CS161 Midterm Exam, Summer 2022

You have **85 minutes** to complete the exam. You may use up to 5 pages (front/back) of notes that you may have prepared, but you may not use other materials or confer with anyone.

The last page contains information (e.g., the Master Theorem, rules for red-black trees) that you may find convenient, although it is not an exhaustive resource. (It was posted on Ed on Tuesday night.) You may detach it before the exam begins, and need not hand it in.

Unless explicitly otherwise stated in a question, assume that:

- Big-O and big-Ω are meant in their informal "tight" sense. E.g., if we ask for the big-O or big-Ω running time of MergeSort, it would **not** be acceptable to say $O(n^2)$ or $\Omega(1)$.
- Operations such as addition, multiplication, array lookups – the ones that we generally consider to take $O(1)$ time in our course – also take $O(1)$ time here.

Please also familiarize yourself with directions for the first two parts of the exam:

- Problem 1: "Determine whether each of the following statements is true or false, and provide a brief but specific and convincing justification of your answer. Explanations that merely reword the statement (or its opposite) will receive no credit; insightful but incorrect explanations may receive some credit."
- Problem 2: "For these problems, you do not need to show your work, and grading will be based **only** on the final answer, although some answers may receive partial credit."

Because some people are taking the exam remotely, **we will not provide clarifications during the exam itself.** There are no intentional ambiguities, but if you believe you have encountered an ambiguity, please briefly note any assumptions you are making and move on.

Finally, please consider the following advice:

- The exam has many (small) problems, none of which require detailed explanations, and some of which require no explanations at all. Make sure you provide what each problem asks for, but do not waste time providing overly detailed or artfully-worded explanations; this will probably not help you (and will probably cost you valuable time!)
- Do not get too hung up on any one problem, especially not until you have looked at everything. The goal is to provide you many different opportunities to show what you know, instead of just a few high-stakes ones. Perfection is not expected. Good luck!

Name: ______________________ SUID (e.g., itullis): ______________________

This page is for any extra work, in case you require extra space for any problems. Please clearly indicate which problem(s) you are using this page for, if any, and please also make a note on those problems themselves.

Problem 1: True/False (5 pts. each, 30 pts. total)

Determine whether each of the following statements is true or false, and provide a brief but specific and convincing justification of your answer. Explanations that merely reword the statement (or its opposite) will receive no credit; thoughtful but incorrect explanations may receive some credit.

(a) RadixSort is a stable sort.

(b) If $f(n)$ is not $\Omega(g(n))$, then $\log_2(f(n))$ is not $\Omega(\log(g(n))$.

For this question, use the exact definition of big-Omega, not the informal "tight" one. Also, you do not need to write a full proof; if the statement is true, give the gist of the argument, and if it is false, exhibit a counterexample.

(c) Suppose that we are MergeSorting a list of size 2^k, with $k > 1$. For any two indices i, j in the original list L, there is exactly one Merge step in which, at some point, $L[i]$ is directly compared to $L[j]$ (i.e., they are at the heads of the two sublists).

(d) Given a list L of n **distinct** elements (with $n \geq 4$), it is possible to do the following using $O(n^2)$ time and $O(n^2)$ space: determine whether there are four different indexes a, b, c, d such that $L[a] \cdot L[b] = L[c] \cdot L[d]$.

(e) Suppose that $\mathcal{H}$ is a family of hash functions that hash the 100 elements in the universe $\mathcal{U} = \{0, 1, ..., 99\}$ to 10 buckets. Also suppose that any hash function in this family hashes exactly 10 elements in $\mathcal{U}$ to each of the 10 buckets. Then $\mathcal{H}$ must be universal.

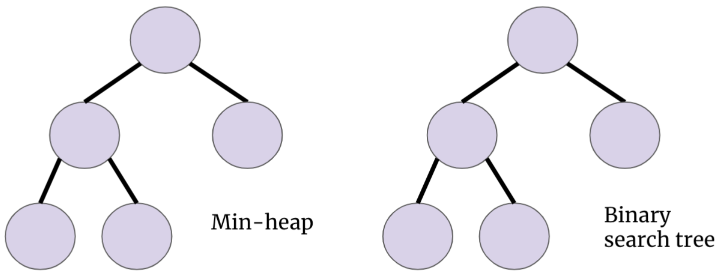
(f) Consider the following (Las Vegas) randomized algorithm for finding an instance of some element (call it E) **that is known to appear at least $\sqrt{n}$ times** in a list L of size n. It chooses a starting index in the list uniformly at random, and then loops through the entire list in left-to-right order, checking each index to see if that element is instance of E. When it finds one, it immediately returns True. If it reaches the end of the list, it wraps around to the beginning.

