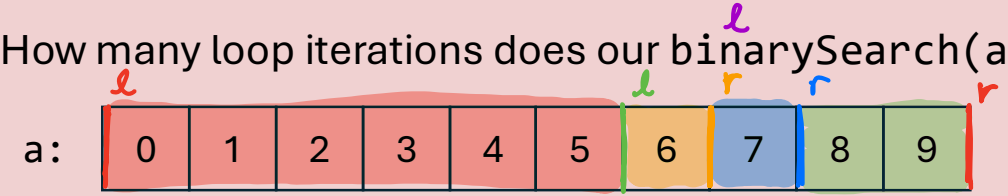
Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

How many loop iterations does our `binarySearch(a,7)` run?




```
1  static int binarySearch(int[] a, int v) {
2    int l = 0;    int r = a.length;
3    /* Loop inv: `a[..l) < v`, `a[r..] >= v` */
4    while (l < r) {
5      int m = l + (r - l) / 2;
6      if (a[m] < v) {  l = m + 1;
7      } else {  r = m; }
8    }
9    return r;
10 }
```

l	r	m
0	10	5
6	10	8
6	8	7
6	7	6
7	7	

3	(A)
4	(B)
5	(C)
6	(D)

Binary Search Runtime Analysis

```
1  static int binarySearch(int[] a, int v) {
2    int l = 0;   int r = a.length;
3    /* Loop inv: `a[..l) < v`, `a[r..] >= v` */
4    while (l < r) {
5      int m = l + (r - l) / 2;
6      if (a[m] < v) {  l = m + 1;
7      } else {  r = m; }
8    }
9    return r;
10 }
```

O(1)

O(1)

O(1)

Runtime = O(# iterations)

- In every iteration, "window" $[l, r)$ shrinks to $\leq$ half its old size.
- Stop when $r - l < 1$

$$N \cdot \left(\frac{1}{2}\right)^{\#\text{ iterations}} \geq 1$$

so # iterations $\leq \log_2(N)$

$= O(\log N)$

$O(\log N)$ runtime

Space Complexity

= the maximum amount of memory needed *at any point in time* during a method's execution (beyond the memory used to store its parameters).

the caller was responsible for this memory allocation

- "scratch space for computation"
- can reuse space later (unlike time)
- all methods we saw last lecture had O(1) space complexity
- gets more subtle when we analyze recursive methods

Lecture 6: Recursion

CS 2110

September 10, 2026

Today's Learning Outcomes

10. Develop recursive methods in Java from their specifications.

27. Determine the number of recursive calls and the maximum depth of the call stack of a recursive method and use these to compute its time and space complexities.

Recursive Methods

A method is **recursive** if it can be invoked from within its own definition.

Recursion offers another way, in addition to loops, to achieve conditional repetition of code.

Loops	Recursion
Loop Vars	Parameters
Loop Body	Method Body
Loop Invariant	Method Specs
Loop Guard	Base Case(s)

"simplest" inputs whose result is computed directly

When we write a recursive method, we're both its implementer and its client!

Computing Factorials

The **factorial** of an integer n ≥ 0 is the product of all positive integers ≤ n.

Ex. $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$

$6! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 = 720$

$1! = 1$

$0! = 1$ (mult. identity)

```java
/** Returns `n!`. Requires `0 <= n <= 12` */
static int factorial(int n) {
  int product = 1;
  /* loop inv: product = (i-1)!                */
  for (int i = 1; i <= n; i++) {
    product *= i;
  }
  return product;
}
```

A Recursive Implementation

Base Case: $0! = 1$ (also $1! = 1$) so factorial(n) should directly return 1 for $n \leq 1$

Recursive Case: Delegate work to recursive call

Ask "How can calling factorial() on a smaller input help compute it for a larger input?"

