2- FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

What is «analysis»?

- The word “analysis” is defined as “the separation of an intellectual or substantial whole into its constituent parts for individual study.”

- But the term “analysis of algorithms” is usually used in a narrower, technical sense to mean an investigation of an algorithm’s efficiency with respect to two resources: running time and memory space.

- Unlike simplicity and generality, efficiency can be studied in precise quantitative terms.

2.1 The Analysis Framework

- Time efficiency, also called time complexity, indicates how fast an algorithm in question runs.

- Space efficiency, also called space complexity, refers to the amount of memory units required by the algorithm in ad-dition to the space needed for its input and output.

2.1 The Analysis Framework

- Measuring an Input's Size
- It takes longer to sort larger arrays, multiply larger matrices, and so on.
- Therefore, it is logical to investigate an algorithm's efficiency as a function of some parameter n indicating the algorithm's input size.
- In most cases, selecting such a parameter is quite straightforward.
- For example, it will be the size of the list for problems of sorting, searching, finding the list's smallest element, and most other problems dealing with lists.

2.1 The Analysis Framework

- Measuring an Input's Size
- There are situations, of course, where the choice of a parameter indicating an input size does matter.
- One such example is computing the product of two n × n matrices.
- There are two natural measures of size for this problem.
- The first and more frequently used is the matrix order n.
- But the other natural contender is the total number of elements N in the matrices being multiplied.
- Since there is a simple formula relating these two measures, we can easily switch from one to the other, but the answer about an algorithm's efficiency will be qualitatively different depending on which of these two measures we use.

2.1 The Analysis Framework

- Which is more general? n or N?

2.1 The Analysis Framework

- Units for Measuring Running Time
- Which unit(s) can we use for measuring the time of an algorithm?
- Of course, we can simply use some standard unit of time measurement—a second, or millisecond, and so on—to measure the running time of a program implementing the algorithm.

- What is the problem about seconds?

2.1 The Analysis Framework

- Units for Measuring Running Time
- There are obvious drawbacks to such an approach, however: dependence on the speed of a particular computer, dependence on the quality of a program implementing the algorithm and of the compiler used in generating the machine code, and the difficulty of clocking the actual running time of the pro-gram.
- Since we are after a measure of an algorithm's efficiency, we would like to have a metric that does not depend on these extraneous factors.

2.1 The Analysis Framework

- Units for Measuring Running Time
- One possible approach is to count the number of times each of the algorithm's operations is executed.
- This approach is both excessively difficult and, as we shall see, usually unnecessary.
- The thing to do is to identify the most important operation of the algorithm, called the basic operation, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.

2.1 The Analysis Framework

- Units for Measuring Running Time
- As a rule, it is not difficult to identify the basic operation of an algorithm: it is usually the most time-consuming operation in the algorithm's innermost loop.
- For example, most sorting algorithms work by comparing elements (keys) of a list being sorted with each other; for such algorithms, the basic operation is a key comparison.
- As another example, algorithms for mathematical problems typically involve some or all of the four arithmetical operations: addition, subtraction, multiplication, and division.
- Of the four, the most time-consuming operation is division, followed by multiplication and then addition and subtraction, with the last two usually considered together.

2.1 The Analysis Framework

- Orders of Growth

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}		

2.1 The Analysis Framework

- Worst-Case, Best-Case, and Average-Case Efficiencies
- The worst-case efficiency of an algorithm is its efficiency for the worst-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the longest among all possible inputs of that size.
- The way to determine the worst-case efficiency of an algorithm is, in principle, quite straightforward: analyze the algorithm to see what kind of inputs yield the largest value of the basic operation's count C(n) among all possible inputs of size n and then compute this worst-case value Cworst (n).

2.1 The Analysis Framework

- Worst-Case, Best-Case, and Average-Case Efficiencies
- The best-case efficiency of an algorithm is its efficiency for the best-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the fastest among all possible inputs of that size.
- Accordingly, we can analyze the best-case efficiency as follows.
- First, we determine the kind of inputs for which the count C(n) will be the smallest among all possible inputs of size n.
- Then we ascertain the value of C(n) on these most convenient inputs.
- For example, the best-case inputs for sequential search are lists of size n with their first element equal to a search key; accordingly, Cbest (n) = 1 for this algorithm.

