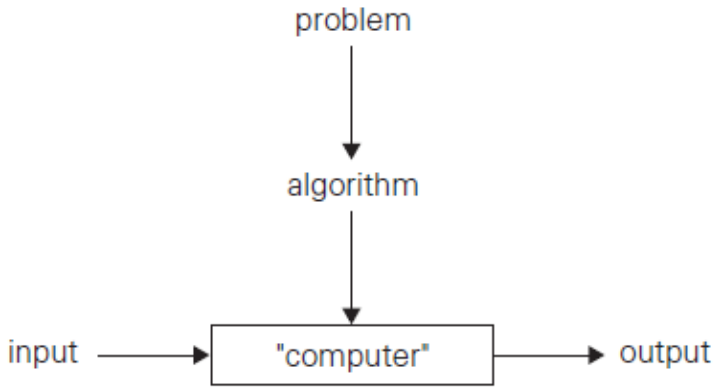
Algorithm – Definition and Properties

Algorithm Definition: - An algorithm is a finite set of instructions that , if followed , accomplishes a particular task



<u>Characteristics/properties of an algorithm</u>:-

1. Input : - An algorithm has **zero (0) or more** inputs.
2. Output : - An algorithm must produce **one (1) or more** outputs.
3. Finiteness : - An algorithm must contain a **finite number of steps**.
4. Definiteness : - Each step of an algorithm must be **clear and unambiguous**.
5. Effectiveness : - An algorithm should be effective i.e., operations can be performed with the given inputs in a finite period of time by a person using paper and pencil

Example- Algorithm specification

```
1. Algorithm Max(A, n)
2. // A is an array of size n.
3. {
4         Result:=A[1];
5         for i :=2 to n do
6              if A[i] > Result then Result:=A[i];
7          return Result;
8    }
```

Fundamentals of Algorithmic problem solving

The following are the steps for solving the problem algorithmically

- Understanding the problem
- Ascertaining(knowing) the capabilities of the Computational devices
- Choosing between exact and Approximate problem solving
- Algorithm Design techniques
- Designing algorithm and data structures
- Methods of specifying an Algorithm
- Proving an Algorithm’s correctness
- Analyze an Algorithm
- Coding an Algorithm

Need of Understanding the problem

- Understand completely the problem given by ***reading the description*** carefully and ask the questions of self .
- Some of the problems that arise in computing applications quite often are known algorithms. If the problem in question is one of the known algorithm, then use it to known its strengths and weakness.

Ascertaining(Knowing) the Capabilities of a Computational Device

- After understanding the problem, know the architecture of the computational device (which is based on Von-Neumann architecture) . The essence will be captured using random-access machine(RAM) , which executes instructions one after another i.e., sequential.
- Sequential algorithms vs parallel algorithms(not suitable for RAM model)

Choosing between exact and Approximate problem solving

- There exists some important problems which cannot be solved exactly for most of the instances. *Example* : square roots, solving non-linear equations, and evaluating definite integrals.
- Solving the problem exactly can be unacceptably slow because of problem's intrinsic complexity.

Deciding on Appropriate Data Structures

- Algorithms + Data Structures = Programs

Algorithm Design Techniques

- Algorithm Design technique is a general approach to solving the problems algorithmically that is applicable to a variety of problems from different areas of computing
- The reasons for knowing these techniques are
  - Provide guidance for designing algorithms for new problems.
  - It is cornerstone of computer science

Methods of specifying an Algorithm

- Using the natural language for specifying the algorithm makes difficult because of inherent ambiguity of the language.
- Pseudocode – a mixture of a natural language and programming language constructs.
- Flow chart – obsolete, because it is convenient to represent simple algorithm but not all.

Proving an Algorithm's Correctness

- Use proof of correctness by observation but , it is simple for some algorithms like GCD , factorial and so on, for others it is quite complex
- Use Mathematical Induction.

