# CS 4310 Algorithms: I. Introduction to Algorithm Efficiency

E. de Doncker[1]

[1] Dept. of Computer Science, WMU, U.S.A.

January 2024

# Outline

1. Course Description
2. Performance analysis
   - Frequency counts
   - Order of complexity
   - Practical considerations

# Outline


1. Course Description
2. Performance analysis
   - Frequency counts
   - Order of complexity
   - Practical considerations

## Catalog description

This course is a continuation of the study of data structures and algorithms, emphasizing methods useful in practice. It provides a theoretical foundation in designing algorithms as well as their efficient implementations. The focus is on the advanced analysis of algorithms and on how the selections of different data structures affect the performance of algorithms. Topics covered include: sorting, search trees, heaps, and hashing; divide-and-conquer; dynamic programming; backtracking; branch-and-bound; amortized analysis; graph algorithms; shortest paths; network flow; computational geometry; number-theoretic algorithms; polynomial and matrix calculations; and parallel computing. It comprises four hours of lecture and recitation experience every week.

## Course description

### Main course sections: Algorithm paradigms and analysis

- Performance analysis
- Divide & conquer
- Recurrence relations
- Greedy methods
- Dynamic programming
- Backtracking
- Branch & bound
- P, NP, NP-hard, NP-complete problem classes

Spring 2024 TR class 11:30 - 12:45)

Course web page: http://www.cs.wmich.edu/elise

## Frequency counts

The execution time of a program depends on many factors, such as processor speed, the compiler used, and of course the amount of work done in the program for given data. We introduce the frequency count of a program statement to indicate the number of times the statement is executed as a function of a program parameter, say $n$. Consider the program section below, with its frequency counts on the right.

| *line* | *Program* | *Freq. cnt.* |
|---|---|---|
| 1 | $x = 0;$ | 1 |
| 2 | **for** $( i = 0;\ i < n;\ i{+}{+} )$ | $n + 1$ |
| 3 | $\quad x = x + 1;$ | $n$ |

The statement at line 1 is executed once. Line 2 for the **for** loop is executed $n + 1$ times, including the last time where $i = n$ and the test $i < n$ fails. Line 3 for the statement in the body of the loop is executed for each iteration, thus $n$ times.

The total frequency count of the program section equals $2n + 2$, which is linear in $n$.

## Frequency counts

A second program section:

| *line* | *Program* | *Freq. cnt.* |
|---|---|---|
| 1 | $x = 0;$ | 1 |
| 2 | $y = 0;$ | 1 |
| 3 | **for** $(i = 0;\ i < n;\ i{+}{+})\ \{$ | $n + 1$ |
| 4 | $\quad x = x + 1;$ | $n$ |
| 5 | $\quad$ **for** $(j = 0;\ j < n;\ j{+}{+})$ | $n(n + 1)$ |
| 6 | $\qquad y = y + 1;$ | $n^2$ |
| 7 | $\}$ | |

The first four lines are similar to those of the previous program section. The second **for** loop statement at line 5 is executed $(n + 1)$ times for each value of $i$, thus a total of $n(n + 1)$ times. Line 6 is inside the second loop and thus executed $n^2$ times.
The total frequency count is the sum of the individual counts, $2n^2 + 3n + 3$.

This is quadratic in $n$.

## Frequency counts

A third program section:

| *line* | *Program* | *Freq. cnt.* |
|---|---|---|
| 1 | $x = 0;$ | $1$ |
| 2 | **for** $(i = 1;\ i <= n;\ i{+}{+})$ | $n + 1$ |
| 3 |     **for** $(j = 1;\ j <= i;\ j{+}{+})$ | $\sum_{i=1}^{n}(i+1)$ |
| 4 |         $x = x + 1;$ | $\sum_{i=1}^{n} i$ |

The first two lines are similar to before. For simplicity the loop iterations start at 1. Line 3 is executed $(i + 1)$ times for iteration $i$ of the outer loop, so the total for line 3 is the sum of this for all iterations ($i = 1, \ldots, n$) of the outer loop, $\sum_{i=1}^{n}(i + 1)$. Similarly, line 4 is executed $i$ times for iteration $i$ of the outer loop, so the total for line 4 is the sum of this for all iterations ($i = 1, \ldots, n$) of the outer loop, $\sum_{i=1}^{n} i$. On the next slide we will digress with a proof by induction showing that the total for line 4 is then

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{n\,(n+1)}{2}.$$

We can easily get line 3 from this, since $\sum_{i=1}^{n}(i + 1) = \sum_{i=1}^{n} i + \sum_{i=1}^{n} 1 = \frac{n\,(n+1)}{2} + n$. The overall total requires some arithmetic; let's just say that it is quadratic in $n$.

