Analysis and Design of Algorithms

BCS401

Text Book: Introduction to the Design and Analysis of Algorithms,

By Anany Levitin, 3rd Edition (Indian), 2017, Pearson.

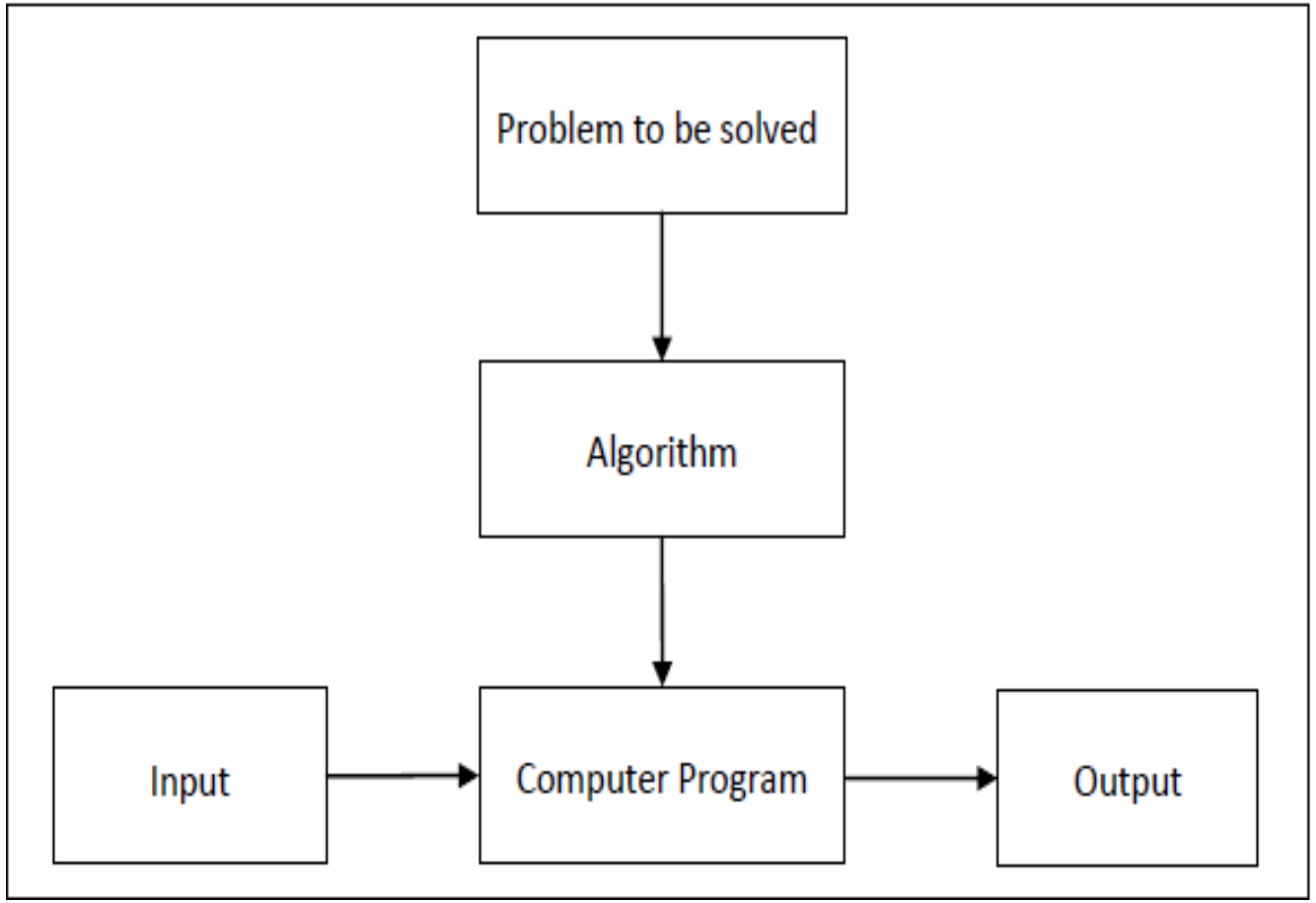
1.1 What Is an Algorithm?

- An ***algorithm is a sequence of unambiguous instructions for solving a problem, i.e.,*** *for* obtaining a required output for any legitimate input in a finite amount of time.

NOTION OF AN ALGORITHM

The notion of the algorithm illustrates some important points:

- The non-ambiguity requirement for each step of an algorithm cannot be compromised.
- The range of inputs for which an algorithm works has to be specified carefully.
- The same algorithm can be represented in several different ways.
- There may exist several algorithms for solving the same problem.
- Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.



Characteristics of an algorithm:

- **Input:** Zero / more quantities are externally supplied.
- **Output:** At least one quantity is produced.
- **Definiteness:** Each instruction is clear and unambiguous.
- **Finiteness:** If the instructions of an algorithm is traced then for all cases the algorithm must terminates after a finite number of steps.
- **Efficiency/Effectiveness:** Every instruction must be very basic and runs in short time.

