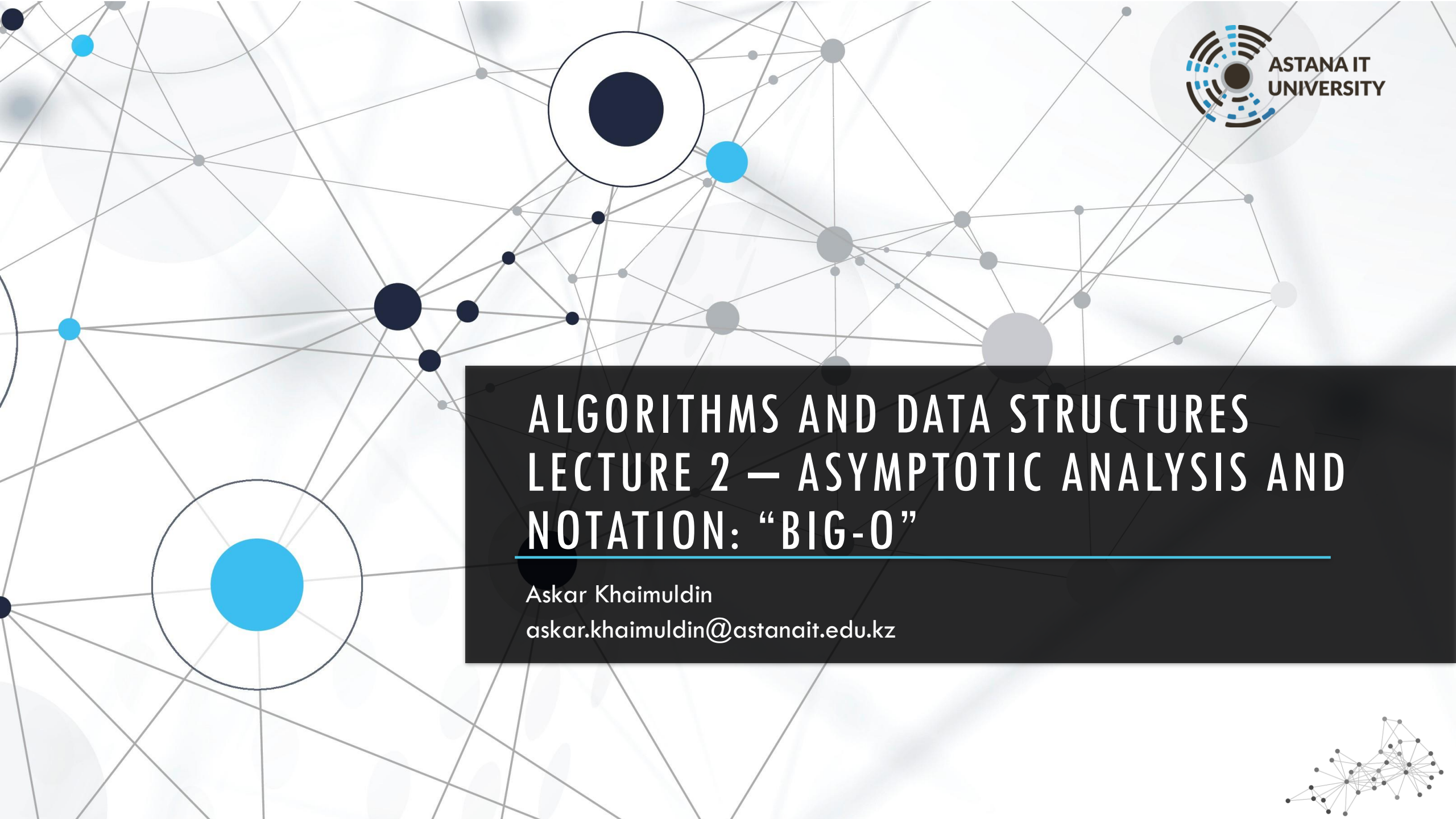


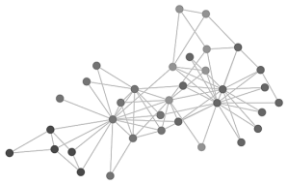
ALGORITHMS AND DATA STRUCTURES LECTURE 2 — ASYMPTOTIC ANALYSIS AND NOTATION: "BIG-O"

