Chapter 1 Introduction

Introduction to Algorithms - Thomas H. Cormen

Mina Gabriel

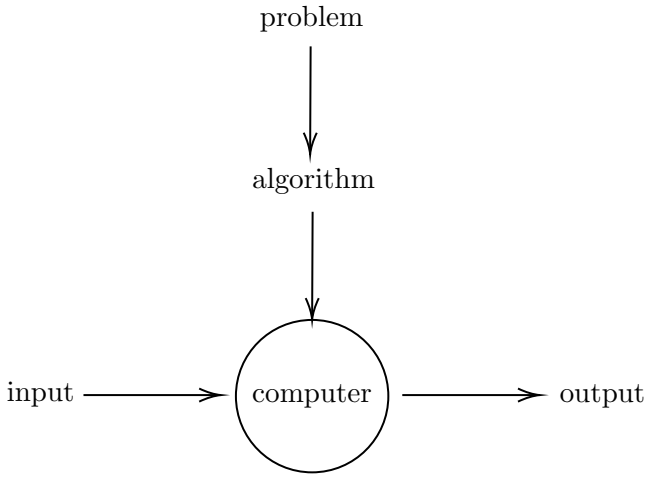
1 The Role of Algorithms in Computing

What are algorithms? Why is the study of algorithms worthwhile? What is the role of algorithms relative to other technologies used in computers?

1.1 Algorithms

An algorithm is thus a sequence of computational steps that transform the input into the output.

For example, we need to add two *int* numbers together:



Our algorithm can be defined as "adding the two variables a and b" or $c = a + b$, input is the two variable and output is c

Another example is to sort a sequence of numbers into non decreasing order, Here is how we formally define the sorting problem:

- **Input:** list of numbers $\{a_1, a_2, \ldots, a_n\}$
- **Output:** A permutation of this list $\{a_1\prime, a_2\prime, \ldots, a_n\prime\}$ such that $\{a_1\prime \leqslant a_2\prime \leqslant \cdots \leqslant a_n\prime\}$

Data structures

A **data structure** is a way to store and organize data in order to facilitate access and modifications. No single data structure works well for all purposes, and so it is important to know the strengths and limitations of several of them.

Technique

This book will teach you techniques of algorithm design and analysis so that you can develop algorithms on your own, show that they give the correct answer, and understand their efficiency.

1.2 Algorithms as a technology

Suppose computers were infinitely fast and computer memory was free. Would you have any reason to study algorithms? The answer is yes, if for no other reason than that you would still like to demonstrate that your solution method terminates and does so with the correct answer.

Efficiency

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software.

As example lets compare two algorithms for sorting:

1. **Insertion sort** takes time equal to $c_1 n^2$ where, c_1 is a constant that also does not depend on n, and n is the number of elements in a list to be sorted.
2. **Merge sort** takes time equals to $c_2\ n\ log_2\ n$

Insertion sort has smaller constant factor than merge sort, $c_1 < c_2$, for example let $n = 10$ we can compare the two algorithms as following:

$$c_1 \times n \times n < c_2\ n\ log_2\ n \tag{1}$$
$$c_1 \times 10 \times 10 < c_2 \times 10 \times 3.3219 \tag{2}$$
$$c_1 \times 10 < c_2 \times 3.3219 \tag{3}$$

Now, let $n = 10^6$, we observer:

$$c_1 \times n \times n < c_2\ n\ log_2\ n \tag{4}$$
$$c_1 \times 10^6 \times 10^6 < c_2 \times 10^6 \times 19.93157 \tag{5}$$
$$c_1 \times 10^6 < c_2 \times 19.93157 \tag{6}$$

No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

Concrete Example

	Computer A	Computer B
Array size (to be sorted)	$n = 10^7$	$n = 10^7$
Performance	10^{10} instructions/second (1000 times faster than computer B)	10^7 instructions/second (1000 times slower than computer A)
Algorithm number of instructions	$2\ n^2$	$50\ nlog_2\ n$
Time	$\frac{2\cdot(10^7)^2 \text{ instructions}}{10^{10} \text{ instructions/second}}$ =20,000$seconds$ =(more than 5.5 hours)	$\frac{50\cdot 10^7 \lg 10^7 \text{ instructions}}{10^7 \text{ instructions/second}}$ $\approx$ 1163 seconds =(less than 20 minutes)

Algorithm and data structure

Similar to a mathematical problem, a computational problem is normally defined as a mapping (or function) from an input domain to an output range, with a specific input/output relationship. An algorithm is a procedure consisting of computational steps that transforms an input value into an output value, where

- the steps are directly executable,
- the procedure is predetermined and has a finite-length description,
- the process will terminate for every valid input with an output.