$5! = \underbrace{1 \cdot 2 \cdot 3 \cdot 4}_{4!} \cdot 5$

recursive call

More generally, $n! = (n-1)! \cdot n$

Coding Demo: Recursive factorial( )

Poll Everywhere

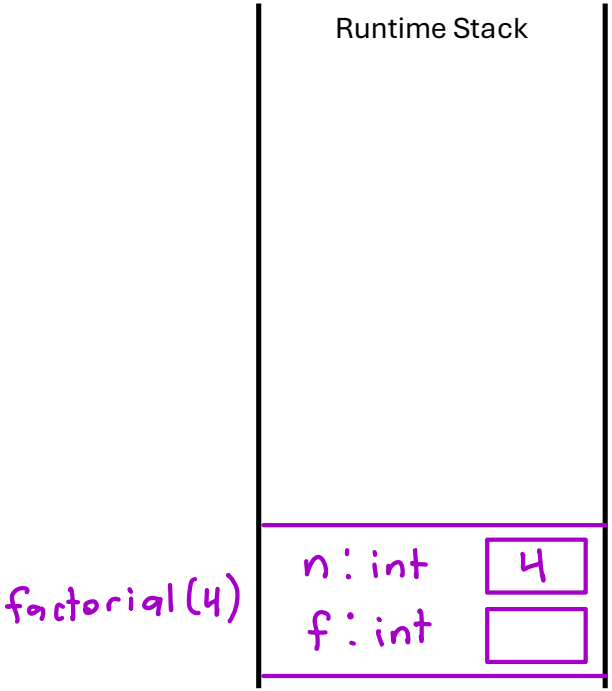
PollEv.com/javabear text **javabear** to 22333

How many total call frames are created when we evaluate `factorial(4)`?

```
1  static int factorial(int n) {
2    assert 0 <= n && n <= 12;
3    if (n <= 1) { return 1; }
4    int f = factorial(n - 1);
5    return n * f;
6  }
```

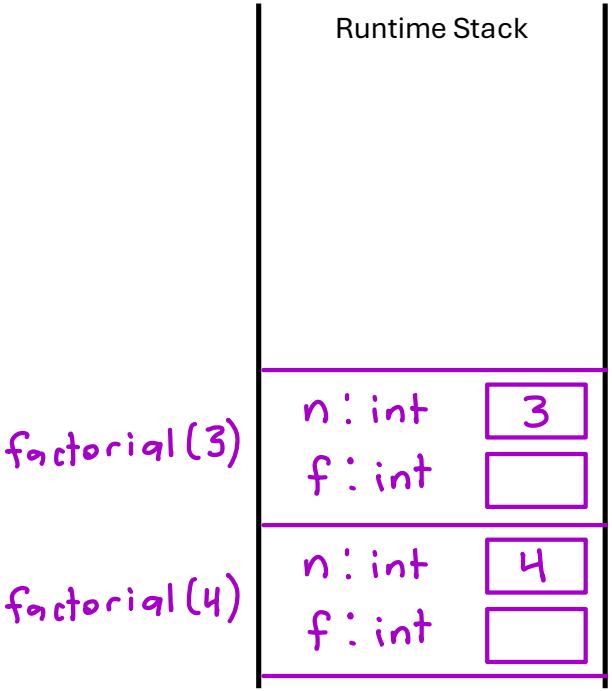
1	(A)
3	(B)
4	(C)
5	(D)

Visualizing Recursion: `factorial(4)`



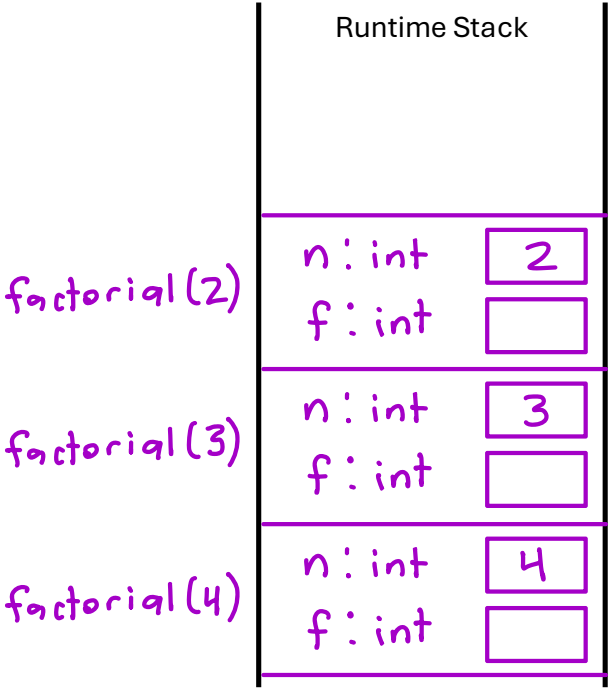

```
1  static int factorial(int n) {
2    assert 0 <= n && n <= 12;
3    if (n <= 1) { return 1; }
4    int f = factorial(n - 1);
5    return n * f;
6  }
```