Algorithm to find GCD of two numbers

This problem can be solved by using 3 different algorithms

1. Euclid's algorithm
2. Consecutive integer checking algorithm
3. Middle-school procedure

1. Euclid's algorithm

- It is based on applying repeatedly the equality **gcd*(m, n) = gcd(n, m mod n)*,** where *m mod n is the remainder of the division of m by n, until m mod n is equal to 0. Since* ***gcd(m,0) = m****, the last value of m is also the GCD of the initial m and n.*

Euclid's algorithm for computing gcd*(m, n) in simple steps:*

Input: Two positive numbers m &n

Output: Largest integer which divides m & n

- **Step 1:** If n = 0, return the value of m as the answer and stop; otherwise, proceed to Step 2.
- **Step 2:** Divide m by n and assign the value of the remainder to r.
- **Step 3:** Assign the value of n to m and the value of r to n. Go to Step 1.

gcd(6,10)

m	n	r=m%n
6	10	6
10	6	4
6	4	2
4	2	0
2	0	stop

When n=0 we can stop the iteration and gcd equal to m.
gcd(6,10) = 2

gcd(60,24)=?
(It required only 3 iterations)

Pseudo code: It is a mixture of a natural language and programming language constructs. Pseudocode is usually more precise than natural language.

Euclid's algorithm for computing gcd*(m, n) expressed in pseudocode*

ALGORITHM *Euclid_gcd(m, n)*
//Computes gcd*(m, n) by Euclid's algorithm*
//Input: Two nonnegative, not-both-zero integers *m and n*
//Output: Greatest common divisor of *m and n*
while *n ≠ 0 do*
    $r \leftarrow m \bmod n$
    $m \leftarrow n$
    $n \leftarrow r$
return *m*

2. Consecutive integer checking algorithm

Input: Two positive numbers m &n

Output: Largest integer which divides m & n

- **Step 1:** Assign the value of min{m, n} to t.
- **Step 2:** Divide m by t. If the remainder of this division is 0, go to Step 3; otherwise, go to Step 4.
- **Step 3:** Divide n by t. If the remainder of this division is 0, return the value of t as the answer and stop; otherwise, proceed to Step 4.
- **Step 4:** Decrease the value of t by 1. Go to Step 2.

gcd(6,10)=2

t	m%t	n%t
6	0	4
5	1	
4	2	
3	0	1
2	0	0

gcd(60,24)=?
(It required more than 3 iterations)

This method takes more time compared to Euclid's algorithm. (When m & n are large)

Drawback: It does not work correctly when one of its input numbers is zero.

3. Middle-school procedure

Input: Two positive numbers m &n

Output: Largest integer which divides 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 in the two prime expansions found in Step 1 and Step 2.
- **Step 4:** Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.

gcd(6,10)

- Factors of 6: 2*3
- Factors of 10: 5*2
- Common factor 2 the gcd is 2.

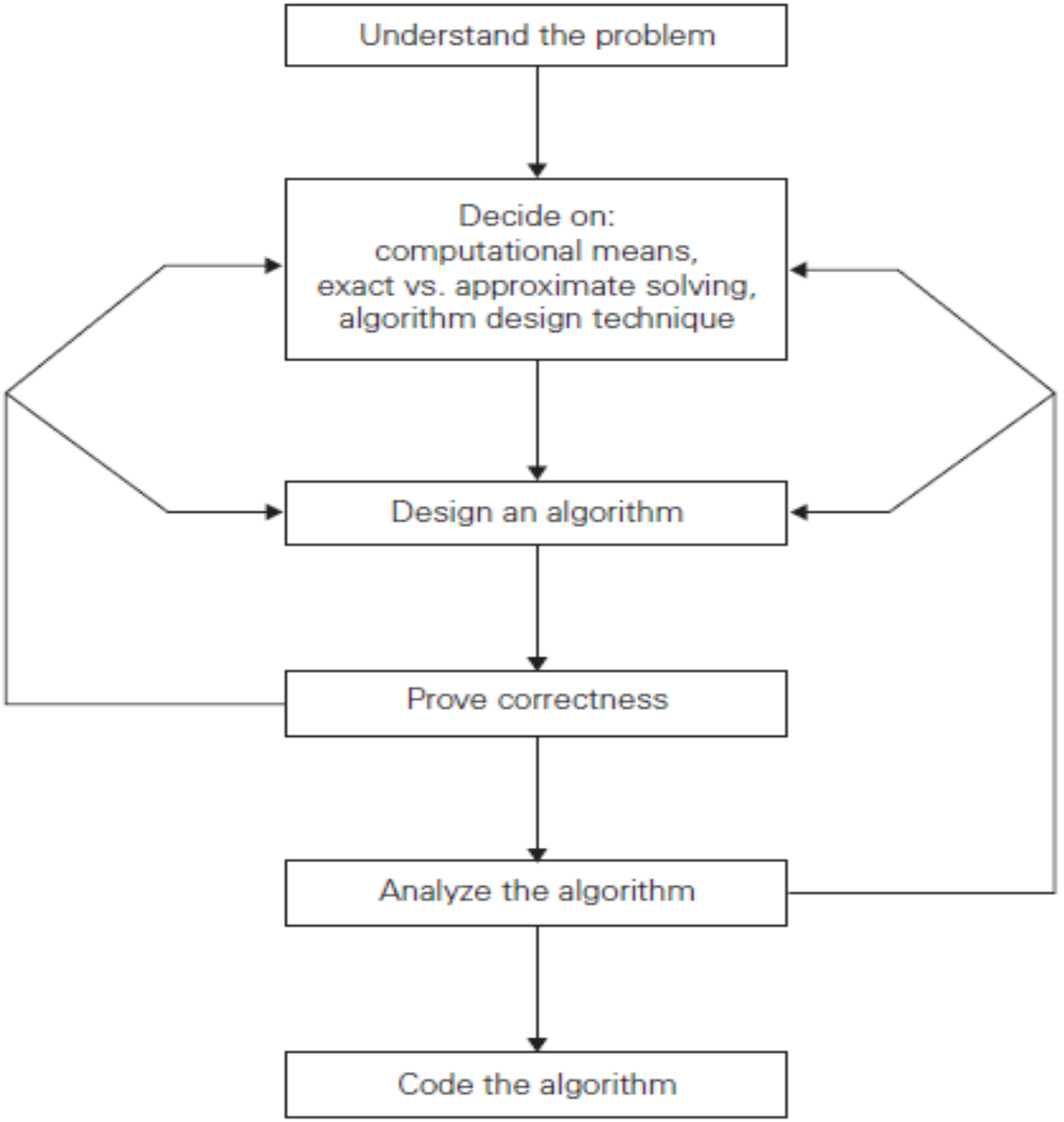
gcd(60,24)=?

