

CS 441: Complexity of Algorithms

PhD. Nils Murrugarra-Llerena
nem177@pitt.edu



[Key CS Link] Complexity

The Card Sorting Problem

Imagine you have a deck of N cards and you need to sort them in ascending order.

What will be your approach? Can we use Bubble Sort? Can we improve it?



Today's topics

- Recap Growth rates of functions
- Applying growth rates to algorithms
 - Resource utilization functions and applying big-O
 - Complexity of algorithms
 - Worst case
 - Best case
 - Average case



[Reminder] What is an algorithm?

Definition: An **algorithm** is a finite sequence of precise instructions for solving a problem

Note these important features!

- **Finite:** In order to execute, it must be finite
- **Sequence:** The steps needs to be in the correct order
- **Precise:** Each step must be unambiguous
- **Instructions:** Each step can be carried out
- Solving a problem: ?

[Reminder] Big-O notation

Definition: Let f and g be functions from the set of integers (or real numbers) to the set of real numbers. We say that $f(x)$ is $O(g(x))$ if there are constants C and k such that $|f(x)| \leq C|g(x)|$ whenever $x \geq k$.

- C and k are referred to as **witnesses** which prove the relationship

Formally, $O(g(x))$ is a set of functions:

$$O(g(x)) = \{f \mid \exists k, C \forall x (x \geq k \rightarrow |f(x)| \leq C|g(x)|)\}$$

Examples:

When considering positive values only, we will often drop the absolute value

- $2x^2$ is $O(x^2)$ because of witnesses $C = 3$ and $k = 1$:
 $2x^2 \leq 3x^2$ whenever $x \geq 1$
- $3x + 5$ is $O(x)$ because of witnesses $C = 4$ and $k = 5$:
 $3x + 5 \leq 4x$ when $x \geq 5$

[Reminder] Related notations to big-O

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

- If big-O represents an asymptotic upper bound, big-Omega represents an asymptotic **lower bound**
- (Asymptotic = at scale, as x increases toward infinity)

Examples:

- $2x^2$ is $\Omega(x^2)$, $\Omega(x)$, and $\Omega(1)$
 - In addition to being $O(x^2)$, $O(x^3)$, $O(x^4)$, ...

When $f(x)$ is both $O(g(x))$ and $\Omega(g(x))$, we say it is $\Theta(g(x))$, so $2x^2$ is $\Theta(x^2)$

- “Big theta”

[Reminder] Why does algorithm analysis matter?

“The growth of $f(x)$ is bounded above by some multiple of $g(x)$.”

- What does this tell us, if $f(x)$ describes an algorithm's **cost** to solve an instance of size x ?

Big-O notation is used in algorithm analysis to group algorithms together

- Simple growth rate is more important than exact runtime
- Algorithm analysis describes how algorithms **scale** to larger and larger problem instances
- The difference between algorithms is much wider than the differences in hardware can overcome
- Hardware improvements are constant multiplicative factors

Today: Applying growth rates to algorithms

Resource utilization functions and applying big-O

Complexity of algorithms

- Worst case
- Best case
- Average case

Let's motivate with an example

Problem: Sum the integers from 1 through n

Algorithm A

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

Algorithm B

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

Algorithm C

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

Analysis idea: Identify repeated instructions, count frequency

How many operations for these algorithms?

	Algorithm A	Algorithm B	Algorithm C
Additions	n	$\frac{n * (n + 1)}{2}$	1
Multiplications			1
Divisions			1
Total operations	n	$\frac{n^2}{2} + \frac{n}{2}$	3

Some operations may take longer...

... but as the input gets larger, the frequency is the most important factor

How many operations does this work out to be, for different inputs?

	Algorithm A	Algorithm B	Algorithm C
n = 1	1	1	3
n = 10	10	55	3
n = 100	100	5,050	3
n = 1000	1,000	500,500	3

*Algorithm analysis focuses on trends as the
problem instances grow in size
(Measure runtimes as input size grows)*

*Next year, computers might be twice as fast, but a
bad algorithm is still 500 times slower*

How do we measure the runtime of an algorithm?

First, consider expressing the runtime as a **function**

- Domain: Natural numbers (**Why?**)
- Preimages represent the size of a problem instance

We rarely need to articulate this function exactly

- Different hardware can change multiplicative constants
- Optimization can reduce constants and lower-order terms
- As such, **growth rates** are effective at describing what is inherent in the algorithm
 - (rather than how it is implemented)

For runtime: Identify the operations that happen most **frequently**, and determine the **growth rate** of how many

Practice: Max algorithm, pseudocode

```
procedure max( $a_1, a_2, \dots, a_n$ : integers)
  max := 1
  for  $i := 2$  to  $n$ 
    if  $a_i > a_{max}$  then
      max :=  $i$ 
  return max
```

What is the most frequent operations?

