0: Previous Knowledge

CPCFI

UNAM's School of Engineering

2021

Based on: Halim S., Halim F.*Competitive Programming 3*. Handbook for ACM ICPC and IOI Contestants. 2013

Table of Contents

Algorithm Analysis

Given the maximum input bound, can the currently developed algorithm, with its time/space complexity, pass the time/memory limit given for that particular problem?

Algorithm Analysis

Computers can run 10^8 operations per second [1]. We can use this information to know if our algorithm will run in time.
Examples:

- Input size: $n = 10^5$. Algorithm complexity: $O(n^2)$. Time: $O(n^2) = (10^5)^2 = 10^{10}$. The algorithm will need hundreds of seconds to finish
- If the algorithm complexity is: $O(n \log_2 n)$, then time would be: $10^5 \log_2 10^5 \approx 1.7 \times 10^6$ which will pass the time limit

The algorithm complexity will decide if its worth it to implement a certain algorithm or not.

> *Start coding an algorithm only when you know it is correct and fast enough*

[1] $10^8 = 100,000,000 = 100M$

Time Complexities for n input

n	Worst AC Algorithm	Comment
$\leq [10..11]$	$O(n!), O(n^6)$	e.g. Enumerating permutations (Section 3.2)
$\leq [15..18]$	$O(2^n \times n^2)$	e.g. DP TSP (Section 3.5.2)
$\leq [18..22]$	$O(2^n \times n)$	e.g. DP with bitmask technique (Section 8.3.1)
≤ 100	$O(n^4)$	e.g. DP with 3 dimensions + $O(n)$ loop, ${}_nC_{k=4}$
≤ 400	$O(n^3)$	e.g. Floyd Warshall's (Section 4.5)
$\leq 2K$	$O(n^2 \log_2 n)$	e.g. 2-nested loops + a tree-related DS (Section 2.3)
$\leq 10K$	$O(n^2)$	e.g. Bubble/Selection/Insertion Sort (Section 2.2)
$\leq 1M$	$O(n \log_2 n)$	e.g. Merge Sort, building Segment Tree (Section 2.3)
$\leq 100M$	$O(n), O(\log_2 n), O(1)$	Most contest problem has $n \leq 1M$ (I/O bottleneck)

Figure: Time complexities for a given n

Table of Contents

Algorithm Analysis

- Algorithms can be analyzed in a machine independent way
- We need techniques to compare the efficiency of algorithms before implementing them
  - RAM Model
  - Asymptotic Analysis - Big O notation
- Algorithms can be analyzed from two perspectives:
  - **Time**: measured in units of time and represents the amount of time needed for an algorithm to complete
  - **Memory**: measured in units of memory and represents the amount of memory used by the algorithm
- **Time** and **Memory** have an indirect relationship. Usually if we want less time we need more memory and viceversa

AA - The RAM Model

In the RAM model we measure run time by counting the number of steps an algorithm takes on a given problem instance
*- Skiena [**?**]*

AA - The RAM Model

- Machine independent algorithms depend on a hypothetical computer called *Random Access Machine*
- Each single operation $(+, -, *, , =, \text{if})$ takes one time step
- Loops and subroutines are a composition of multiple single operations. The time step of a loop depends on the number of iterations the loop makes
- Each memory access takes exactly one time step

AA - The RAM Model

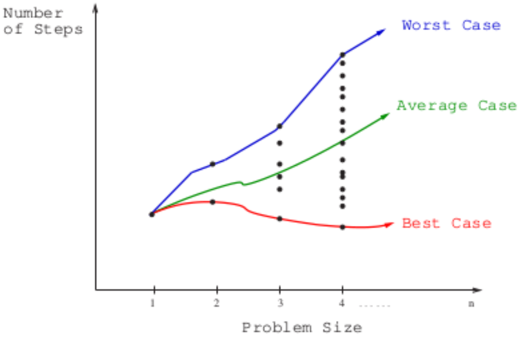


Figure: Best, Worst and Avergae case - See: [?]

AA - Big O notation

- Provides a better intuition than the RAM Model since it:
  - Require too much detail to specify precisely its behavior
  - Many times the steps count depends on the programmer
- Big O notation solves this issues by providing upper and lower bounds of time-complexity functions
- Big O notation ignores multiplicative constants
  - $f(n) = 2n$ is considered the same as $g(n) = n$

AA - Big O notation

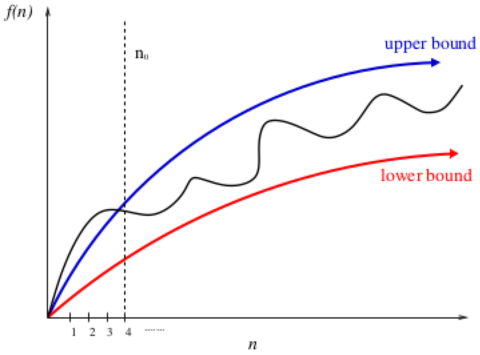
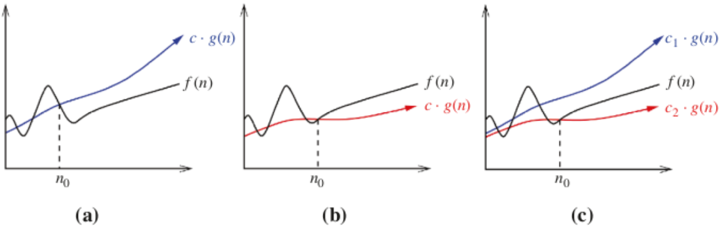


