# Analysing Algorithms

Part 2

# Roadmap

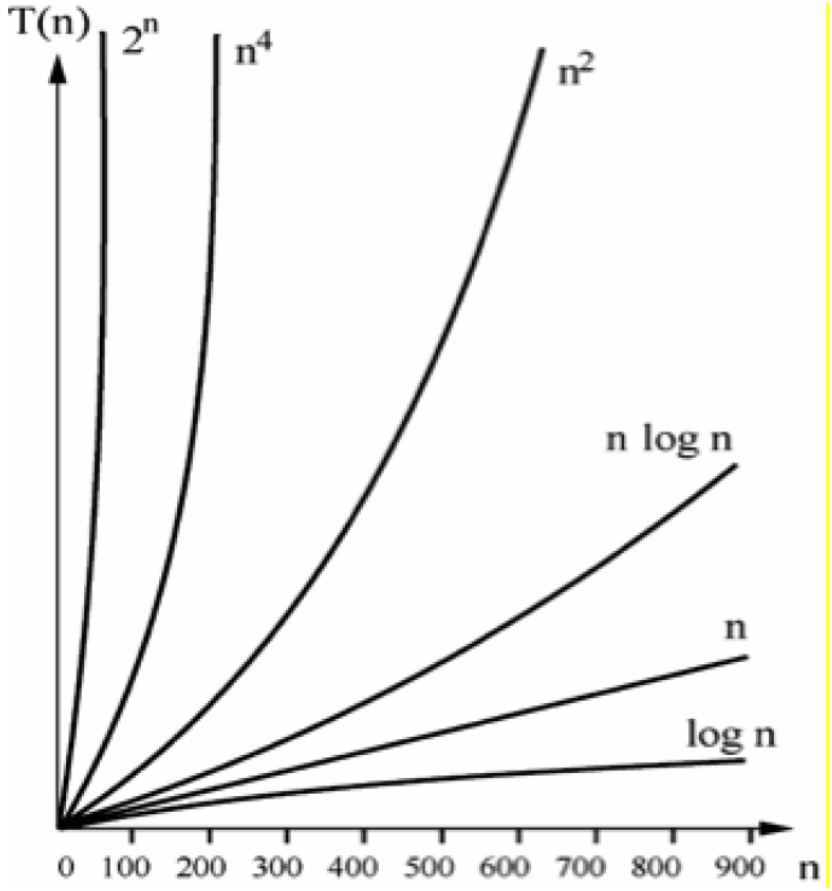
- Review of key concepts
  - Complexity
  - Orders of growth
  - Best, average & worst cases
- Asymptotic notation
  - O (Big O)
  - Ω (omega)
  - Θ (theta)
- Evaluating complexity
- Examples

# Review of complexity

- Complexity measures the efficiency of an algorithm's design, eliminating the effects of platform-specific implementation details (e.g. CPU or complier design)
- We can compare the relative efficiency of algorithms by evaluating their running time complexity on input data of size n (memory or storage requirements of an algorithm could also be evaluated in this manner)
- E.g. how much longer will an algorithm take to execute if we input a list of 1000 elements instead of 10 elements?
- Standard methodology developed over the past half-century for comparing algorithms
- Can determine which algorithms scale well to solve problems of a nontrivial size, by evaluating the complexity the algorithm in relation to the size n of the provided input
- Typically, algorithmic complexity falls into one of a number families (i.e. the growth in its execution time with respect to increasing input size n is of a certain order). The effect of higher order growth functions becomes more significant as the size n of the input set is increased

# Orders of growth

| Running time $T(n)$ is proportional to: | Complexity: |
|---|---|
| $T(n) \propto \log n$ | logarithmic |
| $T(n) \propto n$ | linear |
| $T(n) \propto n \log n$ | linearithmic |
| $T(n) \propto n^2$ | quadratic |
| $T(n) \propto n^3$ | cubic |
| $T(n) \propto n^k$ | polynomial |
| $T(n) \propto 2^n$ | exponential |
| $T(n) \propto k^n;\ k > 1$ | exponential |



