

Lecture 6A

# Complexity Analysis

CMPE223 / ISYE223  
ALGORITHMS AND PROGRAMMING

2024-2025 Spring

# Complexity Analysis Of Algorithms

- **Complexity analysis** is a **tool** that shows how an algorithm behaves as the input grows larger.

**How much time(\*) does it take to run an algorithm?**

- It depends on several factors:
  - How fast is the computer?
  - Are there any other programs running simultaneously when you run it?
  - Which programming language is used?
  - The instruction set of the CPU.

(\*) There are tools to measure how fast a program runs. These tools are called **profilers**, which measure the **runtime** (the time it takes to execute a program on a computer) in milliseconds and can help us optimise code by spotting bottlenecks.

# Complexity Analysis Of Algorithms (cont'd)

How much ~~time~~ does it take to run an algorithm?

How does the **runtime** of the algorithm **grow** as the size of the **input grows**?

This allows us to predict how the algorithm will behave as the input data becomes larger.

# Algorithm Complexity

- Algorithm complexity **analyses an algorithm at the idea level** (the idea of how something is computed) ignoring details such as the programming language used, the hardware the algorithm runs on, or the instruction set of the given CPU.
- The complexity analysis of algorithms can be studied in two aspects:
  - **Time Complexity**: The processing time of the algorithm. Time complexity is a way of showing how the runtime of a function increases as the size of input increases.
  - **Space Complexity**: Memory space required to run the algorithm.

# Counting Instructions

- When conducting the analysis, concentrate on the **basic computation instructions** and not things such as networking tasks or user input and output instructions as they are depending external factors.
- Also assume that the following computation operations are executed as **one instruction** each:
  - Assigning a value to a variable
  - Looking up the value of a particular element in an array
  - Comparing two values
  - Incrementing or decrementing a value
  - Basic arithmetic operations such as addition, multiplication, etc.

# Counting Instructions (cont'd)

Consider the following code fragment which looks up for the maximum element in an array. Given an input array  $A$  of size  $n$  (assume that  $n$  is always at least 1):

```
int M = A[0];  
for( int i = 0; i < n; ++i )  
{  
    if( A[i] >= M )  
    {  
        M = A[i];  
    }  
}
```

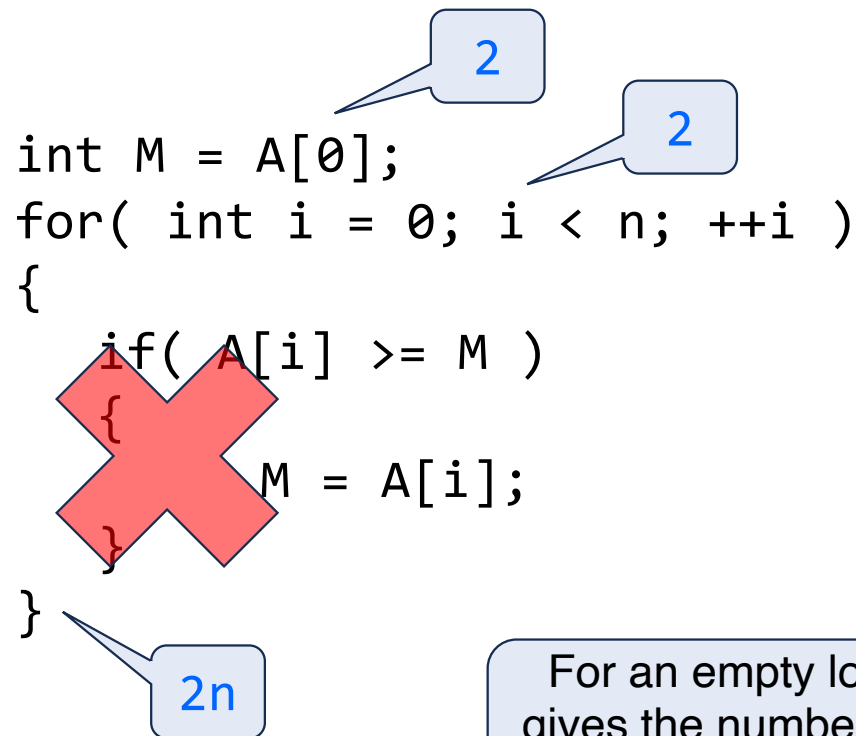
2 instructions: One for looking up value at  $A[0]$  and another for assigning that value to  $M$ .

