PROGRAMMING AND DATA STRUCTURES

RECURSION AND ALGORITHM ANALYSIS

HOURIA OUDGHIRI AND JIALIANG TAN

FALL 2023

OUTLINE

- Recursion
- Recursive methods and helper methods
- Recursion vs. Iteration
- Measuring the performance of programs
- Algorithm analysis and Big-O Notation
- Examples of algorithm complexity analysis

Student Learning Outcomes

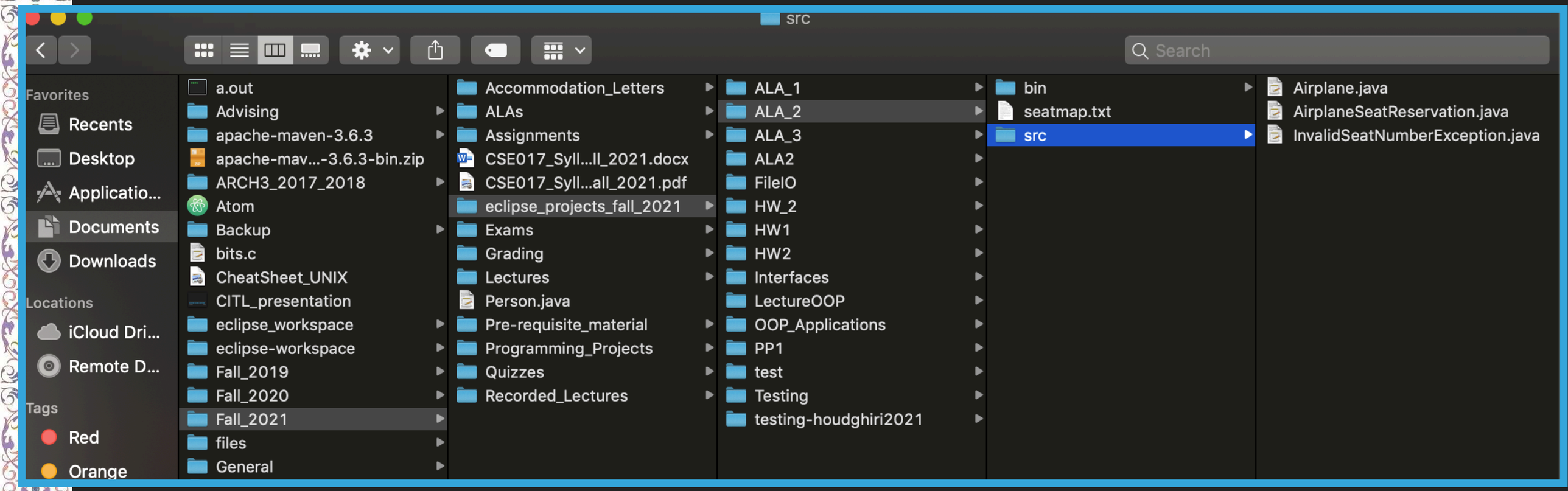
At the end of this chapter, you should be able to:

- Explain the concept of recursion and recursive methods
- Write and test recursive methods
- Compare iterative and recursive methods
- Determine the time complexity of given algorithms using the Big-O notation