Drawback: It is not clearly specified that how to find the prime factor (ambiguity)

This method fails when one of its input number is 1.

1.2 FUNDAMENTALS OF ALGORITHMIC PROBLEM SOLVING

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



(i) Understanding the Problem

- This is the first step in designing of algorithm.
- Read the problem's description carefully to **understand the problem** statement **completely.**
- **Clear all the doubts** about the problem.
- Check whether the problem **belongs to known category or not**.
  - ➢There are a few types of problems that arise in computing applications quite often.
  - ➢If the problem in question is one of them, you might be able to use a **known algorithm** for solving it.
  - ➢It helps to understand how such an algorithm works and to know its strengths and weaknesses, especially if you have to choose among several available algorithms.
  - ➢But often you will not find a readily available algorithm and will have to design your own.

- Identify the input (***instance) to the problem and range of the input get fixed.***
  - An input to an algorithm specifies an **instance** of the problem the algorithm solves.
  - It is very important to specify exactly the set of instances the algorithm needs to handle.
  - If you fail to do this, your algorithm may work correctly for a majority of inputs but crash on some “boundary” value.
  - Remember that a correct algorithm is not one that works most of the time, but one that works correctly for all legitimate inputs.

(ii) Decision making

(a) Ascertaining the Capabilities of the Computational Device

- In ***random-access machine (RAM), instructions are executed one after another (The*** central assumption is that one operation at a time). 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.***

(b) Choosing between Exact and Approximate Problem Solving

- The next 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.

(c) Designing an appropriate Data Structure.

- The choice of proper data structure is required before designing the algorithm.

(d) Algorithm Design Techniques

What is an algorithm design technique?

- An algorithm design technique is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.
- Learning these techniques is of utmost importance for the following reasons.
  - ➢First, they provide guidance for designing algorithms for new problems.
  - ➢Second, algorithms are the cornerstone of computer science.

E.g., Brute Force, Divide and Conquer, Dynamic Programming, Greedy Technique and so on.

(iii) Methods of Specifying an Algorithm

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

  a. Natural language

  b. Pseudocode

  c. Flowchart

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

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

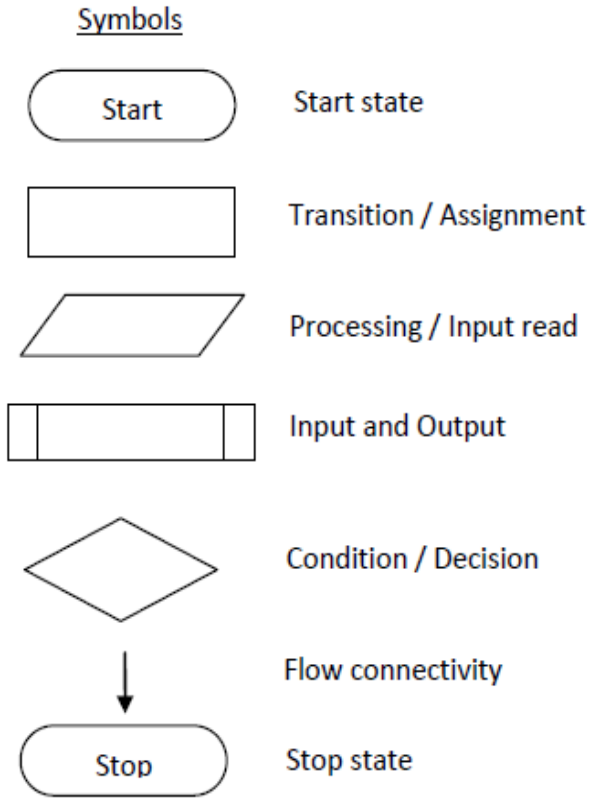
c←a+b

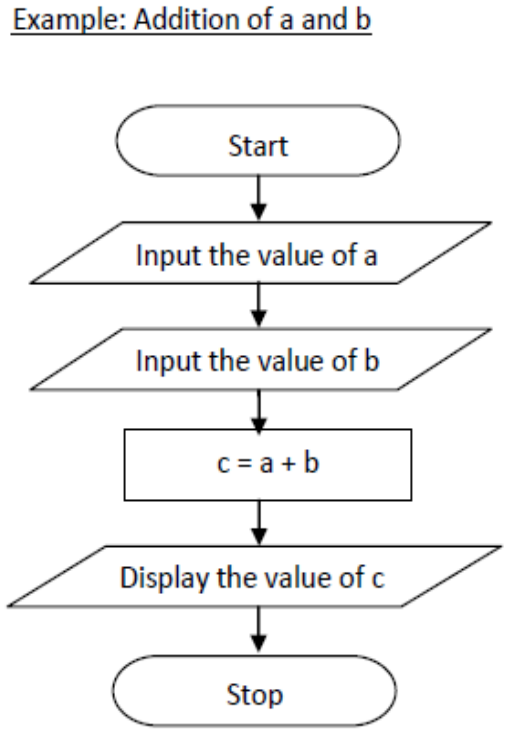
return c

- This specification is more useful for implementation of any language.

