Lecture Notes on
Analysis and Design of Algorithms
BCS401

Module-1: Introduction to Algorithm

Contents

Module-1: Introduction

1.1 Introduction

1.1.1 What is an Algorithm?

Algorithm: An algorithm is a finite sequence of unambiguous instructions to solve a particular problem.

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

1.1.2. Algorithm Specification

An algorithm can be specified in

1) Simple English
2) Graphical representation like flow chart
3) Programming language like C++/java
4) Combination of above methods.

Example: Combination of simple English and C++, the algorithm for **selection sort** is specified as follows.

```
for (i=1; i<=n; i++) {
   examine a[i] to a[n] and suppose
   the smallest element is at a[j];
   interchange a[i] and a[j];
}
```

Example: In C++ the same algorithm can be specified as follows. Here *Type* is a basic or user defined data type.

```
void SelectionSort(Type a[], int n)
// Sort the array a[1:n] into nondecreasing order.
{
    for (int i=1; i<=n; i++) {
       int j = i;
       for (int k=i+1; k<=n; k++)
          if (a[k]<a[j]) j=k;
       Type t = a[i]; a[i] = a[j]; a[j] = t;
    }
}
```

1.1.3. Analysis Framework

Measuring an Input's Size

It is observed that almost all algorithms **run longer on larger inputs.** For example, it takes longer to sort larger arrays, multiply larger matrices, and so on. Therefore, it is logical to investigate an algorithm's efficiency as a function of some parameter ***n*** indicating the algorithm's input size.

There are situations, where the choice of a **parameter indicating an input size does matter**. The choice of an appropriate size metric can be influenced by operations of the algorithm in question. For example, how should we measure an input's size for a spell-checking algorithm? If the algorithm examines individual characters of its input, then we should measure the size by the number of characters; if it works by processing words, we should count their number in the input.

We should make a special note about measuring the size of inputs for algorithms involving **properties of numbers** (e.g., checking whether a given integer n is prime). For such algorithms, computer scientists prefer measuring size by the number ***b*** of bits in the ***n***'s binary representation:= $\log_2 n] + 1$. This metric usually gives a better idea about the efficiency of algorithms in question.

Units for Measuring Running lime

To measure an algorithm's efficiency, we would like to have a **metric 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 xecuted. This approach is both excessively difficult and, as we shall see, usually unnecessary. The thing to do is to identify the most important operation of the algorithm, called the **basic operation**, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.

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 **matrix multiplication** and **polynomial evaluation** require two arithmetic operations: **multiplication and addition**.

Let c_{op} be the execution time of an algorithm's basic operation on a particular computer, and 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)*** of a program implementing this algorithm on that computer by the formula:

$$T(n) \approx c_{op}C(n)$$

Unless ***n*** is extremely large or very small, the formula can give a reasonable estimate of the algorithm's running time.

It is for these reasons that the efficiency analysis framework ignores multiplicative constants and concentrates on the count's **order of growth** to within a constant multiple for large-size inputs.

Orders of Growth

Why this emphasis on the count's order of growth for large input sizes? Because for large values of n, it is the function's order of growth that counts: just look at table which contains values of a few functions particularly important for analysis of algorithms.

Table: *Values of several functions important for analysis of algorithms*

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	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	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Algorithms that require an exponential number of operations are practical for solving only problems of very small sizes.

1.2. Performance Analysis

There are two kinds of efficiency: **time efficiency** and **space efficiency**.

- Time efficiency indicates how fast an algorithm in question runs;
- Space efficiency deals with the extra space the algorithm requires.

In the early days of electronic computing, both resources **time** and **space** were at a premium. The research experience has shown that for most problems, we can achieve much more spectacular progress in speed than inspace. Therefore, we primarily concentrate on time efficiency.

