CS 61B
Spring 2022

Final Exam Review Worksheet

Sometime somewhere, 2022

Note this worksheet is very long and is not expected to be finished in an hour.

1 Mechanical Sorting

a) Suppose that we have the array below.

4	2	1	7	3	6	0

For each subpart, assume we are working with the original state of the array. For each algorithm, assume it is implemented as explained in lab. By a swap, we mean the exchange of two elements within the same array:

```
void swap(int[] x, int a, int b) {
    int temp = x[a];
    x[a] = x[b];
    x[b] = temp;
}
```

Note that if indices a and b are equal, i.e. if we try to swap an element with itself, it does **not** count as a swap.

i) Run selection sort. After how many swaps does 6 *first* go in front of 7?

○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8 ○ 9 ○ 10
○ 11

Solution:
○ 1 ○ 2 ○ 3 [✓ **4**] ○ 5 ○ 6 ○ 7 ○ 8 ○ 9 ○ 10
○ 11

ii) Run insertion sort. After how many swaps does 6 *first* go in front of 7?

○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8 ○ 9 ○ 10
○ 11

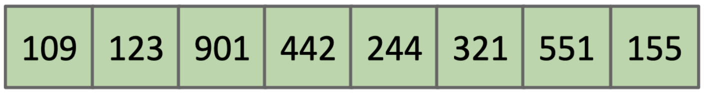
Solution:
○ 1 ○ 2 ○ 3 ○ 4 ○ 5 [✓ **6**] ○ 7 ○ 8 ○ 9 ○ 10
○ 11

iii) Run heapsort. Assume it is implemented in-place with bottom up heapification. Swaps during the bottom up heapification process should be counted. After how many swaps does 6 *first* go in front of 7?

○ 1 ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8 ○ 9 ○ 10
○ 11

Solution:
[✓ **1**] ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8 ○ 9 ○ 10
○ 11

b) Suppose we have the array below.



109	123	901	442	244	321	551	155

Run LSD radix sort. What is the state of the array in the last pass *before* it is fully sorted? Format your answer as a space separated list. For instance, if the array is {`1, 2, 3`}, it would be formatted as `1 2 3`.
State:

Solution:

`901 109 321 123 442 244 551 155`

c) Suppose we have the class `Olympian` below:

Suppose we want to sort all the `Olympians` in ascending order by their `medalCount` breaking ties alphabetically by `event` and then `name`. For simplicity, assume `names` are unique. For instance, if we have the Olympians below:

```
Olympian("Meshan", "Fencing", 0)
Olympian("Xiaowen", "Basketball", 3)
Olympian("Amudha", "Speedwalking", 0)
Olympian("Shreyas", "Basketball", 3)
Olympian("Ethan", "Speedwalking", 3)
Olympian("Ergun", "Speedwalking", 3)
Olympian("Allen", "Speedwalking", 3)
```

we would sort them as:

```
Olympian("Meshan", "Fencing", 0)
Olympian("Amudha", "Speedwalking", 0)
Olympian("Shreyas", "Basketball", 3)
Olympian("Xioawen", "Basketball", 3)
Olympian("Allen", "Speedwalking", 3)
Olympian("Ergun", "Speedwalking", 3)
Olympian("Ethan", "Speedwalking", 3)
```

To achieve this, you may call each of `insertionSort`, `selectionSort`, and `mergeSort` exactly once. `sortingKey` should be one of “name”, “event”, or “medalCount”. The signatures of the methods are below:

```
public List<Olympian> insertionSort(List<Olympian> olympians, String sortingKey)
public List<Olympian> selectionSort(List<Olympian> olympians, String sortingKey)
public List<Olympian> mergeSort(List<Olympian> olympians, String sortingKey)
```

```
public List<Olympian> sortOlympians(List<Olympian> olympians) {
    olympians = __________________________________________;
    olympians = __________________________________________;
    olympians = __________________________________________;
    return olympians;
}
```

Solution:

```
public List<Olympian> sortOlympians(List<Olympian> olympians) {
    olympians = selectionSort(olympians, "name");
    olympians = insertionSort(olympians, "event");
    olympians = mergeSort(olympians, "medalCount");
    return olympians;
}
```

Alternate Solution:

```
public List<Olympian> sortOlympians(List<Olympian> olympians) {
    olympians = selectionSort(olympians, "name");
    olympians = mergeSort(olympians, "event");
    olympians = insertionSort(olympians, "medalCount");
    return olympians;
}
```

2 Mix and Match

For each of the following sorts described below, give the best and worst case runtimes. If the sort described is invalid, answer "Invalid".

