CS-203 – LESSON 02 – ALGORITHM EFFICIENCY ANALYSIS

GIUSEPPE TURINI

Kettering UNIVERSITY

TABLE OF CONTENTS

Introduction to Algorithm Analysis

Algorithm Analysis Framework

- Input Size, Running Time, and Orders of Growth
- Worst-Case, Best-Case, and Average-Case Efficiencies

Asymptotic Notations and Basic Efficiency Classes

- Big O, Big Omega, and Big Theta Notations, and Properties of Asymptotic Notations
- Limits to Compare Orders of Growth, and Basic Efficiency Classes

Theoretical and Empirical Analysis of Algorithms

- Theoretical Analysis of Non-Recursive and Recursive Algorithms
- Empirical Analysis of Algorithms

Algorithm Visualization

STUDY GUIDE

Study Material:

- These slides.
- Your notes.
- * "Introduction to the Design and Analysis of Algorithms (3rd Ed.)", chap. 2, pp. 41-95.

Practice Exercises:

- Exercises from *: 2.1.1-5, 2.1.8-10, 2.2.1-7, 2.2.9-12, 2.3.1-6, 2.3.9, 2.3.11-12, 2.4.1-5, 2.4.7-12, 2.5.4, 2.5.6-9, 2.6.1-4.

Additional Resources:

- "Introduction to Algorithms (3rd Ed.)", chap. 1, pp. 43-64.

INTRODUCTION TO ALGORITHM ANALYSIS

Algorithm Analysis: An investigation of the efficiency of an algorithm, usually with respect to one of these two resources:

- The running time.
- The memory space.

This emphasis on efficiency is easy to explain.

- **Efficiency can be studied in precise quantitative terms** (other characteristics like simplicity and generality cannot).
- **Efficiency considerations are very important in practice.**

ALGORITHM ANALYSIS FRAMEWORK

The following is an outline of a general framework to analyze algorithm efficiency.

Time efficiency: Also called time complexity, it indicates how fast a given algorithm runs.

Space efficiency: Also called space complexity, it refers to the amount of memory required by a given algorithm (in addition to the memory needed for its input and output).

From now on, **we will focus mainly on the analysis of time efficiency**, for two reasons.

- Nowadays, time efficiency is usually more important than space efficiency.
- The methods to analyze time efficiency can be adjusted to evaluate space efficiency.

THE INPUT SIZE

Almost all algorithms run longer on larger inputs. Therefore, it is logical to investigate the algorithm efficiency as a function of some parameters indicating the algorithm input size.

Input Size: The parameters (associated with the input) that control the amount of work a given algorithm does.

Note: Some algorithms require one or more parameters to indicate their input size.

Note: In some cases, the input size is the magnitude of the input. In such situations, you can use the bit-size of the binary representation of the input (n).

$$\text{number of bits in the binary reprentation of } n = b = \lfloor \log_2 n \rfloor + 1$$

THE RUNNING TIME

To measure the running time of a program implementing a given algorithm, we can simply use some standard unit of time measurement (e.g., seconds, milliseconds, etc.).
However, this approach has some drawbacks:

- Dependence on hardware performance (e.g., computer speed, etc.).
- Dependence on implementation quality (e.g., programmer, language, compiler, etc.).
- Difficulty of timing/clocking the actual running time (e.g., time queries accuracy, etc.).

We would like an algorithm efficiency metric that does **not** depend on these factors.

A possible approach is to count the number of times each algorithm operation is executed.
However, this is both excessively difficult, and usually unnecessary.

THE BASIC OPERATION

The correct way is to identify the most important algorithm operation: the basic operation.

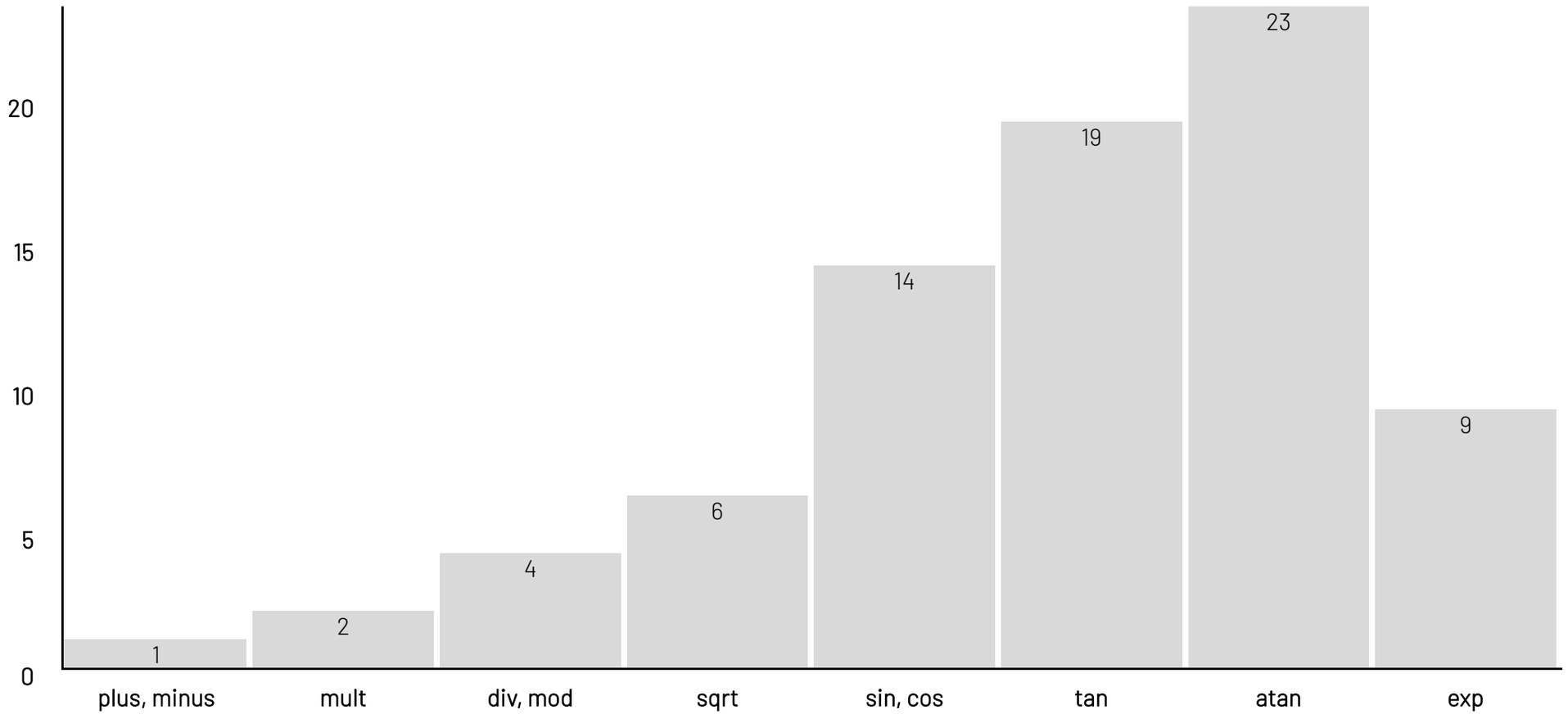
Basic Operation: The operation that contributes the most to the total running time of a given algorithm. Usually, the most time-consuming* operation in the innermost loop of the algorithm.

