Analysis of Algorithms

COSC 1P03 – Lecture 02 (Spring 2024)

Maysara Al Jumaily

amaysara@brocku.ca

Brock University

Tuesday May 07, 2024

Total slides: 23

Lecture Outline

Intuition of Asymptotic Complexity I

- Consider this task: we want to see which runs faster out of the two algorithms: Algorithm A and Algorithm B
- How could we test this?
- Naïvely, we can time them in seconds and see which runs faster
  - This will depend on the computer's speed
  - Running it on a faster/slower computer give you different results
  - Not a good measure of finding which algorithm is faster as the results depends on which computer is being used
  - Is there a measure we can use to see the efficiency of algorithms without relying on computers?
    - Here is when complexity theory notation comes in!
    - We will go over the three asymptotic complexities: Big–$\mathcal{O}$, Big–Θ and Big-Ω

Intuition of Asymptotic Complexity II

- Big–$\mathcal{O}$ complexity is a notation used to measure the **<u>maximum number</u>** of steps needed for an algorithm to complete
- For example: suppose you want to find your test paper in a pile of n test papers. A algorithm could be as follows:
  - Loop through all the papers and get the first paper
  - If it is your paper, grab it and stop looking
  - If it is not your paper, go to the next paper
  - Repeat
- How many steps is required to find your paper?
  - There will be three different cases: *best case*, *average case* and *worst case*
  - Best case: takes 1 step (*i.e.*, it is the first paper)
  - Average case: takes $\frac{n}{2}$ steps (*i.e.*, somewhere in the middle)
  - Worst case: takes n steps (*i.e.*, the last paper in the pile)
  - We say that this example is big-Oh of n because it takes n steps worst case scenario and is written as $\mathcal{O}(n)$
- Big–$\mathcal{O}$ complexity is interested in <u>worst case</u> scenario which is the upper bound of the number of steps required
- Big–Ω (Big-Omega) which deals with the best case and Big–Θ (Big-Theta) which deals with the average case

Mathematical Definition of Asymptotic Complexity I

- Asymptotic complexities are theoretical notion, meaning they are not perfect and focus on large number of steps/large problem size
- By large numbers, we mean as the number of steps become 100 trillion, for example. Mathematicians like to refer to this as "*the number of steps approaches* ∞"
- Constants are not ignored, as the size of n increases, the constants become insignificant
- For example, if a task takes $\frac{501 \cdot n}{2}$ steps to complete, then this is the same as n steps
- Think about $n = 9999^{99999}$, does dividing by 2 influence the value? Not quite
- Exponents are important and will be kept, so n^2 will stay as n^2

Mathematical Definition of Asymptotic Complexity II

- In the example of finding your test paper in a pile of n test papers, the worst case is that you go through all n papers and find it. So, you require n steps to complete the task, or $\mathcal{O}(n)$
- Can you complete the task in $\mathcal{O}(2n)$ steps (loop twice). How about $\mathcal{O}(3n)$ times (loop three times)? How about $\mathcal{O}(n^2)$ times?
  - Of course we can! It would be useless and time consuming but it gets the job done
- This means that the task requires at least n hence $\mathcal{O}(n)$ and can be completed with more steps
- Anything less than $\mathcal{O}(n)$ steps, *i.e.*, $\mathcal{O}(\log n)$, is invalid. This is because $\mathcal{O}(\log n)$ is not enough to satisfy the worst case scenario
- In general, asymptotic complexity focuses on the tightest bound that is required to complete the task

Definition 2.1: Big-oh $\mathcal{O}(\ldots)$

Let $f(x)$ and $g(x)$ be functions from the set of integers or the set of real numbers to the set of real number. We say that $f(x)$ is $\mathcal{O}(g(x))$ if there are constants C and k such that

$$f(x) \leq C \cdot g(x), \tag{1}$$

for $x \geq k$. We read this as "$f(x)$ is big-oh of $g(x)$". The pair (C, k) are referred to as *witnesses*. Note that it is less than or equal to, not strictly less than.

