Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — February 8, 2019
Instructors: Jason Ku, Vinod Vaikuntanathan, and Virginia Williams — Problem Set 1

Problem Set 1

All parts are due on February 15, 2019 at 6PM. 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)**
$f_1 = 6006^n$	$f_1 = n^{3\log n}$	$f_1 = 3^{2^n}$	$f_1 = 2^n$
$f_2 = (\log n)^{6006}$	$f_2 = (\log\log n)^3$	$f_2 = 2^{2^{n+1}}$	$f_2 = \binom{n}{2}$
$f_3 = 6006n$	$f_3 = \log((\log n)^3)$	$f_3 = 2^{2n}$	$f_3 = n^2$
$f_4 = n^{6006}$	$f_4 = \log(3^{n^3})$	$f_4 = 8^n$	$f_4 = \binom{n}{n/2}$
$f_5 = n\log(n^{6006})$	$f_5 = (\log n)^{\log(n^3)}$	$f_5 = 2^{n^3}$	$f_5 = 2(n!)$

Solution:

a. $(f_2, f_3, f_5, f_4, f_1)$. This order follows from knowing that $\log$ grows slower than n^a for all $0 < a$, and n^a grows slower than n^b for $0 < a < b$.

b. $(f_3, f_2, f_4, f_5, f_1)$. This order follows from elementary exponentiation and logarithm rules, converting some of the exponent bases to 2 and remembering that big-O is much more sensitive to changes in exponents.

c. $(f_3, f_4, f_5, f_1, f_2)$. This order follows after converting all the exponent bases to 2 and again remembering that big-O is much more sensitive to changes in exponents.

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 $f_4 = \Theta(2^n/\sqrt{n})$ (by repeated use of Stirling), which grows slower than f_1.

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. [10 points] ***k*-Way Merge Sort**

Merge sort is a divide-and-conquer algorithm based on the idea that merging two sorted subarrays into one large sorted array is possible in linear time. Recall that merge sort divides unsorted input into two equally-sized subarrays, recursively sorts those subarrays, and then merges them.

An excited computer scientist named Talan Uring is wondering whether he can modify the merge sort algorithm to sort even faster! In particular, he wants to k-way merge sort. A k-way merge sort divides the input into k equally-sized subarrays. These k subarrays are then recursively sorted and merged into a new array. To merge them, maintain a pointer for each subarray. Each pointer starts at the beginning of its subarray. The algorithm then determines the minimum of the elements pointed to by the k pointers and copies it to the next position in the new array. The pointer to the minimum element is then moved forward by one. This procedure continues until all the pointers are at the end of their subarrays. Note that a 2-way merge sort corresponds to standard merge sort.

(a) [3 points] Suppose Talan sets k to 5. Write a recurrence for the 5-way merge sort algorithm.

Solution: We perform five recursive calls and then spend $\Theta(n)$ time merging the five subarrays. The recurrence is thus,

$$T(n) = 5\,T\left(\frac{n}{5}\right) + \Theta(n).$$

Rubric:

- 3 points for a correct recurrence
- Partial credit may be awarded

(b) [4 points] Solve the recurrence from (a) using the Master Theorem. How does the running time of this approach compare to standard merge sort?

Solution: $T(n) = \Theta(n \log n)$ by case 2 of the Master Theorem, since $f(n) = n^{\log_5(5)} = \Theta(n)$.

Rubric:

- 3 points for a correct solution to recurrence in part (a), even if part (a) is incorrect
- 1 point for correct comparison to $O(n \log n)$

(c) [3 points] Talan now considers setting k to a **non-constant** value, specifically $k = \sqrt{n}$ where an array of size n is divided into $\sqrt{n}$ subarrays to merge. Write a recurrence for $\sqrt{n}$-way merge sort. You do **not** have to solve this recurrence.