Askar Khaimuldin
askar.khaimuldin@astanait.edu.kz

ASTANA IT UNIVERSITY

CONTENT

1. Algorithm efficiency
2. Order of growth
3. Big-O notation
4. Analyzing the running time of a program (Example)
5. Complexity classes
6. Constant complexity
7. Logarithmic complexity
8. Linear complexity
9. Exponential complexity



ALGORITHM EFFICIENCY

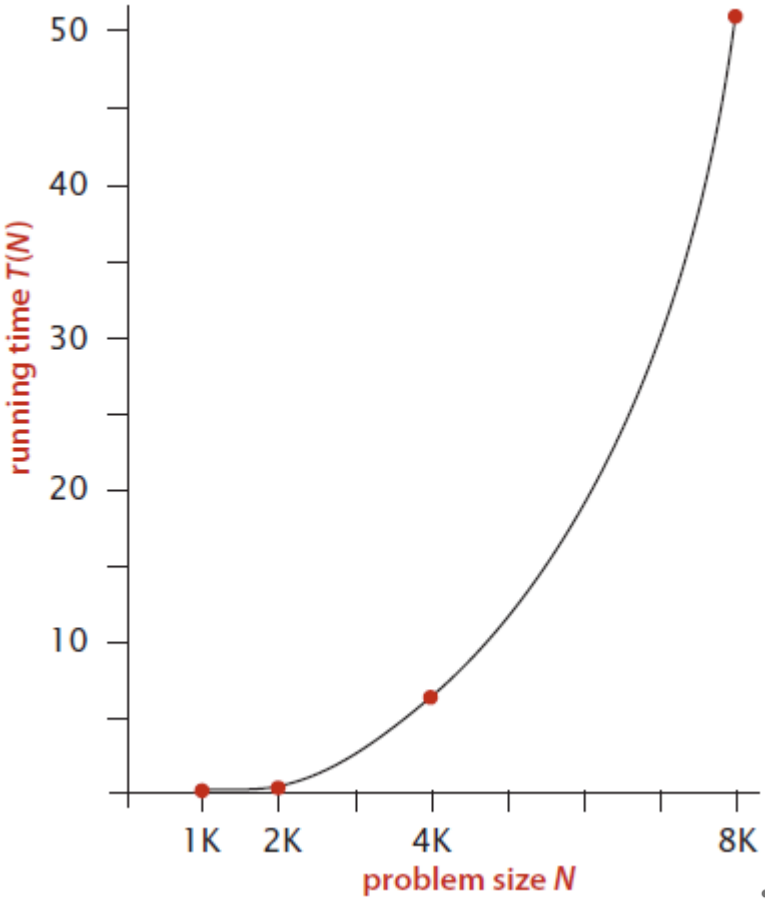
Computers are getting faster

- Does the efficiency matter?
- How about a very large dataset?

A program can be implemented in several ways

How to measure the efficiency?

- Using timer
- Count the number of operations
- **Order of growth**



ALGORITHM EFFICIENCY : USING TIMER

Timers are inconsistent

- Time varies for different inputs but cannot express a relationship between inputs and time
- Can be significantly different in various cases (because of CPU, memory, GC, operating system, network, etc.)

Our first challenge is to determine how to make quantitative measurements of the running time of our program using random numbers for input

Hint: Math.random()

```
long time1 = System.currentTimeMillis();
int triplesNum = countZeroTriples(arr);
long time2 = System.currentTimeMillis();

System.out.println(time2 - time1 + " ms");
```

5000	1000	100
30990 ms	256 ms	3 ms

```
// Calculate how many triples sum
// to exactly zero
public static int countZeroTriples(int[] arr) {
    int n = arr.length, count = 0;

    for (int i = 0; i < n; i++)
        for (int j = i+1; j < n; j++)
            for (int k = j+1; k < n; k++)
                if (arr[i] + arr[j] + arr[k] == 0)
                    count++;
    return count;
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    int n = sc.nextInt(); // number of elements

    int[] arr = new int[n];

    for (int i = 0; i < arr.length; i++) {
        arr[i] = sc.nextInt();
    }

    System.out.println(countZeroTriples(arr));
}
```

ALGORITHM EFFICIENCY : COUNT OPERATIONS

Assume that any of comparison, addition, value setting and retrieving, etc. is just 1 operation

How many of those operations do I use in my algorithm?

