CS 61BL Data Structures & Programming Methodology

Summer 2018

FINAL EXAM SOLUTION

This exam has 8 questions worth a total of 60 points and is to be completed in 110 minutes.
The exam is closed book except for four double-sided, handwritten cheat sheets. No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided.
Write the statement below in the blank provided and sign. You may do this before the exam begins.

"I have neither given nor received any assistance in the taking of this exam."

I have neither given nor received any assistance in the taking of this exam.

Signature: Blear Hug

Question	Points
1	½
2	½
3	10
4	14
5	9
6	12
7	14
8	0
Total	60

Name	Blear Hug
Student ID	1234567890
Lab Section	0 0 1
Name of person to left	Christine Zhou
Name of person to right	Kevin Lin

- There may be partial credit for incomplete answers. Write as much of the solution as you can, but we may deduct points if your answers are much more complicated than necessary.
- **Work through the problems with which you are comfortable first.** Do not get overly captivated by interesting design issues or complex corner cases you're not sure about.
- Not all information provided in a problem may be useful, and **you may not need all lines**. For code-writing questions, **write only one statement per line** and **do not write outside the lines.**
- Unless otherwise stated, all given code on this exam should compile. All code has been compiled and executed, but in the event that we do catch any bugs in the exam, we'll announce a fix. **Unless we specifically give you the option, the correct answer is not, 'does not compile.'**
- ○ means that only one circle may be filled. ☐ means that more than one box may be filled.

1. (½ pt) **Your Thoughts** What makes you strong?

Hopes and dreams.

2. (½ pt) **So It Begins** Write the statement on the front page and sign. Write your name, ID, and your lab section. Write the names of your neighbors. Write your name in the corner of every page.

3. **Wayback Machine**

   (a) (5 pts) Implement `doubleUp`, which takes a non-negative `int width` and returns a symmetric `int[]` of length `width * 2 + 1` with values doubling then halving.

   - `doubleUp(0)` returns $[1]$;
   - `doubleUp(1)` returns $[1, 2, 1]$;
   - `doubleUp(2)` returns $[1, 2, 4, 2, 1]$;
   - `doubleUp(3)` returns $[1, 2, 4, 8, 4, 2, 1]$.

```
public static int[] doubleUp(int width) {
    int[] result = new int[width * 2 + 1];
    int value = 1;
    for (int i = 0; i < width; i += 1 ______________________) {

        result[i] = value _____________________________________;

        value *= 2 ____________________________________________;
    }

    result[width] = value ______________________________;

    for (int i = 0; i < width; i += 1 ______________________) {

        value /= 2 ____________________________________________;

        result[width + 1 + i] = value _________________________;
    }
    return result;
}
```

   (b) (5 pts) Complete the assignment statements for `helper` so that `reverse` reverses the linked list.
   Hint: Draw out small examples and make use of the three local variables, `soFar`, `p`, and `temp`.

```
public class SLList {
    private static class IntNode {
        public int item;
        public IntNode next;
    }
    public IntNode sentinel;
    public void reverse() {
        if (sentinel.next != null)
            sentinel.next = helper(sentinel.next);
    }
```

Continued on the next page.

```
        private static IntNode helper(IntNode front) {
            IntNode soFar = null;
            IntNode p = front;
            while (p != null) {
                IntNode temp = p.next;

                p.next                       = soFar                        ;

                soFar                        = p                            ;

                p                            = temp                         ;
            }
            return soFar;
        }
    }
```

4. **Graphs** The framework below contains part of a class that implements a graph with adjacency lists. Part (a) involves the design of an `EdgeIterator` class that enumerates the edges of the graph.

   Note: The graph is not necessarily connected. It may not even contain any edges at all.

```
public class Graph implements Iterable<Edge>{
    private List<Integer>[] neighbors;
    public Graph(int V) {
        neighbors = (List<Integer>[]) new LinkedList[V];
        for (int i = 0; i < V; i += 1) {
            neighbors[i] = new LinkedList<>();
        }
    }
    public void addEdge(int from, int to) {
        neighbors[from].add(to);
    }
    public static class Edge {
        public final int from;
        public final int to;
        public Edge(int from, int to) {
            this.from = from;
            this.to = to;
        }
    }
    public Iterator<Edge> iterator() {
        return new EdgeIterator();
    }
```

Continued on the next page.

