CS 61B Fall 2025

Midterm Review

Exam-Level Discussion 10: November 03, 2025

1 Recursive

For each of the following functions, give the tightest asymptotic runtime bound.

(a) *Hint: Draw out the recursive tree!*

```
void f(int N) {
    if (N <= 1) {
        return;
    }
    f(N - 1);
    f(N - 1);
    f(N - 1);
}
```

$\Theta(3^N)$. Drawing out the recursive tree, the i-th level does 3^i work. There are N levels total so the total work done is $1 + 3^1 + 3^2 + ... + 3^N = \Theta(3^N)$.

(b) Assume that `constantWork(N)` runs in constant $\Theta(1)$ time.

```
void f(int N) {
    if (N <= 1) {
        return;
    }
    constantWork(N); // Runs in constant time
    f(N / 2);
    f(N / 2);
}
```

$\Theta(N)$. Drawing out the recursive tree, the first level has 1 work, the next level has 2 work, and so, doubling each layer. This means we get a sum $2^{\{0\}} + 2^{\{1\}} + 2^{\{2\}} + ... + 2^{\{h\}}$, where h is the height of the tree. To find h, we ask ourselves how many times we can divide $\frac{N}{2}$ before reaching 1, which is $\log_{\{2\}}(N)$. This gives us, by the aforementioned sum, a runtime of $\Theta\left(2^{\left\{\log_{\{2\}}(N)\right\}}\right) = \Theta(N)$.

(c) What is the best and worst case runtime of `alpha`?

```
public void alpha(int N) {
    if (N % 4 == 0 || N <= 0) {
        return;
    }
    alpha(N - 4);
}
```

Best Case:

- ● $\Theta(1)$
- ○ $\Theta(\log N)$
- ○ $\Theta(N^2)$
- ○ $\Theta(N^3 \log N)$
- ○ $\Theta(\log(\log N))$
- ○ $\Theta(N)$
- ○ $\Theta(N^2 \log N)$
- ○ Worse than $\Theta(N^3 \log N)$
- ○ $\Theta((\log N)^2)$
- ○ $\Theta(N \log N)$
- ○ $\Theta(N^3)$
- ○ Never terminates (infinite loop)

Worst Case:

- ○ $\Theta(1)$
- ○ $\Theta(\log N)$
- ○ $\Theta(N^2)$
- ○ $\Theta(N^3 \log N)$
- ○ $\Theta(\log(\log N))$
- ● $\Theta(N)$
- ○ $\Theta(N^2 \log N)$
- ○ Worse than $\Theta(N^3 \log N)$
- ○ $\Theta((\log N)^2)$
- ○ $\Theta(N \log N)$
- ○ $\Theta(N^3)$
- ○ Never terminates (infinite loop)

2 Comparisons Between Combinations

(a) Consider the four cases below, and decide whether the **`Comparable`** or **`Comparator`** interface is better suited for the job.

We are writing a custom **`61BStudent`** class, and want to define the default comparison behavior to be a comparison between their **`Integer studentID`**s.

● **`Comparable`** ○ **`Comparator`**

We are writing a custom **`Dog`** class with many different attributes, including **`size`**, **`age`**, and **`timesBarked`**. We want to be able to sort 3 lists of **`Dog`**s by each of these attributes separately.

○ **`Comparable`** ● **`Comparator`**

We have a list of **`String`**s and want to sort them by an **`Integer`** frequency value stored in an external **`TreeMap`** (sound familiar?)

○ **`Comparable`** ● **`Comparator`**

We are writing a custom static sorting method that takes in a **`List`** and destructively sorts it. We want to be able to sort **anything** that has a predefined natural order.

