UNIT-1

DATA STRUCTURES AND ALGORITHMS

1.1 PROBLEMS TO PROGRAMS

Half the battle is knowing what problem to solve. When initially approached, most problems have no simple, precise specification. In fact, certain problems, such as creating a "gourmet" recipe or preserving world peace, may be impossible to formulate in terms that admit of a computer solution. Even if we suspect our problem can be solved on a computer, there is usually considerable latitude in several problem parameters

Almost any branch of mathematics or science can be called into service to help model some problem domain. Problems essentially numerical in nature can be modeled by such common mathematical concepts as simultaneous linear equations (e.g., finding currents in electrical circuits, or finding stresses in frames made of connected beams) or differential equations (e.g., predicting population growth or the rate at which chemicals will react). Symbol and text processing problems can be modeled by character strings and formal grammars. Problems of this nature include compilation (the translation of programs written in a programming language into machine language) and information retrieval tasks such as recognizing particular words in lists of titles owned by a library.

Once we have a suitable mathematical model for our problem, we can attempt to find a solution in terms of that model. Our initial goal is to find a solution in the form of an algorithm**, which is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.**

An integer assignment statement such as x := y + z is an example of an instruction that can be executed in a finite amount of effort. In an algorithm instructions can be executed any number of times, provided the instructions themselves indicate the repetition. However, we require that, no matter what the input values may be, an algorithm terminate after executing a finite number of instructions.

1.2 DATA STRUCTURE:

Data structure is basically a group of data elements that are put together under one name. It also defines a particular way of storing and organizing data in a computer, so that it can be used efficiently.

Data Structure is the structural representation of logical relationships between data or element. It represents the way of storing, organizing and retrieving data.

It is classified as

- **Linear data structure**
- **Non Linear data structure**

Examples:

Linear : Lists, Stacks, Queues etc.

Non Linear : Trees, graphs etc.

Normally all the data structures can be implemented using 2 methods

- Array implementation
- Linked List implementation

Applications of data structure:

- Compiler design
- Statistical analysis package
- Numerical Analysis
- Artificial Intelligence
- Operating System
- Data base management system
- Simulation
- Graphics

1.3 ABSTRACT DATA TYPE

Abstract data type is a set of operations of data. It also specifies the logical and mathematical model of the data type . ADT specification includes description of the data, list of operations that can be carried out on the data and instructions how to use these instructions. But ADT does not mention how the set of operations is implemented.

Examples for ADT: lists, sets and graphs along with their operations for ADT.

Examples for data type: Integers, reals and Booleans.

1.4 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.

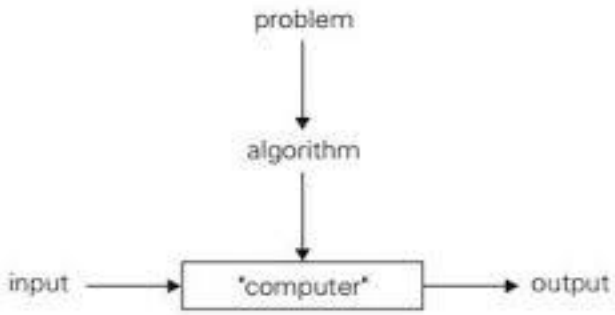


FIGURE 1.1 The notion of the algorithm.

It is a step by step procedure with the input to solve the problem in a finite amount of time to obtain the required output.

1.4.1 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.

1.4.2 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:** Every instruction must be very basic and runs in short time.

1.4.3 Steps for writing an algorithm:

1. An algorithm is a procedure. It has two parts; the first part is **head** and the second part is **body**.
2. The Head section consists of keyword **Algorithm** and Name of the algorithm with **parameter list. E.g. Algorithm name1(p1, p2,…,p3)**
3. The head section also has the following:

   //ProblemDescription:

   //Input:

   //Output:
4. In the body of an algorithm various programming constructs like **if, for, while** and some statements like assignments are used.
5. The compound statements may be enclosed with **{** and **}** brackets. **if**, **for**, **while** can be closed by **endif**, **endfor**, **endwhile** respectively. Proper indention is must for block.
6. Comments are written using // at the beginning.
7. The **identifier** should begin by a letter and not by digit. It contains alpha numeric letters after first letter. No need to mention data types.
8. The left arrow **"←" used as assignment operator. E.g. v←10**
9. **Boolean** operators (TRUE, FALSE), **Logical** operators (AND, OR, NOT) and **Relational operators (<,<=, >, >=,=,** ≠, <>) are also used.
10. Input and Output can be done using **read** and **write**.
11. **Array[]**, **if then else condition**, **branch** and **loop** can be also used in algorithm.

