DESIGN AND ANALYSIS OF ALGORITHMS

Dr.Rashmi S

Associate Professor,Dept of ISE

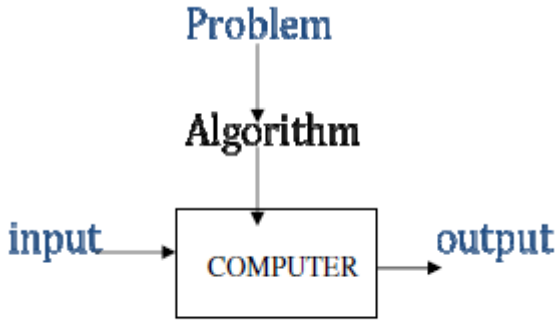
Dayananda Sagar College of Engineering

Module -1 Introduction

1. What is an ALGORITHM

An Algorithm is a sequence of umambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

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 produced.
4. Finiteness. If we trace out the instruction 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 principal, 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.



Notion of an algorithm.

The same algorithm can be represented in several ways. Several algorithms can be used to solve the same problem. • Different ideas have different speed.

Example: Problem:GCD of Two numbers m,n

Euclids algorithm
Step1:if n=0 return value of m & stop else proceed step 2
Step 2:Divide m by n & assign the value of remainder to r
Step 3:Assign the value of n to m ,r to n,Go to step1.

Algorithm Euclid(m,n)
//input:2 nonnegative, not-both-zero integers
//output:GCD of m and n
While n ≠ 0 do
    r ← m mod n
    m ← n
    n ← r
return m

Another algorithm to solve the same problem

Consecutive integer checking algorithm gcd(m,n)

Step1:Assign the value of min(m,n) to t
Step 2:Divide m by t.if remainder is 0,go to step3 else goto step4
Step 3: Divide n by t.if the remainder is 0,return the value of t as the answer and stop,otherwise proceed to step4
Step4 :Decrease the value of t by 1. go to step 2

Middle-school procedure for computing gcd(m,n)
Step 1:Find the prime factors of m
Step 2:Find the prime factors of n
Step 3: Identify all the common factors found in Step 1&2
Step 4: Compute the product of all common factors and return as the gcd

FUNDAMENTALS OF ALGORITHMIC PROBLEM SOLVING

A sequence of steps involved in designing and analyzing an algorithm is shown in the figure below.

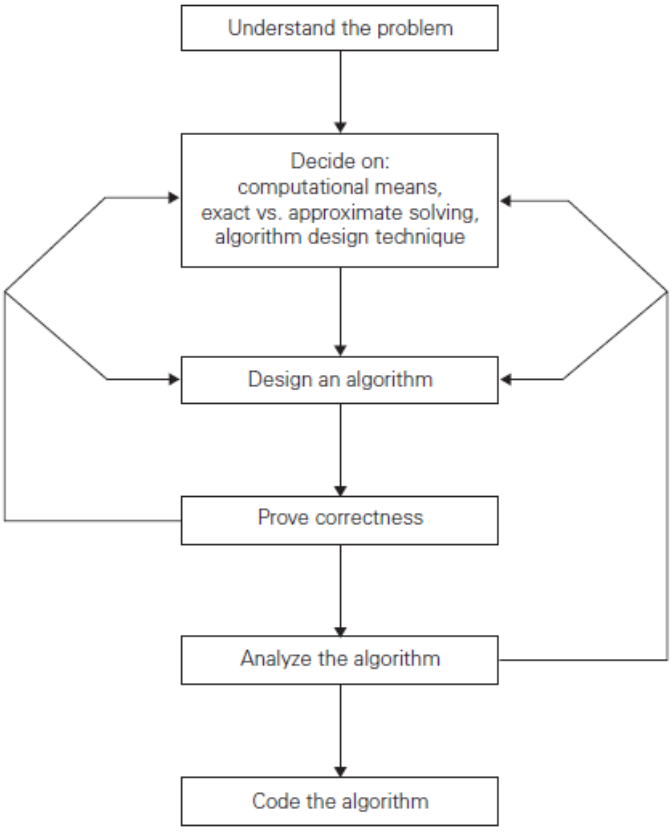


FIGURE 1.2 Algorithm design and analysis process.

(1) Understanding the Problem

- This is the first step in designing of algorithm.
- Read the problem's description carefully to understand the problem statement completely.
- Ask questions for clarifying the doubts about the problem.
- Identify the problem types and use existing algorithm to find solution.
- Input (instance) to the problem and range of the input get fixed.