# Comparing growth functions

| value of $n$ | constant | $log\ n$ (logarithmic) | $n$ (linear) | $n\ log\ n$ (linearithmic) | $n^2$ (quadratic) | $n^3$ (cubic) | $2^n$ (exponential) |
|---|---|---|---|---|---|---|---|
| 8 | 1 | 3 | 8 | 24 | 64 | 512 | 256 |
| 16 | 1 | 4 | 16 | 64 | 256 | 4096 | 65536 |
| 32 | 1 | 5 | 32 | 160 | 1024 | 32768 | 4294967296 |
| 64 | 1 | 6 | 64 | 384 | 4096 | 262144 | 1.84467E+19 |
| 128 | 1 | 7 | 128 | 896 | 16384 | 2097152 | 3.40282E+38 |
| 256 | 1 | 8 | 256 | 2048 | 65536 | 16777216 | 1.15792E+77 |
| 512 | 1 | 9 | 512 | 4608 | 262144 | 1.34E+08 | 1.3408E+154 |

# Best, worst and average cases

- As well as the size n of the input, the actual data that is input may also have an effect on the time which an algorithm takes to run
- There could be many, many instances of size n which would be valid as input; it may be possible to group these instances into classes with broadly similar features
- For many problems, no single algorithm exists which is optimal for every possible input instance
- Therefore, choosing an algorithm depends on understanding the problem being solved and the underlying probability distribution of the instances likely to be encountered, as well as the behaviour of the algorithms being considered
- By knowing the performance of an algorithm under each of these cases, you can judge whether an algorithm is appropriate to use in your specific situation

# Best, average and worst cases

- **Worst case**: Defines a class of input instances for which an algorithm exhibits its worst runtime behaviour. Instead of trying to identify the specific input, algorithm designers typically describe properties of the input that prevent an algorithm from running efficiently.
- **Average case**: Defines the expected behaviour when executing the algorithm on random input instances. While some input problems will require greater time to complete because of some special cases, the vast majority of input problems will not. This measure describes the expectation an average user of the algorithm should have.
- **Best case**: Defines a class of input instances for which an algorithm exhibits its best runtime behaviour. For these input instances, the algorithm does the least work. In reality, the best case rarely occurs.

# Worst case

- For any particular value of n, the number of operations or work done by an algorithm may vary dramatically over all the instances of size n
- For a given algorithm and a given value n, the worst-case execution time is the maximum execution time, where the maximum is taken over all instances of size n
- We are interested in the worst-case behaviour of an algorithm because it often is the easiest case to analyse
- It also explains how slow the program could be in any situation, and provides a lower bound on possible performance
- Good idea to consider worst case if guarantees are required for the maximum possible running time for a given n
- Not possible to find every worst-case input instance, but sample (near) worst-case instances can be crafted given the algorithm's description

# Big O notation

- Big O notation (with a capital latin letter O, not a zero) is a symbolism used in complexity theory, mathematics and computer science to describe the asymptotic behaviours of functions
- In short, Big O notation measures how quickly a function grows or declines
- Also called Landau's symbol, after the German number theoretician Edmund Landau who invented the notation
- The growth rate of a function is also called its *order*
- The capitalised greek letter "omicron" was originally used; this has fallen out of favour and the capitalised latin letter "O" is now commonly used
- Example use: Algoroithm X runs in $O(n^2)$ time

# Big O notation

- Big O notation is used in computer science to describe the complexity of an algorithm in the worst-case scenario
- Can be used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm
- Big O notation can be thought of as a measure of the expected “efficiency” of an algorithm (note that for small sizes of n, all algorithms are efficient, i.e. fast enough to be used for real time applications)
- When evaluating the complexity of algorithms, we can say that if their Big O notations are similar, their complexity in terms of time/space requirements is similar (in the worst case)
- And if algorithm A has a less complex Big O notation than algorithm B, we can infer that it is much more efficient in terms of space/time requirements (at least in the worst case)