1. LSD radix sort with heap sort as a subroutine. Assume that there are at most B digits in any given key being sorted and that the size of the alphabet is R. Using heap sort as a subroutine means that heap sort is used when sorting on a given digit.

   Invalid. LSD radix sort requires the sorting subroutine to be stable. Otherwise, elements sorted at one digit can be placed out of order when the array is sorted on a more significant (further left) digit.

2. Divide the array into two halves. Sort the left half with insertion sort and sort the right half with selection sort. Finally, concatenate the two halves together to form the final sorted array.

   Invalid. Just because the left half is sorted and the right half is sorted doesn't mean that the concatenation will be sorted. For example, the smallest element in the right half might be smaller than the largest element in the left half.

3. Divide the array into two halves. Sort the left half with insertion sort and sort the right half with selection sort. Finally, merge the two halves together to form the final sorted array.

   Best: $\Theta(N^2)$, $Worst: \Theta(N^2)$

   Insertion sort's runtime, at least the in-place version, depends on the number of inversions in the array. If the array is already sorted, insertion sort runs in $\Theta(N)$ time. In the worst case, for example if the array is reversed, insertion sort runs in $\Theta(N^2)$. Selection sorting the right half always has runtime $\Theta(N^2)$ because each iteration looks at all elements left to be sorted. Merging the two halves takes $\Theta(N)$ time.

4. The three steps below describe a mix between merge sort and quick sort.

   (a) Repeatedly divide subarrays in half, like in merge sort, until there are $\leq N/16$ elements in each subarray.

   (b) Then, run quick sort on each subarray.

   (c) Finally, merge subarrays in pairs of two until the sorted original array is recovered.

   Best: $\Theta(N \log N)$, Worst: $\Theta(N^2)$

   In step 1 of our sort, we'll continue to divide the arrays in halves until the subarray contains $\leq N/16$ elements. This means that we stop the "divide" phase of merge sort at the 4th level of recursion.

   For each subarray of $\leq N/16$ elements, the best case runtime to quick sort is approximately $(N/16)\log(N/16)$. In total, the runtime to quick sort all 64 subarrays is $\Theta(N \log N)$. In the worst case, each subarray is quick sorted in approximately $(N/16)^2$ time, which totals to $\Theta(N^2)$ runtime over all subarrays.

Finally, in step 3, we merge the subarrays in pairs. Each level in the recursive tree will perform $\Theta(N)$ work and there are 4 levels. So, the total work for all merging is $\Theta(N)$.

3 Disjoint Sets

For all parts of this question, assume sets implementation **do not** use path compression, and the `WeightedQuickUnion` also implements `DisjointSets`.

a) Consider the following `WeightedQuickUnion` array state that uses negative values to denote set size {`-2, 2, 5, 0, 5, -4`}.

i) How many connected components are there?

◯ 1 ◯ 2 ◯ 3 ◯ 4 ◯ 5 ◯ 6

Solution:
◯ 1 [✓ **2**] ◯ 3 ◯ 4 ◯ 5 ◯ 6

ii) Does `isConnected(2, 3)` evaluate to `true` or `false`?

◯ `true` ◯ `false`

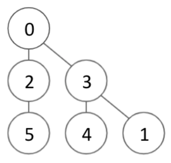
Solution:
◯ `true` [✓ **`false`**]

iii) Suppose we call `connect(0, 1)`. What is the resultant array? If we connect two sets of the same size, put the smaller number as a child of the other. Format your answer like so {`-2, 2, 5, 0, 5, -4`}.

