Running Time Evaluation

Worst-case and average-case performance

Lecturer: Georgy Gimel'farb

COMPSCI 220 Algorithms and Data Structures

Algorithm’s Efficiency

Efficiency of an algorithm is generally measured in following terms:

- the running time (i.e. the time it takes to terminate);
- the amount of memory it requires and
- other required computation resources.

Today, “other resources” often include communication bandwidth.

- An algorithm downloading, rather storing whatever data required does not need much memory but does need bandwidth!
- Examples: Google Earth and Google Maps.

Memory is relatively cheap at present, so that time complexity of an algorithm is very important.

A Brief Quiz on Asymptotic Time Complexity

1. Executing an algorithm with time complexity $O(n^2)$ exactly 10 times gives an algorithm with Big-Oh time complexity $O(n^2)$.
2. Executing an algorithm with time complexity $O(n)$ and then an algorithm with time complexity $O(n \log n)$ gives an algorithm with Big-Oh time complexity $O(n \log n)$.
3. Consider an algorithm consisting of three nested for-loops that iterate each n times, and a body for the innermost loop that executes in $O(n \log n)$. What is the total Big-Oh complexity of the algorithm?
   - The total Big-Oh complexity of the algorithm is $O(n^4 \log n)$.

A Brief Quiz on Asymptotic Time Complexity

1. Executing an algorithm with time complexity $O(n^2)$ exactly 10 times gives an algorithm with Big-Oh time complexity $O(n^2)$.
2. Executing an algorithm with time complexity $O(n)$ and then an algorithm with time complexity $O(n \log n)$ gives an algorithm with Big-Oh time complexity $O(n \log n)$.
3. Consider an algorithm consisting of three nested for-loops that iterate each n times, and a body for the innermost loop that executes in $O(n \log n)$. What is the total Big-Oh complexity of the algorithm?
   - The total Big-Oh complexity of the algorithm is $O(n^4 \log n)$.

A Brief Quiz on Asymptotic Time Complexity

1. Executing an algorithm with time complexity $O(n^2)$ exactly 10 times gives an algorithm with Big-Oh time complexity $O(n^2)$.
2. Executing an algorithm with time complexity $O(n)$ and then an algorithm with time complexity $O(n \log n)$ gives an algorithm with Big-Oh time complexity $O(n \log n)$.
3. Consider an algorithm consisting of three nested for-loops that iterate each n times, and a body for the innermost loop that executes in $O(n \log n)$. What is the total Big-Oh complexity of the algorithm?
    - The total Big-Oh complexity of the algorithm is $O(n^4 \log n)$.

A Brief Quiz on Asymptotic Time Complexity

1. Executing an algorithm with time complexity $O(n^2)$ exactly 10 times gives an algorithm with Big-Oh time complexity $O(n^2)$.
2. Executing an algorithm with time complexity $O(n)$ and then an algorithm with time complexity $O(n \log n)$ gives an algorithm with Big-Oh time complexity $O(n \log n)$.
3. Consider an algorithm consisting of three nested for-loops that iterate each n times, and a body for the innermost loop that executes in $O(n \log n)$. What is the total Big-Oh complexity of the algorithm?
   - The total Big-Oh complexity of the algorithm is $O(n^4 \log n)$.

A Brief Quiz on Asymptotic Time Complexity

1. Executing an algorithm with time complexity $O(n^2)$ exactly 10 times gives an algorithm with Big-Oh time complexity $O(n^2)$.
2. Executing an algorithm with time complexity $O(n)$ and then an algorithm with time complexity $O(n \log n)$ gives an algorithm with Big-Oh time complexity $O(n \log n)$.
3. Consider an algorithm consisting of three nested for-loops that iterate each n times, and a body for the innermost loop that executes in $O(n \log n)$. What is the total Big-Oh complexity of the algorithm?
   - The total Big-Oh complexity of the algorithm is $O(n^4 \log n)$.

