MODULE 1

BASIC CONCEPTS OF DATA STRUCTURES (2019 scheme)

System Life Cycle: How to create programs

Requirements: process of gathering and interpreting facts, diagnosing problems, by analysis of end-user information needs and the removal of any inconsistencies and incompleteness in these requirements.

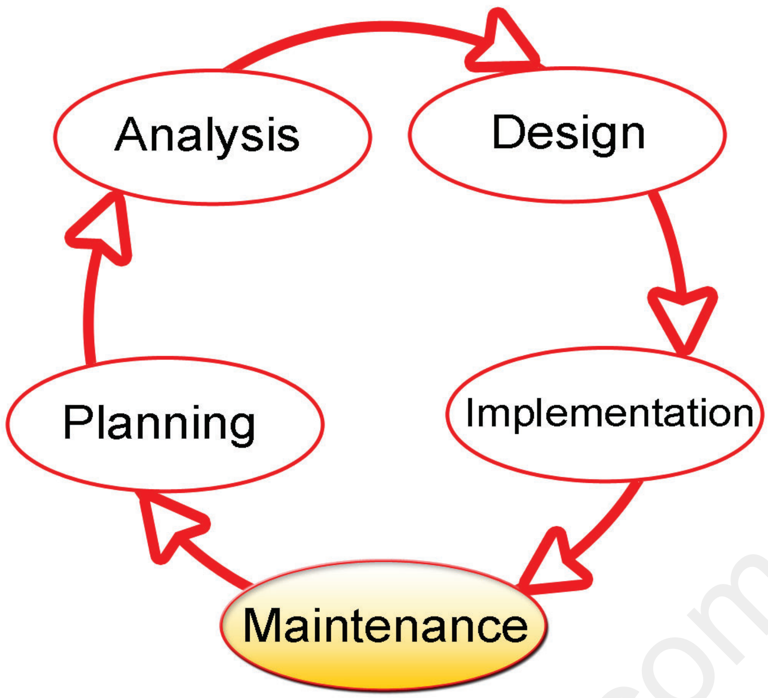
1.Collection of facts: Obtain end user requirements through documentation, client interviews, observation, and questionnaires.

2.Scrutiny of the existing system: Identify pros and cons of the current system in-place, so as to carry forward the pros and avoid the cons in the new system.

3.Analysis of the proposed system: Find solutions to the shortcomings described in step two and prepare the specifications using any specific user proposals.

Analysis: bottom-up vs. top-down

Determine where the problem is, in an attempt to fix the system. This step involves breaking down the system in different pieces to analyze the situation and breaking down what needs to be created.



Design: data objects and operations

describe the new system as a collection of modules or subsystems.

Refinement and Coding - Writing and executing programs and then optimizing them may be effective for small programs. The real code is written here.

Verification

Program Proving: **program** satisfies a formal specification of its behavior.

Testing: All the modules are brought together into a special testing environment, then checked for errors, bugs, and interoperability. Unit, system, and user acceptance testings.

Debugging: routine process of locating and removing computer program bugs, errors or abnormalities, which is methodically handled by software programmers via **debugging** tools. **Debugging** checks, detects and corrects errors (or "bugs") to allow proper program operation.

