Problem Set 1

All parts are due on February 15, 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 `py.mit.edu/6.006`.

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

For each of the following sets of five functions, write down a permutation of the numbers 1 to 5 such that, if a occurs before b in the sequence, then $f_a = O(f_b)$. If $f_a = O(f_b)$ and $f_b = O(f_a)$, then a and b may appear in either order.

a)	**b)**	**c)**	**d)**
$f_1 = 2018n$	$f_1 = (\log\log n)^5$	$f_1 = 9^n$	$f_1 = 4^n$
$f_2 = n\log(n^{2018})$	$f_2 = \log(5^{n^5})$	$f_2 = 3^{3n}$	$f_2 = 4(n!)$
$f_3 = 2018^n$	$f_3 = (\log n)^{\log(n^5)}$	$f_3 = 2^{3^n}$	$f_3 = n^4$
$f_4 = (\log n)^{2018}$	$f_4 = \log((\log n)^5)$	$f_4 = 3^{3^{n+1}}$	$f_4 = \binom{n}{4}$
$f_5 = n^{2018}$	$f_5 = n^{5\log n}$	$f_5 = 3^{n^2}$	$f_5 = \binom{n}{n/4}$

Solution:

a. $(4, 1, 2, 5, 3)$ This order follows from knowing that $(\log n)^a$ 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. $(4, 1, 2, 3, 5)$ This order follows from elementary exponentiation and logarithm rules.

c. $(1, 2, 5, 3, 4)$ This order follows after converting all the exponent bases to 3, specifically for $f_1 = 3^{2n}$ and $f_3 = 3^{(\log_3 2)3^n}$ (constant factors matter inside the exponent).

d. $(\{3, 4\}, 5, 1, 2)$ This order follows from the definition of the binomial coefficient and Stirling's approximation. The tricky one is f_5 which is $O(((4/3)^{3/4})^n/\sqrt{n})$ by repeated application of Stirling and algebra. A weaker bound could also be used with Stirling applied once: $\binom{n}{k} \le n^k/k! < (ne/k)^k$ yielding $O((\sqrt[4]{4e})^n)$.

Rubric:

- 5 points per set for a correct order
- -1 point per inversion
- 0 points minimum

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

(a) [5 points] **Unbalanced Search:** Normal binary search finds an element in a sorted array of n elements in $O(\log n)$ time. In one application, we decide to preferentially search near the beginning of the array, so we modify the algorithm to repeatedly divide the search space into ratio 1:c instead of 1:1, for some constant $c > 2$. Write down and solve a recurrence relation to upper bound the running time of this unbalanced binary search. Does the modified binary search still run in $O(\log n)$ time?

Solution:

$$T(n) = \max\left(T\left(\frac{n}{c+1}\right), T\left(\frac{c}{c+1}n\right)\right) + O(1) < T\left(\frac{c}{c+1}n\right) + O(1) = O(\log n)$$

This recurrence can be solved by master theorem case 2, since $f(n) = \Theta(n^{\log_{c/(c+1)} 1}) = \Theta(n^0) = \Theta(1)$. The tree for this recurrence does not branch; just like normal binary search, it is simply a chain containing a logarithmic number of sub-problems, each requiring constant time to solve.

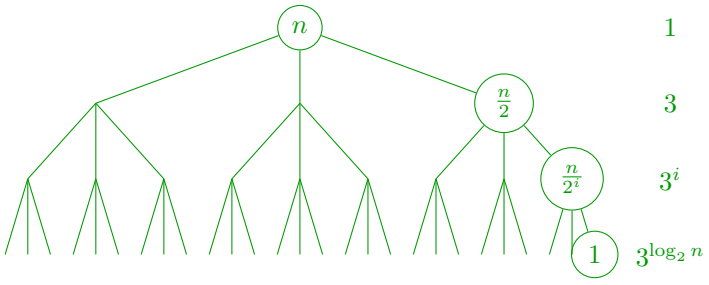
Rubric:

- 2 points for a correct recurrence relation
- 3 points for solution to recurrence, asserting remains $O(\log n)$

(b) [25 points] **Solving recurrences:** Derive solutions to the following recurrences in two ways: draw a recursion tree **and** apply the master theorem. Assume $T(1) = O(1)$.

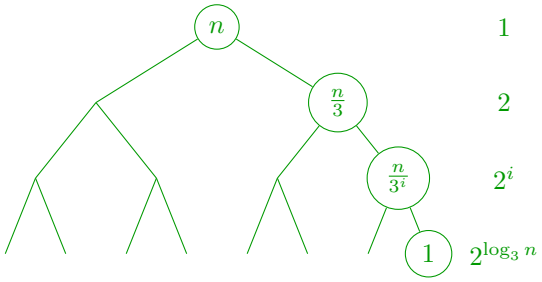
a. $T(n) = 3T(\frac{n}{2}) + O(n)$
Solution: $T(n) = O(n^{\log_2 3})$ by master theorem case 1, since $f(n) = O(n^{\log_2 3 - \varepsilon})$ for e.g. $\varepsilon < (\log_2 3) - 1 \approx 0.585$.