(2) Ascertaining the Capabilities of the Computational Device

- In random-access machine (RAM), instructions are executed one after another. Accordingly, algorithms designed to be executed on such machines are called sequential algorithms.
- In some newer computers, operations are executed concurrently, i.e., in parallel. Algorithms that take advantage of this capability are called parallel algorithms.
- Choice of computational devices like Processor and memory is mainly based on space and time efficiency

(3) Choosing between Exact and Approximate Problem Solving

- The next principal decision is to choose between solving the problem exactly or solving it approximately.
- An algorithm used to solve the problem exactly and produce correct result is called an exact algorithm.
- If the problem is so complex and not able to get exact solution, then we have to choose an algorithm called an approximation algorithm. i.e., produces an approximate answer. E.g., extracting square roots, solving nonlinear equations, and evaluating definite integrals.

(4) Algorithm Design Techniques

- An algorithm design technique (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.
- Algorithms+ Data Structures = Programs
- Though Algorithms and Data Structures are independent, but they are combinedtogether to develop program. Hence the choice of proper data structure is required before designing the algorithm.
- Implementation of algorithm is possible only with the help of Algorithms and DataStructures
- Algorithmic strategy / technique / paradigm are a general approach by which many problems can be solved algorithmically. E.g., Brute Force, Divide and Conquer, Dynamic Programming, Greedy Technique and so on.

(5) Methods of Specifying an Algorithm

There are three ways to specify an algorithm. They are:

a. Natural language
b. Pseudocode
c. Flowchart

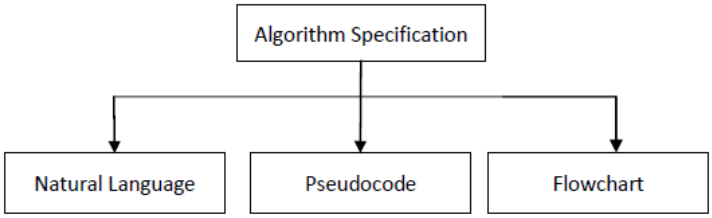


FIGURE 1.3 Algorithm Specifications

Pseudocode and flowchart are the two options that are most widely used nowadays for specifying algorithms.

a. **Natural Language**

It is very simple and easy to specify an algorithm using natural language. But many times specification of algorithm by using natural language is not clear and thereby we get brief specification.

Example: An algorithm to perform addition of two numbers.

```
Step 1: Read the first number, say a.
Step 2: Read the first number, say b.
Step 3: Add the above two numbers and store the result in c.
Step 4: Display the result from c.
```

Such a specification creates difficulty while actually implementing it. Hence many programmers prefer to have specification of algorithm by means of Pseudocode.

b. **Pseudocode**

- Pseudocode is a mixture of a natural language and programming language constructs.

Pseudocode is usually more precise than natural language.

- For Assignment operation left arrow "←", for comments two slashes "//",if condition, for, while loops are used.

```
ALGORITHM Sum(a,b)
//Problem Description: This algorithm performs addition of two numbers
//Input: Two integers a and b
//Output: Addition of two integers
c←a+b
return c
```

This specification is more useful for implementation of any language.

c. **Flowchart**

In the earlier days of computing, the dominant method for specifying algorithms was a flowchart, this representation technique has proved to be inconvenient. Flowchart is a graphical representation of an algorithm. It is a a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.

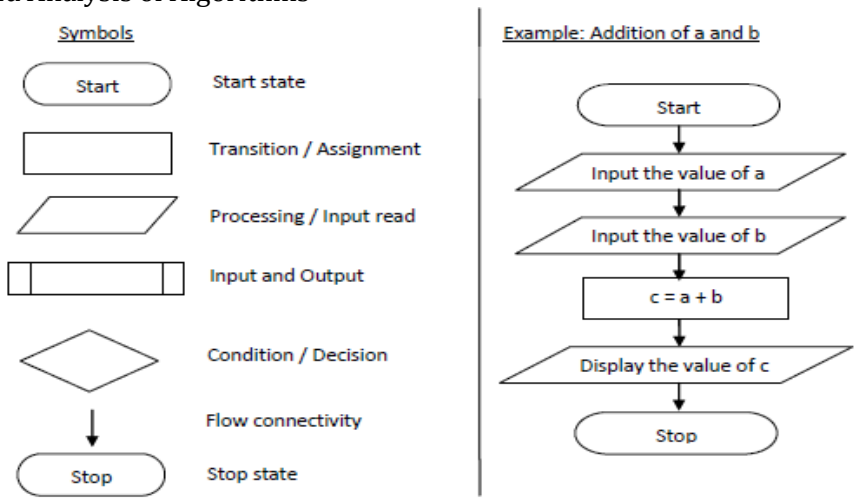


FIGURE 1.4 Flowchart symbols and Example for two integer addition.

(6) Proving an Algorithm's Correctness

- Once an algorithm has been specified then its correctness must be proved.
- An algorithm mustyields a required result for every legitimate input in a finite amount of time.
- For example, the correctness of Euclid's algorithm for computing the greatest common divisor stems from the correctness of the equality gcd(m, n) = gcd(n, m mod n).
- A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs.

