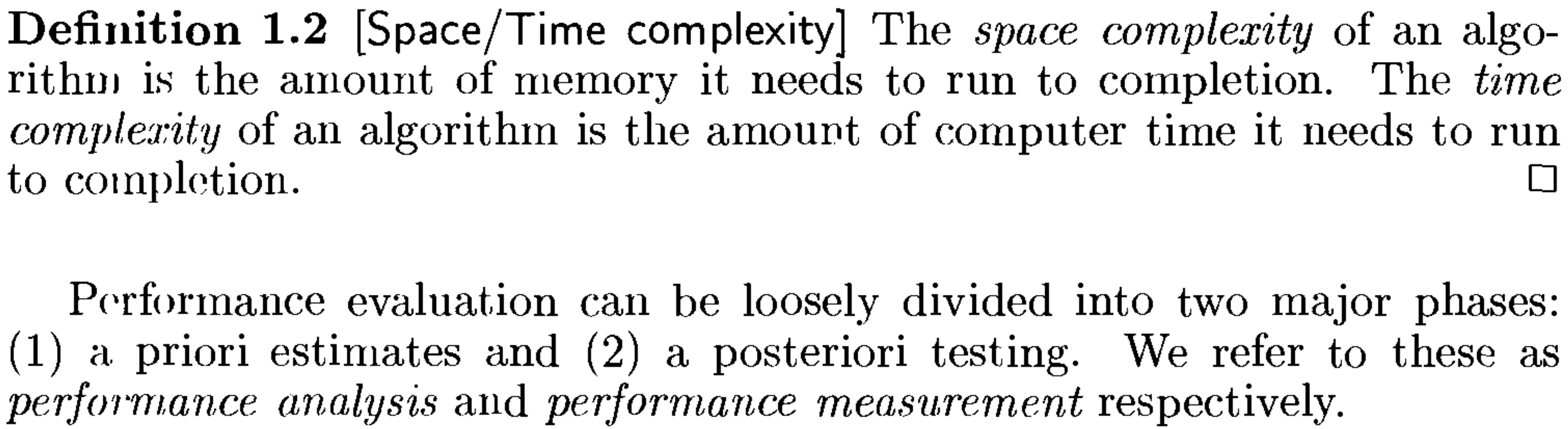
Design and Analysis of Algorithms

Time Complexity and Space Complexity



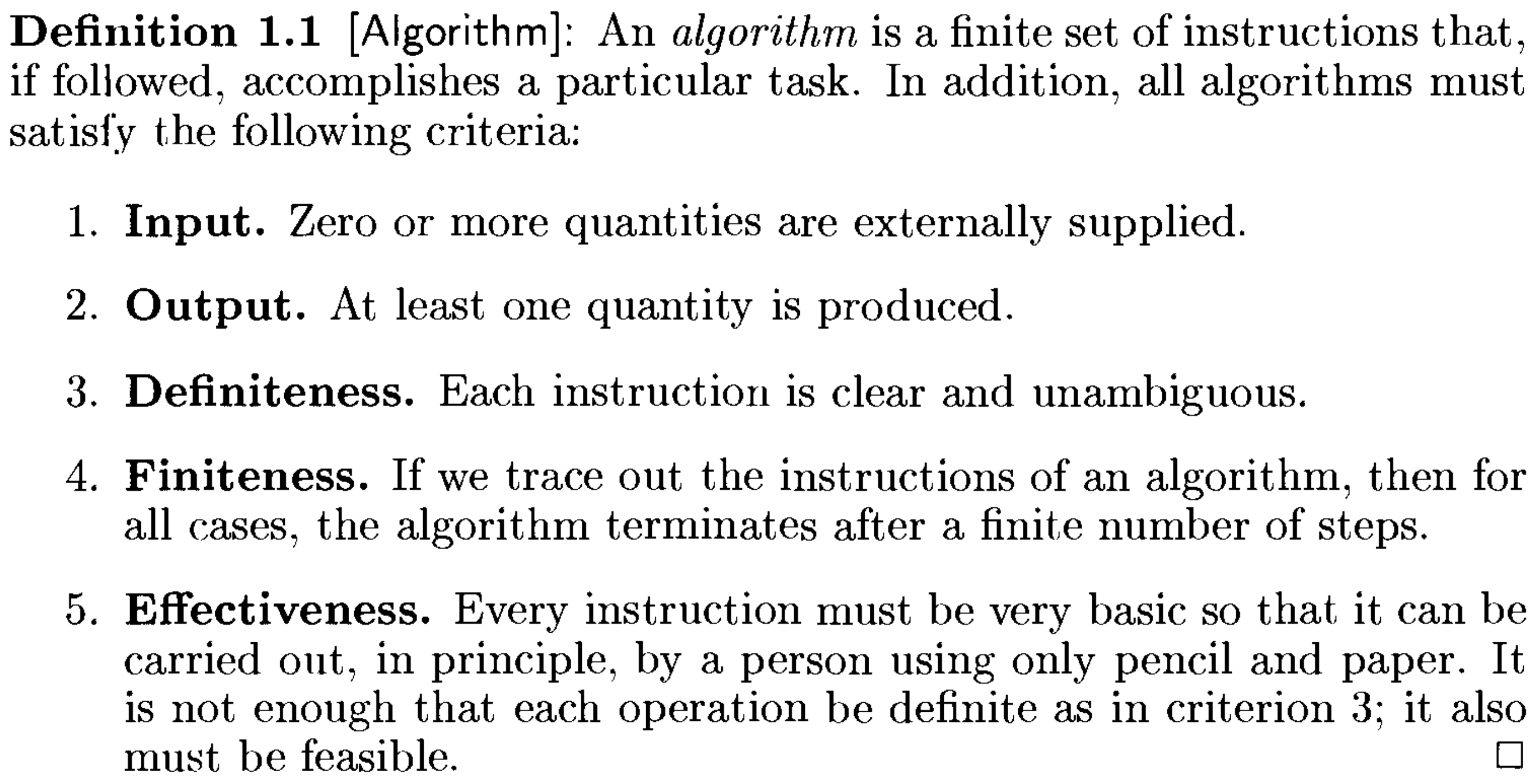
Definition 1.2 [Space/Time complexity] The *space complexity* of an algorithm is the amount of memory it needs to run to completion. The *time complexity* of an algorithm is the amount of computer time it needs to run to completion. □

Performance evaluation can be loosely divided into two major phases: (1) a priori estimates and (2) a posteriori testing. We refer to these as *performance analysis* and *performance measurement* respectively.

Abū ʿAbdallāh Muḥammad ibn Mūsā al-Khwārizmī

In the twelfth century, Latin translations of his work on the Indian numerals introduced the decimal positional number system to the Western world.[4] His Compendious Book on Calculation by Completion and Balancing presented the first systematic solution of linear and quadratic equations in Arabic.

Definition



Definition 1.1 [Algorithm]: An *algorithm* is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

1. **Input.** Zero or more quantities are externally supplied.
2. **Output.** At least one quantity is produced.
3. **Definiteness.** Each instruction is clear and unambiguous.
4. **Finiteness.** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. **Effectiveness.** Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion 3; it also must be feasible. □

Space Complexity

- The space needed by each algorithm is the sum of the following components :
- (i) Instruction space
- (ii) Data space
- (iii) Environment stack space

Contd…

- ***Instruction space***
- The space needed to store the compiled version of the program instructions.
- **(ii) *Data space***
- The space needed to store all constant and variable values.
- **(iii) *Environment stack space***
- The space needed to store information to resume execution of partially completed functions.
- Example : If function 1, invokes function 2, then we must atleast save a pointer to the instruction of function 1 to be executed when function 2 terminates.

1. A fixed part that is independent of the characteristics (e.g., number, size) of the inputs and outputs. This part typically includes the instruction space (i.e., space for the code), space for simple variables and fixed-size component variables (also called *aggregate*), space for constants, and so on.

2. A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that this depends on instance characteristics), and the recursion stack space (insofar as this space depends on the instance characteristics).

