Data Structure

Second Stage – C22023

BIG O

06

Big O notation: **is the language we use for talking about how long an algorithm takes to run. It's how we compare the efficiency of different approaches to a problem. It is used in Computer Science to describe the performance or complexity of an algorithm. Big O specifically describes the worst-case scenario, and can be used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm.**

Imagine you have a list of 10 objects, and you want to sort them in order. There's a whole bunch of algorithms you can use to make that happen, but **not all algorithms are built equal.**

It is simply a **way of comparing algorithms** and how long they will take to run.

.

A simplified analysis of algorithm efficiency. It considers:

1. Complexity in terms of input size, N.
2. Machine independence(Doesn’t consider the machines hardware)
3. Basic computer steps.
4. Time & space(working storage)

O means *order of* (German: "Ordnung von"). **n** is the input size in units of bits needed to represent the input

Measure Algorithm Efficiency

Space utilization: amount of memory required.

Time efficiency: amount of time required to accomplish the task.

As space is not a problem nowadays

- Time efficiency is more emphasized.
- But, in embedded computers or sensor nodes, space efficiency is still important.

General Rules

- **Ignore constants (Considers long term growth only)**
  - 5n becomes O(n)
- **Terms are dominated by others**
  - $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$
    - Ignore lower order terms

O(1)

- O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.

O(N)

- O(N) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set.
- Big O notation will always assume the upper limit where the algorithm will perform the maximum number of iterations.

$O(N^2)$

- $O(N^2)$ represents an algorithm whose performance is directly proportional to the square of the size of the input data set.
- This is common with algorithms that involve nested iterations over the data set.
- Deeper nested iterations will result in $O(N^3)$, $O(N^4)$ etc.

O(2^N)

- O(2^N) denotes an algorithm whose growth doubles with each addition to the input data set.
- An example of an O(2^N) function is the recursive calculation of Fibonacci numbers.

O(log N).

- The iterative halving of data sets described in the binary search example produces a growth curve that peaks at the beginning and slowly flattens out as the size of the data sets increase.
- e.g. an input data set containing 10 items takes one second to complete, a data set containing 100 items takes two seconds, and a data set containing 1000 items will take three seconds.

Constant Time Algorithms – *O(1)*

Time taken per <u>input size</u>

Constant time implies that the number of operations the algorithm needs to perform to complete a given task is independent of the input size.

In Big O notation we use the following **O(1)**.

No matter how much data you add the amount of time to perform a task will not change.

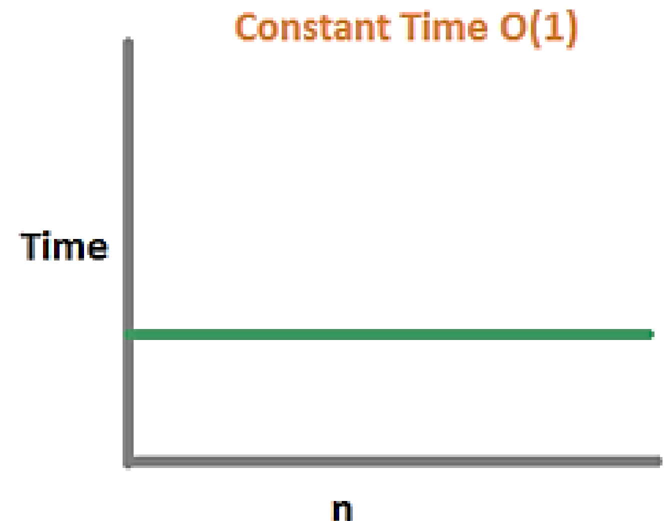
X = 10 + (5 * 2);

Input size(X) is independent

doesn't matter

- **doesn't effect processing of code/algorithm**

<u>So it's O(1)</u>



Constant Time Algorithms – *O(1)*

```
X = 10 + (5 * 2);
Y = 20 – 2;
System.out.print(”x + y”);
```

Total time is = O(1) + O(1) + O(1) = 3 * O(1)

Because we drop constants we would call this code O(1) “Big oh of one”

Best, average, worst-case complexity

In some cases, it is important to consider the best, worst and/or average (or typical) performance of an algorithm:

- **Worst, O(N) or o(N): $\geq$ or > true function**
- **Average, Θ(N): $\cong$ true function**
- **Best, Ω(N): $\leq$ true function**

E.g., when sorting a list into order, if it is already in order then the algorithm may have very little work to do

The worst-case analysis gives a bound for all possible input (and may be easier to calculate than the average case)

Best case: In the best-case analysis, we calculate the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be $\Omega(1)$.

