Unit-I

Introduction: Algorithm, Pseudo code for expressing algorithms, Performance Analysis-Space complexity, Time complexity, Asymptotic Notation- Big oh notation, Omega notation, Theta notation and Little oh notation, Probabilistic analysis, Amortized analysis.

Applications: Designing optimal solution with respect to time for a problem.

Algorithm

- Definition

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

- Characteristics of an algorithm.

  1. Input : Each algorithm must take zero or more number of inputs.
  2. Output : Each algorithm should produce one or more outputs.
  3. Definiteness : Each instruction is clear and unambiguous.

     Statements such as **"add 6 or 7 to x"**

     or

     "Compute 5/0" are not permitted"

4. Finiteness	: The algorithm should terminate after a finite number of steps.
5. Effectiveness	: Each step must be effective, and can be performed exactly in a finite amount of time.

Pseudo code for expressing algorithms

- We present most of our algorithms using pseudo code that looks like C and Pascal code.

1. Comments begin with // and continue until end of the line.
2. Blocks are indicated with matching braces: { and }.
   1. A compound statement.
   2. Body of a function.

3. i). An identifier begins with a letter.

ii). The data types of variables are not explicitly declared.

iii). The types will be clear from the context.

iv). Whether a variable is global or local to a function will also be clear from the context.

v). We assume simple data types such as integer, float, char, boolean, and so on.

vi). Compound data types can be formed with **records**.

```
node = record
        {
                datatype_1 data_1;
                        :
                datatype_n data_n;
                node    *link
        }
```

data items of a record can be accessed with → and period(.)

C style :-

```
struct  node
{
    datatype_1 data_1;
         :
    datatype_n  data_n;

    struct node  *link

}
```

4. Assignment of values to variables is done using the assignment statement.

   $$< \text{variable} > := < \text{expression} >$$

5. There are two boolean values **true** and **false.** To produce these values, logical operators **and**, **or** and **not** and the relational operators $<$, $\leq$, $=$, $\neq$, $\geq$ and $>$ are provided.

6. Elements of multidimensional arrays are accessed using [and]. For example the (i,j)th element of the array A is denoted as A[i,j].

7. The following looping statements are used: for, while, and repeat until.

The general form of a **while** loop:

```
while( condition ) do
{
        statement_1;
            :
        statement_n;
}
```

The general form of a **for** loop:

```
    for variable := value1 to value2  step step do
    {
            statement_1;
                :
            statement_n;
    }
```

– Here value1, value2, and step are arithmetic expressions.
– The clause "**step** *step*" is optional and taken as +1 if it does not occur.
– *step* could be either positive or negative.

```
Ex: 1:  for  i:= 1 to 10 step 2 do        // increment by 2, 5 iterations
    2:  for  i:= 1 to 10 do               // increment by 1, 10 iterations
```

- The for loop can be implemented as a while loop as follows:

```
variable:=value1;
incr:=step;
while( ( variable – value2)*step ≤ 0 ) do
{
     <statement 1>
                  :
    <statement n>
    variable :=variable+incr;
}
```

- The general form of a **repeat-until** loop:

```
repeat
        <statement 1>
             :
        <statement n>
until ( condition )
```

- The statements are executed as long as condition is false.

Ex:

```
repeat
        sum := sum + number;
        number := number - 1;
until   number = 0;
```

8. A conditional statement has the following forms:

```
**if**< $condition$ > **then** < $statement$ >
**if**< $condition$ > **then** < $statement$ 1> else
 < $statement$ 2>
```

9. Input and output are done using the instructions **read** and **write.**

10. Procedure or function starts with the word **Algorithm.**

General form :

```
Algorithm Name( <parameter list> )
{
        body
}
```

where *Name* is the name of the procedure.

- Simple variables to functions are passed by value.
- Arrays and records are passed by reference.

Ex:-Algorithm that finds and returns the maximum of n given numbers.

```
Algorithm max(a,n)
// a is an array of size n
{
        Result:=a[1];
        for i:=2 to n do
            if a[i]>Result  then Result:=a[i];
                        return Result;
}
```

Ex:-Write an algorithm to sort an array of n integers using bubble sort.

```
Algorithm sort(a,n)
  // a is an array of size n
  {
            for i:=1 to n-1 do
            {
                 for j:=1 to n-i do
                 {
                      if( a[j] >a[j+1] ) then
                           t=:a[j]; a[j]:=a[j+1]; a[j]:=t;
                 }
            }

  }
```

Space Complexity

- The space needed by an algorithm is called a space complexity.
- The space needed by an algorithm has the following components.
  - Instruction space
    - Programs space –It does not depend on the number of inputs.(Instance characteristics of a problem).
    - Data space
      - Constants & simple variables – Don't depend on the number of inputs.
      - Dynamically allocated objects - Depends on the number of inputs.

    -Stack space - Generally does not depend on the number of inputs unless recursive functions are in use.

