PROGRAMMING AND DATA STRUCTURES

RECURSION

HOURIA OUDGHIRI

FALL 2021

OUTLINE

- Recursion
- Recursive methods
- Recursion vs. Iteration
- Examples of recursion

STUDENT LEARNING OUTCOMES

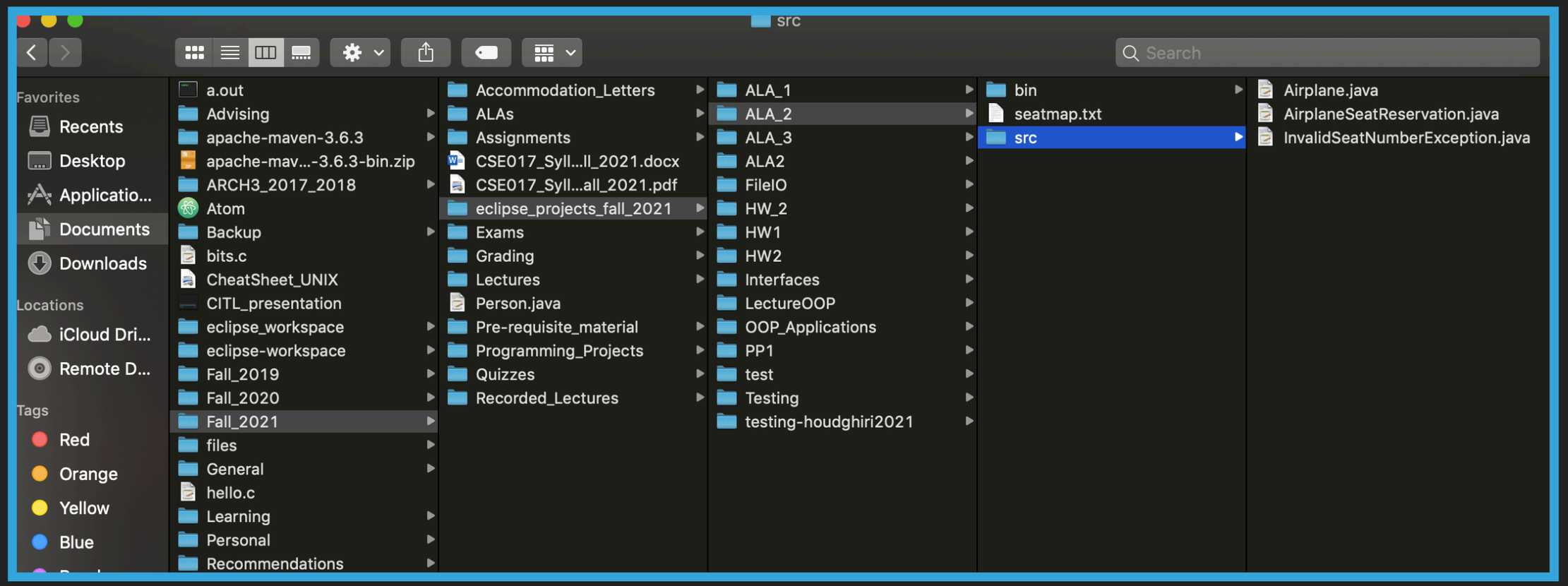
At the end of this chapter, you should be able to:

- Explain the concept of recursion and recursive method
- Describe how recursive methods are executed
- Write and test recursive methods
- Compare iterative and recursive methods