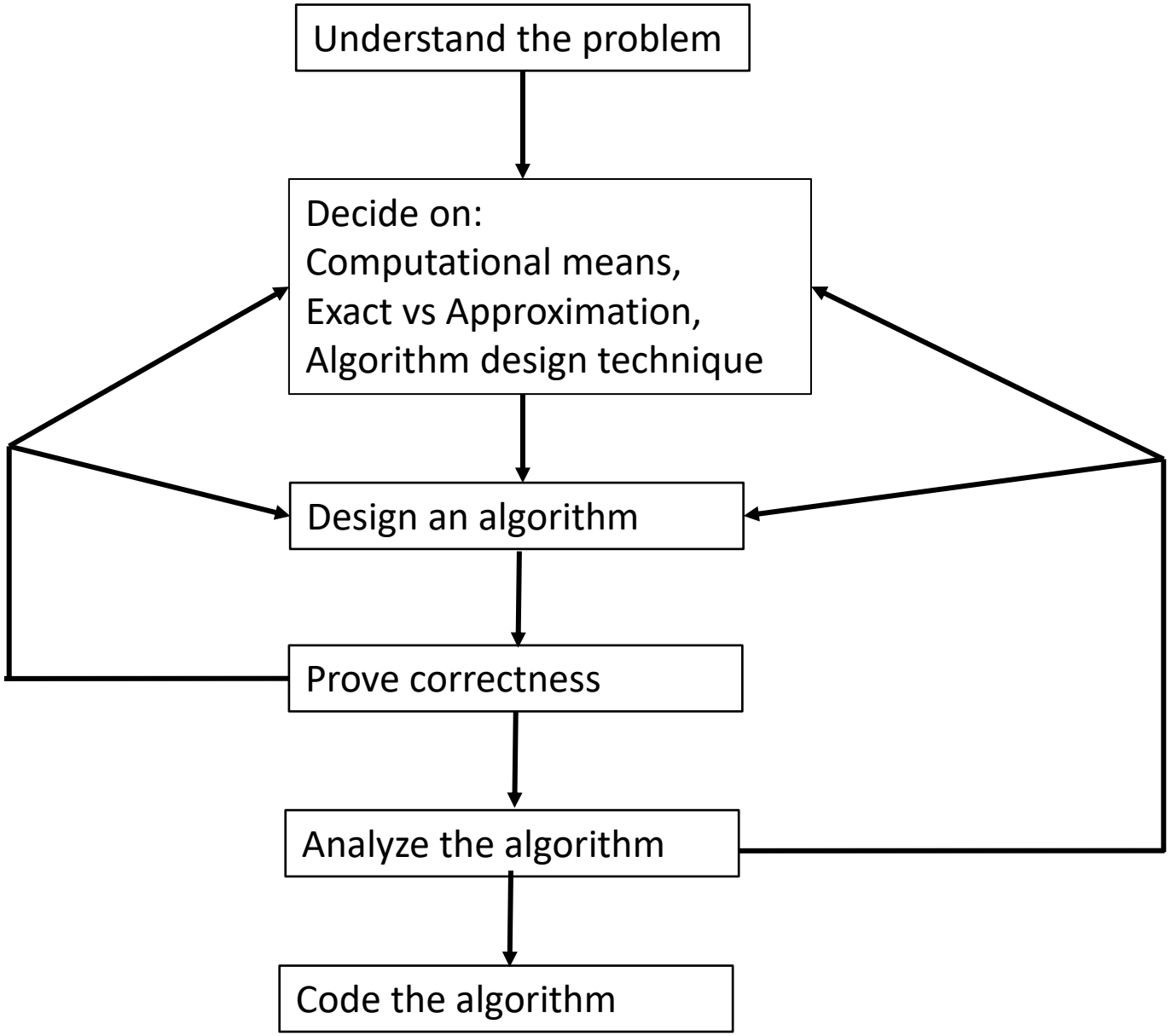
Analyzing an Algorithm

- Time efficiency and space efficiency
- Time efficiency indicate how fast the algorithm runs, where as space efficiency indicate how much extra memory the algorithm needs.

Coding an Algorithm

- Implemented as computer programs

Algorithm Design and Analysis Process



Fundamentals of Analysis of Algorithm efficiency

Why do we need to Analyze Algorithms?	*To pick the efficient algorithm among many algorithms(solutions) of the problem.*
What is Analyzing an Algorithm?	***Analyzing*** an algorithm means predicting the resources that the algorithm requires . The resources are ***time*** and ***space***.
<u>Framework for Analyzing algorithms</u>	
Language used for specification	Natural language – difficult because of inherent ambiguity. **Pseudocode** – A mixture of English language and programming language Flowchart – obsolete . Easy for simple algorithms
Assumptions for Analyzing an Algorithm?	Implementation model (for writing programs) : **Random Access machine (RAM) model** that contains instructions and data types like an ordinary computer , and instructions are executed sequentially.
Methods of Analyzing	1. **Apriori Estimates**(performance Analysis) 2. **Posteriori testing**(performance Measurement).