This could be used to give us a sense of efficiency

2 + 2 + (n+1) + n + n + 2n and 1 more for “return”

= **5n + 6 (worst case)**

The worst-case scenario: all elements are zeros

```
public static int countZeros(int[] arr) {
    int count = 0;
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == 0) {
            count++;
        }
    }

    return count;
}
```

operation	Sample cost in ns	frequency
variable declaration	2/5	2
assignment statement	1/5	2
less than compare	1/5	$n+1$
equal to compare	1/10	n
array access	10	n
increment	1/10	n to $2n$

ORDER OF GROWTH

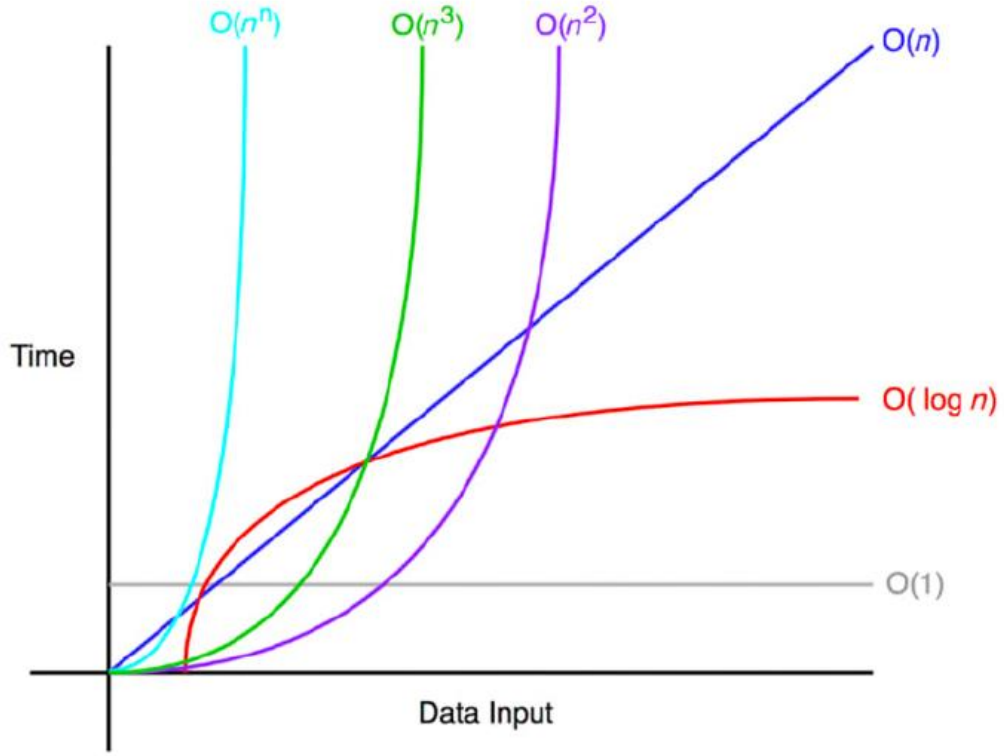
If f(n) ~ Cg(n) for some constant C > 0, then the order of growth of f(n) is g(n)

- Ignores leading coefficient
- Ignores lower-order terms
- Focuses on dominant terms

Examples:

1) $2n^3 + 5n^2 + 0.15n + 7 \approx n^3$

2) $2n^3 + 3^n \approx 3^n$



*Big-O notation is a mathematical notation that describes the **limiting behavior** of a **function** when the **argument** tends towards a particular value or infinity.*

BIG-O NOTATION

Law of addition: $O(f(n)) + O(g(n)) = O(f(n) + g(n))$

- Used with sequential statements

$$O(n) + O(n^2) = O(n + n^2) = O(n^2)$$

```
for (int number : arr) {
    System.out.println(number);
}

int N = arr.length;
for (int i = 0; i < N * N; i++) {
    System.out.println(i);
}
```