Algorithm

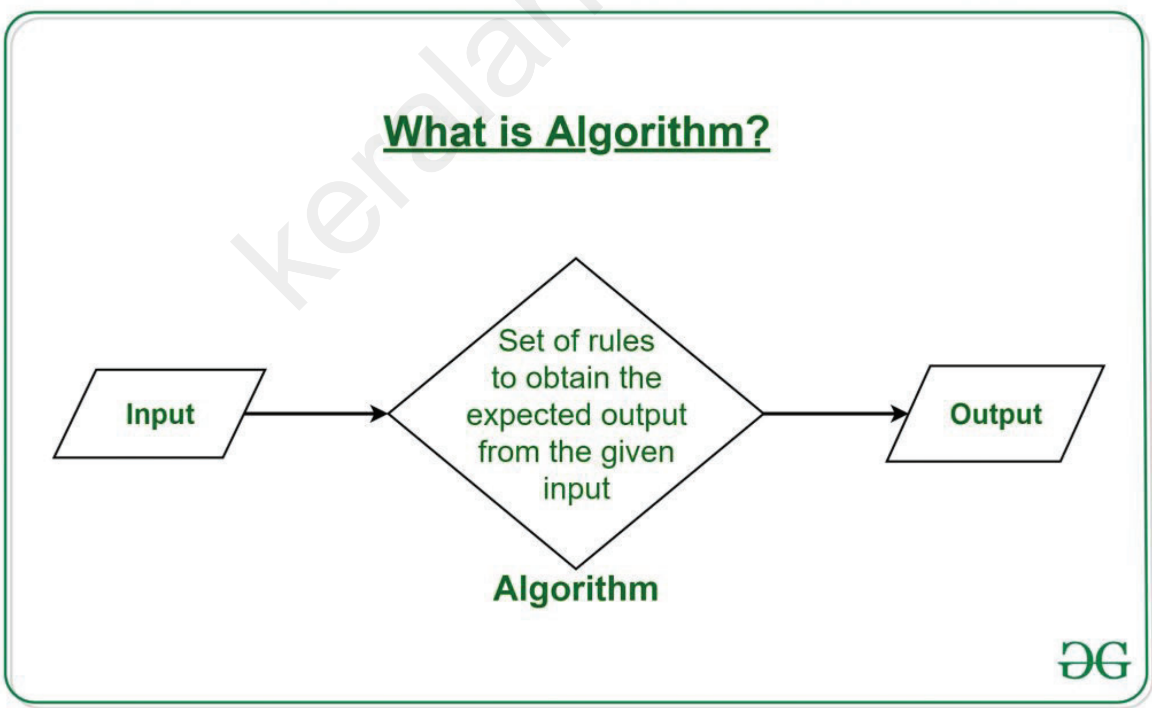
Definition

An *algorithm* is a finite set of instructions that accomplishes a particular task.

Algorithm is a step-by-step finite sequence of instruction, to solve a well-defined computational problem.

Criteria

- input
- output
- definiteness: clear and unambiguous
- finiteness: terminate after a finite number of steps
- effectiveness: instruction is basic enough to be carried out



Measurements

Criteria

Is it correct?

Is it readable?

…

Performance Analysis (machine independent)

space complexity: storage requirement

time complexity: computing time

Space Complexity
$S(P)=C+S_P(I)$

Fixed Space Requirements (C)

Independent of the characteristics of the inputs and outputs

- instruction space
- space for simple variables, fixed-size structured variable, constants

Variable Space Requirements ($S_P(I)$)

depend on the instance characteristic I

- number, size, values of inputs and outputs associated with I
- recursive stack space, formal parameters, local variables, return address

Time Complexity

$$T(P)=C+T_P(I)$$

Compile time (C)
independent of instance characteristics

run (execution) time T_P

Definition
A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

Example

abc = a + b + b * c + (a + b - c) / (a + b) + 4.0

abc = a + b + c

$$\boxed{T_P(n)=c_a ADD(n)+c_s SUB(n)+c_l LDA(n)+c_{st} STA(n)}$$

Methods to compute the step count

Introduce variable count into programs

Tabular method

Determine the total number of steps contributed by each statement
step per execution × frequency

add up the contribution of all statements

Tabular Method

***Figure 1.2:** Step count table for Program 1.10 (p.26)

Iterative function to sum a list of numbers

steps/execution

Statement	s/e	Frequency	Total steps
float sum(float list[ ], int n)	0	0	0
{	0	0	0
float tempsum = 0;	1	1	1
int i;	0	0	0
for(i=0; i <n; i++)	1	n+1	n+1
tempsum += list[i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

Matrix Addition

***Figure 1.4:** Step count table for matrix addition (p.27)

Statement	s/e	Frequency	Total steps
Void add (int a[ ][MAX_SIZE]□ □ □)	0	0	0
{	0	0	0
int i, j;	0	0	0
for (i = 0; i < row; i++)	1	rows+1	rows+1
for (j=0; j< cols; j++)	1	rows□ (cols+1)	rows□ cols+rows
c[i][j] = a[i][j] + b[i][j];	1	rows□ cols	rows□ cols
}	0	0	0
Total			2rows□ cols+2rows+1

Asymptotic Notations

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

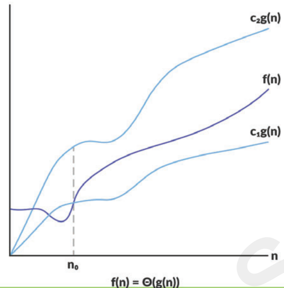
But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations: Theta notation, Omega notation and Big-O notation.

Theta Notation (Θ-notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average case complexity of an algorithm



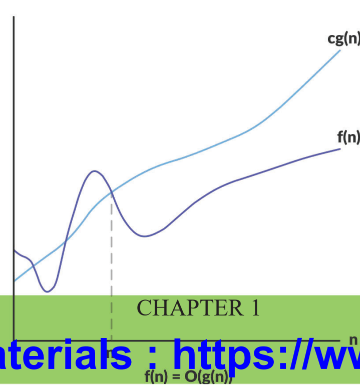
For a function g(n), Θ(g(n)) is given by the relation:

Θ(g(n)) = { f(n): there exist positive constants c1, c2 and n0 such that 0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ n0 }
The above expression can be described as a function f(n) belongs to the set Θ(g(n)) if there exist positive constants c1 and c2 such that it can be sandwiched between c1g(n) and c2g(n), for sufficiently large n.

