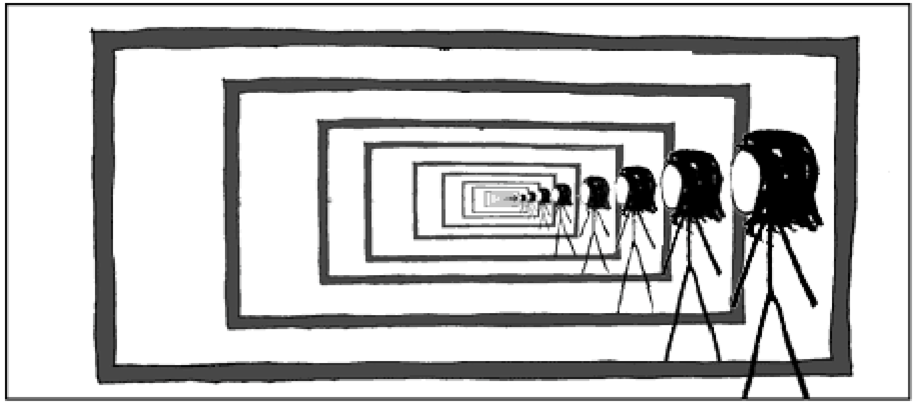
Data Structures and Algorithms

Recursion



Prepared by:

Eng. Malek Al-Louzi

School of Computing and Informatics– Al Hussein Technical University

Spring 2021/2022

Outlines

- Introduction
- Factorial – A simple Recursion
- Fibonacci Sequence
- Time Complexity Analysis of Recursive Programs
- General Rules - Recurrence Relations
- General Rules - Masters Theorem for decreasing functions

Introduction

- Recursion is a very important and powerful programming concept.
- Let us begin with an example.
- We will write a program to find the factorial of a positive integer.

Factorial – A simple Recursion

- In mathematics, factorial of n is defined as the product of all the integers from n to 1.

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 1$$

- We can notice that (n-1) x (n-2) x … x 1 is (n-1)!.

$$n! = n \times \underbrace{(n-1) \times (n-2) \times \ldots \times 1}_{(n-1)!}$$

Factorial – A simple Recursion

- So, we can say:

$$n! = n \times (n-1)!$$

- This is true for all n greater than 0.
- Zero factorial is a special case, and it is equal to 1.

$$n! = \begin{cases} n \times (n-1)! & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$$

Factorial – A simple Recursion

- When we write a function in **a simpler form of itself**, we call such a function a recursive function.

$$n! = \begin{cases} n \times (n-1)! & \text{if } n>0 \\ 1 & \text{if } n=0 \end{cases}$$

Recursive function

- The concept of a recursive function is also valid in the context of a programming language.

Factorial – A simple Recursion

- Let us write a function that returns an integer, and the name of the function is factorial.

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

Factorial – A simple Recursion

- We can see within a function factorial we are calling the function itself though with a reduced argument.
- When we call a function within itself, we say that such a call is a recursive call.

```
int  Factorial (int  n)
{
   if (n == 0)
      return  1;
   else
     return  n x Factorial(n-1);
}
```

Recursive
call

Factorial – A simple Recursion

- Let us see how computer executes the previous factorial function.
- We want to calculate factorial(4), F(4) for simplicity.
- When the computer try to calculate F(4), it finds that it is calling F(3) recursively.
- It will Pause the execution of F(4), it will go to calculate F(3) first, and then it will resume the execution after that.

	return	State
F(4)	4×F(3)	P

Factorial – A simple Recursion

- It will save the current state of F(4) into memory and goes to calculate F(3).
- F(3) again makes a call to F(2), so computer again pauses the execution of F(3) and goes to calculate F(2).

	return	State
F(4)	4×F(3)	P
F(3)	3×F(2)	P

Factorial – A simple Recursion

- F(2) again makes a call to F(1), so the computer pauses again and go to execute F(1).

	return	State
F(4)	4 × F(3)	P
F(3)	3 × F(2)	P
F(2)	2 × F(1)	P

Factorial – A simple Recursion

- F(1) again makes a call to F(0), so the computer pauses again and go to execute F(0).

	return	State
F(4)	4 × F(3)	P
F(3)	3 × F(2)	P
F(2)	2 × F(1)	P
F(1)	1 × F(0)	P

Factorial – A simple Recursion

- Now, when we come to F(0), then there is no recursive call further.

	return	State
F(4)	4×F(3)	P
F(3)	3×F(2)	P
F(2)	2×F(1)	P
F(1)	1×F(0)	P
F(0)	1	

- F(0) is kind of a base condition, and if it is not there then this recursion would have gone endlessly.