(a) (5 pts) Implement `EdgeIterator`, an inner class which returns a new `Edge` on each call to `next`, until all edges in the graph have been enumerated.

```
public class EdgeIterator implements Iterator<Edge> {
    private int index = 0;
    private Iterator<Integer> iter;
    public EdgeIterator() {

        while (index < neighbors.length) {

            if (!neighbors[index].isEmpty()) {

                iter = neighbors[index].iterator();
                return;
            }
            index += 1;
        }
    }
    public boolean hasNext() {

        return iter.hasNext() || index < neighbors.length;
    }
    public Edge next() {
        Edge toReturn = new Edge(index, iter.next());

        if (!iter.hasNext()) {
            index += 1;

            while (index < neighbors.length) {

                if (!neighbors[index].isEmpty()) {

                    iter = neighbors[index].iterator();
                    return toReturn;
                }

                index += 1;
            }
        }
        return toReturn;
    }
} // EdgeIterator class

} // Graph class
```

Name: Blear Hug

(b) (1½ pts) Give the runtime of DFS on a undirected, unweighted, **complete graph**, $G = (V, E)$. In a *complete graph*, every vertex has an edge to every other vertex. **Mark all that apply.**

☐ $\Theta(1)$ ☐ $\Theta(|V|)$ [✓ $\Theta(|V|^2)$] ☐ $\Theta(|V|^3)$ [✓ $\Theta(|E|)$] ☐ $\Theta(|E|^2)$ ☐ $\Theta(|E|^3)$

(c) (1½ pts) Suppose we have a connected graph $G = (V, E)$ with **3 edges of weight 0** but all other edges have **distinct, positive** weights. What is the minimum and maximum number of **minimum spanning trees** G could have? Give exact values.

Min: 1; **Max**: 3

(d) (1½ pts) Given a sorted list of edge weights, find a **minimum spanning tree** on a graph, $G = (V, E)$. Assume that G is *fully-connected*: a path exists between every pair of vertices. Which one of Kruskal's or Prim's Algorithm runs faster given the sorted edge list? Then, give the optimized runtime in terms of $|V|$ and $|E|$, the number of vertices and edges.

○ Prim's Algorithm: $\Theta($ ______ $)$ [✓ Kruskal's Algorithm: $\Theta(|E| \cdot \alpha(|V|))$]

Let the graph G have *distinct* weights. Let G' be the same graph except each weight increased by 1.

(e) (1½ pts) Suppose we run DFS on G and G' to find a path between two vertices in the graph. Assume that neighbors are added to the stack in the same order. Would the paths be the same?

[✓ Always] ○ Never ○ Depends on the graph, G

(f) (1½ pts) Suppose we run Prim's Algorithm on G and G'. Would the MSTs be the same?

[✓ Always] ○ Never ○ Depends on the graph, G

(g) (1½ pts) Suppose we run Dijkstra's Algorithm on G and G' to find the shortest path between two vertices in the graph. Would the shortest paths be the same?

○ Always ○ Never [✓ Depends on the graph, G]

5. **Sorting**

(a) (1½ pts) How many inversions are in the list, $[3, 6, 2, 5, 4]$? **Inversions**: 5

(b) (1½ pts) Give the array representation for a min-heap of five integers between $[1, 5]$ (possibly duplicated) that results in the **best-case runtime** for heapsort.

1 1 1 1 1

(c) (1½ pts) Suppose we have a priority queue whose operations (`insert`, `poll`, `changePriority`) all execute in $\Theta(1)$ time, regardless of key type. Give a tight asymptotic runtime bound to heapsort a list of N items using this priority queue.

Runtime: $\Theta(N)$

(d) (1½ pts) Can a priority queue with all $\Theta(1)$ operations (`insert`, `poll`, `changePriority`) exist?

In the general scenario, this would mean that we could sort an arbitrary number of items in linear time, which would violate the comparison sort runtime lower bound of $\Omega(N \log N)$. However, we also just showed the existence of a $\Theta(1)$-time operation priority queue above which can exist when the number of unique objects in a class is constant!

○ Yes, it can exist for any key type ○ No, it cannot exist [✓ Depends on the key type]

(e) (3 pts) Suppose we're considering alternatives to the counting sort algorithm used in LSD radix sort. When used as the sorting algorithm in LSD radix sort, which of the following sorts is guaranteed to correctly sort a list in $\Theta(WN \log N)$ time, where W is the length of the longest key and N is the number of keys? Assume the radix, R, is constant. **Mark all that apply.**

☐ Insertion sort ☐ Selection sort ☐ Binary tree sort [✓ Balanced tree sort]
☐ Heapsort [✓ Merge sort] ☐ Quicksort (*three-way partition*) ☐ Counting sort

Name: Blear Hug

6. **Runtime** For each problem, give the worst-case runtime in $\Theta(\cdot)$ notation as a function of N. Your answer should be simple with no unnecessary leading constants or summations.

