CSE 373 19su: Final, Part 1 Solutions

Name:	
UW email address:	

Instructions

- **Do not start the exam until told to do so; immediately stop writing when time is called.**
- No electronic devices (including calculators and cell phones) are allowed.
- You are **not** permitted notes on this part of the final.
- You have 60 minutes to complete the exam.
- Write your answers neatly in the provided space. Be sure to leave some room in the margins: we will be scanning your answers.
- If you need extra space, you may use the back of a sheet, **but you must clearly indicate in each subproblem where you want us to read the back**
- If you have a question, raise your hand to ask the course staff for clarification.
- Closed-forms of some summations, some logarithm identities, and Master Theorem are on a separate sheet.

Advice

- Write down your assumptions, thoughts and intermediate steps (this makes it easier to award partial credit).
- Clearly circle your final answer.
- The questions are not necessarily in order of difficulty; skip around!

Unless otherwise noted, all big-$\mathcal{O}$ and big-Ω analyses must be tight, and all $\mathcal{O}, \Omega, \Theta$ expressions must be simplified.

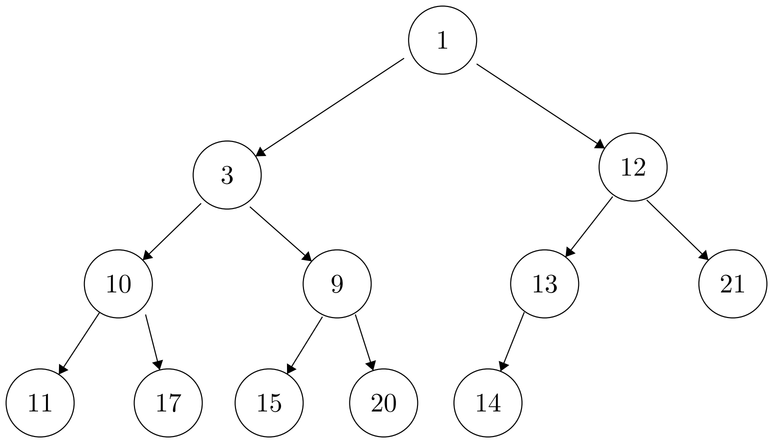
Any question asking for an algorithm or implementation should be as efficient as possible in big-$\mathcal{O}$ terms.

The following table gives **approximate** point values per problem. The values may change slightly during grading; these numbers are intended to help you prioritize what problems to spend time on.

Question	**Max points**
Heaps	3
Code Modeling I	17
Code Modeling II	6
Design Decisions 1: ADTs	9
Design Decisions 2: Data Structures	14
P vs. NP	6
Total	**55**

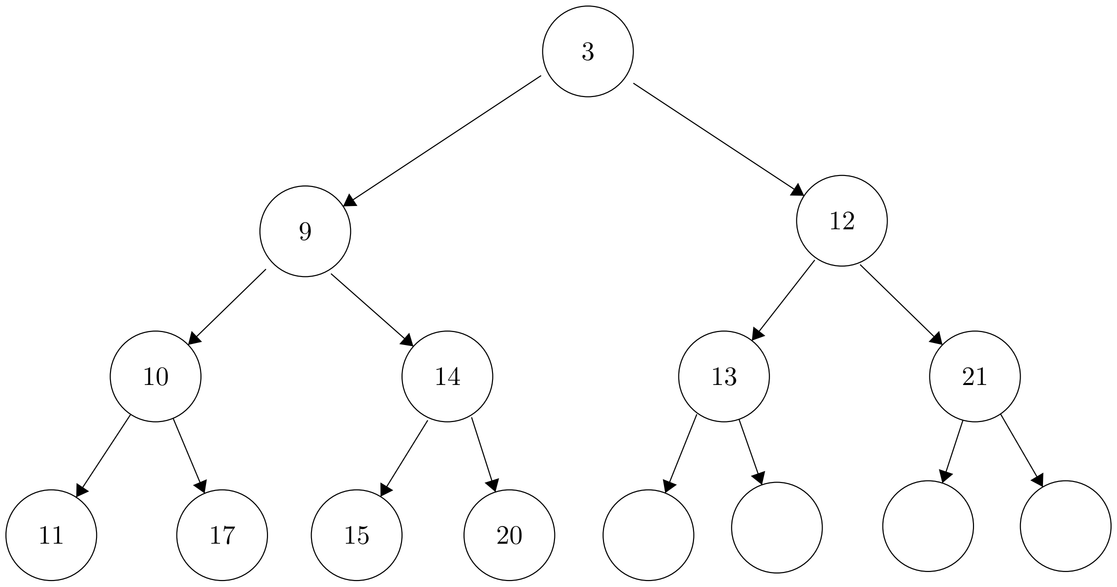
1. Heaps

Consider the following heap:



Suppose we call `removeMin` on the heap above. What is the state of the heap after the call? Draw the heap in the empty tree below. Leave any null/non-existent nodes blank in the tree.

Solution:



2. Code Modeling I

The `DirectedGraph` class below represents a **directed** graph. Each `Vertex` can iterate over its outneighbors, getting the next neighbor in $\Theta(1)$ time. The data structure `g.vertices` can also iterate over all the vertices (in no particular order), with each operation taking $\Theta(1)$ time. Finally the `ArrayDisjointSets` data structure implements both path compression and union-by-rank (as discussed in lecture, and you implemented in project 4).

```
1  public void mystery(DirectedGraph g) {
2      ArrayDisjointSets<Vertex> forest = new ArrayDisjointSets<>();
3      AVLTreeDictionary<Vertex, Boolean> seen = new AVLTreeDictionary<>();
4      for (Vertex v : g.vertices) {
5          seen.put(v, false);                // referenced in parts (c) and (d)
6          forest.makeSet(v);
7      }
8      for(Vertex v : g.vertices) {
9          if (!seen.get(v)) {                // referenced in part (d)
10             seen.put(v, true);             // referenced in part (d)
11             mysteryHelper(g, v, forest, seen);
12         }
13     }
14 }
15
16 private void mysteryHelper(DirectedGraph g, Vertex v, ArrayDisjointSets<Vertex> forest,
17         AVLTreeDictionary<Vertex, Boolean> seen) {
18     LinkedListQueue<Vertex> q = new LinkedListQueue<>();
19     q.enqueue(v);
20
21     while (!q.isEmpty()) {
22         Vertex curr = q.dequeue();
23         for (Vertex u : curr.outNeighbors()) {
24             if (!seen.get(u)) {            // referenced in part (d)
25                 q.enqueue(u);
26                 seen.put(u, true);         // referenced in part (d)
27             }
28             if (forest.findSet(curr) != forest.findSet(u)) {
29                 forest.union(curr, u);     // referenced in parts (a) and (b)
30             }
31         }
32     }
33 }
```

Each of the following questions should be answered in terms of m (the number of edges of g) and/or n (the number of vertices of g).

(a) What is the tight big-$\mathcal{O}$ for the worst-case running time of line 29 the **first** time it is executed?

Solution: $\mathcal{O}(1)$.

(b) What is the tight big-$\mathcal{O}$ for the worst-case running time of line 29 (on any single execution, not necessarily the first time)?

Solution: $\mathcal{O}(1)$. Notice we called `find` on `curr` and `u` just before the union call, so the paths were just compressed, and the union itself will take $\mathcal{O}(1)$ time.

(c) What is the tight big-$\mathcal{O}$ for the worst-case running time of line 5 (on any single execution, not necessarily the first time)?

Solution: $\mathcal{O}(\log n)$.

We're continuing our analysis of `mystery`. The code is copied on the bottom of the page for your convenience.

(d) What is the tight big-$\mathcal{O}$ for the worst-case total time used over all operations involving the `seen` dictionary. I.e. you should add up the total time taken for all executions of lines 5, 9, 10, 24, and 26.

Solution: Line $5, 9, 10$ will be executed n times. Line $24, 26$ will be executed m times. All the operations for AVLTreeDictionary is $\mathcal{O}(\log n)$. So in total the running time is $\mathcal{O}((m+n)\log n)$.

(e) What is the tight big-$\mathcal{O}$ for the largest **number of times** `union` can be called on `forest`?

Solution: $\mathcal{O}(n)$. This happens when the graph is weakly-connected.

(f) What is the tight big-$\mathcal{O}$ for the worst-case overall running time of `mystery`? For this question, assume that operations on `forest` took the in-practice running time, but otherwise do worst-case analysis.

Solution: $\mathcal{O}((m+n)\log n)$.