● **`Comparable`** ○ **`Comparator`**

(b) Consider the **`City`** and **`CityPair`** classes below.

```
public class City {

  private String name;
  private double x;
  private double y;
  private int ranking;

  /* Constructor not shown...*/
}

public class CityPair {

  private City a;
  private City b;

  public CityPair(City a, City b) {
    this.a = a;
    this.b = b;
  }

  @Override
  public boolean equals(Object other) {
    // Two CityPairs are equal if they contain the same two cities.
    if (other instanceof CityPair p) {
      return ((this.a == p.a) && (this.b == p.b)) ||
      ((this.a == p.b) && (this.b == p.a));
    }
    return false;
  }
}
```

Implement the static method **proximitySort**, that takes in a **Set** of cities and returns a **List** of **CityPair** objects, sorted from closest to furthest for every possible pair of distinct cities. Assume **proximitySort** is in the same file as **City** and **CityPair**. Assume that **CityPair** has a valid and good hash function.

Additionally, implement the **pairDistance** helper method and **DistanceComparator** nested class. **DistanceComparator** compares **CityPair** objects such that **CityPair**s with a lower Euclidean distance between their respecitve **City** objects are considered "less than" those with a greater Euclidean distance.

Remember that the Euclidean distance formula is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ for two points (x_1, x_2), (y_1, y_2)

The square root function **Math.sqrt(double a)** and the power function **Math.pow(double base, int exponent)** may come in handy.

List.sort takes in a Comparator, and returns a sorted list according to the ordering defined by the comparator.

```
public static List<CityPair> proximitySort(Set<City> cities) {

  Set<CityPair> allCombinations = new HashSet<>();


  for (City a : cities) {

    for (City b : cities) {

      if (a != b) {

        allCombinations.add(new CityPair(a, b));
      }
    }
  }


  List<CityPair> orderedCombinations = new ArrayList(allCombinations);


  return orderedCombinations.sort(new DistanceComparator())
}

public static double pairDistance(CityPair p) {
  // Returns the euclidean distance between the two cities represented by a CityPair

  return Math.sqrt(Math.pow(p.b.x - p.a.x, 2) + Math.pow(p.b.y - p.a.y, 2));
}


public class DistanceComparator implements Comparator<CityPair> {


  @Override
  public int compare(CityPair p1, CityPair p2) {
    Double distance1 = pairDistance(p1);
    Double distance2 = pairDistance(p2);

    return distance1.compareTo(distance2);

  }
}
```

3 Filter

We want to make a `FilteredList` class that selects only certain elements of a `List` during iteration. To do so, we're going to use the `Predicate` interface defined below. Note that it has a method, `test` that takes in an argument and returns `true` if we want to keep this argument or `false` otherwise.

```
public interface Predicate<T> {
boolean test(T x);
}
```

For example, if `L` is any kind of object that implements `List<String>` (that is, the standard `java.util.List`), then `FilteredList<String> FL = new FilteredList<>(L, filter);` gives an **iterable** containing all items, `x`, in `L` for which `filter.test(x)` is `true`. Here, `filter` is of type `Predicate`.

Fill in the `FilteredList` class on the following page.

```
public class FilteredList<T> implements Iterable<T> {



    public FilteredList (List<T> L, Predicate<T> filter) {



    }

    @Override
    public Iterator<T> iterator() {

    }

    private class FilteredListIterator implements Iterator<T> {



        public FilteredListIterator() {



        }

        @Override
        public boolean hasNext() {




        }

        @Override
        public T next() {
```

```
        }
```

```
    }
}
```

Here is a video walkthrough of the solutions.

```
import java.util.*;

class FilteredList<T> implements Iterable<T> {
    List<T> list;
    Predicate<T> pred;

    public FilteredList(List<T> L, Predicate<T> filter) {
        this.list = L;
        this.pred = filter;
    }

    public Iterator<T> iterator() {
        return new FilteredListIterator();
    }

    private class FilteredListIterator implements Iterator<T> {
        int index;

        public FilteredListIterator() {
            index = 0;
            moveIndex();
        }

        @Override
        public boolean hasNext() {
            return index < list.size();
        }

        @Override
        public T next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            T answer = list.get(index);
            index += 1;
            moveIndex();
            return answer;
        }
        private void moveIndex() {
            while (hasNext() && !pred.test(list.get(index))) {
                index += 1;
            }
        }
    }
}
```