Factorial – A simple Recursion

- Now, F(0) returns 1 and it finishes.
- Then computer resumes F(1), and it calculates F(1) now.

	return	State
F(4)	$4 \times F(3)$	P
F(3)	$3 \times F(2)$	P
F(2)	$2 \times F(1)$	P
F(1)	$1 \times \underbrace{F(0)}_{1} = 1$	R

Factorial – A simple Recursion

- F(1) finishes, then F(2) is resumed and calculated.

	return	State
F(4)	$4 \times F(3)$	P
F(3)	$3 \times F(2)$	P
F(2)	$2 \times \underbrace{F(1)}_{1} = 2$	R

- F(2) finishes, then F(3) is resumed and calculated.

	return	State
F(4)	$4 \times F(3)$	P
F(3)	$3 \times \underbrace{F(2)}_{2} = 6$	R

Factorial – A simple Recursion

- Finally, F(4) resumed, calculated, and returns the final value.

	return	State
F(4)	$4 \times \underbrace{F(3)}_{6} = 24$	R

Fibonacci Sequence

- A very famous mathematical sequence.
- The first Two elements in the sequence is 0 and 1, all other elements are the sum of previous two elements.

0 1 1 2 3 5 8 13 . . .

- Let us say we want to solve the problem of finding nth Fibonacci number.
- We will write a function called F(n) that takes a positive integer n and returns the nth Fibonacci number.

Fibonacci Sequence

- If n is 2 then F(2) is 1.

0 1 1 2 3 5 8 13 . . .

- If n is 4 then F(4) is 3.

0 1 1 2 3 5 8 13 . . .

Fibonacci Sequence

- F(n) can be written using the following recurrence relation:

$$F(n) = \begin{cases} F(n-1) + F(n-2) & \text{if } n > 1 \\ n & \text{if } n = 0, 1 \end{cases}$$

- From the above function:

0	1	1	2	3	5	8	13	...
F(0)	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)	F(7)	...

Fibonacci Sequence

- Now, John and Sarah are two students, and they have both written programs to find the nth element in the sequence.
- They have written two different solutions.
- Let us see what these two different solutions.

0	1	1	2	3	5	8	13	...
F(0)	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)	F(7)	...

Fibonacci Sequence

John

```
Fib(n)
{
  if (n <= 1)
      return n
  F1 ← 0
  F2 ← 1
  for i ← 2 to n
      F ← F1 + F2
      F1 ← F2
      F2 ← F
  return F
}
```

Sarah

```
Fib(n)
{
  if (n <= 1)
      return n
  else
   return Fib(n-1) + Fib(n-2)
}
```

base condition

Fibonacci Sequence

- John has written an iterative program.
- Sarah has recently learned recursion and found it a lot simpler to solve this problem.
- If we try the two solutions, we will notice for large n for example F(45), Sarah's solution will take several seconds to finish!

Fibonacci Sequence

- Johan has written an iterative program.
- In this case, to calculate F(n):
  - He already has F(0) and F(1).
  - He goes to calculate F(2) from F(0) and F(1).
  - Then goes to calculate F(3) from F(2) and F(1).
  - It will go on till F(n).