Define the terms running time and input size In general, the time taken by the algorithm grows with size of the input, so define running time as a function of input size.	
Input size	It depends on the problem. For example sorting- the input size is an array size, for multiplying two integers – input size is the binary representation of the integer, for graph problems – input size is nodes and edges.
Running time	The ***running time*** of an algorithm on a particular input is the number of primitive operations or **"steps"** executed.
Notion of step	A **step** is considered as machine-independent as possible (or) a line in pseudocode.
How to find the number of steps of an algorithm/pseudocode?	*Use* ***step count or frequency method*** *.* *It produces a function like* $\boldsymbol{2n^2+4n+100}$ **or** $\boldsymbol{4n^3+10}$ *,which is inconvenient for comparing the algorithms.* *So, we estimate at what rate the algorithms grows with the increase of input size, known as* ***rate of growth*** *or* ***order of growth***
Rate of growth or Order of growth	Considering only leading term(highest degree term) and dropping constants and other terms , which are insignificant for large values of input(n). i.e., for $2n^2+4n+100 = n^2$ *or* $4n^3+10 = n^3$
Asymptotic notation	Asymptotic notations provide a convenient way to represent the rate of growth of an algorithm and compare them. **(Big-Oh, Big-Omega, Big-Theta, small-oh, small-omega)**

Analysis of Algorithm

- The performance/efficiency of algorithms can be measured on the scales of **time** and **space.**
- The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program.

1) An analytical approach known as Apriori estimates (known as **performance analysis**)

2) An experimental study known as a posteriori testing(known as **performance measurement**)

In a A**priori analysis** we obtain a function (of some relevant parameters) which bounds the algorithm's computing time.

In a **posteriori testing** we collect actual statistics about the algorithm's consumption of time and space, while it is executing.(it is difficult to measure because , it contains actual running time + other processing time of the system to be considered)

A Priori Analysis

- In a **priori analysis** we obtain a function (of some relevant parameters) which bounds the algorithm's computing time.
- Uses **count or frequency table** method to find the order of magnitude of the algorithm in a function of input.

Frequency method (or) step count:-

It denotes the number of times a statement to be executed.

Generally, count value will be given depending upon corresponding statements.

- For comments, declarations the frequency count is zero (0).
- For Assignments, return statements the frequency count is 1.

Using count variable method

```
1    Algorithm Sum(a, n)
2    {
3        s := 0.0;
4        for i := 1 to n do
5            s := s + a[i];
6        return s;
7    }
```

```
1    Algorithm Sum(a, n)
2    {
3        s := 0.0;
4        count := count + 1; // count is global; it is initially zero.
5        for i := 1 to n do
6        {
7            count := count + 1; // For for
8            s := s + a[i]; count := count + 1; // For assignment
9        }
10       count := count + 1; // For last time of for
11       count := count + 1; // For the return
12       return s;
13   }
```

Step count is 2n+3

Using frequency table method

Statement	s/e	frequency	total steps
1 **Algorithm** Sum(a, n)	0	–	0
2 {	0	–	0
3 $s := 0.0$;	1	1	1
4 **for** $i := 1$ **to** n **do**	1	$n + 1$	$n + 1$
5 $s := s + a[i]$;	1	n	n
6 **return** s;	1	1	1
7 }	0	–	0
Total			$2n + 3$

Step count is 2n+3

Example -2

Statement	s/e	Frequency	Total steps
Algorithm matrix_add(int a[][],int b[][])			
{			
int i , j;			
int c[n][n];			
for i:=1 to n do			
{			
for j:=1 to n do			
{			
c[i][j] = a[i][j]+b[i][j];			
}			
}			
}			