1.2.1 Space complexity

Total amount of computer memory required by an algorithm to complete its execution is called as **space complexity** of that algorithm. The Space required by an algorithm is the sum of following components

- A **fixed** part that is independent of the input and output. This includes memory space for codes, variables, constants and so on.
- A **variable** part that depends on the input, output and recursion stack. (We call these parameters as instance characteristics)

Space requirement *S(P)* of an algorithm *P*, ***S(P) = c + Sp*** where ***c*** is a constant depends on the fixed part, ***Sp*** is the instance characteristics\

Example-1: Consider following algorithm **abc()**

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

Here fixed component depends on the size of a, b and c. Also instance characteristics *Sp=0*

Example-2: Let us consider the algorithm to find sum of array. For the algorithm given here the problem instances are characterized by ***n***, the number of elements to be summed. The space needed by ***a[ ]***depends on ***n***.So the space complexity can be written as;$S_{sum}(n) \geq (n+3)$; n for a[ ], One each for n, i and s.

```
float Sum(float a[], int n)
{   float s = 0.0;
    for (int i=1; i<=n; i++)
       s += a[i];
    return s;
}
```

1.2.2 Time complexity

Usually, the execution time or run-time of the program is refereed as its time complexity denoted by t_p(instance characteristics). This is the sum of the time taken to execute all instructions in the program. Exact estimation runtime is a complex task, as the number of instructions executed is dependent on the input data. Also different instructions will take different time to execute. So for the estimation of the time complexity **we count only the number of program steps**. We can determine the **steps needed by a program** to solve a particular problem instance in two ways.

Method-1: We introduce a new variable ***count*** to the program which is initialized to zero. We also introduce statements to increment ***count*** by an appropriate amount into the program. So when each time original program executes, the ***count*** also incremented by the step count.

Example: Consider the algori hm ***sum()***. After the introduction of the count the program will be as follows. We can estimate that invocation of ***sum()*** executes total number of **2n+3** steps.

```
float Sum(float a[], int n)
{   float s = 0.0;
    count++; // count is global
    for (int i=1; i<=n; i++) {
       count++; // For 'for'
       s += a[i]; count++; // For assignment
    }
    count++; // For last time of 'for'
    count++; // For the return
    return s;
}
```

Method-2: Determine the step count of an algorithm by building a table in which we list the total number of steps contributed by each statement. An example is shown below. The code will find the sum of n numbers.

Example: Matrix addition

Statement	s/e	freq	total
void Add(Type a[][SIZE], ...)	0	–	0
{ for (int i=1; i<=m; i++)	1	$m+1$	$m+1$
for (int j=1; j<=n; j++)	1	$m(n+1)$	$mn+m$
c[i][j] = a[i][j] + b[i][j];	1	mn	mn
}	0	–	0
}			
Total			$2mn+2m+1$
Total			$2n+3$

The above method is both excessively difficult and, usually unnecessary. The thing to do is to identify the most important operation contributing the m operation of the algorithm, called the basic operation, the st to the total running time, and compute the number of times the basic operation is executed.

Trade-off

There is often a **time-space-tradeoff** involved in a problem, that is, it cannot be solved with few computing time and low memory consumption. One has to make a compromise and to exchange computing time for memory consumption or vice versa, depending on which algorithm one chooses and how one parameterizes it.

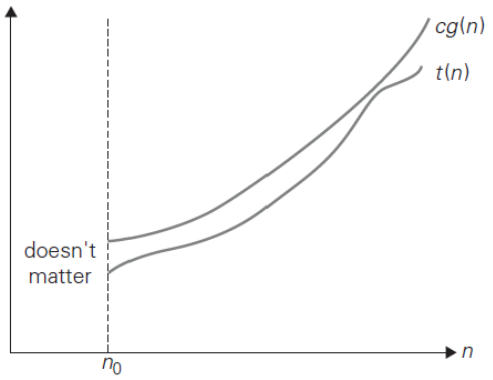
1.3. Asymptotic Notations

The efficiency analysis framework concentrates on the order of growth of an algorithm's basic operation count as the principal indicator of the algorithm's efficiency. To compare and rank such orders of growth, computer scientists use three notations: O(big oh), Ω(big omega), Θ (big theta) and o(little oh)

1.3.1. Big-Oh 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

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



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

Informally, *O(g(n))* is the set of all functions with a lower or same order of growth as *g(n)*. Note that the definition gives us a lot of freedom in choosing specific valuesfor constants *c* and n_0.

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

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

Strategies to prove Big-O: Sometimes the easiest way to prove that f (n) = O(g(n)) is to take c to be the sum of the positive coefficients off(n). We can usually ignore the negative coefficients.

Example: To prove $5n^2 + 3n + 20 = O(n^2)$, we pick $c = 5 + 3 + 20 = 28$. Then if $n \geq n_0 = 1$,

$$5n^2 + 3n + 20 \leq 5n^2 + 3n^2 + 20n^2 = 28n^2,$$

thus $5n^2 + 3n + 20 = O(n^2)$.