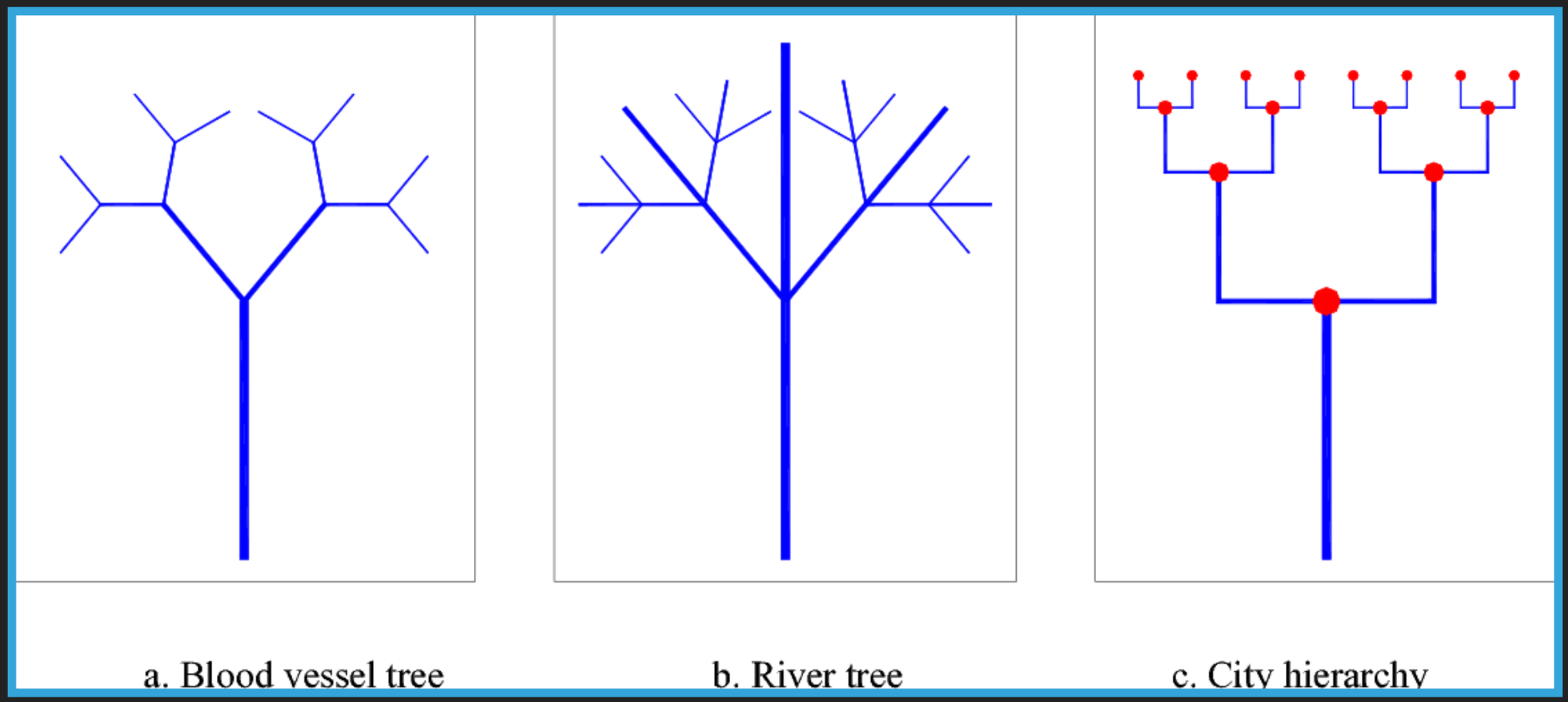
RECURSION

- Recursion is an algorithmic paradigm to solve iterative problems that are difficult to solve using loops
- Recursive mathematical series
- Exploring hierarchical or tree structures

