Determining big-O complexity:

1. Counting *primitive operations*

2. Expressing them as a function of the *problem size* → $T(n)$

3. Finding the dominant part of that function that represents its growth rate or "big-Oh" complexity

Determining big-O complexity:

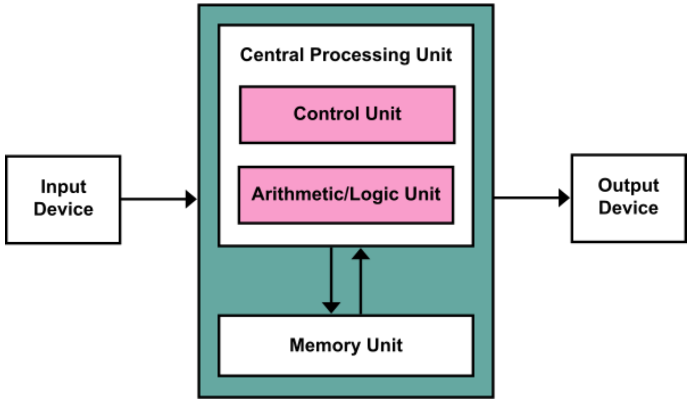
1. Counting *primitive operations*

2. Expressing them as a function of the *problem size* → $T(n)$

3. Finding the dominant part of that function that represents its growth rate or “big-Oh” complexity

First: A Model of Implementation Hardware

- We assume a generic processor, where instructions are executed one after another, *with no concurrent operations*.
    - This is called the **RAM model**
- In real-life situations involving processors with parallel processing capabilities (which is most processors now), concurrency considerations need to be taken into account
    - We aren't going there in this course, but go here for a quick *deep dive*



Identifying Primitive operations (“steps”) in *Algorithmic Analysis*

- Basic computations performed by an algorithm
- Low-level instructions commonly found in real computers: arithmetic (+, -, /, x), data movement, control.
- Assumed to take a common amount of time
- Identifiable in pseudocode
- Largely independent from the programming language
- We ignore memory hierarchies, cache hits/misses etc
- *Exact* definition not important (We will see why later)

- Examples
  - Fundamental arithmetic operations (ie: addition, multiplication, …)
  - Value assignment to a variable
  - Array indexing
  - Function call
    - (only the act of *calling* a function is “primitive”; executing that called function can span many primitive operations, and will need to be analysed in its own right)
  - Returning from a method

Which of these is a primitive operation?

- `val = a+b`
- `array2 = sort_ascending(array1)`

Which of these is a primitive operation?

- `val = a+b` YES
- `array2 = sort_ascending(array1)` NO!

Counting primitive operations

- By inspecting the pseudocode, we can determine:
  - the worst-case (ie, maximum) number of primitive operations executed by an algorithm,
  - as a function of the input size

$$InputSize \underset{f}{\rightarrow} MaxOps$$

or

$$n \underset{f}{\rightarrow} T(n)$$

Counting Primitive Operations Example: “ArrayMax” Problem

- Given an Array A of numerical values, find its maximum element:

```
ARRAY-MAX(A)
  max := A[0]
  for i = 1 to n-1
    if A[i] > max then
      max := A[i]
  {increment counter i}
  return max
```

Counting primitive operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size
- Example: find the maximum value in an array A of integers (with indices between 0 and n–1)

```
ARRAY-MAX(A)
  max := A[0]
  for i = 1 to n-1
    if A[i] > max then
      max := A[i]
  {increment counter i}
  return max
```

Operations	
2	assignment and array indexing

Counting primitive operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size
- Example: find the maximum value in an array A of integers (with indices between 0 and n–1)

```
ARRAY-MAX(A)
  max := A[0]
  for i = 1 to n-1
    if A[i] > max then
      max := A[i]
  {increment counter i}
  return max
```

