Quiz 1

- Do not open this quiz booklet until directed to do so. Read all the instructions on this page.
- When the quiz begins, write your name on the top of every page of this quiz booklet.
- You have 120 minutes to earn a maximum of 120 points. Do not spend too much time on any one problem. Skim them all first, and attack them in the order that allows you to make the most progress.
- **You are allowed one double-sided letter-sized sheet with your own notes**. No calculators, cell phones, or other programmable or communication devices are permitted.
- Write your solutions in the space provided. Pages will be scanned and separated for grading. If you need more space, write "Continued on S1" (or S2, S3) and continue your solution on the referenced scratch page at the end of the exam.
- Do not spend time and paper rederiving facts that we have presented in lecture or recitation. Simply cite them.
- When writing an algorithm, a **clear** description in English will suffice. Pseudo-code is not required. Be sure to argue that your **algorithm is correct**, and analyze the **asymptotic running time of your algorithm**. Even if your algorithm does not meet a requested bound, you **may** receive partial credit for inefficient solutions that are correct.
- **Pay close attention to the instructions for each problem**. Depending on the problem, partial credit may be awarded for incomplete answers.

Problem	Parts	Points
1: Information	2	2
2: Frequentest	2	8
3: Haphazard Heaps	2	10
4: Transforming Trees	2	10
5: Sorting Sock	4	20
6: Triple Sum	1	15
7: Where am i?	1	15
8: Methane Menace	1	20
9: Vapor Invite	1	20
Total		120

81

Name: PGTD

MIT Kerberos Username:

Problem 1. [2 points] **Information** (2 parts) 2

(a) [1 point] Write your name and email address on the cover page.

ok 1

(b) [1 point] Write your name at the top of each page.

ok 1

Problem 2. [8 points] **Frequentest** (2 parts) 8

The following two Python functions correctly solve the problem: given an array `X` of n positive integers, where the maximum integer in `X` is k, return the integer that appears the most times in `X`. Assume: a Python `list` is implemented using a dynamic array; a Python `dict` is implemented using a hash table which randomly chooses hash functions from a universal hash family; and `max(X)` returns the maximum integer in array `X` in worst-case $O(|X|)$ time. For each function, state its **worst-case** and **expected** running times **in terms of n and k**.

(a) [4 points]

```
def frequentest_a(X):
    k = max(X)                  O(|X|)
    H = {}
    for x in X:                 O(|X|)
        H[x] = 0
    best = X[0]
    for x in X:
        H[x] += 1               O(|X|)
        if H[x] > H[best]:
            best = x
    return best
```

worst $O(|X|)$
expected $O(1)$

i. Worst-case:

$O(|X|^2)$

ii. Expected:

$O(|X|)$ 4

(b) [4 points] $\forall e \in X, 0 \le x \le k$.

```
def frequentest_b(X):
    k = max(X)
    A = []
    for i in range(k + 1):      O(k)
        A.append(0)
    best = X[0]
    for x in X:                 O(|X|)
        A[x] += 1
        if A[x] > A[best]:
            best = x
    return best
```

i. Worst-case:

$O(k+|X|)$

ii. Expected: 4

If $k = O(|X|)$,

$O(|X|)$.

Problem 3. [10 points] **Haphazard Heap** (3 parts)

Array `[A,B,C,D,E,F,G,H,I,J]` represents a **binary min-heap** containing 10 items, where the key of each item is a **distinct** integer. State which item(s) in the array could have the key with:

(a) the smallest integer

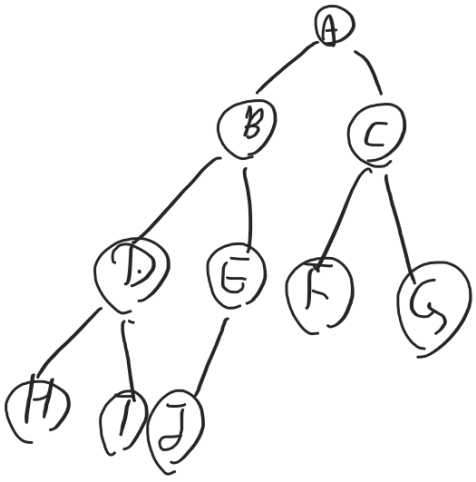
A

(b) the third smallest integer

B, C, D, E, F, G

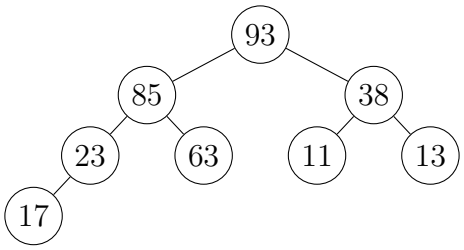
(c) the largest integer

H, I, J, F, G



Problem 4. [10 points] **Transforming Trees** (2 parts) 6

The tree below contains 8 items, where each stored item is an integer which is its own key.