File Hierarchy

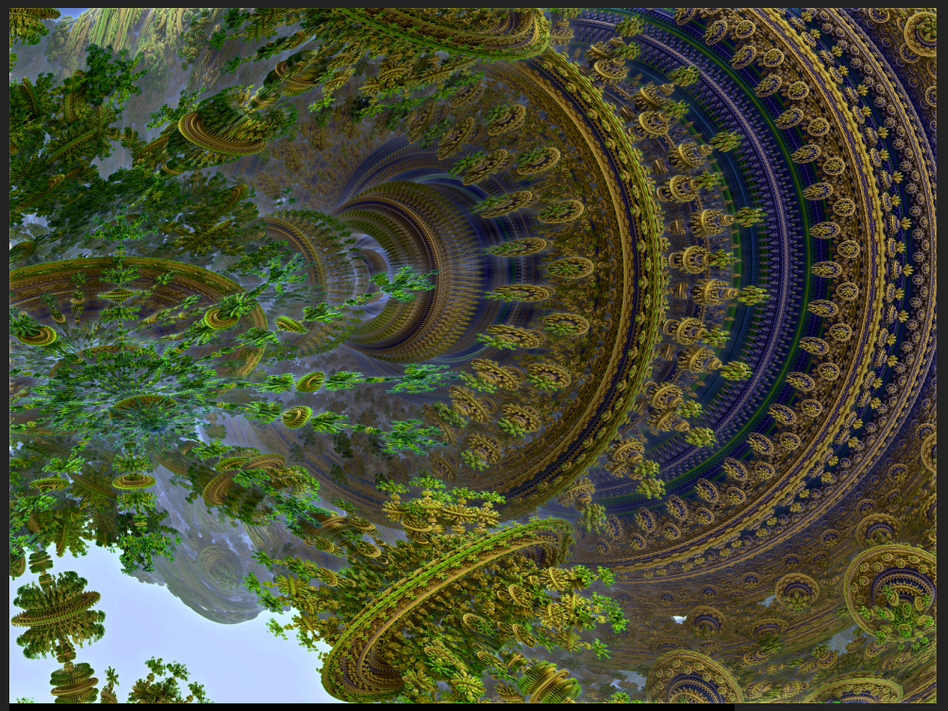


Drawing or exploring tree structures



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

Fractals



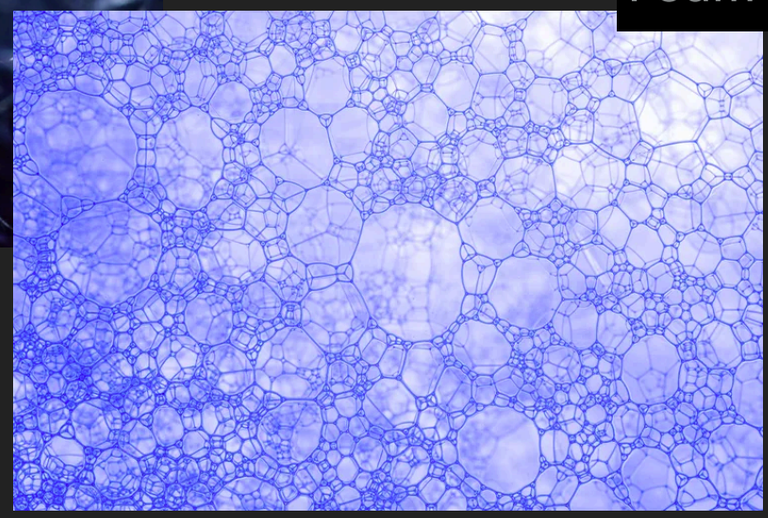
https://commons.wikimedia.org

Trees

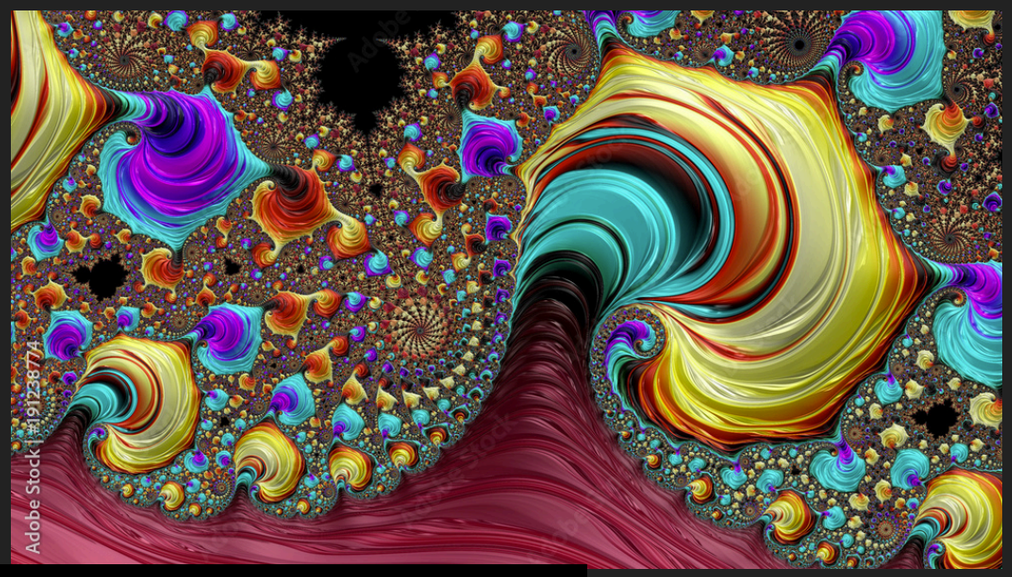
Broccoli

Ice and Snow

Foam



https://www.treehugger.com



https://stock.adobe.com

Recursive methods

- A method that calls itself
- Example: Recursive series in mathematics (Calculating Factorial)

```
n! = n x (n-1)!   1! = 0! = 1
5! = 5 x 4!
4! = 4 x 3!
3! = 3 x 2!
2! = 2 x 1!
1! = 1
```

Recursive methods

Iterative Factorial

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

Recursive Factorial

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

Recursive methods

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

120

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

5

Recursive methods

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

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

24(4*6)

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

6 (3* 2)

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

2 (2 * 1)