Operations

2

1+n   assignment and test

In general, the loop header is executed one time more than the loop body

Counting primitive operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size
- Example: find the maximum value in an array A of integers (with indices between 0 and n–1)

```
ARRAY-MAX(A)
  max := A[0]
  for i = 1 to n-1
    if A[i] > max then
      max := A[i]
  {increment counter i}
  return max
```

Operations

2

$1+n$

$2(n-1)$ array indexing and test

Counting primitive operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size
- Example: find the maximum value in an array A of integers (with indices between 0 and n–1)

```
ARRAY-MAX(A)
  max := A[0]
  for i = 1 to n-1
    if A[i] > max then
      max := A[i]
  {increment counter i}
  return max
```

Operations

2

1+n

2(n–1)

2(n–1) array indexing and assignment

Note: Worst case analysis – we assume *max* is updated at every iteration

Counting primitive operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size
- Example: find the maximum value in an array A of integers (with indices between 0 and n–1)

```
ARRAY-MAX(A)
  max := A[0]
  for i = 1 to n-1
    if A[i] > max then
      max := A[i]
  {increment counter i}
  return max
```

Operations

2
1+n
2(n–1)
2(n–1)
2(n–1) assignment and addition

Counting primitive operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size
- Example: find the maximum value in an array A of integers (with indices between 0 and n−1)

```
ARRAY-MAX(A)
  max := A[0]
  for i = 1 to n-1
    if A[i] > max then
      max := A[i]
  {increment counter i}
  return max
```

Operations	
2	
1+n	
2(n−1)	
2(n−1)	
2(n−1)	
1	return

Counting primitive operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size
- Example: find the maximum value in an array A of integers (with indices between 0 and n–1)

ARRAY-MAX(A)	Operations
`max := A[0]`	2
`for i = 1 to n-1`	1+n
`if A[i] > max then`	2(n–1)
`max := A[i]`	2(n–1)
`{increment counter i}`	2(n–1)
`return max`	1
Total	7n – 2

Determining big-O complexity:

1. Counting *primitive operations*

2. Expressing them as a function of the *problem size* → $T(n)$

3. Finding the dominant part of that function that represents its growth rate or “big-Oh” complexity

Estimating *running time* from *counted operations*

Algorithm ARRAY-MAX executes 7n – 2 primitive operations in the *worst case*

- Worst case being: input array is in ascending order with maximum value at last position; hence `max` updated on every step

As discussed earlier, we *assume all steps take the same amount of time**, so the time taken is is directly proportional to the number of steps

**Armed with your know-how of Assembly language, you know better! E.g. a = b+c and a = (b+c)/(d+e), both may look like a single step in a high-level language, but the latter will translate to more machine level instructions or "steps". For such analysis as we are doing now though, these variations can be ignored.*

Estimating running time from counted operations

Algorithm ARRAY-MAX executes 7n – 2 primitive operations in the *worst case*

- Worst case being: input array is in ascending order with maximum value at last position; hence `max` updated on every step

As discussed earlier, we *assume all steps take the same amount of time**, so the time taken is is directly proportional to the number of steps

So: $$T(n) = 7n - 2$$

**Armed with your know-how of Assembly language, you know better! E.g. a = b+c and a = (b+c)/(d+e), both may look like a single step in a high-level language, but the latter will translate to more machine level instructions or "steps". For such analysis as we are doing now though, these variations can be ignored.*

It gets simpler!

The actual asymptotic analysis can be done way more simply than what we just did.

We don't need an *exact* expression for number steps.

We only need an estimate of how quickly the function grows as the problem size increases.