Example: To prove **100n + 5 ∈ O(n^2)**

$100n + 5 \leq 105n^2$. *(c=105, n_0=1)*

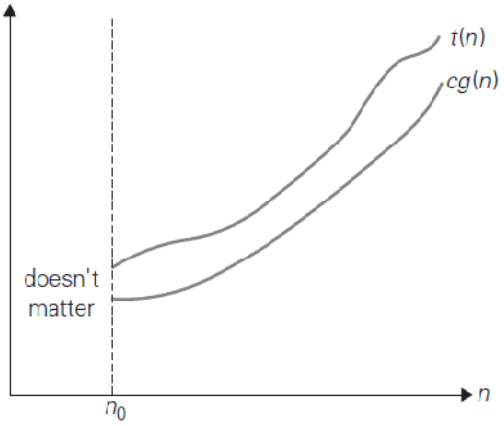
Example: To prove **n^2 + n = O(n^3)**

Take c = 1+1=2, if n ≥n_0=1, then n^2 + n = O(n^3)

i) Prove 3n+2=O(n) ii) Prove $1000n^2+100n-6 = O(n^2)$

1.3.2. Omega 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 some positive constant multiple of g(n) for all lar n,i.e., if there exist some positive constant c and son nonnegative integer n_0 suchthat ***t(n) ≥ c g(n)*** fo all n ≥ n_0.



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

Here is an example of the formal proof that ***n^3 ∈Ω(n^2):*** $n^3 \geq n^2$ for all n ≥ 0, i.e., we can select c = 1 and n_0 = 0.

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

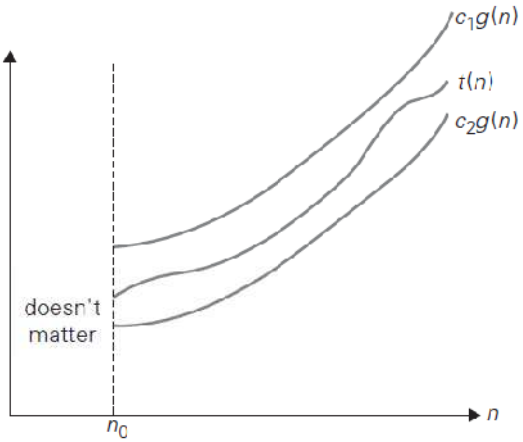
Example: To prove $n^3 + 4n^2 = \Omega(n^2)$

We see that, if n≥0, $n^3+4n^2 \geq n^3 \geq n^2$; Therefore $n^3+4n^2 \geq 1n^2$ for alln≥0 Thus, we have shown that $n^3+4n^2 = \Omega(n^2)$ where c = 1 & n_0=0

1.3.3. Theta notation

A function t(n) is said to be in **Θ(g(n)),** denoted **t(∈ Θ(g(n)),**if t (n) is bounded both above and belo by some positive constant multiples ofg(n) for all large n, i.e., if there exist some positive constants c_1 and c_2 and somenonnegative integer n_0 such that

$c_2g(n) \leq t(n) \leq c_1g(n)$ for all n ≥ n_0.



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):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2 \quad \text{for all } n \geq 0.$$

Second, we prove the left inequality (the lower bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n \text{ (for all } n \geq 2) = \frac{1}{4}n^2.$$

Hence, we can select $c_2 = \frac{1}{4}$, $c_1 = \frac{1}{2}$, and $n_0 = 2$.

Example: $n^2 + 5n + 7 = \Theta(n^2)$

When $n \geq 1$,

$$n^2 + 5n + 7 \leq n^2 + 5n^2 + 7n^2 \leq 13n^2$$

When $n \geq 0$,

$$n^2 \leq n^2 + 5n + 7$$

Thus, when $n \geq 1$

$$1n^2 \leq n^2 + 5n + 7 \leq 13n^2$$

Thus, we have shown that $n^2 + 5n + 7 = \Theta(n^2)$ (by definition of Big-Θ, with $n_0 = 1$, $c_1 = 1$, and $c_2 = 13$.)