1

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

Recursive methods

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

Base Case

Stopping Case

Recursion

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

Recursive methods and the Stack

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

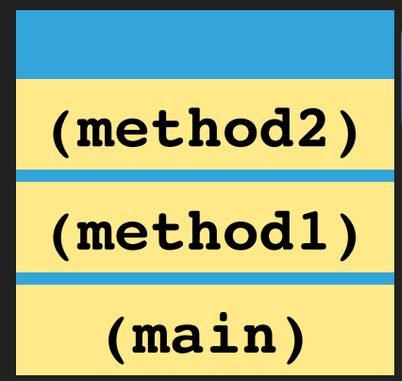
Recursive methods and the Stack

```
public class Stack{
    public static void method2(){
        // some code here
        return;
    }
    public static void method1(){
        // some code here
        method2();
        // some code here
        return;
    }
    public static void main(String[] args){
        // some code here
        method1();
        // some code here
        return;
    }
}
```

3

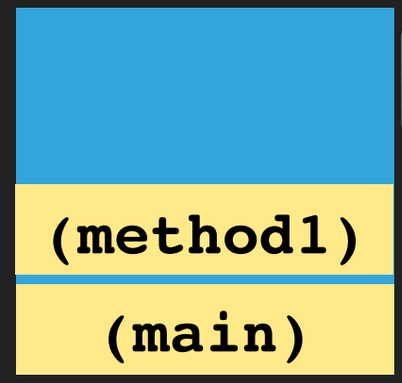
2

1



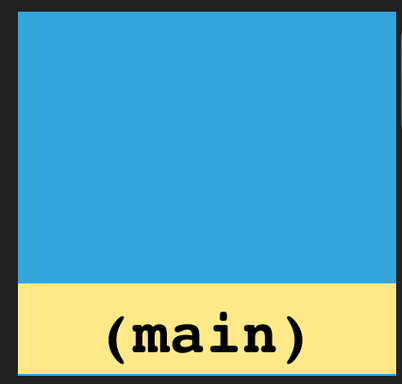
3

Method2 is at the top of the stack



2

method1 is at the top of the stack



1

main is at the top of the stack

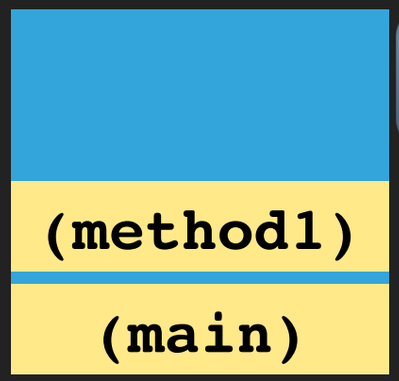
Recursive methods and the Stack

```
public class Stack{
    public static void method2(){
        // some code here
        return;
    }
    public static void method1(){
        // some code here
        method2();
        // some code here
        return;
    }
    public static void main(String[] args){
        // some code here
        method1();
        // some code here
        return;
    }
}
```

1

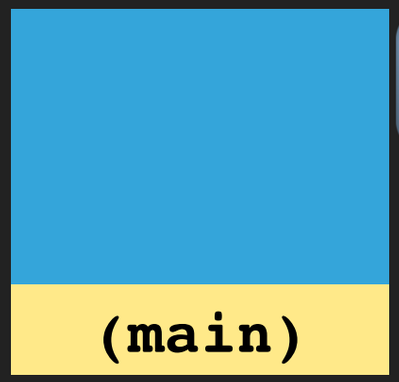
2

3



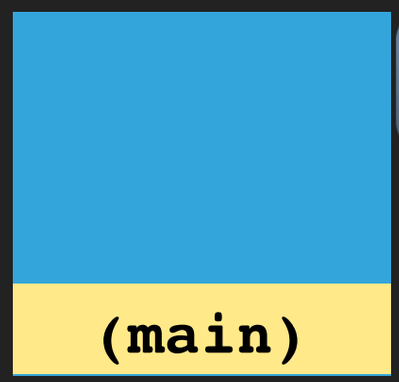
1

Method1 is at the top of the stack



2

main is at the top of the stack



3

main is at the top of the stack

Recursive methods

Practice: What is the output of the following code?

```
public class Test{
    public static void one(int n){
        if(n > 0){
            System.out.print(n + " ");
            one(n-1);
        }
    }
    public static void two(int n){
        if(n > 0){
            two(n-1);
            System.out.print(n + " ");
        }
    }
    public static void main(String[] args) {
        one(5);
        System.out.println();
        two(5);
    }
}
```

Recursive methods

Practice: What is the output of the following code?

```
public class Test{
    public static void main(String[] args) {
        calculation(1000);
    }
    public static void calculation(double n){
        if(n != 0){
            System.out.println(n + " ");
            calculation(n/10);
        }
    }
}
```

Recursive methods

- Write a recursive method to compute the following series and test it for $m = 10$

```
double series(int m)
```

$$series(m) = \frac{1}{3} + \frac{2}{5} + \frac{3}{7} + \ldots + \frac{m}{2m+1}$$