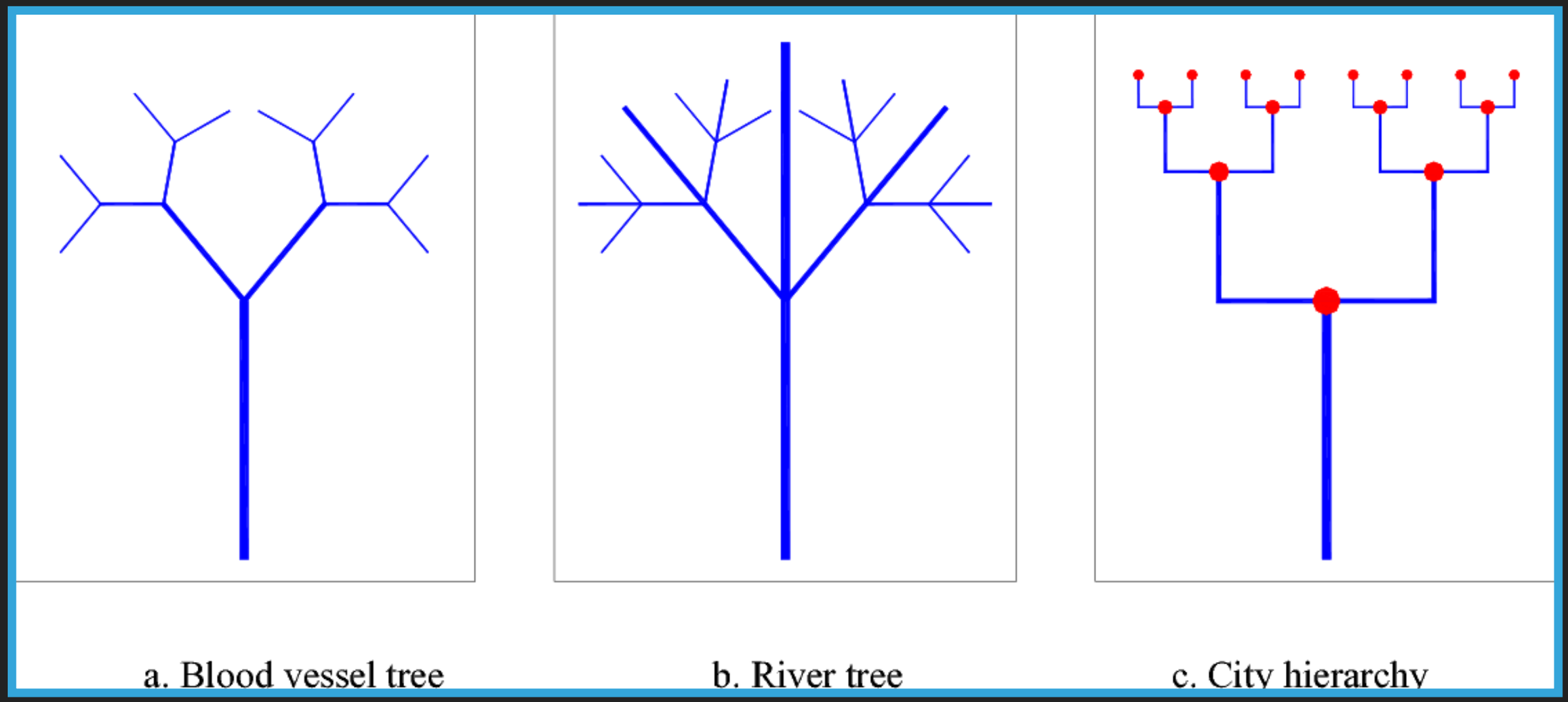
RECURSION

- Recursion is a technique to solve iterative problems that are difficult using simple loops
- Finding a file in a file hierarchy
- Drawing or traversing a tree structure

File Hierarchy



Drawing or traversing a tree structure



https://www.semanticscholar.org/paper/Fractals-and-fractal-dimension-of-systems-of-blood-Chen/

Recursive method

- Recursive method is a method that calls itself
- Example: Calculating Factorial

```
n! = n x (n-1)!
5! = 5 x 4!
4! = 4 x 3!
3! = 3 x 2!
2! = 2 x 1!
1! = 1
```

Recursive method

- Example: Calculating Factorial

Iterative Factorial

```
int factorial(int n){
    int fact=1;
    for(int i=2; i<n; i++)
       fact = I * fact;
    return fact;
}
```

Recursive method

- Example: Calculating Factorial

Iterative Factorial

```
int factorial(int n){
    int fact=1;
    for(int i=2; i<n; i++)
        fact = i * fact;
    return fact;
}
```

Recursive Factorial

```
int factorial(int n){
  if (n == 1 || n == 0)
    return 1;
  else
    return n * factorial(n-1);
}
```

Recursive method

```
public static void main(String[] args)
{
 int N = 5;
 int f = factorial(N);
}
```

```
public static int factorial(int n)
{
 if (n==1 || n==0)
   return 1;
 else
   return n * factorial(n-1);
}
```

5

Recursive method

```
int factorial(int n){ n = 5
 if (n == 1 || n == 0)
   return 1;
 else
   return n * factorial(n-1);}
```

```
int factorial(int n){ n = 4
 if (n == 1 || n == 0)
   return 1;
 else
   return n * factorial(n-1);}
```

```
int factorial(int n){ n = 3
 if (n == 1 || n == 0)
   return 1;
 else
   return n * factorial(n-1);}
```

```
int factorial(int n){ n = 2
 if (n == 1 || n == 0)
   return 1;
 else
   return n * factorial(n-1);}
```

```
int factorial(int n){ n = 1
 if (n == 1 || n == 0)
   return 1;
 else
   return n * factorial(n-1);}
```

Recursive method

```
int factorial(int n){ n = 5
 if (n == 1 || n == 0)
   return 1;
 else
   return n * factorial(n-1);}
```