Note: The basic operation must be an operation with an almost-constant execution time (so, it can be a function, but it cannot be a function with a variable-size iteration etc.).

Once selected, we just compute the number of times the basic operation is executed.

THE BASIC OPERATION 2

* EXAMPLE OF APPROX-RELATIVE COST OF FLOATING-POINT OPERATIONS IN C++



Note: Operations mult, div, mod by powers of 2, and bit-wise operators (e.g., shift, not, etc.) have a very low computational cost.

INPUT SIZE AND BASIC OPERATION EXAMPLES

PROBLEM	INPUT SIZE MEASURE	BASIC OPERATION
Searching for item in list	Number of items in list	Comparison of 2 items
Multiplication of 2 matrices	Matrix dimensions	Multiplication of 2 numbers
Checking primality of number	Magnitude/bit-size of number	Division of 2 numbers
Typical graph problem	Number of vertices and/or edges	Visit/traverse a vertex/edge

COUNT OF BASIC OPERATIONS

So, to measure the time efficiency of an algorithm we have to count the number of times the basic operation is executed depending on the input size.

Example: Consider an implementation of an algorithm running on a specific hardware.

Let c_{op} be the execution time of the basic operation of that algorithm on that HW. Let **n** be the input size. Let **C(n)** be the count of number of times the basic operation is executed depending on the input size (n).

Then, we can approximate the running time **T(n)** of a program implementing this algorithm on that hardware by the formula:

$$T(n) \approx c_{op} \times C(n)$$

Of course, this formula should be used with caution.

COUNT OF BASIC OPERATIONS 2

$$\mathrm{T}(n) \approx c_{op} \times \mathrm{C}(n)$$

In the formula above, the count **C(n)** does not consider non-basic operations, and in most cases the count itself is computed approximatively.

Further, the constant $\mathbf{c_{op}}$ is also an approximation.

Still, unless **n** is extremely large or very small, the formula above can give us a reasonable estimate of the running time of the implementation of an algorithm on a specific hardware.

Note: When n is extremely large, the approximations in computing C(n) can be significant, and the input could be so big to require special handling (see out-of-core techniques).

Note: When n is very small, the approximations in computing C(n) can be significant.

COUNT OF BASIC OPERATIONS 3

$$\mathrm{T}(n) \approx c_{op} \times \mathrm{C}(n)$$

The formula above makes it possible to answer useful questions.

Question: How much faster would this algorithm run on a hardware that is 10 times faster?

Answer: The algorithm would run 10 times faster on a hardware that is 10 times faster.

Question: Assuming C(n) = n (n-1) / 2,
how much longer will the algorithm run if we double its input size?

Answer: About 4 times longer.
C(2n) = 2n (2n-1) / 2 ≈ 4 [n (n-1) / 2] = 4 C(n).

ORDER OF GROWTH

Note: In answering the last question, both the value of c_{op} and the multiplicative constant (1/2) were neglectable, because they cancel out computing the ratio T(2n) / T(n).

This is why, in analyzing efficiency, we ignore multiplicative constants and concentrates on: the order of growth of the count of basic operations to within a multiplicative constant for large-size inputs.

Question: Why this emphasis on the order of growth of count C(n) for large input sizes?

Answer: A difference in running times on small input sizes is not significant in distinguishing efficient algorithms from inefficient ones. For large input sizes, it is the order of growth of the function/count C(n) that counts!

ORDER OF GROWTH 2

These examples of simple functions show the differences among orders of growth.

Table 1. Example of commonly used functions in algorithm analysis.

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

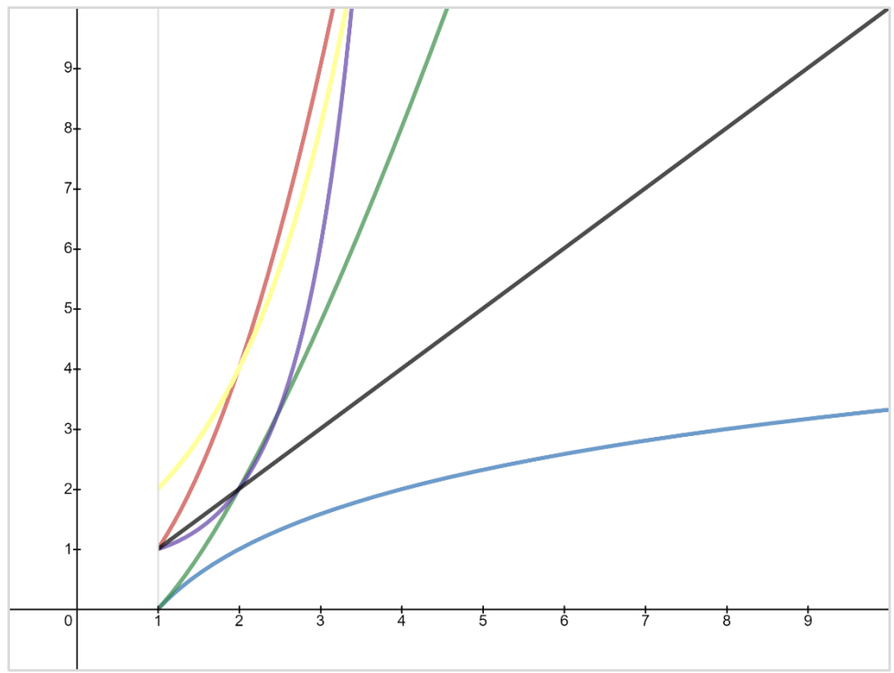


Figure 1. Most common functions in analyzing orders of growth.