Recursive methods

```
public static double series(int n){

   if (n == 1)

      return 1/3.0;

   else

      return (double)n/(2*n+1) + series(n-1);

}
```

1

2

3

Base Case will be reached, n is decremented

Base case returns $\frac{1}{3}$ = m(1)

Recursion returns

$$\frac{n}{2n+1} + series(n-1)$$

Recursive helper methods

✦ Recursive Binary Search

Finding a key in an array of ordered numbers. A problem that is inherently recursive

```
public static 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;
  }
```

Iterative Binary Search

Recursive helper methods

```
public static int rBinarySearch(int[] list, int key){
    int first = 0;
    int last = list.length-1;
    return rBinarySearch(list, key, first, last);
}
```

```
public static int rBinarySearch(int[] list, int key,
                                int first, int last){
   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 rBinarySearch(list, key, first, last);
   }
}
```

Recursive Binary Search

Helper Method that is recursive and accepts additional parameters

Recursion vs. Iteration

Fibonacci sequence

- Used to model many natural phenomena (rabbit population growth, petal structure in a flower, pine cones, financial market analysis)

$$F_n = F_{n-1} + F_{n-2}, n > 0$$

$$F_2 = 1, F_1 = 1$$

https://en.wikipedia.org/wiki/Fibonacci_number

- 1 1 2 3 5 8 13 21 ($f_8 = 21$)

Recursion vs. Iteration

Iterative Fibonacci

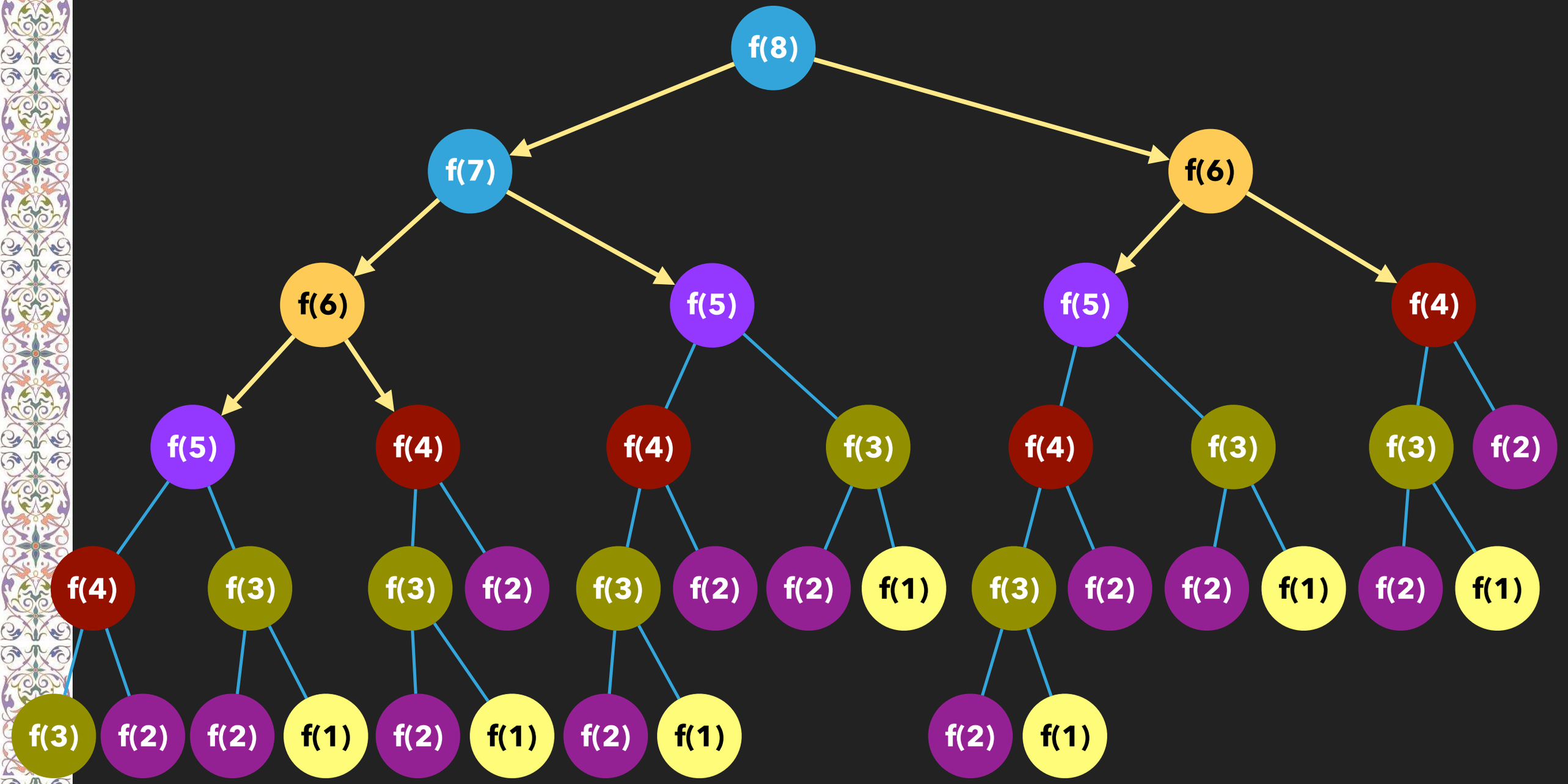
```
public static int fibonacci(int n){
        int f1=1, f2=1, f=0;
        if (n <= 2)
            return 1;
        else{
            while (n > 2){
                f = f1 + f2;
                f1 = f2;
                f2 = f;
                n = n -1;
            }
        }
        return f;
  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

Recursive Fibonacci