The space requirement $S(P)$ of any algorithm P may therefore be written as $S(P) = c + S_P(\text{instance characteristics})$, where c is a constant.

```
Algorithm abc($a, b, c$)
{
    return $a + b + b * c + (a + b - c)/(a + b) + 4.0$;
}
```

Algorithm 1.5 Computes $a + b + b * c + (a + b - c)/(a + b) + 4.0$

Example 1.4 For Algorithm 1.5, the problem instance is characterized by the specific values of a, b, and c. Making the assumption that one word is adequate to store the values of each of a, b, c, and the result, we see that the space needed by `abc` is independent of the instance characteristics. Consequently, $S_P(\text{instance characteristics}) = 0$. $\square$

```
Algorithm Sum($a, n$)
{
    $s := 0.0$;
    for $i := 1$ to $n$ do
        $s := s + a[i]$;
    return $s$;
}
```

Algorithm 1.6 Iterative function for sum

Example 1.5 The problem instances for Algorithm 1.6 are characterized by n, the number of elements to be summed. The space needed by n is one word, since it is of type *integer*. The space needed by a is the space needed by variables of type array of floating point numbers. This is at least n words, since a must be large enough to hold the n elements to be summed. So, we obtain $S_{\mathsf{Sum}}(n) \geq (n+3)$ (n for $a[\,]$, one each for n, i, and s). $\square$

```
Algorithm RSum(a, n)
{
    if ($n \leq 0$) then return 0.0;
    else return RSum($a, n - 1$) + $a[n]$;
}
```

Algorithm 1.7 Recursive function for sum

Example 1.6 Let us consider the algorithm RSum (Algorithm 1.7). As in the case of Sum, the instances are characterized by n. The recursion stack space includes space for the formal parameters, the local variables, and the return address. Assume that the return address requires only one word of memory. Each call to RSum requires at least three words (including space for the values of n, the return address, and a pointer to $a[\,]$). Since the depth of recursion is $n + 1$, the recursion stack space needed is $\geq 3(n + 1)$. $\square$

Time Complexity

A *program step* is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics. For example, the entire statement

$$\textbf{return } a + b + b * c + (a + b - c)/(a + b) + 4.0;$$

- (a) Comments – zero step
- (b) Assignment statement – one step
- (c) Iterative statement – finite number of steps. (for, while, repeat - until)

```
Algorithm Sum($a, n$)
{
    $s := 0.0$;
    $count := count + 1$; // $count$ is global; it is initially zero.
    for $i := 1$ to $n$ do
    {
        $count := count + 1$; // For for
        $s := s + a[i]$; $count := count + 1$; // For assignment
    }
    $count := count + 1$; // For last time of for
    $count := count + 1$; // For the return
    return $s$;
}
```

Algorithm 1.8 Algorithm 1.6 with count statements added

```
Algorithm Sum($a, n$)
{
    for $i := 1$ to $n$ do $count := count + 2$;
    $count := count + 3$;
}
```

Algorithm 1.9 Simplified version of Algorithm 1.8

```
Algorithm RSum($a, n$)
{
    $count := count + 1$; // For the if conditional
    if ($n \leq 0$) then
    {
        $count := count + 1$; // For the return
        return 0.0;
    }
    else
    {
        $count := count + 1$;  // For the addition, function
                               // invocation and return
        return RSum($a, n - 1) + a[n]$;
    }
}
```

Algorithm 1.10 Algorithm 1.7 with count statements added

When analyzing a recursive program for its step count, we often obtain a recursive formula for the step count, for example,

$$t_{\mathsf{RSum}}(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_{\mathsf{RSum}}(n-1) & \text{if } n > 0 \end{cases}$$

Matrix addition

```
Algorithm Add($a, b, c, m, n$)
{
    for $i := 1$ to $m$ do
        for $j := 1$ to $n$ do
            $c[i, j] := a[i, j] + b[i, j]$;
}
```