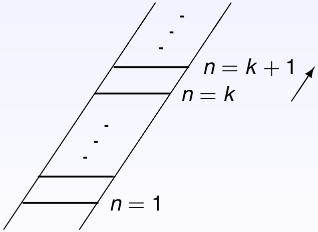
Many other examples for frequency counts are given in the text by Horowitz et al. [1],

Section 1.3.2 (Time complexity).

## Proof by induction

Prove by induction on $n$: $\sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{n\,(n+1)}{2}$, for $n \geq 1$.

The induction principle for this proof can be compared to proving that we can climb a ladder (from a beginning step $n_0$) up to any step $n$. Since the above claim is for $n = n_0 = 1$, we first show that we can get onto the first step (the *basis*). Then it suffices to show that, assuming we have reached step $n = k$ (induction hypothesis), it will be possible to get from $k$ to $k + 1$ (induction step).



## Proof by induction

*Proof:*
*Basis*: $n = 1$. In the summation we have $\sum_{i=1}^{1} i = 1$, and the right hand side is $\frac{n\,(n+1)}{2} = \frac{1\,(2)}{2} = 1$. This establishes the basis.
*Induction hypothesis*: Assume the property is true for $n = k$ :
$\sum_{i=1}^{k} i = 1 + 2 + \ldots + k = \frac{k\,(k+1)}{2}$.
*Induction step*: Show that the property is valid for $n = k + 1$. The left hand side is

$$\sum_{i=1}^{k+1} i = 1 + 2 + \ldots + k + (k+1)$$

$$= \frac{k\,(k+1)}{2} + \ (k+1) \quad \text{(in view of the induction hypothesis)}$$

$$= (k+1)\,(\frac{k}{2} + 1) = (k+1)\,(\frac{k+2}{2})$$

This establishes the property for $n = k + 1$.

## Order

For the frequency count examples in the previous section, we obtained linear and quadratic functions for the total count. Fig. [Order] shows the functions $\log x, x, x\log x, x^2, x^3$ and $e^x$ as a function of $x$, illustrating various classes of growth rate in increasing order.

Observe that the order emerges only at large enough $x$. For example, we have $4^3 > e^4$ but $5^3 < e^5$, i.e., the function graphs intersect between $x = 4$ and $x = 5$, but after the intersection point, $x^3$ remains below $e^x$. This is an example of an asymptotic relationship expressed as $x^3 = \mathcal{O}(e^x)$ (" $x^3$ is big oh of $e^x$ "; in this sense, $e^x$ is an upper bound for $x^3$), or $e^x = \Omega(x^3)$ (" $e^x$ is omega of $x^3$ "; in this sense $x^3$ is a lower bound for $e^x$).

Formal definitions of $\mathcal{O}, \Omega$ and $\Theta$ (adhering to the notation of Horowitz et al. in [1], Section 1.3.2 (Asymptotic notation)), are given next for non-negative function $f(n)$ and $g(n)$. It should be noted that the book by Neapolitan [2] formulates these as function classes, with corresponding definitions, e.g., $f(n) \in \mathcal{O}(g(n))$.

## Order

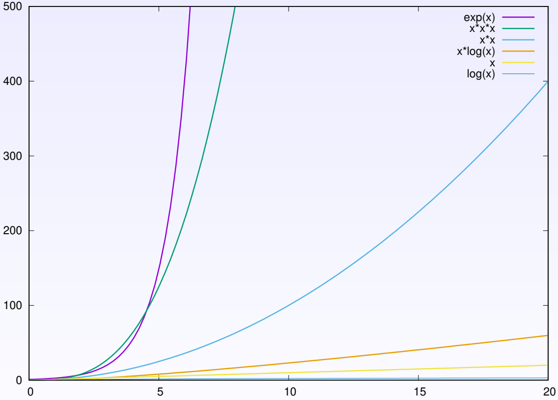



Figure: [Order] functions $\log x, x, x \log x, x^2, x^3, e^x$

## $\mathcal{O}$ order

### Definition

For non-negative functions $f(n)$ and $g(n)$, $f(n) = \mathcal{O}(g(n))$ iff there are positive constants $c$ and $n_0$ such that $f(n) \leq c\,g(n)$ for all $n \geq n_0$.