(a) [6 points] Suppose the tree drawn above is the implicit tree of a binary max-heap `H`. State the array representation of `H`, **first before** and **then after** performing the operation `H.delete_max()`.

before ... : [93, 85, 38, 23, 63, 11, 13, 17]

after : [85, 63, 38, 23, 17, 11, 13]

(b) [4 points] Suppose instead that the original tree drawn above is a Sequence AVL Tree `S` (note Sequence data structures are zero-indexed). The items in **the leaves** of `S` in traversal order are $(17, 63, 11, 13)$. Perform operation `S.delete_at(3)` on `S` including any rotations, and then **list the items** stored in **the leaves** of `S` in traversal order, after the operation has completed. (You do not need to draw the tree.)

In traversal order of S, third node is 85. After deleting 85 and height-balancing, the list of the items stored in the lives is as follows.

[17, 63, 11, 13] in traversal order

using 1-indexing

Problem 5. [20 points] **Sorting Sock** (4 parts) 15

At Wog Hearts School of Wizcraft and Witcherdry, n incoming students are sorted into four houses by an ancient magical artifact called the Sorting Sock. The Sorting Sock first sorts the n students by each of the four houses' attributes and then uses the results to make its determinations. For each of the following parts, state and justify what type of sort would be most efficient. (By "efficient", we mean that faster correct algorithms will receive more points than slower ones.)

(a) [5 points] For House Puffle Huff, students must be sorted by ***friend number***, i.e., how many of the other $n - 1$ incoming students they are friends with, which can be determined in $O(1)$ time.

friend number $k \in \{0, 1, \ldots, n-1\}$. So it would be proper to sorting in linear time. We can use counting sort here. Counting sort takes $O(n+k)$ time, but in this case, $k = O(n)$. So $O(n)$ time.

(b) [5 points] For House Craven Law, students must be sorted by the weight of their books. Book weights cannot be measured precisely, but the Sorting Sock has a ***scale*** that can determine in $O(1)$ time whether one set of books has total weight greater than, less than, or equal to another set of books.

In this case, sorting can be only sorted by comparison. In comparison model, optimized sorting algorithm takes $\Omega(n \log(n))$ time. We can achieve this boundary using merge sort.

PGD

(c) [5 points] For House Driven Gore, students must be sorted by ***bravery***, which can't be directly measured or quantified, but for any set of students, the Sorting Sock can determine the bravest among them in $O(1)$ time, e.g., by presenting the students with a scary situation.

For selection sort, finding the lagest one takes $O(1)$-time, so selection sort takes $O(n \cdot 1) = O(n)$ time.

So using selection sort, we can solve the problem in linear time.

(d) [5 points] For House Leather Skin, students must be sorted by their ***magical lineage***: how many of a student's ancestors within the previous $3\lceil \log n \rceil + 4$ generations were magical. Recall that humans, magical or not, always have two parents in the previous generation, unlike binary tree nodes which have at most one. Assume the Sorting Sock can compute the magical lineage of a student in $O(1)$ time.

A student's magical lineage $\in \{0, 1, \ldots, 3\lceil \log n \rceil + 4\}$.

Max(Magical lineage) = $\Theta(\lceil \log n \rceil)$. So we can use a counting sort in $O(n + \lceil \log n \rceil) = O(n)$ time.

I had to consider $3\lceil \log(n) \rceil + 4$ generations.

not $3\lceil \log(n) \rceil + 4$ ancestors

Problem 6. [15 points] **Triple Sum** 10

Given three arrays A, B, C, each containing n integers, give an $O(n^2)$-time algorithm to find whether some $a \in A$, some $b \in B$, and some $c \in C$ have zero sum, i.e., $a + b + c = 0$. State whether your running time is worst-case, expected, and/or amortized.

Alg: First of all, A and B must be sorted. Sorting can be done within $O(n^2)$-time.