# Formally

Suppose $f(x)$ and $g(x)$ are two functions defined on some subset of the set of real numbers

$$f(x) = \mathrm{O}\big(\mathrm{g}(x)\big) \text{ for } x \to \text{infinity}$$

if and only if there exist constants $N$ and $C$ such that

$$|f(x)| \leq \mathrm{C}|\mathrm{g}(x)| \text{ for all } x > \mathrm{N}$$

Intuitively, this means that $f$ does not grow more quickly than $g$

(Note that N is size of the input set which is large enough for the higher order term to begin to dominate)

# Tightest upper bound

- Note that when using Big O notation, we aim to identify the tightest upper bound possible
- An algorithm that is $O(n^2)$ is also $O(n^3)$ , but the former information is more useful
- Specifying an upper bound which is higher than necessary is like saying: “This task will take at most one week to complete”, when the true maximum time to complete the task is in fact five minutes!

# $\Omega$ (omega) notation

- We can use $\Omega$ (omega) notation to describe the complexity of an algorithm in the best case
- Best case may not occur often, but still useful to analyse
- Represents the lower bound on the number of possible operations
- E.g. an algorithm which is $\Omega(n)$ exhibits a linear growth in execution time in the best case, as n is increased

# $\Theta$ (theta) notation

- Finally, $\Theta$ (theta) notation is used to specify that the running time of an algorithm is no greater or less than a certain order
- E.g. we say that an algorithm is $\Theta(n)$ if it is both $O(n)$ and $\Omega(n)$, i.e. the growth of its execution time is no better or worse than the order specified (linear in this case)
- The actual functions which describe the upper and lower limits do not need to be the exact same in this case, just of the same order

# Separating an algorithm and its implementation

- Two concepts:
  - The input data size $n$, or the number of individual data items in a single data instance to be processed
  - The number of elementary operations $f(n)$ taken by an algorithm, or its running time
- For simplicity, we assume that all elementary operations take the same amount of "time" to execute (not true in practice due to architecture, cache vs. RAM vs. swap/disk access times etc.)
- E.g. an addition, multiplication, division, accessing an array element are all assumed to take the same amount of time
- Basis of the RAM (Random Access Machine) model of computation

# Separating an algorithm and its implementation

- The running time $\mathrm{T(n)}$ of an implementation is: $\mathrm{T(n)} = c * f(n)$
  - $f(n)$ refers to the fact that the running time is a function of the size n of the input dataset
  - $c$ is some constant
  - The constant factor $c$ can rarely be determined and depends on the specific computer, operating system, language, compiler, etc. that is used for the program implementation

# Evaluating complexity

- When evaluating the complexity of an algorithm, keep in mind that you must identify the most expensive computation within an algorithm to determine its classification
- For example, consider an algorithm that is subdivided into two tasks, a task classified as linear followed by a task classified as quadratic.
  - Say the number of operations/execution time is:
  - $\mathrm{T}(n) = 50 + 125n + 5n^2$
  - The overall complexity of the algorithm must therefore be classified as quadratic, we can disregard all lower order terms as the $n^2$ term will become dominant for input sizes of $n$=6 or above

# Evaluating complexity

- An algorithm with better asymptotic growth will eventually execute faster than one with worse asymptotic growth, regardless of the actual constants
- The actual breakpoint will differ based on the constants and size of the input, but it exists and can be empirically evaluated
- During asymptotic analysis we only need to be concerned with the fastest-growing term of the T(n) function. For this reason, if the number of operations for an algorithm can be computed as $T(n) = c * n^3 + d * n \log(n)$, we would classify this algorithm as $O(n^3)$ because that is the dominant term which grows far more rapidly than $n \log(n)$