Drawing a tree, the total work at depth i is (# nodes $\times$ work) = $3^i \times \frac{n}{2^i} = n(\frac{3}{2})^i$. So total work is $n \sum_{i=0}^{\log_2 n} \left(\frac{3}{2}\right)^i = nO((\frac{3}{2})^{\log_2 n} - 1) = nO(n^{(\log_2 3)-1}) = O(n^{\log_2 3})$.



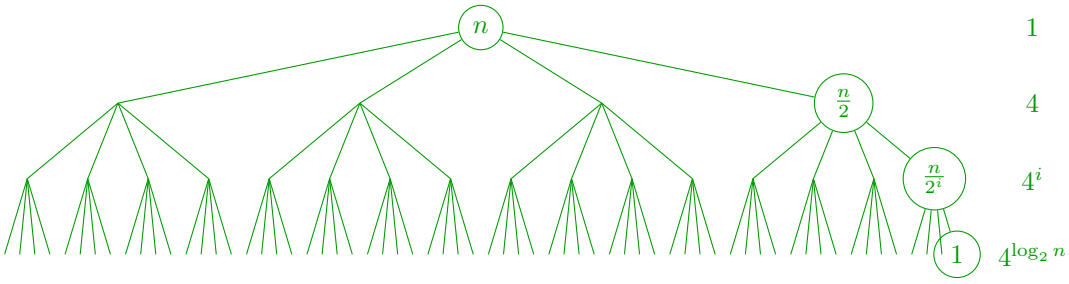
b. $T(n) = 2T(\frac{n}{3}) + O(n)$
Solution: $T(n) = O(n)$ by master theorem case 3, since $f(n) = \Omega(n^{\log_3 2 + \varepsilon})$ for e.g. $\varepsilon < 1 - (\log_2 3) \approx 0.369$ and $2(n/3) < c(n)$ for e.g. $2/3 < c < 1$.

Drawing a tree, the total work at depth i is (# nodes $\times$ work) = $2^i \times \frac{n}{3^i} = n(\frac{2}{3})^i$. So total work is $n\sum_{i=0}^{\log_3 n}\left(\frac{2}{3}\right)^i = nO(1-(\frac{2}{3})^{\log_3 n}) = O(n)$.



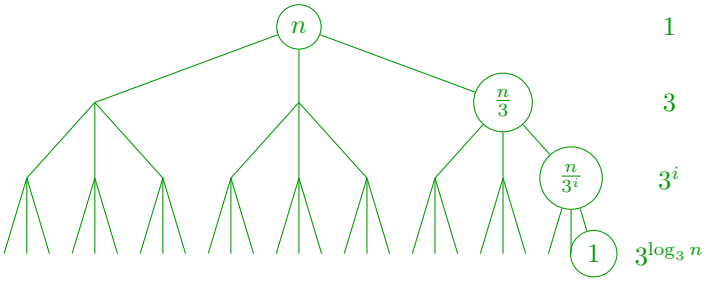
c. $T(n) = 4T(\frac{n}{2}) + \Theta(n^2)$
Solution: $T(n) = \Theta(n^2 \log n)$ by master theorem case 2, since $f(n) = O(n^{\log_2 4}) = \Theta(n^2)$.

Drawing a tree, the total work at depth i is (# nodes $\times$ work) = $4^i \times (\frac{n}{2^i})^2 = n^2$. So total work is $\sum_{i=0}^{\log_2 n} n^2 = \Theta(n^2 \log n)$.



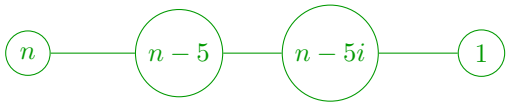
d. $T(n) = 3T(\frac{n}{3}) + \Theta(n)$
Solution: $T(n) = \Theta(n \log n)$ by master theorem case 2, since $f(n) = O(n^{\log_3 3}) = \Theta(n)$.

Drawing a tree, the total work at depth i is (# nodes $\times$ work) = $3^i \times (\frac{n}{3^i}) = n$. So total work is $\sum_{i=0}^{\log_3 n} n = \Theta(n \log n)$.



e. $T(n) = T(n-5) + O(\log n)$ (Tree only)
Solution: The 'tree' is simply a chain where the node at height i does $O(\log 5i)$ work. So total work is $\sum_{i=1}^{n/5} \log 5i = \log\left(\prod_{i=1}^{n/5} 5i\right) = \log(5^{n/5}(n/5)!) =$

$O(n \log n)$, where the last relation can be derived via Stirling's approximation.



Rubric:

- 1 points per tree drawing
- 2 points per correct solution via tree, 4 points for part (e)
- 2 points per correct solution via master theorem

Problem 1-3. [50 points] **2D Peak Finding**

A **peak** in a two-dimensional $n \times n$ array of integers is an element of the array that is greater than or equal to each of its eight adjacent neighbors (some of which may not exist if the element is on the boundary). In this problem, you will design and implement an algorithm to find such a peak.