Note: These table/graph (Table 1 and Figure 1) are very important for algorithm analysis.

ORDER OF GROWTH 3

In the previous table/graph:

The slowest-growing function is the logarithmic function ($\log_2(n)$). An implementation with a logarithmic basic-operation count runs instantaneously on inputs of all realistic sizes.
Also, a logarithm property ($\log_a(x) = \log_a(b) \times \log_b(x)$) make it possible to change the logarithm base by multiplying by a specific constant, so we usually omit the logarithm base whenever we are interested only in the order of growth of a function.

The fastest-growing functions are the exponential function (2^n) and the factorial function ($n!$).
Their values become very large even for small input sizes.
Both these functions are referred to as "exponential-growth functions".
Algorithms with exponential-growth basic-operation counts are practical for solving only problems with very small input sizes.

ORDER OF GROWTH 4

In the previous table/graph:

Question: How these functions react to a twofold (×2) increase in the input size?

Answer: The function ($\log_2(n)$) increases by just 1: $\log_2(2n) = \log_2(2) + \log_2(n) = 1 + \log_2(n)$

The **linear** function (n) obviously increases twofold (×2).

The **linearithmic** function ($n \log_2(n)$) increases slightly more than twofold (×2).
$2n \log_2(2n) = 2n [\ \log_2(2) + \log_2(n)\ ] = 2n [\ 1 + \log_2(n)\ ] = 2n + 2 [\ n \log_2(n)\ ]$

The **quadratic** (n^2), and **cubic** (n^3) functions increase (×4) and (×8), respectively.

The **exponential** function (2^n) gets squared: $2^{2n} = (2^n)^2$

The **factorial** function ($n!$) gets a lot bigger than squared.
$(2n)! = n! \times (n+1) \times (n+2) \times \ldots \times (2n) > n! \times n^n > (n!)^2$

ANALYSIS EXAMPLE: SUM OF CUBICS

Problem: Compute the cubic of each item in a container, and return their sum.

Sum of Cubics Algorithm

```
int SumOfCubics( int[] A ) {
   int sum = 0;
   for( int i = 0; i < A.length; i++ ) {
      sum = sum + ( A[i] * A[i] * A[i] );
   }
   return sum;
}
```

Note: The input size is the number of items in the input container (let's call it, n), because the longer the container the more work this algorithm has to do.

Note: The basic operation is the multiplication, because is the most expensive operation among the operations executed the most, and it is also the most "meaningful".

Note: The basic operation count (the multiplications performed) is: $C(n) = \sum_{i=0}^{n-1} 2 = 2n$

WORST-CASE, BEST-CASE, AND AVERAGE-CASE EFFICIENCIES

In some algorithms the running time depends on the input size and on the specs of an input.

In these cases, it is not possible to determine a single formula for the basic operation count.

So, we have to extend our analysis identifying specific scenarios:

- **The worst-case:** when the algorithm does the max amount of work.
- **The best-case:** when the algorithm does the min amount of work.
- **The average-case:** computing the average amount of work done by the algorithm.

These specific analyzes allow us to understand the behavior of the basic operation count, even if we cannot determine a formula for it.

Example: If we cannot determine a formula for the count C(n), we could still be able to delimit the count (e.g., $3n < C(n) < n^2$) or approximate its average (e.g., $C(n) \approx 0.1\, n^2$).

WORST-CASE EFFICIENCY

Worst-Case Efficiency: The algorithm efficiency for the inputs of a specific size (n), for which the algorithm does the max amount of work (runs the longest).

To determine the worst-case efficiency $C_{worst}(n)$ of an algorithm:

Step 1: Find the inputs yielding the max count C(n) among all possible inputs of size n.

Step 2: Compute the count in the worst-case: $C_{worst}(n)$.

Note: The worst-case analysis provides very important information about the efficiency of an algorithm by providing an upper-bound to the basic operation count.

This guarantees that for any input of size n, the count will never exceed $C_{worst}(n)$.

BEST-CASE EFFICIENCY

Best-Case Efficiency: The algorithm efficiency for the inputs of a specific size (n), for which the algorithm does the min amount of work (runs the fastest).

To determine the best-case efficiency $C_{best}(n)$ of an algorithm:

Step 1: Find the inputs yielding the min count C(n) among all possible inputs of size n.

Step 2: Compute the count in the best-case: $C_{best}(n)$.

Note: The best-case analysis is not as important as the worst-case analysis, because it provides a lower-bound to the basic operation count.

This guarantees that for any input of size n, the count will never be lower than $C_{best}(n)$.

Sometimes the best-case efficiency can be used to design new algorithms (e.g., Insertion Sort), or to discard an algorithm if its best-case efficiency is unsatisfactory.

AVERAGE-CASE EFFICIENCY

Average-Case Efficiency: The algorithm efficiency for a "typical/random" input of a specific size (n). The average-case analysis is considerably more difficult than the worst-case and best-case analyses.

Note: The average-case efficiency is not the average of the worst-case and best-case.

$$C_{avg}(n) \neq \frac{C_{worst}(n) + C_{best}(n)}{2}$$

To determine the average-case efficiency $C_{avg}(n)$, we must make some assumptions:

- **Step 1:** Find all possible values of the count C(n) among all possible inputs of size n.
- **Step 2:** Associate each value of C(n) with a "case" (e.g., successful search at index 0).
- **Step 3:** Assume a probability distribution for all the "cases" (e.g., 50% failed search).
- **Step 4:** Compute the weighted average of the counts for all cases/percentages: $C_{avg}(n)$.

ANALYSIS EXAMPLE: SEQUENTIAL SEARCH

In some algorithms the running time depends on the input size and on the specs of an input.

Problem: Search for a given key in a container by searching its items sequentially. If successful, return the index where the key was found, otherwise return an error.

Sequential Search Algorithm

```
int SequentialSearch( int k, int[] A ) {
   int i = 0;
   for( ; ( ( i < A.length ) && ( A[i] != k ) ); i++ ) {}
   if( i < A.length ) { return i; }
   else { return -1; }
}
```

Note: The input size is the number of items in the input container, because the longer the container the more work the Sequential Search has to do, potentially.

Note: The basic operation is the comparison (A[i] != k), because is the most-expensive and most-executed operation, and it is also the most "meaningful".

ANALYSIS EXAMPLE: SEQUENTIAL SEARCH 2

The word done by the sequential search varies for different containers of the same size, because it depends also on the container content. So, we cannot determine a formula for the basic operation count C(n), then we need worst-case, best-case, and average-case analyses.