(a) (3 pts) `TreeMap` is implemented using a red-black tree. **Worst Case**: $\Theta(N \log N)$

```
Map<Integer, String> map = new TreeMap<>();
for (int i = 0; i < N; i += 1) {
    map.put(i, "foo");
}
```

(b) (3 pts) `LinkedList` is doubly-linked with front/back pointers. **Worst Case**: $\Theta(N^2)$

```
List<Integer> list = new LinkedList<>();
for (int i = 0; i < N; i += 1) {
    if (!list.contains(i * 2)) {
        list.add(i);
    }
}
for (int j = 0; j < N; j += 1) {
    list.add(j);
}
```

(c) (3 pts) Assume `HashMap` and `HashSet` use linked lists for external chaining, double in size when the load factor exceeds $L = 0.75$, and `Integer::hashCode` distributes values uniformly and can be computed in constant time.

Worst Case: $\Theta(N^2)$

```
Map<Integer, Integer> map = new HashMap<>();
for (int i = 0; i < N; i += 1) {
    map.put(i, i + 1);
}
int sum = 0;
for (int i : map.keySet()) {
    Set<Integer> copy = new HashSet<>(map.values());
    if (copy.contains(i * 2)) {
        sum += 1;
    }
}
```

(d) (3 pts) `ArrayList` is implemented with geometric resizing. **Worst Case**: $\Theta(N^2)$

```
PriorityQueue<Integer> heap = new PriorityQueue<>();
for (int i = 0; i < N; i += 1) {
    heap.add(i);
}
List<Integer> list = new ArrayList<>();
while (!heap.isEmpty()) {
    list.add(0, heap.poll()); // add(int index, E element)
}
```

7. **Design** For each of the following design questions, a complete solution will include:
   - a description of the algorithm in enough detail that another student could easily implement it;
   - which data structures or algorithms enable your algorithm to execute as quickly as possible;
   - the worst-case asymptotic runtime bound in $\Theta(\cdot)$ notation with a justification in the description.

   If you think a certain step of your algorithm could be made more efficient but aren't sure exactly *how* to make it faster, precisely state the inefficiency and other ideas you considered for partial credit.

   (a) (4 pts) Consider the *hasheap table*, a twist on the usual hash table, that works as follows. First, create an array of length M. Then, for each integer x we'd like to add, compute its index with $x \mod M$ (`x % M`). To resolve collisions, use binary min-heaps for external chaining. Describe an efficient algorithm for **hasheap sort**, which sorts a list of integers using a *hasheap table*.

   ◯ I wish to receive ½ pt for this question; don't grade my answer.

   Since each external chain is a binary min-heap, we want a way to efficiently determine which of the M buckets contains the next smallest item. An inefficient solution would loop over all the buckets and keep track of the smallest element. But we can do better than that by creating an additional priority queue of M items for keeping track of the current minimum item in each bucket. In order to remember which bucket an item came from, define a wrapper class that holds both the minimum value (used for comparison) and the originating bucket. When we want to remove the minimum from this additional priority queue (in $O(\log M)$ time), replace it with the next smallest item from the same originating bucket. The total runtime is then upper-bounded by $O(N \log N)$ in the case that all items collide into the same bucket: no worse than regular heap sort. In the case that integers are uniformly distributed, the runtime is $O(N \log M \log \frac{N}{M})$, or $O(N \log N)$ since we suppose M is constant.

   Give the worst-case runtime for **hasheap sort** on an unsorted array in terms of N, the number of integers to be sorted. Assume M is constant.

   Worst Case: $\Theta(\underline{N \log N})$

   (b) (4 pts) A **spanning tree of a connected graph**, $G = (V, E)$, is *itself a graph*, $G' = (V', E')$. G' is a *tree* (acyclic connected graph) where $V' = V$ and $E' \subseteq E$ (all edges in E' are also in E). Describe an efficient algorithm that determines if G' is a spanning tree of some other graph, G.

   ◯ I wish to receive ½ pt for this question; don't grade my answer.

   The algorithm involves two steps. First, verify that G' is a tree by counting its edges—there must be exactly $|V| - 1$—or perform a depth-first search to make sure G' contains no cycles. (Note that G' is connected.) Then make sure every edge in G' is also in G.

   G' has to be in adjacency list format to the make the edge counting fast. G has to be in adjacency matrix format to make the edge checking fast. Counting the edges is $\Theta(|E'|)$; verifying containment is $\Theta(|E'|)$ since checking for an edge in G takes constant time. (Note that $|E'| = |V|$, at least after the edge count is verified.)

   With both graphs as adjacency lists, the running time is $\Theta(|V| \cdot |E|)$; with both as matrices, it's $\Theta(|V|^2)$.