Figure: Upper and lower bounds valid for $n > n_0$

AA - Big O notation



- **a)**: $f(n) = O(g(n))$

$$f(n) \leq c \cdot g(n)$$

- **b)**: $f(n) = \Omega(g(n))$

$$f(n) \geq c \cdot g(n)$$

- **c)**: $f(n) = \Theta(g(n))$

$$c_2 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n)$$

AA - Big O notation: Examples

We want to identify $g(n)$ from $f(n)$:

$f(n)$	Possible $g(n)$	Test with c	Result
$3n^2 - 100n + 6$	n^2	$f(n) < 3n^2$	$f(n) = O(n^2)$

AA - Big O notation: Examples

We want to identify $g(n)$ from $f(n)$:

$f(n)$	Possible $g(n)$	Test with c	Result
$3n^2 - 100n + 6$	n^2	$f(n) < 3g(n)$	$f(n) = O(n^2)$
$3n^2 - 100n + 6$	n^3	$f(n) < 1g(n)$	$f(n) = O(n^3)$
$3n^2 - 100n + 6$	n	$f(n) > g(n)$	$f(n) \neq O(n)$

Table: Tests for O

AA - Big O notation: Examples

We want to identify $g(n)$ from $f(n)$:

$f(n)$	Possible $g(n)$	Test with c	Result
$3n^2 - 100n + 6$	n^2	$f(n) > 2g(n)$	$f(n) = \Omega(n^2)$

AA - Big O notation: Examples

We want to identify $g(n)$ from $f(n)$:

$f(n)$	Possible $g(n)$	Test with c	Result
$3n^2 - 100n + 6$	n^2	$f(n) > 2g(n)$	$f(n) = \Omega(n^2)$
$3n^2 - 100n + 6$	n^3	$f(n) < g(n)$	$f(n) \neq \Omega(n^3)$
$3n^2 - 100n + 6$	n	$f(n) > g(n)$	$f(n) = \Omega(n)$

Table: Tests for Ω

AA - Big O notation: Examples

We want to identify $g(n)$ from $f(n)$:

$f(n)$	Possible $g(n)$	Test with c	Result
$3n^2 - 100n + 6$	n^2	Both O and Ω apply	$f(n) = \Theta(n^2)$
$3n^2 - 100n + 6$	n^3	Θ does not apply	$f(n) \neq \Theta(n^3)$
$3n^2 - 100n + 6$	n	O does not apply	$f(n) \neq \Omega(n)$

Table: Tests for Θ

AA - Big O notation: Dominance Relations

- Big O notations groups functions into a set of classes such that all functions within a particular class are essentially equivalent
  - $f(n) = 230n$ is equivalent to $g(n) = 1.5n$
  - Both $f(n)$ and $g(n)$ belong to $\Theta(n)$
- A faster growing function dominates a slower growing one
  - We say that $g(n)$ dominates $f(n)$ when $f(n) = O(g(n))$
  - $g \gg f$

AA - Big O notation: Function Classes I

Remember that we always need to measure the performance of our function when $n \to \infty$

- *Constant functions*, $f(n) = 1$: no dependence between f and n. Might measure the cost of adding two numbers, obtaining min, max, ...
- *Logarithmic functions*, $f(n) = \log n$: f grows slowly as n gets big but grows faster than the constant function.
- *Linear functions*, $f(n) = n$: might measure the cost of looking at each item in an n-element array.
- *Superlinear functions*, $f(n) = n \log n$: grow a little faster than linear and might measure the cost of sorting an n-element array

AA - Big O notation: Function Classes II

- *Quadratic functions*, $f(n) = n^2$: measure the cost of looking at most or all pairs of items in an n-element universe (array, sets, lists, ...)
- *Cubic functions*, $f(n) = n^3$
- *Exponential functions*, $f(n) = c^n$: for a given constant $c > 1$. Might measure the cost when enumerating all subsets of n items
- *Factorial functions*, $f(n) = n!$: measure the cost of generating all permutations or orderings of n items

AA - Big O notation: Dominance relations

$$n! \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \log n \gg 1$$

AA - Running time of common function classes

n	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Figure: Running time of function classes in nano seconds

References

- Halim S., Halim F., *Competitive Programming 3*, Handbook for ACM ICPC and IOI Contestants. 2013
- Stroustrup B. *The C++ Programming Language*. Fourth ed.
- Skiena S. *The Algorithm Design Manual*. Springer. 2020