Visualizing Recursion: `factorial(4)`



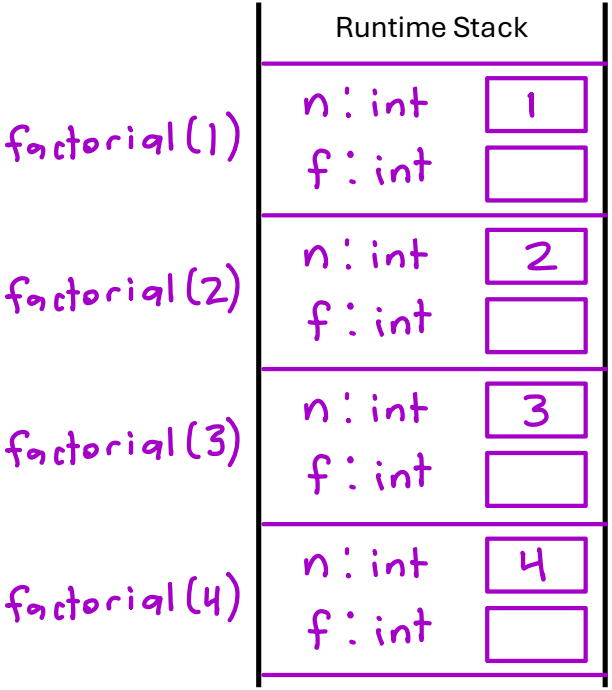

```
1  static int factorial(int n) {
2    assert 0 <= n && n <= 12;
3    if (n <= 1) { return 1; }
4    int f = factorial(n - 1);
5    return n * f;
6  }
```

Visualizing Recursion: `factorial(4)`



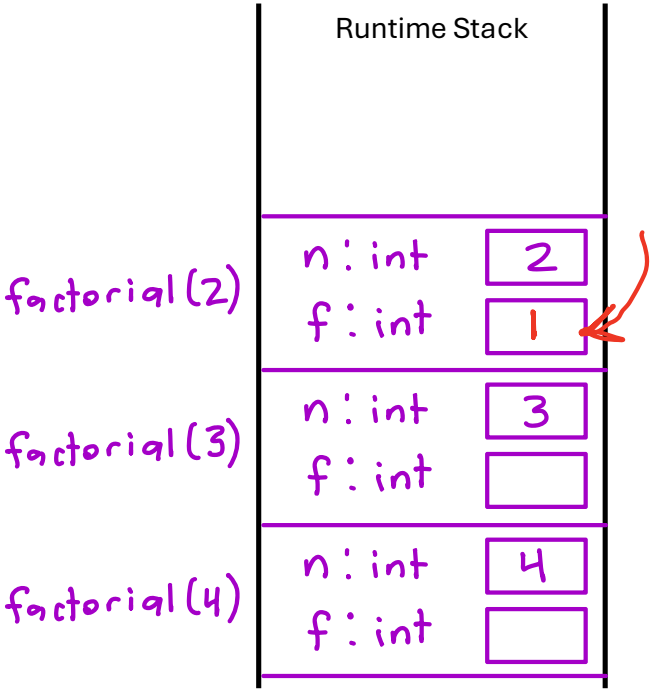

```
1  static int factorial(int n) {
2    assert 0 <= n && n <= 12;
3    if (n <= 1) { return 1; }
4    int f = factorial(n - 1);
5    return n * f;
6  }
```