Worst case: In the worst-case analysis, we calculate the upper bound on the running time of an algorithm. We must know the case that causes a maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x) is not present in the array. When x is not present, the search() function compares it with all the elements of arr[] one by one. Therefore, the worst-case time complexity of the linear search would be $O(n)$.

average case: In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs.

How do we calculate big-O?

Five guidelines for finding out the time complexity of a piece of code

1	Loops
2	Nested loops
3	Consecutive statements
4	If-then-else statements
5	Logarithmic complexity

Guideline 1: Loops

The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations.

executed *n* times

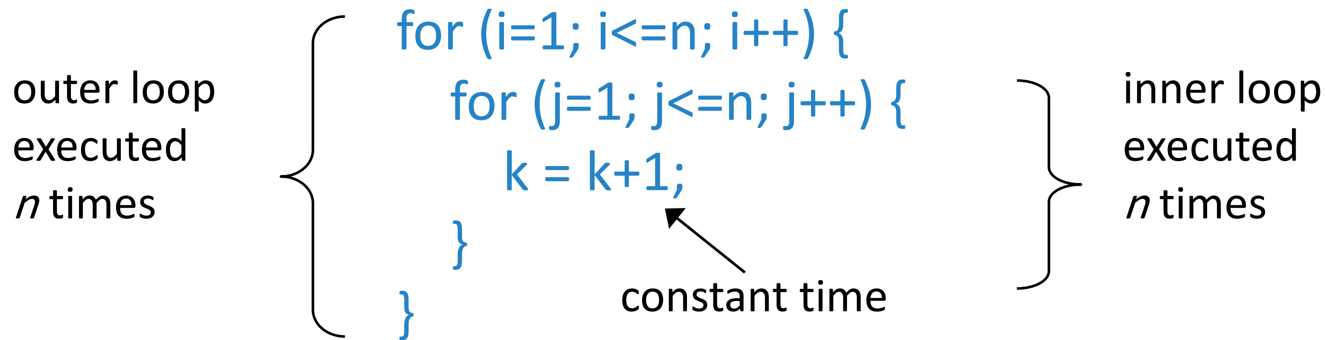
```
for (i=1; i<=n; i++)
{
    m = m + 2;
}
```