$O(n)$

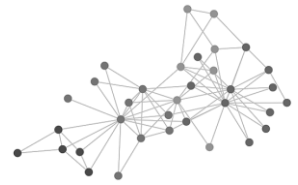
$O(n^2)$

Law of multiplication $O(f(n)) * O(g(n)) = O(f(n) * g(n))$

- Used with nested statements/loops or recursion

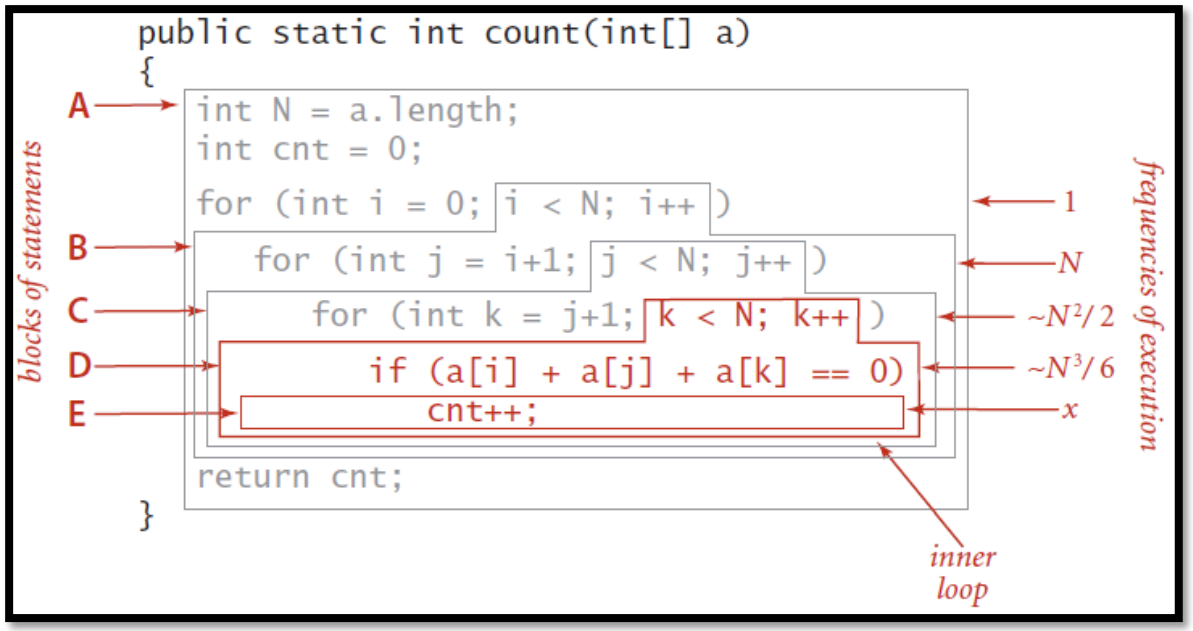
$$O(n) * O(n) = O(n^2)$$

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        System.out.println( i + j);
    }
}
```



ANALYZING THE RUNNING TIME OF A PROGRAM

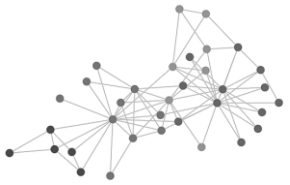
statement block	time in seconds	frequency	total time
E	t_0	x *(depends on input)*	$t_0 x$
D	t_1	$N^3/6 - N^2/2 + N/3$	$t_1 (N^3/6 - N^2/2 + N/3)$
C	t_2	$N^2/2 - N/2$	$t_2 (N^2/2 - N/2)$
B	t_3	N	$t_3 N$
A	t_4	1	t_4
		grand total	$(t_1/6)\, N^3 + (t_2/2 - t_1/2)\, N^2 + (t_1/3 - t_2/2 + t_3)\, N + t_4 + t_0 x$
		tilde approximation	$\sim (t_1/6)\, N^3$ *(assuming x is small)*
		order of growth	N^3



ASTANA IT UNIVERSITY

COMPLEXITY CLASSES

order of growth	name	typical code framework	description	example	$T(2n) / T(n)$
1	**constant**	`a = b + c;`	statement	add two numbers	1
$\log n$	**logarithmic**	`while (n > 1) { n = n/2; ... }`	divide in half	binary search	~ 1
n	**linear**	`for (int i = 0; i < n; i++) { ... }`	single loop	find the maximum	2
$n \log n$	**linearithmic**	*see mergesort*	divide and conquer	mergesort	~ 2
n^2	**quadratic**	`for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) { ... }`	double loop	check all pairs	4
n^3	**cubic**	`for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) for (int k = 0; k < n; k++) { ... }`	triple loop	check all triples	8
2^n	**exponential**	*see combinatorial search*	exhaustive search	check all subsets	2^n



CONSTANT COMPLEXITY

Complexity independent of inputs

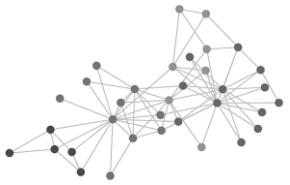
- No matter how many operations it performs exactly as long as it doesn`t depend on the number of inputs
- Time complexity is constant