Worst-Case: The Sequential Search does the max amount of work when:

- The search is unsuccessful (search key not found, return -1).

  OR

- The search is successful at the last index in the container (return n-1).

In both these cases, the Sequential Search makes the max number of comparisons. So, considering an input of size n, the worst-case count is:

$$C_{worst}(n) = n$$

ANALYSIS EXAMPLE: SEQUENTIAL SEARCH 3

Best-Case: The Sequential Search does the min amount of work when:

- The input container is empty (return -1).

  OR

- The search is successful at the first index in the container (return 0).

In both these cases, the Sequential Search makes the min number of comparisons. So, considering an input of size n, the best-case count is:

$$C_{best}(n) = 1$$

ANALYSIS EXAMPLE: SEQUENTIAL SEARCH 4

Average-Case: To compute the average-case efficiency we need to assume a probability distribution for all the possible "cases". For example:

- The probability of a successful search is p ($0 \leq p \leq 1$), and the probability of a failed search is 1-p.

  AND

- The probability of a successful search at index i, is the same for every i ($p_0 = p_1 = \ldots = p_{n-1} = p/n$).

$$C_{avg}(n) = \left[1\frac{p}{n} + \cdots i\frac{p}{n} \cdots + n\frac{p}{n}\right] + n(1-p) = \frac{p(n+1)}{2} + n(1-p)$$

This formula for $C_{avg}(n)$ yields some quite reasonable answers:

- If p=1 (always successful search): $C_{avg}(n) = (n+1)/2$.
- If p=0 (always failed search): $C_{avg}(n) = n$.

ANALYSIS EXAMPLE: SEQUENTIAL SEARCH 5

In analyzing the efficiency of Sequential Search, we cannot determine a formula for the basic operation count C(n). However, the worst-case, best-case, and average-case analyses yield:

- $C_{best}(n) = 1 \leq C(n) \leq n = C_{worst}(n)$.
- Given a probability distribution for successful/failed searches, we can compute $C_{avg}(n)$ (e.g., for p=0.9 and equal probability of successful searches at different indices).

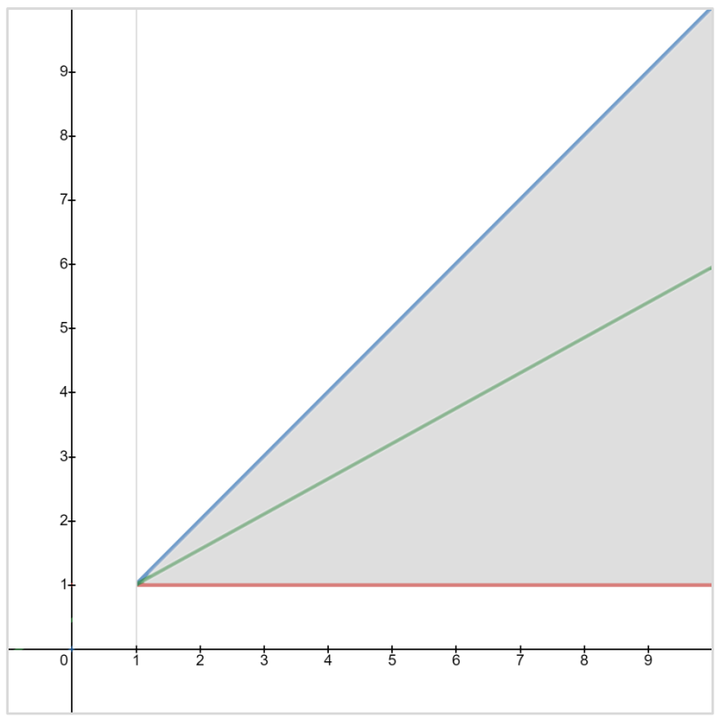


Figure 2. Efficiency of Sequential Search (gray area).

ASYMPTOTIC NOTATIONS AND BASIC EFFICIENCY CLASSES

The efficiency analysis framework focuses on the order of growth of the basic operation count of an algorithm as the principal indicator of its efficiency.

To compare/rank orders of growth, computer scientists use 3 mathematical notations:

- **Big O notation:** if f(n) is in Big O of g(n) write $f(n) \in O(g(n))$
- **Big Omega notation:** if f(n) is in Big Omega of g(n) write $f(n) \in \Omega(g(n))$
- **Big Tetha notation:** if f(n) is in Big Tetha of g(n) write $f(n) \in \Theta(g(n))$

Note: Consider f(n) and g(n) as non-negative functions defined on positive integers.

Note: Usually, f(n) is the basic operation count and g(n) is a function to compare f(n) with.

Note: These notations can also be used with multiple variables (e.g., $f(n,m)=2n^m+3$ is in $O(n^m)$).

BIG O NOTATION

O(g(n)) is the set of all functions with a lower or same order of growth as g(n)
(to within a constant multiple, as n goes to infinity).

Example: Consider the following assertions that are all true.

$$n \in O(n^2), \quad 100n + 5 \in O(n^2), \quad \frac{1}{2} n(n-1) \in O(n^2).$$

$$n^3 \notin O(n^2), \quad 0.0001 n^3 \notin O(n^2), \quad n^4 + n + 1 \notin O(n^2).$$

The functions (n, and 100n+5) are linear, so have lower order of growth than n^2.

The function (1/2 n (n-1)) is quadratic, so has same order of growth as n^2.

The functions (n^3, and 0.0001 n^3) are cubic, so have higher order of growth than n^2.

(n^4+n+1) is quartic (4^{th}-degree polynomial), so has higher order of growth than n^2.

