DATA STRUCTURES & ALGORITHMS

Big O Notation

Instructor: Engr. Laraib Siddiqui

Asymptotic Analysis

- Asymptotic analysis deals with analyzing the properties of the running time when the input size goes to infinity (this means a very large input size).
- The differences between orders of growths are more significant for larger input size. Analyzing the running times on small inputs does not allow us to distinguish between efficient and inefficient algorithms.
- The objective of asymptotic analysis is to describe the behavior of a function T(N) as it goes to infinity.
- Asymptotic notations are used to describe the asymptotic analysis.

Function Bounds

Lets understand with the help of example. Suppose we have a function $10N^2$

Can we say it is bounded by $11N^2$ and $9N^2$ for all $N \geq 1$?

- i.e $10N^2$ cannot go above 11N2 and doesn't come down below 9N2 for all values of N. 10N2 is sandwiched between 9N2 and 11N2
- Now if f(n) is $10N^2$ and g(n) is N^2
- Then we say that f(n) is Θ (g(n))

Big Oh Notation

Sometimes we are only interested in proving one bound.

We use O-notation, when we have only an asymptotic upper bound.

Big Oh Notation

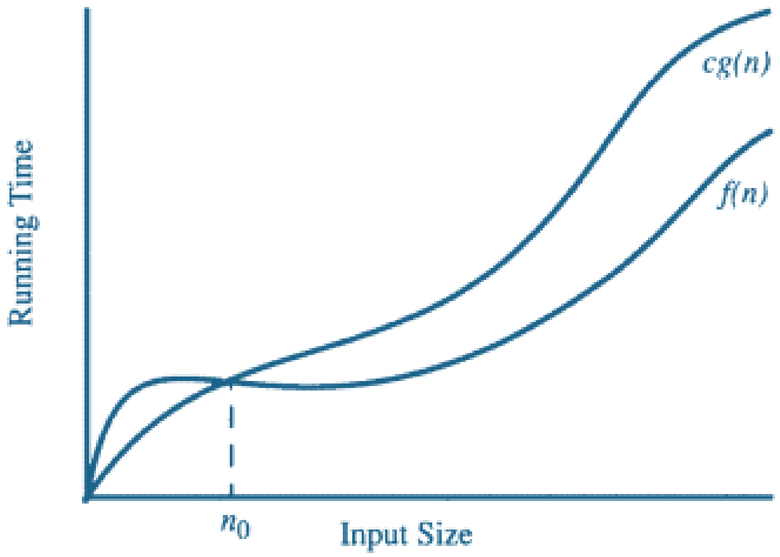
Simplified analysis of an algorithm's efficiency.

- The letter O is used because the rate of growth of a function is also called its order
- Used in complexity theory, computer science and mathematics to describe the behavior of functions.
- It determines how fast a function grows or declines.

For worst cases

Big Oh Notation

- Let f (n) and g(n) be functions mapping non-negative numbers to non-negative numbers.
- Big-Oh. f (n) is O(g(n)) if there is a constant $c > 0$ and a constant $n_0 \geq 1$ such that f (n) $\leq c \cdot g(n)$ for every number $n \geq n_0$.



Big Oh

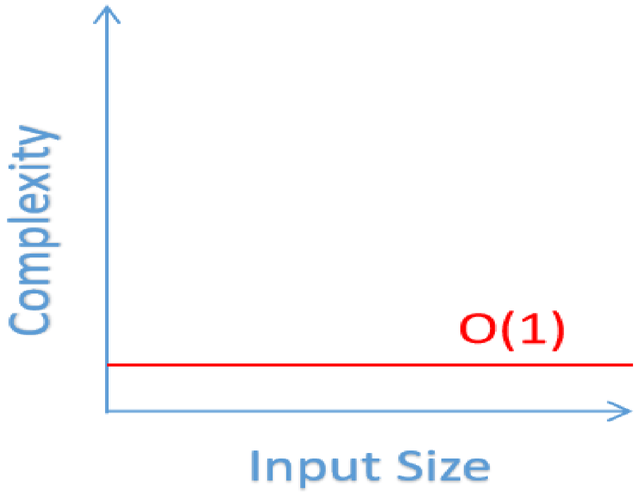
BETTER

• $O(1)$	constant time
• $O(\log n)$	log time
• $O(n)$	linear time
• $O(n \log n)$	log linear time
• $O(n^2)$	quadratic time
• $O(n^3)$	cubic time
• $O(2^n)$	exponential time

WORSE

Constant Time: O(1)

Run in constant time if it requires the same amount of time regardless of the input size.