A Brief Quiz on Asymptotic Time Complexity

1. Executing an algorithm with time complexity $O(n^2)$ exactly 10 times gives an algorithm with Big-Oh time complexity $O(n^2)$.
2. Executing an algorithm with time complexity $O(n)$ and then an algorithm with time complexity $O(n \log n)$ gives an algorithm with Big-Oh time complexity $O(n \log n)$.
3. Consider an algorithm consisting of three nested for-loops that iterate each n times, and a body for the innermost loop that executes in $O(n \log n)$. What is the total Big-Oh complexity of the algorithm?
   - The total Big-Oh complexity of the algorithm is $O(n^4 \log n)$.

A Brief Quiz on Asymptotic Time Complexity

1. If an algorithm A always executes in the same number or fewer steps than an algorithm B with time complexity $O(n^2)$, then what can you say about the algorithm A?

   The algorithm A has the same time complexity $O(n^2)$,

2. An algorithm A has time complexity $\Theta(n \log n)$, and an algorithm B always runs in 5% of the time that the algorithm A takes. What are the Big-Oh, Big-Omega, and Big-Theta time complexities of the algorithm B?
   - The algorithm B is $O(n \log n)$, $\Omega(n \log n)$, and $\Theta(n \log n)$.

3. Executing an algorithm A with time complexity $\Theta(n)$ exactly $\lfloor n/10 \rfloor$ times gives an algorithm with Big-Theta time complexity $\Theta(n^2)$.

A Brief Quiz on Asymptotic Time Complexity

1. If an algorithm A always executes in the same number or fewer steps than an algorithm B with time complexity $O(n^2)$, then what can you say about the algorithm A?

   The algorithm A has the same time complexity $O(n^2)$,

2. An algorithm A has time complexity $\Theta(n \log n)$, and an algorithm B always runs in 5% of the time that the algorithm A takes. What are the Big-Oh, Big-Omega, and Big-Theta time complexities of the algorithm B?
   - The algorithm B is $O(n \log n)$, $\Omega(n \log n)$, and $\Theta(n \log n)$.

3. Executing an algorithm A with time complexity $\Theta(n)$ exactly $\lfloor n/10 \rfloor$ times gives an algorithm with Big-Theta time complexity $\Theta(n^2)$.

A Brief Quiz on Asymptotic Time Complexity

1. If an algorithm A always executes in the same number or fewer steps than an algorithm B with time complexity $O(n^2)$, then what can you say about the algorithm A?

   The algorithm A has the same time complexity $O(n^2)$,

2. An algorithm A has time complexity $\Theta(n \log n)$, and an algorithm B always runs in 5% of the time that the algorithm A takes. What are the Big-Oh, Big-Omega, and Big-Theta time complexities of the algorithm B?

   - The algorithm B is $O(n \log n)$, $\Omega(n \log n)$, and $\Theta(n \log n)$.

3. Executing an algorithm A with time complexity $\Theta(n)$ exactly $\lfloor n/10 \rfloor$ times gives an algorithm with Big-Theta time complexity $\Theta(n^2)$.

A Brief Quiz on Asymptotic Time Complexity

1. If an algorithm A always executes in the same number or fewer steps than an algorithm B with time complexity $O(n^2)$, then what can you say about the algorithm A?

   The algorithm A has the same time complexity $O(n^2)$,

2. An algorithm A has time complexity $\Theta(n \log n)$, and an algorithm B always runs in 5% of the time that the algorithm A takes. What are the Big-Oh, Big-Omega, and Big-Theta time complexities of the algorithm B?

   - The algorithm B is $O(n \log n)$, $\Omega(n \log n)$, and $\Theta(n \log n)$.

3. Executing an algorithm A with time complexity $\Theta(n)$ exactly $\lfloor n/10 \rfloor$ times gives an algorithm with Big-Theta time complexity $\Theta(n^2)$.

A Brief Quiz on Asymptotic Time Complexity

1. If an algorithm A always executes in the same number or fewer steps than an algorithm B with time complexity $O(n^2)$, then what can you say about the algorithm A?