C. Flowchart

- It is a graphical representation of an algorithm.
- It is a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.





(iv) Proving an Algorithm's Correctness

- Once an algorithm has been specified then its ***correctness must be proved.***
- We have to prove that an algorithm must yields a required **result for every legitimate input in a finite amount of** time.
- For example, the correctness of Euclid's algorithm for computing the GCD stems from the correctness of the equality gcd*(m, n) = gcd(n, m mod n)*, The simple observation that the second integer gets smaller on every iteration of the algorithm, and the fact that the algorithm stops when the second integer becomes 0.
- 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.
- The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms. The error produced by the algorithm should not exceed a predefined limit.

(v) Analyzing an Algorithm

- For an algorithm the most important is efficiency. In fact, there are two kinds of algorithm efficiency. They are:

    1. **Time efficiency ,** indicating how fast the algorithm runs
    2. **Space efficiency,** indicating how much extra memory it uses.

- The efficiency of an algorithm is determined by measuring both time efficiency and space efficiency.
- So factors to analyze an algorithm are:
    - Time efficiency of an algorithm
    - Space efficiency of an algorithm
    - Simplicity of an algorithm
    - Range of input

(vi) Coding an Algorithm

- The coding / implementation of an algorithm is done by a suitable programming language like 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.
- Typically, such improvements can speed up a program only by a constant factor, whereas a better algorithm can make a difference in running time by orders of magnitude. But once an algorithm is selected, a 10–50% speedup may be worth an effort.
- It is very essential to write an optimized code (efficient code) to reduce the burden of
- compiler.

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.

1.3 Analysis Framework

1. **Time efficiency (Time Complexity):** indicating how fast the algorithm runs or How long a program takes to process a given input.
2. **Space efficiency (Space Complexity):** indicating how much extra memory it uses. It is the amount of space required by an algorithm to solve the problem.

The algorithm analysis framework consists of the following:

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

(i) Measuring an Input's Size

- Depending on the input it will execute the algorithm
- If it is a longer input size, the algorithm run longer time.
- If it is a smaller input size, the algorithm run less time.
- An algorithm's efficiency is **defined 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.
- 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 an* input size does matter.
- Consider a **spell-checking algorithm**. If the algorithm examines individual characters of its input, then the size is measured by the number of characters.

Units for Measuring Running Time

- We can simply use some standard unit of time measurement such as a second, or millisecond, and so on to measure the running time of a program after implementing the algorithm.
- There are obvious drawbacks to such an approach
- Time efficiency depend on
  - Dependence on the **speed of a particular computer**.
  - Dependence on the **quality of a program implementing the algorithm**.
  - Dependence on the **compiler used in generating the machine code.**
  - difficulty of clocking the actual running time of the program. (We cant find the actual running time of a program because it varies from one system to another)

- One possible approach is to count the number of times each of the algorithm's **basic operations** is executed.
- ***Basic operation:*** The most important operation (+, -, *, /) of the algorithm ie, the operation contributing the most to the total running time.,
- Computing the number of times the basic operation is executed is easy. **The total running time is determined by basic operations count.**
- 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.

- Let ***Cop*** *be the* ***execution time of an algorithm's basic operation*** on a particular computer,
- let ***C(n)*** *be the* ***number of*** **times this operation needs to be executed** for this algorithm.
- Then we can estimate the running time *T (n)* by the formula

$$T(n) = Cop * C(n).$$

*****Problem : Time Complexity of Various code's (please refer your note book)**

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

- **Depending on how many times the basic operation gets executed for a particular input, time complexity of an algorithm can be expressed as Best case, Worst case and Average Case.**
- *Consider Sequential Search algorithm some search key K*

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

$i \leftarrow 0$

while *i* < *n and A[i]* ≠ *K do*

$i \leftarrow i + 1$

if *i* < *n return i*

else return -1

- Clearly, the running time of this algorithm can be quite different for the same list size *n*.

1. Worst-case efficiency

- In the worst case, there is **no matching of elements** or the first **matching element can found at last on the list.**
- **Maximum number of times the basic operation gets executed for input n.**
- The algorithm runs the longest among all possible inputs of that size.
- For the input of size *n*, *the running time is* $C_{worst(n)} = n$.