24(4*6)

```
int factorial(int n){ n = 4
 if (n == 1 || n == 0)
   return 1;
 else
   return n * factorial(n-1);}
```

6 (3* 2)

```
int factorial(int n){ n = 3
 if (n == 1 || n == 0)
   return 1;
 else
   return n * factorial(n-1);}
```

2 (2 * 1)

```
int factorial(int n){ n = 2
 if (n == 1 || n == 0)
   return 1;
 else
   return n * factorial(n-1);}
```

1

```
int factorial(int n){n = 1
 if (n == 1 || n == 0)
   return 1;
 else
   return n * factorial(n-1);}
```

Recursive method

```
static void main(String[] args){
 int N = 5;
 int f = factorial(N);
}
```

120

```
int factorial(int n){n = 5
 if (n == 1 || n == 0)
   return 1;
 else
   return n * factorial(n-1);}
```

5

5*24

Recursive method

```
int factorial(int n){
  if (n == 1 || n == 0)
    return 1;
  else
    return n * factorial(n-1);
}
```

Base Case

Stopping Case

Recursion

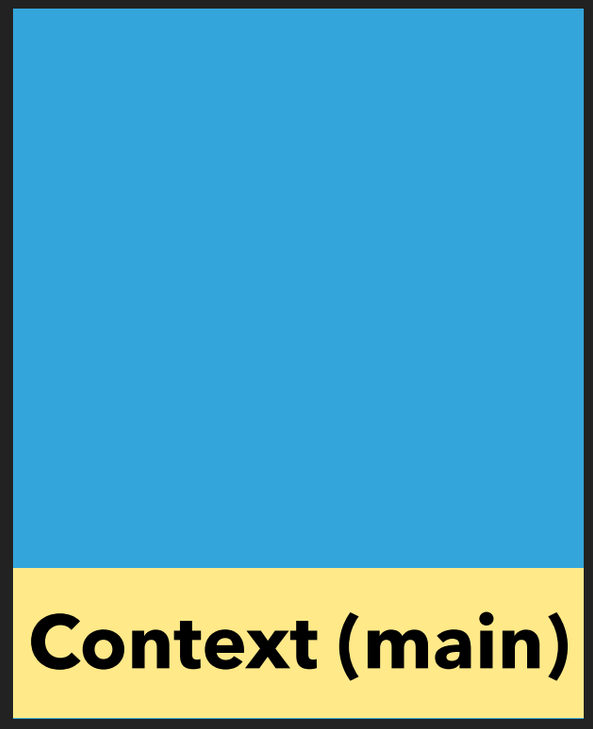
- Base Case or Stopping Case must be present
- Infinite recursion if there is no base case
- Stack overflow

Recursive method

```
public static main(String[] args){
    . . .
    int n = method1(3);
    . . .
}
```

```
public static int method1(int n){
    . . .
    method2();
    return 4;
}
```

```
public static void method2(int n){
    . . .
    return;
}
```



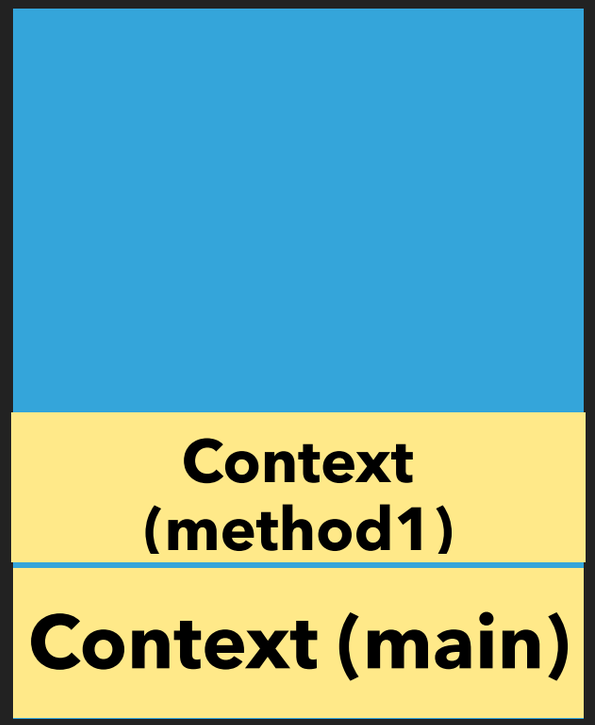
Stack

Recursive method

```
public static main(String[] args){
    . . .
    int n = method1(3);
    . . .
}
```

```
public static int method1(int n){
    . . .
    method2();
    return 4;
}
```

```
public static void method2(int n){
    . . .
    return;
}
```