Therefore, an algorithm provides a solution for the problem by specifying how to actually accomplish the mapping.

For example, **"sorting"** is a problem where the input is a sequence of items of a certain type, with a total order defined between any pair of them, and the output should be a sequence of the same items, arranged according to the order. A sorting algorithm should specify, step by step, how to turn any valid input into the corresponding output in finite time, then stop (halt) there.

An algorithm is correct if for every valid input instance, it halts with the output as specified in the problem. An algorithm is incorrect if there is a valid input instance for which the algorithm produces an incorrect answer or no answer at all (i.e. does not halt).

For a given problem, if there are multiple candidate algorithms, which one should be used? There are several factors to be considered:

- correctness
- time and space efficiency
- conceptual simplicity

They are usually considered in the above order, though very often a compromise is needed among these factors.

When the (input, output, or intermediate) data of a problem contain multiple components, they are usually organized in a data structure, which represents both the data items and some relation among them.

A data structure can be specified either abstractly, in terms of the operations (as computations) that can be carried out on it, or concretely, in terms of the storage organization and the algorithms implementing the operations. An abstract data structure often corresponds to multiple concrete ones.

In the design and selection of data structures, the analysis of the algorithms involved is a central topic.

Programming means to implement an algorithm in a computer language. A program is language-specific, while an algorithm is language-independent.

Time efficiency analysis Traditionally, algorithm analysis has been focused on the time efficiency of (correct) algorithms, though space efficiency and the correctness of an algorithm can also be analyzed.

For a given program, the actual time it takes to solve a given problem instance depends on

- the algorithm implemented in the program,
- the given problem instance,
- the software and hardware in which the program is executed.

Since the aim of algorithm analysis is to compare algorithms, the other factors need to be somehow removed from the discussion. As a starting point, the following instructions are usually assumed to be directly executable and each takes a constant amount of time.

- **arithmetic**: add, subtract, multiply, divide, remainder, floor, ceiling;
- **comparison**: equal to, larger than, smaller than;
- **data movement**: load, store, copy;
- **control**: conditional and unconditional branch, subroutine call and return.

More complicated blocks, such as loops, can be built from the above instructions. The time taken by a block is not necessarily a constant anymore.

To measure the time complexity of algorithms, the common practice is to define a size (usually using n) for each instance of the problem (which intuitively measures the relative difficulty of processing the instance for the problem), then to represent the running time as a function of this instance size

(as T(n) in the following). Finally, the increasing rate of the function, with respect to the increasing of the instance size, is used as the indicator of the efficiency or complexity of the algorithm.

With such a procedure, algorithm analysis becomes a mathematical problem, which is well-defined, and the result has universal validity.

Though it is a powerful technique, we need to keep in mind that many relevant factors have been ignored in this process, and therefore, if for certain reason some of the factors have to be taken into consideration, the traditional algorithm analyzing approach may become improper to use.

Conventions about pseudocode

To be independent of programming languages, the algorithms in the textbook are written in *pseudocode*, according to the following conventions:

- The looping and the conditional constructs are similar to those in C/Java/Python/Pascal.
- Indentation indicates block structure, without delimiters.
- Assignment sign is equal sign, "=", and multiple assignment is allowed.
- Double equal sign, "==", is for equality.
- The boolean operators "and" and "or" are short circuiting.
- Variables are local by default and do not require declaration.
- Array elements are represented as Array[index], and index usually starts at 1.
- A field in an object is represented as "object.field", and array length is Array.length.
- Parameters are passed to a procedure by value.
- A return statement transfers control back to the calling procedure.
- Double slash ("//") is used for comments.
- Lines are numbered.

Example: sort using an incremental approach

Sorting: the problem and its input/output.

```
INSERTION–SORT(A)
for j = 2 to A.length
    key = A[j]
    //textit{Insert A[j] into the sorted sequence A[1 ... j-1]}
    i = j - 1
    while i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```