For each c, I will find a, b where a+b+c=0. For $0 \le i \le |C|-1$, let $c = C[i]$. If there does not exist a and b such that $a+b+c=0$, let $c = C[i+1]$. Iterate this procedure until succeeded to find appropriate a, b, and c or $i > |C|-1$. ($O(n)$)

For each iteration, start at $a = A[0]$, $b = B[|B|-1]$.

If $a+b+c > 0$, let $b = B[i-1]$,

If $a+b+c < 0$, let $a = A[i+1]$. ($O(n)$)

for some i

until $a = [|A|-1]$ OR $b = B[0]$.

Running time: $2O(n^2) + O(n) \cdot O(n) = O(n^2)$.

This is worst-case running time, since

worst-case: there is no a, b, and c such that $a+b+c=0$.

This takes $2O(n^2) + O(n) \cdot O(n) = O(n^2)$, so

$O(n^2)$-time is worst-case.

No Correctness

PGT

15

Problem 7. [15 points] **Where Am i?**

Given a Sequence AVL Tree T containing n nodes, and a pointer to a node v from T, describe an $O(\log n)$-time algorithm to return the (zero-indexed) index i of node v in the traversal order of T. (Recall that every node `u` in a Sequence AVL Tree T stores an item `u.item`, parent `u.parent`, left child `u.left`, right child `u.right`, subtree height `u.height`, and subtree size `u.size`.)

Let $X = \{x \mid x \in T,\ order(x) < order(v) \wedge x \in ancestors(v)\}$.

Then $i = \sum_{x \in X} [x.left.size + 1] + v.left.size$.

Claim! $\forall subtree(y), y \in T,\ y \in ancestors(v),\ y \in ancestors(v)$,

let $U = \{x \mid x \in subtree(y),\ order(x) < order(v) \wedge x \in ancestors(v)\}$,

$\sum_{t \in U} [t.left.size + 1] + v.left.size = |predecessors(v)|$.

Pf. By induction on $d = height(subtree(y)) - depth(v)$.

Base case ($d = 0$): v is the root node in subtree(y), so

$U = \emptyset$, $v.left.size = |predecessors(v)|$. So true.

Inductive step: Assume $\sum_{t \in U} [t.left.size + 1] + v.left.size = |predecessor(v)|$

for $d = d'$. Let's look at the case $d = d' + 1$.

For $d = d' + 1$, let S denote subtree(y). Make S' with root node y' by removing y and its subtree which does not contain v from S. By assumption, $\sum_{t \in U} [t.left.size + 1] + v.left.size = |predecessors(v)|$ for S'. Let's reattach y such that $y = y'.parent$.

Continued on S1

5

Problem 8. [20 points] **Methane Menace**

FearBird is a supervillian who has been making small holes in the methane gas pipe network of **mahtoG City**. The network consists of n pipes, each labeled with a distinct positive integer. A hole i is designated by a pair of positive integers (p_i, d_i), where p_i denotes the label of the pipe containing the hole, and d_i is a positive integer representing the ***distance*** of the hole from the ***front*** of pipe p_i. Assume any two holes in the same pipe p_i will be at different distances from the front of p_i. When a new hole (p_i, d_i) is spotted, the city receives a ***report*** of the hole to keep track of. The city will periodically patch holes using the following priority scheme:

- if each pipe contains at most one hole, patch any hole (if one exists);
- otherwise, among pairs of holes (p_i, d_i) and (p_j, d_j) appearing on the **same pipe**, i.e., $p_i = p_j$, identify any pair with smallest distance $|d_i - d_j|$ between them, and patch one of them.

Describe a database supporting the following operations, where k is the number of recorded but unpatched holes in the network at the time of the operation. State whether your running times are worst-case, expected, and/or amortized.

`initialize(`H`)`	Initialize the database with n holes $H = \{(p_0, d_0), \ldots, (p_{n-1}, d_{n-1})\}$, with one hole on each pipe, in $O(n)$ time
`report(`p_i`,` d_i`)`	Record existence of a hole in pipe p_i at distance d_i in $O(\log k)$ time
`patch()`	Patch any hole that follows the priority scheme above in $O(\log k)$ time

Let B denote a binary heap.

For every node $\in B$, a node contains a pair $((p_i, d_i), (p_j, d_j))$ if $p_i = p_j$, XOR, only a (p_i, d_i) if there does not exist (p_k, d_j) in H where $p_i = p_j$.

I think I came up with a data structure that goes well for initialize and patch operations so I graded a quarter.

