INFO1113 Object-Oriented Programming

Week 9A: Recursion
Recursive methods and caching

Topics

- Recursion (s. 4)
- Recursion with OOP (s. 17)
- Memoization (Caching Results) (s. 27)

Recursion

Recursion is a technique within computer science that allows calling a function within itself.

Recursive functions are aligned with recursive sequences or series. Where the output of a function is dependent on the output from the same function with a change of input.

Refer to Chapter 11.1 The Basics Of Recursion, p 856-876 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Recursion

Recursive function is made of

- A base case (or many base cases). Where the function terminates.
- Recursive case (or many recursive cases). Which will converge to a base case.

Refer to Chapter 11.1 The Basics Of Recursion, p 856-876 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Recursion

Problems can often been represented easier with recursion. However, we are able to translate any recursive function to an iterative counterpart.

Drawbacks from recursion

- The java programming model does not allow for **infinite** recursion.
- Inefficient with memory
- Potentially more computationally demanding due to the overhead caused by method calls.

Refer to Chapter 11.2 Programming With Recursion, p. 876-890 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Let's examine the following

```java
public class Fibonacci {

    public int[] generateSequence(int n) {

        if(n < 0) {
            return new int[0];
        } else if(n == 0) {
            return new int[] {0};
        } else if(n == 1) {
            return new int[] {0, 1};
        } else {
            int[] f1 = generateSequence(n-1);
            int[] f2 = generateSequence(n-2);
            int[] newF = new int[f1.length+1];
            newF[newF.length-1] = f1[f1.length-1] + f2[f2.length-1];
            for(int i = 0; i < f1.length; i++) {
                newF[i] = f1[i];
            }
            return newF;
        }
    }
}
```

Let's examine the following

```
public class Fibonacci {

    public int[] generateSequence(int n) {

        if(n < 0) {
            return new int[0];
        } else if(n == 0) {
            return new int[] {0};
        } else if(n == 1) {
            return new int[] {0, 1};
        } else {
            int[] f1 = generateSequence(n-1);
            int[] f2 = generateSequence(n-2);
            int[] newF = new int[f1.length+1];
            newF[newF.length-1] = f1[f1.length-1] + f2[f2.length-1];
            for(int i = 0; i < f1.length; i++) {
                newF[i] = f1[i];
            }
            return newF;
        }
    }
}
```

generateSequence is a recursive method that takes in an integer.

Let's examine the following

```java
public class Fibonacci {

    public int[] generateSequence(int n) {

        if(n < 0) {
            return new int[0];
        } else if(n == 0) {
            return new int[] {0};
        } else if(n == 1) {
            return new int[] {0, 1};
        } else {
            int[] f1 = generateSequence(n-1);
            int[] f2 = generateSequence(n-2);
            int[] newF = new int[f1.length+1];
            newF[newF.length-1] = f1[f1.length-1] + f2[f2.length-1];
            for(int i = 0; i < f1.length; i++) {
                newF[i] = f1[i];
            }
            return newF;
        }
    }
}
```

Our base cases

Let's examine the following

```java
public class Fibonacci {

    public int[] generateSequence(int n) {

        if(n < 0) {
            return new int[0];
        } else if(n == 0) {
            return new int[] {0};
        } else if(n == 1) {
            return new int[] {0, 1};
        } else {
            int[] f1 = generateSequence(n-1);
            int[] f2 = generateSequence(n-2);
            int[] newF = new int[f1.length+1];
            newF[newF.length-1] = f1[f1.length-1] + f2[f2.length-1];
            for(int i = 0; i < f1.length; i++) {
                newF[i] = f1[i];
            }
            return newF;
        }
    }
}
```

With the base cases we simply return an element to the caller.