Example 2.1: Big-oh $\mathcal{O}(\dots)$

Suppose we have the function $f(x) = 2 \cdot x^3 + 5 \cdot x^2 + 12 \cdot x + 42$. Show that $f(x)$ is $\mathcal{O}(x^3)$. We need to choose two values (C, k) such that:

$$\begin{array}{ccc} f(x) & \leq & C \cdot g(x) \\ 2 \cdot x^3 + 5 \cdot x^2 + 12 \cdot x + 42 & \leq & C \cdot x^3 \end{array}$$

Let us replace each term in $f(x)$ with x^3 but keep the constant:

$$\begin{array}{ccc} f(x) & \leq & C \cdot g(x) \\ 2 \cdot x^3 + 5 \cdot x^2 + 12 \cdot x + 42 & \leq & 2 \cdot x^3 + 5 \cdot \boxed{x^3} + 12 \cdot \boxed{x^3} + 42 \cdot \boxed{x^3} \\ 2 \cdot x^3 + 5 \cdot x^2 + 12 \cdot x + 42 & \leq & 61 \cdot x^3 \end{array}$$

Hence, having the witnesses $(C = 61, k = 1)$ concludes that $f(x)$ is $\mathcal{O}(x^3)$.

- It wouldn't be wrong to say that $f(x)$ is $\mathcal{O}(x^4)$, or $\mathcal{O}(x^5)$, or even $\mathcal{O}(x^{10000})$, but we care about the tightest bound
- However, it would be wrong to say $f(x)$ is $\mathcal{O}(x^2)$

Definition 2.2: Big-omega $\Omega(\dots)$

Let $f(x)$ and $g(x)$ be functions from the set of integers or the set of real numbers to the set of real number. We say that $f(x)$ is $\Omega(g(x))$ if there are constants C and k such that

$$f(x) \geq C \cdot g(x). \tag{2}$$

for $x \geq k$. We read this as "$f(x)$ is big-Omega of $g(x)$". The pair (C, k) are referred to as *witnesses*. Note that it is less than or equal to, not strictly less than.

Example 2.2: Big-omega example

Suppose we have the function $f(x) = 2 \cdot x^6 + 3 \cdot x^5 + x + 9$. Show that $f(x)$ is $\Omega(x^6)$. We need to choose two values (C, k) such that:

$$\begin{aligned} f(x) &\geq C \cdot g(x) \\ x^6 + 3 \cdot x^5 + x + 9 &\geq C \cdot x^3 \end{aligned}$$

Let us choose the C value so that it is as close as possible to the function $f(x)$. We can simply use the constant of the dominant term (*i.e.*, x^6), which is 2:

$$\begin{aligned} f(x) &\geq C \cdot g(x) \\ 2 \cdot x^6 + 3 \cdot x^5 + x + 9 &\geq \boxed{2} \cdot \boxed{x^6} \\ 2 \cdot x^6 + 3 \cdot x^5 + x + 9 &\geq 2 \cdot x^6 \end{aligned}$$

Hence, having the witnesses $(C = 2, k = 1)$ concludes that $f(x)$ is $\Omega(x^6)$.

Definition 2.3: Big-Theta $\Theta(\ldots)$

Let $f(x)$ and $g(x)$ be functions from the set of integers or the set of real numbers to the set of real number. We say that $f(x)$ is $\Theta(g(x))$ if there are constants c_1 and c_2 such that

$$c_1 \cdot g(x) \leq f(x) \leq c_2 \cdot g(x), \tag{3}$$

for some $x > k$. We read this as "$f(x)$ is big-Theta of $g(x)$". This concludes that the function $f(x)$ is of order $g(x)$, and that $f(x)$ and $g(x)$ are of the same order. In other words, the dominant term is the same in both $f(x)$ and $g(x)$.

Example 2.3: Big-theta example

Suppose we have the function $f(x) = 5 \cdot x^3 + 2 \cdot x^2 + 10$. Show that $f(x)$ is $\Omega(x^3)$. We need to choose three values (c_1, c_2, k) such that:

$$c_1 \cdot g(x) \quad \leq \quad 5 \cdot x^3 + 2 \cdot x^2 + 10 \quad \leq \quad c_2 \cdot g(x)$$

Let us choose:

- c_1 as constant of the dominant term in $f(x)$, which is 5
- c_2 as the sum of the constants in $f(x)$, which is $5 + 2 + 10 = 17$
- $g(x)$ as the dominant function in $f(x)$, which is x^3
- This will work for values of $x \geq 1$ (*i.e.*, $k = 1$)

$$\begin{array}{ccccc}
c_1 \cdot g(x) & \leq & 5 \cdot x^3 + 2 \cdot x^2 + 10 & \leq & c_2 \cdot g(x) \\
c_1 \cdot \boxed{x^3} & \leq & 5 \cdot x^3 + 2 \cdot x^2 + 10 & \leq & c_2 \cdot \boxed{x^3} \\
\boxed{5} \cdot \boxed{x^3} & \leq & 5 \cdot x^3 + 2 \cdot x^2 + 10 & \leq & \boxed{17} \cdot \boxed{x^3}
\end{array}$$

Hence, having the witnesses $(c_1 = 5, c_2 = 17, k = 1)$ concludes that $f(x)$ is $\Theta(x^3)$.

Big–$\mathcal{O}$ Complexity – Order of Functions I

- Here is a chart of the most common functions with names assuming the number of elements is n:

Function	Name	Efficiency
$\mathcal{O}(1)$ or $\mathcal{O}(c)$ or $\mathcal{O}(C)$	Constant	Fast
$\mathcal{O}(\log n)$	Logarithmic	Fast
$\mathcal{O}(n)$	Linear	Fast
$\mathcal{O}(n \cdot \log n)$	Loglinear	Fast
$\mathcal{O}(n^2)$	Quadratic	Slow
$\mathcal{O}(n^3)$	Cubic	Slow
$\mathcal{O}(n^a)$	Polynomial for integer $a \geq 1$	Slow
$\mathcal{O}(2^n)$	Exponential	Very slow
$\mathcal{O}(3^n)$	Exponential	Very slow
$\mathcal{O}(a^n)$	Exponential for integer $a > 1$	Very slow
$\mathcal{O}(n!)$	Factorial	Very slow

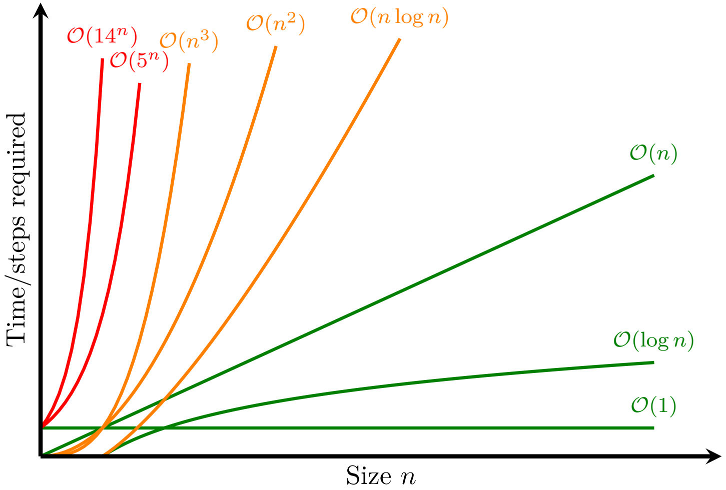
- Here is the order of functions, the ones in green (top) are the fastest and the ones in red (bottom) are the slowest:
$\mathcal{O}(c) < \mathcal{O}(\log n) < \mathcal{O}(\sqrt{n}) < \mathcal{O}(n) < \mathcal{O}(n \log n) <$
$\mathcal{O}(n^2) < \mathcal{O}(n^3) < \mathcal{O}(n^4) < \mathcal{O}(n^a) <$
$\mathcal{O}(a^n) < \mathcal{O}(n!) < \mathcal{O}(n^n) < \mathcal{O}(n^{2^n})$

- To understand the significance, compare the functions with different n values (note that $\mathcal{O}(1)$, which is constant, is independent of n):

Function	n=2	n=10	n=20	n=1000
$\mathcal{O}(1)$	1	1	1	1
$\mathcal{O}(\log n)$	0.3	1	1.3	3
$\mathcal{O}(n)$	2	10	20	1000
$\mathcal{O}(n \cdot \log n)$	0.6	10	1561.24	3600
$\mathcal{O}(n^2)$	4	100	400	1000000
$\mathcal{O}(n^3)$	8	1000	8000	1000000000
$\mathcal{O}(2^n)$	4	1024	1048576	Too large
$\mathcal{O}(3^n)$	9	59049	3486784401	Too large
$\mathcal{O}(n!)$	2	3628800	2432902008176640000	Too large