Strategies for Ω and Θ

- Proving that a f(n) = Ω(g(n)) often requires more thought.
  - Quite often, we have to pick c < 1.
  - A good strategy is to pick a value of c which you think will work, and determine which value of n_0 is needed.
  - Being able to do a little algebra helps.
  - We can sometimes simplify by ignoring terms of f(n) with the positive coefficients.
- The following theorem shows us that proving f(n) = Θ(g(n)) is nothing new:

  Theorem: f(n) = Θ(g(n)) if and only if f(n) = O(g(n)) and f(n) = Ω(g(n)).

  Thus, we just apply the previous two strategies.

Show that $\frac{1}{2}n^2 + 3n = \Theta(n^2)$

Notice that if $n \geq 1$,

$$\frac{1}{2}n^2 + 3n \leq \frac{1}{2}n^2 + 3n^2 = \frac{7}{2}n^2$$

Thus,

$$\frac{1}{2}n^2 + 3n = O(n^2)$$

Also, when $n \geq 0$,

$$\frac{1}{2}n^2 \leq \frac{1}{2}n^2 + 3n$$

So

$$\frac{1}{2}n^2 + 3n = \Omega(n^2)$$

Since $\frac{1}{2}n^2 + 3n = O(n^2)$ and $\frac{1}{2}n^2 + 3n = \Omega(n^2)$,

$$\frac{1}{2}n^2 + 3n = \Theta(n^2)$$

Show that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$

We need to find positive constants c_1, c_2, and n_0 such that

$$0 \le c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2 \text{ for all } n \ge n_0$$

Dividing by n^2, we get

$$0 \le c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2$$

$c_1 \le \frac{1}{2} - \frac{3}{n}$ holds for $n \ge 10$ and $c_1 = 1/5$

$\frac{1}{2} - \frac{3}{n} \le c_2$ holds for $n \ge 10$ and $c_2 = 1$.

Thus, if $c_1 = 1/5$, $c_2 = 1$, and $n_0 = 10$, then for all $n \ge n_0$,

$$0 \le c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2 \text{ for all } n \ge n_0.$$

Thus we have shown that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

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 aboutfour 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 $t_1(n) \le c_1 g_1(n)$ for all $n \ge n_1$.

Similarly, since $t_2(n) \in O(g_2(n))$, $t_2(n) \le c_2 g_2(n)$ for all $n \ge n_2$.

Let us denote $c_3 = \max\{c_1, c2\}$ and consider n>=max{n1,n2} so that

we can use both inequalities. Adding them yields the following: $t_1(n) + t_2(n) \le c_1 g_1(n) + c_2 g_2(n)$

$\le c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)]$

$\le c_3 2 \max\{g_1(n), g_2(n)\}$.

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.

3.4. Little Oh The function **f(n)= o(g(n))** [i.e f of n is a little oh of g of n] if and only if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Example: The function $3n + 2 = o(n^2)$ since $\lim_{n\to\infty} \frac{3n+2}{n^2} = 0$. $3n + 2 = o(n \log n)$. $3n + 2 = o(n \log\log n)$. $6 * 2^n + n^2 = o(3^n)$. $6 * 2^n + n^2 = o(2^n \log n)$. $3n + 2 \ne o(n)$. $6 * 2^n + n^2 \ne o(2^n)$. □

For comparing the order of growth limit is used

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

If the case-1 holds good in the above limit, we represent it by little-oh.

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.) ■

1.3.5. 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.

1.3.6. Mathematical Analysis of Non-recursive & Recursive Algorithms

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 innermost loop.)
3. Check whether the number of times the basic operation is execut d depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if separately.
4. Set up a sum expressing the number of times the algorithm's executed.
5. Using standard formulas and rules of sum manipulation, either

Example-1: To find maximum element in the given array

ALGORITHM *MaxElement*$(A[0..n-1])$
//Determines the value of the largest element in a given a
//Input: An array $A[0..n-1]$ of real numbers
//Output: The value of the largest element in A
$maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
  if $A[i] > maxval$
    $maxval \leftarrow A[i]$
return $maxval$

Here comparison is the basic operation. Note that number of comparisions will be same for all arrays of size n. Therefore, no need to distinguish worst, best and average cases. Total number of basic operations are,

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

Example-2: To check whether all the elements in the given array 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 $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[i] = A[j]$ **return false**
return true

Here basic operation is comparison. The maximum no. of comparisons happens in the worst case. i.e. all the elements in the array are distinct and algorithms return *true*).