Recursive algorithm

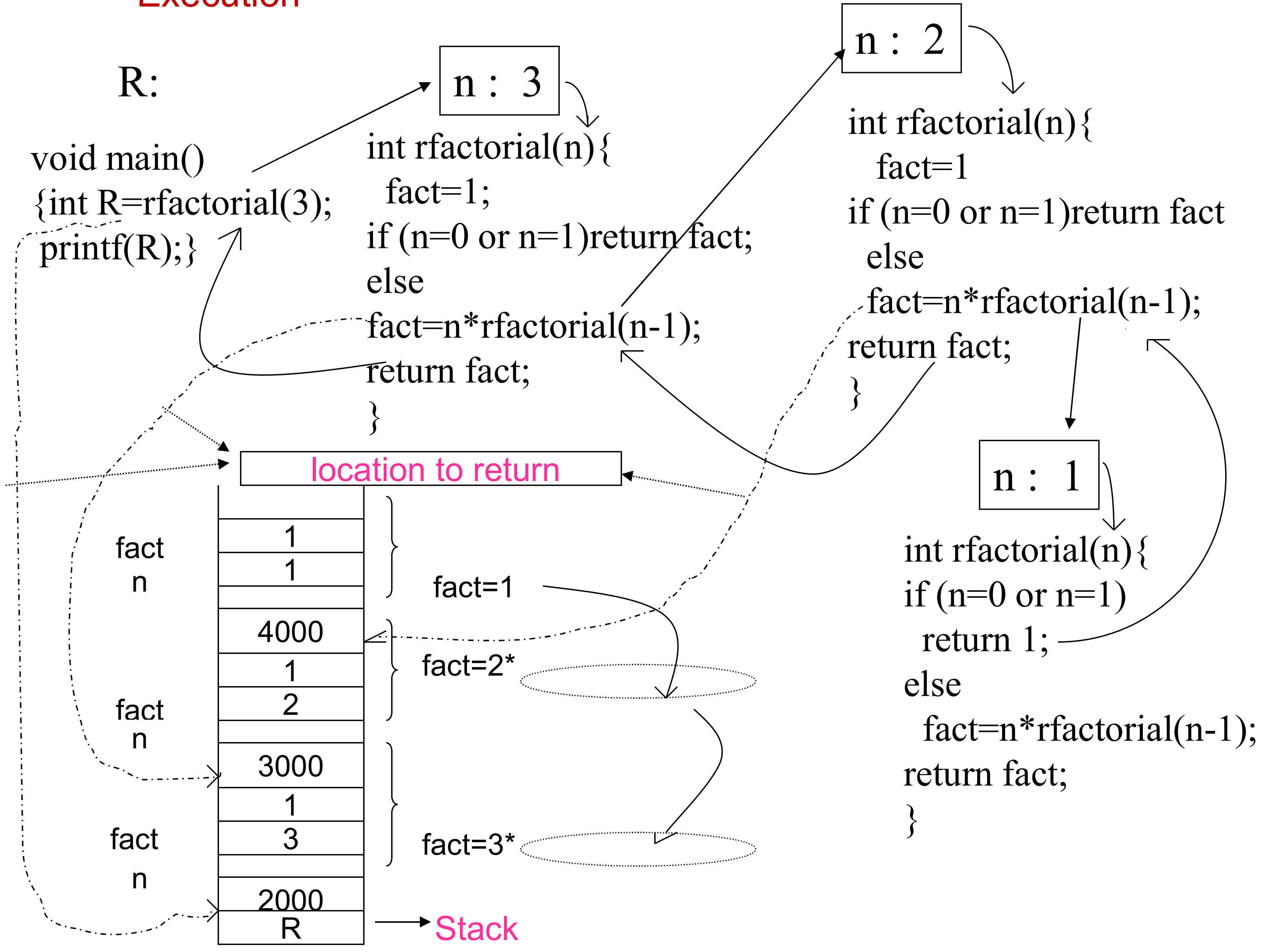
Algorithm rfactorial(n)

```
// n is an integer
{       fact=1;
        if(n=1 or n=0) return fact;
                else
                fact=n*rfactorial(n-1);
        return fact;
}
```

Each time the recursive function rfactorial is invoked,

the current values of n, fact and the program location to return to on completion are saved on the stack.

Execution



```
void main()
{int R=rfactorial(3);
 printf(R);}
```

```
int rfactorial(n){
  fact=1;
if (n=0 or n=1)return fact;
else
fact=n*rfactorial(n-1);
return fact;
}
```

```
int rfactorial(n){
   fact=1
if (n=0 or n=1)return fact
  else
 fact=n*rfactorial(n-1);
return fact;
}
```

```
int rfactorial(n){
if (n=0 or n=1)
  return 1;
else
  fact=n*rfactorial(n-1);
return fact;
}
```

Contd..

Therefore, We can divide the total space needed by a program into two parts:

i) Fixed Space Requirements (*C*)

Independent of the characteristics of the problem instance (I)

- instruction space
- space for simple variables and constants.

ii) Variable Space Requirements ($S_P(I)$)

depend on the characteristics of the problem instance (I)

- Number of inputs associated with I
- Recursive stack space (formal parameters, local variables, return address).

– Therefore, the space requirement of any problem P can be written as

$S(p)=\mathbf{C} + \mathbf{S_p}($ **Instance characteristics** $\mathbf{)}$.

Note:

- We concentrate only on estimating S_p (Instance characteristics).
- We don’t concentrate on estimating fixed part c .
- We need to identify the instance characteristics of the problem to measure S_p

Example1

```
Algorithm abc(a,b,c)
{
   return a+b+b*c+(a+b-c)/(a+b)+4.0;
}
```

- Problem instance characterized by the specific values of a,b,and c.
- If ***we assume*** one word (4 bytes) is adequate to store the values of each a, b, and c , then the space needed by abc is independent of the instance characteristics.

  Therefore, S_{abc}(instance characteristics)=0

Example2

```
Algorithm sum(a,n)
{
    s:=0;
    for i:=1 to n do
            s:=s+a[i];
      return s;
}
```

- Problem instance characterized by n.
- The amount of space depends on the value of n.

Therefore, $S_{sum}(n) = n$

Example3

```
Algorithm RSum(a,n)
{
    if(n ≤ 0) then return 0;
    else return RSum(a,n-1)+a[n];
}
```

Type	Name	Number of bytes
formal parameter: int	a	2
formal parameter: int	n	2
return address (used internally)		2
Total per one recursive call		6

Total no.of recursive calls n, therefore **$S_{RSum}(n)=6n$**

Contd..

- Time Complexity:

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

  - Time Complexity, $T(P)$ is the sum of its compile time C plus its run time, $T_P(I)$.
  - Compile time does not depend on the instance characteristics(*Number of inputs*).
  - We will concentrate on estimating run time $T_p(I)$.

- If we know the compiler used, then we can find out the
  - No. of additions , subtractions, and so on.
- Then we can obtain an expression for $T_p(n)$ Of the form

$$T_P(n)=c_a ADD(n)+c_s SUB(n)+c_m MUL(n)+c_d DIV(n)+\ldots\ldots\ldots$$

where,

  - c_a, c_s, c_m, c_d and so on denote the time required for an addition , subtraction and so on.
  - ADD, SUB, MUL, DIV, and so denote no.of additions, subtractions, and so on.

- Obtaining such an exact formula is an impossible, beause time required for an addition, subtraction, and so on, depends an numbers being added, subtracted, and so on.

- So, the value of $T_p(n)$ for any given n can be obtained only experimentally.

- Even with the experimental approach, we face difficulties.

- The execution time of a program p depends on the number of other programs running on the computer at the time program p is running.

Contd..

- As there are some problems in finding the execution time using earlier methods, we will go one step further and count only the number of *program steps.*

program step

program step is *loosely* defined as a syntactically or semantically meaningful program *statement* whose execution time is independent of the *number of inputs.*

Example :

```
result = a + b + b * c + (a + b - c) / (a + b) + 4.0;
sum = a + b + c;
```

- The number of steps assigned to any *program statement* depends on the kind of statement.

- Comments are counted as zero number of steps.

- An assignment statement which does not involve any function calls to other functions counted as one step.

- For loops, such as the for, while, and repeat-until, we consider the step counts only for the control part of the statement.

- The control parts for for and while statements have the following forms:

```
for i:= <expr1> to <expr2> do
while ( <expr> ) do
```

- Each execution of the control part of a while statement is one, unless <expr> is a function of instance characteristics.

- The step count for each execution of the control part of a for statement is one, unless <expr1> and <expr2> are functions of the instance characteristics.