(g) What are the sets in `forest` when the method ends? (Your answer should be a graph theory term.)

Solution: Weakly connected components.

3. Code Modeling II

```java
public void stoogeSort(int[] input) {
    stoogeSortHelper(input, 0, input.size-1);
}

private void stoogeSortHelper(int[] input, int lo, int hi) {
    if (input[lo] > input[hi]) {
        int temp = input[lo];
        input[lo] = input[hi];
        input[hi] = temp;
    }

    int third = ((hi - lo) + 1) / 3;
    if (third == 0)
        return; //ends this call, takes O(1) time.

    stoogeSortHelper(input, lo, hi-third);
    stoogeSortHelper(input, lo+third, hi);
    stoogeSortHelper(input, lo, hi-third);
}
```

(a) Write a recurrence $T(n)$ for the worst-case runtime of the method `stoogeSortHelper`, where n is `hi - lo + 1` (i.e., the number of elements in the current subarray). Do not worry about finding the exact constants for the non-recursive term (for example, if the running time is $A(n) = 4A(n/3) + 25n$, you need to get the 4 and 3 right, but don't have to worry about getting the 25 right).

Solution:

$$T(n) = \begin{cases} 3T\left(\frac{2}{3}n\right) + 6 & \text{if } n \geq 3 \\ 5 & \text{otherwise} \end{cases}$$

(b) What are a, b, c in the Master Theorem for your recurrence? (You may want to consult the cheatsheet.)

Solution: $a = 3, b = 3/2, c = 0$.

(c) Write the inequality for Master Theorem, and use it to find the big-Θ worst-case running time of the algorithm. Do not worry about making the big-Θ bound look nice.

Solution: $\Theta\left(n^{\log_{3/2}(3)}\right)$

4. Design Decisions 1: ADTs

The startup you're working at has finally gotten big enough to start a real code-base for internal projects. Your project manager has put you in charge of each of the following projects.

For each of them choose the most appropriate ADT below, and briefly explain how you would use it in the scenario. In your explanation, please provide ADT specific information as needed. For example if your answer involves a dictionary, explicitly declare what the keys and values are. (1 sentence)
Briefly justifying your thought process may help us award partial credit if you interpret the problem differently than we expect (at most 1 sentence).

Dictionary, Priority queue, Union-Find, List

(a) The CEO is convinced projects will go faster if he just puts more people on them. He keeps taking teams and combining them into one big team. You need to be able to quickly tell any employee who their boss is, even through all the reorganization. (The CEO never splits teams up, he just combines them)

Solution: Union-Find. Each team is represented by a set, the root can be the boss as long as new team is led by one of old bosses (otherwise can add an extra field to root listing boss).

Finding boss can be implemented with `findSet` operation. And combining teams can be implemented by `union` operation.

(b) You are in charge of implementing a storage system for organizing all the bug reports clients have made. Because clients often complain about nothing, you've marked each report with how bad of a bug it is, and want to ensure developers handle the worst bugs first.

Solution: Priority queue.

Each report is weighted by the badness of the bug. Then report with worst bug will appear first in the queue and be handled first.

(c) The co-founder has decided everyone needs to relax with a "fun" team-bonding scavenger hunt. She comes to you with a piece of paper, showing all employees divided into 3 groups. You need to set up a data structure so that whenever an employee enters a code, it will show them the names of everyone on their team (but not the other teams).

Solution: Dictionary. The key is the employee, and the value is the names of everyone on their team.

5. Design Decisions 2: Data Structures

After the team-bonding scavanger hunt disaster, you quit the startup job and started contributing code to Wikipedia. For each of the following scenarios, we have given two potential data structures, you will analyze the trade-offs.

(a) You are creating redirect pages for Wikipedia. You want both of the following to be possible:

- When someone enters a "redirect word" on Wikipedia, you should find the article associated with that word (where the user's web browser will be redirected)
- You should be able to list all redirect words **in alphabetical order** along with their associated articles (there needs to be a human-readable page that lists all the redirections).

You decide to use a dictionary, where "redirect words" are keys, and the destination articles are values.

You will analyze two implementations: an AVL tree and a separate-chaining hash table.

(i) What is the simplified big-Θ in-practice running time of finding the associated page given a redirect word? Assume you have n pairs in your dictionary.