Total number of basic operations (comparison) in the worst case are,

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

Other than the worst case, the total comparisons are**less than** $\frac{1}{2} n^2$. For example if the first two elements of the array are equal, only one comparison is computed. So in general **C(n) =O(n^2)**

Example-3: To perform matrix multiplication

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

```
//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two n × n matrices A and B
//Output: Matrix C = AB
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
```

Number of basic operations (multiplications) is

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

Total running time: $T(n) \approx c_m M(n) = c_m n^3$

Suppose if we take into account of addition; Algoritham also have same number of additions $A(n) = n^3$

Total running time: $T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a) n^3$

Example-4: To count the bits in the binary representation

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
```

The basic operation is count=count + 1 repeats $\lfloor \log_2 n \rfloor + 1$ no. of times

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 varyon different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
4. Solve the recurrence or, at least, ascertain the order of growth of its solution.

Example-1 Compute the factorial function $F(n) = n!$ for an arbitrary nonnegative integer n. Since

$$n! = 1 \cdot \ldots \cdot (n-1) \cdot n = (n-1)! \cdot n \quad \text{for } n \geq 1$$

and $0! = 1$ by definition, we can compute $F(n) = F(n-1) \cdot n$ with the following recursive algorithm.

ALGORITHM $F(n)$

```
//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!
if n = 0 return 1
else return F(n − 1) ∗ n
```

Since the function *F(n)* is computed according to the formula

$F(n) = F(n-1) \cdot n \quad \text{for } n > 0,$

The number of multiplications*M(n)* needed to compute it must satisfy the equality

$$M(n) = \underbrace{M(n-1)}_{\substack{\text{to compute} \\ F(n-1)}} + \underbrace{1}_{\substack{\text{to multiply} \\ F(n-1) \text{ by } n}} \quad \text{for } n > 0.$$

Such equations are called **recurrence relations**

Condition that makes the algorithm stop***if n = 0 return 1.*** Thus recurrence relation and initial conditionfor the algorithm's number of multiplications M(n) can be stated as

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0,$$
$$M(0) = 0.$$

We can use backward substitutions method to solve this

$$\begin{aligned} M(n) &= M(n-1) + 1 && \text{substitute } M(n-1) = M(n-2) + 1 \\ &= [M(n-2) + 1] + 1 = M(n-2) + 2 && \text{substitute } M(n-2) = M(n-3) + 1 \\ &= [M(n-3) + 1] + 2 = M(n-3) + 3. \\ &\ldots \\ &= M(n-i) + i = \cdots = M(n-n) + n = n. \end{aligned}$$

Example-2: Tower of Hanoi puzzle. In this puzzle, There are ***n*** disks of different sizes that canslide onto any of three pegs. Initially, all the disks are on the first peg in order ofsize, the largest on the bottom and the smallest on top. The goal is to move all thedisks to the third peg, using the second one as an auxiliary, if necessary. We canmove only one disk at a time, and it is forbidden to place a larger disk on top of asmaller one.The problem has an elegant recursive solution, which is illustrated in Figure.

1. If n = 1, we move the single disk directly from the source peg to the destination peg.
2. To move n>1 disks from peg 1 to peg 3 (with peg 2 as auxiliary),
   - we first move recursively n-1 disks from peg 1 to peg 2 (with peg 3 as auxiliary),
   - then move the largest disk directly from peg 1 to peg 3, and,
   - finally, move recursively n-1 disks from peg 2 to peg 3 (using peg 1 as auxiliary).

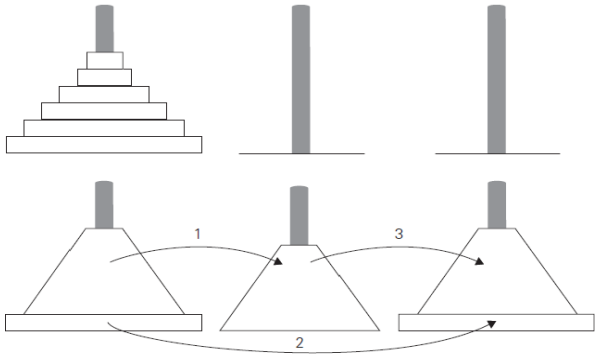


Figure: Recursive solution to the Tower of Hanoi puzzle

