ET0736

Lesson 7

Recursive programming

Topics

- Recursive Method – basics
- Solving Factorial using Recursion
- Finding Fibonacci number using Recursion
- Coding Tower of Hanoi using Recursion
- Big-O notation

Introduction - Recursive method

A method that calls itself, like this one (endless):

```
class TestRecursive {
  public static void main(String[] args) {
    shout();
  }



  // recursive method
  public static void shout() {
    System.out.println("AHHHHH! ");
       shout();
  }
}
```

```
No input argument

Output:
AHHHHH!
AHHHHH!
AHHHHH!
:
:
(endless)
```

Introduction - Recursive method

A recursive method must have a condition to stop the further calling of itself.

The demo below stops further calling when the input argument, x, is 1.

```
class TestRecursive {
  public static void main(String[] args) {
    shout(5);
  }




  public static void shout(int x) {
    System.out.println(x + ".AHHHHH! ");
   if (x>1) shout(x-1);
  }
}
```

Add an input integer argument

<u>Output:</u>

```
5.AHHHHH!
4.AHHHHH!
3.AHHHHH!
2.AHHHHH!
1.AHHHHH!
```

(stop)

Introduction Recursive

To better understand the flow of the recursion, add another output message:

```
class TestRecursive {
  public static void main(String[] args) {
    shout(5);
  }

  public static void shout(int x) {
    System.out.println(x + “.AHHHHH! ");
    if (x>1) shout(x-1);
    System.out.println(x + “.Cheeeew! ");
  }
}
```

Output:

```
5.AHHHHH!
4.AHHHHH!
3.AHHHHH!
2.AHHHHH!
1.AHHHHH!
1.Cheeeew!
2.Cheeeew!
3.Cheeeew!
4.Cheeeew!
5.Cheeeew!
```

Introduction Recursive

- The code looks short.
- Take a closer look at the output.
- IMPORTANT: During the execution, every recursion generates a *new* copy of *shout()* that is different from the previous *shout()*, which has its own copy of variable x .

This is the first copy of ***shout()*** called by ***main()*** where its ***x*** has value of ***5***.

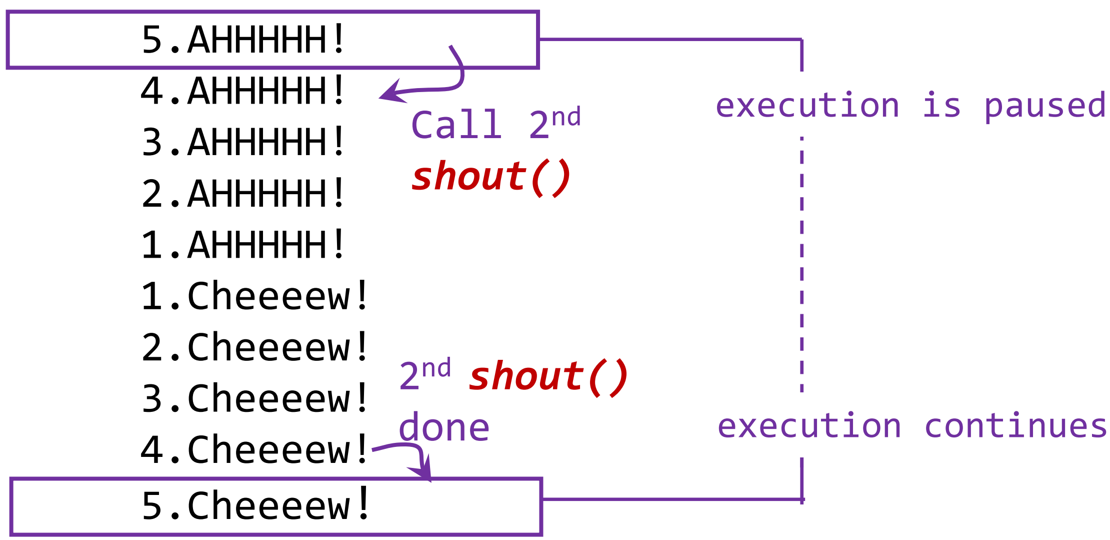
```
Output:
5.AHHHHH!
4.AHHHHH!
3.AHHHHH!
2.AHHHHH!
1.AHHHHH!
1.Cheeeew!
2.Cheeeew!
3.Cheeeew!
4.Cheeeew!
5.Cheeeew!
```

This is the last copy of ***shout()*** where its ***x*** has value of ***1***

Introduction Recursive

Take a closer look at the 1st copy of *shout()*

- What is its values of x? (*x=5*)
- Who called it? (the *main()*)
- Who did it call? (the *2nd* copy of *shout()*)
- *What value did it pass to 2nd* copy of *shout()? (5-1=4)*
- What happened after it called *2nd shout()*? (it paused its execution)
- When did it resume execution? (after *2nd shout()* is done – quite a long wait as in between, 3rd, 4th and last copy of shout() have been called and done sequentially)



Introduction Recursive

Next add a return value to the method.

```
class TestRecursive {
  public static void main(String[] args) {
        System.out.println(shout(5));
  }

  public static int shout(int x) {
    if (x>1) {
       return (x*shout(x-1));
    }
    return(1);
  }
}
```

Output:
120

Recursive nature

Start of forward

Main() calls shout(5)

shout(5) calls shout(4) when executing (5 *shout(4))

shout(4) calls (4 *shout(3))

shout(3) calls (3*shout(2))

forward

shout(2) calls (2*shout(1))

shout(1) not calling further but returns 1

End of forward

Recursive nature

End

main() receives 120 from shout(5)

return 120

shout(5) calls (5*shout(4))

return 24

shout(4) calls (4*shout(3))

return 6

shout(3) calls (3*shout(2))

return 2

shout(2) calls (2*shout(1))

return 1

shout(1) not calling further but returns 1

return

Start of return

Factorial

This is in fact a program for factorial 5!

```
class TestRecursive {
  public static void main(String[] args) {
        System.out.println(shout fac(5));
  }

  public static int shout fac(int x) {
    if (x>1) {
       return (x*shout fac(x-1));
    }
    return(1);
  }
}
```

