Design and analysis of algorithms

BCS401

What is algorithms?

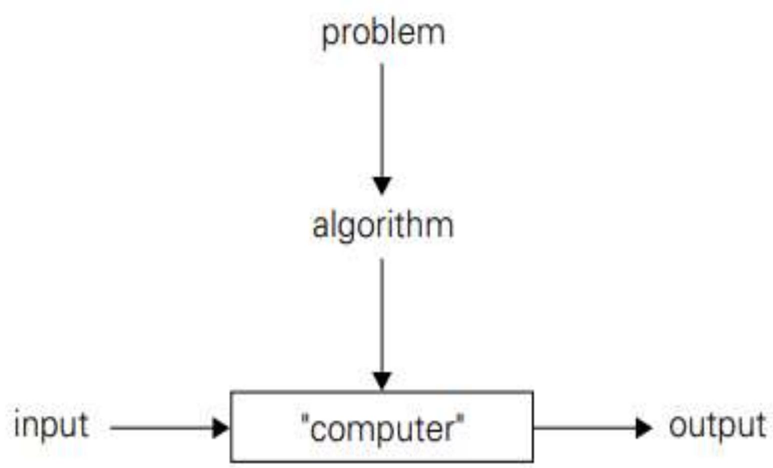

FIGURE 1.1 The notion of the algorithm.

1.2 Fundamentals of Algorithmic Problem Solving

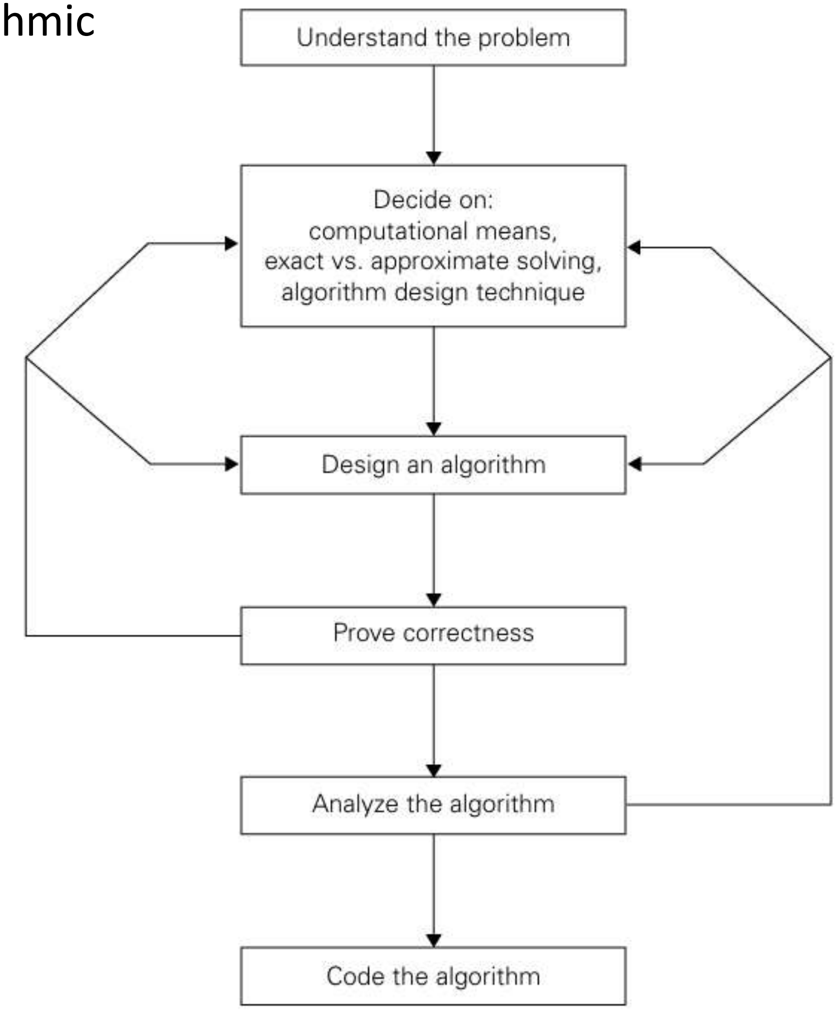

FIGURE 1.2 Algorithm design and analysis process.