```
public static int rFibonacci(int n){
  if (n <= 2)
    return 1;
  else
   return rFibonacci(n-1) + rFibonacci(n-2);
}
```

Recursion vs. Iteration

- Fibonacci Series - **`rFibonacci(8)`**



Recursion vs. Iteration

- Recursion often results in less code than iteration
- Iterations are more computationally efficient than recursion
- Recursion requires consecutive calls to the same function (stack push/pop operations)
- Use recursion only when the problem is hard to solve using loops

Summary

- Recursive method - calls itself
- Base case and recursive case - finite number of recursive calls
- Recursion vs Iteration - Iteration is more efficient if you can use it
- Use helper recursive methods when you need to change the signature of the method

ALGORITHM ANALYSIS

- Algorithm Design: Finding a computational solution to a problem
- Often many solutions are possible

How to select a solution?

- Use Algorithm Analysis
- Example: Binary Search and Linear Search

ALGORITHM ANALYSIS

- Compare Binary Search/Linear Search
  - Measure the execution time of the two methods
  - Use different sizes of the array
  - Use a random value as the key

```
public static int linearSearch(
                int[] list, int key){
 for (int i=0; i<list.length; i++){
    if (list[i] == key) {
       return i;
    }
 }
 return -1;
}
```

```
public static 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;
    }
```

Linear vs. Binary Search

Size	Linear(ns)	Binary(ns)
100	2287	1229
1000	10041	610
10000	64322	767
100000	65377	3702
1000000	32123	5379
10000000	1963618	3701
100000000	4542373	4829

ALGORITHM ANALYSIS

- Algorithm Analysis: Predict the performance of an algorithm before implementing it (coding)
- Determine the upper bound on the performance of the algorithm based on the problem size (iterations)

Linear vs. Binary Search

Size	Linear(iterations)	Binary(iterations)
100	100	6
1000	682	8
10000	10000	13
100000	100000	16
1000000	435045	19
10000000	10000000	24
100000000	100000000	27

Big-O notation

- **Growth rate:** How fast an algorithm's execution time increases as the input size increases
- Big-O notation: Upper Bound on the growth function
- Theoretical approach independent of computers (machines) and specific input
- Time complexity: Growth of the execution time as a function of the input size
- Space complexity: Growth of the amount of memory space as a function of the input size

Big-O notation

```
public static int linearSearch(int[] list, int key){
 for (int i=0; i<list.length; i++){
     if (list[i] == key) {
         return i;
     }
 }
 return -1;
}
```

n: size of list

Number of iterations = n (worst case)

Time(n) = n * constant time = *O(n)*

- Time grows linearly with the input size (**n**)

O(n) - Order of n - Linear Algorithm

Big-O notation

- Property:
  - Multiplicative constants are ignored
  - Non-dominating terms are ignored
  - **O(100 x n) ≃ O(n/5) ≃ O(n)**
  - **O(n-1) ≃ O(n-1000) ≃ O(n)**
  - **O((5n² + 2n + 11)/7) ≃ O(n²)**
  - **O(1)**: execution time is constant or not related to the input size

Big-O notation

- Useful mathematical formulas:

$1+ 2+ 3+ \ldots + n= n(n+1)/2 - \mathbf{O(n^2)}$

$\mathbf{n(n+1)/2= n^2/2+n/2 \cong n^2+n \cong n^2 \cong O(n^2)}$

$a^0+a^1+a^2+ \ldots +a^n = a^{(n+1)}-1/a-1 \mathbf{\cong O(a^n)}$

$2^0 + 2^1+ \ldots + 2^n = 2^{(n+1)}-1 \mathbf{\cong O(2^n)}$

Big-O notation

- What is the order of the following expressions?

1. $\frac{(n^2+1)^2}{n}$
2. $3n * 2log_2 n + \frac{1}{n}$
3. $nlog_2 n + 2log_2(n^2)$
4. $2^n + 25n^3 + 45log_2 n$

Big-O notation

- Examples