Example:

The greatest common divisor(GCD) of two nonnegative integers *m* and *n* (not-both-zero), denoted gcd*(m, n)*, is defined as the largest integer that divides both *m* and *n* evenly, i.e., with a remainder of zero.

Euclid's algorithm 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 greatest common divisor of the initial *m* and *n*. gcd*(*60, 24*)* can be computed as follows:gcd*(*60, 24*)* = gcd*(*24, 12*)* = gcd*(*12, 0*)* = 12.

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

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.

.

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 ←m mod n
        m←n
        n←r
return m
```

1.5 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

- Pseudocode is a mixture of a natural language and programming language constructs. Pseudocode is usually more precise than natural language.

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

ALGORITHM *Sum(a,b)*

//Problem Description: This algorithm performs addition of two numbers

//Input: Two integers a and b

//Output: Addition of two integers

c←a+b

return c

This specification is more useful for implementation of any language.

c. **Flowchart**

In the earlier days of computing, the dominant method for specifying algorithms was a ***flowchart***, this representation technique has proved to be inconvenient.

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

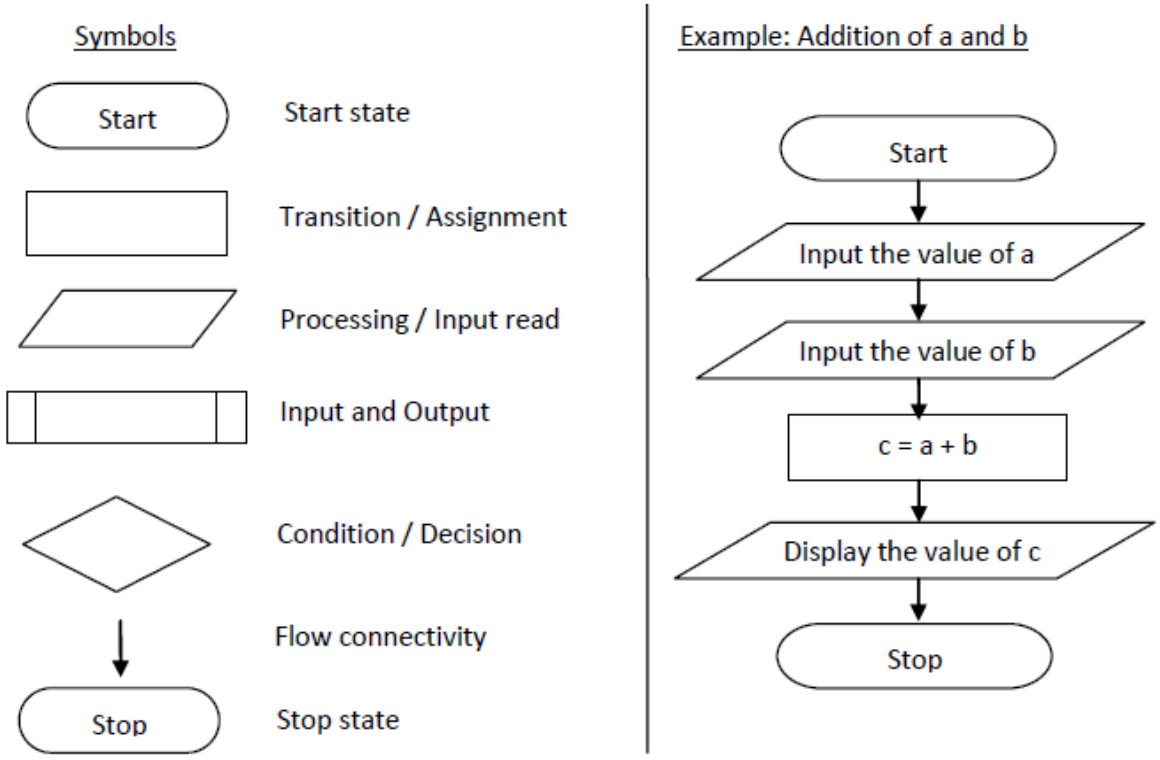


FIGURE 1.4 Flowchart symbols and Example for two integer addition.

1.6 RECURSION

A problem-solving technique in which problems are solved by reducing them to smaller problems of the same form.

In programming, recursion simply means that a function will call itself:

```
int sum() {
        sum();
        return 0;
 }
```

A recursive function always has to say when to stop repeating itself. There should always be two parts to a recursive function:

- the **recursive case** and
- the **base case**.

The recursive case is when the function calls itself.

Base case

- **Base case** in which the problem is simple enough to be solved directly without the need for any more calls to the same function
- The base case is when the function stops calling itself. This prevents infinite loops.

Recursive case

- The problem at hand is initially subdivided into simpler sub-parts.
- The function calls itself again, but this time with sub-parts of the problem obtained in the first step.
- The final result is obtained by combining the solutions of simpler sub-parts.

A recursive function is said to be **well-defined** if it has the following two properties:

- There must be a base criteria in which the function doesn't call itself.
- Every time the function is called itself, it must be closer to the base criteria.

Example:

The power() function: Write a recursive function that takes in a number (x) and an exponent (n) and returns the result of x^n

```
int power(int x,int exp)
{
  If(exp==0)
        return 1;
  else
        return X * power(X, exp-1)
}
```

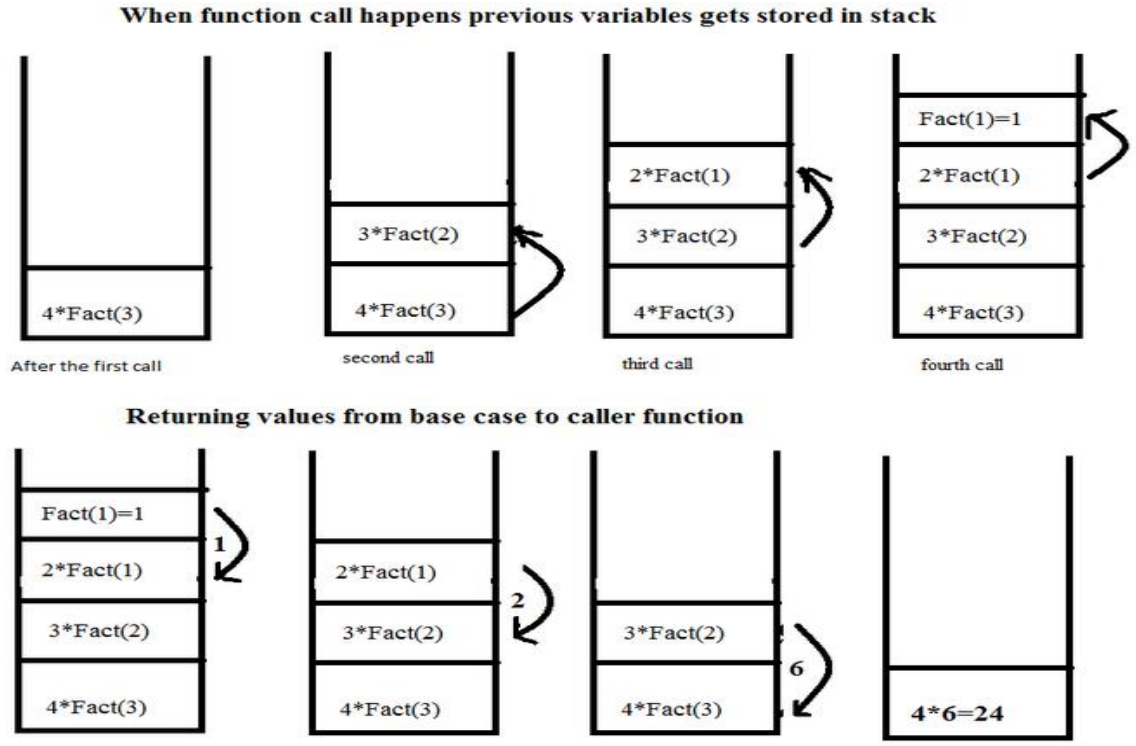
1.6.1 The Call Stack

Recursive functions use something called "the call stack." When a program calls a function, that function goes on top of the call stack. This similar to a stack of books. You add things one at a time. Then, when you are ready to take something off, you always take off the top item. Here is a recursive function to calculate the factorial of a number:

```
function fact(x) {
 if (x == 1) {
```

```
        return 1;
          } else {
              return x * fact(x-1);
              }
    }
```

Now let's see what happens if you call fact(3) The illustration bellow shows how the stack changes, line by line. The topmost box in the stack tells you what call to fact you're currently on.



1.6.2 Example : Towers of Hanoi

Consider the following puzzle

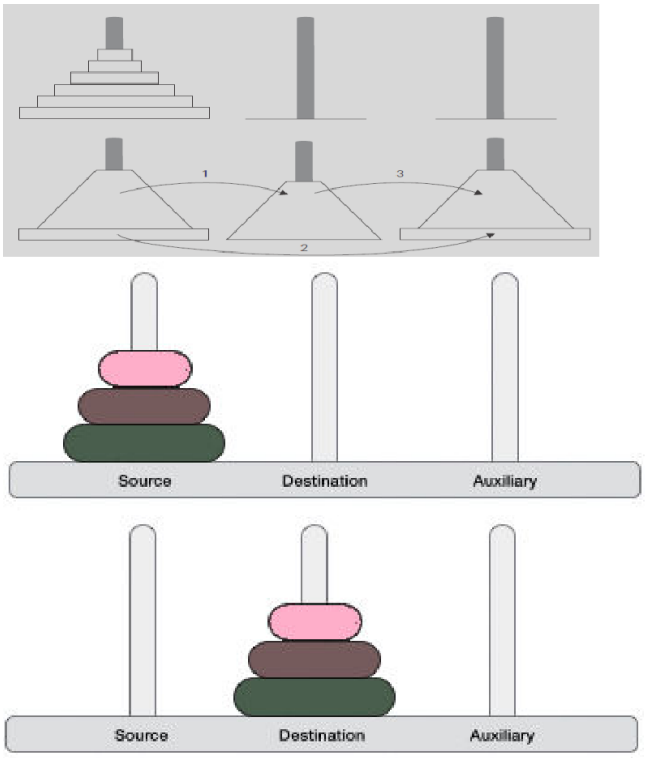
- There are 3 pegs (posts) a, b, c and n disks of different sizes
- Each disk has a hole in the middle so that it can fit on any peg
- At the beginning of the game, all n disks are on peg a, arranged such that the largest is on the bottom, and on top sit the progressively smaller disks, forming a tower
- Goal: find a set of moves to bring all disks on peg c in the same order, that is, largest on bottom, smallest on top

constraints

- the only allowed type of move is to grab one disk from the top of one peg and drop it on another peg

- a larger disk can never lie above a smaller disk, at any time

Consider educational workhorse of recursive algorithms: the ***Tower of Hanoi*** puzzle. We have n disks of different sizes that can slide onto any of three pegs. Consider A (source), B (auxiliary), and C (Destination). Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top. The goal is to move all the disks to the third peg, using the second one as an auxiliary.



ALGORITHM TOH(n, A, C, B)

//Move disks from source to destination recursively //Input: *n disks and* 3 pegs A, B, and C

//Output: Disks moved to destination as in the source order.

if n=1

Move disk from A to C

else

Move top n-1 disks from A to B using C

TOH(n - 1, A, B, C)

Move top n-1 disks from B to C using A

TOH(n - 1, B, C, A)

Animation 3 disks : https://www.youtube.com/watch?v=5QuiCcZKyYU

Animation :https://www.youtube.com/watch?v=9nwXi9m31RI

Animation 5 disks : https://www.youtube.com/watch?v=BalWjeY2O9g

https://www.youtube.com/watch?v=0u7g9C0wSIA

C program