Daily life Example???

Example: accessing any element in array

Linear Time: O(n)

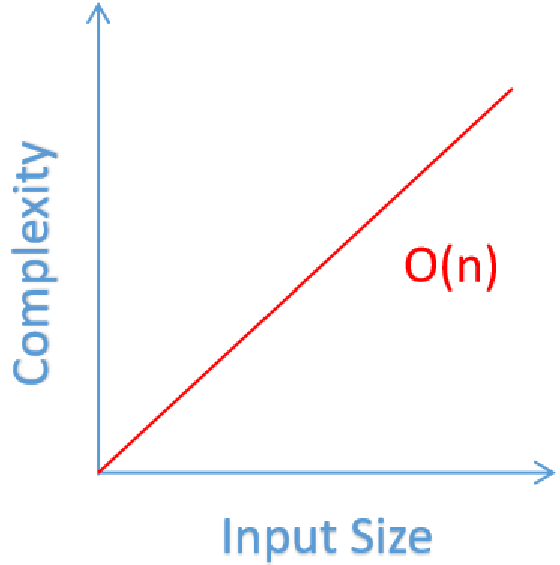
n= number of items

What will be the worst case???

best

average

Worst case
need ‘n’ steps
for ‘n’ items



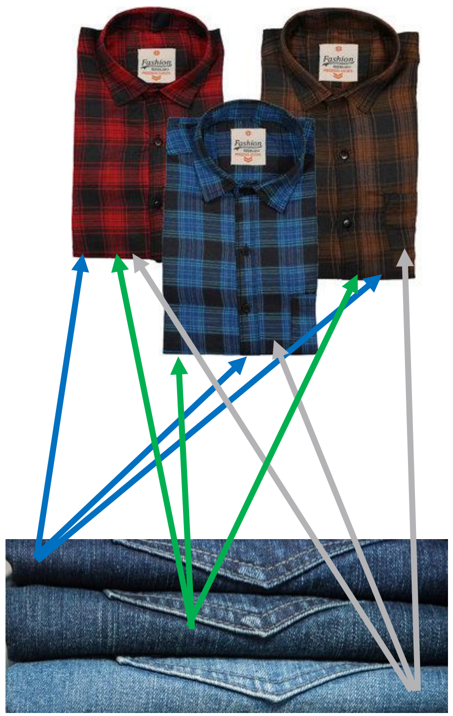
Example: traversing an array

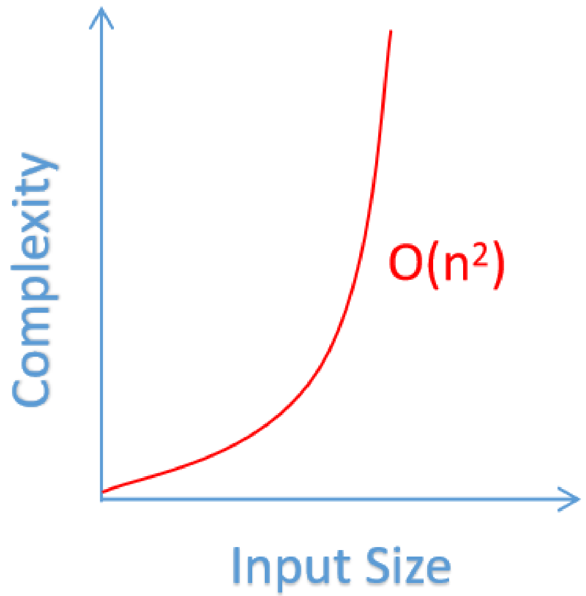
Quadratic Time: $O(n^2)$

- n= number of items

What will be the worst case???