```
for(int i=0; i<n; i++){
    k = k + 5;
}
```

n times

Constant time

$$Time(n) = n * const \approx O(n)$$

```
for(int i=1; i<= n; i++){
  for(int j=1; j<= n; j++){
    k = k + i + j;
  }
}
```

n times

n times

constant time

$$Time(n) = n * n * const \approx O(n^2)$$

Big-O notation

```
for(int i = 1; i <= n; i++){
   for(int j = 1; j <= i; j++){
     k = k + i + j;
   }
}
```

n times

1, 2, … or n times

constant time

$$Time(n) = \frac{n(n+1)}{2} * const \approx O(n^2)$$

```
for(int i = 1; i <= n; i++){
    k = k + 4;
}
for(int i = 1; i <=n; i++){
    for(int j=1; j<=20; j++){
        k = k + i + j;
    }
}
```

n times

constant time

n times

20 times

constant time

$$Time(n) = n * const + n * 20 * const \approx O(n)$$

Big-O notation

```
result=1;
for(int i=1; i<=n; i++)
  result = result * a;
```

n times

constant time

$$Time(n) = n * const \approx O(n)$$

```
result = a;
for(int i=1; i<=k; i++)
  result = result * result;
```

k times, n = 2^k

constant time

$$Time(n) = log_2 n * const \approx O(log_2 n)$$

Big-O notation

```
int count = 1;
while(count < n)
    count = count * 2;
```

log n times

constant time

$$Time(n) = log_2 n * const \approx O(log_2 n)$$

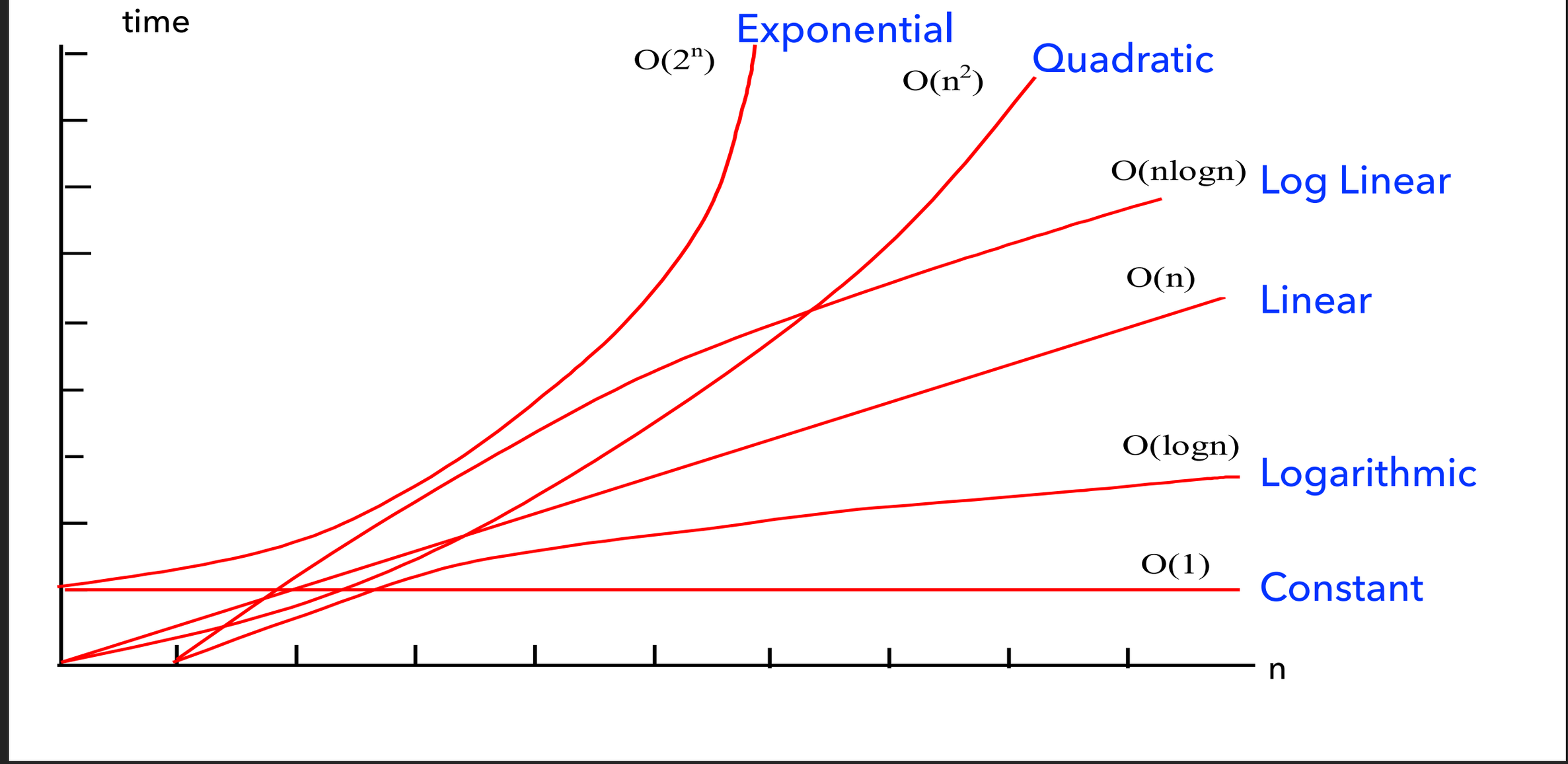
Big-O notation

- Practice: Determine the time complexity of the code below using the Big-O notation