The `for` loop initialization part also has to always execute once. This gives 2 more instructions. An assignment ( $i = 0$ ) and a comparison ( $i < n$ )

At the end of each iteration, 2 more instructions are executed. An increment of  $i$  ( $++i$ ) and a comparison ( $i < n$ ) to check if the iterations will continue ( $2n$ )

# Counting Instructions (cont'd)

```
int M = A[0];  
for( int i = 0; i < n; ++i )  
{  
    if( A[i] >= M )  
    {  
        M = A[i];  
    }  
}
```



Ignoring the loop body for now, the number of instructions executed is  $2 + 2 + 2n$  (that is 4 instructions at the beginning of the **for** loop and 2 instructions at the end of each iteration).

We can now define a mathematical function  $f(n)$  that, given an  $n$ , gives us the number of instructions the algorithm needs.

$$f(n) = 2n + 4$$

For an empty loop body  $f(n)$  gives the number of instructions the algorithm needs

# Types of Measurement

Worst-case

Best-case

Average-case



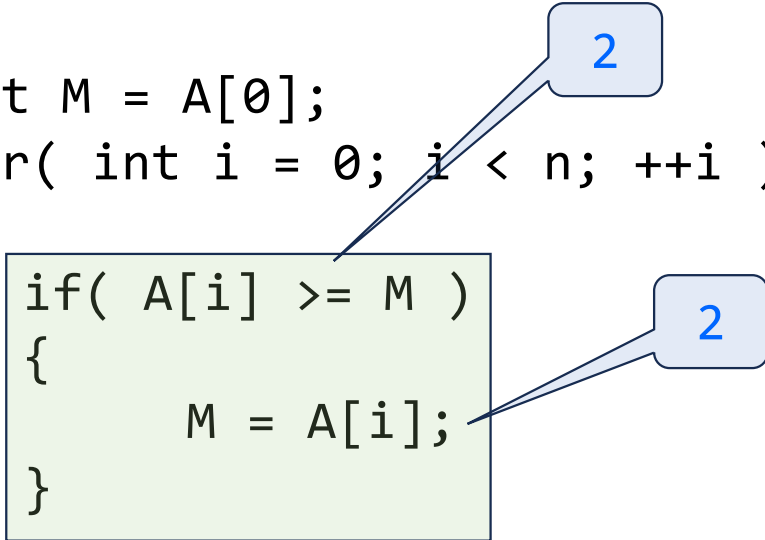
# Worst-Case Analysis

- When analysing algorithms, we need to consider the **worst-case** scenario.
- What's the worst that can happen when running an algorithm?
  - The case that the algorithm executes **the most number of instructions** to complete the task.
- In complexity analysis, we are interested in how the algorithm behaves in the worst case.

# Worst-Case Analysis (cont'd)

Continuing with the same example;

```
int M = A[0];  
for( int i = 0; i < n; ++i )  
{  
    if( A[i] >= M )  
    {  
        M = A[i];  
    }  
}
```



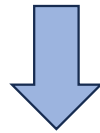
- Looking into the **for** loop body block, there is an array lookup operation and a comparison that always executes(2 instructions).
- But the code inside **if** body block may or may not be executed, depending on what the array values are.
- Considering the worst-case scenario, **assume** that the condition (**A[i] >= M**) is **always** true, hence 2 additional instructions (an array lookup and an assignment) will always be executed.