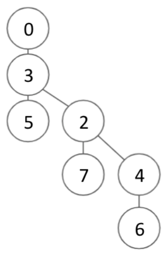
Array: ____________________________

Solution:
Array: {`5, 2, 5, 0, 5, -6`}

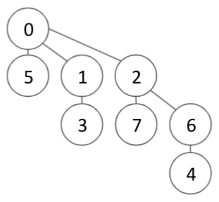
b) Consider the 7 tree states depicted below. For both parts, do not assume anything about how ties are broken when we `connect` two sets of the same size.



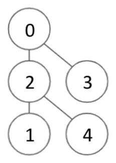
a.)



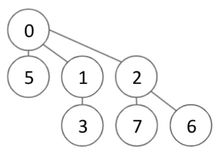
b.)



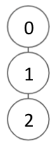
c.)



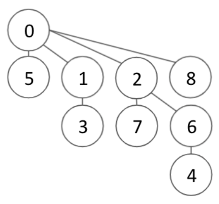
d.)



e.)



f.)



g.)

i) What states can be produced with a series of `QuickUnion` `connect` operations?

☐ a ☐ b ☐ c ☐ d ☐ e ☐ f ☐ g ☐ None of the above

Solution:
■ a ■ b ■ c ■ d ■ e ■ f ■ g ☐ None of the above

ii) What states can be produced with a series of `WeightedQuickUnion connect` operations?

☐ a ☐ b ☐ c ☐ d ☐ e ☐ f ☐ g ☐ None of the above

Solution:
☐ a ☐ b ■ c ☐ d ■ e ☐ f ■ g ☐ None of the above

c) Now consider a `QuickUnion` variation called `CrummyQuickUnion`, which roots the larger tree underneath the provided argument of the smaller tree during a `connect(a, b)` operation, and does not perform path compression. All other implementation details are unchanged.

i) Does `CrummyQuickUnion` implement the `DisjointSets` interface correctly?

○ yes ○ no

Solution: [✓ **yes**] ○ no

ii) Regardless of its correctness, describe the best and worst case runtime of any single call to `isConnected` when `CrummyQuickUnion` contains `N` total elements.

Best Case:

○ $\Theta(1)$ ○ $\Theta(log(logN))$ ○ $\Theta((logN)^2)$ ○ $\Theta(logN)$ ○ $\Theta(N)$ ○ $\Theta(NlogN)$ ○ $\Theta(N^2)$ ○ $\Theta(N^2logN)$ ○ $\Theta(N^3)$ ○ $\Theta(N^3$ log N) ○ $\Theta(N^4)$ ○ $\Theta(N^4$ log N) ○ Worse than $\Theta(N^4logN)$ ○ Never terminates (infinite loop)

Solution: [✓ $\Theta(1)$] ○ $\Theta(log(logN))$ ○ $\Theta((logN)^2)$ ○ $\Theta(logN)$ ○ $\Theta(N)$ ○ $\Theta(NlogN)$ ○ $\Theta(N^2)$ ○ $\Theta(N^2logN)$ ○ $\Theta(N^3)$ ○ $\Theta(N^3$ log N) ○ $\Theta(N^4)$ ○ $\Theta(N^4$ log N) ○ Worse than $\Theta(N^4logN)$ ○ Never terminates (infinite loop)

Worst Case:

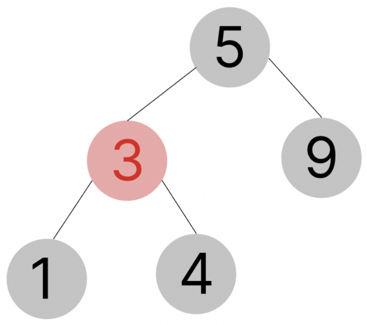
○ $\Theta(1)$ ○ $\Theta(log(logN))$ ○ $\Theta((logN)^2)$ ○ $\Theta(logN)$ ○ $\Theta(N)$ ○ $\Theta(NlogN)$ ○ $\Theta(N^2)$ ○ $\Theta(N^2logN)$ ○ $\Theta(N^3)$ ○ $\Theta(N^3$ log N) ○ $\Theta(N^4)$ ○ $\Theta(N^4$ log N) ○ Worse than $\Theta(N^4logN)$ ○ Never terminates (infinite loop)