Claim: The average-case running time of this randomized algorithm is $O(\sqrt{n})$.

Problem 2: Short Exercises (4 pts. each, 20 pts. total)

For these problems, you do not need to show your work, and **grading will be based only on the final answer**, although some answers may receive partial credit.

(a) In each of the trees below, fill in the values 1, 2, 3, 4, 5 once each, such that the left tree is a min heap, and the right tree is a binary search tree. (If there is more than one possible answer, any valid answer is acceptable.)



(b) Suppose that one of the children of the root of a red-black tree is a black leaf (with two NILs as usual for a leaf). **How many nodes long, including the root and leaf but not counting NILs**, could the longest root-to-leaf path in such a tree possibly be? (Note: nodes along a root-to-leaf path might have more than one child. But we are counting only nodes *along* the path.)

(c) An *inversion* in a list L is a pair of indices i, j, with $i < j$, such that $L[i] > L[j]$. Suppose we take $L = [1, 2, ..., n]$ – notice that every element is distinct – and we randomly shuffle it such that any ordering is equally likely. What is the **expected number of inversions** in the shuffled list, as an exact function of n?

(d) (This is the one place on the exam where you should treat multiplication as not being $O(1)$.) Suppose we have two $2^n \times 2^n$ matrices, each of which has entries represented as 2^n-bit integers. Then combining Karatsuba's multiplication algorithm and Strassen's matrix multiplication algorithm (the efficient one from Homework 1) allows us to multiply two such matrices in (tight) $O(k^n)$ time, for a particular integer k. **What is k?**

(Hint: even if you did not memorize the running times of these two algorithms, you can re-derive them by remembering how they work; the reference sheet also has details on them. Then do some log tricks.)

(e) Indy is running a new developers' con with entry tickets that consist of three comma-separated groups of three digits. E.g., one possible ticket is "000,999,064", and another is "369,103,161". All three groups in a ticket are always different – no tickets like 161,103,161 can exist.

Indy uses a Bloom filter with 1000 bits (000 through 999) and two hash functions h_1 and h_2 that select a ticket's first and second groups of three digits, respectively. For example, $h_1(369, 103, 161) = 369$, and $h_2(369, 103, 161) = 103$.

An attendee with a ticket that the Bloom filter thinks it has seen is not allowed in. Otherwise, the attendee is let in, and the Bloom filter is updated according to that ticket's two hashes.

Suppose that people with all possible valid ticket numbers show up once each at the con, in some order. What is the **largest number of those people who could be let in, in the best case?** (Unlike in the HW problem, do not worry about people leaving, trying to reuse tickets or make invalid tickets, etc.)

Problem 3: Best Case (2 pts. each, 16 pts. total)

When we determine the big-O running time of an algorithm – whether it is deterministic or randomized – the convention unless otherwise stated is to assume a worst-case input. Let's consider an analogous situation for big-Ω, in which we will assume a **best-case input**! (In the case of preexisting data structures, we also assume **best-case contents**.)

For each of the following algorithms/operations, check **exactly one** box corresponding to the **tightest possible big-Ω** running time, assuming **best-case inputs and/or structure contents**. No explanation necessary (grading will be based **only on the check marks**).

- The problems are asking about the operations and algorithms (and their implementations) that we saw in class, **not** about *any* way to implement the operation / *any* algorithm that could solve the same problem.
- As usual in asymptotic analysis, assume that n can be arbitrarily large – the "best-case input" is the best-case input of size n.

Algorithm	$\Omega(1)$	$\Omega(\log n)$	$\Omega(n)$	$\Omega(n \log n)$	$\Omega(n^2)$
SAMPLE: **InsertionSort** on a list of n elements *(here the best-case input is an already-sorted list)*			✓		
SAMPLE: Add two $n \times n$ matrices *(here it doesn't really matter what the input is)*					✓
MergeSort on a list of n elements					
k-Selecting the median of a list of n elements					
QuickSort (with best-case randomness as well) on a list of n **distinct** elements					
One **Delete** from a hash table with $\Theta(n)$ buckets (and chaining via doubly linked lists) currently holding n^2 **distinct** elements in total*					
One **Insert** into a min heap** currently holding n **distinct** elements					
One **Delete-Min** from a min heap** currently holding n **distinct** elements					
One **Insert** into a binary search tree currently holding n **distinct** elements					
Breadth-First Search on an unweighted, undirected n-node tree, starting from a root given as input, to definitively find the largest number of steps to any leaf					

* Assume that the hash table only tries to find and delete a single element.