1.2 Fundamentals of Algorithmic Problem Solving

1.Understanding the Problem
2. Ascertaining the Capabilities of the Computational Device
3. Choosing between Exact and Approximate Problem Solving
4. Algorithm Design Techniques
5.Designing an Algorithm and Data Structures
6.Methods of Specifying an Algorithms
7. Proving an Algorithm's Correctness
8. Analyzing an Algorithm
9. Coding an Algorithm

2.1 The Analysis Framework

1. Measuring an Input’s Size
2.Units for Measuring Running time
3. Orders of Growth
4. Worst-Case, Best-Case, and Average-Case Efficiencies

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

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

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

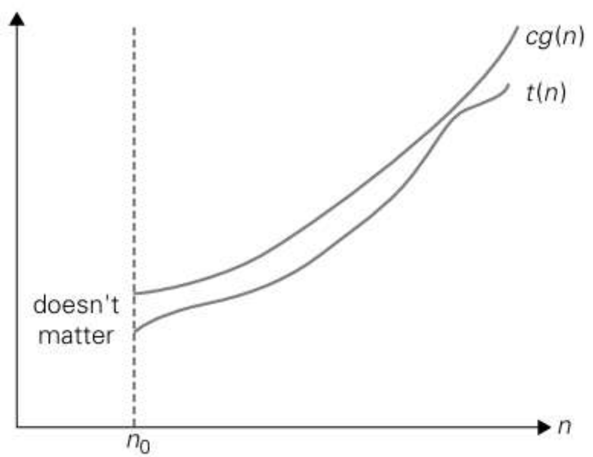

FIGURE 2.1 Big-oh notation: $t(n) \in O(g(n))$.

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.

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.

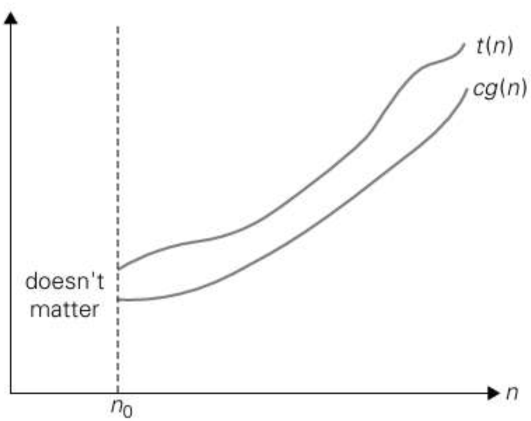

FIGURE 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$.

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 $n_0 = 0$.

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.

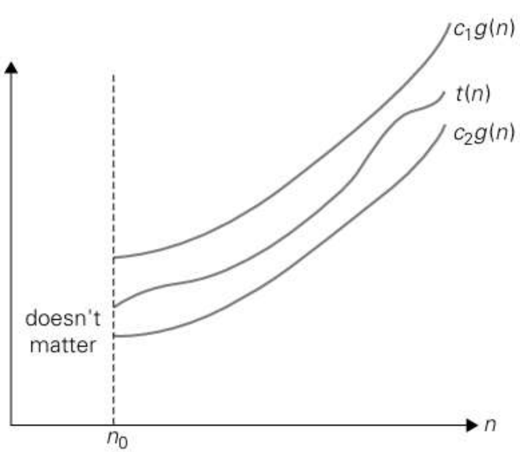

FIGURE 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$.

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 $n_0 = 2$.

Useful Property Involving the Asymptotic Notations

TABLE 2.2 Basic asymptotic 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.

Mathematical Analysis of Non recursive Algorithms

General Plan for Analyzing the Time Efficiency of Nonrecursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation. (As a rule, it is located in the innermost loop.)
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.[4]
5. Using standard formulas and rules of sum manipulation, either find a closed-form formula for the count or, at the very least, establish its order of growth.

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or, at least, ascertain the order of growth of its solution.

Brute Force and Exhaustive Search

Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

Selection Sort

We start selection sort by scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list. Then we scan the list, starting with the second element, putting the second smallest element in its final position. Generally, on the ith pass through the list, which we number from 0 to n-2, the algorithm searches for the last n-I elements and swaps it with Ai:

$$A_0 \le A_1 \le \ldots\ldots \le A_{i-1} \mid A_i \ldots\ldots A_{min} \ldots\ldots A_{n-1}$$

in their final positions the last n-i elements

After n-1 passes, the list is sorted.