Max-heap property is as follows:

For some i, j such that $i = \text{parent}(j)$, $B[i]$ and $B[j]$ have one of the three properties

① $B[i]$ has a pair, and $B[j]$ has only one

② If both $B[i]$ and $B[j]$ have a pair, $|d_s - d_t| \geq |d_k - d_l|$ where $B[i]$ contains (p_s, d_s), (p_t, d_t), $B[j]$ contains (p_k, d_k), (p_l, d_l).

Continued on S2

10

Problem 9. [20 points] **Vapor Invite**

Vapor is an online gaming platform with n users. Each user has a unique positive integer ***ID*** d_i and an updatable ***status***, which can be either active or inactive. Every day, Vapor will post online an ***active range***: a pair of positive integers (a, b) with the property that **every user** having an ID d_i contained in the range (i.e., with $a \leq d_i \leq b$) **must be active**. Vapor wants to post an active range containing as many active users as possible, and invite them to play in a special tournament. Describe a database supporting the following **worst-case** operations:

`build(`D`)`	Initialize the database with user IDs $D = \{d_0, \ldots, d_{n-1}\}$, setting all user statuses initially to active, in $O(n \log n)$ time
`toggle_status(`d_i`)`	Toggle the status of the user with ID d_i, e.g., from active to inactive or vice versa, in $O(\log n)$ time
`big_active_range()`	Return an active range (a, b) containing the largest number of active users possible in $O(1)$ time

user (d_i : ID
status : updatable, active/inactive

Set a AVL tree T containg user data.
For nodes $s, t \in T$, traversal_order(s) < traversal_order(t) IFF s.ID < t.ID.
And T contains two pointers to a node which has either the minimum ID or the maximum ID respectively.

Continued on S3

did not understand "active range"

SCRATCH PAPER 1. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write "Continued on S1" on the problem statement's page.

Case 1. order(y') < order(y). i.e. y.left = y', y ∉ U
subtree(y.left) = S', and y and y.right have an order bigger than that of v. So in this case, y doesn't affect i.

Case 2. order(y') > order(y). i.e. y.right = y', y ∈ U.
Every node ∈ y.left and y are in previous order of v. So

$$|predecessors(v)| = y.left.size + 1 + \sum_{t \in U'} [t.left.size + 1] + v.left.size$$
$$= \sum_{t \in U} [t.left.size + 1] + v.left.size,$$

where $U' = U - \{y\}$, $\left(\sum_{t \in U'} [t.left.size + 1] + v.left.size\right)$ follows the assumption. □

By the claim, the algorithm is correct.

And finding ancestors takes $O(\log(n))$ in case of AVL tree and getting size of a subtree takes $O(1)$-time, so the algorithm takes $O(\log(n)) \cdot O(1) = O(\log(n))$-time.

SCRATCH PAPER 2. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write "Continued on S2" on the problem statement's page.

③ Both B[i] and B[j] don't have a pair.

initialize(H): H must be sorted.

Counting sort on d_i for some i. $O(n)$

Counting sort on p_i for some i. $O(n)$

Making H a binary heap B. $O(n)$.

By the radix sort, every element $(p_i, d_i) \in H$ is sorted so that

- every element with the same p_i is adjacent each other
- For a chunk of elements with the same p_i, it is sorted by d_i.

$\forall (p_i, d_i)$, if there exists some (p_j, d_j) in H where $p_i = p_j$, make (p_i, d_i) and (p_j, d_j) a pair and then put it to B, otherwise just put a single (p_i, d_i) to B.

$O(n) + O(n) + O(n) = O(n)$.

patch: delete_max of priority queue. it takes $O(\log(k))$

SCRATCH PAPER 3. DO NOT REMOVE FROM THE EXAM.

You can use this paper to write a longer solution if you run out of space, but be sure to write "Continued on S3" on the problem statement's page.

build(D): For every $d \in D$, add d to T keeping the property of T. $O(n \log n)$

And checking if d has either maximum or minimum id. $O(1)$.

$$O(n \log(n)) \cdot O(1) = O(n \log(n))$$

toggle status(d_i): T is sorted by IDs, so finding a node with d_i takes $O(\log(n))$-time. Toggle takes $O(1)$-time.

$$O(\log(n)) \cdot O(1) = O(\log(n))$$

big_active_range(): Including every node is the largest range.

$a = T.min\_id$ $\quad$ $b = T.max\_id$

This takes

$$O(1) + O(1) = O(1) \text{ -time.}$$

6.006 Introduction to Algorithms

Spring 2020