   The algorithm A has the same time complexity $O(n^2)$,

2. An algorithm A has time complexity $\Theta(n \log n)$, and an algorithm B always runs in 5% of the time that the algorithm A takes. What are the Big-Oh, Big-Omega, and Big-Theta time complexities of the algorithm B?

   - The algorithm B is $O(n \log n)$, $\Omega(n \log n)$, and $\Theta(n \log n)$.

3. Executing an algorithm A with time complexity $\Theta(n)$ exactly $\lfloor n/10 \rfloor$ times gives an algorithm with Big-Theta time complexity $\Theta(n^2)$.

A Brief Quiz on Asymptotic Time Complexity

1. If an algorithm A always executes in the same number or fewer steps than an algorithm B with time complexity $O(n^2)$, then what can you say about the algorithm A?

   The algorithm A has the same time complexity $O(n^2)$,

2. An algorithm A has time complexity $\Theta(n \log n)$, and an algorithm B always runs in 5% of the time that the algorithm A takes. What are the Big-Oh, Big-Omega, and Big-Theta time complexities of the algorithm B?
   - The algorithm B is $O(n \log n)$, $\Omega(n \log n)$, and $\Theta(n \log n)$.

3. Executing an algorithm A with time complexity $\Theta(n)$ exactly $\lfloor n/10 \rfloor$ times gives an algorithm with Big-Theta time complexity $\Theta(n^2)$.

Informal Definition

Measuring time complexity

A function $f(n)$ such that the running time $T(n)$ of a given algorithm is $\Theta(f(n))$ measures the **time complexity** of the algorithm.

- A **polynomial time** algorithm: $T(n)$ is $O(n^k)$ where k is a fixed positive integer.
- An **intractable** computational problem: iff no deterministic algorithm with polynomial time complexity exists for it.
    - Many problems are classed as intractable only because a polynomial solution is unknown.
    - **It is a very challenging task to find such a solution for one of them.**

Relative Growth of the Running Time $T(n)$

Let $T(n) = cf(n)$ where $f(n)$ is some function of n and c is a fixed constant.

Then the relative growth of the running time is

$$\tau(n) = \frac{T(n)}{T(n^\circ)} = \frac{cf(n)}{cf(n^\circ)} = \frac{f(n)}{f(n^\circ)}$$

where n° is the reference input size.

For $n^\circ = 8$:

$f(n)$	1	$\lg n$	$\lg^2 n$	n	$n \lg n$	n^2	n^3	2^n
$f(8)$	1	$\lg 8 = 3$	9	8	24	64	512	$2^8 = 256$
$\tau(n) = \frac{f(n)}{f(8)}$	1	$\frac{\lg n}{3}$	$\frac{\lg^2 n}{9}$	$\frac{n}{8}$	$\frac{n \lg n}{24}$	$\frac{n^2}{64}$	$\frac{n^3}{512}$	$\frac{2^n}{2^8} = 2^{n-8}$

For simplicity: $\lg n \equiv \log_2 n$

Relative Growth of the Running Time $T(n)$

$\tau(n) = \frac{f(n)}{f(8)}$ when the input size increases from $n = 8$ to $n = 1024$:

Time complexity		**Input size n**				$\tau(n)$
Function f	*Notation*	*8*	*32*	*128*	*1024*	
Constant	1	1	1	1	1	1
Logarithmic	$\lg n$	1	1.67	2.67	3.33	$\frac{\lg n}{3}$
Log-squared	$\lg^2 n$	1	2.78	5.44	11.1	$\frac{\lg^2 n}{9}$
Linear	n	1	4	16	128	$\frac{n}{8}$
Linearithmic	$n \lg n$	1	6.67	37.3	427	$\frac{n \lg n}{24}$
Quadratic	n^2	1	16	256	16384	$\frac{n^2}{64}$
Cubic	n^3	1	64	4096	$2.1 \cdot 10^6$	$\frac{n^3}{512}$
Exponential	2^n	1	2^{24}	2^{120}	2^{1016}	$\frac{2^n}{2^8} = 2^{n-8}$