2. Best case efficiency

- In the best case, there is matching of elements at first on the list.
- **Minimum number of times the basic operation gets executed for input 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 $C_{best(n)} = 1$

3. Average case efficiency

- The Average case efficiency lies between best case and worst case.
- To analyze the algorithm's average case efficiency, we must make some assumptions about possible inputs of size n.
- The standard assumptions are that
  - p: The probability of a successful search is equal to p ($0 \leq p \leq 1$)
  - N : number of elements in list
  - In the case of a successful search, the probability of the first match occurring in the ith position of the list is **p/n** for every i, and the number of comparisons made by the algorithm in such a situation is obviously i.
  - In the case of an unsuccessful search, the number of comparisons will be n with the probability of such a search being **(1− p)**

$$C_{avg}(n) = [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1 - p)$$

$$= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1 - p)$$

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

Orders of Growth

- A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones.
- For example, the greatest common divisor of two small numbers, it is not immediately clear how much more efficient Euclid's algorithm is compared to the other algorithms, the difference in algorithm efficiencies becomes clear for larger numbers only.

- **It is used to measure the performance of an algorithm based on input size.**
- For large values of n, it is the function's order of growth that counts just like the Table, which contains values of a few functions particularly important for analysis of algorithms.

$\sqrt{n} < \log n < n < n\log n < n^2 < n^3 < 2^n < n!$

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}		

1.4 Asymptotic Notations and its properties

- **It is used to compare and rank the order of growth of a function.**
- It use three notations, they are:
  - O - Big oh notation
  - Ω - Big omega notation
  - Θ - Big theta notation

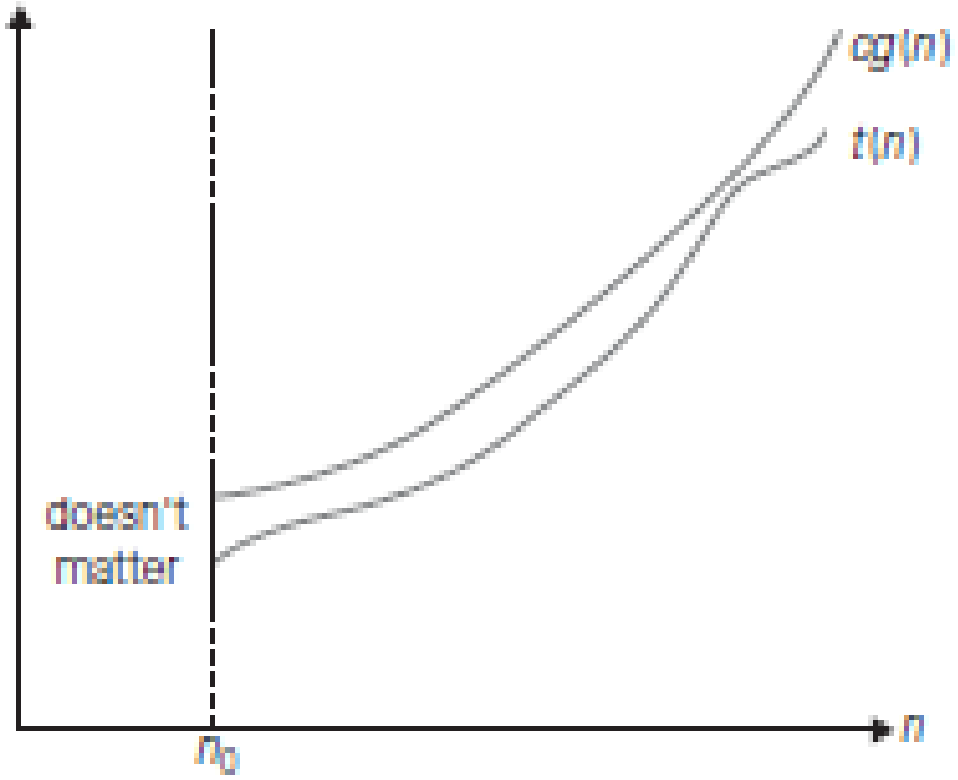
O - Big oh notation

- 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

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

- Where t(n) and g(n) are nonnegative functions defined on the set of natural numbers.
- O = Asymptotic upper bound = Useful for **worst case analysis**