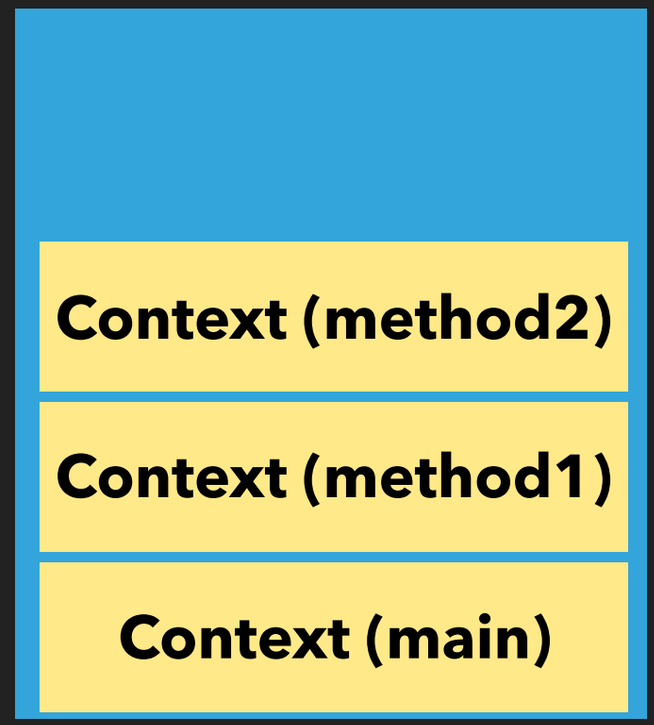
Stack

Recursive method

```
public static main(String[] args){
    . . .
    int n = method1(3);
    . . .
}
```

```
public static int method1(int n){
    . . .
    method2();
    return 4;
}
```

```
public static void method2(int n){
    . . .
    return;
}
```



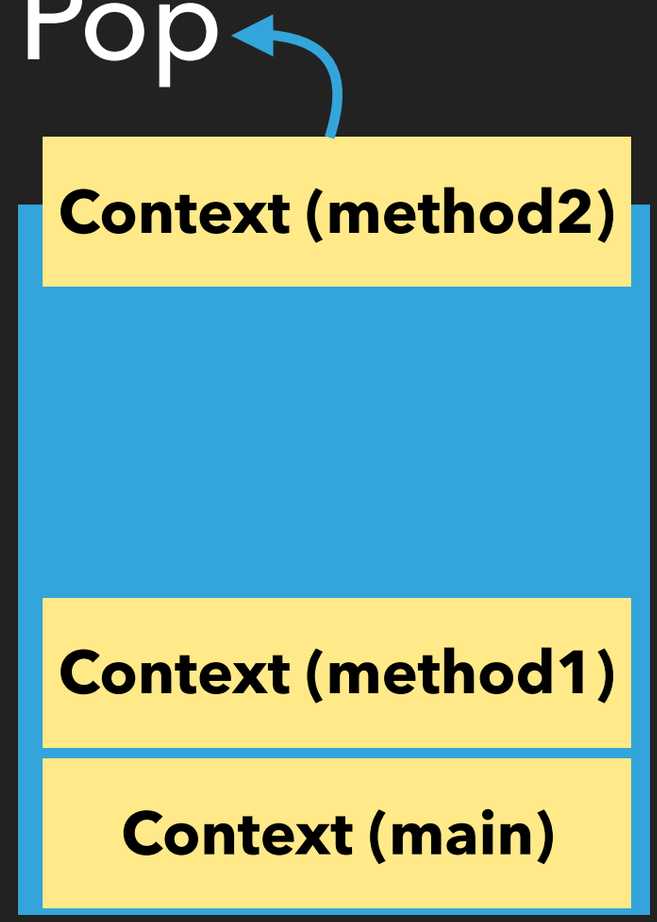
Stack

Recursive method

```
public static main(String[] args){
    . . .
    int n = method1(3);
    . . .
}
```

```
public static int method1(int n){
    . . .
    method2();
    return 4;
}
```

```
public static void method2(int n){
    . . .
    return;
}
```