** (the array-based binary tree implementation we saw in class)

Problem 4: SelectSort (12 pts. total)

Waverly proposes the following **SelectSort** algorithm for sorting a list of n distinct comparable objects:

- If $n \leq 99$, sort the list in constant time using MergeSort. Otherwise:
- Run k-Select 9 separate times to find the $\lfloor \frac{n}{10} \rfloor$-th, $\lfloor \frac{2n}{10} \rfloor$-th, ..., $\lfloor \frac{9n}{10} \rfloor$-th smallest elements of the list. (Don't worry about the floors here – the idea is to find 9 pivots that are as evenly spaced as possible.)
- Partition the list into 10 groups: everything no smaller than the $\lfloor \frac{n}{10} \rfloor$-th smallest element, everything larger than that element but no smaller than the $\lfloor \frac{2n}{10} \rfloor$-th smallest element, etc. Because there are 9 pivots to consider, each *individual* element can be placed in the appropriate group in $O(1)$ time.
- Recursively call SelectSort on each of these 10 groups.
- Concatenate the results of those calls together (assume this is also $O(1)$) to get the final sorted list.

(a) (4 pts.) Write a recurrence for SelectSort. You may assume that the base case is $T(n) = 1$ for $n \leq 99$; you don't need to write this out again. You do not need to worry about floors, ceilings, etc. – for the purposes of this problem, you can just pretend the split is into equally-sized parts. Your recurrence should include a tight big-O term. No justification needed.

(b) (4 pts.) Give a tight big-O bound on the running time of SelectSort, and briefly justify or demonstrate how you got your answer. (You should **not** need to write a formal proof for this part.)

(c) (4 pts.) Terry wonders whether there is some way to modify k-Select to be able to find *multiple* values efficiently. For example, consider this proposed algorithm, which (like Waverly's) runs on a list of distinct values that are not necessarily integers:

MultiKSelect(L, q): finds the $\lfloor \frac{n}{q} \rfloor$-th, $\lfloor \frac{2n}{q} \rfloor$-th, ..., $\lfloor \frac{(q-1)n}{q} \rfloor$-th smallest values of an n-element list L, in $O(n \log(\log q))$ time, and returns them in order. (Note: $2 \leq q \leq n$.)

For example, for $L = [8, 7, 9, 6, 11, 10]$, $q = 4$, the algorithm would find the $\lfloor \frac{6}{4} \rfloor$-th $=$ 1st, $\lfloor \frac{12}{4} \rfloor$-th $=$ 3rd, and $\lfloor \frac{18}{4} \rfloor$-th $=$ 4th smallest elements of the list, which are $6, 8, 9$, and return the list $[6, 8, 9]$.

Either informally describe how an algorithm that accomplishes the claimed task could be implemented within the claimed time bound, or informally explain why no such algorithm can exist. (Notice that the algorithm is not claiming that the $q - 1$ values are necessarily *exactly* evenly spaced, and your argument should not center around details of divisibility.)

Problem 5: (Low-)Weighted Graphs (12 pts. total)

Both parts of this problem deal with undirected graphs with n vertices and m edges, in which the edges are weighted and **all the weights are integers between** 1 **and** 10**, inclusive**. The two parts are totally independent and can be solved separately.

(a) (8 pts.) **Without modifying the graph at all** (or, e.g., creating or representing a new copy of the graph), explain how to alter Dijkstra's algorithm to run in $O(n + m)$ time (and $O(n + m)$ space) on such graphs. You can assume that the graph is represented as an adjacency list that itself takes $O(n + m)$ space. Also assume that, as in class, we just want the total cost of a minimum-cost path, and not the path itself.

You can just say which part(s) you are modifying, and how; you don't need to reiterate *all* of the details and data structures of the algorithm. You do not need to justify correctness or running time, but make sure you describe all parts of your new apparatus that are needed to ensure that the time and space limits really are satisfied.