(7) Analyzing an Algorithm

- For an algorithm the most important is efficiency. In fact, there are two kinds of algorithm efficiency. They are:
  - Time efficiency, indicating how fast the algorithm runs, and
  - Space efficiency, indicating how much extra memory it uses.

factors to analyze an algorithm are:

- Time efficiency of an algorithm
- Spaceefficiency of an algorithm
- Simplicity of an algorithm
- Generality of an algorithm

(8) Coding an Algorithm

- The coding / implementation of an algorithm is done by a suitable programming languagelike C, C++, JAVA.
- The transition from an algorithm to a program can be done either incorrectly or very inefficiently. Implementing an algorithm correctly is necessary. The Algorithm power should not reduced by inefficient implementation.
- Standard tricks like computing a loop's invariant (an expression that does not change its value) outside the loop, collecting common subexpressions, replacing expensive operations by cheap ones, selection of programming language and so on should be known to the programmer.

FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

The efficiency of an algorithm can be in terms of time and space. The algorithm efficiency can be analyzed by the following ways.

a. Analysis Framework.
b. Asymptotic Notations and its properties.
c. Mathematical analysis for Recursive algorithms.
d. Mathematical analysis for Non-recursive algorithms.

a. Analysis Framework

There are two kinds of efficiencies to analyze the efficiency of any algorithm. They are:

- Time efficiency, indicating how fast the algorithm runs, and
- Space efficiency, indicating how much extra memory it uses.

The algorithm analysis framework consists of the following:

- Measuring an Input's Size
- Units for Measuring Running Time
- Orders of Growth
- Worst-Case, Best-Case, and Average-Case Efficiencies

(i) Measuring an Input's Size

- An algorithm's efficiency is defined as a function of some parameter n indicating thealgorithm's input size.

For example, it will be the size of the list for problems of sorting, searching.

- For the problem of evaluating a polynomial $p(x) = a_n x^n + \ldots + a_0$ of degree n, the size of the parameter will be the polynomial's degree or the number of its coefficients, which is larger by 1 than its degree.
- In computing the product of two n × n matrices, the choice of a parameter indicating aninput size does matter.
- Inmeasuring input size for algorithms solving problems such as checking primality of a positive integer n. the input is just one number and it is this number's magnitude that determines the input size. In such situations, it is preferable to measure size by the number b of bits in the n's binary representation b=(log2 n)+1.

(ii) Units for Measuring Running Time

Some standard unit of time measurement such as a second, or millisecond, and so on can be used to measure the running time of a program after implementing the algorithm.

Drawbacks

- Dependence on the speed of a particular computer.
- Dependence on the quality of a program implementing the algorithm.
- The compiler used in generating the machine code.
- The difficulty of clocking the actual running time of the program.

So, we need metric to measure an algorithm's efficiency that does not depend on these extraneous factors. One possible approach is to count the number of times each of the algorithm's operations is executed. This approach is excessively difficult. The most important operation (+, -, *, /) of the algorithm, called the **basic operation**. Computing the number of times the basic operation is executed is easy**. The total running time is determined by basic operations count**.

(iii) Orders of Growth

For large values of n, it is the function's order of growth that counts just like the Table 1.1, which contains values of a few functions particularly important for analysis of algorithms

TABLE 1.1 Values (approximate) of several functions important for analysis of algorithms

n	$\sqrt{n}$	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
1	1	0	1	0	1	1	2	1
2	1.4	1	2	2	4	4	4	2
4	2	2	4	8	16	64	16	24
8	2.8	3	8	$2.4 \cdot 10^1$	64	$5.1 \cdot 10^2$	$2.6 \cdot 10^2$	$4.0 \cdot 10^4$
10	3.2	3.3	10	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
16	4	4	16	$6.4 \cdot 10^1$	$2.6 \cdot 10^2$	$4.1 \cdot 10^3$	$6.5 \cdot 10^4$	$2.1 \cdot 10^{13}$
10^2	10	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	31	10	10^3	$1.0 \cdot 10^4$	10^6	10^9	Very big computation	
10^4	10^2	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	$3.2 \cdot 10^2$	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	10^3	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

(iv) Worst-Case, Best-Case, and Average-Case Efficiencies

its upper bound, fails.

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element in A that matches K
//        or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

Worst-case efficiency

- The worst-case efficiency of an algorithm is its efficiency for the worst case input of size n.
- The algorithm runs the longest among all possible inputs of that size.
- For the input of size n, the runningtime is Cworst(n) = n.

Best case efficiency

- The best-case efficiency of an algorithm is its efficiency for the best case input of size n.
- The algorithm runs the fastest among all possible inputs of that size n.
- In sequential search, If we search a first element in list of size n. (i.e. first element equal to a search key), then the running time is Cbest(n) = 1

Average case efficiency

- The Average case efficiency lies between best case and worst case.
- To analyze the algorithm's average caseefficiency, we must make some assumptions about possible inputs of size n.
- The standard assumptions are that
  - The probability of a successful search is equal to p ($0 \le p \le 1$) and
  - The probability of the first match occurring in the ith position of the list is the same for every i.