```
for (int i = n/2; i > 0; –i){   n/2 times
  int x = n * 3;
  while(x > 1){   log(n-3) times
    for (int j=0; j < 100; j+=2) 50 times
      System.out.prinltn("X: " + x);
    x = x / 2;
  }
} Time(n) = n/2 * log(n-3) * 50 ~ O(n logn)
```

Big-O notation

Common growth functions



Big-O notation

- Analyzing Binary Search

```
public static 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;
}
```

Big-O notation

- Analyzing Binary Search

Iteration 1: size = **n/2**

Iteration 2: size = **n/4**

Iteration i = size = **$n/(2^i)$**

Last iteration k: $n/2^k = 1$ (no splitting)

$k = \log_2(n)$

Binary Search: $O(\log_2 n)$ - Logarithmic Growth

Big-O notation

✦ Analyzing Selection Sort

```
public static int selectionSort(int[] list) {
   for(int i=0; i<list.length-1; i++){
      int minIndex = i;
      for(int j=i+1; j<list.length; j++){
         if (list[i] < list[minIndex]){
            minIndex = i;
         }
      }
      int temp = list[i];
      list[i] = list[minIndex];
      list[minIndex] = temp;
   }
}
```

Big-O notation

Analyzing Selection Sort

```
Iteration1(outer loop) i=0
    (n-1)iterations(inner loop)      1, n

Iteration 2 (outer loop) i=1
    (n-2) iterations (inner loop)    2, n

Iteration k (outer loop) i=k-1
    (n-k) iterations (inner loop)    k, n

Iteration n-1 (outer loop) i=n-2
    1 iteration (inner loop)         n-1, n-1

(n-1) + (n-2) + … + 1 = (n-1)n/2
```

Selection Sort: $O(n^2)$ - Quadratic Growth

Big-O notation

- Analyzing Recursive Fibonacci sequence

```
public static int rFibonacci(int n){
 if (n <= 2)
    return 1;
 else
    return rFibonacci(n-1)+
           rFibonacci(n-2);
}
```

```
Time(n) = Time(n-1) + Time(n-2)

Time(n) ≅ 2 * Time(n-1)
Time(n-1) = 2 * Time(n-2)
Time(n) = 2 * 2 * Time(n-2)
…
Time(n) = 2^k * Time(n-k)
Time(n) = 2^(n-2) Time(2)
Time(n) = 2^(n) * constant
```

Recursive Fibonacci: $O(2^n)$ - Exponential Growth

Big-O notation

- Analyzing Iterative Fibonacci sequence

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

Time Complexity: (n-3) * const

Iterative Fibonacci: O(n) - Linear Growth

Efficient Algorithms

- Designing algorithms while minimizing their time complexity
- Binary search improves linear search complexity by having the data sorted prior to the search
- Iterative Fibonacci sequence reuses prior calculations to determine next terms

Summary

- Algorithm analysis - predict the performance of an algorithm independently of its implementation
- Estimate the time complexity or space complexity (growth rate as a function of the problem size)
- Big-O notation - Upper bound for the growth rate
- Algorithm efficiency - reduce time complexity - tradeoff between time and space complexity