Solution: We perform $\sqrt{n}$ recursive calls, each on a $\sqrt{n}$-sized subarray. Merging these subarrays does not take $O(n)$ time as the number of subarrays is not constant, so finding the minimum of pointers cannot be done in constant time. Merging then takes $\Theta(n\sqrt{n})$ time. The recurrence overall is then:

$$T(n) = \sqrt{n}\,T(\sqrt{n}) + \Theta(n\sqrt{n}).$$

The solution to this recurrence is $T(n) = \Theta(n\sqrt{n})$ which can be verified by substitution. Note a heap can be used to speed up the merge to $\Theta(n \log n)$, which would yield the recurrence:

$$T(n) = \sqrt{n}\,T(\sqrt{n}) + \Theta(n \log n).$$

The solution to this recurrence is $T(n) = \Theta(n \log n)$ which can also be verified by substitution. Students do **not** need to solve their recurrence.

Rubric:

- 3 points for a correct recurrence
- Partial credit may be awarded

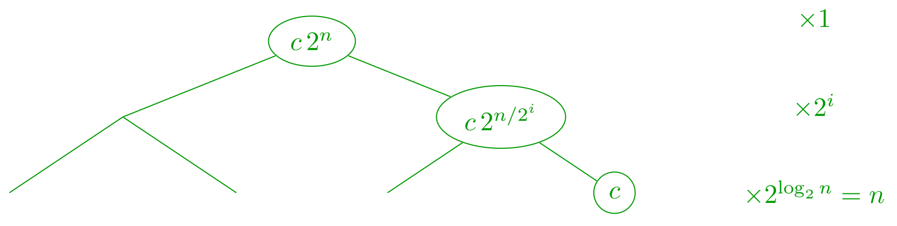
Problem 1-3. [20 points] **Solving recurrences**

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

(a) [5 points] $T(n) = 2\,T(\frac{n}{2}) + \Theta(2^n)$

Solution: $T(n) = \Theta(2^n)$ by case 3 of the Master Theorem, since $f(n) = 2^n = \Omega(n^{1+\epsilon})$ for some (in fact, every) constant $\epsilon > 0$.

Drawing a tree, there are 2^i nodes at depth i each doing $2^{n/2^i}$ work, so the total work at depth i is $2^i\,2^{n/2^i} = 2^{n/2^i+i}$. Note that this quantity decreases very quickly as i increases, so this running time is dominated by the first term, $O(2^n)$.

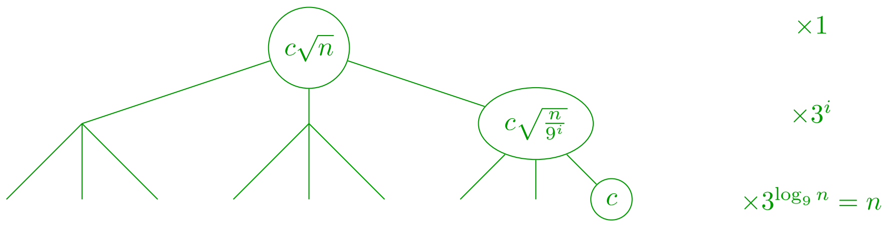


(b) [5 points] $T(n) = 3\,T(\frac{n}{9}) + O(\sqrt{n})$

Solution: $T(n) = O(\sqrt{n}\log n)$ by case 2 of the Master Theorem, since $f(n) = O(\sqrt{n}) = O(n^{\log_9 3})$.

Drawing a tree, there are 3^i nodes at depth i, each doing at most $c\sqrt{\frac{n}{9^i}}$ work for some constant c. The total work is then upper-bounded by:

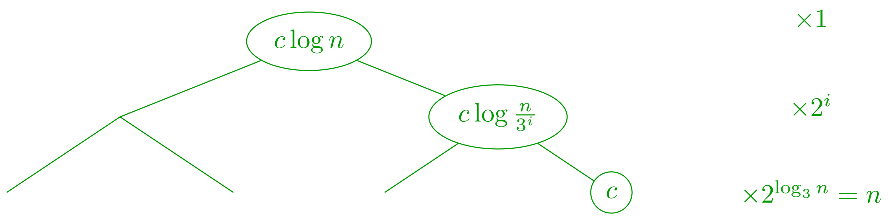
$$\sum_{i=0}^{\log_9 n} c\,3^i \left(\sqrt{\frac{n}{9^i}}\right) = c\sqrt{n}\sum_{i=0}^{\log_9 n} 1 = c\sqrt{n}\log_9 n = O(\sqrt{n}\log n).$$



(c) [5 points] $T(n) = 2\,T(\frac{n}{3}) + \log_7 n + 6$

Solution: Notice that $\log_7 n + 6 = \Theta(\log n)$. So $T(n) = O(n^{\log_3 2})$ by case 1 of the Master Theorem, since $f(n) = O(n^{\log_3 2 - \epsilon})$ for some $\epsilon > 0$ (in particular, all $\epsilon \in (0, \log_3 2)$).

Drawing a tree, the total work at depth i is (# nodes $\times$ work) = $2^i \times \log\frac{n}{3^i}$. Summing this over the different depths, we see that the total work is dominated by the work at the leaves, so the total work is $O(2^{\log_3 n}) = O(n^{\log_3 2})$.



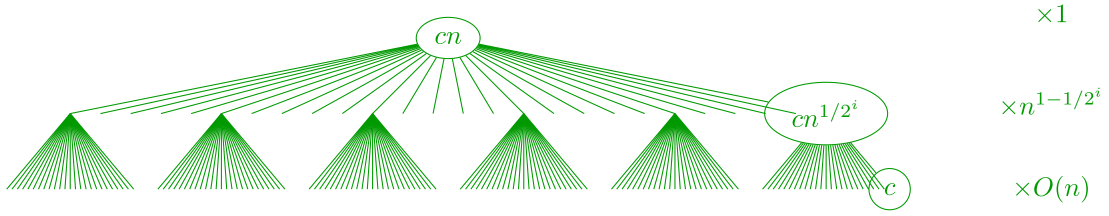
(d) [5 points] $T(n) = \sqrt{n}\,T(\sqrt{n}) + O(n)$ Assume $T(2) = \Theta(1)$ and only draw a recursion tree.

Solution:

Problem size reduces by a power of $1/2$ at each level, so problem size at depth i is $n^{1/2^i}$. The work is linear in problem size, so a node at depth i does $O(n^{1/2^i})$ work. The number of nodes $N(i)$ at depth i is $n^{1/2^i}N(i-1)$ where $N(0) = 1$, i.e., $N(i) = \prod_{k=1}^{i} n^{1/2^k} = n^{\sum_{k=1}^{i} 1/2^k}$. The geometric sum in the exponent evalutes to $1 - 1/2^i$, so the number of nodes at depth i is $n^{1-1/2^i}$. The work done at depth i is then bounded above by $(c\,n^{1/2^i}) \cdot n^{1-1/2^i}$, i.e., $c\,n$, so each depth does the same amount of work. To find how many levels there before a constant-size problem is performed, we solve for i such that the problem size at depth i is constant:

$$c = n^{1/2^i} \implies 2^i \log c = \log n \implies i + \log\log c = \log\log n \implies i = O(\log\log n).$$

Multiplying by the work at each layer yields a total running time of $O(n \log\log n)$.



Rubric:

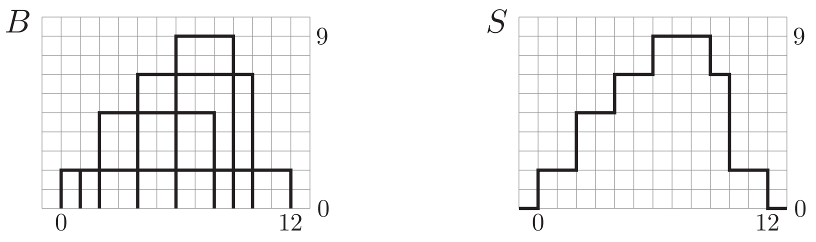
- 1 point per tree drawing
- 2 points per correct solution via tree, 4 points for part (iv)
- 2 points per correct solution via Master Theorem
- -1 point globally (on first instance only) of writing $O(\cdots)$ in every node in a recurrence tree, instead of using $c\cdots$ (need to use a single constant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