That is, we can ignore lower-order terms

Or, in other words, we are only interested in the big-Oh complexity of $T(n)$

Determining big-O complexity:

1. Counting *primitive operations*

2. Expressing them as a function of the *problem size* → $T(n)$

3. Finding the dominant part of that function that represents its growth rate or “big-Oh” complexity

Thinking about the **growth rate** of running time

- Expressions like 7n-2, or $6x^2+2x+124$, are "*too precise*" for the purposes of asymptotic analysis
  - Context: Analyse the complexity of an algorithm by estimating how the number of steps grow, in the limit as the size of the problem grows → ∞

- For an algorithm like ARRAY−MAX:
  - It is the *linear growth rate of the running time T(n)* the intrinsic property of the algorithm that we wish to focus on

- For example, the following running times are all (asymptotically) linear:

$$7n - 2 = O(n)$$
$$7239n = O(n)$$
$$\frac{9}{350}n + 100 = O(n)$$

Managing (simplifying) asymptotic analysis

- We can simplify the analysis by looking at the behaviour of different factors that we may find in a $T(n)$ expression:

- **Constant factors**

- **Linear factors**

- **Power/Exponential/Factorial factors**

The Insight

- Growth rate is **not** affected by
  - constant factors
  - lower-degree terms

- That is, at large enough n, the function $T(n)$ is largely determined by the **highest degree term** in its expression