John (Iterative)

F(0)
F(1)
→ F(2)
→ F(3)
⋮
→ F(n)

Fibonacci Sequence

- For example, to calculate F(5):

F(5)

↳ F(0)

↓

F(1)

↓

F(2)

↓

F(3)

↓

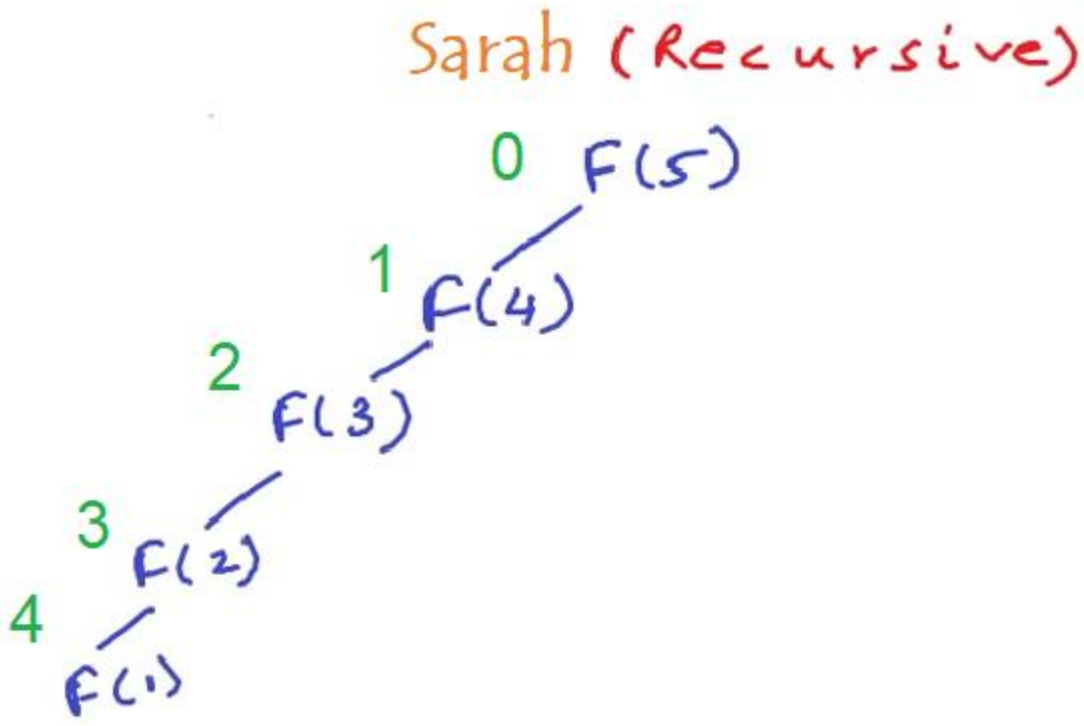
F(4)

↓

→ F(5)