- The step count for each execution of the condition of a conditional statements is one, unless condition is a function of instance characteristics.

- If any statement (assignment statement, control part, condition etc.) involves function calls, then the step count is equal to the number of steps assignable to the function plus one.

- Methods to compute the step count

  1) Introduce global variable count into programs with initial value zero.
     - Statements to increment count by the appropriate amount are introduced into the program.
     - The value of the count by the time program terminates is the number steps taken by the program.

  2) Tabular method
     - Determine the total number of steps contributed by each statement per execution × frequency
     - Add up the contribution of all statements

Method-I: Introduce variable count into programs

EX:- Iterative sum of n numbers

```
Algorithm sum(a, n)
{
        s:=0;
        count:=count+1;  // for assignment statement
        for i:=1 to n do
        {     count:=count+1; // For for
              s:=s+a[i];
              count:=count+1; // for assignment statement
        }
            count:=count+1; // for last time of for
        return s;
    count:=count+1; // for return
}
```

2n + 3 steps

Note : *Step count tells us how the run time for a program changes with changes in the instance characteristics.*

Ex:- Addition of two m×n matrices

```
Algorithm Add(a,b,c,,m,n)
{
    for i:=1 to m do
    {
                for j:=1 to n do
                {
                    c[i,j]:=a[i,j]+b[i,j];

                }
    }

 }
```

2mn + 2m+1 steps

EX:- Recursive sum of n numbers

```
Algorithm RSum(a,n)
{
      count:=count+1; // for the if conditional
      if(n ≤ 0) then
      {
                 return 0;
                count:=count+1; // for the  return
      }
      else
     {
                return RSum(a,n-1)+a[n];
                count:=count+1;   // For the addition, function invocation and return
      }
}
```

- When analyzing a recursive program for its step count, we often obtain a recursive formula for the step count.
- We obtain the following recursive formula for above (RSum)

$$t_{RSum}(n) = \begin{cases} 2 & \text{If } n=0 \\ 2 + t_{RSum}(n-1) & \text{If } n>0 \end{cases}$$

- One way of solving such recursive formula is by repeated substitutions for each occurrence of the function t_{RSum} on the right side until all such occurrences disappear:

$$
\begin{aligned}
t_{RSum}(n) &= 2+t_{RSum}(n-1) \\
&= 2+2+t_{RSum}(n-2) \\
&= 2(2)+t_{RSum}(n-2) \\
&\vdots \\
&\vdots \\
&= n(2)+t_{RSum}(n-n) \\
&= 2n+t_{RSum}(0) \\
&= 2n+2
\end{aligned}
$$

The step count for Rsum is 2n+2

Method-II: Tabular method

- EX:- Iterative sum of n numbers

Statement	s/e	frequency	Total steps
Algorithm sum(a, n)	0	--	0
{	0	--	0
s:=0 ;	1	1	1
for i:=1 to n do	1	n+1	n+1
s:=s+a[i];	1	n	n
return s;	1	1	1
}	0	--	0
Total			2n+3

- EX:- Addition of two m×n matrices

Statement	s/e	frequency	Total steps
Algorithm Add(a,b,c,m, n)	0	--	0
{	0	--	0
for i:=1 to m do	1	m+1	m+1
for j:=1 to n do	1	m(n+1)	mn+m
c[i,j]:=a[i,j]+b[i,j] ;	1	mn	mn
}	0	--	0
Total			2mn+2m+1

- EX:- Recursive sum of n numbers

Statement	s/e	Frequency		Total steps	
		n=0	n>0	n=0	n>0
Algorithm RSum(a,n)	0	--	--	0	0
{	0	--	--	0	0
if(n ≤ 0) then	1	1	1	1	1
return 0;	1	1	0	1	0
else return					
Rsum(a,n-1)+a[n] ;	1+x	0	1	0	1+x
}	0	--	--	0	0
Total				2	2+x

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

Best, Worst, Average Cases

- ✓ All the inputs of a given size for a given problem, don't take the same number of program steps.
- ✓ Sequential search for *K* in an array of *n* integers:
  - Begins at the first element in array and look at each element in turn until *K* is found.

1. Best-Case Step count:-

   Minimum number of steps executed by the algorithm for the given parameters.