2.1 The Analysis Framework

- Worst-Case, Best-Case, and Average-Case Efficiencies

- Neither the worst-case analysis nor its best-case counterpart yields the necessary information about an algorithm's behavior on a "typical" or "random" input.
- This is the information that the average-case efficiency seeks to provide.
- To analyze the algorithm's average-case efficiency, we must make some assumptions about possible inputs of size n.

2.2 Asymptotic Notations and Basic Efficiency Classes

To compare and rank such orders of growth, computer scientists use three notations: O (big oh), Ω (big omega), and Θ (big theta).

2.2 Asymptotic Notations and Basic Efficiency Classes

- t (n) and g(n) can be any nonnegative functions defined on the set of natural numbers. In the context we are interested in, t (n) will be an algorithm's running time (usually indicated by its basic operation count C(n)), and g(n) will be some simple function to compare the count with.

2.2 Asymptotic Notations and Basic Efficiency Classes

O-notation

DEFINITION A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq cg(n) \quad \text{for all } n \geq n_0.$$

The definition is illustrated in Figure 2.1 where, for the sake of visual clarity, n is extended to be a real number.

As an example, let us formally prove one of the assertions made in the introduction: $100n + 5 \in O(n^2)$. Indeed,

$$100n + 5 \leq 100n + n \text{ (for all } n \geq 5) = 101n \leq 101n^2.$$

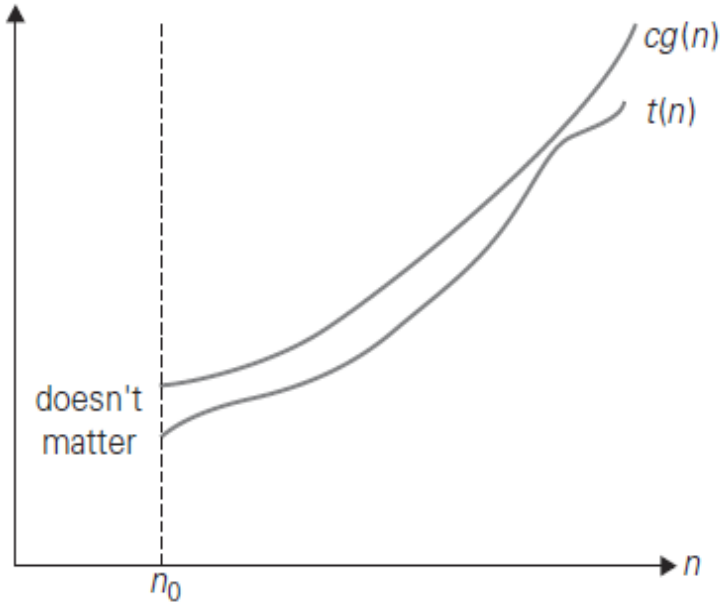
Thus, as values of the constants c and n_0 required by the definition, we can take 101 and 5, respectively.

Note that the definition gives us a lot of freedom in choosing specific values for constants c and n_0. For example, we could also reason that

$$100n + 5 \leq 100n + 5n \text{ (for all } n \geq 1) = 105n$$

to complete the proof with $c = 105$ and $n_0 = 1$.

2.2 Asymptotic Notations and Basic Efficiency Classes



Big-oh notation: $t(n) \in O(g(n))$.