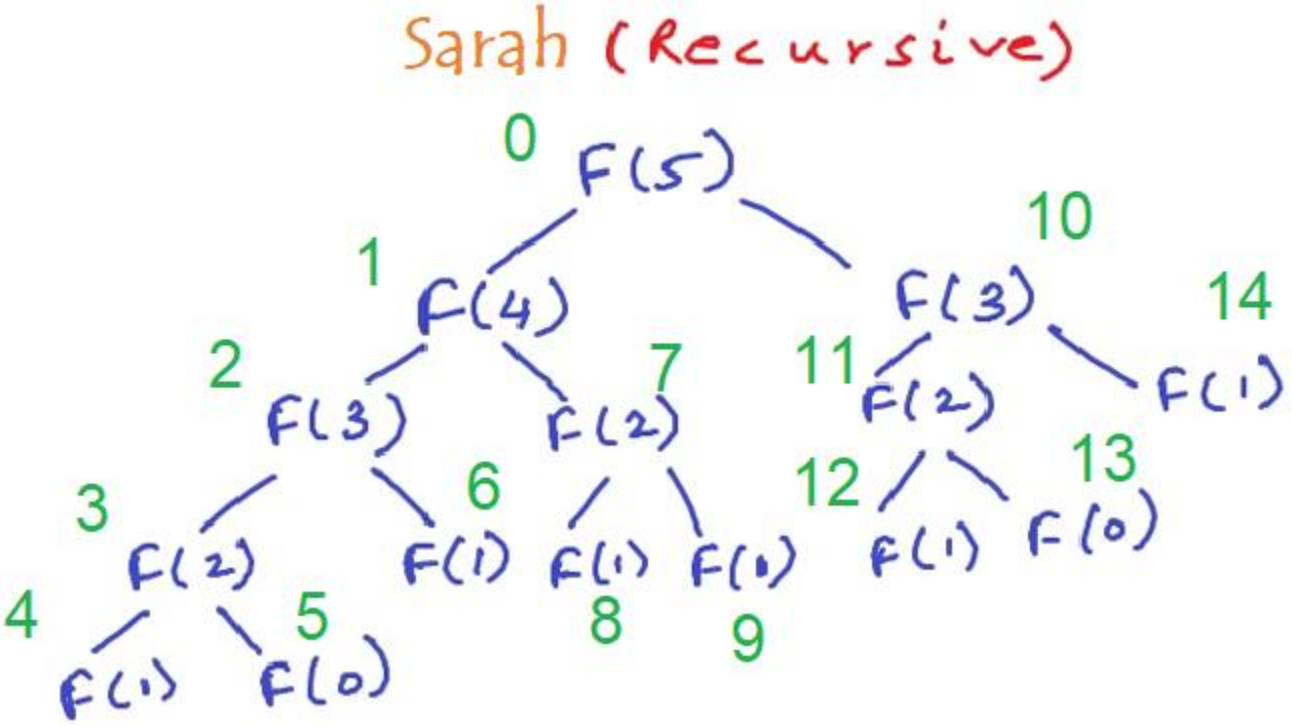
Fibonacci Sequence

- Now, if we want to calculate F(5) using Sarah's code:
- It will make a recursive call to F(4) and F(3).