***** Refer your note book for problems**

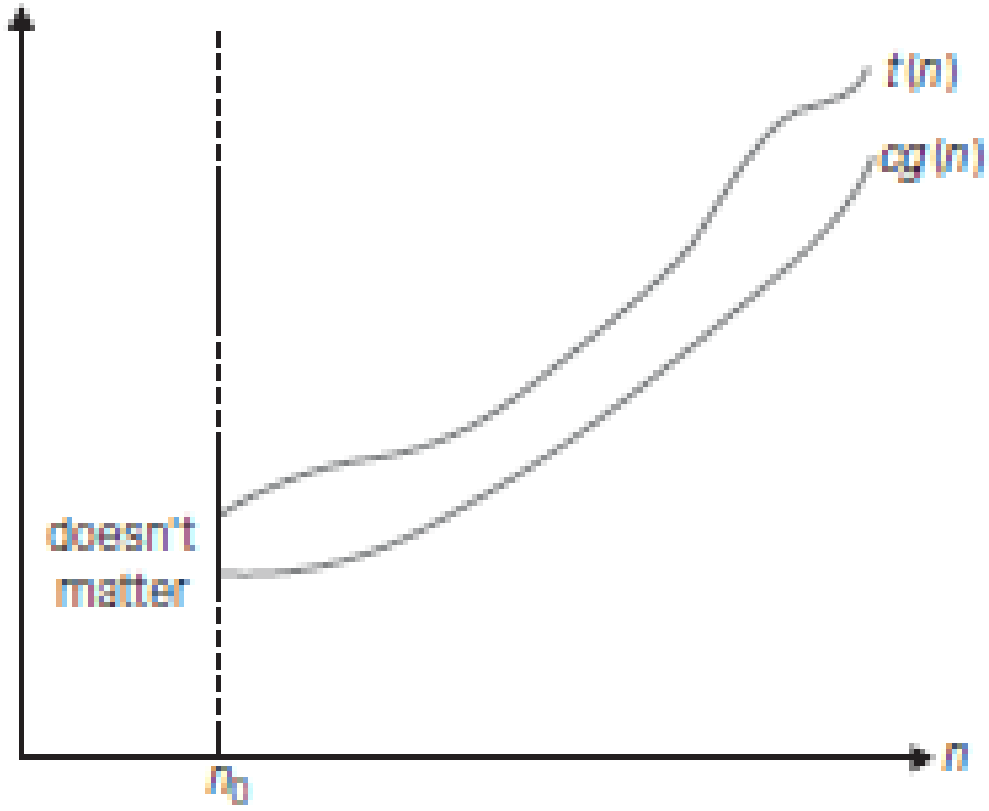


Ω - Big omega notation

- 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 positive 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) ≥ cg(n) for all n ≥ n0.

- Where *t(n) and g(n) are nonnegative functions defined on the set of natural numbers.*
- Ω = Asymptotic lower bound = Useful for **best case analysis**



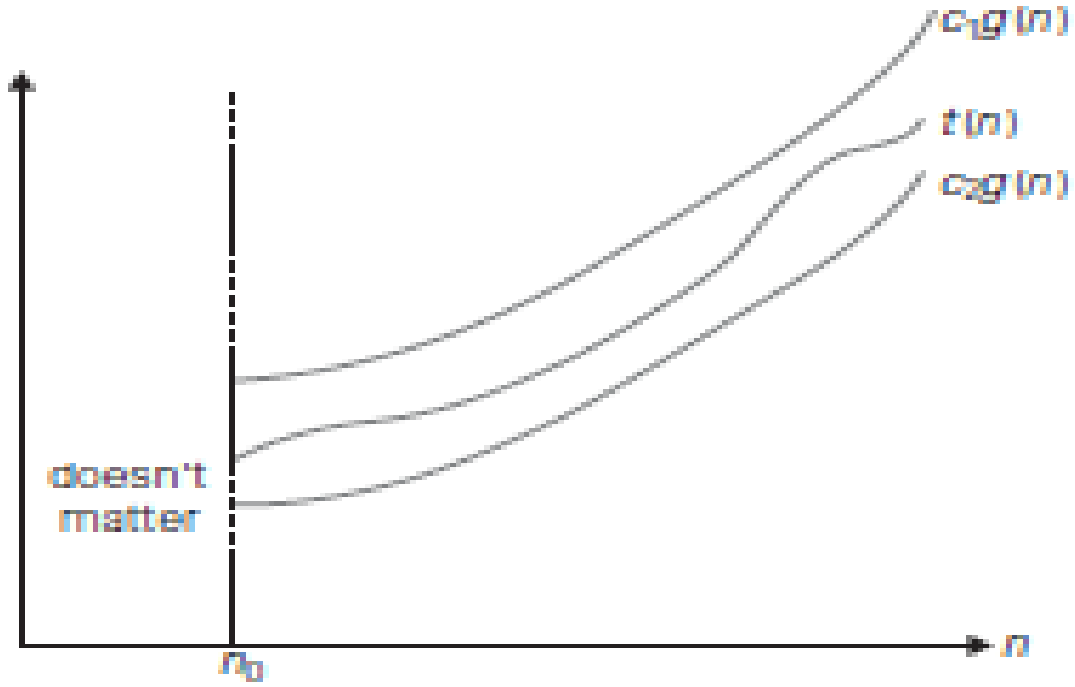
***** Refer your note book for problems**

Θ - Big theta notation

- 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 positive constant multiples of g(n) for all large n, i.e., if there exist some positive constants c1 and c2 and some nonnegative integer n0 such that

c2g(n) ≤ t (n) ≤ c1g(n) for all n ≥ n0.

- Where t(n) and g(n) are nonnegative functions defined on the set of natural numbers.
- Θ = Asymptotic tight bound = Useful for average case analysis



***** Refer your note book for problems**

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 nonnegative integer n_1 such that

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

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

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

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 definition O being $2c_3 = 2 \max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively.

The property implies that the algorithm's overall efficiency will be determined by the part with a higher order of growth, i.e., its least efficient part.

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

Using Limits for Comparing Order of Growth

- For comparing the orders of growth of two specific functions.
- A much more convenient method for doing so is based on computing the limit of the ratio of two functions in question.
- Three principal cases may arise:

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

- $0 \rightarrow t(n) < O(g(n))$ *ie*, $t(n) \in O(g(n))$
- $C \rightarrow t(n) = O(g(n))$ *ie*, $t(n) \in \Theta\, O(g(n))$
- $\infty \rightarrow t(n) > O(g(n))$ *ie*, $t(n) \in \Omega(g(n))$

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

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

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.)

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

Since the limit is equal to zero, $\log_2 n$ has a smaller order of growth than $\sqrt{n}$. (Since $\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = 0$, we can use the so-called ***little-oh notation***: $\log_2 n \in o(\sqrt{n})$. Unlike the big-Oh, the little-oh notation is rarely used in analysis of algorithms.)