Recursion

Let's examine the following

```java
public class Fibonacci {

    public int[] generateSequence(int n) {

        if(n < 0) {
            return new int[0];
        } else if(n == 0) {
            return new int[] {0};
        } else if(n == 1) {
            return new int[] {0, 1};
        } else {
            int[] f1 = generateSequence(n-1);
            int[] f2 = generateSequence(n-2);
            int[] newF = new int[f1.length+1];
            newF[newF.length-1] = f1[f1.length-1] + f2[f2.length-1];
            for(int i = 0; i < f1.length; i++) {
                newF[i] = f1[i];
            }
            return newF;
        }
    }
}
```

Our recursive case

Let's examine the following

```
public class Fibonacci {

    public int[] generateSequence(int n) {

        if(n < 0) {
            return new int[0];
        } else if(n == 0) {
            return new int[] {0};
        } else if(n == 1) {
            return new int[] {0, 1};
        } else {
            int[] f1 = generateSequence(n-1);
            int[] f2 = generateSequence(n-2);
            int[] newF = new int[f1.length+1];
            newF[newF.length-1] = f1[f1.length-1] + f2[f2.length-1];
            for(int i = 0; i < f1.length; i++) {
                newF[i] = f1[i];
            }
            return newF;
        }
    }
}
```

In this instance, the recursive method calls the same method with a change of input. **N-1 and N-2**

Let's demo this!

This sounds horrible! Why would we write it as such?

A few reasons

- Recursive methods can be simpler to read and write.
- Immediately writing an iterative method where there is an established recursive method can be considered a **premature optimisation**.
- Other systems that afford recursion, do not suffer the same limitations as Java.

Recursion with OOP

To extend from regular recursion we are able to utilise objects with recursion.

We are able to write the recursive method for class instances.

For example, within an instance object of type **FamilyMember** we could invoke, **getChildren()** which may call the same method on all **FamilyMember** objects that are children of the callee.

Recursion with OOP

This kind of recursion is common with linked data structures such as.

- Trees
- Linked List
- Graphs
- Stacks
- Queues
- Heaps

Recursion with OOP

Not only are we writing recursive methods we are writing them for instances and generalising the usage for all instances.

For any given instance, we can apply this method but we are able to extend this with the use of polymorphism.

Recursion with OOP

Let's examine the following:

```
class FamilyMember {

    private String name;
    private int age;
    List<FamilyMember> children;

    public FamilyMember(String name, int age) {
        this.name = name;
        this.age = age;
        children = new ArrayList<FamilyMember>();
    }

    // Snipped

    public List<FamilyMember> getAllParents() {
        List<FamilyMember> parents = new ArrayList<FamilyMember>();
        if(this.getChildren().size() > 0) {
            parents.add(this);
            for(int i = 0; i < this.getChildren().size(); i++) {
                parents.addAll(this.getChildren().get(i).getAllParents());
            }
        }
        return parents;
    }
}
```

Recursion with OOP

Let's examine the following:

```java
class FamilyMember {

    private String name;
    private int age;
    List<FamilyMember> children;

    public FamilyMember(String name, int age) {
        this.name = name;
        this.age = age;
        children = new ArrayList<FamilyMember>();
    }

    // Snipped

    public List<FamilyMember> getAllParents() {
        List<FamilyMember> parents = new ArrayList<FamilyMember>();
        if(this.getChildren().size() > 0) {
            parents.add(this);
            for(int i = 0; i < this.getChildren().size(); i++) {
                parents.addAll(this.getChildren().get(i).getAllParents());
            }
        }
        return parents;
    }
}
```

Normal class with name and age as per the requirements.

Recursion with OOP

Let's examine the following:

```java
class FamilyMember {

    private String name;
    private int age;
    List<FamilyMember> children;

    public FamilyMember(String name, int age) {
        this.name = name;
        this.age = age;
        children = new ArrayList<FamilyMember>();
    }

    // Snipped

    public List<FamilyMember> getAllParents() {
        List<FamilyMember> parents = new ArrayList<FamilyMember>();
        if(this.getChildren().size() > 0) {
            parents.add(this);
            for(int i = 0; i < this.getChildren().size(); i++) {
                parents.addAll(this.getChildren().get(i).getAllParents());
            }
        }
        return parents;
    }
}
```

We contain a list of **children** or in a more abstract sense, **links**.

