4. 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.

1.1 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 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 $n \times n$ 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.
- In measuring input size for algorithms solving problems such as checking primality of a positive integer n. the input is just one number.
- The input size by the number b of bits in the n's binary representation is $b=(\log_2 n)+1$.

(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 basic operations count.

(iii)ORDERS OF GROWTH

- A difference in running times on small inputs is not what really distinguishes efficient algorithms from in efficient 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.
- For large values of *n*,
- it is the function's order of growth that counts just like theTable1.1, 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	$\sqrt{}$	$\log_2 n$	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**	**$6.4 \cdot 10^1$**	**$2.6 \cdot 10_2$**	**$4.1 \cdot 10_3$**	**$6.5 \cdot 10^4$**	**$2.1 \cdot 10^1_3$**
10^2	**10**	**6.6**	**10_2**	**$6.6 \cdot 10^2$**	**10^4**	**10^6**	**$1.3 \cdot 10^3_0$**	**$9.3 \cdot 10^1_{57}$**
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}		

***(iii)* Worst-Case, Best-Case, and Average-Case Efficiencies** *Consider Sequential Search* algorithm some search key *K* **ALGORITHM** *Sequential Search (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** $A[i] \neq K$ **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*.

In the worst case, there is no matching of elements or the first matching element can found at last on the list. In the best case, there is matching of elements at first on the list.

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 running time is $C_{worst}(n) = n$.

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 $C_{best}(n) = 1$

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
  - 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

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

list is the same for every *i*. Yet another type of efficiency is called ***amortized efficiency***. It applies not to a single run of an algorithm but rather to a sequence of operations performed on the same data structure.