Output:

```
120
```

Factorial 5! – iteration method

```
public static void main(String[] args) {
{
   int result = 1;
   for (int i=1; i<=5; i++) result *=i;
    System.out.println("result "+ result);

   return 0;
}
```

We want to achieve 5! = 1*2*3*4*5

- Start with result =1,
- In the for-loop, when i=1, result *= i yields ➔ (1*1)
- Next loop, when i=2, result *= i yields ➔(1*1*2)
- Next loop, when i=3, result *= i yields➔(1*1*2 *3)
- Next loop, when i=4, result *= i yields ➔ (1*1*2 *3*4)
- Next loop, when i=5, result *= i yields ➔ (1*1*2 *3*4*5)

Recursive programming

General case

Generally, for a value of x,

```
return( x* Fac (x-1) );
```

Base case

To stop further recursive calling

Except when x=1

```
if (x == 1) return 1;
```

Recursive programming

- Process of solving a problem by reducing it to smaller versions of itself
- General case for which the solution is obtained by calling the recursive function further
- Base case for which the solution is obtained directly and stop the recursion
- All recursive algorithms can be implemented using iteration (conventional loop-structures)

Iterative vs Recursive

Iterative

- Shorter performance time
- More coding
- Usually start from the “bottom” and iterate up to build the result.
- Must have condition to end loop, else infinite loop.

Recursive

- Longer performance time
- Simpler coding
- start from the “top” to reduce the problem until the bottom.
- Must have base case to stop recursion, else infinite recursion.

Some points to consider :

- Recursion can take up significant resources
  - Every call upon itself would cause a set of incomplete data and/or “unfinished” operations to be stored onto a “stack” in memory
  - Solution that requires a long nested recursion will therefore take up a lot of memory space
- Recursion may not be the only way to solve a problem
  - There may be an iterative solution or even a modified recursion that does not keep too many incomplete data pending the arrival of the base case
- Iterative solutions should be considered
  - Iterative solutions usually involve updating over the same variable spaces so less memory is used

Recursion can be heavy on resources

- However, there are certain problems where recursion provides a much easier to understand solution
- Recursion is a fascinating concept(to mathematicians) and some problems are more intuitively expressed in a recurrence relationship
- Natural candidates are like
  - Tower of Hanoi
  - Fibonacci numbers

Fibonacci number

In mathematics, the **Fibonacci numbers**, commonly denoted F_n form a sequence, called the **Fibonacci sequence**, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,

$$F_0 = 0, \quad F_1 = 1,$$

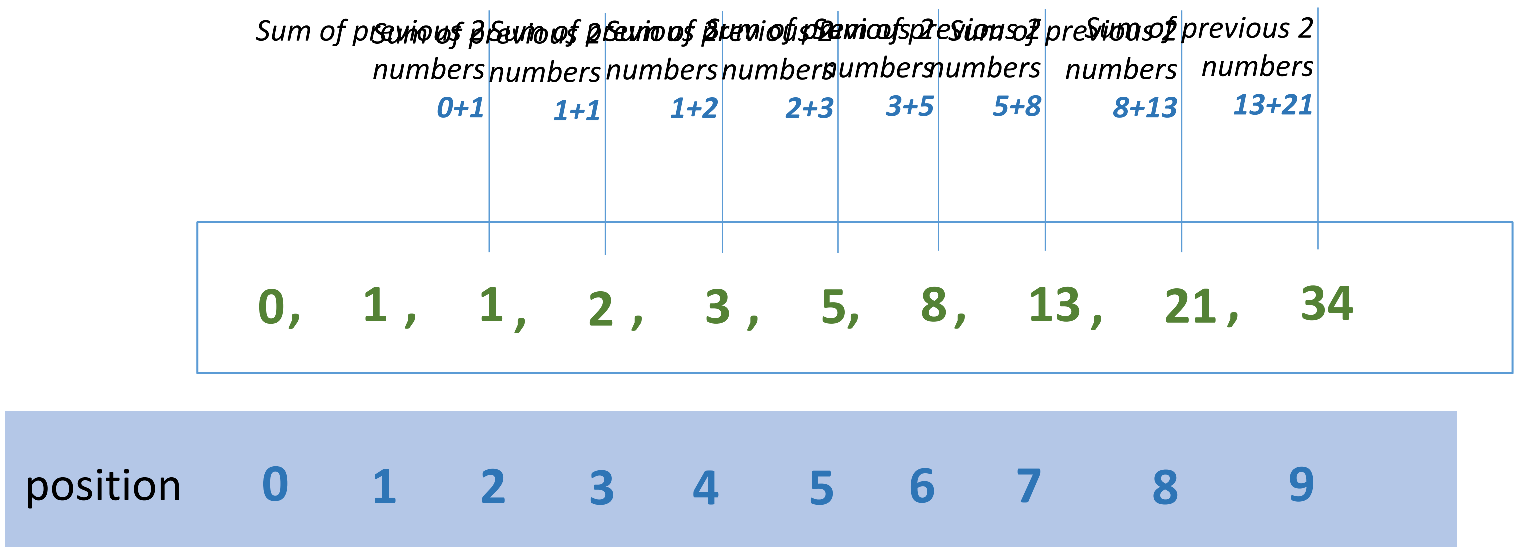
and

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

for $n > 1$.