Recursion with OOP

Let's examine the following:

```java
class FamilyMember {

    private String name;
    private int age;
    List<FamilyMember> children;

    public FamilyMember(String name, int age) {
        this.name = name;
        this.age = age;
        children = new ArrayList<FamilyMember>();
    }

    // Snipped

    public List<FamilyMember> getAllParents() {
        List<FamilyMember> parents = new ArrayList<FamilyMember>();
        if(this.getChildren().size() > 0) {
            parents.add(this);
            for(int i = 0; i < this.getChildren().size(); i++) {
                parents.addAll(this.getChildren().get(i).getAllParents());
            }
        }
        return parents;
    }
}
```

Each FamilyMember contain a list of children, when retrieving a list of parents from a **FamilyMember** we will need to check each link if they are also a parent.

Let's examine the following:

```
class FamilyMember {

    private String name;
    private int age;
    List<FamilyMember> children;

    public FamilyMember(String name, int age) {
        this.name = name;
        this.age = age;
        children = new ArrayList<FamilyMember>();
    }

    // Snipped

    public List<FamilyMember> getAllParents() {
        List<FamilyMember> parents = new ArrayList<FamilyMember>();
        if(this.getChildren().size() > 0) {
            parents.add(this);
            for(int i = 0; i < this.getChildren().size(); i++) {
                parents.addAll(this.getChildren().get(i).getAllParents());
            }
        }
        return parents;
    }
}
```

Since each child is a **FamilyMember** type, we are able to call the method **getAllParents()** recursively, since the method adds all the elements to a list we are able to add it to the caller's list.

After all… It is an OOP course

Let’s demo this!

Memoization

Memoization is a technique for storing the results of a computation.

You may know it as a different term: ***Caching***

We keep the result as we may want to reuse it later.

Let's say we had a website that computes a simple page, the page doesn't differ between each user accessing, we could keep the result and send it every time it is asked.

So what does recursion and memoization have to do with each other?

Memoization

Recursive calls can be computationally **expensive** and if we are repeatedly calling dependent values or the same values it makes sense to keep a record of that.

Simply, we are maintaining a copy of the answer because other computations depend on it.

Memoization

Welcome back to the fibonacci program!

```java
public class Fibonacci {

    public int[] generateSequence(int n) {

        if(n < 0) {
            return new int[0];
        } else if(n == 0) {
            return new int[] {0};
        } else if(n == 1) {
            return new int[] {0, 1};
        } else {
            int[] f1 = generateSequence(n-1);
            int[] f2 = generateSequence(n-2);
            int[] newF = new int[f1.length+1];
            newF[newF.length-1] = f1[f1.length-1] + f2[f2.length-1];
            for(int i = 0; i < f1.length; i++) {
                newF[i] = f1[i];
            }
            return newF;
        }
    }
}
```

Okay, so how can to give it a cache?

Memoization

Let's consider these following calls

```
public static void main(String[] args) {
        Fibonacci f = new Fibonacci();
        System.out.println(Arrays.toString(f.generateSequence(0)));
        System.out.println(Arrays.toString(f.generateSequence(1)));
        System.out.println(Arrays.toString(f.generateSequence(2)));
        System.out.println(Arrays.toString(f.generateSequence(4)));
        System.out.println(Arrays.toString(f.generateSequence(4)));
        System.out.println(Arrays.toString(f.generateSequence(8)));
        System.out.println(Arrays.toString(f.generateSequence(8)));
        System.out.println(Arrays.toString(f.generateSequence(12)));
    }
```

This simply calls the base cases, it will return {0} and {0, 1}

Memoization

Let's consider these following calls

```
public static void main(String[] args) {
        Fibonacci f = new Fibonacci();
        System.out.println(Arrays.toString(f.generateSequence(0)));
        System.out.println(Arrays.toString(f.generateSequence(1)));
        System.out.println(Arrays.toString(f.generateSequence(2)));
        System.out.println(Arrays.toString(f.generateSequence(4)));
        System.out.println(Arrays.toString(f.generateSequence(4)));
        System.out.println(Arrays.toString(f.generateSequence(8)));
        System.out.println(Arrays.toString(f.generateSequence(8)));
        System.out.println(Arrays.toString(f.generateSequence(12)));
    }
```

We can see that generateSequence(2) will depend on the base case but what if we can just return an answer already computed? Even on recursive calls?