Alternate Solution: Although this solution provides the right functionality, it is not as efficient as the first one. Imagine you only want the first couple items from the iterable. Is it worth processing the entire list in the constructor? It is not ideal in the case that our list is millions of elements long. The first solution is different in that we "lazily" evaluate the list, only progressing our index on every call to `next` and `hasNext`. However, this solution may be easier to digest.

```
import java.util.*;

class FilteredList<T> implements Iterable<T> {
    List<T> list;
    Predicate<T> pred;

    public FilteredList(List<T> L, Predicate<T> filter) {
        this.list = L;
        this.pred = filter;
    }

    public Iterator<T> iterator() {
        return new FilteredListIterator();
    }

    private class FilteredListIterator implements Iterator<T> {
        LinkedList<T> items;

        public FilteredListIterator() {
            items = new LinkedList<>();
            for (T item: list) {
                if (pred.test(item)) {
                    items.add(item);
                }
            }
        }

        @Override
        public boolean hasNext() {
            return !items.isEmpty();
        }

        @Override
        public T next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            return items.removeFirst();
        }
    }
}
```

4 A Tree Takes on Graphs

Your friend at Stanford has come to you for help on their homework! For each of the following statements, determine whether they are true or false; if false, provide counterexamples.

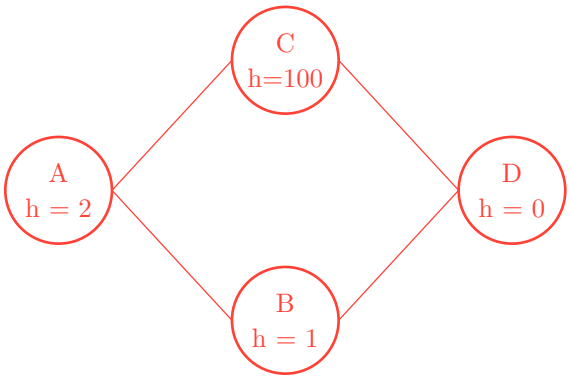
(a) "A graph with edges that all have the same weight will always have multiple MSTs."

Solution: False: Consider a tree (N nodes and N - 1 edges, no cycles, connected) - there is only one way to connect a tree, so it will be its own MST (so there is only one MST for a tree).

(b) "No matter what heuristic you use, A* search will always find the correct shortest path."

Solution: False: Here, A* would incorrectly return A - B - D as the shortest path from A to D. Starting at A, we would add B to the queue with priority 1+1 (the known distance to B, as well as our estimated distance from B to the goal), and we would add C to the queue with priority 1+100 (the known distance to C, as well as our estimated distance from C to the goal). We then pop B off the queue, and add D to the queue with priority 11+0 (the known distance to D, as well as the estimated distance from D to the goal). Our queue now contains C with priority 101, and D with priority 11, so we pop D off the queue and complete our search, returning A - B - D as the shortest path instead of the correct answer: A - C - D.

In general, A* is only guaranteed to be correct if the heuristic is good-specifically, it should be both admissible and consistent (note that applying admissibility and consistency are out of scope for this class, you only need to know the definition). In the example given, our heuristic is neither admissible nor consistent.



(c) "If you add a constant factor to each edge in a graph, Dijkstra's algorithm will return the same shortest paths tree."

False: this can be disproved with the example below, where we add a constant $c = 2$ to every edge. Adding a constant factor per edge will disadvantage paths with more edges. In our example, though A - B - C had more edges, in our original graph it still has shorter total path cost, at $1 + 1 = 2$. On the other hand, A - C had fewer edges but larger total path cost, 3. After adding a constant factor, however, the A - B - C path cost was $(1 + 2) + (1 + 2) = 6$ and the A - C had a path cost of $(2 + 3) = 5$. So before the addition, Dijkstra's shortest path tree would have said the shortest path from A to C was A - B - C, but afterwards it would say A - C.