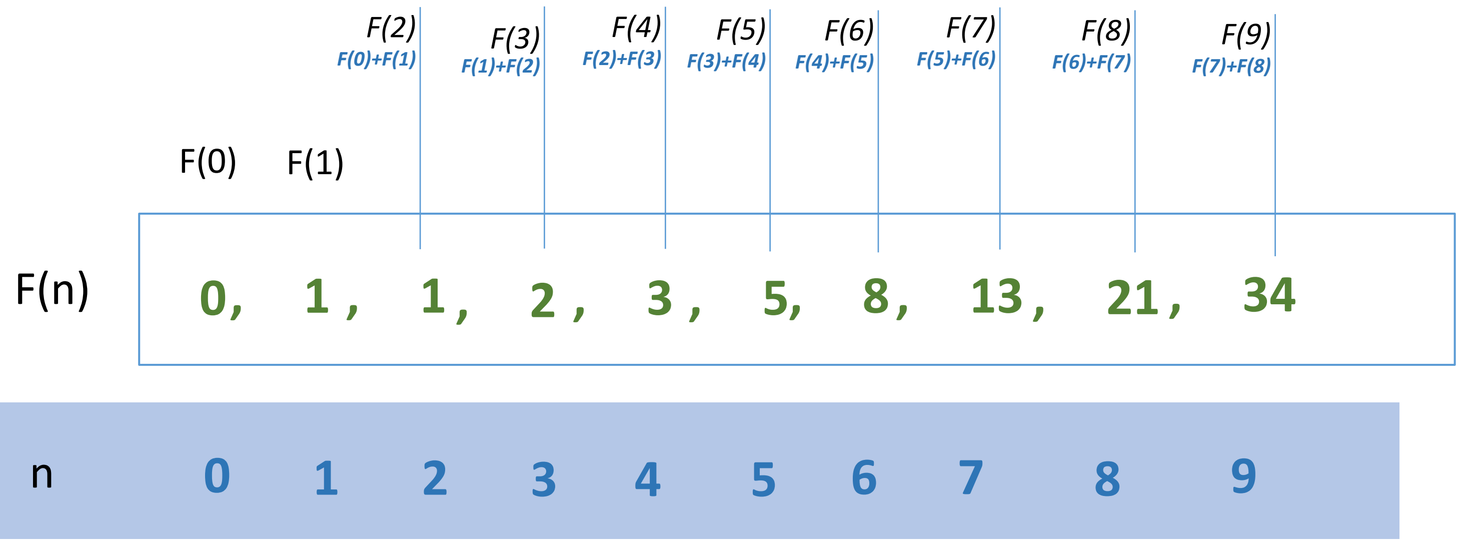
Fibonacci Number

Sequence: 0,1, 1, 2, 3, 5, 8, 13, 21, 34 . . .



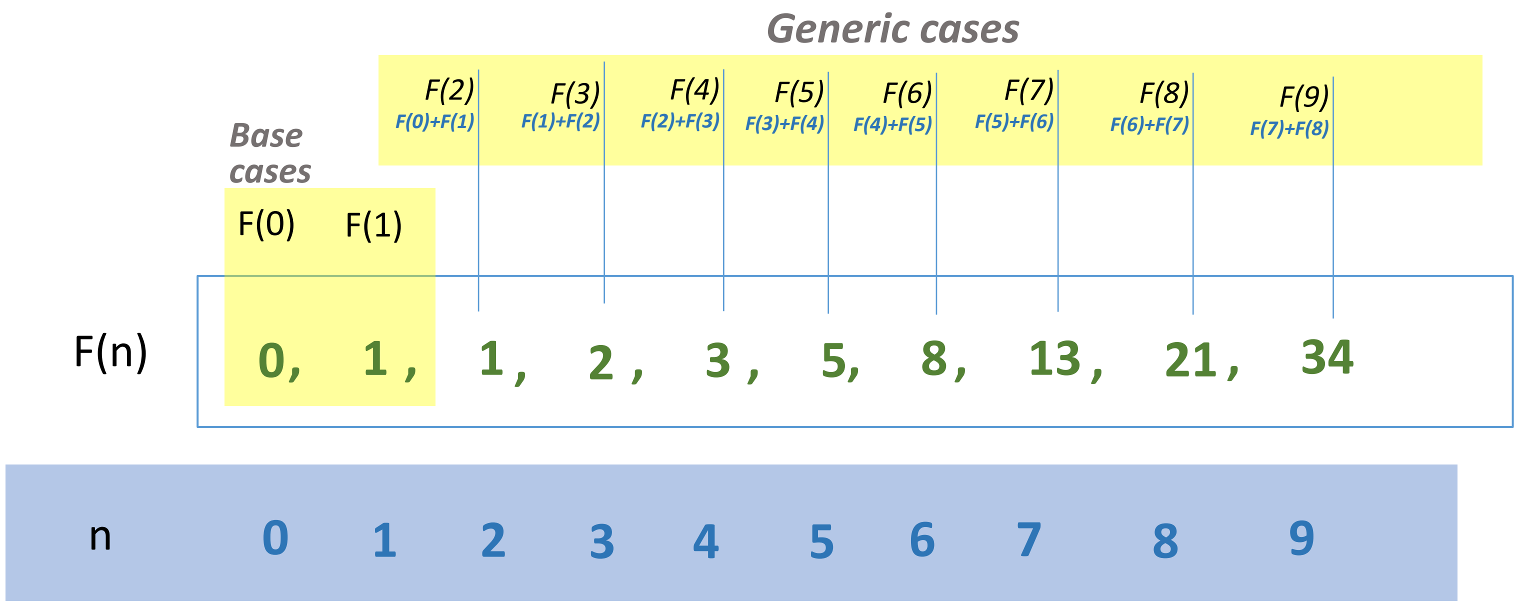
Fibonacci Number

Sequence: 0,1, 1, 2, 3, 5, 8, 13, 21, 34 . . .



Fibonacci Number

Sequence: 0,1, 1, 2, 3, 5, 8, 13, 21, 34 . . .