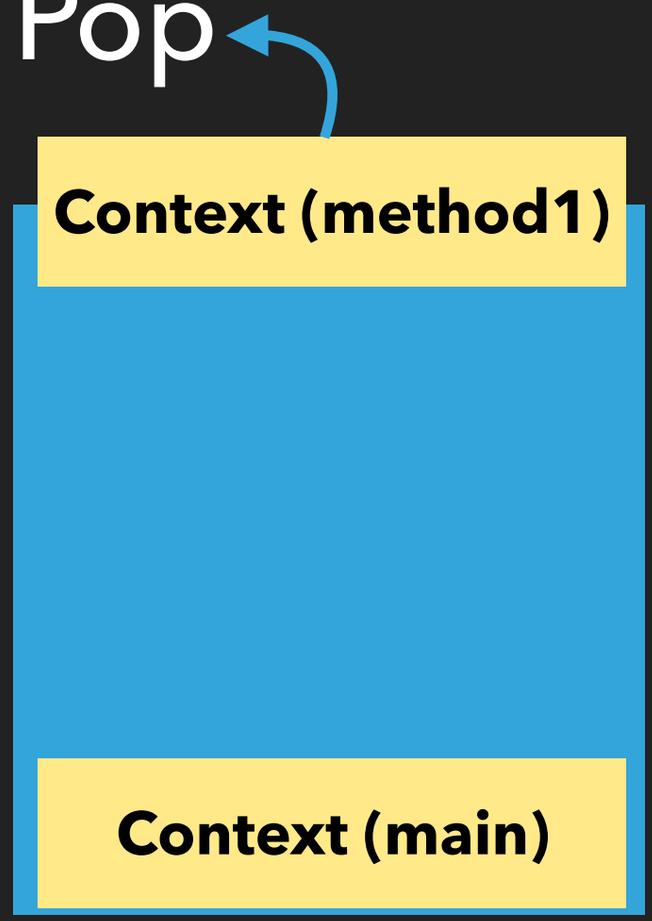


Recursive method

```
public static main(String[] args){
    . . .
    int n = method1(3);
    . . .
}
```

```
public static int method1(int n){
    . . .
    method2();
    return 4;
}
```

```
public static void method2(int n){
    . . .
    return;
}
```



Recursive method

- Stack overflow - Stack is full
- Number of calls exceeds the size of the stack
- Recursion without base case - infinite number of calls

Recursive method

- Recursive method to return the power n of a number x
- `int power(int x, int n)`
  - x^n
  - $x^n = x \,.\, x^{n-1}$

Recursive method

```
int power(int x, int n){

 if (n==0)

   return 1;

 else

   return x * power(x, n-1);

}
```

1 — Base Case will be reached, n is decremented

2 — Base case returns 1, x^0

3 — Recursion returns $x \cdot x^{n-1}$

Recursive method

- Recursive binary search
- Finding a key in an array of ordered numbers

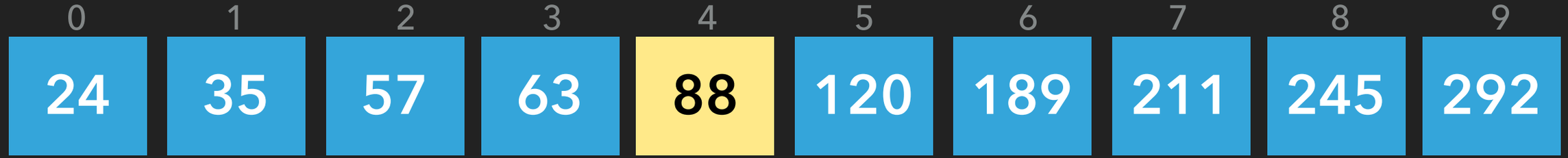
Recursive binary search

- Finding a key (number) in an ordered list of numbers
- Divide the list in halves

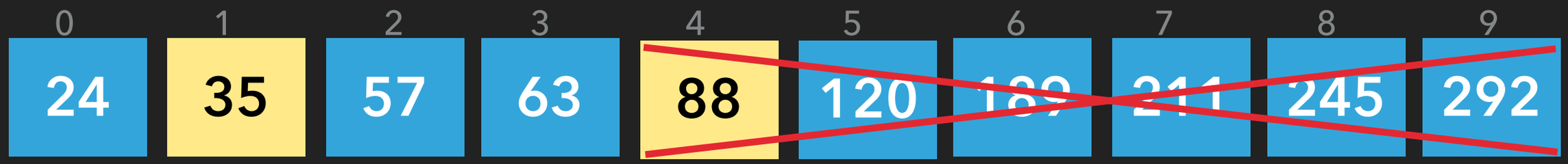
```
List = {24, 35, 57, 63, 88, 120,
        189, 211, 245, 292}
Key = 35 or key = 200
```