Solution: ○ $\Theta(1)$ ○ $\Theta(log(logN))$ ○ $\Theta((logN)^2)$ ○ $\Theta(logN)$ [✓ $\Theta(N)$] ○ $\Theta(NlogN)$ ○ $\Theta(N^2)$ ○ $\Theta(N^2logN)$ ○ $\Theta(N^3)$ ○ $\Theta(N^3$ log N) ○ $\Theta(N^4)$ ○ $\Theta(N^4$ log N) ○ Worse than $\Theta(N^4logN)$ ○ Never terminates (infinite loop)

4 LLRB Insertions

Here is a video walkthrough of the solutions.

Given the LLRB below, perform the following insertions and draw the final state of the LLRB. In addition, for each insertion, write the fixups needed in the correct order. A fixup can either be a rotate right, a rotate left, or a color flip. If no fixups are needed, write "Nothing".



1. Insert 7

2. Insert 6

3. Insert 2

4. Insert 8

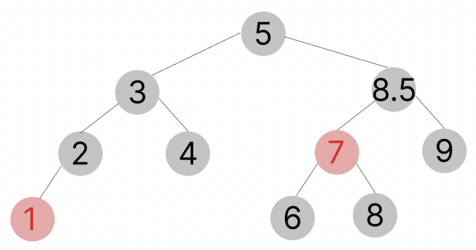
5. Insert 8.5

Final state:

Solution: For a visualization of the process, see **here**

1. Insert 7
   - Nothing
2. Insert 6
   - rotateRight(9)
   - colorFlip(7)
   - colorFlip(5)
3. Insert 2
   - rotateLeft(1)
4. Insert 8
   - Nothing
5. Insert 8.5
   - rotateLeft(8)
   - rotateRight(9)
   - colorFlip(8.5)
   - rotateLeft(7)

Final state:



5 Tries

The `MyTrieSet` class has the following class definition:

```java
public class MyTrieSet implements TrieSet61BL {

    private Node root;

    private class Node {
        private boolean isKey;
        private Map<Character, Node> map;

        private Node() {
            isKey = false;
            map = new HashMap<>();
        }
    }
        ...
}
```

a) Implement the method `numPrefixesOf` which accepts a `String key` and returns the number of words in the Trie that are **prefixes** of the given `key`. For instance, if the Trie contains the words `"be"`, `"bee"`, `"beetle"`, `"been"`, `"beetlebugs"`, and `"meals"`, then `numPrefixesOf("beetlebug")` would evaluate to **3** since *only* `"be"`, `"bee"`, and `"beetle"` are prefixes of `"beetlebug"`.

Note that the `key` is a prefix of itself, so `"beetlebug"` is a prefix of `"beetlebug"`.

```java
public int numPrefixesOf(String key) {
    if (key == null) {
        return 0;
    }
    int count = 0;
    Node curr = root;
    for (int i = 0; i < key.length(); i++) {

        char c = _____________________________;

        if (___________________________________) {
            break;
        }

        ________________________________________________;

        if (_______________________________________) {

            _________________________________________;
        }
    }
    return count;
}
```

Solution:
If a word is a prefix of `key`, then we'll encounter the word's `isKey = true` node while we traverse down the trie. For instance, in the example given in the question prompt, the 2nd letter "e", 3rd letter "e" and 6th letter "e" would all have `isKey = true`, so we should count those when traversing down to the key "beetlebug".

```
public int numPrefixesOf(String key) {
    if (key == null) {
        return 0;
    }
    int count = 0;
    Node curr = root;
    for (int i = 0; i < key.length(); i++) {
        char c = key.charAt(i);
        if (!curr.map.containsKey(c)) {
            break;
        }
        curr = curr.map.get(c);
        if (curr.isKey) {
            count += 1;
        }
    }
    return count;
}
```

b) Implement the method `similarity` which accepts a `String key` and returns an **aggregate, decimal score** between 0 and 1 of how *similar* the `key` is to **all** the words in the Trie.