Fibonacci Number

To display n Fibonacci numbers from the beginning

```
class TestRecursive {
  public static void main(String[] args) {
    int count = 6;
     for (int i=0; i<count; i++)
       System.out.println( “Result = ” + f(i));
 }
}

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

Base cases

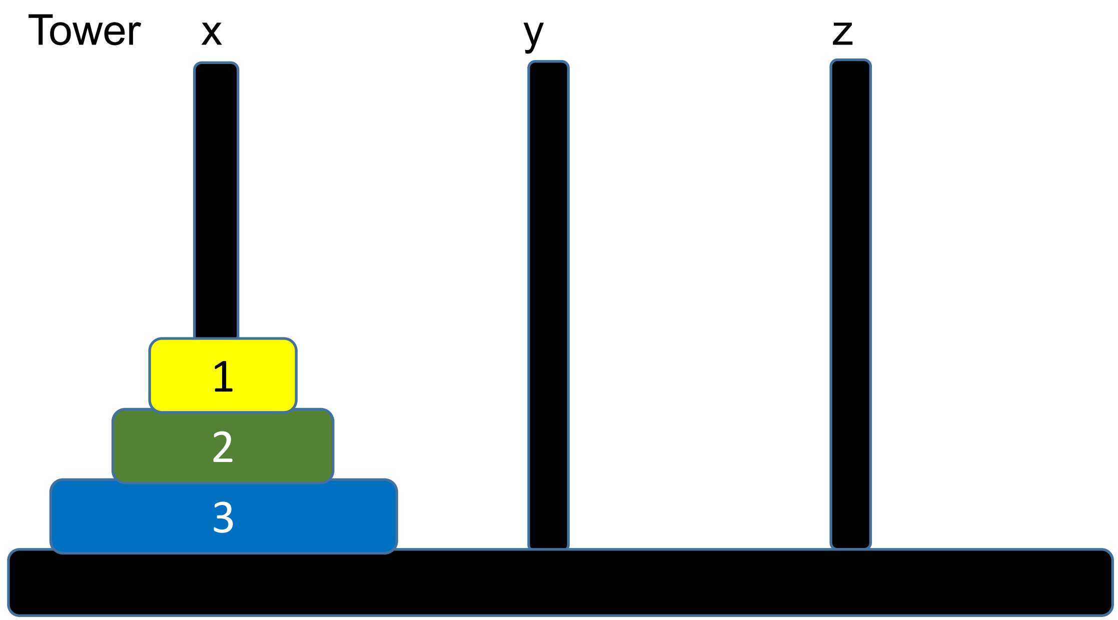
General cases

Tower of Hanoi

The aim is to move all disks from needle x to z via y following these rules:

- Only one disk can be moved at a time
- Removed disk must be placed on one of the towers
- A larger disk cannot be placed on top of a smaller disk

Play !!! https://www.mathsisfun.com/games/towerofhanoi.html



Tower of Hanoi (cont’d.)

- Base Case: first needle contains 0 disk
  - Stop recursion
- Generic Case: first needle contains only 1 or more disks
  - Recursive algorithm in pseudocode

Algorithm

Assuming there are n needles (where n>= 1).

The start tower: x

The destination tower: z

The middle tower: y

The required actions:

1. Move the top (n-1) disks from x to middle y (using z as intermediate)
2. Move n directly from x to destination z
3. Move the top (n-1) disks from y to z (using x as intermediate)

Tower of Hanoi (cont'd.)

Assuming there are n needles (where n>= 1).
The start tower: *source*
The destination tower: *dest*
The middle tower: *intermediate*
The required actions:

1. Move the top (n-1) disks from *source* to *intermediate* y (via *dest*)
2. Move n directly from *source* to *dest*
3. Move the top (n-1) disks from *intermediate* to *dest* (via *source*)

```
public static void move(int n, char source, char dest, char intermediate) {
  if (n>=1) {
     move (n-1, source, intermediate, dest);
     System.out.println ("Move "+n+ " from "+source +" to "+ dest);
     move (n-1, intermediate, dest, source);
  }
}
```

Tower of Hanoi (cont'd.)

Examine the recursive nature of the problem with an example.

E.g. Move 8 disks from needle x to z via y

Ultimate mission
Start needle : x
End needle : z

Move 7 disks from needle x to y via z

Sub-mission #1
Start : x
End : y

Move 6 disks sub-mission#1

Move 6 disks sub-mission#2

Move disk **8** directly to z
(All other 7 disks are at needle y)

Move 7 disks from needle y to z via x

Sub-mission #2
Start : y
End : z

Movement of disk is always from ***Source*** to ***Destination*** via ***Intermediate*** tower

Big O Time Complexity

- This notation represents the performance of an algorithm as the input size grows
- This notation provides an upper bound (or worst-case scenario) on the growth rate of an algorithm's running time needed to solve a problem

Big O Time Complexity

Given: an array of integers

```
int[] givenArray = { 1, 5, 8, 10, 33, 47 };
```

Given: a method that compute the sum of all elements in a given array:

```
int getSum(int[] givenArray) {
   int[] total = 0;
   for (int i=0; i<givenArray.length; i++)
       total += givenArray[i];
   return (total);
}
```

Linear Time Complexity

How does the running time of this method grow as the input (i.e. number of element in the array) grows?

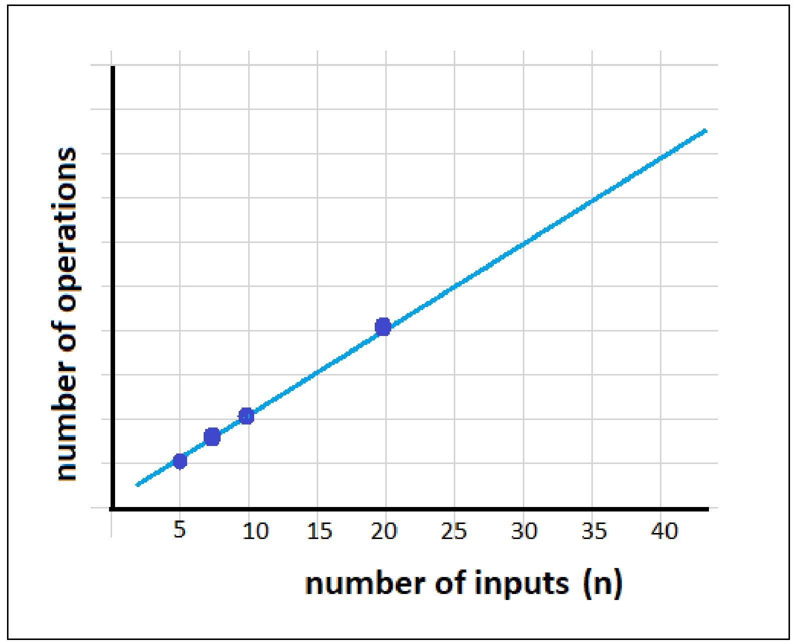
```
int getSum(int[] givenArray) {
   int[] total = 0;
   for (int i=0; i<givenArray.length; i++)
       total += givenArray[i];
   return (total);
}
```

{ 1, 5, 8, 10, 33, 47 }

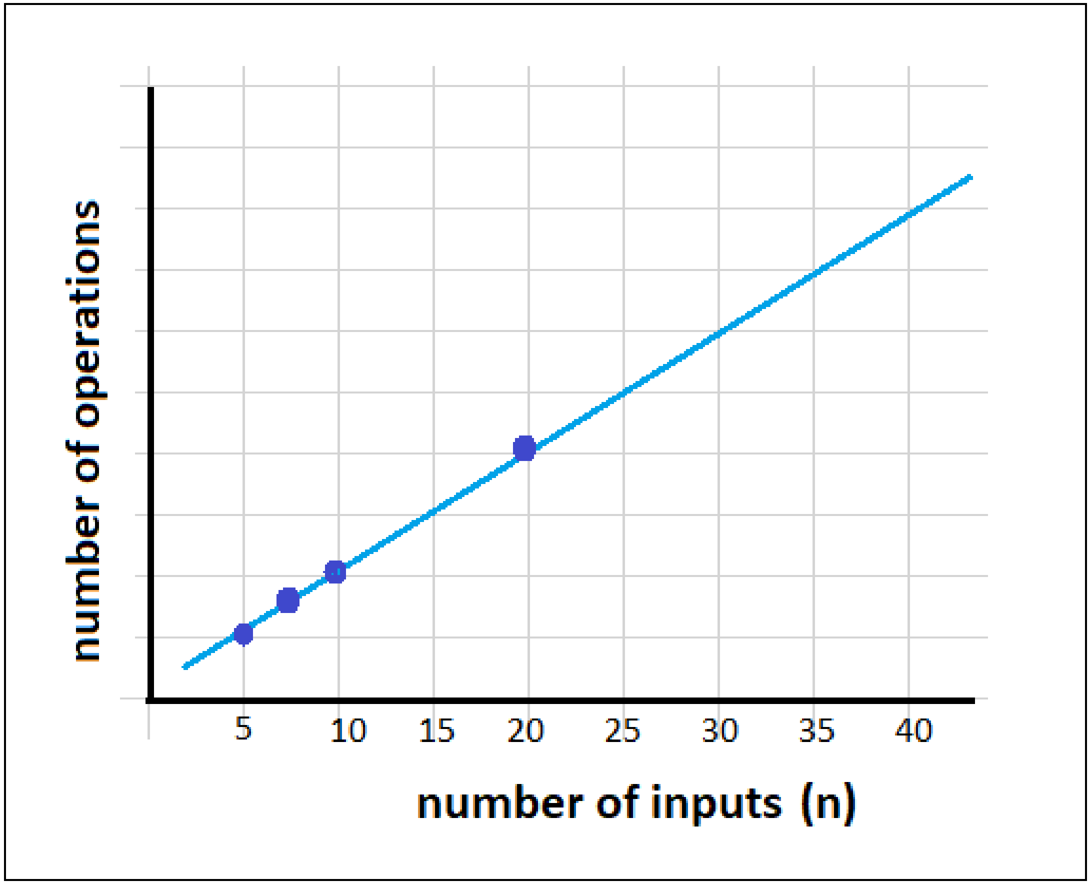
{ 2, 4, 6, 8, 10, 12 }

{ 71, 25, 83, 13, 43, 7, 92, 6, 6, 10 }

{ 1, 5, 3, 11, 4, 8, 2, 2, 66, 99, 1,5, 8}



Linear Time Complexity



O(n)

(The time complexity is O-of-n)

Drop Coefficient

Another method:

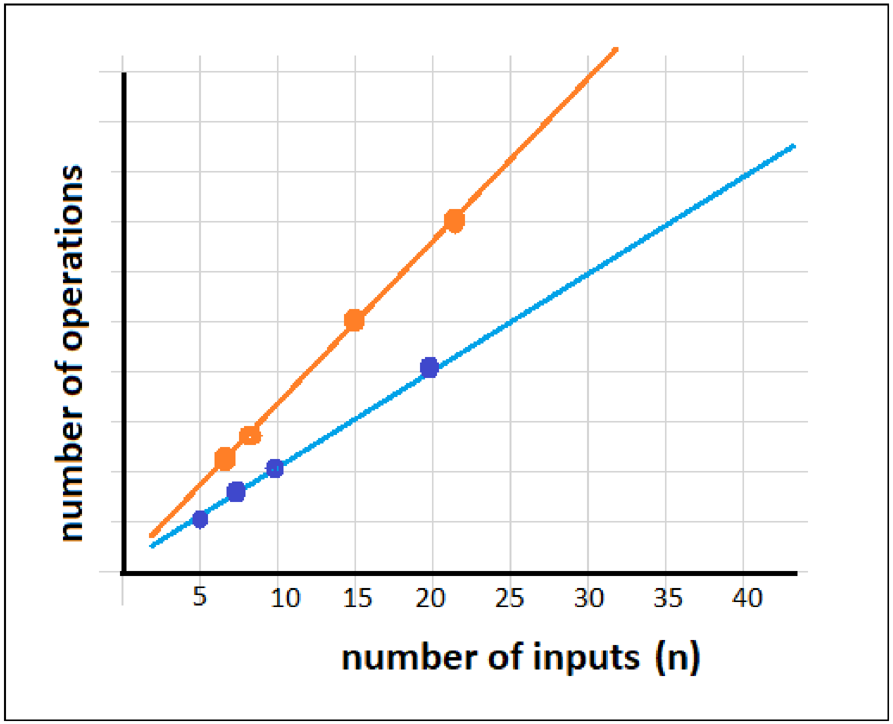
```
void printItems(int[] givenArray) {
   for (int i=0; i<givenArray.length; i++)
       System.out.println(givenArray[i]);
   for (int i=0; i<givenArray.length; i++)
       System.out.println(givenArray[i]);
}
```

Number of operations = **n + n = 2n**

Drop Coefficient

First rule of simplification of Big O is to *drop the coefficient*:

Analysis of Time Complexity focuses on the "trend" as the input grows *bigger*.



n + n = 2n

➔ O(n)

Fastest Growing Trend

Second rule of simplification of Big O is to take the *fastest growing* term as it is most significant when the input gets larger.

$$5 + n \rightarrow O(n)$$

$$5 + 99n + n^2 \rightarrow O(n^2)$$

Quadratic Time Complexity

Method with 2 nested for -loops:

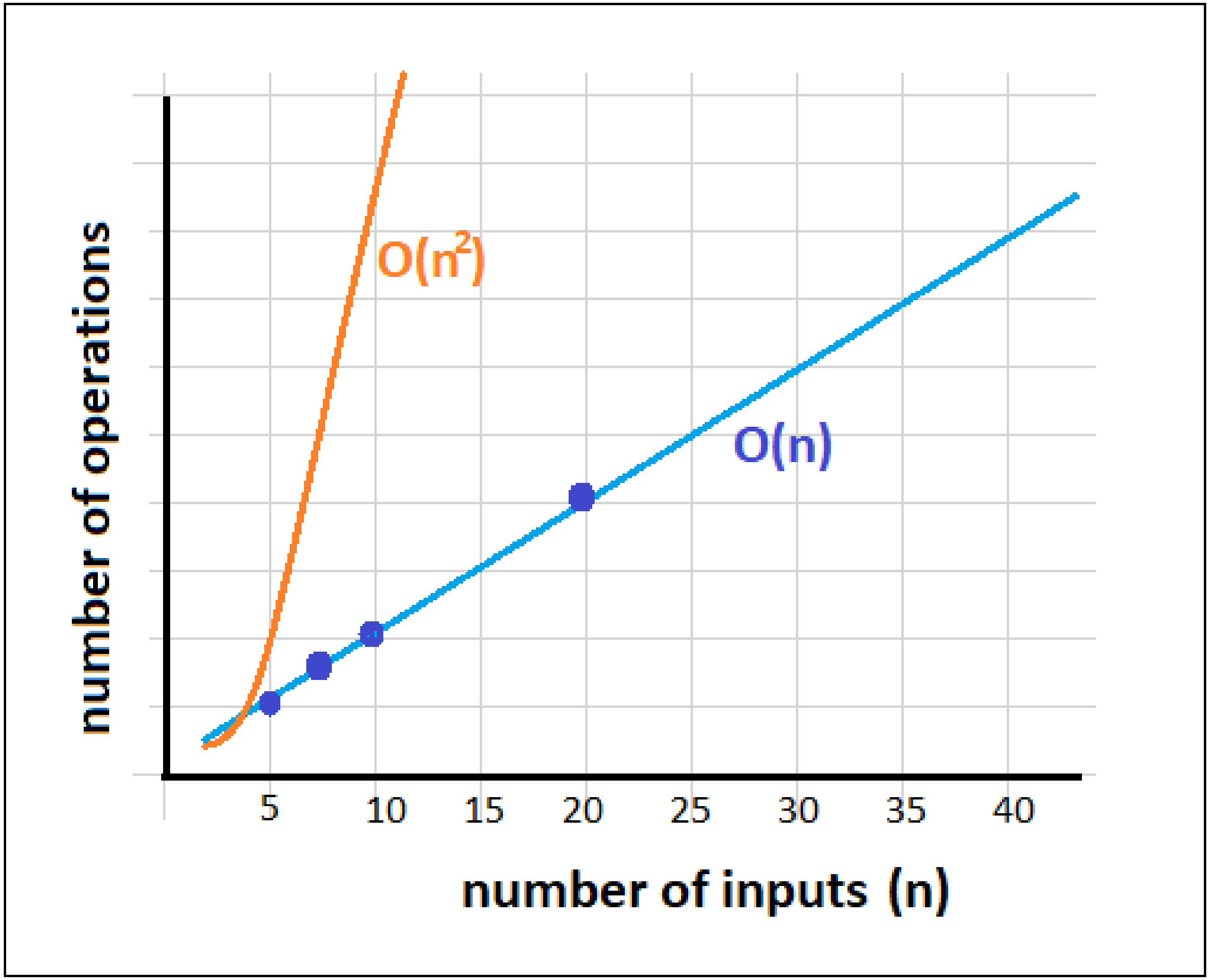
```
void printItems(int n ) {
   for (int i=0; i< n; i++)
     for (int j=0; j< n; j++)
       System.out.println( i + “ ” + j );
}
```

Number of operations = **n*n = n²**

O(n²)

printItems(10):