- **So: Big-Oh notation is used to express asymptotic upper bounds**

Ignore constant factors

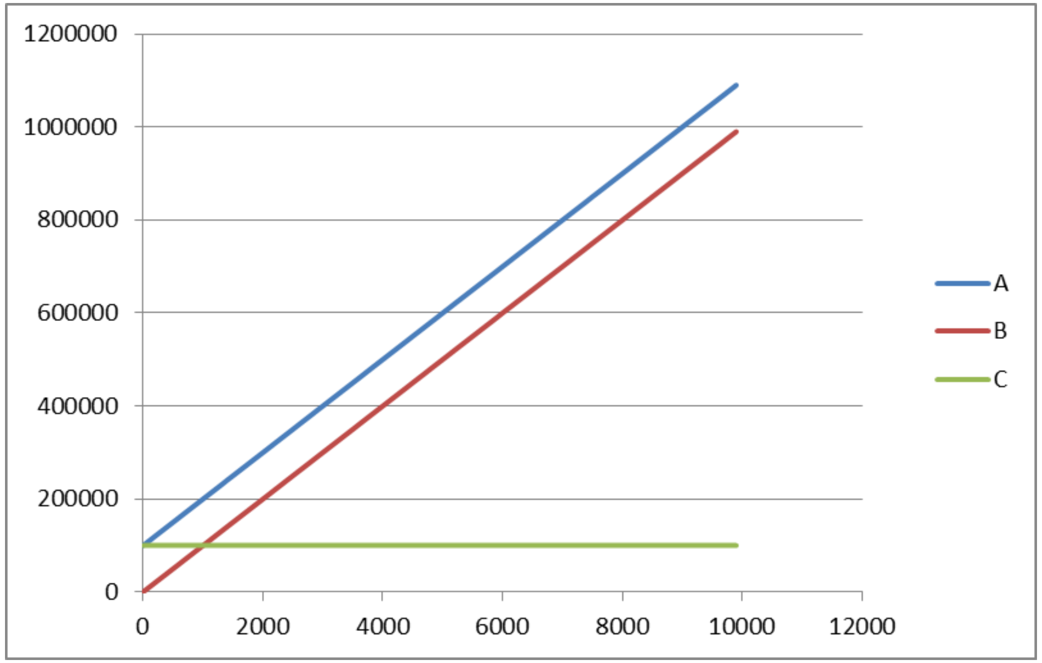
- Example

  $A = 10^2 n + 10^5$ is a linear function

  $A = B + C$, where

  $B = 10^2 n$ is a linear function

  $C = 10^5$ is a constant *(impact on growth rate can be ignored)*



Ignore lower-degree terms

- Example

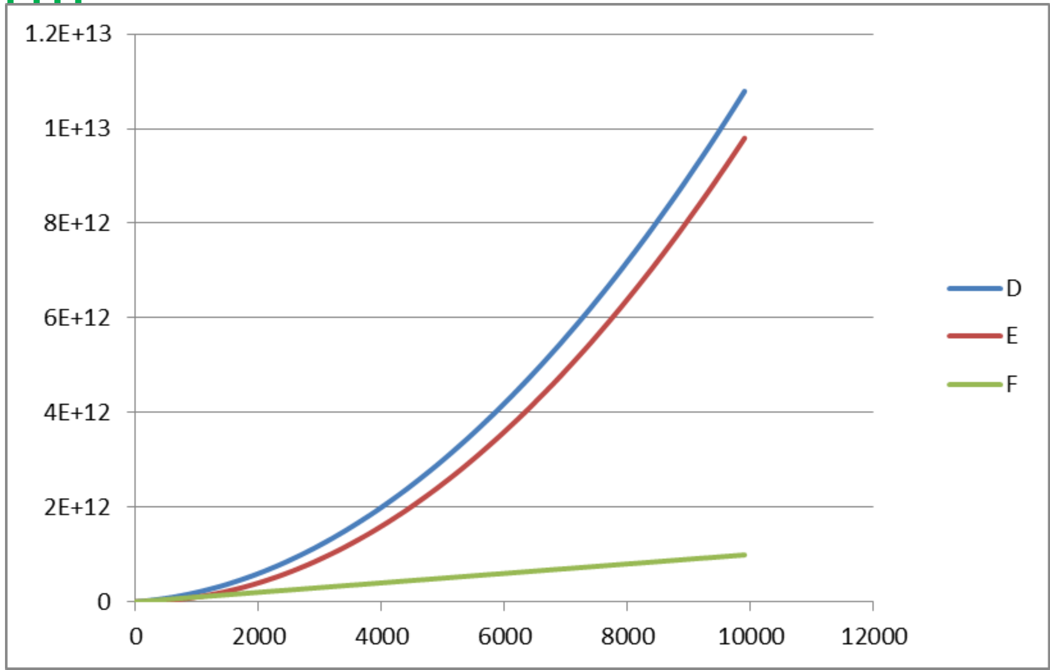
  $D = 10^5 n^2 + 10^8 n$ is a quadratic function

  $E = 10^5 n^2$ is a quadratic function

  $F = 10^8 n$ is a lower-degree (linear) term

  (impact on growth rate can be ignored)

  $D = E + F$



Key Intuition (informal)