Worst case
need 'n*n' steps for desired output

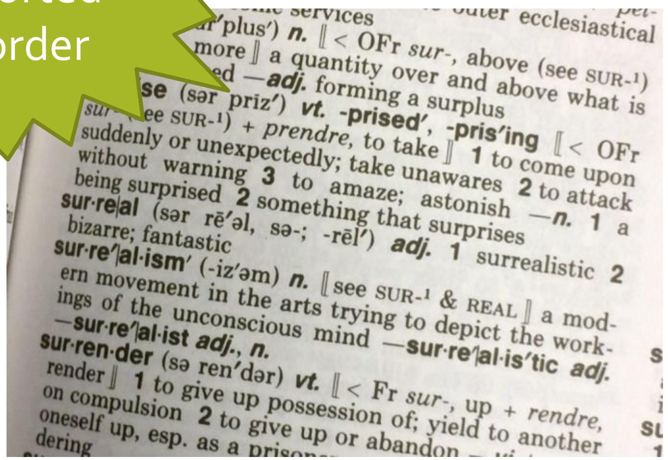




Example: bubble sort, selection sort, insertion sort

Logarithmic Time: O(log n)

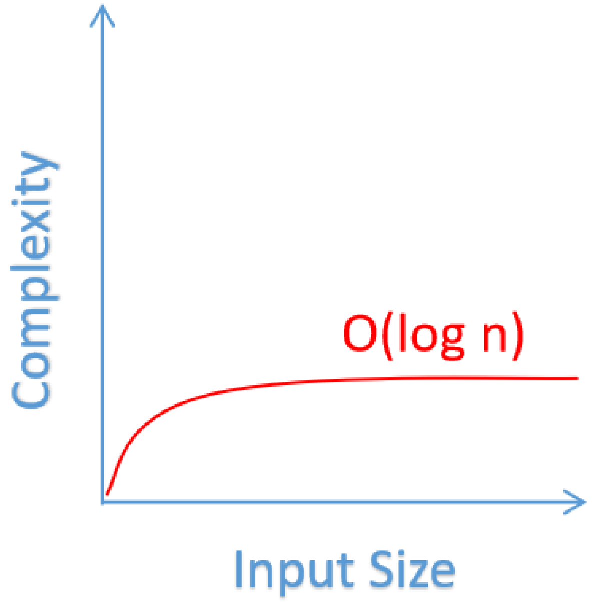
Sorted order



log 10 = ?

Log 20 =?

Log 100 = ?