2.2 Asymptotic Notations and Basic Efficiency Classes

Ω-notation

DEFINITION A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \geq cg(n) \quad \text{for all } n \geq n_0.$$

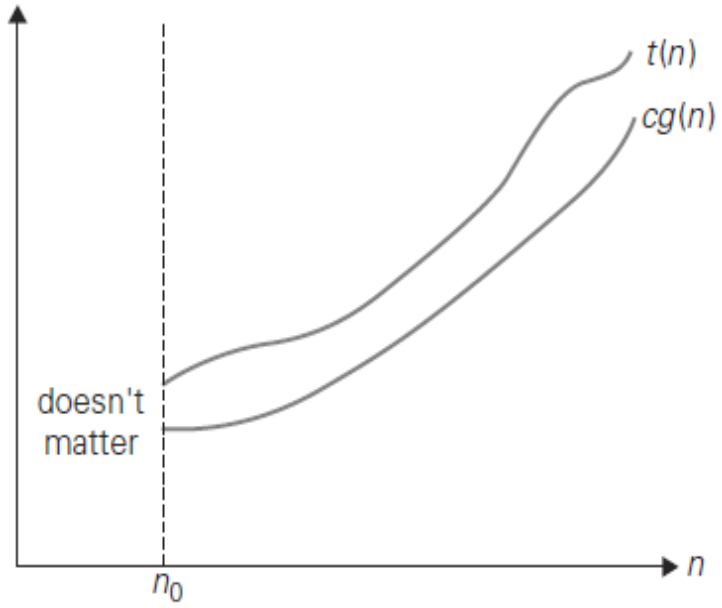
The definition is illustrated in Figure 2.2.

Here is an example of the formal proof that $n^3 \in \Omega(n^2)$:

$$n^3 \geq n^2 \quad \text{for all } n \geq 0,$$

i.e., we can select $c = 1$ and $n_0 = 0$.

2.2 Asymptotic Notations and Basic Efficiency Classes



Big-omega notation: $t(n) \in \Omega(g(n))$.

2.2 Asymptotic Notations and Basic Efficiency Classes

Θ-notation

DEFINITION A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n, i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2 g(n) \le t(n) \le c_1 g(n) \quad \text{for all } n \ge n_0.$$

The definition is illustrated in Figure 2.3.

For example, let us prove that $\frac{1}{2}n(n-1) \in \Theta(n^2)$. First, we prove the right inequality (the upper bound):

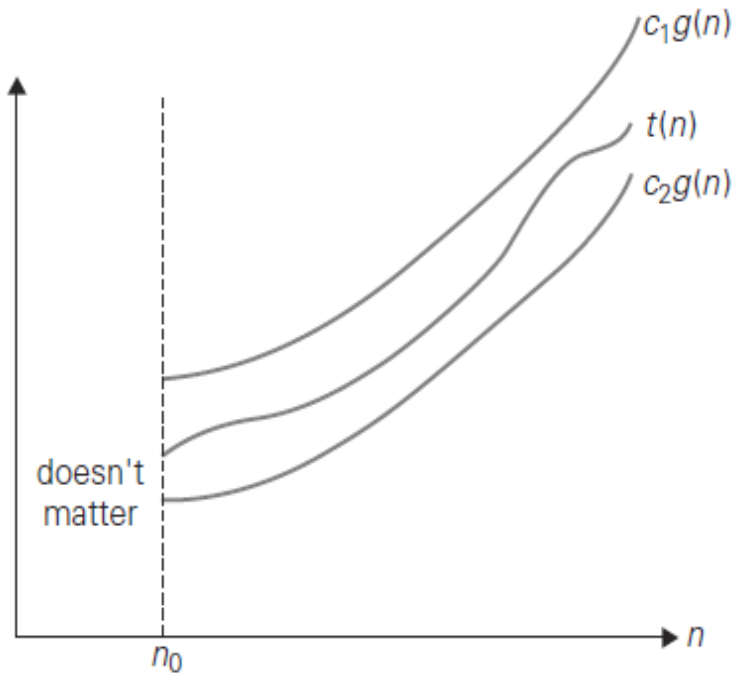
$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \le \frac{1}{2}n^2 \quad \text{for all } n \ge 0.$$

Second, we prove the left inequality (the lower bound):

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

Hence, we can select $c_2 = \frac{1}{4}$, $c_1 = \frac{1}{2}$, and $n_0 = 2$.

2.2 Asymptotic Notations and Basic Efficiency Classes



Big-theta notation: $t(n) \in \Theta(g(n))$.