Recursive binary search

- List = {24, 35, 57, 63, 88, 120, 189, 211, 245, 292}
- Key = 35



Key < 88



Key = 35

Key = 35, Found at index 1

Recursive binary search

- List = {24, 35, 57, 63, 88, 120, 189, 211, 245, 292}
- Key = 200

0	1	2	3	4	5	6	7	8	9
24	35	57	63	88	120	189	211	245	292

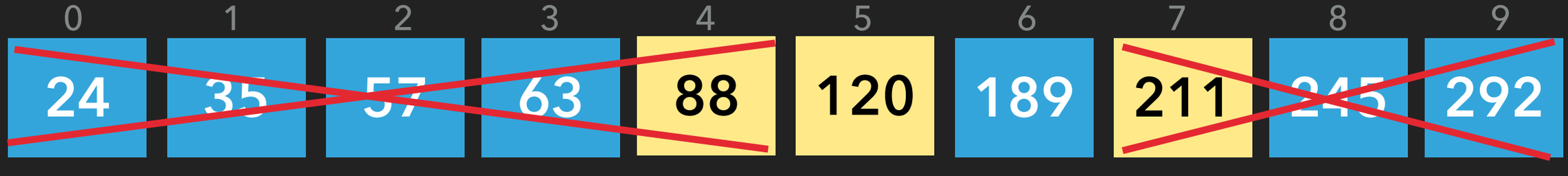
Key > 88

0	1	2	3	4	5	6	7	8	9
~~24~~	~~35~~	~~57~~	~~63~~	~~88~~	120	189	211	245	292

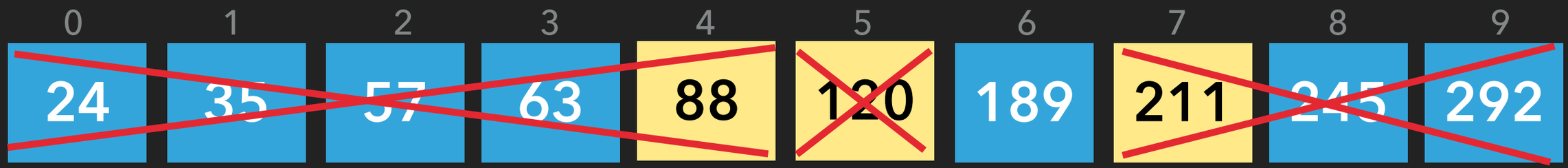
Key < 211

Recursive binary search

- List = {24, 35, 57, 63, 88, 120, 189, 211, 245, 292}
- Key = 200



Key > 120



Key > 189, Not found

Recursive binary search

Iterative Binary Search

```
int binarySearch(int[] list, int key){
 int first, last, middle;
 first = 0;
 last = list.length-1;
 while (first <= last){
    middle = (last + first) / 2;
    if (key == list[middle]) return middle;
    else if (key < list[middle])
        last = middle - 1;
    else
        first = middle + 1;
 }
 return -1;
}
```

Recursive binary search

```
int binarySearch(int[] list, int key){

 int first = 0;

 int last = list.length-1;

 return binarySearch(list, first, last, key);
}
```

Helper Method that is recursive

Recursive binary search

```
int binarySearch(int[] list,int first,int last,int key){
 if (first > last) return -1;
 else{
     int middle = (last + first) / 2;
     if (key == list[middle]) return middle;
     else if (key < list[middle])
        last = middle - 1;
     else
         first = middle + 1;

     return binarySearch(list, first, last, key);
  }
}
```

Recursive Binary Search

Recursion vs. Iteration

- Recursion usually requires less code
- Recursion reflects the divide-and-conquer strategy for solving a problem
- Recursion requires consecutive calls to the same function (context switching - stack push/pop operations)
- Iterations are preferred by compilers
- Iterations may be more efficient (computationally)

Recursion vs. Iteration

- Example: Fibonacci Series
  - Used to model rabbit population growth
  - $F_n = F_{n-1} + F_{n-2}, n > 0$

$$F_2 = 1, F_1 = 1$$

- 1 1 2 3 5 8 13 (f7 = 13)

Recursion vs. Iteration

- Fibonacci Series

Iterative

```
int fibonacci(int n){
 int f1=1, f2=1, f=0;
 if (n <= 2)
  return 1;
 else{
   while (n > 0){
      f = f1 + f2;
      f1 = f2;
      f2 = f;
      n = n -1;
   }
 }
 return f;
}
```