- Here is the order of functions, the ones in green (top) are the fastest and the ones in red (bottom) are the slowest:
  $\mathcal{O}(c) < \mathcal{O}(\log n) < \mathcal{O}(\sqrt{n}) < \mathcal{O}(n) < \mathcal{O}(n \log n) <$
  $\mathcal{O}(n^2) < \mathcal{O}(n^3) < \mathcal{O}(n^4) < \mathcal{O}(n^a) <$
  $\mathcal{O}(a^n) < \mathcal{O}(n!) < \mathcal{O}(n^n) < \mathcal{O}(n^{2^n})$

- The graphs of the typical functions are shown next
- In computer science, our aim is to have algorithms that have minimal growth:



Big–$\mathcal{O}$ of a Function

- The previous slide only contained a single-term functions. For example, we had $\mathcal{O}(n^2)$ instead of, say, $\mathcal{O}(2n^2 + \frac{n}{4} - 5)$
- Suppose that an algorithm takes $\mathcal{O}(2n^2 + \frac{n}{4} - 5)$ steps to complete, what is its big-$\mathcal{O}$?
  - Remember, $\mathcal{O}$ notation is theoretical, not exact and cares about large values of n
  - We don't care about constants as they don't have significance when n approaches ∞
  - Hence, $\mathcal{O}(2n^2 + \frac{n}{4} - 5)$ becomes $\mathcal{O}(n^2 + n)$
  - We'll be interested in the highest term, the biggest term. We know that $n^2 > n$. We ignore n and have $\mathcal{O}(n^2)$
  - Hence $\mathcal{O}(2n^2 + \frac{n}{4} - 5)$ is $\mathcal{O}(n^2)$
- Another approach is to break them down like so: $\mathcal{O}(2n^2+\frac{n}{4}-5) = \mathcal{O}(2n^2)+\mathcal{O}(\frac{n}{4})-\mathcal{O}(5) = \mathcal{O}(n^2)+\mathcal{O}(n)-\mathcal{O}(5) = \mathcal{O}(n^2)$

Asymptotic Complexity in Code I

There are six situations to keep in mind:

- Constant/independent of the size
  - This will always have a constant value, regardless of the input size n
- Non-nested loops
  - **Add** their big-$\mathcal{O}$ complexities and then keep the most dominant term
- Nested loops
  - **Multiply** their big-$\mathcal{O}$ complexities and then keep the most dominant term
- `if`/`else if`/`else` statements containing loops
  - The condition in the `if` statement itself is constant
  - Go with the worst-case scenario and choose the branch that has the largest running time
- Loops with counters that get divided or multiplied each iterations (*i.e.*, the counter doesn't increase/decrease by 1 or constant)

Asymptotic Complexity in Code II

- Constant/independent of n:

```
int n = 100;
for(int i = 0; i <= 7; i++){
  ...
}
```

$\mathcal{O}(8) = \mathcal{O}(C) = \mathcal{O}(c)$

- Non-nested loops:

```
for(int i = 0; i <= n; i++){
  ...
}
```

$\mathcal{O}(n+1) = \mathcal{O}(n)$

```
for(int i = 0; i < 2*n; i++){
  ...
}
```

$\mathcal{O}(2 \cdot n) = \mathcal{O}(n)$

Total is: $\mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(2 \cdot n) = \mathcal{O}(n)$

- Nested loops:

```
for(int i = 0; i < n; i++){          // O(n)
  for(int i = 0; i < 2*n; i++){      // O(2·n) = O(n)
    ...
  }
}
```

$\mathcal{O}(n)$

$\mathcal{O}(2 \cdot n) = \mathcal{O}(n)$

Total is: $\mathcal{O}(n) \cdot \mathcal{O}(n) = \mathcal{O}(n^2)$. Another way would be to solve all of the terms together: $\mathcal{O}(n \cdot (2 \cdot n)) = \mathcal{O}(2 \cdot n^2) = \mathcal{O}(n^2)$