BIG O NOTATION 2

Definition: A function t(n) is in O(g(n)), if there exist a positive constant c and a non-negative integer n_0 such that: t(n) is less or equal than cg(n) for any n greater or equal than n_0. That is, if t(n) is bounded above by some positive constant multiple of g(n) for all large n.

$$t(n) \in O(g(n)) \iff \exists\, c > 0, n_0 \geq 0 : \quad t(n) \leq c\, g(n), \quad \forall\, n \geq n_0.$$

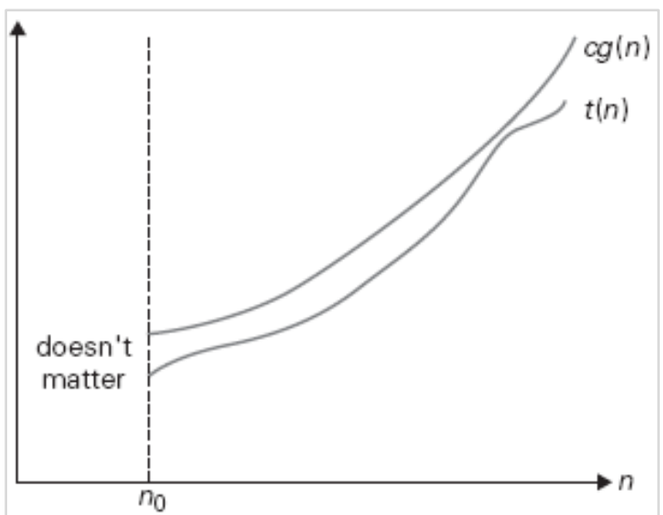


Figure 3. The function t(n) is in O(g(n)).

BIG O NOTATION 3

Question: Can we prove that 100n+5 is in $O(n^2)$?

Answer: For any n greater than 5 (n_0):

$$100\,n + 5 \leq 100\,n + n, \quad \forall\, n \geq 5$$

$$100\,n + n = 101\,n \leq 101\,n^2, \quad \forall\, n \geq 1$$

$$100\,n + 5 \leq 100\,n + n = 101\,n \leq 101\,n^2, \quad \forall\, n \geq 5$$

So, as required by the definition of Big O, we can choose c=101 and n_0=5.

Note: The Big O definition gives us freedom in choosing the values for constants c and n_0. So, we could find multiple values allowing us to satisfy the definition requirements.

For example, we can also complete the previous proof choosing c=105 and n_0=1.

BIG OMEGA NOTATION

Ω(g(n)) is the set of all functions with a higher or same order of growth as g(n) (to within a constant multiple, as n goes to infinity).

Example: Consider the following assertions that are all true.

$$n^3 \in \Omega(n^2), \quad 100\,n + 5 \in \Omega(n), \quad \frac{1}{2}\,n\,(n-1) \in \Omega(n^2).$$

$$n^3 \notin \Omega(n^4), \quad 0.0001\,n^3 \notin \Omega(n^4), \quad n^4 + n + 1 \notin \Omega(n^5).$$

BIG OMEGA NOTATION 2

Definition: A function t(n) is in Ω(g(n)), if there exist a positive constant c and a non-negative integer n_0 such that: t(n) is greater or equal than cg(n) for any n greater or equal than n_0. That is, if t(n) is bounded below by some positive constant multiple of g(n) for all large n.

$$\mathrm{t}(n) \in \Omega\big(\mathrm{g}(n)\big) \iff \exists\, c > 0, n_0 \geq 0 : \quad \mathrm{t}(n) \geq c\, \mathrm{g}(n), \quad \forall\, n \geq n_0.$$

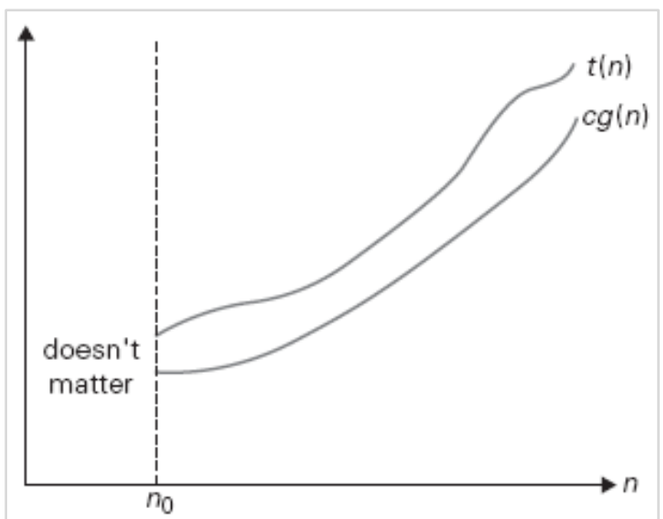


Figure 4. The function t(n) is in Ω(g(n)).

BIG THETA NOTATION

Θ(g(n)) is the set of all functions that have the same order of growth as g(n) (to within a constant multiple, as n goes to infinity).

Example: Consider the following assertions that are all true.

$$n^3 \in \Theta(n^3), \quad 100\,n + 5 \in \Theta(n), \quad \frac{1}{2}\,n\,(n-1) \in \Theta(n^2).$$

$$n^3 \notin \Theta(n^4), \quad 0.0001\,n^3 \notin \Theta(n^2), \quad n^4 + n + 1 \notin \Theta(n^5).$$

BIG THETA NOTATION 2

Definition: A function t(n) is in $\Theta(g(n))$, if there exist 2 positive constants c_1 and c_2, and a non-negative integer n_0 such that: t(n) is less or equal than $c_1 g(n)$, and greater or equal than $c_2 g(n)$, for any n greater or equal than n_0. That is, if t(n) is bounded above and below by some positive constant multiples of g(n) for all large n.

$$t(n) \in \Theta(g(n)) \quad \Leftrightarrow \quad \exists\, c_1 > 0, c_2 > 0, n_0 \geq 0 : \quad c_2\, g(n) \leq t(n) \leq c_1\, g(n), \quad \forall\, n \geq n_0.$$

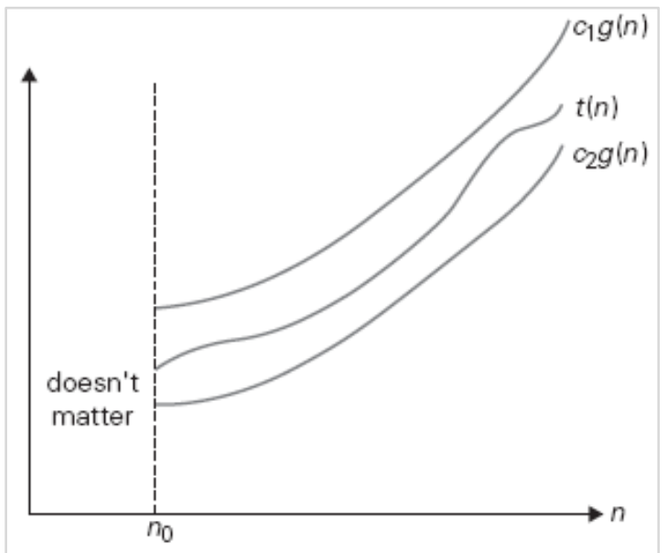


Figure 5. The function t(n) is in $\Theta(g(n))$.