```
0 0
0 1
0 2
0 3
:
:
0 9
1 0
1 1
:
:
9 7
9 8
9 9
```



Time complexity:

Linear ➔ O(n)

Quadratic ➔ O(n^2)

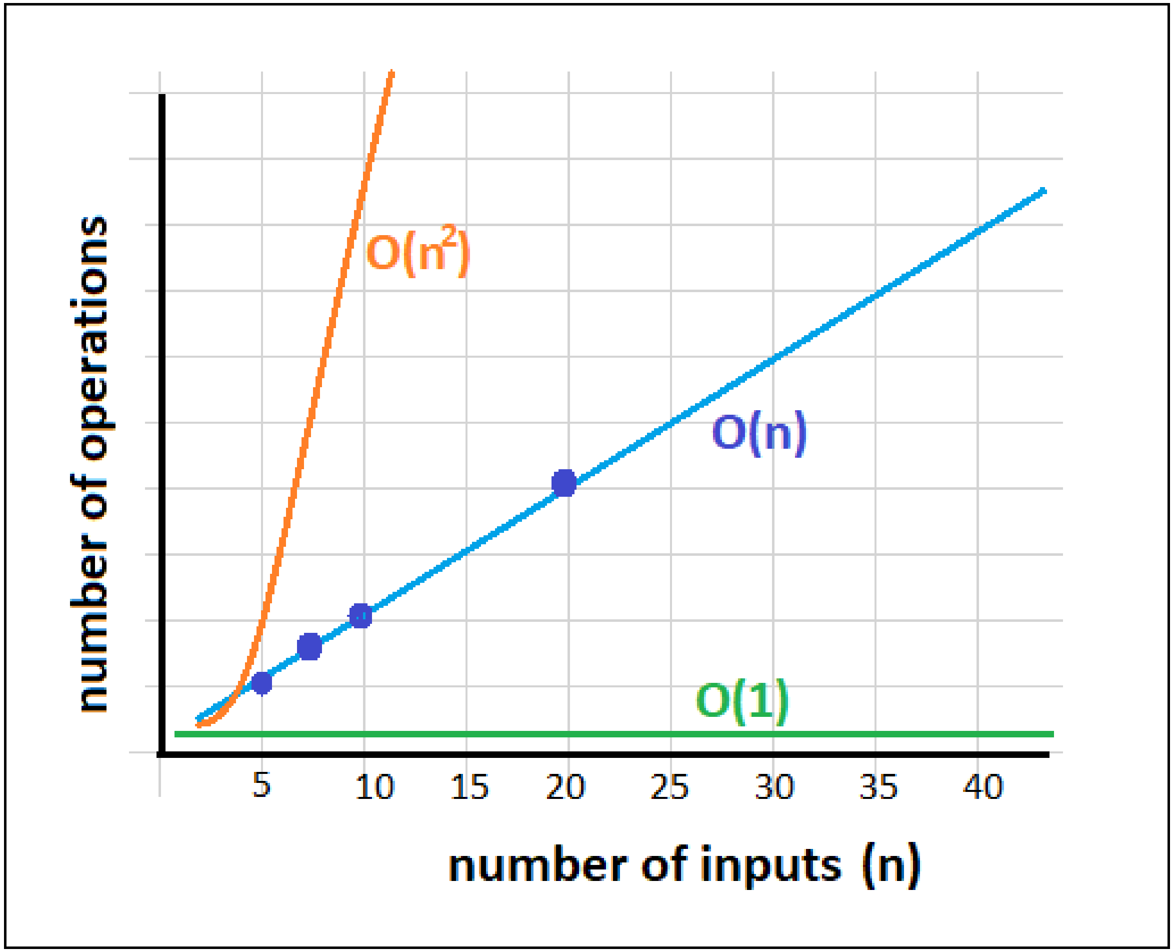
Constant Time Complexity

Consider this method:

```
int squareLength (int[] a) {
    return (a.length * a.length);
}
```

Regardless of the size of the input, there is always only 1 operation.

O(1)



Time complexity:

Linear ➔ O(n)

Quadratic ➔ $O(n^2)$

Constant ➔ O(1)

Logarithmic Time Complexity

Consider this method:

```
void printItems(int n ) {
   for (int i=1; i< n; i=i*2)
     System.out.println( i );
}
```

printItems(16):

```
1
2
4
8
```

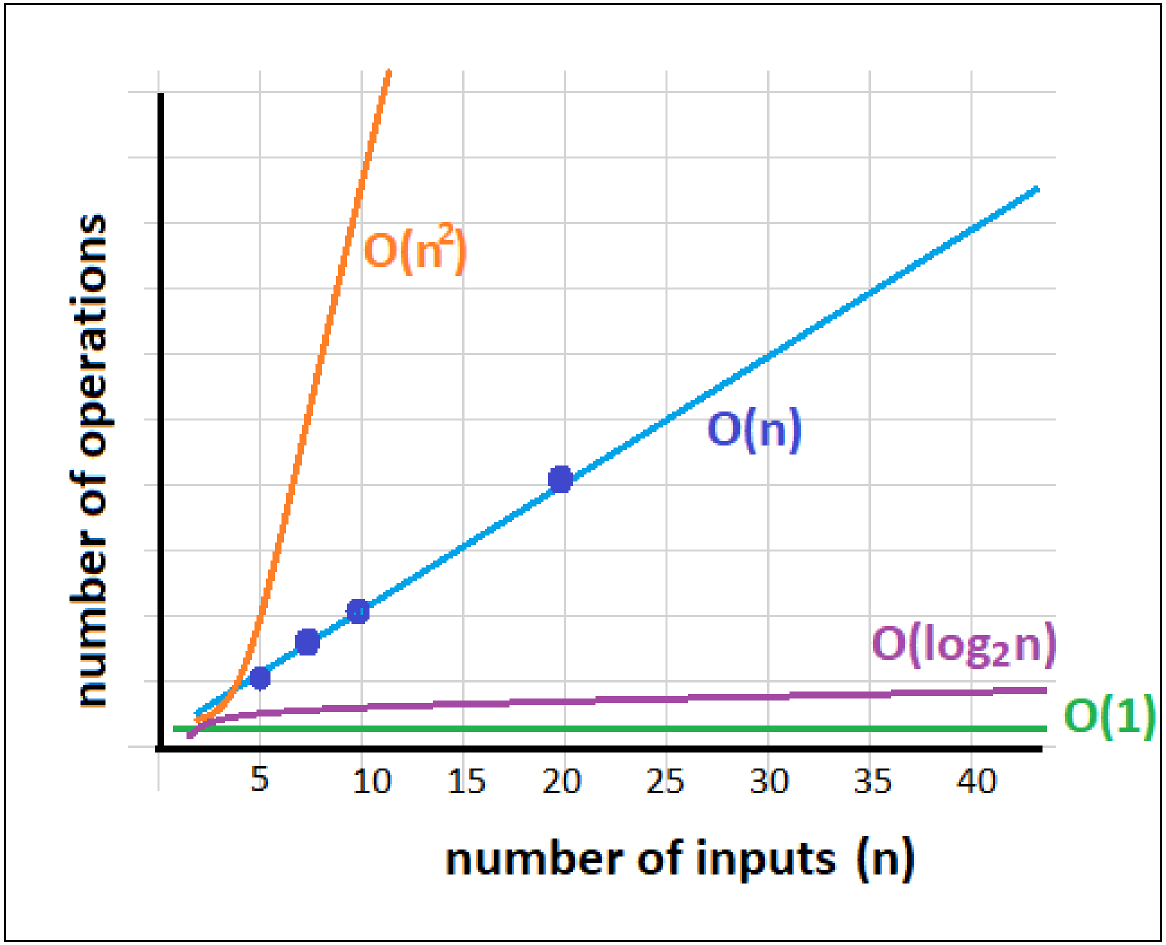
printItems(64):