For simplicity: $\lg n \equiv \log_2 n$

Time Complexity Vs. Problem's Sizes

The largest data sizes n that can be processed by an algorithm with time complexity $f(n)$ provided that $T(10) = 1$ *minute*:

$f(n)$	**Length of time to run an algorithm**					
	1 *min*	1 *hour*	1 *day*	1 *week*	1 *year*	1 *decade*
n	10	600	$14,400$	$100,800$	$5.26 \cdot 10^6$	$5.26 \cdot 10^7$
$n \lg n$	10	250	$3,997$	$23,100$	$883,895$	$7.64 \cdot 10^6$
$n^{1.5}$	10	153	$1,275$	$4,666$	$65,128$	$302,409$
n^2	10	77	379	$1,003$	$7,249$	$22,932$
n^3	10	39	112	216	807	$1,738$
2^n	10	15	20	23	29	32

Practicable algorithms in most applications: up to linearithmic, $n \log n$, complexity.

Time Complexity Vs. Problem's Sizes

The largest data sizes n that can be processed by an algorithm with time complexity $f(n)$ provided that $T(10) = 1$ *microsecond*:

$f(n)$	**Length of time to run an algorithm**					
	1 *sec*	1*min*	1 *hour*	1 *day*	1 *year*	1 *decade*
n	$1.0 \cdot 10^7$	$6.0 \cdot 10^8$	$3.6 \cdot 10^{10}$	$8.6 \cdot 10^{11}$	$3.2 \cdot 10^{14}$	$3.2 \cdot 10^{15}$
$n \lg n$	$1.6 \cdot 10^6$	$7.6 \cdot 10^7$	$3.8 \cdot 10^9$	$7.9 \cdot 10^{10}$	$2.4 \cdot 10^{13}$	$2.2 \cdot 10^{14}$
$n^{1.5}$	$1.0 \cdot 10^5$	$1.5 \cdot 10^6$	$2.3 \cdot 10^7$	$2.0 \cdot 10^8$	$1.0 \cdot 10^{10}$	$4.6 \cdot 10^{10}$
n^2	10000	77460	$6.0 \cdot 10^5$	$2.9 \cdot 10^6$	$5.6 \cdot 10^7$	$1.8 \cdot 10^8$
n^3	1000	3915	15326	$4.4 \cdot 10^4$	$3.2 \cdot 10^5$	$6.8 \cdot 10^5$
2^n	30	36	42	46	55	58

Practicable algorithms: in most applications – up to linearithmic, $n \log n$, complexity.

Check the hidden constants!

Order relations can be drastically misleading.

$g(n) \in O(f(n))$ means that $g(n) \le cf(n)$ for all $n > n_0$ where

- c is the fixed amount of computations per data item and
- n_0 is the lowest data size, after which the relation holds.

Check whether the constant c is sufficiently small.

Practical rule:

Roughly estimate the computation volume per data item after time complexities of the algorithms are compared in a "Big-Oh" sense.

- "Big-Oh": The linear algorithm A is better than the linearithmic one B.
- Estimated computation volumes: $g_{\mathrm{A}}(n) = 4n$ and $g_{\mathrm{B}} = 0.1n \lg n$.
- A outperforms B: $4n < 0.1n \lg n$, or $\lg n > 40$, i.e. $n > 2^{40} \approx 10^{12}$ items.
- Therefore, in practice the algorithm B is mostly better.

Check the hidden constants!

Order relations can be drastically misleading.

$g(n) \in O(f(n))$ means that $g(n) \leq cf(n)$ for all $n > n_0$ where

- c is the fixed amount of computations per data item and
- n_0 is the lowest data size, after which the relation holds.

Check whether the constant c is sufficiently small.

Practical rule:

Roughly estimate the computation volume per data item after time complexities of the algorithms are compared in a "Big-Oh" sense.