BIG THETA NOTATION 3

Question: Can we prove that ½ n (n-1) is in $\Theta(n^2)$?

Answer: The right inequality (upper bound) is:

$$\frac{1}{2}\, n\,(n-1) = \frac{1}{2}\, n^2 - \frac{1}{2}\, n \leq \frac{1}{2}\, n^2, \quad \text{for all } n \geq 0.$$

The left inequality (lower bound) is:

$$\frac{1}{2}\, n\,(n-1) = \frac{n^2}{2} - \frac{n}{2} \geq \frac{n^2}{2} - \frac{n}{2}\frac{n}{2} (\text{for all } n \geq 2) = \frac{1}{4} n^2$$

So, we can select c_2=1/4, c_1=1/2, and n_0=2.

Note: The Big Theta definition gives us freedom in choosing the values for constants c_1, c_2, and n_0. So, we could find multiple values allowing us to satisfy the definition requirements.

MAIN PROPERTIES OF ASYMPTOTIC ORDER OF GROWTH

Theorem: $t_1(n) \in O(g_1(n)), t_2(n) \in O(g_2(n)) \implies t_1(n) + t_2(n) \in O(\max\{ g_1(n), g_2(n) \})$

Analogous assertions are true for Big Omega and Big Theta notations as well.

Theorem: $t_1(n) \in O(g_1(n)), t_2(n) \in O(g_2(n)) \implies t_1(n) \times t_2(n) \in O(g_1(n) \times g_2(n))$

Analogous assertions are true for Big Omega and Big Theta notations as well.

Theorem: $t(n) \in O(g(n)) \iff g(n) \in \Omega(t(n))$

Theorem: $\max\{ g_1(n), g_2(n) \} \in \Theta(g_1(n) + g_2(n))$

MAIN PROPERTIES OF ASYMPTOTIC ORDER OF GROWTH 2

Question: What is the efficiency of an algorithm performing 2 consecutive tasks?

Answer: The algorithm efficiency is determined by the task with the higher order of growth, that is the least efficient task of the algorithm.

Example: Check whether an array has equal elements using this algorithm.

Step 1: Sort the array (with max ½ n (n-1) comparisons, so this task is in $O(n^2)$).

Step 2: Scan the sorted array to check its consecutive elements for equality (with max n-1 comparisons, so this task is O(n)).

So, the efficiency of this algorithm will be in $O(\max\{ n^2, n \}) = O(n^2)$.

This efficiency analysis highlights the fact that Step 1 is the least efficient task (i.e., bottleneck); so, any efficiency optimization should address that task first.

USING LIMITS TO COMPARE ORDERS OF GROWTH

To compare the orders of growth of two functions you can compute the limit of their ratio.
Three principal cases may arise:

$$\lim_{n\to\infty} \frac{\mathrm{t}(n)}{\mathrm{g}(n)} = \begin{cases} 0 \\ c \\ \infty \end{cases}$$

- If this limit equals c (a positive constant), then t(n) is in Θ(g(n)).
- If this limit is zero (0), then t(n) is in O(g(n)).
- If this limit is infinity (∞), then t(n) is in Ω(g(n)).

Note: If this limit does not exist, use the definitions to compare the two functions.

USING LIMITS TO COMPARE ORDERS OF GROWTH 2

Example: Compare the orders of growth of ½ n (n-1) and n^2.

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

This limit is equal to ½ (a positive constant); so, these functions (1/2 n (n-1) and n^2) have the same order of growth.

USING LIMITS TO COMPARE ORDERS OF GROWTH 3

Example: Compare the orders of growth of $\log_2(n)$ and sqrt(n).

We can use the rule of L'Hôpital's: $\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$

$$\lim_{n\to\infty} \frac{\log_2(n)}{\sqrt{n}} = \lim_{n\to\infty} \frac{(\log_2(n))'}{(\sqrt{n})'} = \lim_{n\to\infty} \frac{(\log_2(e))\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2\log_2(e) \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0$$

This limit is equal to 0 (zero); so, $\log_2(n)$ has a smaller order of growth than sqrt(n) (i.e., $\log_2(n)$ is in O(sqrt(n))).

USING LIMITS TO COMPARE ORDERS OF GROWTH 4

Example: Compare the orders of growth of n! and 2^n.

We can use the Stirling's formula: $n! \approx \sqrt{2\,\pi\,n}\left(\frac{n}{e}\right)^n$, for large values of n.

$$\lim_{n\to\infty}\frac{n!}{2^n} = \lim_{n\to\infty}\frac{\sqrt{2\,\pi\,n}\left(\frac{n}{e}\right)^n}{2^n} = \lim_{n\to\infty}\sqrt{2\,\pi\,n}\,\frac{n^n}{2^n\,e^n} = \lim_{n\to\infty}\sqrt{2\,\pi\,n}\left(\frac{n}{2e}\right)^n = \infty$$

This limit is equal to ∞ (infinity); so, even though 2^n grows very fast, n! still grows faster (i.e., n! is in $\Omega(2^n)$).

BASIC EFFICIENCY CLASSES

The analysis framework for algorithm efficiency puts together all functions whose orders of growth differ by a constant multiple, however there are still infinitely many such classes.
However, most of algorithm efficiencies fall into only few classes of order of growth.

CLASS	NAME	COMMENTS
$O(1)$	Constant efficiency	For best-case efficiencies (with Big Omega)
$O(\log(n))$	Logarithmic efficiency	Algorithms reducing problem size by constant factor
$O(n)$	Linear efficiency	Algorithms with a single scan of a list/array
$O(n \log(n))$	Linearithmic efficiency	Many divide-and-conquer algorithms
$O(n^2)$	Quadratic efficiency	Algorithms with two nested iterations
$O(n^3)$	Cubic efficiency	Algorithms with three nested iterations
$O(2^n)$	Exponential efficiency (2)	Algorithms that generate all subsets of an n-items set
$O(3^n)$	Exponential efficiency (3)	This efficiency class is different than $O(2^n)$
$O(n!)$	Factorial efficiency	Algorithms generating all permutations of an n-items set

THEORETICAL AND EMPIRICAL ANALYSIS OF ALGORITHMS

In this section we will discuss different methods to analyze algorithms depending on their type, our goals, and the data available.

