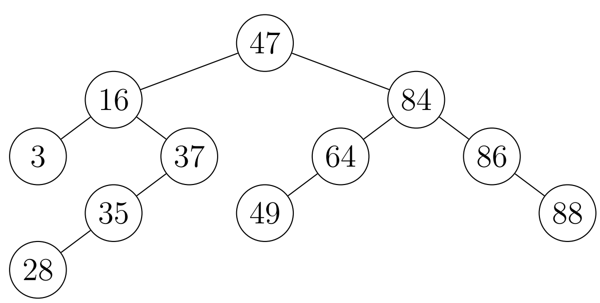
Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — February 29, 2020
Instructors: Erik Demaine, Jason Ku, and Justin Solomon — Problem Set 4

Problem Set 4

All parts are due on March 6, 2020 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 4-1. [10 points] **Binary Tree Practice**

(a) [2 points] The Set Binary Tree `T` below is **not height-balanced** but does satisfy the **binary search tree** property, assuming the key of each integer item is itself. Indicate the keys of all nodes that are not height-balanced and compute their skew.



(b) [5 points] Perform the following insertions and deletions, one after another in sequence on `T`, by adding or removing a leaf while maintaining the binary search tree property (a key may need to be swapped down into a leaf). For this part, **do not** use rotations to balance the tree. Draw the modified tree after each operation.

```
T.insert(2)
T.delete(49)
T.delete(35)
T.insert(85)
T.delete(84)
```

(c) [3 points] For each unbalanced node identified in part (a), draw the two trees that result from rotating the node in the **original** tree left and right (when possible). For each tree drawn, specify whether it is height-balanced, i.e., all nodes satisfy the AVL property.

Note: Material on this page requires material that will be covered in **L08 on March 3, 2020**. We suggest waiting to solve these problem until after that lecture. All other pages of this assignment can be solved using only material from L07 and earlier.

Problem 4-2. Heap Practice [10 points]

For each array below, draw it as a **complete**[1] binary tree and state whether the tree is a max-heap, a min-heap, or neither. If the tree is neither, turn the tree into a min-heap by repeatedly swapping items that are **adjacent in the tree**. Communicate your swaps by drawing a sequence of trees, marking on each tree the pair that was swapped.

(a) `[4, 12, 8, 21, 14, 9, 17]`

(b) `[701, 253, 24, 229, 17, 22]`

(c) `[2, 9, 13, 8, 0, 2]`

(d) `[1, 3, 6, 5, 4, 9, 7]`

Problem 4-3. [10 points] Gardening Contest

Gardening company Wonder-Grow sponsors a nation-wide gardening contest each year where they rate gardens around the country with a positive integer[2] ***score***. A garden is designated by a ***garden pair*** (s_i, r_i), where s_i is the garden's assigned score and r_i is the garden's unique positive integer ***registration number***.

(a) [5 points] To support inclusion and reduce competition, Wonder-Grow wants to award identical trophies to the top k gardens. Given an unsorted array A of garden pairs and a positive integer $k \leq |A|$, describe an $O(|A| + k \log |A|)$-time algorithm to return the registration numbers of k gardens in A with highest scores, breaking ties arbitrarily.

(b) [5 points] Wonder-Grow decides to be more objective and award a trophy to every garden receiving a score strictly greater than a reference score x. Given a max-heap A of garden pairs, describe an $O(n_x)$-time algorithm to return the registration numbers of all gardens with score larger than x, where n_x is the number of gardens returned.

---

[1] Recall from Lecture 8 that a binary tree is ***complete*** if it has exactly 2^i nodes of depth i for all i except possibly the largest, and at the largest depth, all nodes are as far left as possible.

[2] In this class, when an integer or string appears in an input, without listing an explicit bound on its size, you should assume that it is provided inside a constant number of machine words in the input.

Problem 4-4. [15 points] **Solar Supply**

Entrepreneur Bonty Murns owns a set S of n solar farms in the town of Fallmeadow. Each solar farm $(s_i, c_i) \in S$ is designated by a unique positive integer ***address*** s_i and a farm ***capacity*** c_i: a positive integer corresponding to the maximum energy production rate the farm can support. Many buildings in Fallmeadow want power. A building (b_j, d_j) is designated by a unique ***name*** string b_j and a ***demand*** d_j: a positive integer corresponding to the building's energy consumption rate.

To receive power, a building in Fallmeadow must be connected to a **single** solar farm under the restriction that, for any solar farm s_i, the sum of demand from all the buildings connected to s_i may not exceed the farm's capacity c_i. Describe a database supporting the following operations, and for each operation, specify whether your running time is worst-case, expected, and/or amortized.

`initialize(`S`)`	Initialize database with a list $S = ((s_0, c_0), \dots, (s_{n-1}, c_{n-1}))$ corresponding to n solar farms in $O(n)$ time.
`power_on(`b_j`,`d_j`)`	Connect a building with name b_j and demand d_j to any solar farm having available capacity at least d_j in $O(\log n)$ time (or return that no such solar farm exists).
`power_off(`b_j`)`	Remove power from the building with name b_j in $O(\log n)$ time.
`customers(`s_i`)`	Return the names of all buildings supplied by the farm at address s_i in $O(k)$ time, where k is the number of building names returned.

Problem 4-5. [15 points] **Robot Wrangling**

Dr. Squid has built a robotic arm from $n+1$ rigid bars called ***links***, each connected to the one before it with a rotating joint (n joints in total). Following standard practice in robotics[3], the orientation of each link is specified locally relative to the orientation of the previous link. In mathematical notation, the change in orientation at a joint can be specified using a 4×4 ***transformation matrix***. Let $\mathcal{M} = (M_0, \dots, M_{n-1})$ be an array of transformation matrices associated with the arm, where matrix M_k is the change in orientation at joint k, between links k and $k+1$.