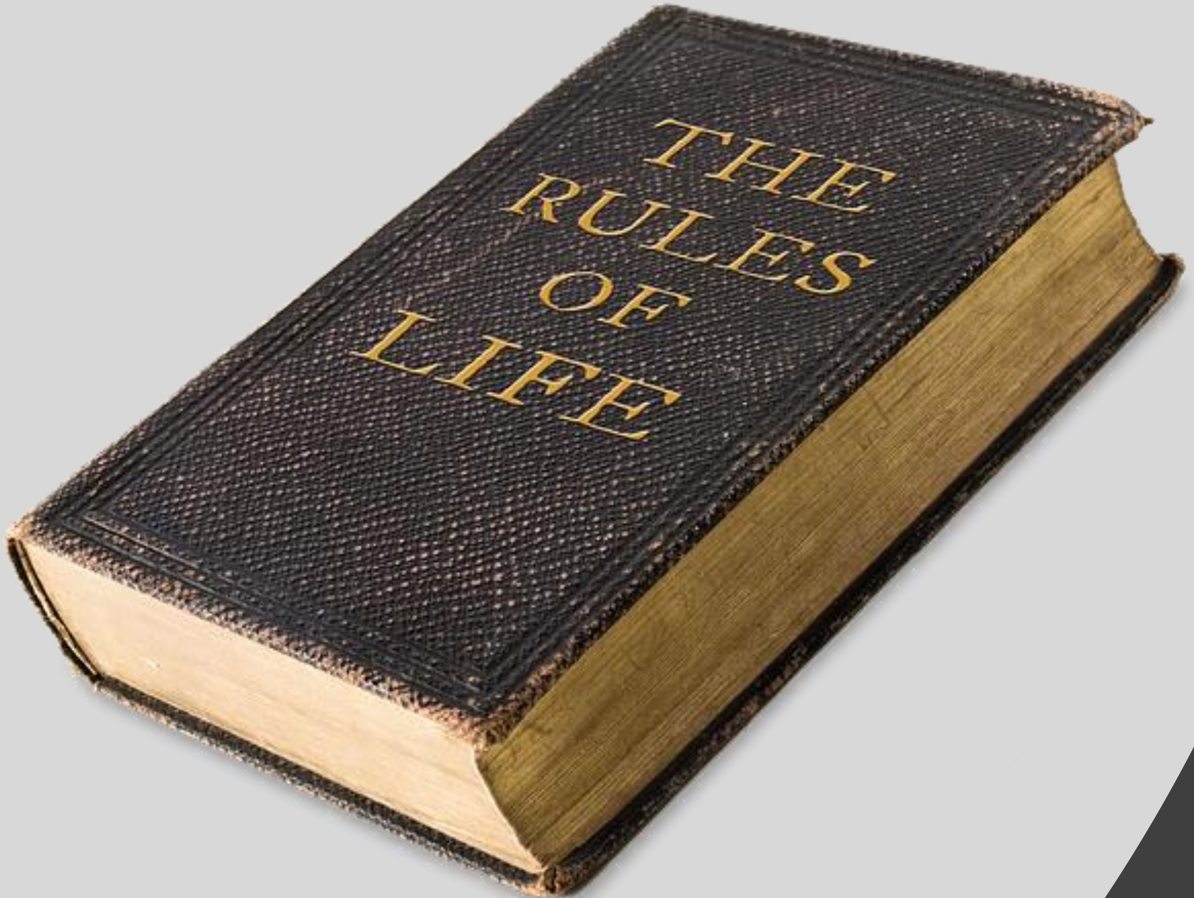
Notation

$$f(x) = O(g(x))$$

Is the ***asymptotic*** analogue of traditional ordering relation

$$f(x) \leq g(x)$$

That is, $g(x)$ provides an asymptotic upper-bound to $f(x)$



Rules for finding the Big-O complexity of algorithms

Rules to compute running times

- ## Rule 1 – Loops
  - The running time of a loop is at most the running time of the statements inside the loop (including loop bound tests) multiplied by the number of iterations

```
ALG1(n)
  for i = 0 to n-1
   increment x
```

Rules to compute running times

- Rule 1 – Loops
  - The running time of a loop is at most the running time of the statements inside the loop (including loop bound tests)) multiplied by the number of iterations

- Rule 2 – Nested loops
  - Total running time of a statement inside a group of nested loops is running time of statement multiplied by the *product* of the sizes of all the loops
  - This way we focus on the fastest changing (inner-most) loop, as that will determine the dominant term

```
ALG1(n)
  for i = 0 to n-1
   for j = 0 to n-1
     for k = 0 to n-1
       increment x
```

Rules to compute running times

- Rule 3 – Consecutive statements
  - Just add

```
ALG1(n)
  for i = 0 to n-1
     read x from array
     increment x
     store x back in array
```

Rules to compute running times

- **Rule 3 – Consecutive statements**
  - Just add
- **Rule 4 – If-then-else**
  - Running time is never more than the time of the *test* (condition) plus the *worst* (ie, maximum) of the running times of the two branches
  - Similarly for multiple/nested else statements