The following examples prove $\mathcal{O}$ relationships by using the definition. It suffices to find pairs of constants $c$ and $n_0$, and these are not unique. Many more examples are given in the texts [1, 2].

*(i)* $1000n + 2 = \mathcal{O}(n)$ since $1000n + 2 \leq 1001n$ for $n \geq 2$,
or $1000n + 2 \leq 1002n$ for $n \geq 1$.

*(ii)* $6(2^n) + n^2 = \mathcal{O}(2^n)$ since $6(2^n) + n^2 \leq 7(2^n)$ for $n \geq 4$.

*(iii)* $2n + 1 = \mathcal{O}(n^2)$ since $2n + 1 \leq 2n^2$ for $n \geq 2$,
or $2n + 1 \leq 3n^2$ for $n \geq 1$, or $2n + 1 \leq n^2$ for $n \geq 3$.

*(iv)* $2n + 1 \neq \mathcal{O}(1)$ (constant); $2^n \neq \mathcal{O}(n^2)$; $3^n \neq \mathcal{O}(2^n)$.

## $\Omega$ Order

### Definition

For non-negative functions $f(n)$ and $g(n)$, $f(n) = \Omega(g(n))$ iff there are positive constants $c$ and $n_0$ such that $f(n) \geq c\, g(n)$ for all $n \geq n_0$.

We can express the $\mathcal{O}$ examples in terms of $\Omega$ order, as $u(n) = \mathcal{O}(v(n))$ implies $v(n) = \Omega(u(n))$.
*(i)* $n = \Omega(1000n + 2)$ since $n \geq \frac{1}{1002}(1000n + 2)$ for all $n \geq 1$.
*(ii)* $2^n = \Omega(6(2^n) + n^2)$ since $2^n \geq \frac{1}{7}(6(2^n) + n^2)$ for $n \geq 4$.
*(iii)* $n^2 = \Omega(2n + 1)$ since $n^2 \geq \frac{1}{2}(2n + 1)$ for $n \geq 2$.
*(iv)* $n^2 \neq \Omega(2^n)$; $2^n \neq \Omega(3^n)$; $n^2 \neq \Omega(n^3)$.

For many other examples see the texts [1, 2].

## $\Theta$ Order

We opt for a definition that uses $\mathcal{O}$ and $\Omega$ order.

**Definition**

For non-negative functions $f(n)$ and $g(n)$, $f(n) = \Theta(g(n))$ iff both $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$.

In this case, $f(n)$ and $g(n)$ are of exactly the same order. Examples where we established $\mathcal{O}$ and $\Omega$ order:
*(i)* $1000n + 2 = \Theta(n)$
*(ii)* $6(2^n) + n^2 = \Theta(2^n)$

The properties stated in Theorem [Ratios] below may help determining the asymptotic relationship between functions.

## Limit of ratio

**Theorem [Ratios]**

For non-negative functions $f(n)$ and $g(n)$,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \implies f(n) = \mathcal{O}(g(n)) \text{ and } f(n) \neq \Theta(g(n)) \quad (1)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \implies f(n) = \Omega(g(n)) \text{ and } f(n) \neq \Theta(g(n)) \quad (2)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \text{constant} > 0 \implies f(n) = \Theta(g(n)) \quad (3)$$

Note that the implications are only in one direction (left to right).