Asymptotic and Basic Efficiency Classes

Asymptotic Notations

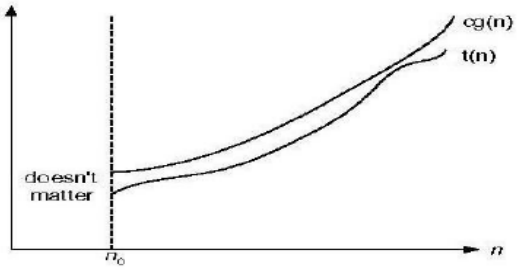
Asymptotic notation is a way of comparing functions that ignores constant factors and small input sizes. Three notations used to compare orders of growth of an algorithm's basic operation count are:**O, Ω, Θ notations.**

Big Oh- O notation

Definition:

A function t(n) is said to be in O(g(n)), denoted **t(n) € 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 n0 such that

$t(n) \le cg(n)$ for all $n \ge n0$



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

Ex:

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

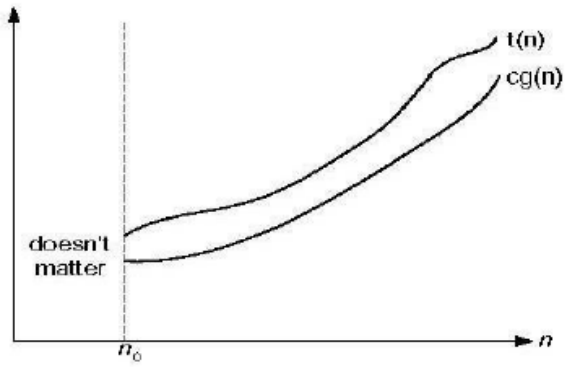
$$n^3 \notin O(n^2), \qquad 0.00001n^3 \notin O(n^2), \qquad n^4 + n + 1 \notin O(n^2).$$

Big Omega- Ω notation

Definition:

A function t (n) is said to be in Ω (g(n)), denoted **t(n) € Ω (g (n)),** if t (n) is bounded below by some constant multiple of g (n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n0 such that

$t(n) \ge cg(n)$ for all $n \ge n0$



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

Ex: -n^3€ Ω $\mathbf{O(n^2)}$, 100n+5 € Ω(n^2)

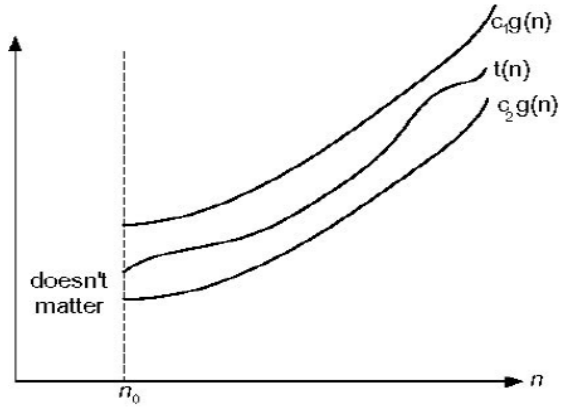
$n^3 \in \Omega(n^2)$, $\frac{1}{2}n(n-1) \in \Omega(n^2)$, but $100n + 5 \notin \Omega(n^2)$.

Big Theta- Θ notation

Definition:

A function t (n) is said to be in Θ(g (n)), denoted **t(n)** □ **Θ(g (n)),** if t (n) is bounded both above and below by some constant multiple of g (n) for all large n, i.e., if there exist some positive constant c1 and c2 and some nonnegative integer n0 such that **c2 g (n) ≤ t (n) ≤ c1 g (n) for all n ≥ n0**

ex: ½ n(n-1)€ Θ(n^2)



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

FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

Analysis of algorithms means to investigate an algorithm's efficiency with respect to resources:

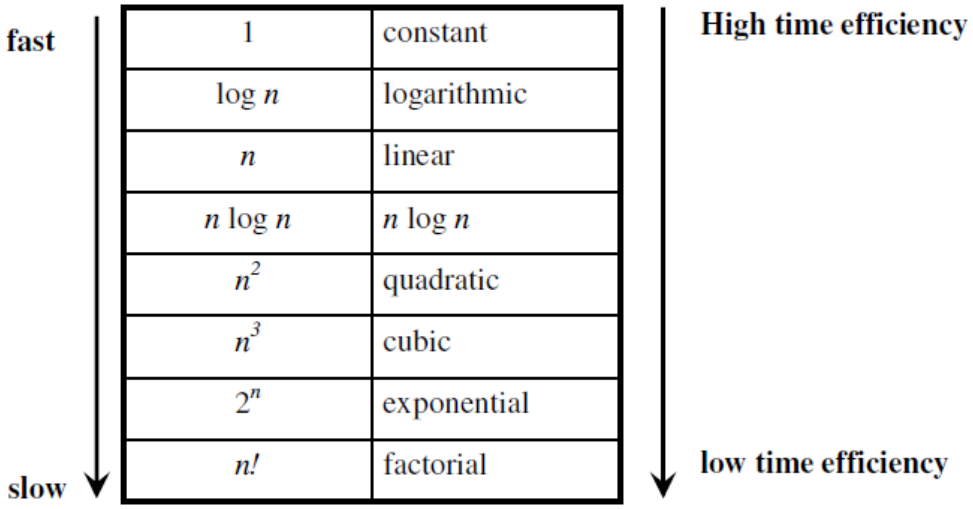
- **running time (time efficiency)**
- **memory space (space efficiency)**

Time being more critical than space, we concentrate on Time efficiency of algorithms. The theory developed, holds good for space complexity also.

Experimental Studies: requires writing a program implementing the algorithm and running the program with inputs of varying size and composition. It uses a function, like the built-in clock() function, to get an accurate measure of the actual running time, then analysis is done by plotting the results.

Basic Efficiency classes

The time efficiencies of a large number of algorithms fall into only a few classes.



Useful Property Involving the Asymptotic Notations

THEOREM If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

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

(The analogous assertions are true for the Ω and Θ notations as well.)

PROOF The proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, a_2, b_2: if $a_1 \le b_1$ and $a_2 \le b_2$, then $a_1 + a_2 \le 2\max\{b_1, b_2\}$.

Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some non-negative integer n_1 such that

$$t_1(n) \le c_1 g_1(n) \quad \text{for all } n \ge n_1.$$

Similarly, since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \le c_2 g_2(n) \quad \text{for all } n \ge n_2.$$

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \ge \max\{n_1, n_2\}$ so that we can use both inequalities. Adding them yields the following:

$$\begin{aligned} t_1(n) + t_2(n) &\le c_1 g_1(n) + c_2 g_2(n) \\ &\le c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)] \\ &\le c_3 2\max\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$, with the constants c and n_0 required by the O definition being $2c_3 = 2\max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively. ■

So what does this property imply for an algorithm that comprises two consecutively executed parts? It implies that the algorithm's overall efficiency is determined by the part with a higher order of growth, i.e., its least efficient part:

$$\left.\begin{array}{|l} t_1(n) \in O(g_1(n)) \\ t_2(n) \in O(g_2(n)) \end{array}\right\} \quad t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

Using Limits for Comparing Orders of Growth

Three principal cases may arise:

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n).^3 \end{cases}$$

Note that the first two cases mean that $t(n) \in O(g(n))$, the last two mean that $t(n) \in \Omega(g(n))$, and the second case means that $t(n) \in \Theta(g(n))$.

The limit-based approach is often more convenient than the one based on the definitions because it can take advantage of the powerful calculus techniques developed for computing limits, such as L'Hôpital's rule

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$$

and Stirling's formula

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \quad \text{for large values of } n.$$

Here are three examples of using the limit-based approach to comparing orders of growth of two functions.

EXAMPLE 1 Compare the orders of growth of $\frac{1}{2}n(n-1)$ and n^2. (This is one of the examples we used at the beginning of this section to illustrate the definitions.)

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

Since the limit is equal to a positive constant, the functions have the same order of growth or, symbolically, $\frac{1}{2}n(n-1) \in \Theta(n^2)$. ■

EXAMPLE 2 Compare the orders of growth of $\log_2 n$ and $\sqrt{n}$. (Unlike Example 1, the answer here is not immediately obvious.)

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

EXAMPLE 3 Compare the orders of growth of $n!$ and 2^n. (We discussed this informally in Section 2.1.) Taking advantage of Stirling's formula, we get

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

Thus, though 2^n grows very fast, $n!$ grows still faster. We can write symbolically that $n! \in \Omega(2^n)$; note, however, that while the big-Omega notation does not preclude the possibility that $n!$ and 2^n have the same order of growth, the limit computed here certainly does. ■

MATHEMATICAL ANALYSIS (TIME EFFICIENCY) OF NON-RECURSIVE ALGORITHMS

General plan for analyzing efficiency of non-recursive algorithms:

1. Decide on parameter n indicating **input size**
2. Identify algorithm's **basic operation**
3. Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, investigate **worst, average, and best case efficiency** separately.
4. Set up **summation** for C(n) reflecting the number of times the algorithm's basic operation is executed.
5. Simplify summation using standard formulas and establish the order of growth

ALOGORITHM *MaxElement*(A[*0..n-1*])

