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);
}
```

(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);
}
```

(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:

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

Worst Case:

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

This page was intentionally left blank.

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) {

  __________<CityPair> allCombinations = ________________________________;


  for (____________________ : ____________________) {

    for (____________________ : ____________________) {

      if (____________________) {

        allCombinations.add(__________________);
      }
    }
  }


  __________<CityPair> orderedCombinations = new ________________(____________________);


  return orderedCombinations.sort(________________________)
}

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

  return ________________________________________________________________________;
}


public class DistanceComparator implements ____________________<CityPair> {

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

    return ______________________________________________________;
  }
}
```

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> ______________________________ {


    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() {










        }








    }
}
```

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."

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

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