Example: binary search

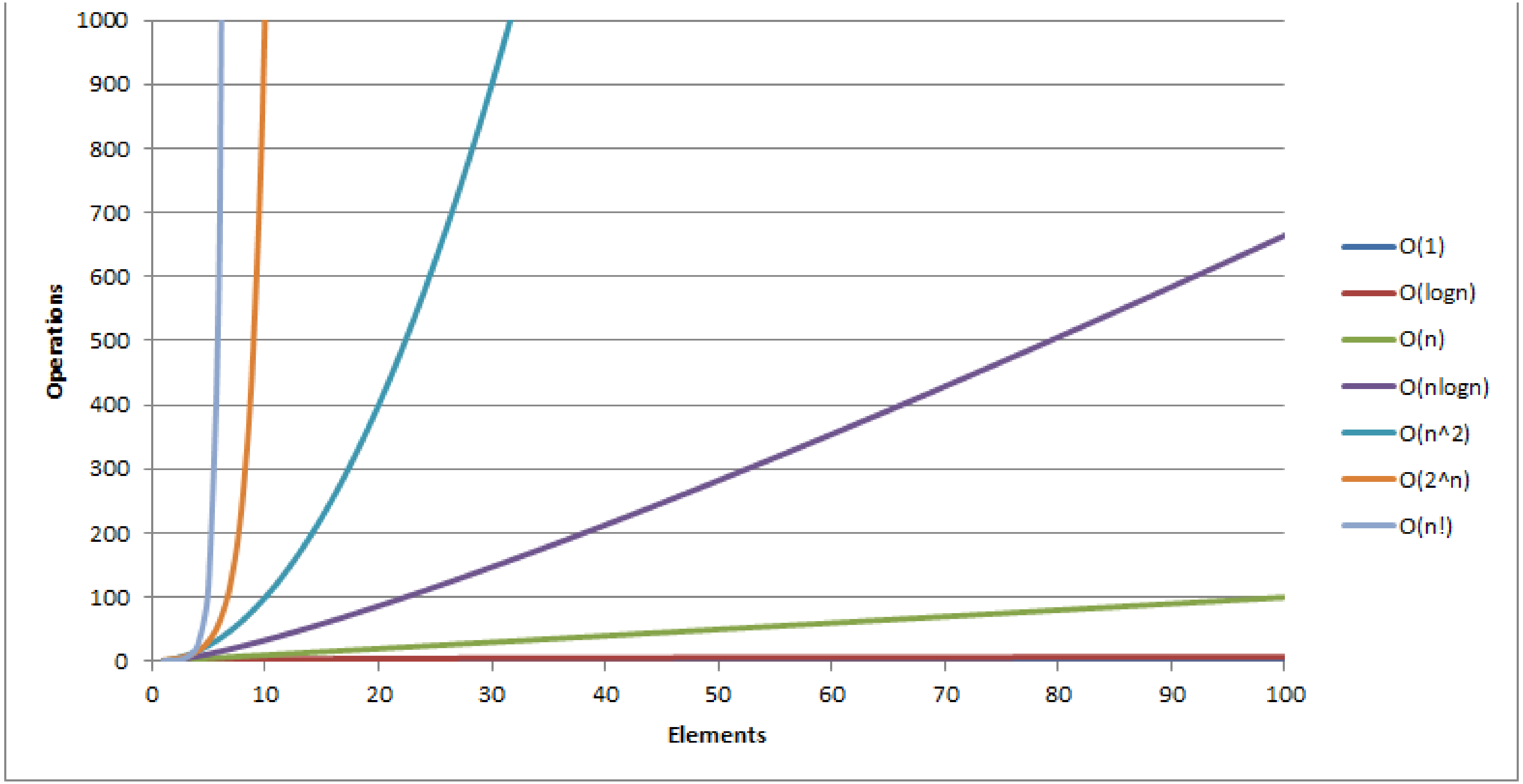
O(nlog n)

- Growth rate is faster as compared to linear and log functions

Consider buying pair for your shirts from store but not going through every item

Example: merge sort

Complexity graph



Rules for analysis

Ignore multiplicative constants

$6n \rightarrow O(N)$

'n' gets larger & constant no longer matter

Certain terms dominate others

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$

Ignore lower order terms

Rules for analysis

Loops	Number of iterations
Nested loops	Complexity of inner loop * outer loop
Consecutive statements	Addition
If/else	Block which take long time
Switch case	Block which take long time

Example

x= 5 + (15 * 30); // O(1)

(independent of input size)

x= 5 + (15 * 30); // O(1)

y= 6-4; // O(1)

print x+y; // O(1)

Total Time = O(1) + O(1) + O(1) = > O(1)

(drop constant)

Example

```
for (int i = 0; i < n; i ++)
        sum = sum – i;
```

O(**n**)

```
for (int i = 0; i < n * n ; i ++)
        sum = sum + i;
```

O($\mathbf{n}^2$)

```
sum = 0
for (int i = 0; i < n; i ++)
        for (int j = 0; j < n; j ++)
        sum + =  i * j;
```

O($\mathbf{n}^2$)

Practice

```
int sum( int n )
{
        int partialSum;

        partialSum = 0;
        for( int i = 1; i <= n; ++i )
                partialSum += i * i * i;
        return partialSum;
}
```

Complexity??

Practice

```
Fiblist(n)
create an array F [0 . . . n]
F [0] ← 0
F [1] ← 1
for i from 2 to n:
    F [i ] ← F [i – 1] + F [i – 2]
return F [n]
```

Complexity??