# Worst-Case Analysis (cont'd)

- So we have :

$$f(n) = 2n + 4$$

$$f(n) = 2n + 4 + 4n$$



$$f(n) = 6n + 4$$

In the worst case, we have 4 more instructions to execute within the **for** loop body (hence 4n)

The number of instructions that would be needed in the **worst case**.

# Asymptotic Behavior

- In complexity analysis we are interested in how the algorithm behaves in the worst-case.
- We want to know what happens to the instruction-counting function as the program input ( $n$ ) grows very large (goes towards infinity).
  - If an algorithm is fast for a large input, it's true that it remains faster when given an easier, smaller input.

# Asymptotic Behavior (cont'd)

- To find the asymptotic behaviour **drop** the terms from the instruction-counting function  $f(n)$  that **grow slower** and **keep the ones that grow faster** as  $n$  becomes larger.
- The act of "*keeping the largest growing term*" will give us the **asymptotic behavior** of the algorithm.

- Continuing with the example:

$$f(n) = 6n + 4$$

$$f(n) = 6n$$

$$f(n) = n$$

Drop 4 as it remains constant as  $n$  grows larger

Drop the constant multiplier 6.

- Hence the asymptotic behavior of  $f(n) = 6n + 4$  is  $n$  (linear).

# Examples

$$f(n) = n + 12$$

Drop **12** as it remains constant as **n** grows larger

$$f(n) = n$$

Rewrite as **109\*1** and drop the multiplier **109**

$$f(n) = 109$$