Visualizing Recursion: `factorial(4)`



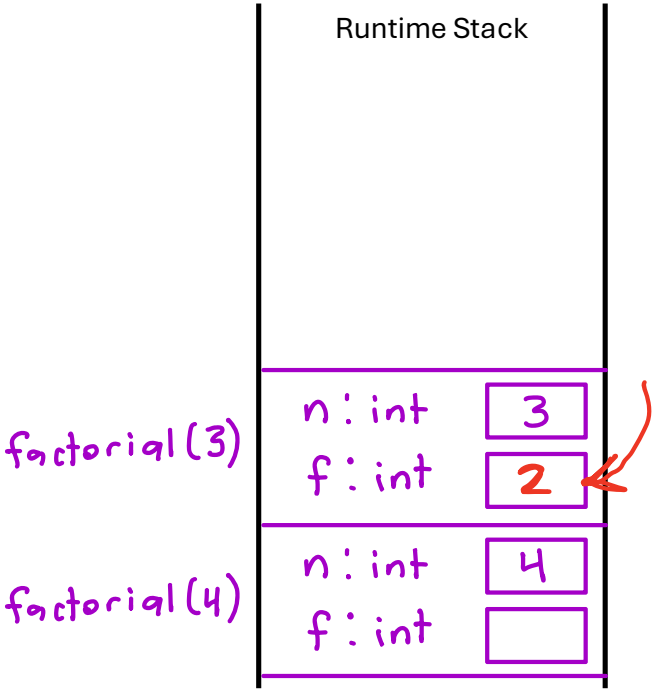

```
1  static int factorial(int n) {
2    assert 0 <= n && n <= 12;
3    if (n <= 1) { return 1; }
4    int f = factorial(n - 1);
5    return n * f;
6  }
```

Visualizing Recursion: `factorial(4)`



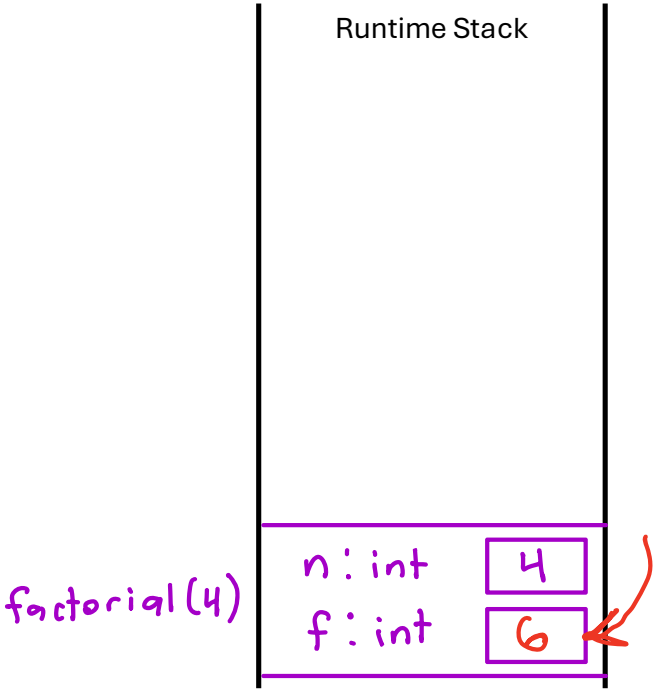

```
1  static int factorial(int n) {
2    assert 0 <= n && n <= 12;
3    if (n <= 1) { return 1; }
4    int f = factorial(n - 1);
5    return n * f;
6  }
```

Visualizing Recursion: `factorial(4)`




```
1  static int factorial(int n) {
2    assert 0 <= n && n <= 12;
3    if (n <= 1) { return 1; }
4    int f = factorial(n - 1);
5    return n * f;
6  }
```

Visualizing Recursion: `factorial(4)`




```
1  static int factorial(int n) {
2    assert 0 <= n && n <= 12;
3    if (n <= 1) { return 1; }
4    int f = factorial(n - 1);
5    return n * f;
6  }
```

Visualizing Recursion: `factorial(4)`

Runtime Stack

```
1  static int factorial(int n) {
2    assert 0 <= n && n <= 12;
3    if (n <= 1) { return 1; }
4    int f = factorial(n - 1);
5    return n * f;
6  }
```

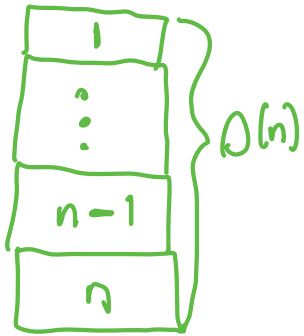
returns 24

Time Complexity of Recursive Code

We must account for the operations performed across **all** the recursive calls (not just the outermost one).