   Give the worst-case runtime in terms of $|V|$, $|E|$, and $|E'|$. **Worst Case**: $\Theta(\underline{|E'|})$

Name: Blear Hug

(c) (6 pts) A k-d tree is a binary tree where each node contains a point of dimension k. Recall,

- "As one moves down the tree, one cycles through the axes used to select the splitting planes. (For example, in a 3-dimensional tree, the root would have an x-aligned plane, the root's children would both have y-aligned planes, the root's grandchildren would all have z-aligned planes, the root's great-grandchildren would all have x-aligned planes, [...].)
- Points are inserted by selecting the median of the points being put into the subtree, with respect to their coordinates in the axis being used to create the splitting plane. (Note the assumption that we feed the entire set of n points into the algorithm up-front.)" (Wikipedia)

Describe an efficient algorithm to construct a **perfectly balanced** k-d tree of dimension 2 given a list of N (x, y) decimal points. Assume that points are distinct, that points can have infinitely-many decimal places, and that the x and y values are comparable and that their `hashCode` uniformly distributes values and can be computed in constant time.

For full credit, your algorithm must have a worst-case runtime strictly better than $\Theta(N \log^2 N)$.

◯ I wish to receive ½ pt for this question; don't grade my answer.

To help guide our intuition for the problem, let's establish some lower and upper bounds on k-d tree construction.

We must consider all the points to be inserted, so our algorithm must run in $\Omega(N)$ time. As an upper bound for a naive algorithm, we can simply sort the sublists on each recursive call. Each recursive call divides the problem in half, and each half can then be merge-sorted in $\Theta(M \log M)$ time, where M represents the size of the list in some particular recursive call. The work at each level then sums to $\Theta(M \log \frac{M}{d}) = \Theta(M \log M - \log d)$ where d is the depth of the node in the tree. To compute the overall runtime, we substitute in N, the total number of points, for M. Since our resulting tree will be perfectly balanced, we have $\Theta(\log N)$ levels. Plugging in the runtime per level from earlier and simplify the d factor using the arithmetic sum, the overall runtime as a result of re-sorting the list at each call is in $\Theta(\log N \cdot (N \log N - \log^2 N)) = \Theta(N \log^2 N)$.

Our goal is to develop an $\Theta(N \log N)$ algorithm for balanced k-d tree construction. There are two problems we need to solve: **finding the median**, and **computing the next set of less-than and greater-than sublists**. In order to achieve this, we will need to make sure that each recursive call only does work linear to the number of elements at that call, $O(M)$, rather than in $\Theta(M \log M)$ as in the recursive re-sorting approach. We cannot use comparison-based sorting algorithms as they are lower-bounded by $\Omega(M \log M)$. We also cannot use radix-based sorting algorithms as the length of the key is potentially infinite.

To **find the median**, pre-sort the points by making two copies of the list in $\Theta(N)$ time and then sorting them on x and y with merge sort in $\Theta(N \log N)$ time. Then, finding the median is constant-time with `get(size() / 2)` so long as we maintain the relative order of the sublists.

To **compute the next set of less-than and greater-than sublists** in $\Theta(M)$ time *while maintaining the sorted order of the lists*, we'll need our constructor to take in the two lists sorted on x and y. We'll show how to compute the sets in the case where the current node splits on x. Computing the less-than and greater-than set of x is simple: just take the less-than sublist, X_{lt}, from $[0,$ `size() / 2`$)$ and the greater-than sublist, X_{gt}, from $[$`size() / 2 + 1`$,$ `size()`$)$.

Computing the less-than and greater-than sets for y is a little more tricky since we want all the points in X_{lt} sorted by their y-value, and all the points in X_{gt} sorted by their y-value, and we need to get both of these in constant time!

What we can do is put all the points in X_{lt} and X_{gt} into two `HashSet` objects to optimize containment checks. Then, loop through the points in the list sorted on y, and for each point p in the list, p is in the less-than set if p is in the `HashSet` of X_{lt}. We can define the greater-than set analogously: p is in the greater-than set if p is in the `HashSet` of X_{gt}. Each of these operations can be done in constant time for any one point (given the hashing assumptions in the problem), so computing the entire less-than set and the entire greater-than set can be done in $\Theta(M)$ time, where M is the size of the problem at any given recursive call.

Alternatively, we can avoid sorting the list entirely and instead use a complicated but asymptotically-optimal median finding algorithm like quickselect with median of medians as the pivot selection strategy. Quickselect is normally a worst-case quadratic time algorithm, but we can use median of medians to choose an *approximate median* as the pivot to avoid the worst-case runtime. There's no need to maintain any sorted order in this solution, and the constructor only needs to take in a single list of M points. This approach also guarantees $O(M)$ runtime at each recursive call.

Give the worst-case runtime in terms of N. **Worst Case**: $\Theta(N \log N)$

8. (0 pts) **Telepathy** Write *one* word which you believe the largest number of students in the class will also write. For each occurrence of the most commonly-written word, the entire class will receive some amount of extra credit points.

?______________

Exam scratch paper.