(a) [5 points] Prove that every nonempty two-dimensional array contains a peak. **Please review the 'Writing Proofs' handout on Stellar for guidelines on writing proofs.**

Solution: Because the integers are **well-ordered**, there exists a maximal integer p among the $n \times n$ integers that is at least as large as all the others. In particular, p is at least as large as its eight neighbors, so p is a peak.

Rubric:

- 5 points for complete proof
- Partial credit may be awarded

(b) [10 points] Consider a two-dimensional array A and rectangular sub-array $B \subseteq A$. Prove that if sub-array B contains an element that is greater than or equal to every element of A directly outside and adjacent to B, then B contains a peak.

Solution: Again, because the integers are **well-ordered**, there exists a maximal integer p among the integers in B that is at least as large as all the other integers in B. Then the only way for p to **not** be a peak in A is if it were adjacent to a larger integer outside of B. Since B contains an integer (witness) that is greater than or equal to every element of A directly outside and adjacent to B, then p, being at least as large as the witness, will also be greater than or equal to the elements outside and adjacent to B. Thus p is a peak.

Rubric:

- 10 points for complete proof
- Partial credit may be awarded

(c) [10 points] Given an $n \times m$ array with $n \geq m$, divide it by splitting the longer dimension, n, in half. Show how to identify a witness in one of the halves to certify that it contains a peak in $O(m)$ time.

Solution: Consider the set of $O(m)$ elements comprising the two rows or columns of elements adjacent to the line separating the two halves of the array. A maximal integer p among those elements will again be at least as large as any array element outside and adjacent to the half containing p. Thus p a witness certifying a peak in the half containing p.

Rubric:

- 10 points for complete argument
- Partial credit may be awarded

(d) [5 points] Our algorithm will divide the array in half by splitting its longer dimension, identify a witness in one of the halves, and then recursively search within the half containing the witness. Write and solve a recurrence relation based on this algorithm. How does the running time of this algorithm relate to input size, i.e., is the algorithm sub-linear, linear, or super-linear?

Solution: The input size of this problem is $\Theta(nm)$. A recurrence relation for this algorithm is $T(nm) = T(nm/2) + O(\min(n, m))$. Here, we note that $\min(n, m) = O(\sqrt{nm}) = O(\max(n, m))$, so by letting $N = nm$, we get the recurrence relation $T(N) = T(N/2) + O(\sqrt{N})$. This recurrence relation evaluates to $T(nm) = T(N) = O(\sqrt{N}) = O(\max(n, m))$ by case 3 of the master theorem, since $f(N) = \Omega(N^{\log_2 1}) = \Omega(1)$. When $m = O(n)$, this algorithm is $O(\sqrt{nm})$, which is **sub-linear** in the input $O(nm)$.

Rubric:

- 2 points for recurrence relation
- 2 points for derivation of solution
- 1 point for stating the algorithm is sub-linear

(e) [20 points] Write a Python function `find_peak_2D` that implements your algorithm. You can download a code template containing some test cases from the website. Submit your code online at `py.mit.edu/6.006`.

```
def find_peak_2D(A, r = None, w = None):
    '''
    Find a peak in a two dimensional array.
    Input: 2D integer array A, subarray indices r, witness w
    '''
    if r is None:
        r = (0, 0, len(A[0]) - 1, len(A) - 1)
    px, py, qx, qy = r  # A[py][px] upper left, A[qy][qx] lower right
    if w is None:
        w = (0, 0)
    wx, wy = w          # A[wy][wx] witness
    ##################
    # YOUR CODE HERE #
    ##################
    return (0, 0)
```

Solution:

```
def find_peak_2D(A, r = None, w = None):
    '''
    Find a peak in a two dimensional array.
    Input: 2D integer array A, subarray indices r, witness w
    '''
    if r is None:
        r = (0, 0, len(A[0]) - 1, len(A) - 1)
    px, py, qx, qy = r
    if w is None:
        w = (0, 0)
    wx, wy = w
    if (px == qx) and (py == qy):
        return (px, py)          # base case
    if qx - px > qy - py:        # larger dimension in x
        c = (px + qx + 1) // 2   # center
        for x in [c - 1, c]:     # find new witness
            for y in range(py, qy + 1):
                if A[y][x] > A[wy][wx]:
                    wx, wy = x, y
        if wx < c:               # new witness in right half
            qx = c - 1
        else:                    # new witness in left half
            px = c
    else:                        # larger dimension in y
        c = (py + qy + 1) // 2   # center
        for y in [c - 1, c]:     # find new witness
            for x in range(px, qx + 1):
                if A[y][x] > A[wy][wx]:
                    wx, wy = x, y
        if wy < c:               # new witness in bottom half
            qy = c - 1
        else:                    # new witness in top half
            py = c
    return find_peak_2D(A, (px, py, qx, qy), (wx, wy))
```

Rubric:

- This part automatically graded at `py.mit.edu/6.006`.