To do this, we'll determine 2 things:

1. "Non-Recursive" work done in each call (as a function of the parameters)
   O(1) for factorial()
2. Recursive call structure



Add non-recursive work over all calls
O(n) for factorial()

Space Complexity of Recursive Code

Again, there are two things to consider:

1. Additional space of objects constructed by any calls (none for factorial())
2. Space taken up by call frames

   O(1) * max # call frames present at any time during the execution

   = "recursive depth"

For factorial(): 0 + O(1) * n = O(n)

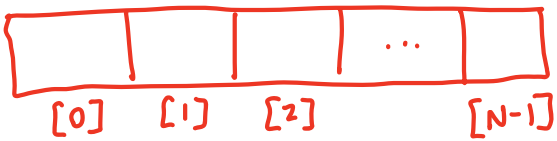
The additional space from all stack frames often makes recursion less efficient than iteration.

Recursion on Arrays

```
/** Returns the maximum value in `nums`. Requires that `nums` is non-empty. */
static double maxValue(double[] nums) { ... }
```

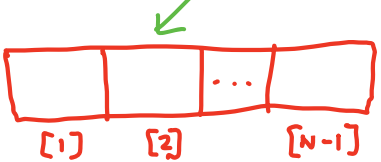
Base Case: If nums.length is 1, then the only value is the max value

Recursive Case:

max(nums 

)

Recursive call on smaller array

= max(nums[0], max(nums 

))

Coding Demo: Recursive `maxValue()`

Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

What are the time and space complexities of our recursive `maxValue()` implementation?

Each of O(N) recursive calls used O(N) space and did O(N) work

Time Complexity: **O(N)**	Space Complexity: **O(N)**	**(A)**
Time Complexity: **O(N)**	Space Complexity: **O(N^2)**	**(B)**
Time Complexity: **O(N^2)**	Space Complexity: **O(N)**	**(C)**
Time Complexity: **O(N^2)**	Space Complexity: **O(N^2)**	**(D)**

Array Views

Constructing a new, smaller array to pass into a recursive call is expensive!

Instead, we'd like to use only one array (passing an alias reference) and "tell" the recursive call to "only look at certain entries".

Solution: Use additional method parameters to define smaller view of same array

```
maxValueRecursive(double[] nums, int begin) { }
```

Coding Demo: `maxValue()`, Take 2

Poll Everywhere

PollEv.com/javabear text **javabear** to 22333

What are the time and space complexities of our new `maxValue()` implementation?

Each of O(N) recursive calls used O(1) space and did O(1) work

- Time Complexity: **O(N)** Space Complexity: **O(N)** **(A)**
- Time Complexity: **O(N)** Space Complexity: **O(N^2)** **(B)**
- Time Complexity: **O(N^2)** Space Complexity: **O(N)** **(C)**
- Time Complexity: **O(N^2)** Space Complexity: **O(N^2)** **(D)**

Another Recursive Method on Arrays

```
/** Returns true if there is a subset of entries from `coins` whose sum
 *  is equal to `total`, otherwise returns `false`. */
static boolean canMakeChange(int total, int[] coins) { ... }
```

Ex. canMakeChange(30, 

) = true

canMakeChange(28, 

) = false

* Think about base cases + recursive calls *

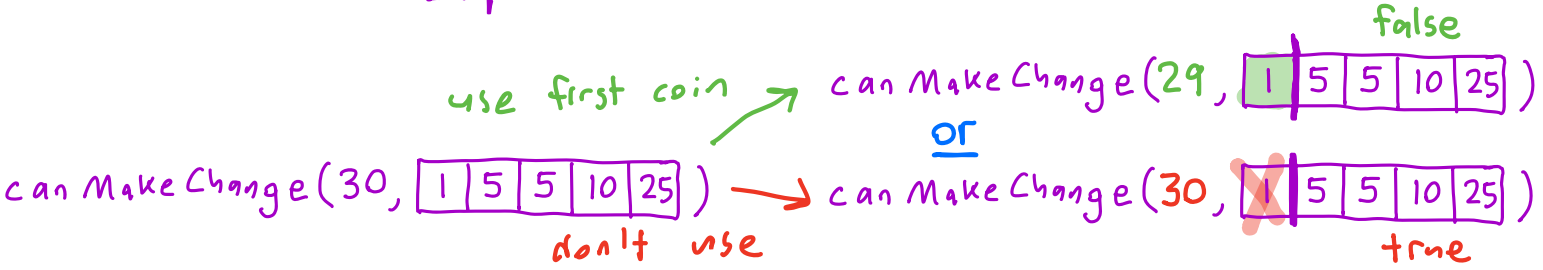
Designing the Recursion

```
/** Returns true if there is a subset of entries from `coins` whose sum
 *  is equal to `total`, otherwise returns `false`. */
static boolean canMakeChange(int total, int[] coins) { ... }
```