```
public static int add(int a, int b) {
    return a + b;
}
```

Can have loops or recursive calls, but **only if** number of iterations or calls independent of input size

- It is still O(1) if it performs constant number of operations

```
public static void sayHello() {
    for (int i = 0; i < 1000; i++) {
        System.out.println("Hello");
    }
}
```



LOGARITHMIC COMPLEXITY

Complexity grows as log of size of one of its inputs

- When the number of iterations is divided to any constant K > 1 at each iteration (or halved in most cases)

```
public static void outputMiddles(int[] arr) {
    int middle = arr.length / 2;
    int sum;

    while (middle > 0) {
        System.out.println(arr[middle]);
        middle /= 2;
    }
}
```

```
10
43 -35 49 3 12 -39 -23 42 15 16
-39
49
-35
```

How about this?

```
public static int sumHalf(int[] arr) {
    int n = arr.length;
    int sum = 0;

    for (int i = 0; i < n / 2; i++) {
        sum += arr[i];
    }

    return sum;
}
```

It is not decreasing at each iteration

$$O\left(\frac{n}{2}\right) = O(n)$$

LINEAR COMPLEXITY

Can be in form of iterative loops or recursion

Iterative loops: depends on number of iterations

Recursion: depends on number of recursive function calls

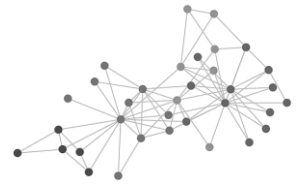
```java
public static int factorial(int N) {
    int product = 1;
    for (int i = 1; i <= N; i++) {
        product *= i;
    }

    return product;
}
```

```java
public static int factorial(int N) {
    if (N <= 1) return 1; // base case

    return factorial( N: N - 1) * N;

}
```

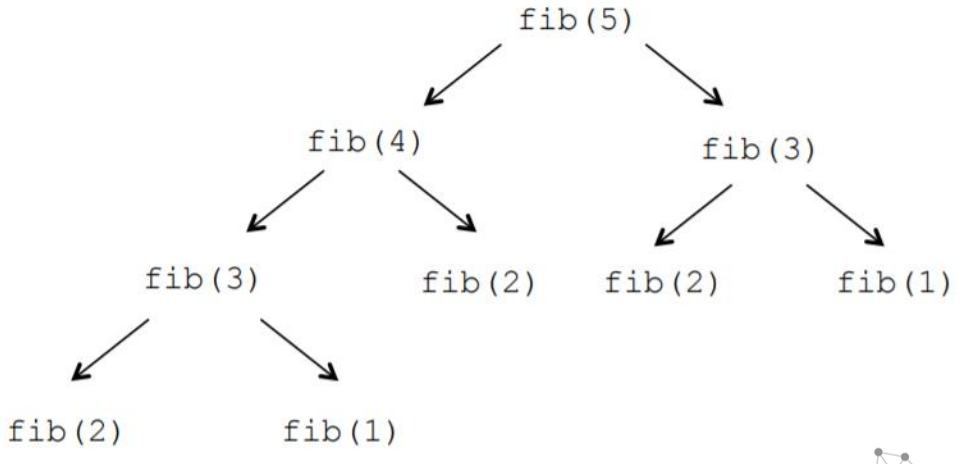


EXPONENTIAL COMPLEXITY

Exponential Time complexity denotes an algorithm whose growth is increasing exponentially (doubles in most cases) with each addition to the input data set

A good example is the recursive solution for Fibonacci Complexity is 2^n

```
public static int fib(int n) {
    if (n <= 1) return n; // base case
    // no need to write "else", since the
    // previous one will return
    return fib( n: n-2) + fib( n: n-1);
}
```

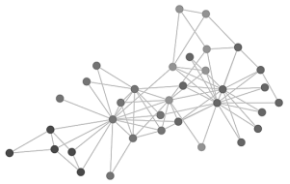


ASTANA IT UNIVERSITY

LITERATURE

Algorithms, 4th Edition, by Robert Sedgewick and Kevin Wayne, Addison-Wesley

- Chapter 1.4



ASTANA IT UNIVERSITY

GOOD LUCK!