Computational Thinking with Algorithms - Analysing Algorithms Part Deux[1]

Dominic Carr

Atlantic Technological University

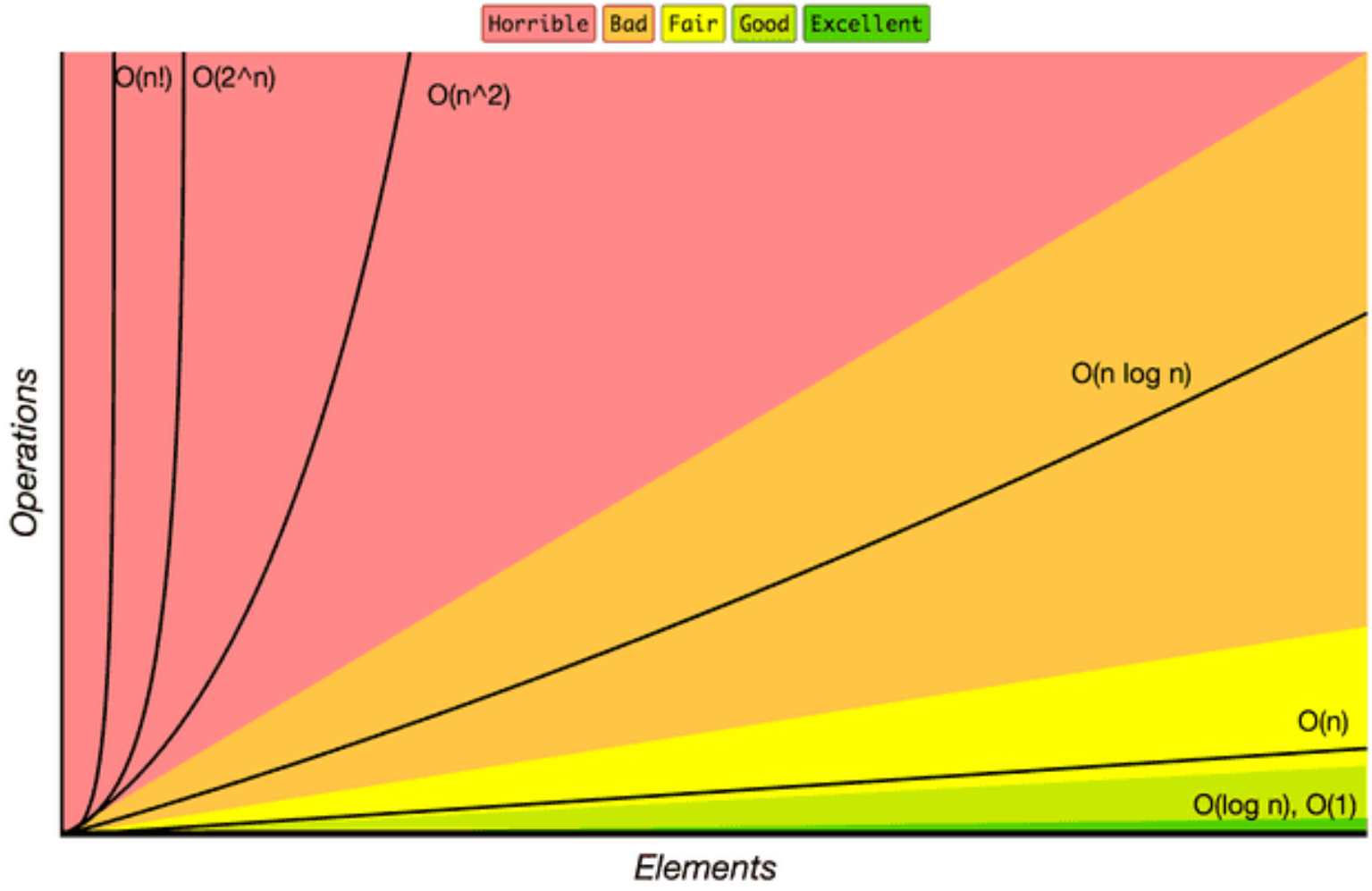
dominic.carr@atu.ie

[1] This time it's personal

Big O Notation I

- Big O notation is a symbol used in computer science to describe the asymptotic behaviours of functions
  - Basically means fits to a curve
- 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
- Example use: Algorithm X runs in $\mathcal{O}(n \log n)$ time

