Lecture 06: Recursion

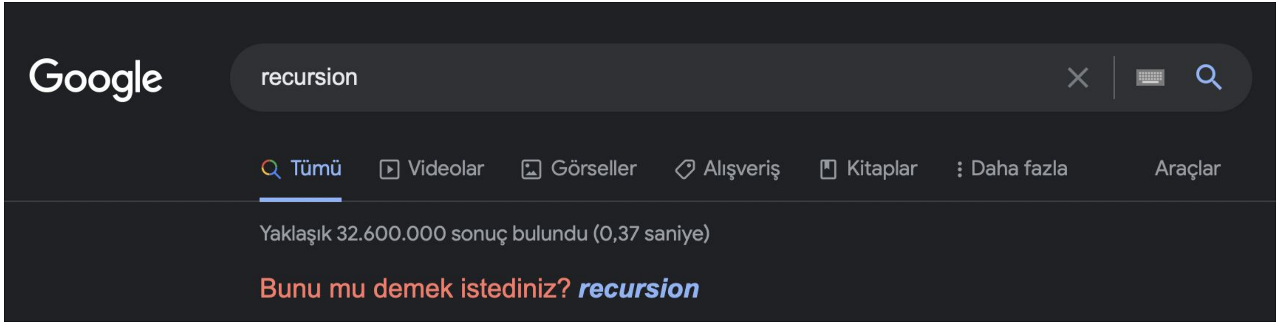
SE115: Introduction to Programming I

Previously on SE 115…

- Functions
- Scope

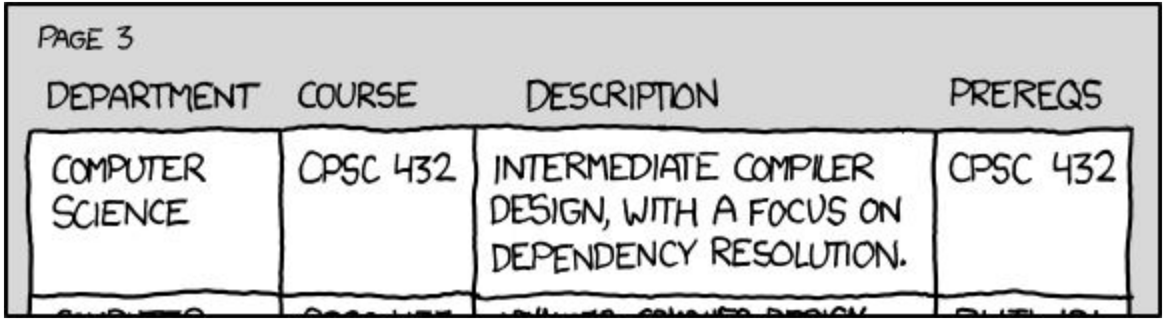
Recursion

- Let's Google it.



So many jokes about recursion!

Reference: https://xkcd.com/754/



```java
public class Recursive {
  public static void recurse() {
    System.out.println("Here we go again…");
    recurse();
  }
}
```

Recursion

- If a function calls itself, it is recursive.
- Check above. What would happen if we call the **recurse()** function?
- The function will print “here we go again”,
- It will call **recurse**,
  - The function will print “here we go again”,
  - It will call **recurse**,
    - The function will print “here we go again”,
    - It will call **recurse**,
      - …
- Quick question: What happens to the call stack?

```java
public class Recursive {
  public static void recurse() {
    System.out.println("Here we go again…");
    recurse();
  }
}
```

Recursion

- Is the **recurse()** function an infinite loop?
- Since it did not have a condition for NOT calling itself, it kept on doing so for infinity.
- Let's make it stop.
- We need a variable to get the number of recursions.
- This variable can be passed as a parameter.