//Determines the value of largest element in a given array
//Input: An array A[0..n-1] of real numbers
//Output: The value of the largest element in A
Max =*A*[0]
for *i* = 1 to *n* -1 do
if *A*[*i*] > *Max*
    Max = *A*[*i*]

return *Max*

Analysis:
1. Input size: number of elements = n (size of the array)
2. Basic operation:
a) Comparison
b) Assignment
3. NO best, worst, average cases.
4. Let C (*n*) denotes number of comparisons: Algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bound between 1 and *n* – 1.

$$C(n) = \sum_{i=1}^{n-1} 1$$

5. **Simplify summation** using standard formulas

$$C(n) = \sum_{i=1}^{n-1} 1 \quad \begin{matrix} 1+1+1+\ldots+1 \\ [(n-1) \text{ number of times}] \end{matrix}$$

$$C(n) = n-1$$
$$C(n) \in \Theta(n)$$

Example: Element uniqueness problem
Algorithm *UniqueElements (A[0..n-1])*
//Checks whether all the elements in a given array are distinct
//Input: An array A[0..n-1]
//Output: Returns true if all the elements in A are distinct and false otherwise
for i= 0 to n - 2 do
for j = i + 1 to n – 1 do
if A[i] = = A[j] return ***false***
return ***true***
Analysis
1. Input size: number of elements = n (size of the array)
2. Basic operation: Comparison
3. Best, worst, average cases EXISTS.
Worst case input is an array giving largest comparisons.
• Array with no equal elements
• Array with last two elements are the only pair of equal elements
4. Let C (*n*) denotes number of comparisons in worst case: Algorithm makes one comparison for each repetition of the innermost loop i.e., for each value of the loop's variable j between its limits *i + 1 and n – 1*; and this is repeated for each value of the outer loop i.e, for each value of the loop's variable i between its limits *0 and n – 2*

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

5. **Simplify summation** using standard formulas

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

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

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

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

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

$$C(n) = (n-1)(n-1) - \frac{(n-2)(n-1)}{2}$$

$$C(n) = (n-1)\left((n-1) - \frac{(n-2)}{2}\right)$$

$$C(n) = (n-1)\frac{(2n-2-n+2)}{2}$$

$$C(n) = (n-1)(n)/2$$
$$= (n^2 - n)/2$$
$$= (n^2)/2 - n/2$$
$$\mathbf{C(n) \in \Theta(n^2)}$$

Example:

```
Algorithm Matrixmult(A[0..n-1,0..n-1], B[0..n-1,0..n-1])
For i= 0 to n-1 do
  For j= 0 to n-1 do
    C[i,j]=0.0
      For k= 0 to n-1 do
        C[i,j] = C[i,j] + A[i,k] * B[k,j]
Return C
```

$$M(n) = \sum_{I=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = n^3$$

MATHEMATICAL ANALYSIS (TIME EFFICIENCY) OF RECURSIVE ALGORITHMS

General plan for analyzing efficiency of recursive algorithms:
1. Decide on parameter n indicating **input size**
2. Identify algorithm's **basic operation**
3. Check whether the number of times the basic operation is executed depends only on the input size n. If it also depends on the type of input, investigate **worst, average, and best case efficiency** separately.
4. Set up **recurrence relation**, with an appropriate initial condition, for the number of times the algorithm's basic operation is executed.
5. **Solve** the recurrence or establish the order of growth

Example: Factorial function
ALGORITHM *Factorial (n)*
//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!
if $n == 0$ **return** 1
else return Factorial $(n-1) * n$

Analysis:
1. Input size: given number = n
2. Basic operation: multiplication
3. NO best, worst, average cases.
4. Let M (n) denotes number of multiplications.

M (n) = M $(n-1)$ + 1 for $n > 0$
M (0) = 0 *initial condition*

Where: M $(n-1)$: to compute Factorial $(n-1)$
1 :to multiply Factorial $(n-1)$ by n
5. Solve the recurrence: Solving using *"Backward substitution method":*
M (n) = M $(n-1)$ + 1
= [M $(n-2)$ + 1] + 1

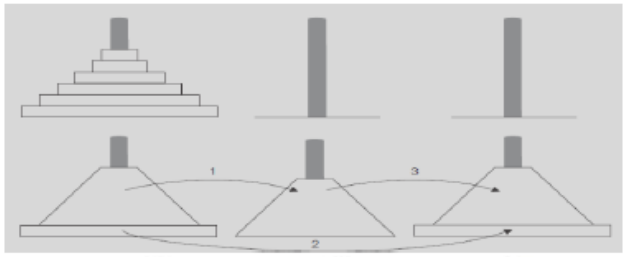
= M (*n* – 2) + 2
= [M (*n* – 3) + 1] + 3
= M (*n* – 3) + 3
…
In the ith recursion, we have
= M (*n* – *i*) + *i*
When i = n, we have
= M (*n* – *n*) + *n* = M (0) + *n*
Since M (0) = 0
= *n*