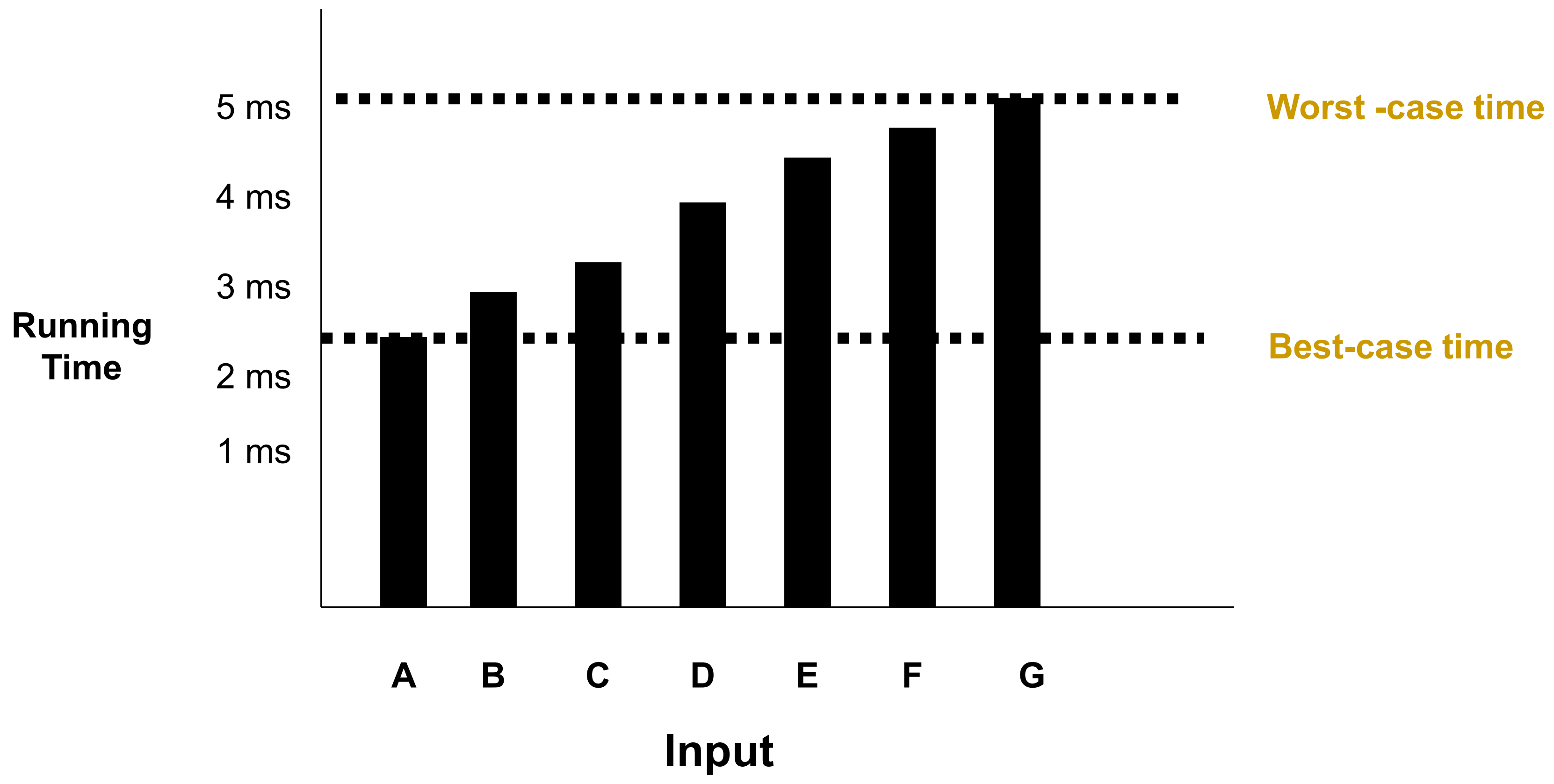
2. Worst-Case Step count:-

   Maximum number of steps executed by the algorithm for the given parameters.

3. Average-Case Step count:-

   Average number of steps executed by an algorithm.

Contd..



$$\text{Average-case time} = \frac{\text{A time + B time+ ....+G time}}{\text{7 (total number of possible inputs)}}$$

Inexactness of step count.

- Both the instructions x=y; and x=y+z+(x/y) count as one step.
- Therefore, two analysts may arrive at $4n^2+6n+2$ and $7n^2+3n+4$ as the step count for the same program.

- Because of the inexactness of a step count , the exact step count is not very useful for comparison of algorithms.

Asymptotic efficiency

- Asymptotic efficiency means study of algorithms *efficiency* for large inputs.

- To compare two algorithms with running times *f(n)* and *g(n),* we need a rough measure that characterizes how fast each function grows as n grows.

- *Hint:* use *rate of growth*
- Compare functions **asymptotically!**

  (i.e., for large values of *n*)

Rate of Growth

- Ex:- $F(n)=n^2+100n+\log_{10}n+1000$

n	f(n)	N2		100n		log10n		1000	
	value	Value	%	value	%	value	%	value	%
1	1,101	1	**0.1**	100	**9.1**	0	**0.0**	1,000	**90.83**
10	2,101	100	**4.76**	1,000	**47.6**	1	**0.05**	1,000	**47.60**
100	21,002	10,000	**47.6**	10,000	**47.6**	2	**0.001**	1,000	**4.76**
1,000	1,101,003	1,000,000	**90.8**	100,000	**9.1**	3	**0.0003**	1,000	**0.09**
10,000	101,001,004	100,000,000	**99.0**	1,000,000	**0.99**	4	**0.0**	1,000	**0.001**
1,00000	10,010,001005	10,000,000,000	**99.9**	10,000,000	**0.099**	5	**0.0**	1,000	**0.00**

- The low order terms and constants in a function are relatively insignificant for **large** *n*

$$n^2 + 100n + \log_{10}n + 1000 \quad \sim \quad n^2$$

i.e., we say that $n^2 + 100n + \log_{10}n + 1000$ and n^2 have the same **rate of growth**

Some more examples

- $n^4 + 100n^2 + 10n + 50$ is $\sim n^4$
- $10n^3 + 2n^2$ is $\sim n^3$
- $n^3 - n^2$ is $\sim n^3$
- constants
  - 10 is ~ 1
  - 1273 is ~ 1

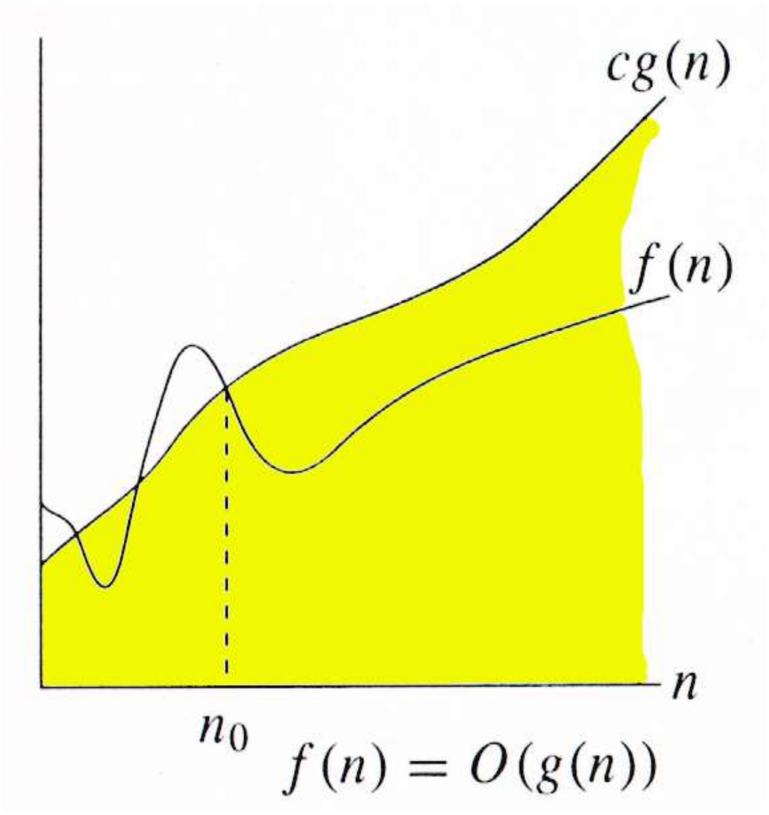
Asymptotic Notations

- Asymptotic notation describes the behavior of functions for the large inputs.
- **Big Oh(*O*)** notation:
  - The big oh notation describes an upper bound on the asymptotic growth rate of the function f.

Definition: [Big "oh'']

- f(n) = O(g(n)) (read as "*f* of *n* is big oh of *g* of *n*") iff there exist positive constants c and n_0 such that

  $f(n) \le cg(n)$ for all n, $n \ge n_0$.