Theoretical Analysis: Using the analysis framework to analyze the efficiency of algorithms theoretically, independently from a specific implementation. The results are more general but usually include several approximations. The mathematical tools needed will differ to analyze non-recursive or recursive algorithms.

Empirical Analysis: Using empirical methods to study the efficiency of algorithms, considering a specific implementation (and hardware, and inputs). The results are exact but valid for a specific setup (code, hardware, inputs). This analysis requires the design of an experiment, data collection, and data analysis.

THEORETICAL ANALYSIS OF NON-RECURSIVE ALGORITHMS

This is a step-by-step process to perform a theoretical analysis of non-recursive algorithms:

Step 1: Decide the input size (the parameter, or parameters, affecting the amount of work done by the algorithm).

Step 2: Identify the basic operation (usually, the most executed, most time-consuming, and most meaningful operation performed by the algorithm).

Step 3: Check if the count of basic operations depends only on the input size, or not. If it also depends on other properties, the worst-case, average-case, and best-case efficiencies have to be investigated separately.

Step 4: Set up a summation expressing the count of basic operations.

Step 5: Using mathematical tools convert the summation into a closed-form formula or, at least, establish its order of growth.

THEORETICAL ANALYSIS OF NON-RECURSIVE ALGORITHMS 2

These are some summation rules useful to convert a sum into a closed-form formula:

Summation Rule 1: $\sum_{i=l}^{u} ca_i = c \sum_{i=l}^{u} a_i$

Summation Rule 2: $\sum_{i=l}^{u} (a_i \mp b_i) = \sum_{i=l}^{u} a_i \mp \sum_{i=l}^{u} b_i$

Summation Rule 3: $\sum_{i=l}^{u} 1 = u - l + 1, \quad \text{where } l \leq u$

Summation Rule 4: $\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{1}{2} n (n + 1)$

ANALYSIS EXAMPLE: FIND MAX IN LIST

Example: Finding the max in a list (array) of n integers.

```
int FindMax( int[] list ) {
   if( list.length > 0 ) {
      int max = list[0];
      for( int i = 1; i < list.length; i++ ) {
         if( list[i] > max ) { max = list[i]; }
      }
      return max;
   }
   else { return -1; }
}
```

The input size is the item-size of the list/array (n)(more items, more work).
The basic operation is the comparison (most executed, most meaningful).
The count of basic operation depends only on the input size; so, in this case, there is no need to distinguish among worst-case, average-case, and best-case.

$$\text{count of basic operation} = C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

ANALYSIS EXAMPLE: CHECK ITEM UNIQUENESS IN LIST

Example: Check whether all items in a given list/array are distinct (unique):

```
boolean CheckItemUniqueness( int[] list ) {
   for( int i = 0; i < list.length - 1; i++ ) {
      for( int j = i+1; j < list.length; j++ ) {
         if( list[i] == list[j] ) { return false; }
      }
   }
   return true;
}
```

The input size is the item-size of the list/array (n)(more items, more work).
The basic operation is the comparison (most executed, most meaningful).

Note that, in this case, the count of basic operations depends not only on the input size, but also on whether there are equal elements in the list and, if so, which positions they occupy.

So, in this case, we need to analyze worst-case, average-case, and best-case separately (because usually we will find different formulas for the count).

ANALYSIS EXAMPLE: CHECK ITEM UNIQUENESS IN LIST 2

Example: (continued) Check whether all items in a given list/array are distinct (unique):

- In the **worst-case** we consider an input list of size n, and we focus on the scenario that requires the max amount of work done by the algorithm (i.e., lists with no equal items, and lists with all different items except the last 2).

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{1}{2}\, n\,(n-1) \in \Theta(n^2)$$

- In the **best-case** we consider an input of size n, and we focus on the scenario that requires the min amount of work done by the algorithm (i.e., lists with the first pair of items is a pair of equal items).

$$C_{best}(n) = 1 \in \Theta(1)$$

- So, the general count C(n) is in: $C(n) \in O(n^2)$ and $C(n) \in \Omega(1)$

THEORETICAL ANALYSIS OF RECURSIVE ALGORITHMS

This is a step-by-step process to perform a theoretical analysis of recursive algorithms:

Step 1: Decide the input size (the parameter, or parameters, affecting the amount of work done by the algorithm).

Step 2: Identify the basic operation (usually, the most executed, most time-consuming, and most meaningful operation performed by the algorithm).

Step 3: Check if the count of basic operations depends only on the input size, or not. If it also depends on other properties, the worst-case, average-case, and best-case efficiencies have to be investigated separately.

Step 4: Set up a recursive definition expressing the count of basic operations.

Step 5: Using mathematical tools convert the recursive definition into a closed-form formula or, at least, establish its order of growth.

THEORETICAL ANALYSIS OF RECURSIVE ALGORITHMS 2

These are math tools useful to convert a recursive definition into a closed-form formula:

Backward Substitutions: Start from the count C(n).
Substitute the count C(n) using its recurrence, repeatedly.
After few substitutions, a parametric pattern should emerge.
Select a parameter value to convert the pattern into a formula.

Smoothness Rule: Sometimes the recursive definition is not easy to convert.
In these cases, we can analyze only a subset of input sizes (e.g., $n=2^k$).
Then, using the Smoothness Rule we can generalize our results.

Recursive Calls Tree: If a recursive algorithm has multiple recursive calls, you can analyze it building a tree of recursive calls (tree nodes represent recursive calls). Then, using tree properties we can facilitate the analysis.

Note: These math tools, here briefly introduced, will be explained in more detail later.

ANALYSIS EXAMPLE: COMPUTE FACTORIAL FUNCTION

Example: Compute the factorial function (Factorial(n) = n!) for a non-negative integer n:

```
int Factorial( int n ) {
   if( n == 0 ) { return 1; }
   else { return ( Factorial( n-1 ) * n ); }
}
```

This algorithm is associated with this recursive definition (describing its logic).

$$\text{Factorial}(n) = \begin{cases} \text{Factorial}(\, n-1 \,) \times n, & \text{if } n \geq 1. \\ 1, & \text{if } n = 0. \end{cases}$$

The input size is the **magnitude of the input integer** (bigger integer, more work).
The basic operation is the multiplication (most executed, most meaningful).
The count of basic operation depends only on the input size; so, in this case, there is no need to distinguish among worst-case, average-case, and best-case.

ANALYSIS EXAMPLE: COMPUTE FACTORIAL FUNCTION 2

Example: (continued) Compute the factorial function for a non-negative integer n.