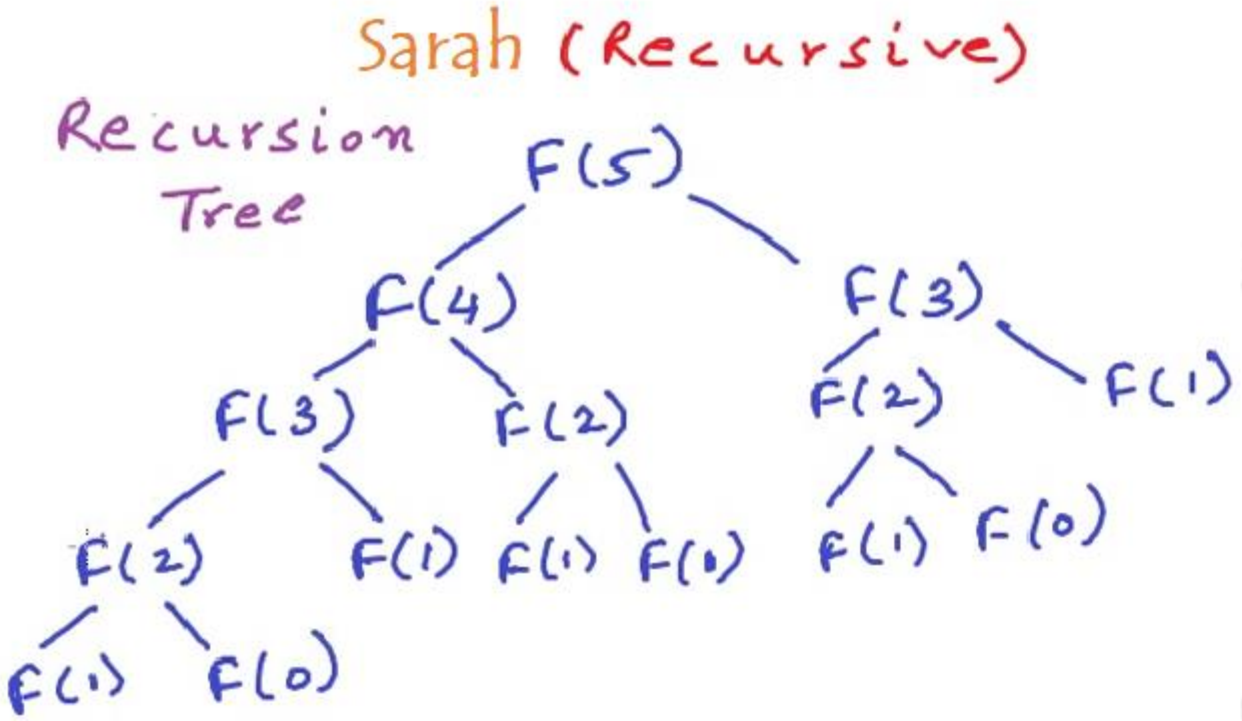
Fibonacci Sequence

- We will go in order in which the functions are called in the actual program.



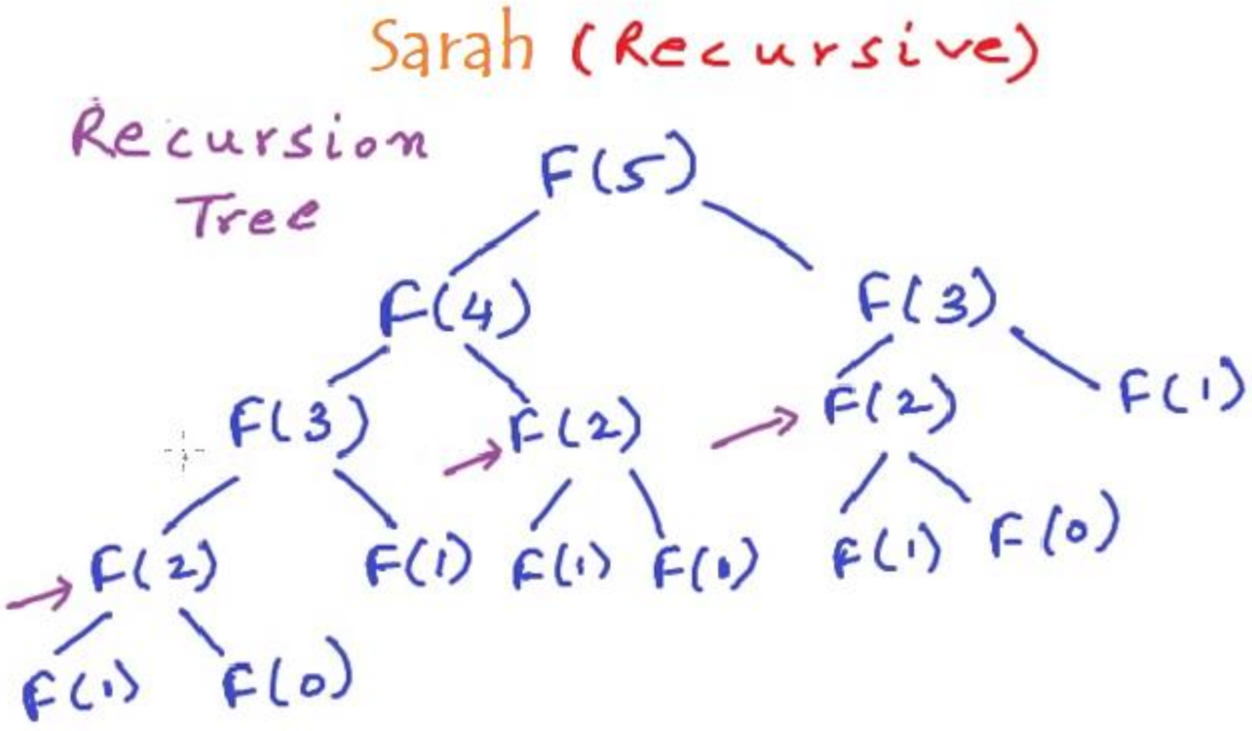
Fibonacci Sequence

- The structure that we have drawn is called a Tracing Tree or Recursion Tree.



Fibonacci Sequence

- We can see that the value F(2) is being calculated three times.



Fibonacci Sequence

- Similarly, F(3) is calculated twice.
- This is unnecessary overhead or redundancy.
- In iterative implementation, it calculates each value F(i) exactly once.
- In recursive implementation, we are calculating F(i) multiple times.