- The definition states that the function f(n) is at most c times the function g(n) except when n is smaller than n_0.
- *In other words, $f(n)$ grows slower than or same rate as" g(n).*
- When providing an upper –bound function g for f, we normally use a single term in n.
- **Examples**
  - $f(n) = 3n+2$
    - $3n + 2 <= 4n$, for all $n >= 2$, $\therefore 3n + 2 = \boldsymbol{O(n)}$
  - $f(n) = 10n^2+4n+2$
    - $10n^2+4n+2 <= 11n^2$, for all $n >= 5$, $\therefore 10n^2+4n+2 = \boldsymbol{O(n^2)}$
  - $f(n)=6*2^n+n^2=\boldsymbol{O(2^n)}$ /* $6*2^n+n^2 \leq 7*2^n$ for $n \geq 4$ */

- It also possible to write $10n^2+4n+2 = O(n^3)$ since $10n^2+4n+2 <= 7n^3$ for $n>=2$

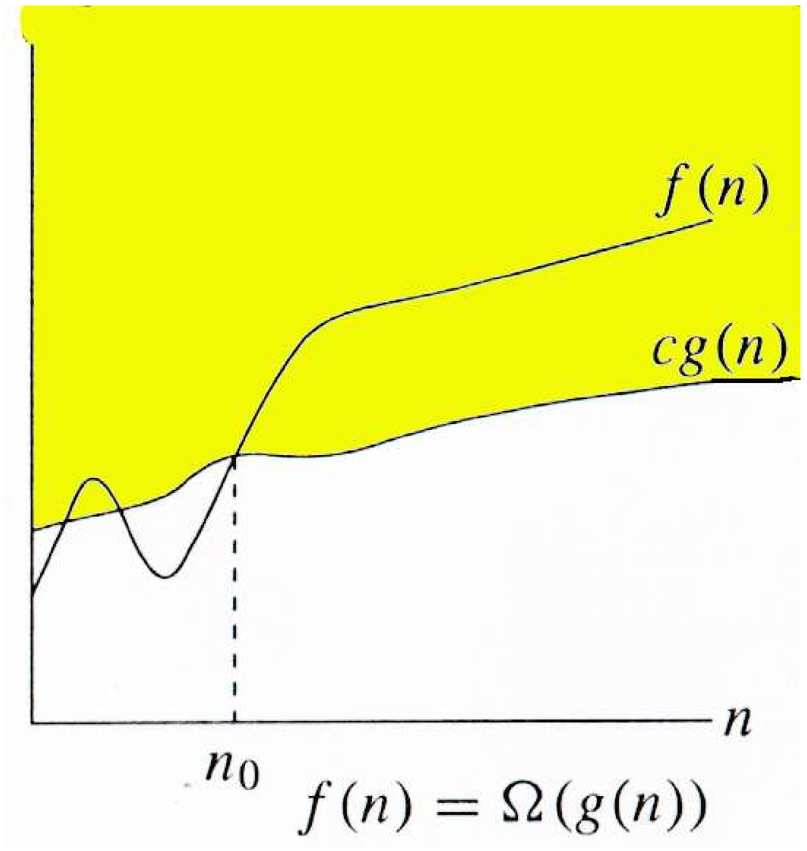
- Although n^3 is an upper bound for $10n^2+4n+2$, it is not a tight upper bound; we can find a smaller function (n^2) that satisfies big oh relation.

- But, we can not write $10n^2+4n+2 = O(n)$, since it does not satisfy the big oh relation for sufficiently large input.

- ***Omega* (Ω)** notation:
  - The omega notation describes a lower bound on the asymptotic growth rate of the function f.

Definition: [Omega]

- $f(n) = \Omega(g(n))$ (read as "*f* of *n* is omega of *g* of *n*") iff there exist positive constants *c* and n_0 such that $f(n) \geq cg(n)$ for all *n*, $n \geq n_0$.

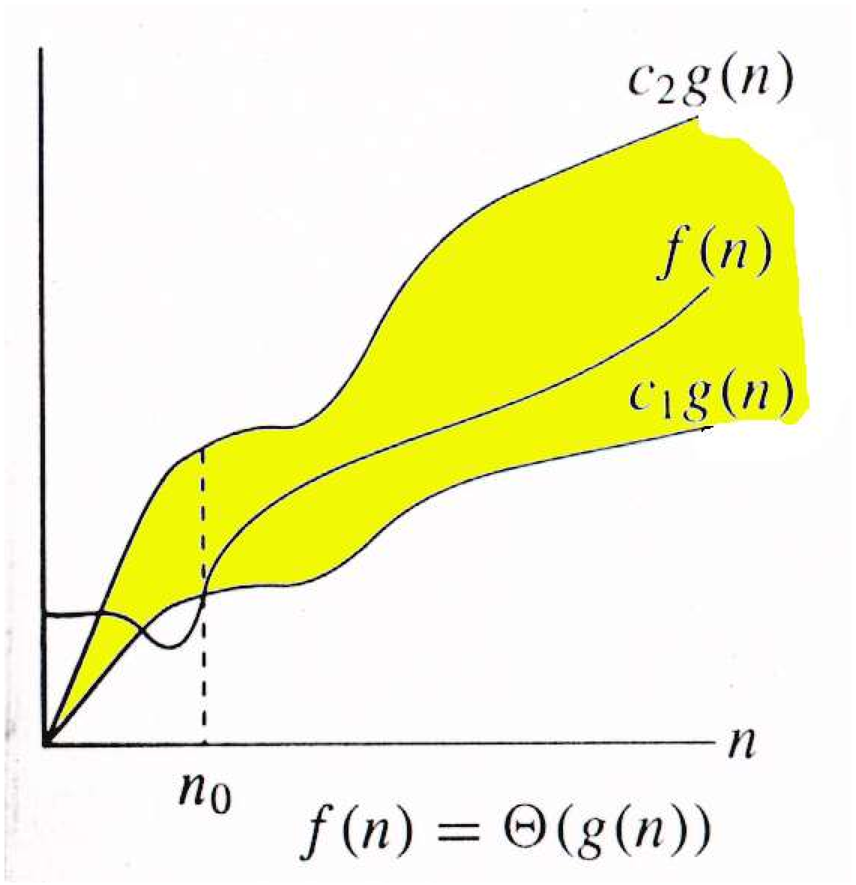


- The definition states that the function f(n) is at least c times the function g(n) except when n is smaller than n_0.
- *In other words,f(n)* grows faster than or same rate as" g(n).
- **Examples**
  - $f(n)$ = 3n+2
    - 3n + 2 >= 3n, for all n >= 1, $\therefore$ 3n + 2 = $\Omega(n)$
  - $f(n) = 10n^2+4n+2$
    - $10n^2+4n+2 >= n^2$, for all n >= 1, $\therefore$ $10n^2+4n+2 = \Omega(n^2)$
- It also possible to write $10n^2+4n+2 = \Omega(n)$ since $10n^2+4n+2 >= n$ for n>=0
- Although n is a lower bound for $10n^2+4n+2$, it is not a tight lower bound; we can find a larger function (n^2) that satisfies omega relation.
- But, we can not write $10n^2+4n+2 = \Omega(n^3)$, since it does not satisfy the omega relation for sufficiently large input.