$$f(n) = 1$$

Drop **112** as it remains constant as **n** grows larger

$$f(n) = n^2 + 3n + 112$$

$$f(n) = n^2 + 3n$$

$$f(n) = n^2$$

**n<sup>2</sup>** grows faster than **3n**, so drop **3n**

## Examples (cont'd)

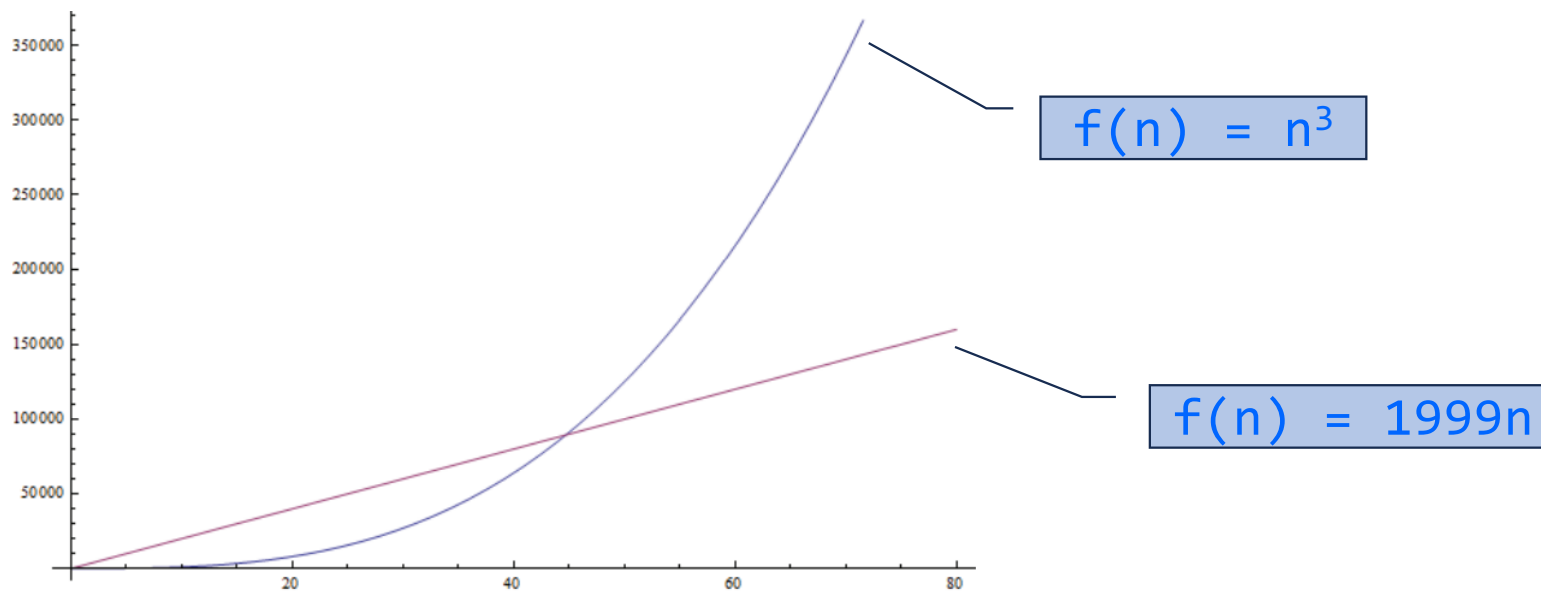
$$f(n) = n^3 + 1999n + 1337$$

$$f(n) = n^3 + 1999n$$

$$f(n) = n^3$$

Drop 1337 as it remains constant as  $n$  grows larger

Even though the factor in front of  $n$  is quite large(1999),  $n^3$  grows faster than  $1999n$ .



# Exercises

Find the asymptotic behavior of the following functions by keeping the terms that grow the fastest and dropping the constant factors.

$$1. f(n) = 2n^6 + \cancel{3n} \Rightarrow f(n) = \cancel{2}n^6 \Rightarrow f(n) = n^6$$

$$2. f(n) = 4n^2 + \cancel{12} \Rightarrow f(n) = \cancel{4}n^2 \Rightarrow f(n) = n^2$$

$$3. f(n) = 3^n + \cancel{2^n} \Rightarrow f(n) = 3^n$$

$$4. f(n) = n^n + \cancel{n} \Rightarrow f(n) = n^n$$



# Short-cut Methods for Instruction Counting

- A **single loop** over  $n$  items is  $f(n) = n$
- Two **loops nested** is  $f(n) = n^2$
- Three **loops nested** is  $f(n) = n^3$
- Given a **series** of loops that are **sequential**, the **slowest** of them determines the asymptotic behaviour of the code.
- Two nested loops followed by a single loop are **asymptotically the same as the nested loops** alone because the nested loops dominate the single loop.