(Hint: we do not recommend trying to change the overall way the algorithm works. Instead, think about how you can change any part(s) of the standard setup that would cause the new, stricter $O(n + m)$ time bound to be exceeded.)

(b) (4 pts.) **Without modifying Karger's Algorithm at all**, explain how to alter the graph so that Karger's can be used to find a min cut of the graph, i.e., a cut with the smallest total of cut edge costs. Also give a tight big-O expression (in terms of n and/or m) for the expected number of trials that it will take for Karger's to run on the graph.

You do not need to justify correctness of the algorithm on the modified graph, or justify the bound on the expected number of trials. Note that we are not asking about the running time of the algorithm.

Reference information

Some log stuff

$$\log_b(xy) = \log_b x + \log_b y \qquad \log_b \frac{x}{y} = \log_b x - \log_b y$$

$$\log_b x^a = a \log_b x \qquad b^{\log_b a} = a$$

Big-O definitions

We say that $f(n)$ is $O(g(n))$ if and only if there exist some positive real constant c and some integer n_0 such that for all integers $n \geq n_0, f(n) \leq c \cdot g(n)$.

We say that $f(n)$ is $\Omega(g(n))$ if and only if there exist some positive real constant c and some integer n_0 such that for all integers $n \geq n_0, f(n) \geq c \cdot g(n)$.

The Master Theorem

For recurrences of the form $T(n) = aT(\frac{n}{b}) + O(f(n))$,

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

where a is the number of sub-problems, b is the factor by which the input size shrinks, and n^d is the work required to create all the sub-problems and combine their solutions. (Note that the big-O values here are tight; for instance, if $\Theta(n^2)$ work is required, use $d = 2$.)

Karatsuba's algorithm

$$xy = (a \cdot 10^{n/2} + b)(c \cdot 10^{n/2} + d) = ac \cdot 10^n + (ad + bc)10^{n/2} + bd$$

Recursively compute $ac, bd, (a + b)(c + d)$.

Insertion sort

Take the second element in the list, and as long as there is a larger element to its left, keep moving our element left. Repeat for the third, etc. elements in the list.

Combinatorics corner

$$\boxed{\binom{n}{2} = \frac{n(n-1)}{2}}$$

Strassen's algorithm

$$\begin{aligned} P_1 &= A(F-H) & P_5 &= (A+D)(E+H) \\ P_2 &= (A+B)H & P_6 &= (B-D)(G+H) \\ P_3 &= (C+D)E & P_7 &= (A-C)(E+F) \\ P_4 &= D(G-E) \end{aligned}$$

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

Karger's Algorithm analysis quick summary

Let c be the size of some particular min cut. Consider any stage in the edge contraction algorithm; let k be the number of remaining vertices. We survive this round if and only if the next edge we contract is not in our min cut. Now, the degree of every vertex in the graph must be at least c (or else we could have just detached such a vertex as an even smaller min cut), so there are at least $\frac{kc}{2}$ edges total. So our probability of surviving this round is $\geq 1 - \frac{c}{\frac{kc}{2}} = \frac{k-2}{k}$. Then the probability of surviving all rounds is $\geq \frac{k-2}{k} \cdot \frac{k-3}{k-1} \cdot \ldots \cdot \frac{2}{4} \cdot \frac{1}{3} = \frac{2}{n(n-1)}$, and therefore the expected number of rounds to find this min cut is the reciprocal of that, i.e., $\frac{n(n-1)}{2}$.

Universal hashing

Let $\mathcal{U}$ be the universe of all elements we could put in our hash functions, and let $\mathcal{H} = \{h_1, ..., h_m\}$ be a family of hash functions hashing these elements to n buckets. Then $\mathcal{H}$ is universal if and only if for every pair u_i, u_j of distinct elements in $\mathcal{U}$,

$$\mathbf{Pr}_{h \text{ in } \mathcal{H}}[h(u_i) = h(u_j)] \leq \frac{1}{n}$$

where the probability is over a (uniformly) random choice of a hash function h from the family $\mathcal{H}$.

Red-black tree rules

- Every node is colored red or black.
- The root node is a black node.
- Any missing children (a leaf has two of these, an internal node with only one child has one of these) are NILs that count as black nodes.
- Children of a red node are black nodes.
- For any node x, all paths from x to NILs have the same number of black nodes.