- ***Theta*** **(Θ)** notation:
  - The Theta notation describes a tight bound on the asymptotic growth rate of the function f.

Definition: [Theta]

- $f(n) = \Theta(g(n))$ (read as "*f* of *n* is theta of *g* of *n*") iff there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$.



- The definition states that the function f(n) lies between c1 times the function g(n) and c2 times the function g(n) except when n is smaller than n_0.
- *In other words,f(n)* grows same rate as" g(n).
- *Examples:-*
  - $f(n) = 3n+2$
    - $3n <= 3n + 2 <= 4n$, for all $n >= 2$, $\therefore$ $3n + 2 = \Theta(n)$
  - $f(n) = 10n^2+4n+2$
    - $n^2 <= 10n^2+4n+2 <= 11n^2$, for all $n >= 5$, $\therefore$ $10n^2+4n+2 = \Theta(n^2)$
- But, we can not write either $10n^2+4n+2= \Theta(n)$ or $10n^2+4n+2= \Theta(n^3)$, since neither of these will satisfy the theta relation.

- **Little Oh(*o*)** notation:
  - The little oh notation describes a strict upper bound on the asymptotic growth rate of the function f.

Definition: [Little “oh”]

  - f(*n*) = o(g(*n*)) (read as “*f* of *n* is little oh of *g* of *n*”) iff

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

- The definition states that the function f(n) is less than c times the function g(n) except when n is smaller than n_0.
- *In other words,* *f*(*n*) grows slower than” g(n).
- **Examples**
  - *f*(*n*) = $3n+2=o(n^2)$
    - Since

$$\lim_{n\to\infty} \frac{3n+2}{n^2} = 0$$

  - *However,* *$3n+2 \neq o(n)$*

  - *The little oh notation is often used in step-count analyses.*
  - *The step count of $3n+o(n)$ would mean that the step count is 3n plus terms that are asymptotically smaller than n.*
  - *When performing such an analysis , one can ignore portion of the program that are known to contribute less than* Θ(n) steps.

Big-Oh, Theta, Omega and Little-oh

Tips :

- Think of $O(g(n))$ as "less than or equal to" g(n)
  - Upper bound: "grows slower than or same rate as" g(n)

- Think of $\Omega(g(n))$ as "greater than or equal to" g(n)
  - Lower bound: "grows faster than or same rate as" g(n)

- Think of $\Theta(g(n))$ as "equal to" g(n)
  - "Tight" bound: same growth rate

- Think of $o(g(n))$ as "less than to" g(n)
  - Strict Upper bound: "grows slower than " g(n)

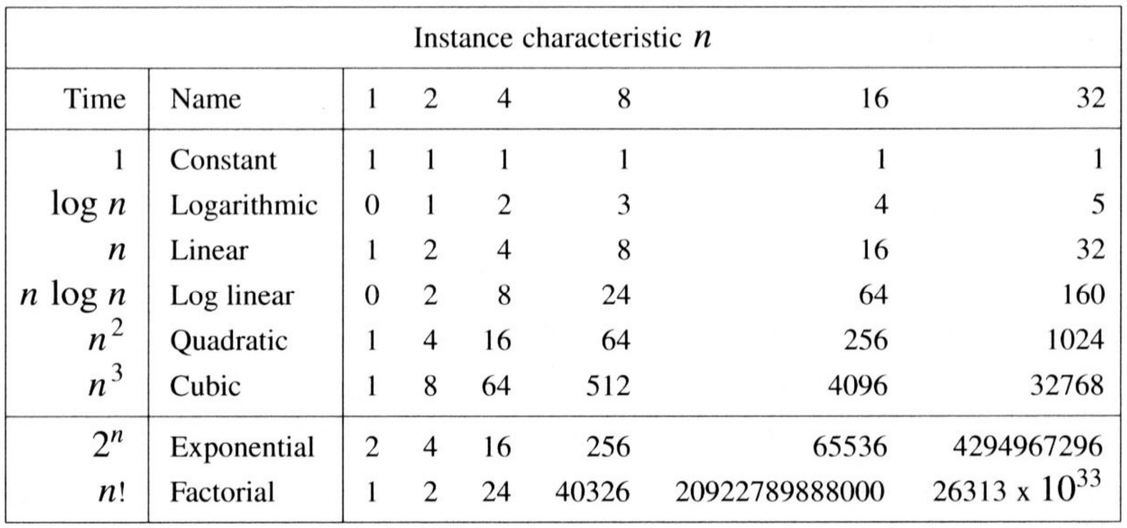
- (*True for large N*)

Functions ordered by growth rate

Function	**Name**
1	Growth is constant
logn	Growth is logarithmic
n	Growth is linear
nlogn	Growth is n-log-n
n^2	Growth is quadratic
n^3	Growth is cubic
2^n	Growth is exponential
n!	Growth is factorial

$1 < \log n < n < n\log n < n^2 < n^3 < 2^n < n!$

– To get a feel for how the various functions grow with n, you are advised to study the following figs:



		Instance characteristic n					
Time	Name	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 Function values