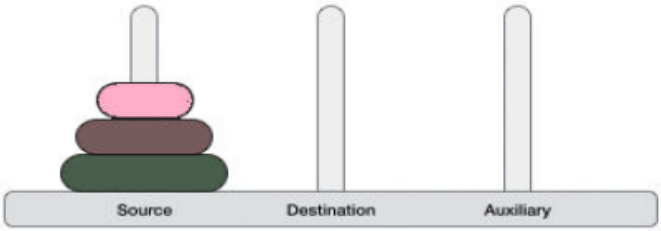
Example: Find the number of binary digits in the binary representation of a positive decimal integer
ALGORITHM *BinRec (n)*
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
if *n* = = 1 **return** 1
else return *BinRec* (⌊ n/2⌋) + 1

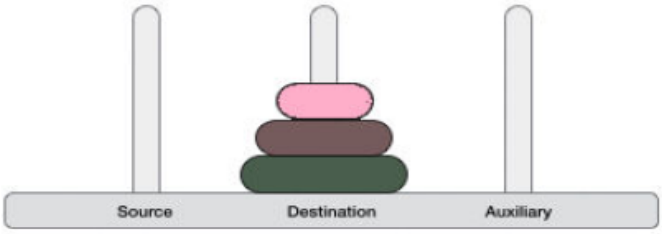
Analysis:
1. Input size: given number = n
2. Basic operation: addition
3. NO best, worst, average cases.
4. Let A (*n*) denotes number of additions.
A (*n*) = A (⌊ n/2⌋) + 1 for *n* > 1
A (1) = 0 *initial condition*
Where: A (⌊ n/2⌋) : to compute *BinRec* (⌊ n/2⌋)
1 : to increase the returned value by 1
5. Solve the recurrence:
A (*n*) = A (⌊ n/2⌋) + 1 for *n* > 1

Assume n = 2^k (smoothness rule)
A (2^k) = A (2^{k-1}) + 1 for *k* > 0; A (20) = 0
Solving using "Backward substitution method":
A (2^k) = A (2^{k-1}) + 1
= [A (2^{k-2}) + 1] + 1
= A (2^{k-2}) + 2
= [A (2^{k-3}) + 1] + 2
= A (2^{k-3}) + 3
…
In the ith recursion, we have
= A (2^{k-i}) + i
When i = k, we have
= A (2^{k-k}) + k = A (2^0) + k
Since A (2^0) = 0
A (2^k) = k
Since n = 2^k, HENCE $k = \log_2 n$
A (*n*) = $\log_2 n$
A (*n*) ∈ Θ (log n)

Example: Tower of Hanoi
We have n disks of different sizes that can slide onto any of three pegs. Consider A (source), B (auxiliary), and C (Destination). Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top. The goal is to move all the disks to the third peg, using the second one as an auxiliary.
Solution to Tower of Hanoi: To move n>1 disks from peg1 to peg3, first move recursively n-1 disks from peg1 to peg2, then move largest disk from peg1 to peg3,then move recursively n-1 disks from peg2 to peg3







ALGORITHM TOH(n, A, C, B)

//Move disks from source to destination recursively
//Input: *n disks and* 3 pegs A, B, and C
//Output: Disks moved to destination as in the source order.
if n=1
Move disk from A to C
else
Move top n-1 disks from A to B using C
TOH(n - 1, A, B, C)
Move top n-1 disks from B to C using A
TOH(n - 1, B, C, A)

Algorithm analysis

The number of moves $M(n)$ depends on n only, and we get the following recurrence equation for it: $M(n) = M(n-1) + 1 + M(n-1)$ for $n > 1$.

With the obvious initial condition $M(1) = 1$, we have the following recurrence relation for the number of moves $M(n)$:

$$M(n) = 2M(n-1) + 1 \text{ for } n > 1,$$
$$M(1) = 1.$$

We solve this recurrence by the same method of backward substitutions:

$M(n) = 2M(n-1) + 1$ sub. $M(n-1) = 2M(n-2) + 1$
$= 2[2M(n-2) + 1] + 1$
$= 2^2M(n-2) + 2 + 1$ sub. $M(n-2) = 2M(n-3) + 1$
$= 2^2[2M(n-3) + 1] + 2 + 1$
$= 2^3M(n-3) + 2^2 + 2 + 1$ sub. $M(n-3) = 2M(n-4) + 1$
$= 2^4M(n-4) + 2^3 + 2^2 + 2 + 1$
…
$= 2^iM(n-i) + 2^{i-1} + 2^{i-2} + \ldots + 2 + 1 = 2^iM(n-i) + 2^i - 1.$
…

Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$,

$M(n) = 2^{n-1}M(n-(n-1)) + 2^{n-1} - 1 = 2^{n-1}M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1.$

Thus, we have an exponential time algorithm

2. BRUTE FORCE

Introduction

Brute force is a straightforward approach to problem solving, usually directly based on the problem's statement and definitions of the concepts involved. For some important problems (e.g., sorting, searching, string matching), the brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size Even if too inefficient in general, a brute-force algorithm can still be useful for solving small-size instances of a problem. A brute-force algorithm can serve an important theoretical or educational purpose.

the brute-force approach should not be overlooked as an important algorithm design strategy.