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. ■

Basic Efficiency Classes

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.

1.5 MATHEMATICAL ANALYSIS OF NON RECURSIVE ALGORITHMS

General Plan for Analyzing the Time Efficiency of Non Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation (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.
5. Using standard formulas and rules of sum manipulation either find a closed form formula for the count or at the least, establish its order of growth.

Basic rules of sum manipulation

$$\sum_{i=l}^{u} ca_i = c \sum_{i=l}^{u} a_i, \qquad \textbf{(R1)}$$

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i, \qquad \textbf{(R2)}$$

Summation formulas

$$\sum_{i=l}^{u} 1 = u - l + 1 \quad \text{where } l \le u \text{ are some lower and upper integer limits,} \qquad \textbf{(S1)}$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2). \qquad \textbf{(S2)}$$

EXAMPLE 1: Consider the **element uniqueness problem**

- Check whether all the Elements in a given array of n elements are distinct.

```
ALGORITHM UniqueElements(A[0..n − 1])
//Determines 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
```

Algorithm analysis

- The natural measure of the input’s size here is again n (the number of elements in the array).
- Since the innermost loop contains a single operation (the comparison of two elements), we should consider it as the algorithm’s basic operation.
- The number of element comparisons depends not only on n but also on whether there are same elements in the array and, if there are, which array positions they occupy. We will limit our investigation to the worst case only**.**
- One comparison is made for each repetition of the innermost loop, i.e., for each value of the loop variable j between its limits i + 1 and n − 1; this is repeated for each value of the outer loop, i.e., for each value of the loop variable i between its limits 0 and n − 2.

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

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

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

We also could have computed the sum $\sum_{i=0}^{n-2} (n-1-i)$ faster as follows:

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

EXAMPLE 4 The following algorithm **finds the number of binary digits** in the binary representation of a positive decimal integer.

```
ALGORITHM Binary(n)
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
count ←1
while n > 1 do
        count ←count + 1
        n←n/2
return count
```

Algorithm analysis

- An input's size is n.
- The loop variable takes on only a few values between its lower and upper limits; therefore, we have to use an alternative way of computing the number of times the loop is executed.
- Since the value of n is about halved on each repetition of the loop, the answer should be about log2 n.
- The exact formula for the number of times the comparison n>1 will be executed is actually $\lfloor \log_2 n \rfloor + 1$. – the number of bits in the binary representation of n

1.6 MATHEMATICAL ANALYSIS OF RECURSIVE ALGORITHMS

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

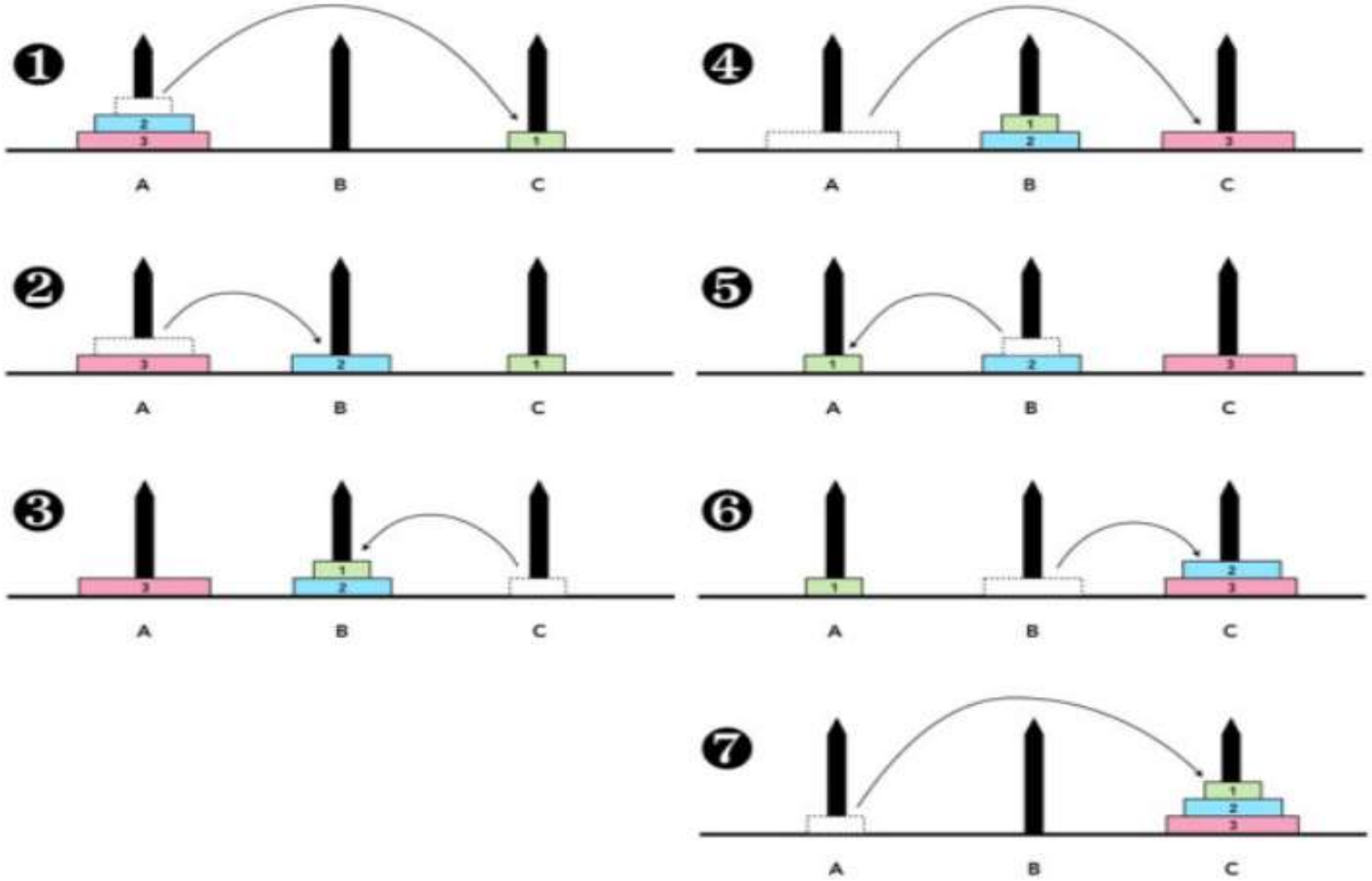
1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must 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.