$m = m + 2;$ ← constant time

Total time = a constant c * n = c n = **O(*N*)**

Guideline 2: Nested loops

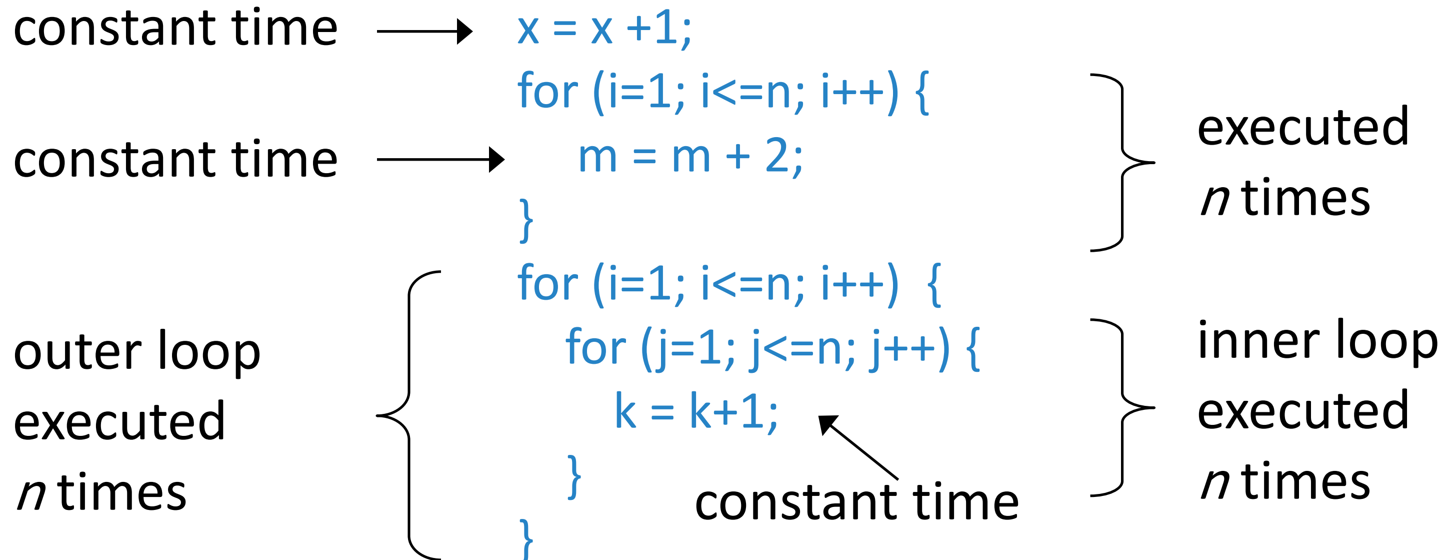
Analyse inside out. Total running time is the product of the sizes of all the loops.



Total time = c * n * n * = cn^2 = $\mathbf{O(\mathit{N}^2)}$

Guideline 3: Consecutive statements

Add the time complexities of each statement.

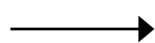


Total time = $c_0 + c_1 n + c_2 n^2 = \mathbf{O(N^2)}$

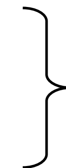
Guideline 4: If-then-else statements

Worst-case running time: the test, plus *either* the **then** part *or* the **else** part (whichever is the larger).

test:
constant



```
if (depth( ) != otherStack.depth( ) ) {
  return false;
}
else {
  for (int n = 0; n < depth( ); n++) {
  if (!list[n].equals(otherStack.list[n]))
    return false;
  }
}
```



then part:
constant

else part:
(constant +
constant) * n

another if :
constant +
constant
(no else part)

Total time = $c_0 + c_1 + (c_2 + c_3) * n = \mathbf{O(}\boldsymbol{N}\mathbf{)}$

Guideline 5: Logarithmic complexity

An algorithm is O(log N) if it takes a constant time to cut the problem size by a fraction (usually by ½)

Example algorithm (binary search):
finding a word in a dictionary of n pages

- **Look at the center point in the dictionary**
- **Is word to left or right of center?**
- **Repeat process with left or right part of dictionary until the word is found**

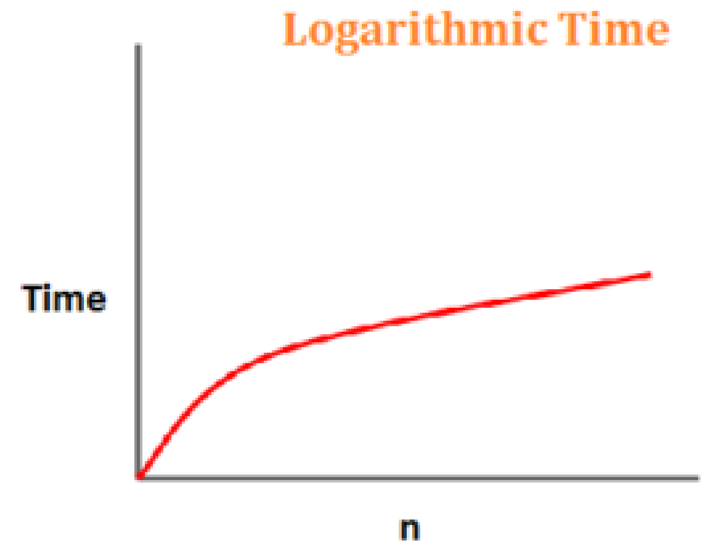
Logarithmic Time Algorithms – *O(log n)*

Logarithmic time implies that an algorithms run time is proportional to the logarithm of the input size. In Big O notation this is usually represented as O(log n).