```
1
2
4
8
16
32
```

Number of operations = $\log_2 n$

$$O(\log_2 n)$$

n	$\log_2 n$
8	$\log_2 8 = 3$
16	$\log_2 16 = \mathbf{4}$
32	$\log_2 32 = 5$
64	$\log_2 64 = \mathbf{6}$



Time complexity:

Linear ➔ O(n)

Quadratic ➔ O(n^2)

Constant ➔ O(1)

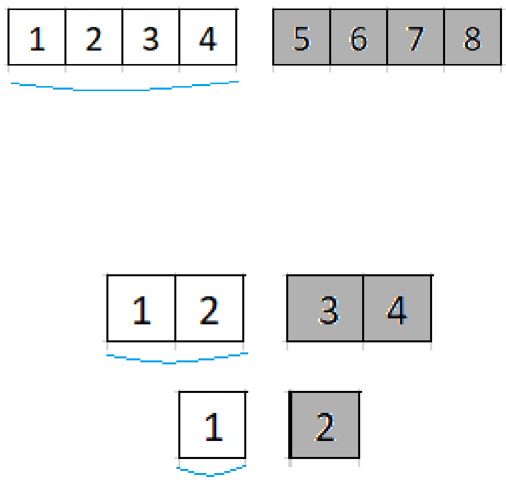
Logarithmic ➔ O(log_2n)

Binary Search

Binary Search (on sorted list of 8) algorithm.

1	2	3	4	5	6	7	8

Assuming searching for the value 1



1^{st} iteration

Search space: 8

Check the mid-point value;

No actual mid-point. Use either [3] (value 4) or [4] (value 5). No match.

Half the search space. As 1 is smaller than [3] or [4], continue on the lower half.

2^{nd} iteration

Search space: 4

Repeat the process. Mid-point value is not the value of 1 that we are looking for yet.

Continue splitting the search space.

3^{rd} iteration

Search space: 1

Found!

Binary Search

- Best case: key is found during the 1^{st} round of iteration
- Worst case: key is found at the last round of iteration or key not found

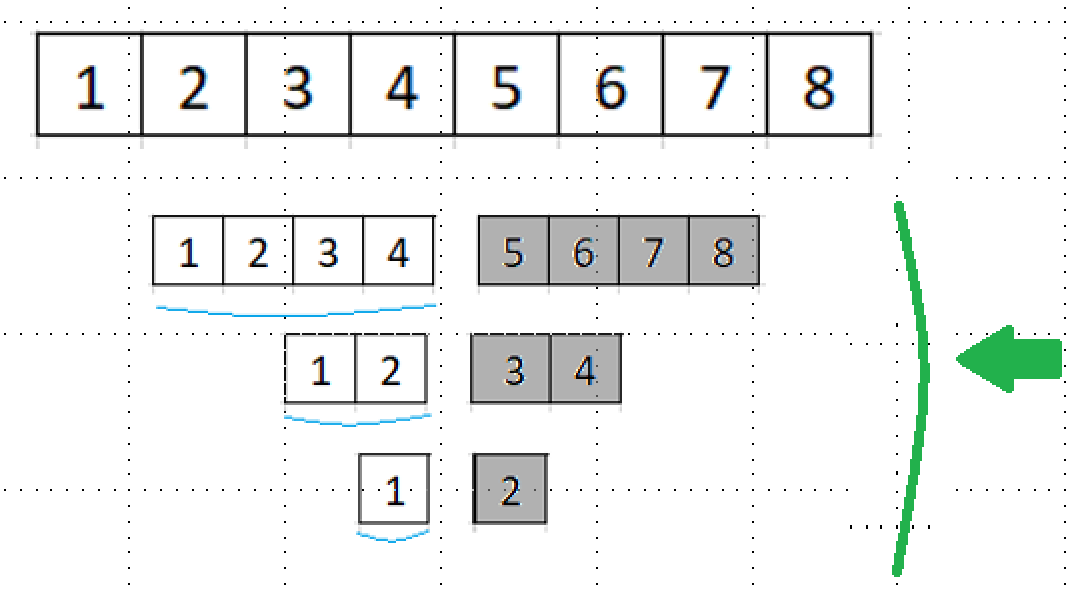
E.g. Search space 40 integers (stop if match is found)

1^{st} iteration – Search space:40; check the mid-point value;
2^{nd} iteration - Search space:20; check the mid-point value
3^{rd} iteration - Search space:10; check the mid-point value
4^{th} iteration - Search space:5; check the mid-point value
5^{th} iteration – Search space: 3 or 2 ; check the mid-point value
6^{th} iteration – need 1 last comparison

E.g. Search space 100 integers (stop if match is found)

7^{th} iterations

Logarithmic Time Complexity



Size n = 8

Number of operations = **3** = **$\log_2 n$**

$O(\log_2 n)$