Recursion vs. Iteration

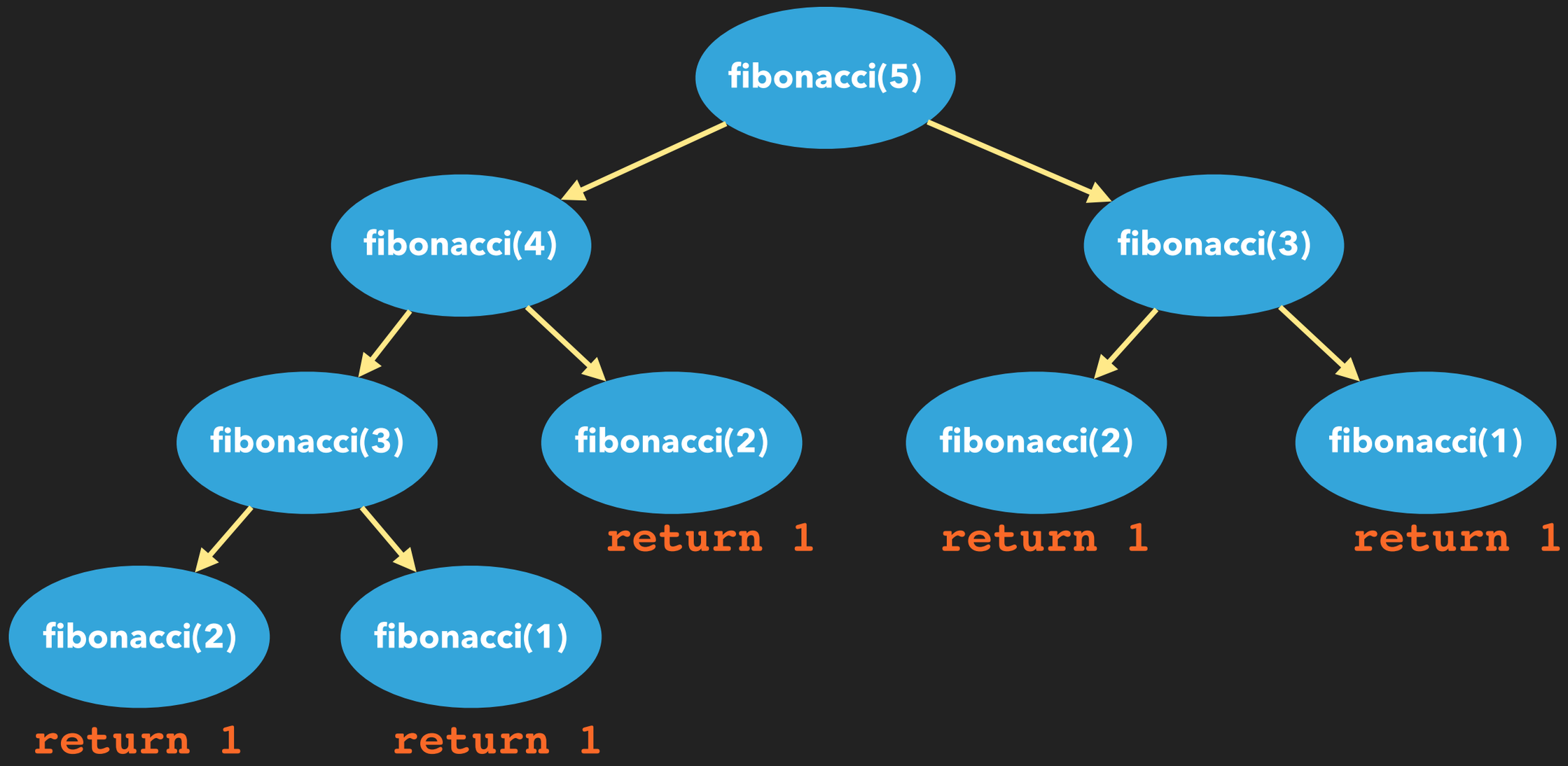
- Fibonacci Series

Recursive

```
int fibonacci(int n){

 if (n <= 2)

  return 1;

 else

   return fibonacci(n-1) + fibonacci(n-2);
}
```