Next fastest after constant time

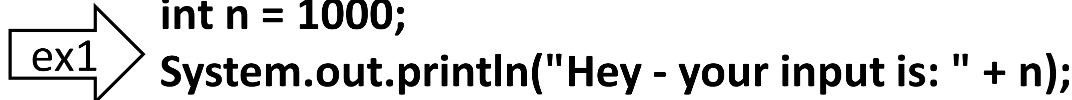
Any code that involves a divide and conquer strategy

Binary search algorithm is O(log n)



Examples - Constant Time Algorithms – *O(1)*

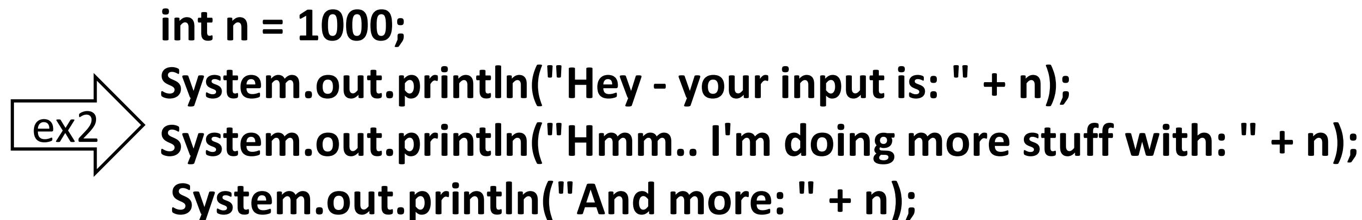
Consider this simple piece of code:




```
int n = 1000;
System.out.println("Hey - your input is: " + n);
```

Clearly, it doesn't matter what n is, above. This piece of code takes a constant amount of time to run. It's not dependent on the size of n.

Similarly:




```
int n = 1000;
System.out.println("Hey - your input is: " + n);
System.out.println("Hmm.. I'm doing more stuff with: " + n);
 System.out.println("And more: " + n);
```

The above example is also constant time. Even if it takes 3 times as long to run, it doesn't depend on the size of the input, n. We denote constant time algorithms as follows: O(1). Note that O(2), O(3) or even O(1000) would mean the same thing. We don't care about exactly how long it takes to run, only that it takes constant time.

Examples - Linear Time Algorithms – O(n)

ex1

```
for (int i = 0; i < n; i++) {
   System.out.println("Hey - I'm busy looking at: " + i);
   System.out.println("Hmm.. Let's have another look at: " + i);
   System.out.println("And another: " + i);
}
```

The runtime would still be linear in the size of its input, n. We denote linear algorithms as follows: **O(n).**

ex2

```
for (int i = 0; i < n; i++) {
   System.out.println("Hey - I'm busy looking at: " + i);
   System.out.println("Hmm.. Let's have another look at: " + i);
   System.out.println("And another: " + i);
}
```

The runtime would still be linear in the size of its input, *n*. We denote linear algorithms as follows: ***O(n)*.**

Examples - Logarithmic Time Algorithms – *O(log n),O(n log n)*

ex1

```
for (int i = 1; i < n; i = i * 2){
   System.out.println("Hey - I'm busy looking at: " + i);
}
```

If **n is 8**, the output will be the following:

Hey - I'm busy looking at: 1

Hey - I'm busy looking at: 2

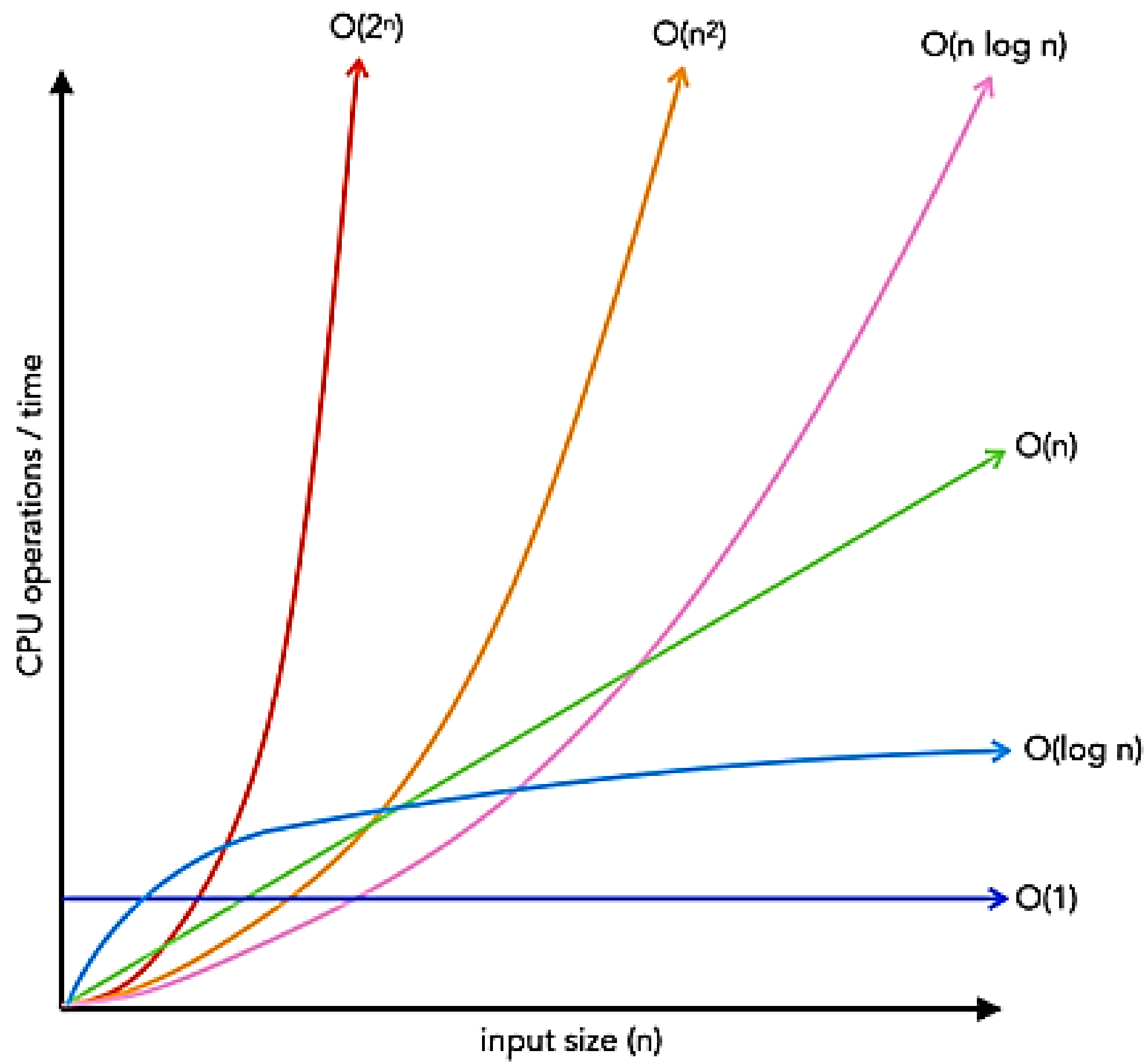
Hey - I'm busy looking at: 4

Our simple algorithm ran **log(8)** = 3 times. = **log (n)**

ex2

```
for (int i = 1; i <= n; i++){
   for(int j = 1; j < n; j = j * 2) {
      System.out.println("Hey - I'm busy looking at: " + i + " and " + j);
   }
}
```

For example, if the n is 8, then this algorithm will run 8 * log(8) = 8 * 3 = 24 times. Whether we have strict inequality or not in the for loop is irrelevant for the sake of a Big O Notation= **O(n log n)**


O(2ⁿ)
O(n²)
O(n log n)
O(n)
O(log n)
O(1)
CPU operations / time
input size (n)

n	Constant O(1)	Logarithmic O(log n)	Linear O(n)	Linear Logarithmic O(n log n)	Quadractic $O(n^2)$	Cubic $O(n^3)$
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4,096
1,024	1	10	1,024	10,240	1,048,576	1,073,741,824