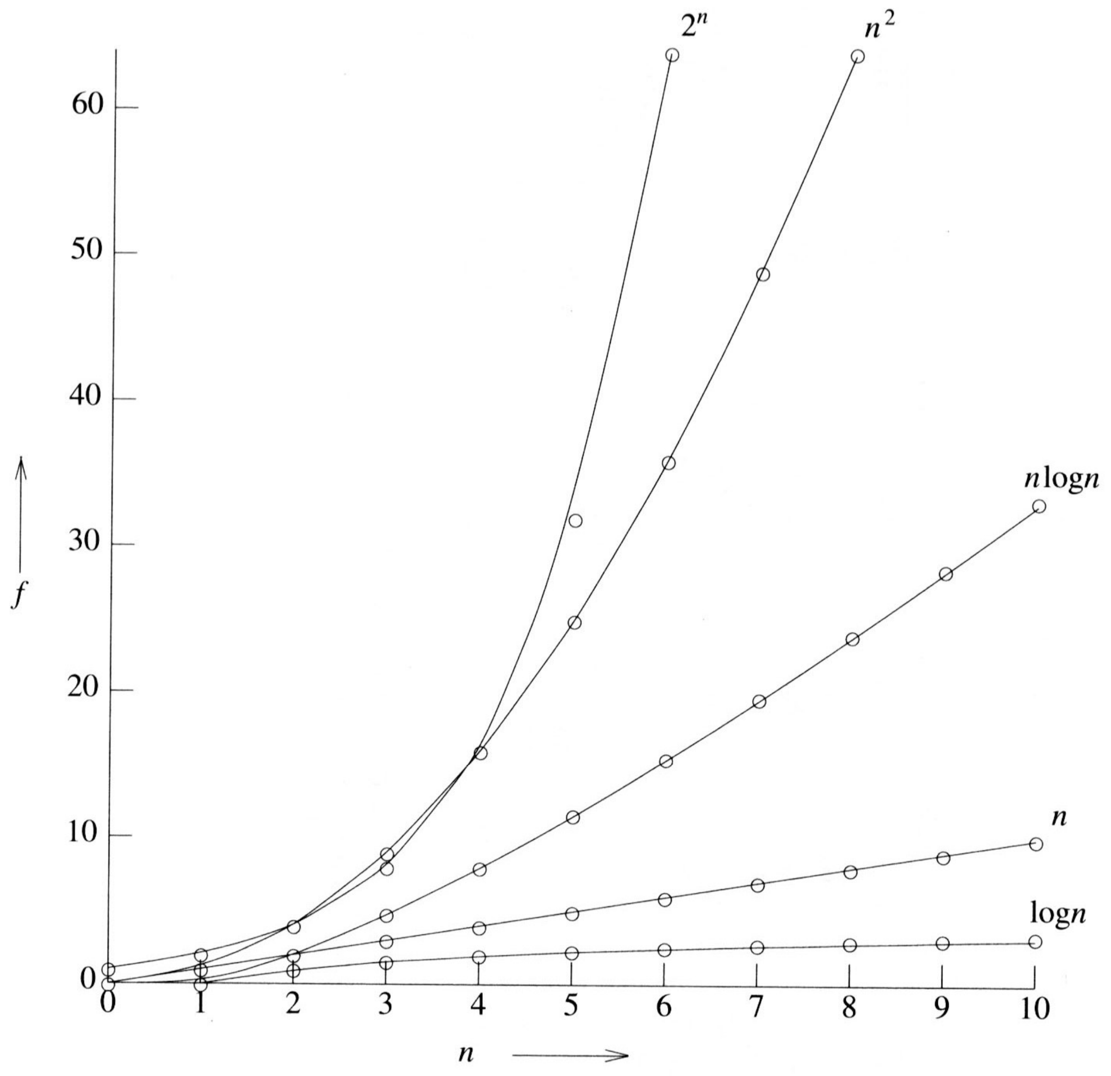
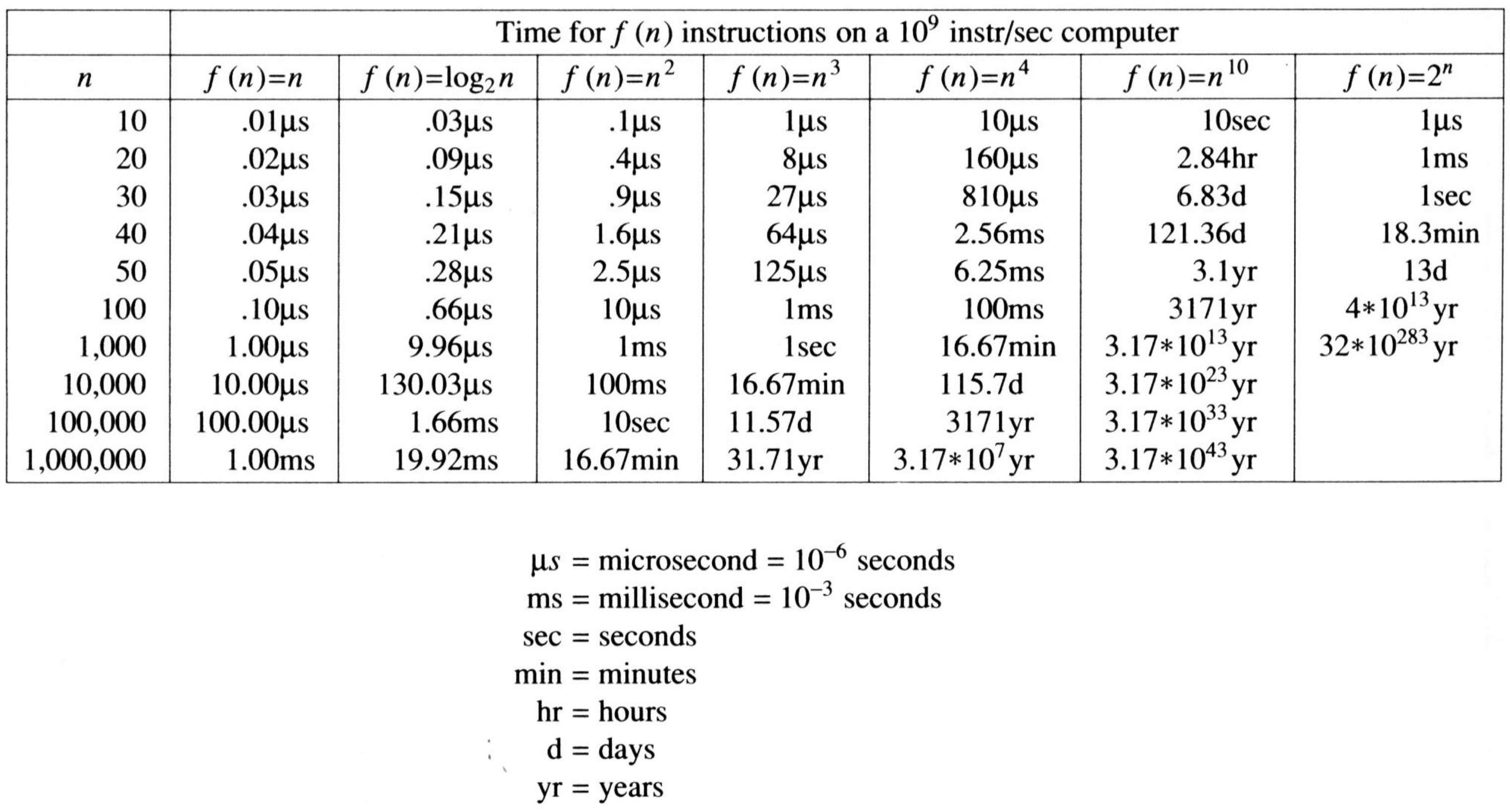


Figure 2 Plot of function values

- The following fig gives the time needed by a 1 billion instructions per second computer to execute a program of complexity *f*(*n*) instructions.