Recursion

```
public class BasicRecurse {
    public static void recurse(int c) {
        if (c != 3) {
            System.out.println("hello");
            recurse(c + 1);
        }
        System.out.println("The function with c = " + c + " is complete");
    }

    public static void main(String args[]) {
        recurse(0);
    }
}
```

The call stack!

c=3

```
public static void recurse(int c) {
    if (c != 3) {
        System.out.println("hello");
        recurse(c + 1);
    }
    System.out.println("The function with c = " + c + " is complete");
}
```

Stopping condition! c=3 so no more recursion!

returns to the line it has been called from…

c=2

```
public static void recurse(int c) {
    if (c != 3) {
        System.out.println("hello");
        recurse(c + 1);
    }
    System.out.println("The function with c = " + c + " is complete");
}
```

returns to the line it has been called from…

c=1

```
public static void recurse(int c) {
    if (c != 3) {
        System.out.println("hello");
        recurse(c + 1);
    }
    System.out.println("The function with c = " + c + " is complete");
}
```

returns to the line it has been called from…

c=0

```
public static void recurse(int c) {
    if (c != 3) {
        System.out.println("hello");
        recurse(c + 1);
    }
    System.out.println("The function with c = " + c + " is complete");
}
```

```
public static void main(String args[]) {
    recurse(0);
}
```

we are back at main.

Recursion

- So, a function can call itself!
- It is much better if I say: **“it calls a copy of itself”**
- This is required in many cases, and it is not just for simple “looping”
- Think about comment threads in a social application.
- There is a comment.
    - A reply to that comment,
    - another reply,
        - a reply to another reply,
            - a reply to “a reply to another reply”
                - …
        - a reply to another reply…

Recursion

- **Recursion is not an alternative to looping.**
  - Each function call is a context switch, and it is not free.
- Use a loop if it solves your problem.

- Also, please pay attention to the call stack; a function returns to the point it was called from.
- Even though the call stack grows, we want to get back to the main eventually, so that we can exit.

- Let’s go over a few examples.

Summation

- Whenever I see sigma, I see a for loop.

$$\sum_{i=1}^{N} i = 1 + 2 + 3 + \cdots + (N-1) + N$$

Summation

```
int sum = 0;
for (int i=1;i<=N;i++) {
  sum += i;
}
```

$$\sum_{i=1}^{N} i = 1 + 2 + 3 + \cdots + (N-1) + N$$

But let’s make it a function!

Summation

$$\sum_{i=1}^{N} i = 1 + 2 + 3 + \cdots + (N-1) + N$$

- The summation starts from i=1 and continues until i=N.
- The variable N can be a parameter to our function.
- Since it is summation with integer values, the return type of our function should be **int**.
- We can write a loop (for or while) and add values to find the sum at each step.
- Then finally, we can **return** the sum.

Iterative Summation

```java
public class IterativeSum {
    public static int summation(int N) {
        int sum = 0;
        for(int i=1;i<=N;i++) {
            sum += i;
        }
        return sum;
    }

    public static void main(String[] args) {
        System.out.println("Sum(10) = " + summation(10));
    }
}
```

Recursive Summation

- To make the function recursive, we have to write the mathematical function in terms of itself as well.

$$\sum_{i=1}^{N} i = N + \sum_{i=1}^{N-1} i$$

- The function is f(x) = x + f(x-1)
- We have to provide the base case otherwise the function can never calculate f(x-1).
- So, f(1) = 1 is given in the definition.

Recursive Summation

- To calculate f(4) = 4 + f(3), we need f(3)
  - to calculate f(3) = 3 + f(2), we need, f(2)
    - to calculate f(2) = 2 + f(1), we need f(1), which is the base case. So f(1) returns 1.
  - f(3) becomes f(3) = 3 + f(2) = 3 + 3 = 6.
- f(4) = 4 + f(3) = 4 + 6 = 10.
- Let’s write the function.