Solution: AVL tree:

separate chaining hash table:

(ii) How do you print out all the pairs, with the keys appearing **in alphabetical order** (we're looking for the high-level idea, not pseudocode—at most 1 sentence should suffice)?

Solution: AVL tree: An in-order traversal of the AVL tree will print the keys in order (recall that AVL trees are "fully sorted").

separate chaining hash table: Iterate through the entries, placing them in a new array, then sort the array (with, say, mergesort).

(iii) What is the big-Θ of the in-practice running time of listing all the keys in order?

Solution: AVL tree: $\Theta(n)$.

separate chaining hash table: $\Theta(n \log n)$.

(b) Whenever someone edits a Wikipedia page, a score is automatically generated representing how suspicious it is (e.g., edits to controversial pages or large edits will have higher scores; a single character change has a low score, since it's probably fixing a typo). Human moderators then review the most suspicious changes when they can, but sometimes the number of edits gets so large that a group of non-suspicious ones just have to be deleted without review. You want to both

- `removeMax`: Remove the change with the largest score from the data structure (to give it to an moderator)
- `deleteSmaller`: Delete all elements with scores less than some value v.

You decide to use a maximum priority queue as your ADT.

The two implementations you will consider are:

- a 4-heap—unlike Project 3, this heap does not use a dictionary to record the locations of the values; it just has the array. Additionally this is a max-heap, not a min-heap.
- sorted double linked list (i.e., a double linked list where the elements are increasing as you go from front to back).

(i) What is the simplified, worst-case big-Θ running time of inserting a new edit into the priority queue? Assume you have n entries in your priority queue.

> **Solution:** 4-Heap (assume the heap does not need to resize for this insertion): $\Theta(\log n)$.
>
> sorted DLL: $\Theta(n)$.

(ii) What is the simplified, worst-case big-Θ running time of the `removeMax` method?

> **Solution:** 4-Heap: $\Theta(\log n)$.
>
> sorted DLL: $\Theta(1)$.

(iii) Briefly (1–2 sentences) describe how the `deleteSmaller` method will work on the given data structure. We are looking for minimal detail (e.g., no edge cases or error checking), just a sentence-or-two of an outline. Your implementation should be as efficient as possible.

> **Solution:** 4-Heap: iterate over all elements in the heap and remove all the smaller elements, move all remaining elements to the front, then `buildHeap` the remaining data.
>
> Two slightly slower solutions were awarded partial credit:
>
> - Iterate through the heap, calling the `remove` method from the programming project
> - `buildHeap` into a min-heap, call `removeMin` until last element smaller than v is removed, and `buildHeap` back into a max-heap.
>
> sorted DLL: start from whichever end of your list contains the smallest elements, and delete elements until we reach some item that is larger than the value v. Alternative solutions could additionally start from both ends, and if the largest element smaller than v is discovered, simply set one of its pointers to null, and update the `head` or `tail` pointer (i.e. simply cut-off everything to be deleted and let the garbage collector handle it without iterating through all of it)

(iv) Suppose you call the `deleteSmaller` method and k elements are removed. What is the simplified, worst-case big-Θ running time of the method? (Your answer may depend on k and/or n.)

> **Solution:** 4-Heap: $\Theta(n)$
>
> sorted DLL: $\Theta(k)$

6. P vs. NP

(a) We discussed the complexity class "NP" What does NP stand for?

(i) Not polynomial

(ii) Normally polynomial

(iii) Nondeterministic polynomial

(iv) Nonuniform polynomial

Solution: iii

(b) Which of the following is the best realistic option when the problem you want to solve is NP-complete?

(i) Find a library that is guaranteed to efficiently give you the best answer to the problem.

(ii) Look for an algorithm that guarantees you a good answer (but not the best one) efficiently.

(iii) Abandon all hope, a computer will never do anything useful.

Solution: ii

(c) Which of the following best describes the status of P vs. NP?

(i) Most computer scientists believe $P \neq NP$, but we have no idea how to prove it.

(ii) Computer scientists are evenly split on whether $P = NP$ or not, but we should know whether it's true for sure in the next ten years.

(iii) Most computer scientists believe $P = NP$, but are evenly split on whether we will prove it in the near future.

Solution: i

7. Extra Credit

Draw yourself in a world where P = NP, and you are the only one with an efficient algorithm for an NP-complete problem.