1. List’em all!

List all the asymptotic runtimes from quickest to slowest.

$\theta(n^2), \theta(n^{0.5}), \theta(\log n), \theta(3^n), \theta(c), \theta(n), \theta(n!)$

**Solution:** $\theta(c), \theta(\log n), \theta(n^{0.5}), \theta(n), \theta(n!)$

2. What’s that runtime?

For each of the methods below, please specify the runtime in BigO, Big$\Theta$ or Big$\Omega$ Notation. Please give the tightest bound possible.

**Solution:** $\theta(n^3)$, the $i$ loop runs $n$, the $j$ loop runs $n$, each time doing a function that takes $n$, so $n \cdot n \cdot n = n^3$.

```java
private static void f(int n) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            linear(n); // runs in linear time with respect to input
        }
    }
}
```

**Solution:** $\theta(n \log n)$,

```java
private static void g(int n) {
    if (n < 1) return;
    for(int i = 0; i < n; i++) {
        linear(100);
    }
    g(n/2);
    g(n/2);
}
```
Solution: $O(n)$

```java
public static void h(int n) {
    Random generator = new Random();
    for(int i = 0; i < n; i++) {
        if(generator.nextBoolean()) {
            /* nextBoolean returns true with probability .5. */
            break;
        }
    }
}
```

3 How fast?

Given a IntList of length $N$, provide the runtime bound for each operation. Recall that IntList is the naive linked list implementation from class.

<table>
<thead>
<tr>
<th>Operations</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>size()</td>
<td>$\theta(N)$</td>
</tr>
<tr>
<td>get(int index)</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>addFirst(E e)</td>
<td>$\theta(1)$</td>
</tr>
<tr>
<td>addLast(E e)</td>
<td>$\theta(N)$</td>
</tr>
<tr>
<td>remove(int index)</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>remove(Node n)</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>
4 The ABCs of OOP

Indicate what each line the main program in class D would print, if the line prints anything. If any lines error out, identify the errors as compile-time or runtime errors and cross out the corresponding lines.

```java
public class A {
    public void x() { System.out.println("Ax"); }
    public void y(A z) { System.out.println("Ay"); }
}

public class B extends A {
    public void y() { System.out.println("By"); }
    public void y(B z) { System.out.println("Byz"); }
}

public class C extends A {
    public void x() { System.out.println("Cx"); }
}

public class D {
    public static void main(String[] args) {
        A e = new B();
        A f = new C();
        B g = new A();  // Solution: Compile-Time Error. A is not a subclass of B
        B h = new C();  // Solution: Compile-Time Error. Although B and C are both children classes of A, B and C are not related to each other.
        C i = (C) new A();  // Solution: Runtime Error. Casting would trick the compiler to think of the new object as type C and then assign it to i. While running the program, casting will crash because the new object is type A in dynamic binding, which cannot be assigned to class C (as A is not a subclass of C).
        B j = (A) new C();  // Solution: Compile-Time Error. Casting will trick the compiler to think of the new object as type A. However in runtime when looking at the dynamic types, we cannot assign it to type B since A is not a subclass of B.
        B k = (B) e;  // Solution: e is type B in dynamic type, so the assignment works out fine in run-time.
    }
}
```
f.x(); \textbf{Solution: } Cx

e.x(); \textbf{Solution: } Ax

e.y(); \textbf{Solution: } Compile-Time Error. \( e \) is treated as an object under class \( A \) when compiling. Class \( A \) doesn’t have the \( y \) method whose input is empty.

(B) \( e.y() \); \textbf{Solution: } Compile-Time Error. This attempts to cast the value returned by \( e.y() \) to \( B \), rather than actually casting \( e \) to type \( B \). Therefore this does nothing to fix the issue described in the above part.

((B) \( e \)).\( y() \); \textbf{Solution: } By

e.y(e); \textbf{Solution: } Ay

e.y(f); \textbf{Solution: } Ay

5 Classy Cats

Look at the \texttt{Animal} class defined below.

```java
public class Animal {
    protected String name, noise;
    protected int age;

    public Animal(String name, int age) {
        this.name = name;
        this.age = age;
        this.noise = "Huh?";
    }

    public String makeNoise() {
        if (age < 2) {
            return noise.toUpperCase();
        }
        return noise;
    }

    public String greet() {
        return name + ": " + makeNoise();
    }
}
```

(a) Given the \texttt{Animal} class, fill in the definition of the \texttt{Cat} class so that it makes a ”Meow!” noise when \texttt{greet()} is called. Assume this noise is all caps for kittens, i.e. Cats that are less than 2 years old.

\textbf{Solution:}
public class Cat extends Animal {
    public Cat(String name, int age) {
        super(name, age);
        this.noise = "Meow!";
    }
}

Solution:  Inheritance is powerful because it allows us to reuse code for related classes. With the Cat class here, we just have to re-write the constructor to get all the goodness of the Animal class.

Why is it necessary to call super(name, age); within the Cat constructor? It turns out that a subclass’s constructor by default always calls its parent class’s constructor (aka a super constructor). If we didn’t specify the call to the Animal super constructor that takes in a String and an int, we’d get a compiler error. This is because the default super constructor (super();) would have been called. Only problem is that the Animal class has no such zero-argument constructor!

By explicitly calling super(name, age); in the first line of the Cat constructor, we avoid calling the default super constructor.

Similarly, not providing any explicit constructor at all in the Cat implementation would also result in code that does not compile. This is because when there are no constructors available in a class, Java automatically inserts a no-argument constructor for you. In that no-argument constructor, Java will then attempt to call the default super constructor, which again, does not exist.

Also note that declaring a noise field at the top of the Cat class would not be correct. Since in Java, fields are bound at compile time, when the parent class’s makeNoise() function calls upon noise, we will receive "Huh?". Because of this confusing subtlety of Java, which is called field hiding, it is generally a bad idea to have an instance variable in both a superclass and a subclass with the same name.

(b) "Animal" is an extremely broad classification, so it doesn’t really make sense to have it be a class. Look at the new definition of the Animal class below.

public abstract class Animal {
    protected String name;
    protected String noise = "Huh?";
    protected int age;

    public String makeNoise() {
        if (age < 2) {
            return "Meow!";
        }
    }
}
return noise.toUpperCase();
}
return noise;

public String greet() {
    return name + " : " + makeNoise();
}

public abstract void shout();
abstract void count(int x);

Fill out the Cat class again below to allow it to be compatible with Animal (which is now an abstract class) and its two new methods.

Solution:

class Cat extends Animal {
    public Cat() {
        this.name = "Kitty";
        this.age = 1;
        this.noise = "Meow!";
    }

    public Cat(String name, int age) {
        this();
        this.name = name;
        this.age = age;
    }

    @Override
    public void shout() {
        System.out.println(noise.toUpperCase());
    }

    @Override
    void count(int x) {
        for(int i = 0; i < x; i++) {
            System.out.println(makeNoise());
        }
    }
}

Solution: To override an abstract method, the method signature’s access modifiers must match exactly. Since shout is declared to be public abstract in Animal, our Cat class must declare it to be public to ensure that access modifiers match. The default access modifier for abstract classes is the same as the default access modifier for regular Java classes. Since count has the default access modifier in the Animal abstract class, count has the default access modifier when we override it in the Cat class.