2.2 Asymptotic Notations and Basic Efficiency Classes

- Basic Efficiency Classes

Class	Name	Comments
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 4.4). Note that a logarithmic algorithm cannot take into account all its input or even a fixed fraction of it: any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	*linearithmic*	Many divide-and-conquer algorithms (see Chapter 5), including mergesort and quicksort in the average case, fall into this category.
n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on $n \times n$ matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and larger orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

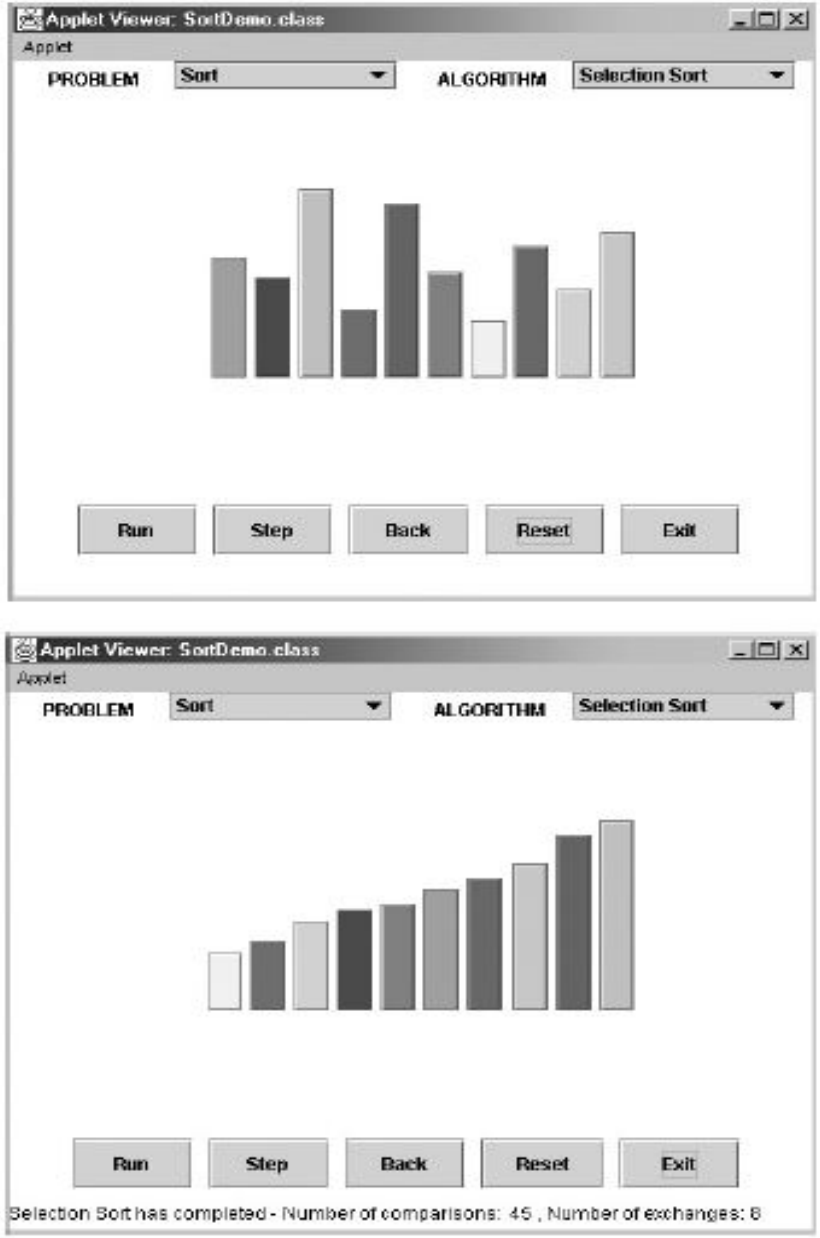
Empirical Analysis of Algorithms

- General Plan for the Empirical Analysis of Algorithm Time Efficiency
  - Understand the experiment's purpose.
  - Decide on the efficiency metric M to be measured and the measurement unit (an operation count vs. a time unit).
  - Decide on characteristics of the input sample (its range, size, and so on).
  - Prepare a program implementing the algorithm (or algorithms) for the experimentation.
  - Generate a sample of inputs.
  - Run the algorithm (or algorithms) on the sample's inputs and record the data observed.
  - Analyze the data obtained.