"Simpler" problem = fewer coins

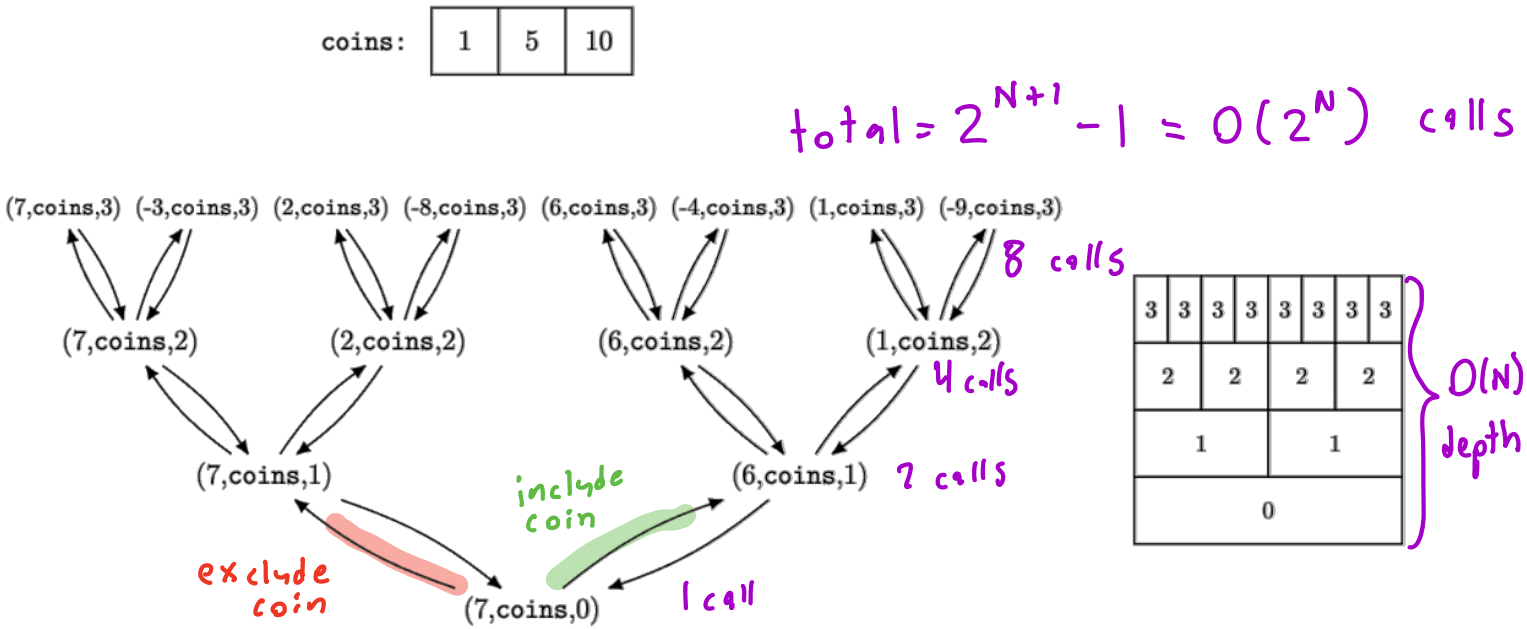
Base Case: if coins is empty, return (total == 0)

Recursive Cases: Explore two scenarios for first coin



Coding Demo: `canMakeChange()`

Visualizing the Call Structure



Time and Space Complexity

Each execution of canMakeChangeRecursive() does O(1) non-recursive work and allocates O(1) memory

time complexity = O(1) * # calls = $O(2^N)$ exponential time ☹

space complexity = O(1) * recursive depth = O(N)