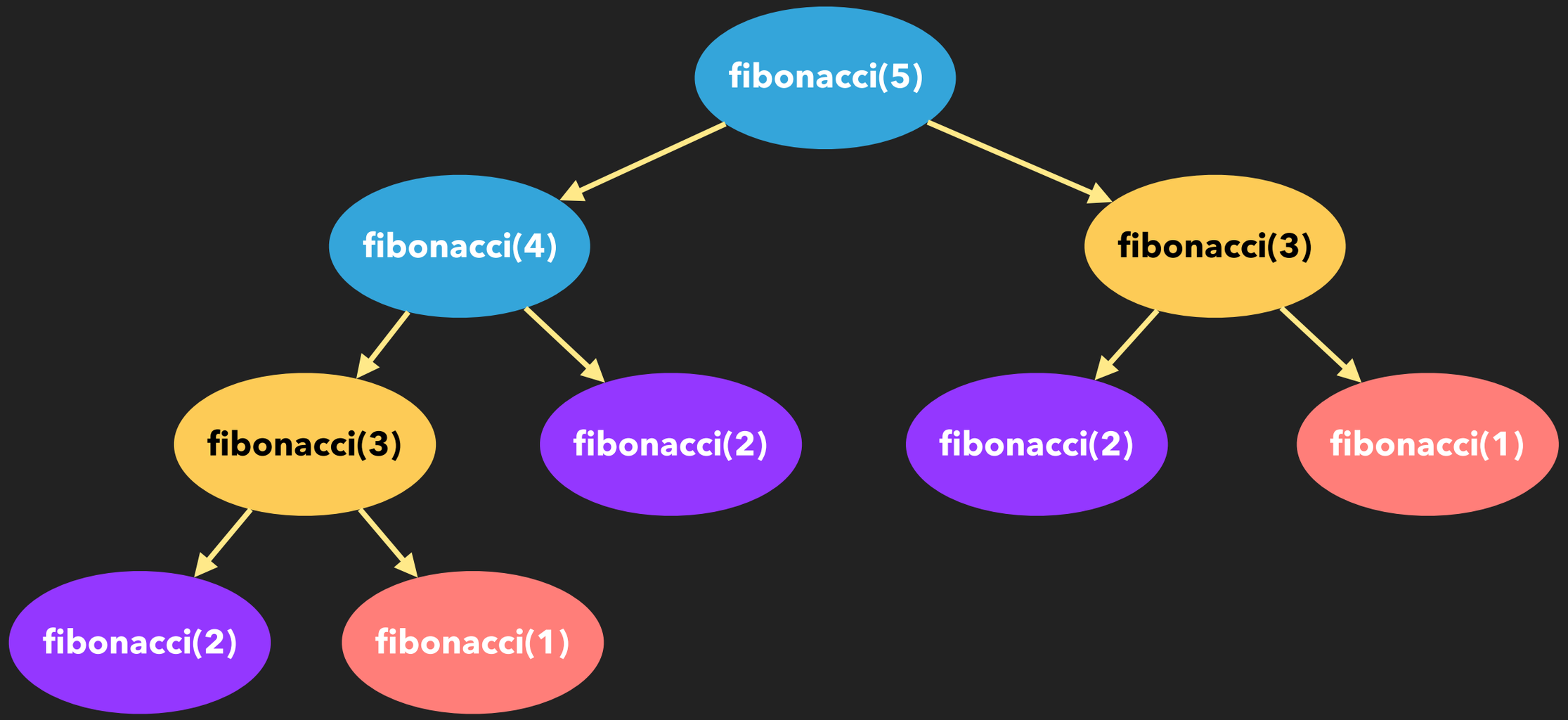
Recursion vs. Iteration

- Fibonacci Series - **`fibonacci(5)`**



Recursion vs. Iteration

- Fibonacci Series - **`fibonacci(5)`**



Recursion vs. Iteration

- Recursion usually results in compact code
- Recursion may be computationally less efficient than iteration
- Use recursion only when the problem is hard to solve using loops

Recursion - Practice

Finding a file in the file system

Example: "**`students.txt`**" is found at

"**`C:\JavaPrograms\Practices\FileIO\students.txt`**"

```
C:\
  JavaPrograms\
    Homework\
       HW#1\
          Test.java
    Practices\
       FileIO\
          students.txt
          FileIO.java
    Writing\
       TermPaper.docx
       LabReport.xlsx
```

Recursion - Practice

```
searchForFile(startPath, filename){
 if (startPath is not a directory)
   return “”
Compare every file in startPath to filename
if found return the path of the file
else
 for every subdirectory in startPath
     (subdirectory)
  return searchForFile(subdirectory, filename)
```

Recursion - Example

```
public static String searchFile(String path, String filename){
   File file = new File(path);
    String found = “”;
   if(file.exists()) {
      if(file.isDirectory()) {
         File[] files = file.listFiles();
         for(int i=0; i<files.length; i++) {
            if(files[i].isFile()) {
               if (files[i].getName().equals(filename))
                  return files[i].getAbsolutePath();
            }
            else {
               found =
                searchFile(files[i].getAbsolutePath(), filename);
               if(!found.equals(""))
                     return found;
               }
            }
      }
   return found;
}
```

Recursion - Example

```
public class Recursion {
   public static void main(String[] args) {
      System.out.println("Enter a directory: ");
      String dir = keyboard.next();
      System.out.println("Enter the filename: ");
      String file = keyboard.next();
      String found = searchFile(dir, file);
      if(!found.equals("")) {
         System.out.println("File " + file +
                            " found in: \n" + found);
      }
      else
         System.out.println("File " + file +
                            " not found.");
   }
}
```

SUMMARY

- Recursive method - calls itself
- Base case and recursive case - finite number of recursive calls
- Recursion vs Iteration - Iteration is better when easy to implement
- Examples of recursive functions