- brute force is applicable to a very wide variety of problems.
- for some important problems—e.g., sorting, searching, matrix multiplication, string matching—the brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size.
- the expense of designing a more efficient algorithm may be unjustifiable if only a few instances of a problem need to be solved and a brute-force algorithm can solve those instances with acceptable speed.
- even if too inefficient in general, a brute-force algorithm can still be useful for solving small-size instances of a problem.
- a brute-force algorithm can serve an important theoretical or educational purpose as a yardstick with which to judge more efficient alternatives for solving a problem.

Sorting Problem Brute force approach to sorting

Problem:Given a list of n orderable items (e.g., numbers, characters from some alphabet, character strings), rearrange them in nondecreasing order.

Selection Sort

```
ALGORITHM SelectionSort(A[0..n - 1])
                //The algorithm 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 = 0 to n - 2 do
        min = i
        for j _=i + 1 to n - 1 do
                if A[j ]<A[min] min= j
                        swap A[i] and A[min]
```

Example:

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

Selection sort's operation on the list 89, 45, 68, 90, 29, 34, 17. 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.

Performance Analysis of the selection sort algorithm: The input's size is given by the number of elements n.

The algorithm's basic operation is the key comparison A[j]<A[min]. The number of times it is executed depends only on the array's size and is given by

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

Thus, selection sort is a O(n2) algorithm on all inputs. The number of key swaps is only O(n) or, more precisely, n-1 (one for each repetition of the i loop).This property distinguishes selection sort positively from many other sorting algorithms.

Bubble Sort

Compare adjacent elements of the list and exchange them if they are out of order.Then we repeat the process,By doing it repeatedly, we end up 'bubbling up' the largest element to the last position on the list ALGORITHM **BubbleSort(A[0..n - 1])**

```
//The algorithm sorts array A[0..n - 1] by bubble sort
//Input: An array A[0..n - 1] of orderable elements
//Output: Array A[0..n - 1] sorted in ascending order
for i =0 to n - 2 do
    for j = 0 to n - 2 - i do

        if A[j + 1]<A[j ]
         swap A[j ] and A[j + 1
```

Example

```
89 ?↔ 45    68    90    29    34    17
45    89 ?↔ 68    90    29    34    17
45    68    89 ?↔ 90 ?↔ 29    34    17
45    68    89    29    90 ?↔ 34    17
45    68    89    29    34    90 ?↔ 17
45    68    89    29    34    17   |90

45 ?↔ 68 ?↔ 89 ?↔ 29    34    17   |90
45    68    29    89 ?↔ 34    17   |90
45    68    29    34    89 ?↔ 17   |90
45    68    29    34    17   |89    90

          etc.
```

The first 2 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

Bubble Sort the analysis

Clearly, the outer loop runs n times. The only complexity in this analysis in the inner loop. If we think about a single time the inner loop runs, we can get a simple bound by noting that it can never loop more than n times. Since the outer loop will make the inner loop complete n times, the comparison can't happen more than O(n2) times. The number of key comparisons for the bubble sort version given above is the same for all arrays of size n

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

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

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

Observation: if a pass through the list makes no exchanges, the list has been sorted and we can stop the algorithm Though the new version runs faster on some inputs, it is still in O(n2) in the worst and average cases. Bubble sort is not very good for big set of input.How ever bubble sort is very simple to code.

General Lesson From Brute Force Approach

A first application of the brute-force approach often results in an algorithm that can be improved with a modest amount of effort.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).

Sequential Search

```
ALGORITHM SequentialSearch2(A[0..n], K)
//The algorithm implements sequential search with a search key
as a // sentinel
//Input: An array A of n elements and a search key K
//Output: The position 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] =K
I= 0
        A[i] = K do
                i = i + 1
                if i < n return i
        else return -1
```

Brute-Force String Matching

Given a string of n characters called the text and a string of m characters (m = 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

ti = p0, . . . , ti+j = pj , . . . , ti+m-1 = pm-1:

t0 . . . ti . . . ti+j . . . ti+m-1 . . . tn-1 text T

p0 . . . pj . . . pm-1 pattern P

1. Pattern: 001011

Text: 10010101101001100101111010

2. Pattern: happy

Text: It is never too late to have a happy childhood.

```
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
```

The algorithm shifts the pattern almost always after a single character comparison. in the worst case, the algorithm may have to make all m comparisons before shifting the pattern, and this can happen for each of the n - m + 1 tries.Thus, in the worst case, the algorithm is in θ(nm).