So let’s cache it?

Memoization

Welcome back to the fibonacci program!

```
public class FibonacciCache {
    private Map<Integer, int[]> cache;
    public FibonacciCache() {
        cache = new TreeMap<Integer, int[]>();
    }
    public int[] generateSequence(int n) {
        if(cache.containsKey(n)) {
            return cache.get(n);
        } else {
            if(n < 0) {
                return new int[0];
            } else if(n == 0) {
                return new int[] {0};
            } else if(n == 1) {
                return new int[] {0, 1};
            } else {
                int[] f1 = generateSequence(n-1);
                int[] f2 = generateSequence(n-2);
                int[] newF = new int[f1.length+1];
                newF[newF.length-1] = f1[f1.length-1] + f2[f2.length-1];
                for(int i = 0; i < f1.length; i++) {
                    newF[i] = f1[i];
                }
                cache.put(n, newF);
                return newF;
            }
        }
    }
}
```

Welcome back to the fibonacci program!

```
public class FibonacciCache {
    private Map<Integer, int[]> cache;
    public FibonacciCache() {
        cache = new TreeMap<Integer, int[]>();
    }
    public int[] generateSequence(int n) {
        if(cache.containsKey(n)) {
            return cache.get(n);
        } else {
            if(n < 0) {
                return new int[0];
            } else if(n == 0) {
                return new int[] {0};
            } else if(n == 1) {
                return new int[] {0, 1};
            } else {
                int[] f1 = generateSequence(n-1);
                int[] f2 = generateSequence(n-2);
                int[] newF = new int[f1.length+1];
                newF[newF.length-1] = f1[f1.length-1] + f2[f2.length-1];
                for(int i = 0; i < f1.length; i++) {
                    newF[i] = f1[i];
                }
                cache.put(n, newF);
                return newF;
            }
        }
    }
}
```

We've introduced a collection that will hold our answers. For convenience we are using a **Map**

Welcome back to the fibonacci program!

```
public class FibonacciCache {
    private Map<Integer, int[]> cache;
    public FibonacciCache() {
        cache = new TreeMap<Integer, int[]>()
    }
    public int[] generateSequence(int n) {
        if(cache.containsKey(n)) {
            return cache.get(n);
        } else {
            if(n < 0) {
                return new int[0];
            } else if(n == 0) {
                return new int[] {0};
            } else if(n == 1) {
                return new int[] {0, 1};
            } else {
                int[] f1 = generateSequence(n-1);
                int[] f2 = generateSequence(n-2);
                int[] newF = new int[f1.length+1];
                newF[newF.length-1] = f1[f1.length-1] + f2[f2.length-1];
                for(int i = 0; i < f1.length; i++) {
                    newF[i] = f1[i];
                }
                cache.put(n, newF);
                return newF;
            }
        }
    }
}
```

Constructing it with an Integer as a key (the nth fibonacci sequence) and int[] as the value.

Welcome back to the fibonacci program!

```
public class FibonacciCache {
    private Map<Integer, int[]> cache;
    public FibonacciCache() {
        cache = new TreeMap<Integer, int[]>();
    }
    public int[] generateSequence(int n) {
        if(cache.containsKey(n)) {
            return cache.get(n);
        } else {
            if(n < 0) {
                return new int[0];
            } else if(n == 0) {
                return new int[] {0};
            } else if(n == 1) {
                return new int[] {0, 1};
            } else {
                int[] f1 = generateSequence(n-1);
                int[] f2 = generateSequence(n-2);
                int[] newF = new int[f1.length+1];
                newF[newF.length-1] = f1[f1.length-1] + f2[f2.length-1];
                for(int i = 0; i < f1.length; i++) {
                    newF[i] = f1[i];
                }
                cache.put(n, newF);
                return newF;
            }
        }
    }
}
```

We have added a check to see if we have already computed this answer before, if we have we simply return it!

Let’s demo this!

See you next time!