Fibonacci Sequence

- For example, if n = 5, we are calculating F(2) three times, as shown in the following table.

n	f(2)
5	3
6	5
8	13
40	63245986

- The running time of Sarah’s implementation is growing **exponentially** as the input increases.
- Let us find how!

Time Complexity Analysis of Recursive Programs

- Let us pickup the recursive implementation of the factorial.

```
Factorial(n)
{
   if n == 0
      return 1

   else
     return n x Factorial(n-1)
}
```

Time Complexity Analysis of Recursive Programs

- The time taken to calculate Factorial(n) is T(n).
- We will assume that each simple operation will costs us One unit of time.

```
Factorial(n)
{                        1
    if n == 0
        return 1

    else
     return n x Factorial(n-1)
}
                1              1
```

Time Complexity Analysis of Recursive Programs

- So, T(n) is:

$$T(n) = T(n-1) + 3$$

- This is true for all n greater than 0.

$$T(n) = T(n-1) + 3 \quad \text{if } n > 0$$

Time Complexity Analysis of Recursive Programs

- If n equals to 0, T(0) is equal to 1 because we only make a comparison in this case and return.

$$T(n) = T(n-1) + 3 \quad \text{if } n > 0$$

$$T(0) = 1$$

- Now, let's try to reduce T(n) in terms of our known value T(0).

$$T(n) = T(n-1) + 3$$

Time Complexity Analysis of Recursive Programs

- T(n-1) can be written as {T(n-2)+3}, So:

$$
\begin{aligned}
T(n) &= T(n-1) + 3 \\
&= T(n-2) + 6
\end{aligned}
$$

- In the same manner:

$$
\begin{aligned}
T(n) &= T(n-1) + 3 \\
&= T(n-2) + 6 \\
&= T(n-3) + 9
\end{aligned}
$$

Time Complexity Analysis of Recursive Programs

- If we reduce it by a generic K:

$$
\begin{aligned}
T(n) &= T(n-1) + 3 \\
&= T(n-2) + 6 \\
&= T(n-3) + 9 \\
&= T(n-k) + 3k
\end{aligned}
$$

- We already know T(0), so:

$$n - k = 0 \Rightarrow k = n$$

Time Complexity Analysis of Recursive Programs

- Finally, we can reduce T(n) as following:

$$
\begin{aligned}
T(n) &= T(n-1) + 3 \\
&= T(n-2) + 6 \\
&= T(n-3) + 9 \\
&= T(n-k) + 3k
\end{aligned}
$$

$$n - k = 0 \Rightarrow k = n$$

$$
\begin{aligned}
\Rightarrow T(n) &= T(0) + 3n \\
&= 3n + 1
\end{aligned}
$$

Time Complexity Analysis of Recursive Programs

- So, we can see that the time complexity is Linear time.
- Which means:

$O(n)$

Time Complexity Analysis of Recursive Programs

- Let us analyze the recursive implementation of Fibonacci sequence.

```
Fib(n)
{
  if n<=1
    return n
  else
   return Fib(n-1)+ Fib(n-2)
}
```

Time Complexity Analysis of Recursive Programs

- The time taken to calculate Fib(n) is T(n).
- Each simple operation takes One unit of time.

```
Fib(n)
{                  1
   if  n<=1
     return  n
   else
    return  Fib(n-1)+Fib(n-2)
}
                  1   1       1
```

Time Complexity Analysis of Recursive Programs

- Now, T(n):

$$T(n) = T(n-1) + T(n-2) + 4$$

- If n less than or equal to One:

$$T(n) = T(n-1) + T(n-2) + 4$$
$$T(0) = T(1) = 1$$

Time Complexity Analysis of Recursive Programs

- We will use the following approximation:

$$T(n-2) \approx T(n-1)$$

- In reality, T(n-1) is greater than T(n-2).
- In this case we are calculating the upper bound for T(n).
- This approximation simplifies our expression as:

$$T(n) = 2T(n-1) + c \qquad c = 4$$

Time Complexity Analysis of Recursive Programs

- Now, we can go on reducing the expression as following:

$$T(n) = 2T(n-1) + c \qquad c = 4$$
$$= 4T(n-2) + 3c$$
$$= 8T(n-3) + 7c$$

- If we reduce it in a generic form:

$$T(n) = 2^k T(n-k) + (2^k - 1)c$$

Time Complexity Analysis of Recursive Programs

- If we write T(n) in terms of T(0):

$$T(n) = 2^k T(n-k) + (2^k - 1)c$$

$$n - k = 0 \Rightarrow k = n$$

$$T(n) = 2^n T(0) + (2^n - 1)c$$

- We already know T(0):

$$\Rightarrow T(n) = (1+c)2^n - c$$

Time Complexity Analysis of Recursive Programs

- In Big O notation:

Fib (recursion) → $O(2^n)$ → exponential time

Fib (Iterative) → $O(n)$ → Linear Time

- Linear time is a lot better than exponential time, in fact exponential time is the worst kind of time complexity.

General Rules
Recurrence Relation #1

```java
void test(int n) {
    if(n > 0) {
        System.out.println(n);
        test(n - 1);
    }
}
```

- Time taken by this function T(n):

$$T(n) = T(n-1) + 1$$

- After solving this recurrence relation:

$$T(n) = n + 1$$

$$O(n)$$

General Rules

Recurrence Relation #2

```
void test(int n) {
    if(n > 0) {

        for(int i = 0; i < n; i++){

            System.out.println(n);
        }

        test(n - 1);
    }
}
```

- Time taken by this function T(n):

$$T(n) = T(n-1) + n$$

- After solving this recurrence relation:

$$T(n) = 1 + n(n+1)/2$$

$$O(n^2)$$

General Rules
Recurrence Relation #3

```
void test(int n) {
    if(n > 0) {

        System.out.println(n);

        test(n - 1);
        test(n - 1);
    }
}
```

- Time taken by this function T(n):

$$T(n) = 2T(n - 1) + 1$$

- After solving this recurrence relation:

$$T(n) = 2^{n+1} - 1$$

$$O(2^n)$$

General Rules

Masters Theorem for decreasing functions

$$T(n) = T(n-1) + 1 \quad \Rightarrow O(n)$$
$$T(n) = T(n-1) + n \quad \Rightarrow O(n^2)$$

$$T(n) = 2T(n-1) + 1 \quad \Rightarrow O(2^n)$$
$$T(n) = 3T(n-1) + 1 \quad \Rightarrow O(3^n)$$
$$T(n) = 2T(n-1) + n \quad \Rightarrow O(n2^n)$$

- The general form of Recurrence relation:

$$T(n) = aT(n - b) + f(n)$$

where:

$a > 0 \quad b > 0 \quad$ and $\quad f(n) = O(n^k) \quad k >= 0$

General Rules
Masters Theorem for decreasing functions

$$T(n) = aT(n - b) + f(n)$$

- Case 1:
  - If a = 1, the answer is:

$$O(nf(n))$$

  - Examples:

$$T(n) = T(n - 1) + 1 \quad \Rightarrow O(n)$$
$$T(n) = T(n - 1) + n \quad \Rightarrow O(n^2)$$

General Rules

Masters Theorem for decreasing functions

$$T(n) = aT(n - b) + f(n)$$

- Case 2:
  - If a > 1, the answer is:

$$O(a^{n/b} f(n))$$

  - Examples:

$$T(n) = 2T(n - 1) + 1 \Rightarrow O(2^n)$$
$$T(n) = 3T(n - 1) + 1 \Rightarrow O(3^n)$$
$$T(n) = 2T(n - 1) + n \Rightarrow O(n2^n)$$

General Rules
Masters Theorem for decreasing functions

$$T(n) = aT(n - b) + f(n)$$

- Case 3:
  - If a < 1, the answer is:

$$O(f(n))$$

  - Examples:

$$T(n) = 0.5T(n - 1) + 1 \quad \Rightarrow O(1)$$
$$T(n) = 0.75T(n - 1) + n \quad \Rightarrow O(n)$$

Any Questions???...