Algorithm Visualization

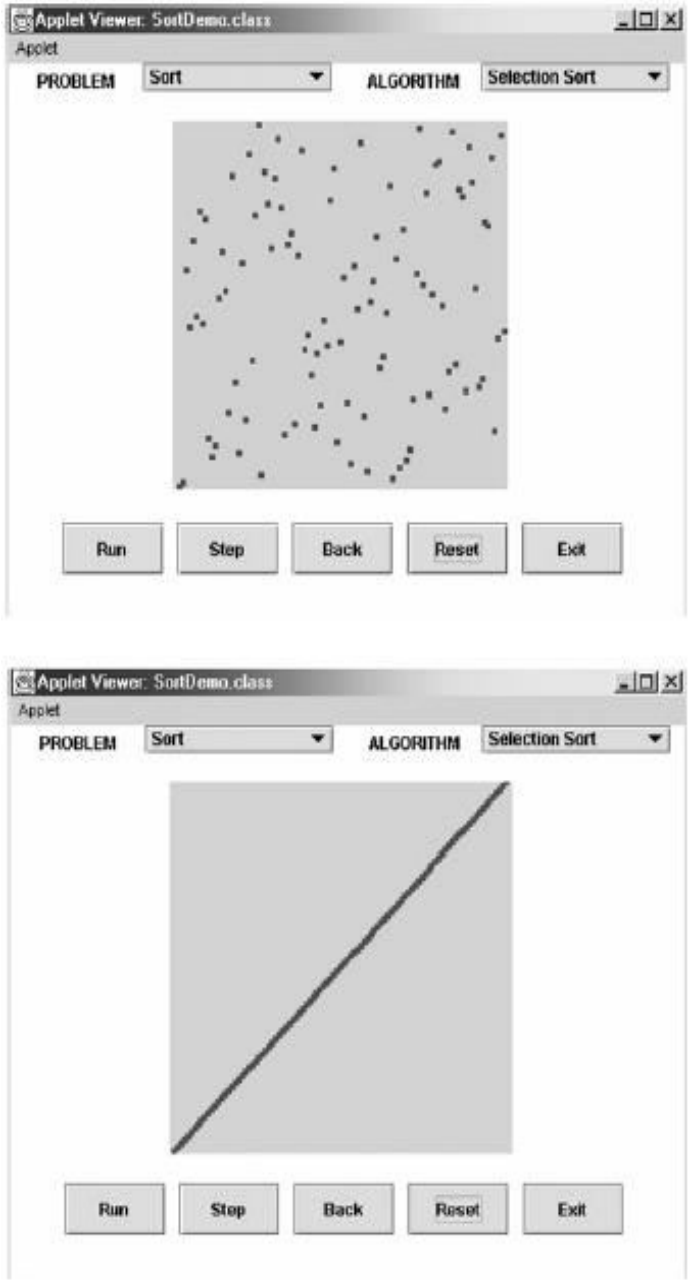
- In addition to the mathematical and empirical analyses of algorithms, there is yet a third way to study algorithms.
- It is called algorithm visualization and can be defined as the use of images to convey some useful information about algorithms.
- That information can be a visual illustration of an algorithm's operation, of its performance on different kinds of inputs, or of its execution speed versus that of other algorithms for the same problem.
- To accomplish this goal, an algorithm visualization uses graphic elements—points, line segments, two- or three-dimensional bars, and so on—to represent some "interesting events" in the algorithm's operation.

Algorithm Visualization



Initial and final screens of a typical visualization of a sorting algorithm using the bar representation.

Algorithm Visualization



Initial and final screens of a typical visualization of a sorting algorithm using the scatterplot representation.