```
#include <stdio.h>
#include <conio.h>
void hanoi(char,char,char,int);
void main()
{
        int num;
        clrscr();
        printf("\nENTER NUMBER OF DISKS: ");
        scanf("%d",&num);
        printf("\nTOWER OF HANOI FOR %d NUMBER OF DISKS:\n", num);
        hanoi('A','B','C',num);
        getch();
}
void hanoi(char from,char to,char other,int n)
{
        if(n<=0)
                printf("\nILLEGAL NUMBER OF DISKS");
        if(n==1)
                printf("\nMOVE DISK FROM %c TO %c",from,other);
        if(n>1)
        {
                hanoi(from,other,to,n-1);
                hanoi(from,to,other,1);
                hanoi(to,from,other,n-1);
        }
}
```

1.7 COMPLEXITY

Complexity

Suppose M is an algorithm and suppose n is the size of the input data. The efficiency of M is measured in terms of time and space used by the algorithm. Time is measured by counting the number of operations and space is measured by counting the maximum amount of memory consumed by M.

The complexity of M is the function f(n) which gives running time and or space in terms of n. In complexity theory, we find complexity f(n) for three major cases as follows,

- **Worst case**: f(n) have the maximum value for any possible inputs.
- **Average case**: f(n) have the expected value.
- **Best case**: f(n) have the minimum possible value.

These ideas are illustrated by taking the example of the Linear search/ *Sequential Search* algorithm.

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

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 list is the same for every i.

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

1.7.1 SPACE TIME TRADE OFF

Space-Time Tradeoff

- Most discussion on algorithm efficiency speaks to run-time, efficiency in CPU utilization. However, especially with cloud computing, memory utilization and data exchange volume must also be considered. This page gives a brief introduction to this topic.
- A **space-time** or **time-memory tradeoff** in computer science is a case where an algorithm or program trades increased space usage with decreased time. Here, **space** refers to the data

torage consumed in performing a given task (RAM, HDD, etc), and **time** refers to the time consumed in performing a given task (computation time or response time).

- The utility of a given space-time tradeoff is affected by related fixed and variable costs (of, e.g., CPU speed, storage space), and is subject to diminishing returns.

History

- Biological usage of time–memory tradeoffs can be seen in the earlier stages of animal behavior. Using stored knowledge or encoding stimuli reactions as "instincts" in the DNA avoids the need for "calculation" in time-critical situations. More specific to computers, look-up tables have been implemented since the very earliest operating systems.
- In 1980 Martin Hellman first proposed using a time–memory tradeoff for cryptanalysis.

Types of tradeoff

i. **Lookup tables vs. recalculation**

The most common situation is an algorithm involving a lookup table: an implementation can include the entire table, which reduces computing time, but increases the amount of memory needed, or it can compute table entries as needed, increasing computing time, but reducing memory requirements.

ii. **Compressed vs. uncompressed data**

A space–time tradeoff can be applied to the problem of data storage. If data is stored uncompressed, it takes more space but access takes less time than if the data were stored compressed (since compressing the data reduces the amount of space it takes, but it takes time to run the decompression algorithm). Depending on the particular instance of the problem, either way is practical. There are also rare instances where it is possible to directly work with compressed data, such as in the case of compressed bitmap indices, where it is faster to work with compression than without compression.

iii. **Re-rendering vs. stored images**