First, we have to define the count of basic operations as a recursive definition.

$$\text{count of basic operations} = \mathrm{M}(n) = \begin{cases} \mathrm{M}(n-1) + 1, & \text{if } n \geq 1. \\ 0, & \text{if } n = 0. \end{cases}$$

Note the similarities and the differences between these 2 recursive definitions.

$$\text{algorithm computation} = \text{Factorial}(n) = \begin{cases} \text{Factorial}(n-1) \times n, & \text{if } n \geq 1. \\ 1, & \text{if } n = 0. \end{cases}$$

Also note the relation between the algorithm logic and the basic operation count.

```
int Factorial( int n ) {
   if( n == 0 ) { return 1; }
   else { return ( Factorial( n-1 ) * n ); }
}
```

Now we can convert the count M(n)(recursive definition) in a closed-form formula.

ANALYSIS EXAMPLE: COMPUTE FACTORIAL FUNCTION 3

Example: (continued) Compute the factorial function for a non-negative integer n.

Now we have to convert the count M(n) into a closed-form formula.

$$\text{count of basic operations} = \mathrm{M}(n) = \begin{cases} \mathrm{M}(n-1) + 1, & \text{if } n \geq 1. \\ 0, & \text{if } n = 0. \end{cases}$$

We will use the method of Backward Substitutions.

$$\begin{aligned} & M(n) = M(n-1) + 1 = \\ \text{apply recurrence} \quad & = [M(n-2) + 1] + 1 = M(n-2) + 2 = \\ & = [M(n-3) + 1] + 2 = M(n-3) + 3 \\ & \Downarrow \\ \text{create pattern} \quad & M(n) = M(n-i) + i, \ \text{ with } i \in [0, n] \\ & \Downarrow \\ \text{solve pattern} \quad & M(n-i) + i = M(n-n) + n = M(0) + n = n \end{aligned}$$

EMPIRICAL ANALYSIS OF ALGORITHMS

This is a step-by-step process to perform an empirical analysis of an algorithm:

Step 1: Decide the purpose of the experiment (e.g., estimate result error, measure time efficiency, compare algorithms, etc.).

Step 2: Decide the metric to be measured (e.g., count, time, percentage, etc.).

Step 3: Decide the characteristics of the input sample (e.g., sample size, input range, input sizes, randomized inputs, etc.).

Step 4: Implement the algorithm (e.g., programming language, target hardware, etc.).

Step 5: Generate the input sample (e.g., input files, user records, random inputs, etc.).

Step 6: Run the algorithm implementation on the input sample, and record the data observed (i.e., the metric to be measured).

Step 7: Analyze the recorded data (e.g., data tables, visual plots, data analytics, etc.)

EMPIRICAL ANALYSIS OF ALGORITHMS 2

Step 1: Decide the purpose of the experiment.

There are several goals one can pursue in analyzing algorithms empirically:

- Checking a theoretical assertion about the algorithm efficiency.
- Comparing the efficiency of different algorithms for the same problem.
- Developing a hypothesis about the class of the algorithm efficiency.

The design of the experiment depends on the questions we want to answer.

EMPIRICAL ANALYSIS OF ALGORITHMS 3

Step 2: Decide the metric to be measured.

The experiment goal influences what is to be measured (and how):

- Insert a counter into the algorithm implementation.
- Use OS time queries to time the algorithm implementation.

Remember that the data measured could be more or less accurate (e.g., OS time queries, non-real-time OS, etc.).

The timing of different segments of a program is called profiling, and it can be useful to pinpoint a bottleneck (i.e., the slowest stage that should be the first target of any optimization effort).

EMPIRICAL ANALYSIS OF ALGORITHMS 4

Step 3: Decide the characteristics of the input sample.

Step 5: Generate the input sample.

In deciding the input sample, often the goal is to represent a "typical" input.

To develop an input sample, you need to make decisions about:

- The sample size (i.e., how many different inputs will be tested).
- The inputs range (i.e., the range for the inputs to be tested).
- The input characteristics (e.g., size of each input, randomization, etc.).

Then, you have to generate the input sample (manually or using a software).

Remember that, if you expect the measured metric to vary on inputs of the same size, you should include multiple inputs for every size in the sample.

EMPIRICAL ANALYSIS OF ALGORITHMS 5

Step 4: Implement the algorithm.

In coding the algorithm, you need to decide:

- The programming language (e.g., Java, C++, etc.).
- The target software platform (e.g., OS, compiler version, etc.).
- The target hardware platform (e.g., mobile, desktop, CPU, RAM, etc.).

Remember that, the data recorded is affected by all these details, so your results are meaningful only if associated with the information above (always reported).

EMPIRICAL ANALYSIS OF ALGORITHMS 6

Step 6: Run the algorithm implementation on the input sample, and record the data.

Step 7: Analyze the recorded data

The data recorded during the experiment is then presented for analysis:

- As raw data in a table (e.g., to facilitate post-processing.
- As 2D/3D samples in a scatterplot (e.g., to facilitate visual evaluation).

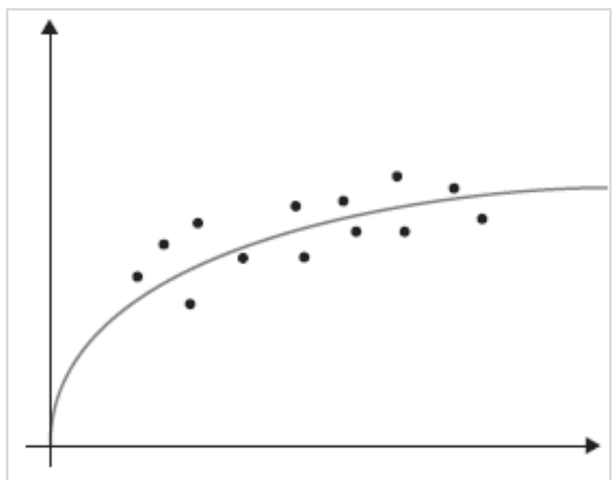


Figure 6. A plot of the experiment data, with a fitting function.

ALGORITHM VISUALIZATION

In addition to theoretical and empirical analyses of algorithms, there is another way to study algorithms: algorithm visualization.

Algorithm Visualization: Using graphic elements to illustrate the algorithm processing.

The visualization can be:

- Performed in real-time (during processing), or offline.
- Illustrating the algorithm logic or performance.
- Tailored to different goals (e.g., optimization, teaching).

See: VisuAlgo