*Examples*:

$\lim_{n\to\infty} \frac{6(2^n)+n^2}{2^n} = \lim_{n\to\infty} \frac{6(2^n)}{2^n} = 6 \implies 6(2^n) + n^2 = \Theta(2^n)$ in view of (3)

$\lim_{n\to\infty} \frac{2n+1}{n^2} = \lim_{n\to\infty} \frac{2n}{n^2} = \lim_{n\to\infty} \frac{2}{n} = 0$

$\implies 2n + 1 = \mathcal{O}(n^2)$ and not $\Theta(n^2)$ in view of (1)

## Limit of ratio

*Examples*:

– Let $f(n) = \log n, \quad g(n) = \sqrt{n}$

$$\lim_{n\to\infty} \frac{\log n}{n^{1/2}} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)} \quad (l'H\hat{o}pital's\ rule)$$

$$= \lim_{n\to\infty} \frac{1/n}{(1/2)\, n^{-1/2}} = lim_{n\to\infty} \frac{2}{n^{1/2}} = 0$$

$$\Longrightarrow \quad \log n = \mathcal{O}(\sqrt{n}) \ \text{ and not } \ \Theta(\sqrt{n})$$

– Is $n^2 + n = \Omega(n^2)$ ? The answer is Yes.

$\lim_{n\to\infty} \frac{n^2+n}{n^2} = 1 \quad \Longrightarrow \quad n^2 + n = \Theta(n^2) \quad \Longrightarrow \quad n^2 + n = \Omega(n^2)$

– Note that the log function in different bases differs only in a multiplicative factor,

$x = \log_a(n) \quad \Longrightarrow \quad a^x = n \quad \Longrightarrow \quad \log_b a^x = \log_b n \quad \Longrightarrow \quad x \log_b a = \log_b n$

$\Longrightarrow \quad \log_b n = (\log_b a) \log_a n \quad \Longrightarrow \quad \log_b n = \Theta(\log_a n)$.

For example, $\lg n = \Theta(\log n)$, where $\log n = \log_e n$ (Naperian logarithm),

and $\lg n = \log_2 n$, thus $\lg n = \Theta(\log n)$.

## Limit of ratio

The implications (1) and (2) on the right of Theorem [Ratios] are abbreviated by the asymptotic notations $f(n) = o(g(n))$ and $f(n) = \omega(g(n))$.

### Definition

$$f(n) = o(g(n)) \text{ iff } f(n) = \mathcal{O}(g(n)) \text{ and } f(n) \neq \Theta(g(n))$$
$$f(n) = \omega(g(n)) \text{ iff } f(n) = \Omega(g(n)) \text{ and } f(n) \neq \Theta(g(n))$$

## Practical considerations

See Section 1.3.5 of [1].
– The time complexity can be used to guide your choice of algorithm for a certain task if you have multiple algorithms available with different complexities. But note the following: Say, program $P_1$ implementing a first algorithm has time complexity $\mathcal{O}(n)$ and program $P_2$ uses an algorithm of complexity $\mathcal{O}(n^2)$. However, the orders hold for large enough $n$. Suppose we know that $P_1$ runs in time $\leq c_1 n = 10^6 n$ microseconds, and $P_2$ in time $\leq c_2 n^2 = n^2$ microseconds, and all we need is $n \leq 1000$. Then the running times for $P_1$ and $P_2$ are at most 1000 seconds and one second, respectively. So the winner is $P_2$ in this case.

– To get the elapsed time of a program section, insert a call to your timing function (such as gettimeofday() in C) just before and after the program section (see the example program time.c at http://www.cs.wmich.edu/elise/courses/cs531/time.c Make sure that nothing is timed that shouldn't be (such as I/O)!

## Practical considerations

– When timing program execution, take the clock resolution into account. Otherwise, short instances may not be timed with enough accuracy so that the timing returns zero. To time short instances, put the program section in a loop and get the total time for the loop, then take the average per iteration.
– Generate suitable data for timing executions according to the problem at hand. For example, you may have to provide data that show the worst case time of a quicksort ($\mathcal{O}(n^2)$ for an array of length $n$), which is not hard to do.

– Addressing actual average case time in practice may, however, not be feasible. For a sort, the average would have to be taken over all possible ($n!$) permutations. In practice we generate a number of data sets at random to execute the algorithm, and average over the obtained execution times.

## Bibliography

S. Sahni E. Horowitz and S. Rajasekeran.
*Computer Algorithms/C++*.
Computer Science Press, 2nd edition, 1998.
ISBN 0-7167-8315-0.

R. E. Neapolitan.
*Foundations of Algorithms*.
Jones and Bartlett, 5th edition, 2015.
ISBN 978-1-284-04919-0.