Storing only the SVG source of a vector image and rendering it as a bitmap image every time the page is requested would be trading time for space; more time used, but less space. Rendering the image when the page is changed and storing the rendered images would be trading space for time; more space used, but less time. This technique is more generally known as caching.

iv. **Smaller code vs. loop unrolling**

Larger code size can be traded for higher program speed when applying loop unrolling. This technique makes the code longer for each iteration of a loop, but saves the

computation time required for jumping back to the beginning of the loop at the end of each iteration.

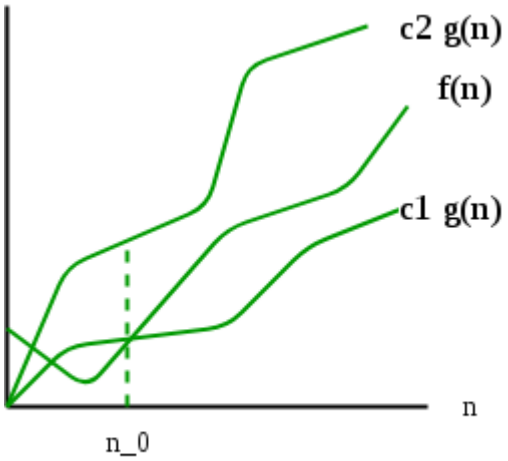
Other examples

- Algorithms that also make use of space-time tradeoffs include:
- Baby-step giant-step algorithm for calculating discrete logarithms
- Rainbow tables in cryptography, where the adversary is trying to do better than the exponential time required for a brute-force attack. Rainbow tables use partially precomputed values in the hash space of a cryptographic hash function to crack passwords in minutes instead of weeks. Decreasing the size of the rainbow table increases the time required to iterate over the hash space.
- The meet-in-the-middle attack uses a space–time tradeoff to find the cryptographic key in only $2n+12n+1$ encryptions (and $O(2n)O(2n)$ space) versus the expected $22n22n$ encryptions (but only $O(1)O(1)$ space) of the naive attack.
- Dynamic programming, where the time complexity of a problem can be reduced significantly by using more memory.

1.8 ASYMPTOTIC NOTATIONS

The main idea of asymptotic analysis is to have a measure of the efficiency of algorithms that don't depend on machine-specific constants and don't require algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent the time complexity of algorithms.

Θ Notation: The theta notation bounds a function from above and below, so it defines exact asymptotic behavior.



f(n) = theta(g(n))

A simple way to get the Theta notation of an expression is to drop low-order terms and ignore leading constants. For example, consider the following expression.

$3n^3 + 6n^2 + 6000 = \Theta(n^3)$

Dropping lower order terms is always fine because there will always be a number(n) after which $\Theta(n^3)$ has higher values than $\Theta(n^2)$ irrespective of the constants involved.

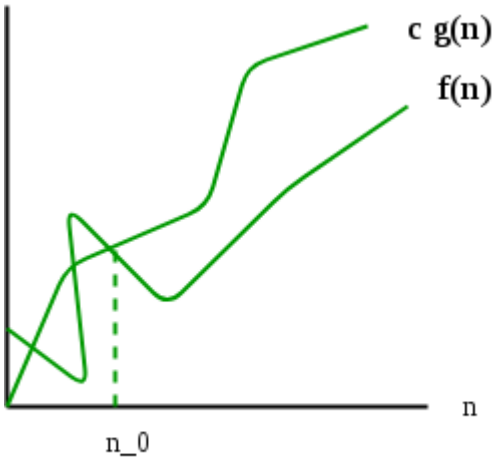
For a given function g(n), we denote $\Theta(g(n))$ is following set of functions.

$\Theta(g(n))$ = {f(n): there exist positive constants c1, c2 and n0 such

that 0 <= c1*g(n) <= f(n) <= c2*g(n) for all n >= n0}

The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.

Big O Notation: The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in the best case and quadratic time in the worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.



f(n) = O(g(n))

If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:

1. The worst-case time complexity of Insertion Sort is Θ(n^2).
2. The best case time complexity of Insertion Sort is Θ(n).

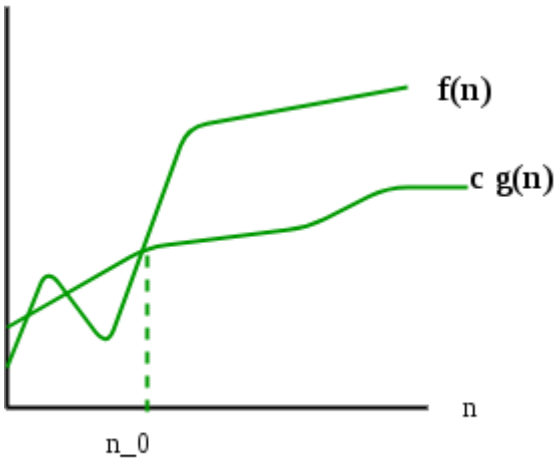
The Big O notation is useful when we only have an upper bound on the time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

O(g(n)) = { f(n): there exist positive constants c and

n0 such that 0 <= f(n) <= c*g(n) for

all n >= n0}

Ω Notation: Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.



f(n) = Omega(g(n))

Ω Notation can be useful when we have a lower bound on the time complexity of an

algorithm. As discussed in the previous post, the best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three.
For a given function g(n), we denote by Ω(g(n)) the set of functions.

Ω (g(n)) = {f(n): there exist positive constants c and

n0 such that 0 <= c*g(n) <= f(n) for

all n >= n0}.

Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as Ω(n), but it is not very useful information about insertion sort, as we are generally interested in worst-case and sometimes in the average case.