```
Algorithm: TowerOfHanoi(n, source, dest, aux)
        If n == 1, THEN
                move disk from
        source to dest else
          TowerOfHanoi (n - 1, source, aux, dest)
          move disk from source to dest
          TowerOfHanoi (n - 1, aux, dest, source)
        End if
```

Computation of Number of Moves

The number of moves ***M(n)*** depends only on n. The recurrence equation is

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

We have the following recurrence relation for the number of moves M(n):

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

We solve this recurrence by t e same method of backward substitutions:

$$\begin{aligned} M(n) &= 2M(n-1) + 1 && \text{sub. } M(n-1) = 2M(n-2) + 1 \\ &= 2[2M(n-2) + 1] + 1 = 2^2 M(n-2) + 2 + 1 && \text{sub. } M(n-2) = 2M(n-3) + 1 \\ &= 2^2[2M(n-3) + 1] + 2 + 1 = 2^3 M(n-3) + 2^2 + 2 + 1. \end{aligned}$$

The pattern of the first three sums on the left suggests that the next one will be

$$M(n) = 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1 = 2^i M(n-i) + 2^i - 1$$

$2^4M(n - 4) + 2^3 + 2^2 + 2 + 1$, and generally, after *i* substitutions, we get

Since the initial condition is specified for n = 1, which is achieved for i = n-1, we get the following formula for the solution to recurrence,

$$\begin{aligned} M(n) &= 2^{n-1} M(n - (n-1)) + 2^{n-1} - 1 \\ &= 2^{n-1} M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1 \end{aligned}$$

Example-3: To count bits of a decimal number in its binary representation

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 $BinRec(\lfloor n/2 \rfloor) + 1$

The recurrence relation can be written as

.

Also note that A(1) = 0.

$$A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1$$

$$A(2^k) = A(2^{k-1}) + 1 \quad \text{for } k > 0,$$
$$A(2^0) = 0.$$

The standard approach to solving such a recurrence is to **solve it only for n = 2^k** and then take advantage of the theorem called the **smoothness rule** which claims that under very broad

assumptions the order of growth observed for n = 2^k gives a correct answer about the order of growth for all values of n.

Now backward substitutions encounter no problems:

$$
\begin{aligned}
A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1\\
&= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1\\
&= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \dots\\
&\dots\\
&= A(2^{k-i}) + i\\
&\dots\\
&= A(2^{k-k}) + k.
\end{aligned}
$$

Thus, we end up with

$$A(2^k) = A(1) + k = k,$$

or, after returning to the original variable $n = 2^k$ and hence $k = \log_2 n$,

$$A(n) = \log_2 n \in \Theta(\log n).$$

1.4. Brute force design technique:

Brute force is straight forward 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:

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

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 $A[min]$

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.

1.4.2 Sequential search

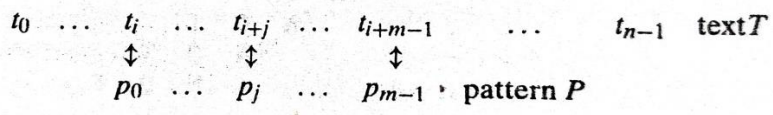
This is also called as Linear search. Here we start from the initial element of the array and compare it with the search key. We repeat the same with all the elements of the array till we encounter the search key or till we reach end of the array.

```
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] ← K
    i ← 0
    while A[i] ≠ K do
        i ← 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.

1.4.3 String matching algorithm with complexity Analysis

Another example of Brute force approach is string matching, where string of n characters called ***text*** and a string of m characters (m<=n) called the ***pattern*** is given. Here job is to find whether the ***pattern*** is present in ***text*** or not. If we want to find i-the index of the leftmost character of the first matching substring in the text—such that $t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$:



We start matching with the very first character, if a match then only j is incremented and again compared with next character of both the strings. If not then I is incremented and j starts from beginning of pattern string. If pattern found we return the position from where the pattern began. Pattern is tried to match till n-m elements, later we need not try to match as the elements will be lesser than pattern. If it doesn't match by n-m elements then pattern is not matched.

```
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 ← 0 to n − m do
        j ← 0
        while j < m and P[j] = T[i + j] do
            j ← j + 1
        if j = m return i
    return −1
```

The worst case is Θ(nm). Best case is Θ(m).