ALGORITHM *SelectionSort*($A[0..n-1]$)
//Sorts a given array by selection sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in ascending order
for $i \leftarrow 0$ **to** $n-2$ **do**
$min \leftarrow i$
for $j \leftarrow i+1$ **to** $n-1$ **do**
if $A[j] < A[min]$ $min \leftarrow j$
swap $A[i]$ and $A[min]$

| 89   45   68   90   29   34   **17**

17 | 45   68   90   **29**   34   89

17   29 | 68   90   45   **34**   89

17   29   34 | 90   **45**   68   89

17   29   34   45 | 90   **68**   89

17   29   34   45   68 | 90   **89**

17   29   34   45   68   89 | 90

FIGURE 3.1 Example of sorting with selection sort. Each line corresponds to one iteration of the algorithm, i.e., a pass through the list's tail to the right of the vertical bar; an element in bold indicates the smallest element found. Elements to the left of the vertical bar are in their final positions and are not considered in this and subsequent iterations.

The number of times the algorithm executed depends only on the array's size and is given by

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i).$$

After solving using summation formulas

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}.$$

Thus selection sort has a Θ(n2) time complexity.

Bubble Sort

Another brute-force application to the sorting problem is to compare adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, we end up "bubbling up" the largest element to the last position on the list. The next pass bubbles up the second largest element, and so on, until after $n - 1$ passes the list is sorted. Pass i ($0 \le i \le n - 2$) of bubble sort can be represented by the following diagram:

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid \underbrace{A_{n-i} \le \cdots \le A_{n-1}}_{\text{in their final positions}}$$

Here is pseudocode of this algorithm.

ALGORITHM $BubbleSort(A[0..n-1])$

//Sorts a given array by bubble sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow 0$ **to** $n-2-i$ **do**
        if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

$$
\begin{array}{ccccccccccccc}
89 & \overset{?}{\leftrightarrow} & 45 & & 68 & & 90 & & 29 & & 34 & & 17 \\
45 & & 89 & \overset{?}{\leftrightarrow} & 68 & & 90 & & 29 & & 34 & & 17 \\
45 & & 68 & & 89 & \overset{?}{\leftrightarrow} & 90 & \overset{?}{\leftrightarrow} & 29 & & 34 & & 17 \\
45 & & 68 & & 89 & & 29 & & 90 & \overset{?}{\leftrightarrow} & 34 & & 17 \\
45 & & 68 & & 89 & & 29 & & 34 & & 90 & \overset{?}{\leftrightarrow} & 17 \\
45 & & 68 & & 89 & & 29 & & 34 & & 17 & | & 90 \\
\\
45 & \overset{?}{\leftrightarrow} & 68 & \overset{?}{\leftrightarrow} & 89 & \overset{?}{\leftrightarrow} & 29 & & 34 & & 17 & | & 90 \\
45 & & 68 & & 29 & & 89 & \overset{?}{\leftrightarrow} & 34 & & 17 & | & 90 \\
45 & & 68 & & 29 & & 34 & & 89 & \overset{?}{\leftrightarrow} & 17 & | & 90 \\
45 & & 68 & & 29 & & 34 & & 17 & | & 89 & & 90
\end{array}
$$

etc.

FIGURE 3.2 First two passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17. A new line is shown after a swap of two elements is done. The elements to the right of the vertical bar are in their final positions and are not considered in subsequent iterations of the algorithm.

The number of key comparisons for the bubble-sort version given above is the same for all arrays of size n; it is obtained by a sum that is almost identical to the sum for selection sort

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1]$$

$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

The number of key swaps, however, depends on the input. In the worst case of decreasing arrays, it is the same as the number of key comparisons:

$$S_{worst}(n) = C(n) = \frac{(n-1)n}{2} \in \Theta(n^2).$$

Sequential Search

This is also called as Linear search. Here we start from the initial element of the array and compare it with the search key. We repeat the same with all the elements of the array till we encounter the search key or till we reach end of the array.

ALGORITHM *SequentialSearch2*$(A[0..n], K)$

//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in $A[0..n-1]$ whose value is
// equal to K or -1 if no such element is found

$A[n] \leftarrow K$
$i \leftarrow 0$
while $A[i] \neq K$ **do**
    $i \leftarrow i + 1$
if $i < n$ **return** i
else return -1

The time efficiency in worst case is O(n), where n is the number of elements of the array. In best case it is O(1), it means the very first element is the search key.

Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

1.4.1 Selection sort

We start selection sort by scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list. Then we scan the list, starting with the second element, putting the second smallest element in its final position. Generally, on the *i*th pass through the list, which we number from 0 to n-2, the algorithm searches for the last n-I elements and swaps it with $A_{i:}$

$$A_0 \leq A_1 \leq \ldots\ldots \leq A_{i-1} \mid A_i \ldots\ldots A_{min} \ldots\ldots A_{n-1}$$

in their final positions the last n-i elements

After n-1 passes, the list is sorted.

ALGORITHM *SelectionSort*($A[0..n-1]$)
//Sorts a given array by selection sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in ascending order
for $i \leftarrow 0$ **to** $n-2$ **do**
    $min \leftarrow i$
        for $j \leftarrow i+1$ **to** $n-1$ **do**
            if $A[j] < A[min]$ $min \leftarrow j$
    swap $A[i]$ and $A[min]$

The number of times the algorithm executed depends only on the array's size and is given by

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i).$$

After solving using summation formulas

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}.$$

Thus selection sort has a Θ(n2) time complexity.

Bubble Sort

Another brute-force application to the sorting problem is to compare adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, we end up “bubbling up” the largest element to the last position on the list. The next pass bubbles up the second largest element, and so on, until after $n - 1$ passes the list is sorted. Pass i $(0 \le i \le n - 2)$ of bubble sort can be represented by the following diagram:

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid \underbrace{A_{n-i} \le \cdots \le A_{n-1}}_{\text{in their final positions}}$$

ALGORITHM *BubbleSort*$(A[0..n-1])$

//Sorts a given array by bubble sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow 0$ **to** $n-2-i$ **do**
        if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

Sequential Search

We have already encountered a brute-force algorithm for the general searching problem: it is called sequential search (see Section 2.1). To repeat, the algorithm simply compares successive elements of a given list with a given search key until either a match is encountered (successful search) or the list is exhausted without finding a match (unsuccessful search). A simple extra trick is often employed in implementing sequential search: if we append the search key to the end of the list, the search for the key will have to be successful, and therefore we can eliminate the end of list check altogether. Here is pseudocode of this enhanced version.

ALGORITHM *SequentialSearch2*$(A[0..n], K)$

//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in $A[0..n-1]$ whose value is
//   equal to K or -1 if no such element is found

$A[n] \leftarrow K$
$i \leftarrow 0$
while $A[i] \neq K$ **do**
        $i \leftarrow i + 1$
if $i < n$ **return** i
else return -1

Brute-Force String Matching

Recall the string-matching problem introduced in Section 1.3: given a string of n characters called the ***text*** and a string of m characters ($m \le n$) called the ***pattern***, find a substring of the text that matches the pattern. To put it more precisely, we want to find i—the index of the leftmost character of the first matching substring in the text—such that $t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$:

$$
\begin{array}{ccccccccccc}
t_0 & \ldots & t_i & \ldots & t_{i+j} & \ldots & t_{i+m-1} & & \ldots & t_{n-1} & \text{text } T \\
 & & \updownarrow & & \updownarrow & & \updownarrow & & & & \\
 & & p_0 & \ldots & p_j & \ldots & p_{m-1} & \text{pattern } P & & &
\end{array}
$$

If matches other than the first one need to be found, a string-matching algorithm can simply continue working until the entire text is exhausted.

ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

//Implements brute-force string matching
//Input: An array $T[0..n-1]$ of n characters representing a text and
//   an array $P[0..m-1]$ of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//   matching substring or -1 if the search is unsuccessful

for $i \leftarrow 0$ **to** $n-m$ **do**
    $j \leftarrow 0$
    while $j < m$ **and** $P[j] = T[i+j]$ **do**
        $j \leftarrow j+1$
    if $j = m$ **return** i
return -1

N	O	B	O	D	Y	_	N	O	T	I	C	E	D	_	H	I	M
N	**O**	**T**															
	N	O	T														
		N	O	T													
			N	O	T												
				N	O	T											
					N	O	T										
						N	O	T									
							N	**O**	**T**								

FIGURE 3.3 Example of brute-force string matching. The pattern's characters that are compared with their text counterparts are in bold type.