EXAMPLE 1 : Tower of Hanoi puzzle. (It is famous recursive problem)

- In this puzzle, we have **n disks of different sizes** that can slide onto any of **three pegs.**
- 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, if necessary.

Note:

1. Only one disk my be moved at t time.
2. At no time can a larger disk be moved on a smaller disk.


1
A
B
C
2
A
B
C
3
A
B
C
4
A
B
C
5
A
B
C
6
A
B
C
7
A
B
C

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

    Move nth disk from A to C

    Move top n-1 disks from B to C using 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

Brute Force Approach

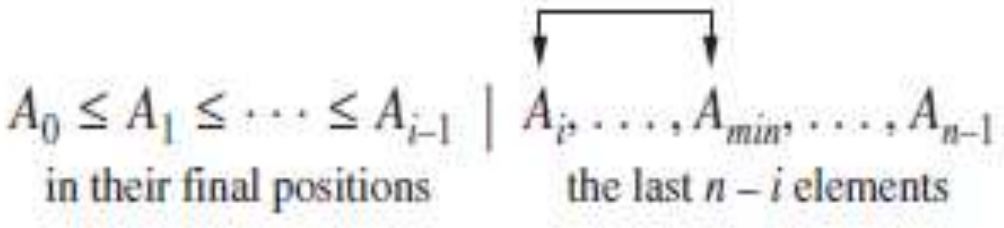
- Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.
- Another way to describe is “Just do it!”
- Example, consider the exponentiation problem: compute a^n for a nonzero number a and a nonnegative integer n. By the definition of exponentiation,

$$a^n = \underbrace{a * \cdots * a}_{n \text{ times}}.$$

.

1.7 Selection Sort

- It is based on brute force approach.
- 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, to find the smallest among the last n − 1 elements and exchange it 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 smallest item among the last n − i elements and swaps it with A_i :



- After n − 1 passes, the list is sorted.

Algorithm

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 nondecreasing 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

Time Complexity

two summation formulas

$$\sum_{i=l}^{u} 1 = u - l + 1 \quad \text{where } l \le u \text{ are some lower and upper integer limits.} \quad \textbf{(S1)}$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2). \quad \textbf{(S2)}$$

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

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

Thus, selection sort is a **Θ(n^2)** algorithm on all inputs.

```c
#include <stdio.h>
#include<time.h>
void selectionSort(int array[], int size)
{
   int i,j,min,temp;
   for (i = 0; i < size - 2; i++)
   {
      min = i;
      for (j = i + 1; j < size-1; j++)
      {
         if (array[j] < array[min])
            min = j;
      }
      temp = array[min];
      array[min] = array[j];
      array[j]= temp;
   }
}
```

```c
int main()
{
   int n, a[2000],k;
   clock_t st,et;
   double ts;
   printf("\n Enter How many Numbers:");
   scanf("%d", &n);
   printf("\nThe Random Numbers are:\n");
   for(k=1; k<=n; k++)
   {
      a[k]=rand();
      printf("%d\t", a[k]);
   }
   st=clock();
   selectionSort(a, n);
   et=clock();
   ts=(double)(et-st)/CLOCKS_PER_SEC;
   printf("\n Sorted Numbers are : \n ");
   for(k=1; k<=n; k++)
   {
      printf("%d\t", a[k]);
   }
   printf("\nThe time taken is %e",ts);
}
```

1. Design and implement C/C++ Program to sort a given set of n integer elements using Selection Sort method and compute its time complexity. Run the program for varied values of n> 5000 and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator.

1.8 Sequential Search

- Here the given search key is simply compared with successive elements of a given list. If a match is encountered (Successful search) else if list is exhausted without finding a match(Unsuccessful search).
- It is based on brute force approach.

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

- ***Example:*** n=100

i) if x=a[0], then only 1 comparison is made

Cbest(n) = 1

ii) But if x!=a[0….n-1] then 100 comparison is made

Cworst(n) = n

iii) In an average case, when all the array elements are distinct and each item is searched for equal frequency the average case is

$$C_{avg}(n) = [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1 - p)$$
$$= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1 - p)$$
$$= \frac{p}{n}\frac{n(n+1)}{2} + n(1 - p) = \frac{p(n+1)}{2} + n(1 - p).$$

For example, if p = 1 (the search must be successful), the average number of key comparisons made by sequential search is **(n + 1)/2**;