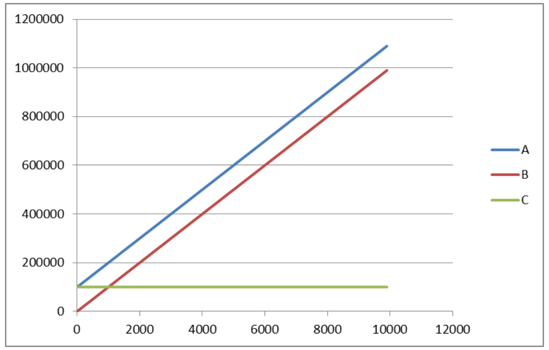
```
ALG1(n)
   if (condition true)
      //O(1)
   else
      //O(n)
```

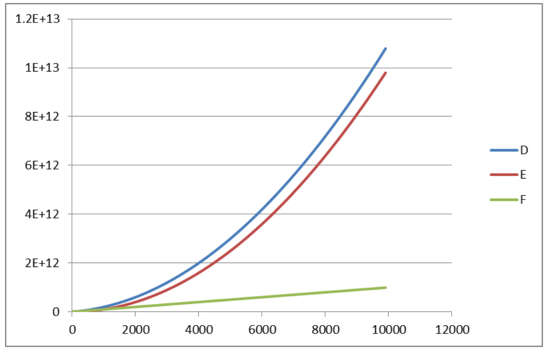
And the following is the best part… we can ignore a whole lot of “clutter” and focus solely on the dominant term

Rules to compute running times

- Rule 5 – Remove constant multipliers
  - E.g. $O(2n) \rightarrow O(n)$

- Rule 6 – Drop “non-dominants” [lower-order terms]
  - $O(n^2) + O(n) + O(1) \rightarrow O(n^2)$





- Rule 7 – Assume the WORST

Let’s have another look at the ARRAY-MAX example using a simpler approach

- Knowing that we will simplify the final expression to focus on the asymptotically dominant expression anyway, we can “look ahead” and make the process of “counting” a lot simpler:

```
ARRAY-MAX(A)
  max := A[0]
  for i = 1 to n-1
    if A[i] > max then
      max := A[i]
  {increment counter i}
  return max
```

SOME ANALYSIS EXAMPLES

Loop – Summing up squares

```
SQUARES1(n)
  i := 0
  sum := 0
  while i < n
    increment i
    sum := sum + (i * i)
  return sum
```

Is there a more efficient implementation ?

The power of maths

- Input: positive integer **n**
- Output: the sum of the first **n** squares

Summation rule

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

```
SQUARES2(n)
  sum := n * (n+1) * (2*n+1)/6
  return sum
```

Operations

O(1)

O(1)

- T(n) = O(1) + O(1) = O(1)
  - No loops!

Hmmm...

- Input: positive integer **n**
- Output: the integer part of the square root of **n**
- How many times will *this* loop run?

```
INT-SQRT1(n)
  i := 1
  while (i*i ≤ n)
    increment i
  return i-1
```

Varying loop limits

```
sum := 0
  for i = 1 to n
    for j = 1 to i
      sum := sum + 1
```

Varying loop limits

```
sum := 0
  for i = 1 to n
    for j = 1 to i
      sum := sum + 1
```

When the range of an inner loop is *not* constant, but depends (grows/shrinks) on a variable of an outer loop:

- For our purposes of Big-O bounds: sufficient to work with the <u>largest</u> size that the range of that loop may take.

- Reason: If you do an exact calculation, then apply the simplification rules, you will be left with this answer anyway!

Recap

- The *asymptotic* analysis of an algorithm determines the running time in **big-O notation, and we focus on the *worst case***
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed, as a *function of the input size*
  - We then simply this function to get it’s big-O complexity
- Example
  - We determined algorithm ARRAY-MAX executes at most $7n - 2$ primitive operations
  - We can, knowing the rules of simplification that isolate the dominant term, simplify the process of “counting” primitive operations.
  - We say that algorithm ARRAY-MAX “runs in $O(n)$ time” or, equivalently, ”has linear running time”

- Since constant factors and lower-order terms are eventually ignored in the big-O notation, we can **disregard** them when counting primitive operations in the first place