) - 1)
```

**Rubric:**

- This part is automatically graded at `alg.mit.edu`.