Recursive Summation

```java
public class RecursiveSum {
    public static int summation(int N) { // assuming N >= 1
        if(N == 1) { // base condition
            return 1;
        } else { // recursive condition
            return N + summation(N-1);
        }
    }

    public static void main(String[] args) {
        System.out.println("Sum(10) = " + summation(10));
    }
}
```

N=3

```
public static int summation(int N) {
    if(N == 1) { // base condition
        return 1;
    } else { // recursive condition
        return N + summation(N-1);
    }
}
```

N=2

```
public static int summation(int N) {
    if(N == 1) { // base condition
        return 1;
    } else { // recursive condition
        return N + summation(N-1);
    }
}
```

This copy of the summation function returns 1 since N=1.

N=1

```
public static int summation(int N) {
    if(N == 1) { // base condition
        return 1;
    } else { // recursive condition
        return N + summation(N-1);
    }
}
```

More Recursion

- I don’t want you to think of recursion as a form of iteration.
- Here are a few more examples.

Fibonacci Numbers

- Fibonacci Numbers are a sequence in which each number is the sum of the two preceding ones.

$$F_n = F_{n-1} + F_{n-2}$$

- The sequence start from 0 and 1.
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …
- Looking at the formula for F_n above, it is possible to write

  fib(n) = fib(n-1) + fib(n-2), where fib(0) = 0, fib(1) = 1.

Fibonacci Numbers

- The recursive function is very obvious: make a call to function fib until you get a zero or a one.

```
public class Fibonacci {
    public static int fib(int n) { // assuming n >= 0
        if (n==1) return 1; // base case
        if (n==0) return 0; // the other base case
        return fib(n-1) + fib(n-2); // recursive case
    }

    public static void main(String[] args) {
        System.out.println("f(10) = " + fib(10));
    }
}
```

Fibonacci Numbers

- Let's analyze this function, say for n=10.

```
public static int fib(int n) {
    if (n==1) return 1;
    if (n==0) return 0;
    return fib(n-1) + fib(n-2);
}
```

1 - The function fib is called for fib(n-1), but this creates a series of recursive function calls all the way down to fib(1).

```
public static int fib(int n) {
    if (n==1) return 1;
    if (n==0) return 0;
    return fib(n-1) + fib(n-2);
}
```

2 - When this copy of fib returns 1, it returns it to the copy it is called from… here

```
public static int fib(int n) {
    if (n==1) return 1;
    if (n==0) return 0;
    return fib(n-1) + fib(n-2);
}
```

3 - Now begins another set of recursive calls for fib(n-2), but is it calculating the same values again?
How many times are we calculating fib(7) for example?
Maybe recursion is not a good approach for Fibonacci numbers.

```
                      fib(7)
                    /       \
               fib(6)         fib(5)
             /       \      /       \
         fib(5)  fib(4)  fib(4)  fib(3)
        /    \   /    \ /     \ /     \
    fib(4) fib(3)fib(3)fib(2)fib(3)fib(2)fib(2)fib(1)
    / \   / \  / \   / \  / \   / \
 fib(3)fib(2)fib(2)fib(1)fib(2)fib(1)fib(1)fib(0)
 / \
fib(2)fib(1)
```

- `fib(5)` is calculated twice.
- `fib(4)` is calculated three times.
- `fib(3)` is calculated five times.
- `fib(2)` is calculated eight times.
- `fib(1)` is calculated five times.
- `fib(0)` is calculated once.

Having Fun?

just checking…

Fibonacci Numbers

- You have already worked on Fibonacci numbers in the lab.
- But, let’s go over the thought process once again.
- We need to keep track of the two previous Fibonacci numbers.
- We know that fib(0) is 0 and fib(1) is 1.
- We want to find fib(n), and we keep on adding two previous values and update them to move forward.
- Let’s call the previous numbers n1 and n2 for 1 previous and 2 previous.
- The next Fibonacci number is f = n1 + n2, but now we have to update the previous ones; n1 becomes n2, and n2 becomes f for the next number.