If a function f(n) lies anywhere in between c1g(n) and c2 > g(n) for all n ≥ n0, then f(n) is said to be asymptotically tight bound.

Big-O Notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst case complexity of an algorithm.



$O(g(n)) = \{ f(n)$: there exist positive constants c and n0 such that $0 \le f(n) \le cg(n$ … n0 }

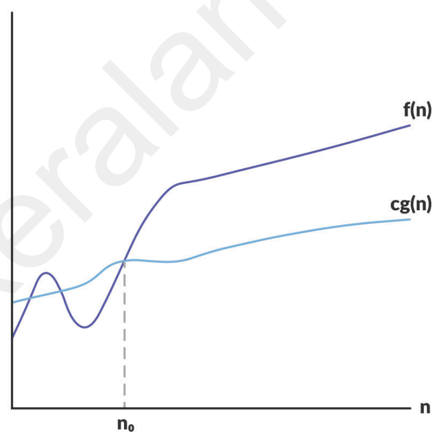
The above expression can be described as a function f(n) belongs to the set O(g(n)) if there exists a positive constant c such that it lies between 0 and cg(n), for sufficiently large n.

For any value of n, the running time of an algorithm does not cross time provided by O(g(n)).

Since it gives the worst case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst case scenario.

Omega Notation (Ω-notation)

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides best case complexity of an algorithm.



f(n) = Ω(g(n))

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n0 \}$

The above expression can be described as a function f(n) belongs to the set Ω(g(n)) if there exists a positive constant c such that it lies above cg(n), for sufficiently large n.

For any value of n, the minimum time required by the algorithm is given by Omega Ω(g(n))

OMEGA (Ω)

Definition
$f(n) = \Omega(g(n))$ iff there exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all n, $n \ge n_0$.

Asymptotic lower bound

Theta (θ)

Definition
$f(n) = \Omega(g(n))$ iff there exist positive constants c and n_0 such that $0 \le c1g(n) \le f(n) \le c2g(n)$ for all n, $n \ge n_0$.

Big "oh" [O]

Definition
$f(n) = O(g(n))$ iff there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all n, $n \ge n_0$.

Asymptotic upper bound.

Examples

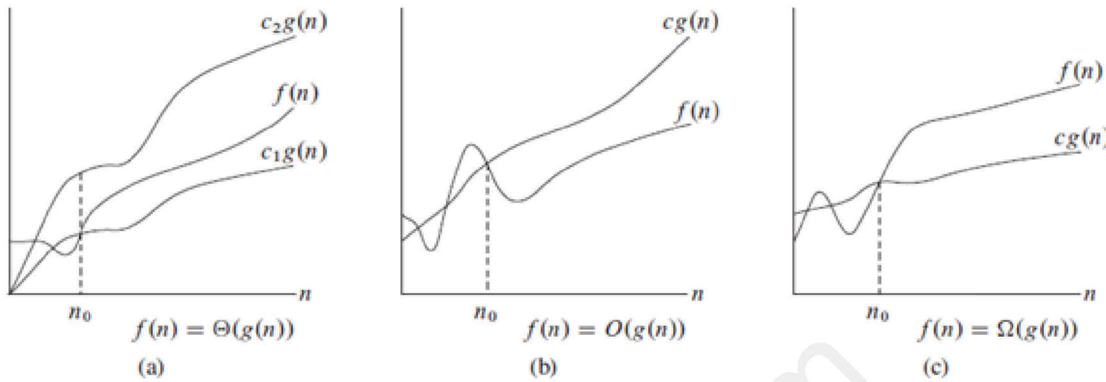
$3n+2=O(n)$ /* $3n+2\le4n$ for $n\ge2$ */

$3n+3=O(n)$ /* $3n+3\le4n$ for $n\ge3$ */

$100n+6=O(n)$ /* $100n+6\le101n$ for $n\ge10$ */

$10n^2+4n+2=O(n^2)$ /* $10n^2+4n+2\le11n^2$ for $n\ge5$ */

$6*2^n+n^2=O(2^n)$ /* $6*2^n+n^2 \le7*2^n$ for $n\ge4$ */



O(1): constant

O(n): linear

O(n^2): quadratic

O(n^3): cubic

O(2^n): exponential

O(logn): logarithmic

O(nlogn): log linear

*Figure 1.7:Function values (p.38)

Time	Name	Instance characteristic n					
		1	2	4	8	16	32
1	Constant	1	1	1	1	1	1
$\log n$	Logarithmic	0	1	2	3	4	5
n	Linear	1	2	4	8	16	32
$n \log n$	Log linear	0	2	8	24	64	160
n^2	Quadratic	1	4	16	64	256	1024
n^3	Cubic	1	8	64	512	4096	32768
2^n	Exponential	2	4	16	256	65536	4294967296
$n!$	Factorial	1	2	24	40326	20922789888000	26313×10^{33}

*Figure 1.8:Plot of function values(p.39)