Problem 1-4. [50 points] **Cityscapes**

Judy Ruliani is the mayor of Yew Nork, a small town located on the Husdon Bay. She has pledged to transform the town into a major metropolitan city by facilitating the construction of skyscrapers, which will remake the city skyline as seen from across the bay. Given a set of construction requests for new buildings, she would like to know what the future skyline of her city will be.

Each proposed new **building** is represented by a triple (x_L, h, x_R), denoting what it will look like from across the bay: a rectangular projection having horizontal extent from $x_L \in \mathbb{N}$ to $x_R \in \mathbb{N}$ with $x_L < x_R$, and vertical extent from height 0 to $h \in \mathbb{N}^+$. The **skyline** of a set of buildings is an ordered sequence of pairs $((x_0, h_0), (x_1, h_1), \ldots, (x_{k-1}, h_{k-1}), (x_k, h_k))$, such that for all $i \in \{0, \ldots, k-1\}$: $x_i < x_{i+1}$, $h_i \neq h_{i+1}$, and for every $x \in [x_i, x_{i+1})$: there exists no input building (x_L, h, x_R) such that $x_L \leq x < x_R$ and $h > h_i$, and if $h_i > 0$, there exists an input building (x_L, h, x_R) such that $x_L \leq x < x_R$ and $h = h_i$. Note, we require $h_k = 0$ and $h_0 \neq 0$.

For example, the skyline of buildings $B = \{(4, 7, 10), (2, 5, 8), (6, 9, 9), (0, 2, 1), (1, 2, 12)\}$ would be $S = ((0, 2), (2, 5), (4, 7), (6, 9), (9, 7), (10, 2), (12, 0))$.



(a) [5 points] Given n buildings, a naïve incremental algorithm can construct a skyline by computing the skyline of the first k buildings recursively, and then add building $k+1$ to the skyline. Describe an $O(k)$ time algorithm to add one building to a skyline which was constructed from k buildings. (For any algorithm you describe in this class, you should **argue that it is correct**, and **argue its running time**.)

Solution: For this part, it is sufficient to construct a skyline $[(x_L, h), (x_R, h)]$ from the new building (x_L, h, x_R) and reduce to part (b). Here is an alternative algorithm. We transform input skyline S into a skyline which includes added building (x_L, h, x_R) directly by: (1) adding x_L and x_R into the existing skyline if they do not already exist, (2) raising each pair in the range of the building to h if its height is smaller than h, and then (3) removing pairs which violate $h_i \neq h_{i+1}$. Then we will have constructed our skyline by definition. (1) For each $x \in \{x_L, x_R\}$, if x does not exist in any pair of S, find the pair (x', h') with largest $x' < x$. If no such pair exists, insert $(x, 0)$ at the start of S; otherwise insert (x, h') after (x, h'). Now x_L and x_R are in the skyline without effecting the shape. (2) Then for each pair $(x_i, h_i) \in S$ where $x_L \leq x_i < x_R$, set $h_i = h$ if $h_i < h$. Now each interval is at the correct height. (3) Lastly for each adjacent pair $(x_i, h_i), (x_{i+1}, h_{i+1}) \in S$, remove pair (x_{i+1}, h_{i+1}) if $h_i = h_{i+1}$. The two insertions, raising the pairs, and removing repeats can each be done in $\Theta(k)$ time, so the whole procedure also runs in $\Theta(k)$ time.