To compute the position of the ***end effector***[4], Dr. Squid will need the arm's ***full transformation***: the ordered matrix product of the arm's transformation matrices, $\prod_{k=0}^{n-1} M_k = M_0 \cdot M_1 \cdot \ldots \cdot M_{n-1}$. Assume Dr. Squid has a function `matrix_multiply(`M_1`,`M_2`)` that returns the matrix product[5] $M_1 \times M_2$ of any two 4×4 transformation matrices in $O(1)$ time. While tinkering with the arm changing one joint at a time, Dr. Squid will need to re-evaluate this matrix product quickly. Describe a database to support the following **worst-case** operations to accelerate Dr. Squid's workflow:

`initialize(`$\mathcal{M}$`)`	Initialize from an initial input configuration $\mathcal{M}$ in $O(n)$ time.
`update_joint(`k`,`M`)`	Replace joint k's matrix M_k with matrix M in $O(\log n)$ time.
`full_transformation()`	Return the arm's current full transformation in $O(1)$ time.

[3] More on forward kinematic robotics computation here: https://en.wikipedia.org/wiki/Forward_kinematics

[4] i.e., the device at the end of a robotic arm: https://en.wikipedia.org/wiki/Robot_end_effector

[5] Recall, matrix multiplication is not commutative, i.e., $M_1 \cdot M_2 \neq M_2 \cdot M_1$, except in very special circumstances.

Problem 4-6. [40 points] **$\pi z^2 a$ Optimization**

Liza Pover has found a Monominos pizza left over from some big-TEX recruiting event. The pizza is a disc[6] with radius z, having n ***toppings*** labeled $0, \ldots, n-1$. Assume z fits in a single machine word, so integer arithmetic on $O(1)$ such integers can be done in $O(1)$ time. Each topping i:

- is located at Cartesian coordinates (x_i, y_i) where x_i, y_i are integers from range $R = \{-z, \ldots, z\}$ (you may assume that **all coordinates are distinct**), and
- has integer ***tastiness*** $t_i \in R$ (note, topping tastiness can be negative, e.g., if it's pineapple[7]).

Liza wants to pick a point (x', y') and make a pair of cuts from that point, one going straight down and one going straight left, and take the resulting ***slice***, i.e., the intersection of the pizza with the two half-planes $x \le x'$ and $y \le y'$. The tastiness of this slice is the sum of all t_i such that $x_i \le x'$ and $y_i \le y'$. Liza wants to find a ***tastiest*** slice, that is, a slice of maximum tastiness. Assume there exists a slice with **positive tastiness**.

(a) [2 points] If point (x', y') results in a slice with tastiness $t \neq 0$, show there exists $i, j \in \{0, 1, \ldots, n-1\}$ such that point (x_i, y_j) results in a slice of equal tastiness t (i.e., a tastiest slice exists resulting from a point that is both vertically and horizontally aligned with toppings).

(b) [8 points] To make finding a tastiest slice easier, show how to modify a Set AVL Tree so that:

- it stores **key–value items**, where each item x contains a value $x.val$ (in addition to its key $x.key$ on which the Set AVL is ordered);
- it supports a new tree-level operation `max_prefix()` which returns in **worst-case** $O(1)$ time a pair $(k^*, \text{prefix}(k^*))$, where k^* is any key stored in the tree T that maximizes the ***prefix sum***, $\text{prefix}(k) = \sum\{x.val \mid x \in T \text{ and } x.key \le k\}$ (that is, the sum of all values of items whose keys are $\le k$); and
- all other Set AVL Tree operations maintain their running times.

(c) [5 points] Using the data structure from part (b) as a black box, describe a **worst-case** $O(n \log n)$-time algorithm to return a triple (x, y, t), where point (x, y) corresponds to a slice of maximum tastiness t.

(d) [25 points] Write a Python function `tastiest_slice(toppings)` that implements your algorithm from part (c), including an implementation of your data structure from part (b). You can download a code template containing some test cases from the website. Submit your code online at `alg.mit.edu`.

---

[6]The pizza has thickness a, so it has volume $\pi z^2 a$.

[7]If you believe that Liza's Pizza preferences are objectively wrong, feel free to assert your opinions on Piazza.

```python
from Set_AVL_Tree import BST_Node, Set_AVL_Tree

class Key_Val_Item:
    def __init__(self, key, val):
        self.key = key
        self.val = val

    def __str__(self):
        return "%s,%s" % (self.key, self.val)

class Part_B_Node(BST_Node):
    def subtree_update(A):
        super().subtree_update()
        ##########################################
        # ADD ANY NEW SUBTREE AUGMENTATION HERE #
        ##########################################

class Part_B_Tree(Set_AVL_Tree):
    def __init__(self):
        super().__init__(Part_B_Node)

    def max_prefix(self):
        '''
        Output: (k, s) | a key k stored in tree whose
                       | prefix sum s is maximum
        '''
        k, s = 0, 0
        ##################
        # YOUR CODE HERE #
        ##################
        return (k, s)

def tastiest_slice(toppings):
    '''
    Input:  toppings | List of integer tuples (x,y,t) representing
                     | a topping at (x,y) with tastiness t
    Output: tastiest | Tuple (X,Y,T) representing a tastiest slice
                     | at (X,Y) with tastiness T
    '''
    B = Part_B_Tree()   # use data structure from part (b)
    X, Y, T = 0, 0, 0
    ##################
    # YOUR CODE HERE #
    ##################
    return (X, Y, T)
```