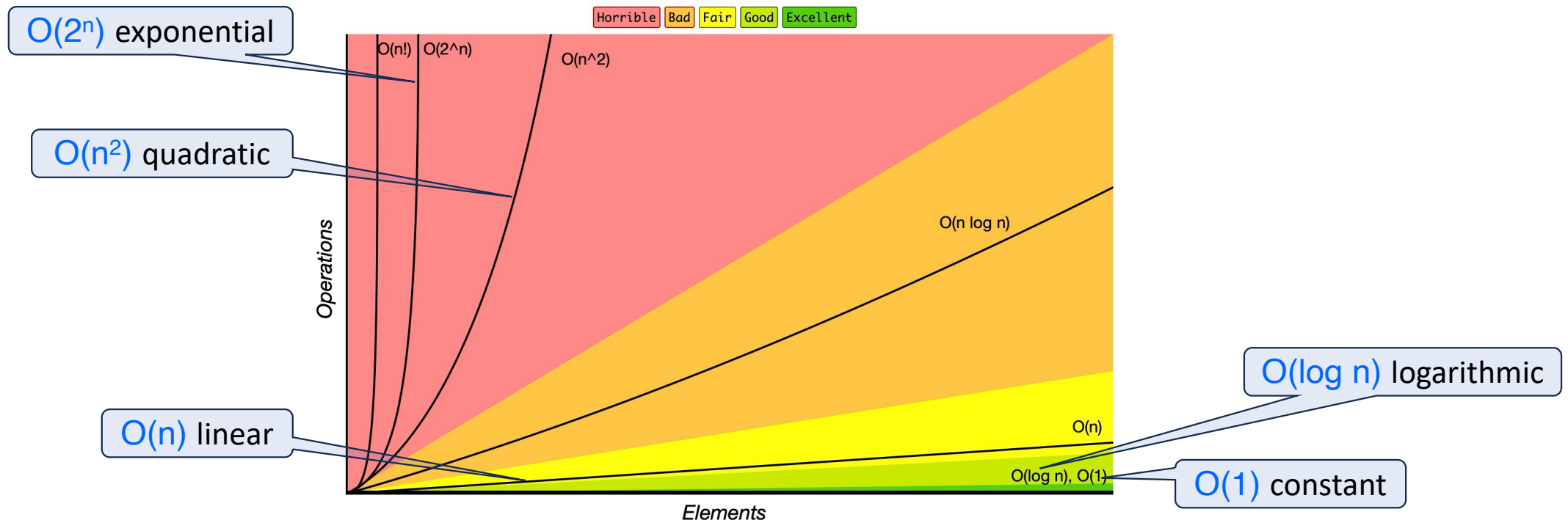
# Big-O Notation

- For more complicated/bigger algorithms it will take a lot of effort and time to figure out the exact instruction count.
- It is easier to say that the asymptotic behaviour of the algorithm will never exceed a certain bound (upper bound).
- $O()$  (pronounced as: “big oh of ...”) is used to denote an upper bound for the complexity of an algorithm.

# Big-O Notation

Algorithms with a **bigger**  $O$  run **slower** than algorithms with a smaller  $O$ .

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



<https://www.bigocheatsheet.com>

# Time Complexity of Bubble Sort

```
int E = listSize - 1;    // E indicates the number of comparisons
while( flag==1 )
{
    flag = 0;
    for( int j=0; j<E; j++ )
    {
        if( list[j] > list[j+1] ) // A swap must be made
        {
            int temp = list[j];
            list[j] = list[j+1];
            list[j+1] = temp;
            flag = 1; // Set flag as 1 to indicate a swap was performed
        }
    }
    E--; // Reduce the number of comparisons by one for the next pass
}
```

The inner loop initially repeats **E** times during the first iteration of the outer loop, then **E-1** times, then **E-2** times and so forth, until the last iteration of the outer loop during which it only runs once (the inner loop always gets one less iteration for every iteration of outer loop.).

The worst-case time complexity of Bubble sort is  **$O(n^2)$** , where "n" is the number of elements in the list or array being sorted.

# Time Complexity of Linear Search

```
int Array[] = { 8, 42, 3, 7, 9, 11, 2, 18, 33, 6, 41, 29, 4, 50 };
int NElements = sizeof(Array)/sizeof(Array[0]);
int SearchValue = 11;
int index = 0;
```

Iterates through each element in the array one by one until it finds the target element or reaches the end of the list.

```
while( (SearchValue != Array[index]) && index <= NElements )
    index++;
```

```
if (index > NElements)
    cout << "Element Not Found\n";
else
    cout << "Value " << Array[index] << " found at index " << index;
```

# Time Complexity of Linear Search (cont'd)

- In the worst-case scenario, the target element is either the last element in the list, requiring a complete traversal, or it is not present at all.
  - In both cases, the algorithm needs to check every element in the list, resulting in a time complexity proportional to the size of the input data.
- Hence the worst-case time complexity of linear search is  $O(n)$ , where "n" is the number of elements in the list or array being searched.

# References & Links

- Introduction to Algorithms, 4th Edition, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. The MIT Press.
- A Gentle Introduction to Algorithm Complexity Analysis  
<https://discrete.gr/complexity/>
- Big-O Complexity Chart (<https://www.bigocheatsheet.com>) by Eric Rowell