- `if`/`else if`/`else` statements containing loops
- The evaluation of a condition is $\mathcal{O}(1)$ or $\mathcal{O}(c)$

```
if(...){        O(1) = O(C) = O(c) Ignored
  for(int i = 0; i <= 7; i++){    O(8) = O(C) = O(c) Ignored
    ...
  }
} else if (...){     O(1) = O(C) = O(c) Chosen
  for(int i = 0; i <= n; i++){    O(n + 1) = O(n)
    for(int i = 0; i < 2*n; i++){    O(2 · n) = O(n)
      ...
    }
  }
} else { Ignored
  for(int i = 0; i <= n; i++){    O(n + 1) = O(n)
    ...
  }
}
```

Total is (`else if` block): $\mathcal{O}(c) + \mathcal{O}(n) \cdot \mathcal{O}(n) = \mathcal{O}(n^2)$. Another way would be to solve all of the terms together:
$\mathcal{O}(c + (n + 1) \cdot (2 \cdot n)) = \mathcal{O}(c + 2 \cdot n^2 + 2 \cdot n) = \mathcal{O}(n^2)$

Asymptotic Complexity in Code IV

- Loops with counters that get divided or multiplied each iterations (*i.e.*, the counter doesn't increase/decrease by 1 or constant)
- We ignore the base when it comes to logarithmic functions

```
for(int i = 0; i < n; i= i / 2){
  ...
}
```

$\mathcal{O}(\log_2 n) = \mathcal{O}(\log n)$

The above is logarithmic, which means each iteration, we focus only on $\frac{1}{2}$ of the data. In case we had `i = i / 3`, then we focus on only $\frac{1}{3}$ of the data each iteration, which is $\mathcal{O}(\log_3 n)$.

- Exponential complexity is found below

```
int x = 0;
for(int i = 0; i < n; i++){
  ...
  if(x < 10){ // say 10 times, watch out from infinite loop!
    n = n * 2;
    x = x + 1;
  }
}
```

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

- Each iteration we complete doubles the data we need to go through. In case we had `n = n * 3`, then the complexity become $\mathcal{O}(3^n)$

What is binary search?

- Binary search is an efficient way to search/find an element in some data
- Two conditions must be satisfied to perform binary search:
  - Elements **must** be sorted
  - Any element can be accessed in $\mathcal{O}(1)$ or $\mathcal{O}(C)$
  - We could use an array to find the index of an element (assuming the values in the array are sorted)
- The game of guessing a number between 1 and n is the best way to understand binary search:
  - You guess a number between 1 and n
  - I try to guess it by saying a number
  - You will tell me I guessed or should guess higher or lower
  - I guess again based on your feedback

Binary Search Example

- From 1 and 16, assume you choose the value 7 as your number:

1	2	3	4	5	6	**7**	8	9	10	11	12	13	14	15	16

- Here is how we will guess as *efficiently* as possible:
- We guess the middle, 8. You tell me lower. The new range is:

1	2	3	4	5	6	**7**	8	9	10	11	12	13	14	15	16

- We guess the middle, 4. You tell me higher. The new range is:

1	2	3	4	5	6	**7**	8	9	10	11	12	13	14	15	16

- We guess the middle, 6. You tell me higher. The new range is:

1	2	3	4	5	6	**7**	8	9	10	11	12	13	14	15	16

- We guess the middle, 7 and it must be the correct element.

The complexity of binary search

- Binary search is efficient because it takes $\mathcal{O}(\log n)$ of steps to reach desired element.
  - It is $\mathcal{O}(\log n)$ because every time we guess, we ignore half of the elements. Our range gets divided by 2 every time
  - In other words, every time we divide our range by 2, it is $\mathcal{O}(\log n)$
  - Again, we can do that because the elements are *sorted*
  - if not sorted, we would visit each single element. The worst case is $\mathcal{O}(n)$
    - Guess the first element, if it is, stop, otherwise, go to second element, etc
- It is not a coincidence that we chose the middle each time we guessed. Let us see another example where my approach is extremely bad:
- Assume the value you chose is 1:
  - We choose 16, you say lower
  - We choose 15, you say lower
  - $\vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots$
  - We choose 3, you say lower
  - We choose 2, you say lower
  - We choose 1, you say correct
  - This approach required to guess all the elements, hence, it's $\mathcal{O}(n)$
- The efficiency depends on the element we choose; In binary search, **always** use the **middle** element