Rubric:

- 2 points for description of a correct algorithm
- 2 points for correctly arguing that algorithm is correct
- 1 point for running time analysis
- Partial credit may be awarded

(b) [10 points] Consider two sets of buildings B_1 and B_2 where $n = |B_1| + |B_2|$, with S_1 the skyline of B_1 and S_2 the skyline of B_2. Describe an $O(n)$ time algorithm to compute the skyline of $B_1 \cup B_2$ from S_1 and S_2.

Solution: Given skylines S_1 and S_2 we compute the merged skyline similiarly to the merge step of merge sort. We will repeatedly increase x (initially $-\infty$) until $x = \infty$, while constructing S one pair at a time, maintaining the invariant that S is the skyline of $B_1 \cup B_2$ up to x. This invariant is vacuously true at initialization as input x values are finite. Now assume for induction that S is the skyline of $B_1 \cup B_2$ up to x. For each $i \in \{1, 2\}$, we maintain the leftmost pair $(x_i, h_i) \in S_i$ strictly to the right of x and the height c_i of S_i at x, initially zero; to ensure a pair is defined for all finite x, add $(\infty, 0)$ to the ends of S_1 and S_2.

Our goal is to increase x until $x = \infty$ and maintain the invariant. Let $h = \max(c_1, c_2)$, the skyline height at x. The next x that might appear in S is at $x' = \min(x_1, x_2)$. For each of $i \in \{1, 2\}$ if $x_i = x'$, set c_i to h_i, move pair (x_i, h_i) one to the right, and let $h' = \max(c_1, c_2)$, the skyline height at x'. If $h = h'$ then the height of the skyline does not change from x to x' so S already maintains the invariant for x'. Alternatively,

$h \neq h'$ and the skyline height does change, so add (x', h') to S, maintaining the invariant for x'. During this process, x is assigned once to each unique x value in S_1 and S_2 in increasing order, in addition to $-\infty$ and ∞, while processing each x takes constant time. The skyline of n buildings can have at most $2n$ pairs, so the running time of this algorithm is $\Theta(|S_1| + |S_2|) = \Theta(|B_1| + |B_2|) = \Theta(n)$.

Rubric:

- 4 points for description of a correct algorithm
- 4 points for correctly arguing that algorithm is correct
- 2 point for running time analysis
- Partial credit may be awarded

(c) [10 points] Describe an $O(n \log n)$ time algorithm to find the skyline of n buildings.

Solution: One correct algorithm is analgous to merge sort: split the buildings into two roughly equal sets, find their skylines recursively, and then combine them in $\Theta(n)$ using part (b). If there is only one building (x_R, h, x_L), return the skyline $((x_R, h), (x_L, 0))$ in constant time. If part (b) is correct, then this algorithm is correct by induction. The recurrence for this algorithm is $T(n) = 2T(n/2) + \Theta(n)$ yielding a running time of $T(n) = \Theta(n \log n)$ as desired.

Rubric:

- 4 points for description of a correct algorithm
- 4 points for correctly arguing that algorithm is correct
- 2 point for running time analysis
- Partial credit may be awarded

(d) [25 points] Write a Python function `build_skyline` 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:

```
def build_skyline(B, i = 0, j = None):
    if j is None: j = len(B)
    if j - i == 1:
        (xL, h, xR) = B[i]
        return [(xL, h), (xR, 0)]
    m = (i + j + 1) // 2
    S1 = build_skyline(B, i, m)
    S2 = build_skyline(B, m, j)
    return merge_skylines(S1, S2)

inf = float('inf')
def merge_skylines(S1, S2):
    S1.append((inf, 0))
    S2.append((inf, 0))
    S = []
    i, j, c1, c2, h, x = 0, 0, 0, 0, 0, -inf
    while x < inf:
        x1, h1 = S1[i]
        x2, h2 = S2[j]
        x_ = x1 if x1 < x2 else x2
        if x1 == x_:
            c1 = h1
            i += 1
        if x2 == x_:
            c2 = h2
            j += 1
        h_ = c1 if c2 < c1 else c2
        if h != h_:
            S.append((x_, h_))
        x, h = x_, h_
    return S
```

Rubric:

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