# Passing an array to a method in Java

- If we want to pass multiple values to a method, the easiest way to do this is to pass in an array
  - E.g. an array of numbers to be sorted

```
// passing an array of integers into a method in Java
void myMethod(int[] elements) {
    // do something with the data here
}
```

Array with 5 elements

# Passing an array to a function in Python

- If we want to pass multiple values to a function, the easiest way to do this is to pass in an array
  - E.g. an array of numbers to be sorted

```
def my_function(elements):
    # do something with the data here

test1 = [5, 1, 12, -5, 16]
my_function(test1)
```

Array with 5 elements

# O(1) example

- $O(1)$ describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set
- Consider the Java code sample to the right
- No matter how many elements are in the array, this method will execute in **constant time**
- This method executes in constant time in the best, worst and average cases

```
boolean isFirstElementTwo(int[] elements)
{
    if(elements[0] == 2) {
        return true;
    }
    else {
        return false;
    }
}
```

# O(1) example

- $O(1)$ describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set
- Consider the Python code sample to the right
- No matter how many elements are in the array, this method will execute in **constant time**
- This method executes in constant time in the best, worst and average cases

```
def is_first_el_two(elements):
    if elements[0] == 2:
        return True
    return False


test1 = [2, 1, 0, 3]
test2 = [0, 2, 3, 4]

print(is_first_el_two(test1))  # prints True
print(is_first_el_two(test2))  # prints False
```

# O(n) example

- O(n) describes an algorithm whose worst case performance will grow linearly and in direct proportion to the size of the input data set
- An algorithm which loops once through an array using a for loop would typically be O(n). A matching number could be found during any iteration of the for loop and therefore the function could return before all elements in the array have been iterated through
- Big O notation will always assume the upper limit where the algorithm will perform the maximum number of operations

# O(n) example

- Consider the Java code sample to the right
- The worst possible time complexity depends linearly on the number of elements in the array
- Execution time for this method is constant in the best case

```
boolean containsOne(int[] elements) {
  for (int i=0; i<elements.size(); i++){
    if(elements[i] == 1) {
      return true;
    }
  }
  return false;
}
```

# O(n) example

- Consider the Python code sample to the right
- The worst possible time complexity depends linearly on the number of elements in the array
- Execution time for this method is constant in the best case

```
def contains_one(elements):
    for i in range(0, len(elements)):
        if elements[i] == 1:
            return True
    return False


test1 = [0, 2, 1, 2]
test2 = [0, 2, 3, 4]

print(contains_one(test1))  # prints True
print(contains_one(test2))  # prints False
```

# $O(n^2)$ example

- $O(n^2)$ represents an algorithm whose worst case performance is directly proportional to the square of the size of the input data set
- This class of complexity is common with algorithms that involve nested iterations over the input data set (e.g. nested for loops)
- Deeper nested iterations will result in higher orders e.g. $O(n^3)$, $O(n^4)$, etc.

# $O(n^2)$ example

- Consider the Java code sample to the right
- The worst execution time depends on the square of the number of elements in the array
- Execution time for this method is constant in the best case

```
boolean containsDuplicates(int[] elements)
{
  for (int i=0; i<elements.length; i++){
    for (int j=0; j<elements.length; j++){
      if(i == j){ // avoid self comparison
        continue;
      }
      if(elements[i] == elements[j]) {
        return true; // duplicate found
      }
    }
  }
  return false;
}
```

# $O(n^2)$ example

- Consider the Python code sample to the right
- The worst execution time depends on the square of the number of elements in the array
- Execution time for this method is constant in the best case

```
def contains_duplicates(elements):
    for i in range(0, len(elements)):
        for j in range(0, len(elements)):
            if i == j:  # avoid self comparison
                continue
            if elements[i] == elements[j]:
                return True  # duplicate found
    return False


test1 = [0, 2, 1, 2]
test2 = [1, 2, 3, 4]

print(contains_duplicates(test1))  # prints True
print(contains_duplicates(test2))  # prints False
```