If we compare the `key` to an arbitrary `word` in the Trie, we say their **similarity score** is calculated as the **`length of longest matching prefix / length of key`**. For instance, if `"chin"` is the `key` and we compare it to `"chair"` then we would say they have a similarity score of `0.5` since the longest matching prefix `"ch"` is of length `2`, `"chin"` is of length `4`, and `2 / 4` is `0.5`.

Then, we calculate the **aggregate, decimal score** as the **average** of all the **similarity scores.** For instance, let's calculate `similarity("been")` for a Trie that contains the words `"belt"`, `\bee"`, `"buggy"`, `"beenyboy"`, and `"chin"`. These words have individual similarity scores of 0.5, 0.75, 0.25, 1, and 0, respectively. Then, to calculate `similarity("been")` we should take the average of them as such: (0.5 + 0.75 + 0.25 + 1 + 0) / 5 is 0.5. So with this Trie, `similarity("been")` evaluates to 0.5.

In this part, and this part only, you may call *any* of the methods in the `TrieSet61B` interface below:

```
public interface TrieSet61B {

    /** Clears all items out of Trie */
    void clear();

    /** Returns true if the Trie contains KEY, false otherwise */
    boolean contains(String key);

    /** Inserts string KEY into Trie */
    void add(String key);

    /** Returns the number of words that start with PREFIX */
    int numKeysWithPrefix(String prefix);

    /** Returns the number of words in the Trie */
    int size();

    /** Returns the longest prefix of KEY that exists in the Trie */
    String longestPrefixOf(String key);

}
```

Implement `similarity` below. You may choose to implement it iteratively **or** recursively. If you choose to do the recursive approach, you may also implement the recursive helper method `similaritySum`. Skeletons for each have been provided. **Please delete the skeleton code for the approach you don't attempt!** If code for both is present, we will only consider the iterative solution. You may call the method from the previous part, but you shouldn't need to. You may not need all the lines. Recall, dividing two integers requires a double cast, e.g. `(double) 4 / 5`, for the result to be a double.

```
/* iterative skeleton */
public double similarity(String key) {
    _______________________________________________________;

    for (int index = ___________________; ___________________; __________________) {

        ___________________________________________________;

        ___________________________________________________;
    }

    return ____________________________________________;
}



/* recursive skeleton */
public double similarity(String key) {

    return __________________________________________;
}

private double similaritySum(String key, int index) {

    if (_____________________________________________) {

        return _____________________________________;
    }
    ____________________________________________________________________;

    return ______________________________________________________________;
}
```

Solution:

The most straightforward was to implement this method would be to iterate over the words in the set, calculate similarity scores, and average them. However, if we look at the TrieSet61B methods we can use, there's no easy way to iterate over words, or calculate similarity scores for a given key and word. So how else can we approach this?

Looking at the provided methods again, there are two that stand out: `numKeysWithPrefix` and `longestPrefixOf`. The method `longestPrefixOf` ends up not being very useful (since many words in the set will end up not being prefixes of `key`), but `numKeysWithPrefix` is critical here.

Notice that our similarity score can be rephrased as "total number of prefix letters" / (key.length() * number of words). If we want to calculate the number of prefix letters, we can go index by index and count how many words have the same prefix as `key` up to that index. We can eagerly divide by key.length() or size() if we want to; we get the same answer regardless.

```
/* iterative solution */
public double similarity(String key) {
    double score = 0;
    for (int index = 1; index <= key.length(); index += 1) {
        int matches = numKeysWithPrefix(key.substring(0, index));
        score += (double) matches / key.length();
    }
    return score / size();
}

/* alternate solution for iterative */
public double similarity(String key) {
   int totalPrefixLetters = 0;
   for (int index = 0; index < key.length(); index++) {
        totalPrefixLetters += numKeysWithPrefix(key.substring(0, index + 1));
   }
   return totalPrefixLetters / (key.length() * size());
}

/* recursive solution */
public double similarity(String key) {
    return similaritySum(key, 1) / size();
}

private double similaritySum(String key, int index) {
    if (index > key.length()) {
        return 0;
    }
    int matches = numKeysWithPrefix(key.substring(0, index));
    return (double) matches / key.length() + similaritySum(key, index + 1);
}
```