Here's the code

- It would have been a great midterm question.

```
public class IterativeFib {
    public static int fib(int n) { // assuming n >= 0
        int f = 0;
        int n1 = 0;
        int n2 = 1;
        if(n == 0) f = n1;
        if(n == 1) f = n2;
        for(int i=2;i<=n;i++) {
            f = n1 + n2;
            n1 = n2;
            n2 = f;
        }
        return f;
    }

    public static void main(String[] args) {
        System.out.println("fib(10) is " + fib(10));
    }
}
```

For `fib(7)`,
the loop runs 6 times (from 2 to 7 inclusive), and each Fibonacci number from `fib(0)` to `fib(7)` is calculated exactly once.

Greatest Common Divisor

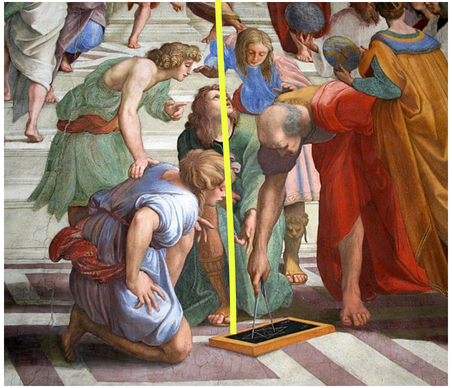
- We have to do this one, too.
- It is similar to “hello, world”: if we don’t talk about it, then we are in trouble.
- Greatest Common Divisor (gcd) (OBEB for some of us) is when we try to find the greatest number that can divide two values.
- For example, what is the gcd for 111 and 259?
- Here is a very easy approach:

$$\gcd(x, y) = \begin{cases} x & \text{if } y = 0 \\ \gcd(y, \operatorname{remainder}(x, y)) & \text{if } y > 0 \end{cases}$$

Greatest Common Divisor

$$\gcd(x, y) = \begin{cases} x & \text{if } y = 0 \\ \gcd(y, \text{remainder}(x, y)) & \text{if } y > 0 \end{cases}$$

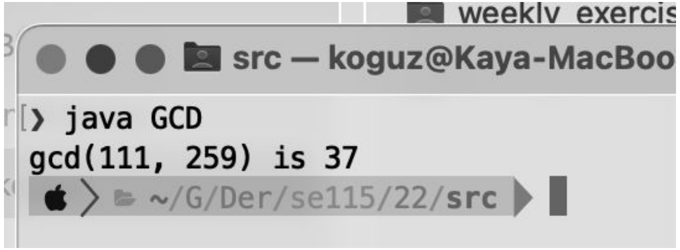
- If y (second parameter) is not 0 but greater than 0, then call gcd again, but this time the second parameter becomes the first, and the remainder of x/y becomes the second.
  - That is x % y, or x MOD y.
- Give credit where it is due: Euclid came up with this solution (no, that's not on an iPad!)
- Let's do it!



Greatest Common Divisor

```java
public class GCD {
    public static int gcd(int x, int y) {
        if (y == 0) return x;
        else return gcd(y, x%y);
    }

    public static void main(String[] args) {
        System.out.println("gcd(111, 259) is " + gcd(111,259));
    }
}
```



Some Remarks

- Recursion is not an alternative for looping!
- In parts of the code I did not write { } for some blocks.
  if (x == 0) return 1;
- If there is only one statement, then you can choose not to write { }, but be careful, it might cause errors that it hard to see.
- If we still have time…

Test yourself

- Take out a piece of paper. Try to write a recursive power function!
- The function should work as follows
    - for myRecursivePower(3, 0) it should return 1.
    - for myRecursivePower(2, 5) it should return 32.
    - for myRecursivePower(10, 6) it should return 1000000.

Homework

- Take out a piece of paper. Try to write a recursive power function!
- At home, try to re-write your function in an iterative manner.
- Considering the call stack, which of the two methods would be more efficient, the one using iterative approach or the one using recursive approach?