Algorithm 1.11 Matrix addition

```
**Algorithm** Add($a, b, c, m, n$)
{
    **for** $i := 1$ **to** $m$ **do**
    {
        $count := count + 1$; // For 'for $i$'
        **for** $j := 1$ **to** $n$ **do**
        {
            $count := count + 1$; // For 'for $j$'
            $c[i, j] := a[i, j] + b[i, j]$;
            $count := count + 1$; // For the assignment
        }
        $count := count + 1$;// For loop initialization and
                             // last time of 'for $j$'
    }
    $count := count + 1$;    // For loop initialization and
                             // last time of 'for $i$'
}
```

Algorithm 1.12 Matrix addition with counting statements

Simplified one

```
**Algorithm** Add$(a, b, c, m, n)$
{
    **for** $i := 1$ **to** $m$ **do**
    {
        $count := count + 2$;
        **for** $j := 1$ **to** $n$ **do**
            $count := count + 2$;
    }
    $count := count + 1$;
}
```

Algorithm 1.13 Simplified algorithm with counting only

Method 2

To determine the step count of an algorithm is to build a table in which we list the total number of steps contributed by each statement. The final total stepcount is obtained by consecutive three steps;

H The number of *steps per execution* (s/e) of the statement is calculated.

H Total number of times each statement is executed (ie *frequency*)

H Multiply (s/e) and frequency to find the total steps of each statement and add the total steps of each statement to obtain a *final stepcount (i.e. total)*

Statement	s/e	frequency	total steps
1 **Algorithm** Sum(a, n)	0	$-$	0
2 {	0	$-$	0
3 $s := 0.0;$	1	1	1
4 **for** $i := 1$ **to** n **do**	1	$n+1$	$n+1$
5 $s := s + a[i];$	1	n	n
6 **return** s;	1	1	1
7 }	0	$-$	0
Total			$2n+3$

Table 1.1 Step table for Algorithm 1.6

Statement	s/e	frequency		total steps	
		$n = 0$	$n > 0$	$n = 0$	$n > 0$
1 **Algorithm** RSum(a, n)	0	–	–	0	0
2 {					
3 **if** $(n \leq 0)$ **then**	1	1	1	1	1
4 **return** 0.0;	1	1	0	1	0
5 **else return**					
6 RSum$(a, n-1) + a[n]$;	$1 + x$	0	1	0	$1 + x$
7 }	0	–	–	0	0
Total				2	$2 + x$

$$x = t_{\mathsf{RSum}}(n-1)$$

Table 1.2 Step table for Algorithm 1.7

Statement	s/e	frequency	total steps
1 **Algorithm** Add(a, b, c, m, n)	0	–	0
2 {	0	–	0
3 **for** $i := 1$ **to** m **do**	1	$m+1$	$m+1$
4 **for** $j := 1$ **to** n **do**	1	$m(n+1)$	$mn+m$
5 $c[i,j] := a[i,j] + b[i,j];$	1	mn	mn
6 }	0	–	0
Total			$2mn+2m+1$

Table 1.3 Step table for Algorithm 1.11

```
Algorithm Fibonacci($n$)
// Compute the $n$th Fibonacci number.
{
    if ($n \leq 1$) then
        write ($n$);
    else
    {
        $fnm2 := 0$; $fnm1 := 1$;
        for $i := 2$ to $n$ do
        {
            $fn := fnm1 + fnm2$;
            $fnm2 := fnm1$; $fnm1 := fn$;
        }
        write ($fn$);
    }
}
```

Algorithm 1.14 Fibonacci numbers

To analyze the time complexity of this algorithm, we need to consider the two cases (1) $n = 0$ or 1 and (2) $n > 1$. When $n = 0$ or 1, lines 4 and 5 get executed once each. Since each line has an s/e of 1, the total step count for this case is 2. When $n > 1$, lines 4, 8, and 14 are each executed once. Line 9 gets executed n times, and lines 11 and 12 get executed $n - 1$ times each (note that the last time line 9 is executed, i is incremented to $n + 1$, and the loop exited). Line 8 has an s/e of 2, line 12 has an s/e of 2, and line 13 has an s/e of 0. The remaining lines that get executed have s/e's of 1. The total steps for the case $n > 1$ is therefore $4n + 1$. $\square$