- "Big-Oh": The linear algorithm A is better than the linearithmic one B.
- Estimated computation volumes: $g_{\mathrm{A}}(n) = 4n$ and $g_{\mathrm{B}} = 0.1n \lg n$.
- A outperforms B: $4n < 0.1n \lg n$, or $\lg n > 40$, i.e. $n > 2^{40} \approx 10^{12}$ items.
- Therefore, in practice the algorithm B is mostly better.

Check the hidden constants!

Order relations can be drastically misleading.

$g(n) \in O(f(n))$ means that $g(n) \le cf(n)$ for all $n > n_0$ where

- c is the fixed amount of computations per data item and
- n_0 is the lowest data size, after which the relation holds.

Check whether the constant c is sufficiently small.

Practical rule:

Roughly estimate the computation volume per data item after time complexities of the algorithms are compared in a "Big-Oh" sense.

- "Big-Oh": The linear algorithm A is better than the linearithmic one B.
- Estimated computation volumes: $g_{\mathrm{A}}(n) = 4n$ and $g_{\mathrm{B}} = 0.1n \lg n$.
- A outperforms B: $4n < 0.1n \lg n$, or $\lg n > 40$, i.e. $n > 2^{40} \approx 10^{12}$ items.
- Therefore, in practice the algorithm B is mostly better.

Asymptotic Bounds on Running Time

Asymptotic notation measures the running time of the algorithm in terms of elementary operations, i.e. asymptotic bounds are independent of inputs and implementation:

$$\begin{cases} \text{Upper bound "Big-Oh":} & g(n) \in O(f(n)) \rightarrow g(n) \leq cf(n) \\ \text{Lower bound "Big-Omega":} & g(n) \in \Omega(f(n)) \rightarrow g(n) \geq cf(n) \\ \text{Tight bound "Big-Theta":} & g(n) \in \Theta(f(n)) \rightarrow c_1 f(n) \geq g(n) \geq c_2 f(n) \end{cases}$$

In general, **the running time varies not only according to the size of the input, but the input itself.**

- Some sorting algorithms take almost no time if the input is already sorted in the desired order, but much longer if it is not.

The most common performance measures for all inputs:
the **worst-case running time** and the **average-case running time**.

The Worst-case Running Time

Advantages:

- An upper bound of the worst-case running time is usually fairly easy to find.
- It is essential for so-called "mission-critical" applications

Drawbacks:

- It may be too pessimistic: actually encountered inputs may lead to much lower running times that the "upper bound".
  - The most popular fast sorting algorithm, `quicksort`, is $\Theta(n^2)$ in the worst case, but $\Theta(n \log n)$ for "random" inputs, being most frequent in practice.
- The worst-case input might be unlikely to be met in practice.
- In many cases it is difficult to specify the worst-case input.

The Average-case Running Time

By contrast, the average-case time is not as easy to define.

- A probability distribution on the inputs has to be specified.
  - Simple assumption: all equally likely inputs of size n.
  - Sometimes this assumption may not hold for the real inputs.
- The analysis might be a difficult math challenge.
- Even if the average-case complexity is good, the worst-case one may be very bad.

Asymptotic optimality of an algorithm with the running time $T(n)$:

It is **asymptotically optimal** for the given problem if
(i) $t(n) \in O(f(n))$ for some function f **AND**
(ii) there is no algorithm with running time $g(n)$ for any function g that grows more slowly than f when $n \to \infty$.

The Average-case Running Time

By contrast, the average-case time is not as easy to define.

- A probability distribution on the inputs has to be specified.
  - Simple assumption: all equally likely inputs of size n.
  - Sometimes this assumption may not hold for the real inputs.
- The analysis might be a difficult math challenge.
- Even if the average-case complexity is good, the worst-case one may be very bad.

Asymptotic optimality of an algorithm with the running time $T(n)$:

It is **asymptotically optimal** for the given problem if
(*i*) $t(n) \in O(f(n))$ for some function f **AND**
(*ii*) there is no algorithm with running time $g(n)$ for any function g that grows more slowly than f when $n \to \infty$.