Example -2

Statement	s/e	Frequency	Total steps
Algorithm matrix_add(int a[][],int b[][])	0	-	0
{	0	-	0
int i , j;	0	-	0
int c[n][n];	0	-	0
for i:=1 to n do	1	n+1	n+1
{	0	-	0
for j:=1 to n do	1	n*(n+1)	n*n+n
{	0	-	0
c[i][j] = a[i][j]+b[i][j];	1	n * n	n*n
}	0	-	0
}	0	-	0
}	0	-	0

Step count is $2n^2 + 2n + 1$

Example -2

Statement	s/e	Frequency	Total steps
Algorithm matrix_add(int a[][],int b[][]) {	0	0	0
int i , j;	0	-	0
int c[n][n];	0	-	0
for(i= 0 ; i < n ; i++) {	1	n+1	n+1
for (j =0; j < n ; j++) {	1	n*(n+1)	n*(n+1)
c[i][j] = a[i][j]+b[i][j];	1	n*n	n*n
}	0	-	0
}	0	-	0
	$2n^2+2n+1$		

Step count is $2n^2+2n+1$

Calculate the total number of steps using frequency method

statement	s/e	frequency		Total steps	
		n=0	n > 0	n=0	n > 0
1. Algorithm RSum(a,n)					
2. {					
3. if(n<=0) then					
4. return 0.0;					
5. else					
6. return Rsum(a,n-1)+a[n];					
7. }					

Calculate the total number of steps using frequency method

statement	s/e	frequency		Total steps	
		n=0	n > 0	n=0	n > 0
1. Algorithm RSum(a,n)	0	-	-	0	0
2. {	0	-	-	0	0
3. if(n<=0) then	1	1	1	1	1
4. return 0.0;	1	1	0	1	0
5. else	0	-	-	0	0
6. return Rsum(a,n-1)+a[n];	1 + x	0	1	0	1+x
7. }	0	-	-	0	
Total				2	2 + x

$$x = t_{\mathbf{Rsum}}(n-1)$$

Statement	s/e	frequency		total steps	
		$n=0$	$n>0$	$n=0$	$n>0$
1 **Algorithm** RSum(a,n)	0	–	–	0	0
2 {					
3 **if** $(n \leq 0)$ **then**	1	1	1	1	1
4 **return** 0.0;	1	1	0	1	0
5 **else return**					
6 RSum$(a,n-1)+a[n]$;	$1+x$	0	1	0	$1+x$
7 }	0	–	–	0	0
Total				2	$2+x$

$$x = t_{\mathsf{RSum}}(n-1)$$

```
1. Algorithm Fibonacci(n)
2. // Compute the nth Fibonacci number.
3. {
4.              if (n < 1) then
5.                   print(n);
6.              else
7.             {
8.                   fnm2=0;fnm1=0;
9.                   for i:=2 to n do
10.                  {
11.                        fn:=fnm1 + fnm2;
12.                        fnm2:= fnm1; fnm1=fn;
13.                  }
14.                  print(fn);
15.            }
16.}
```

	s/ e	n<=1	n>1	n<=1	n>1
1. Algorithm Fibonacci(n)	0	-	-	0	0
2. // Compute the n^{th} Fibonacci number.	0	-	-	0	0
3. {	0	-	-	0	0
4. if (n < 1) then	1	1	1	1	1
5. print(n);	1	1	0	1	0
6. else	0	-	-	0	0
7. {	0	-		0	0
8. fnm2=0;fnm1=0;	2	0	1	0	2
9. for i:=1 to n do	1	0	n+1	0	n+1
10. {	0	-	-	-	0
11. fn:=fnm1 + fnm2;	1	0	n	0	n
12. fnm2:= fnm1; fnm1=fn;	2	0	n	0	2n
13. }	0	-	-	-	0
14. print(fn);	1	0	1	0	1
15. }	0	-	-	0	0
16.}	0	-	-	0	0

If n>1 step count: **4n+5**

```
1. Algorithm Fibonacci(n)
2. // Compute the nth Fibonacci number.
3. {
4.         if (n < 1) then
5.             print(n);
6.         else
7.         {
8.             fnm2=0;fnm1=0;
9.             for i:=1 to n do
10.            {
11.                fn:=fnm1 + fnm2;
12.                fnm2:= fnm1; fnm1=fn;
13.            }
14.            print(fn);
15.        }
16.}
```