How many of these operations occur, expressed as a growth rate?

What about an algorithm with variability, even for a given size?

```
procedure linear search(x: integer, a1, a2, ..., an: distinct integers)
  i := 1
  while (i ≤ n and x ≠ ai)
    i := i + 1
  if i ≤ n then
    location := i
  else
    location := 0
  return location {location is the subscript of the term that equals x, or is 0 if x is
  not found}
```

We can consider different scenarios for an algorithm

Worst case runtime

- Growth rate of the **worst** possible input of size n
 - This is the default, if a case is not specified
- e.g., Linear search for the very last item, or an item that is not found

Best case runtime

- Growth rate of the **best** possible input of size n
- e.g., Linear search for the very first item

Average case runtime

- Growth rate of the **average** input of size n
- Average in what way? Need a probability distribution over possible inputs

Note: We can use big-O, big-Ω, and big-Θ for each case!

Worst case? Best case?

procedure *linear search*(x : integer, a_1, a_2, \dots, a_n : distinct integers)

$i := 1$

while ($i \leq n$ and $x \neq a_i$)

$i := i + 1$

if $i \leq n$ **then**

$location := i$

else

$location := 0$

return $location$ { $location$ is the subscript of the term that equals x , or is 0 if x is not found}

What about average case?

In order to calculate runtime in the average case, we need a **probability distribution** for inputs

- i.e., how frequently each input is expected
- What if we almost always search for the first item?
- What if we almost always search for an item that can't be found?

Most commonly, we'll consider the uniform distribution, where all inputs are **equally likely**

- For instance, consider linear search where the target is found, and each location is equally likely to contain the target
- Average the cost, weighted by the probability for each input

$$\sum_{i \in \text{Inputs}} \text{Pr}(i) \times \text{Cost}(i)$$

Demonstrate for linear search!

What about average case?

On Whiteboard

Let's analyze bubble sort

```
procedure bubble sort( $a_1, a_2, \dots, a_n$ : real numbers)
  for  $i := 1$  to  $n-1$ 
    for  $j := 1$  to  $n-1$ 
      if  $a_j > a_{j+1}$  then
        swap  $a_j$  and  $a_{j+1}$ 
```

How many operations? (i.e., **comparisons**)

- Outer loop has $\Theta(n)$ iterations
- Inner loop has $\Theta(n)$ iterations for each outer-loop iteration
- Work inside loop (plus loop overhead) is $\Theta(1)$
- Remember that repetition can be calculated using multiplication
- Total runtime: $\Theta(n * n * 1) = \Theta(n^2)$

What about an improved version?

```
procedure bubble sort( $a_1, a_2, \dots, a_n$ : real numbers)
  for  $i := 1$  to  $n-1$ 
    for  $j := 1$  to  $n-i$ 
      if  $a_j > a_{j+1}$  then swap  $a_j$  and  $a_{j+1}$ 
```

How many operations? (i.e., **comparisons**)

- Outer loop has $\Theta(n)$ iterations
- Inner loop is variable
 - $O(n)$ iterations
 - $\Omega(1)$ iterations
- $O(n^2)$ and $\Omega(n)$. Can we get an exact bound?

Common growth rates and their terminologies for complexity

Complexity in n	Terminology
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$	Exponential complexity
$\Theta(n!)$	Factorial complexity

These are considered intractable

***Consider increasing the instance size:
How will runtime change for each?***

In-class Activities

Activity 1: Prove that $\log_b(n)$ is $O(\log n)$ for any constant b . [[miro](#)]

Activity 2: What is the worst-case complexity of this algorithm? (Express in terms of n .) [[miro](#)]

```
procedure problem 2( $n$ : integer)
   $x := 1$ 
  result := 0
  while ( $x \leq n$ )
    for  $i := 1$  to  $x$ 
      result := result + 1
     $x := x * 2$ 
  return result
```



Steps:

1. Introduce to a classmate
2. Work in pairs on the exercise
3. Submit answers on miro
4. Volunteers to share answers

Final thoughts

- We can use **growth rates** to study algorithms and their runtime
- **Big-O** and related notations are useful for complexity since they represent the runtime trends at scale
- Next time:
 - Starting number theory: Divisibility and modular arithmetic (Section 4.1)