Big O Notation II



Big O Notation III

- We use it 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
- 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/usable.
  - Issues arise at scale
- 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)

Big O Notation IV

We try to identify the tightest upper bound

- An algorithm that is $\mathcal{O}(n^2)$is also $\mathcal{O}(n^3)$, but the former information is more useful
- Specifying an upper bound 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!

Ω Omega Notation

- We ca use Ω notation for best case complexity
- Best case is rare, still useful to know
- Represents a lower bound on the number of operations i.e. the amount of "work"
- For example: an algorithm which is $\Omega(n)$ exhibits a linear growth in execution time in the best case, as n is increased

Θ Theta Notation

- Θ 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 an algorithm is $\Theta(n)$ if it is both $\mathcal{O}(n)$ and $\Omega(n)$
- 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

“In theory, theory and practice are the same. In practice, they are not”

Separating an Algorithm and it's Implementation(s) I

- 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. Addition, multiplication, division, variable assignment, accessing an array element are all assumed to take the same amount of time
  - They are $\mathcal{O}(1)$ operations.

Separating an Algorithm and it's Implementation(s) II

- The running time $T(n)$ of an implementation is given:
  $T(n) = c * f(n)$
  - $f(n)$ refers to the fact that the running time is a function of the size of the input, n.
  - c is some constant, which can rarely be determined and depends upon the specific computer, OS, programming language, compiler, dependencies, bandwidth etc.

Evaluating Complexity I

- Remember we must identify the most expensive computation to determine the classification
- For example consider an algorithm where there are constant time, linear, and quadratic tasks.
  - $T(n) = 50 + 125n + 5n^2$
  - The overall complexity is quadratic, all lower order terms can be disregarded as n^2 is dominant where $n >= 6$

Evaluating Complexity II

- 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 example: $T(n) = c * n^3 + d * n \log n$
  - The algorithm is classified as $\mathcal{O}(n^3)$ as that is the dominant term, growing much faster than $n \log n$

Programming Examples I

Listing 1: Java

```
// pass the array as argument to the method
void myMethod(int[] elements) {
    // do something with the data here
}
```

Listing 2: Python

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

test1 = [1,2,3,5,6,100,7,-12]
my_function(test1)
```

The array is the input to the function of size n, in the Python example we can see $n = 8$

$\mathcal{O}(1)$ Example I

- $\mathcal{O}(1)$ means the algorithm will always execute in the same time (or space) regardless of the size of the input
- In the below example the code will execute in constant time regardless of the length of the array.
- Best, worst, and average time is the same, constant time.

Listing 3: Constant time example

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

$\mathcal{O}(n)$ Example I

- 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 $\mathcal{O}(n)$.
  - A matching number *could be found* during any iteration of the for loop and therefore the function could return before all elements have been iterated.
- Big O notation will always assume the upper limit where the algorithm will perform the maximum number of operations
  - Worst case!

$\mathcal{O}(n)$ Example II

- Worst case this algorithm has to loop through the entire array
- This is $\mathcal{O}(n)$
- Depends linearly on the number of elements in the array
- Best case is constant i.e 1 is found at position 0;

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

$\mathcal{O}(n^2)$ Example I

- $\mathcal{O}(n^2)$ 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. $\mathcal{O}(n^3)$, $\mathcal{O}(n^4)$, etc.
- Consider the code on the next slide:
  - Worst case depends on the square of the number of elements in the array i.e. we might have to traverse the array completely checking against every other element in the array
  - best case is constant

$\mathcal{O}(n^2)$ Example II

```
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;
}
```

$\mathcal{O}(n^2)$ Example III

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


test1 = [0,1,2,3,4,5]
test2 = [6,12,13,14,6,12]

print(contains_duplicates(test1))
print(contains_duplicates(test2))
```

The End