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.

Input. Zero or more quantities are externally supplied.

a. ***Output***. At least one quantity is produced.
b. ***Definiteness***. Each instruction is clear and unambiguous. It must be perfectly clear what should be done.
c. ***Finiteness***. If we trace out the instruction of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
d. ***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: $b = \lfloor \log_2 n \rfloor + 1$. 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 executed. 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:

$$T(n) = c_{op}C(n)$$

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 algorithm ***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

Statement	s/e	frequency	total steps
`float Sum(float a[], int n)`	0	–	0
`{ float s = 0.0;`	1	1	1
`for (int i=1; i<=n; i++)`	1	$n+1$	$n+1$
`s += a[i];`	1	n	n
`return s;`	1	1	1
`}`	0	–	0
Total			$2n+3$

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			$2mn+2m+1$

The above method is both excessively difficult and, 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.

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.

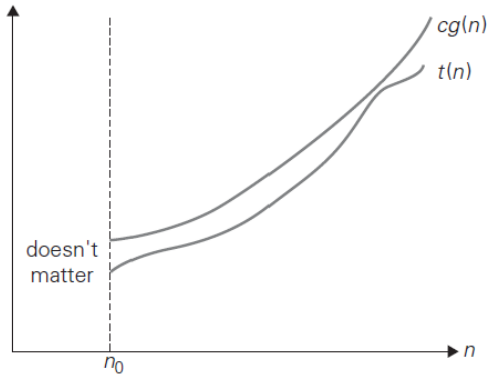
1.3. Asymptotic Notations

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)* ∈*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) ≤ ***cg(n)*** *for all* ***n*** ≥ ***n₀***.



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 values for constants *c* and n_0.

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

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

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 $n \geq n_0 = 1$,

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

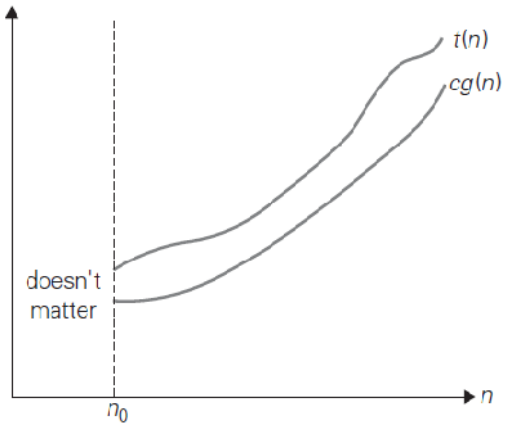
thus $5n^2 + 3n + 20 = O(n^2)$.

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 by some positive constant multiple of g(n) for all large n,i.e., if there exist some positive constant c and some nonnegative integer n_0 such that ***t(n) ≥ c g(n)*** for 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.

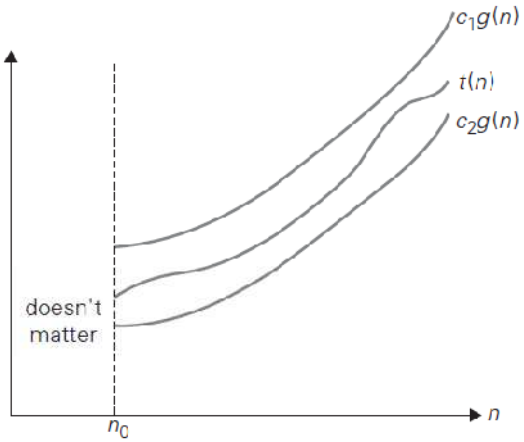
Example: $n^3 \in \Omega(n^2)$, $\frac{1}{2}n(n-1) \in \Omega(n^2)$, but $100n + 5 \notin \Omega(n^2)$.

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(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 c_1 and c_2 and some nonnegative integer n_0 such that

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



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 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 $t_1(n) \le c_1 g_1(n)$ 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 $c_3 = \max\{c_1, c_2\}$ 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.

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.