Times on a 1 billion instruction per second computer



	Time for $f(n)$ instructions on a 10^9 instr/sec computer						
n	$f(n)=n$	$f(n)=\log_2 n$	$f(n)=n^2$	$f(n)=n^3$	$f(n)=n^4$	$f(n)=n^{10}$	$f(n)=2^n$
10	.01µs	.03µs	.1µs	1µs	10µs	10sec	1µs
20	.02µs	.09µs	.4µs	8µs	160µs	2.84hr	1ms
30	.03µs	.15µs	.9µs	27µs	810µs	6.83d	1sec
40	.04µs	.21µs	1.6µs	64µs	2.56ms	121.36d	18.3min
50	.05µs	.28µs	2.5µs	125µs	6.25ms	3.1yr	13d
100	.10µs	.66µs	10µs	1ms	100ms	3171yr	$4*10^{13}$ yr
1,000	1.00µs	9.96µs	1ms	1sec	16.67min	$3.17*10^{13}$ yr	$32*10^{283}$ yr
10,000	10.00µs	130.03µs	100ms	16.67min	115.7d	$3.17*10^{23}$ yr	
100,000	100.00µs	1.66ms	10sec	11.57d	3171yr	$3.17*10^{33}$ yr	
1,000,000	1.00ms	19.92ms	16.67min	31.71yr	$3.17*10^{7}$ yr	$3.17*10^{43}$ yr	

µs	= microsecond = 10^{-6} seconds
ms	= millisecond = 10^{-3} seconds
sec	= seconds
min	= minutes
hr	= hours
d	= days
yr	= years

Probabilistic analysis

The Hiring Problem

- You are using an employment agency to hire a new assistant.
- The agency sends you one candidate each day.
- You interview the candidate and must immediately decide whether or not to hire that person. But if you hire, you must also fire your current office assistant—even if it's someone you have recently hired.
- Cost to interview is c_i per candidate.
- Cost to hire is c_h per candidate.
- You want to have, at all times, the best candidate seen so far.
- When you interview a candidate who is better than your current assistant, you fire the current assistant and hire the candidate.
- You will always hire the first candidate that you interview.
- Problem: What is the cost of this strategy?

Pseudo-code to Model the Scenario

```
Hire-Assistant (n)
best ← 0  ;;Candidate 0 is a least qualified candidate
for i ← 1 to n
    do interview candidate i
        if candidate i is better than candidate best
            then best ← i
                hire candidate i
```

Cost Model:

- We are not concerned with the running time of Hire-Assistant.
- We want to determine the total cost of hiring the best candidate.
- If n candidates interviewed and m hired, then cost is nc_i+mc_h.
- Have to pay nc_i to interview, no matter how many we hire.
- So, focus on analyzing the hiring cost mc_h.
- m varies with order of candidates.

Best-Case Analysis

- Best case is when the agency sends you the best applicant on the first day.
- Cost is nc_i+c_h.

Worst-case Analysis

- In the worst case, we hire all *n* candidates.
- This happens if each candidate is better than all those who came before. Candidates come in increasing order of quality.
- Cost is $nc_i + nc_h$.
- If this happens, we fire the agency.
- What should happen in the average case?

Probabilistic analysis

- Probabilistic analysis is the use of *probability* in the analysis of problems.
- Generally, we use probabilistic analysis to *analyze the running time* of an algorithm.
- Sometimes, we can also use it to analyze *quantities* such as the *hiring cost* in procedure Hire-Assistant.

Hiring problem-Average –case Analysis

- An input to the hiring problem is an ordering of the n applicants, there are n! different inputs.
- To use probabilistic analysis, we assume that the candidates arrive in a random order .
- Then we analyze our algorithm by computing an expected running time.
- The expectation is taken over the distribution of the possible inputs.
- Thus, we are averaging the running time over all possible inputs.

Indicator Random Variable

- A powerful technique for computing the expected value of a random variable.
- Convenient method for converting between probabilities and expectations.
- Takes only 2 values, 1 and 0.
- Indicator Random Variable $X_A = I\{A\}$ for an event A of a sample space is defined as:

$$I\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs,} \\ 0 & \text{if } A \text{ does not occur.} \end{cases}$$

- *The expected value of an indicator Random Variable associated with an event A is equal to the probability that A occurs.*

Indicator RV – Example1

Problem: Determine the expected number of heads in *one* coin flip.

- Sample space is s{ H,T}
- Indicator random variable

  X_A= I{ coin coming up with heads}=1/2
- The expected number of heads obtained in one flip of the coin is equal to the expected value of indicator random variable.

  Therefore, $E[X_A] = 1/2$

Indicator RV – Example2

- Problem: Determine the expected number of heads in n coin flips.
- Let X be the random variable denoting the total number of heads in n coin flips.
- Define n indicator random variables, X_i, $1 \leq i \leq n$.
- Let X_i be the indicator random variable for the event that the i^{th} flip results in a Head.
- $X_i = I\{\text{the } i^{th} \text{ flip results in } H\}$
- Then $X = X_1 + X_2 + \ldots + X_n = \sum_{i=1..n} X_i$.
- $E[X_i] = \Pr\{H\} = ½$, $1 \leq i \leq n$.
- Expected number of heads is

$$\begin{aligned} E[X] &= E[\textstyle\sum_{i=1..n} X_i]. \\ &= \textstyle\sum_{i=1..n} E[X_i] = \sum_{i=1..n} ½ = n/2. \end{aligned}$$

Back to the Hiring Problem

- We want to know the expected cost of our hiring algorithm, in terms of how many times we hire an applicant.
- Random variable X(s) is the number of applicants that are hired, given the input sequence s.
- Indicator random variable X_i for applicant i = 1 if applicant i is hired, 0 otherwise.
- What is E[X]?

Probability of Hiring i-th Applicant

- Probability of hiring i is probability that i is better than the previous i-1 applicants…

  $\Pr[X_i = 1] = 1/i$.

- Suppose n = 4 and i = 3.

Ex:-Probability that 3rd applicant is better than the 2 previous ones.

$\boxed{\mathbf{1234}}$	$\boxed{2134}$	3124	4123
$\boxed{\mathbf{1243}}$	$\boxed{\mathbf{2143}}$	$\boxed{3142}$	4132
1324	**2314**	3214	4213
$\boxed{\mathbf{1342}}$	$\boxed{\mathbf{2341}}$	$\boxed{3241}$	4231
1423	**2413**	3412	4312
1432	**2431**	3421	4321

8/24 = 1/3

Analysis of the Hiring Problem

- Compute E[*X*], the number of candidates we expect to hire.

$$E[X] = E\left[\sum_{i=1}^{n} X_i\right]$$

$$= \sum_{i=1}^{n} E[X_i]$$

$$= \sum_{i=1}^{n} \frac{1}{i}$$

$$= \ln n + \quad 1$$

Expected hiring cost = $O(c_h \ln n)$.

Amortized Analysis

- Data structures are subjected to a sequence of operations rather than one operation.
- In this sequence, one operation possibly performs certain modifications that have an impact on the run time of the next operation in the sequence.
- One way of assessing the worst case run time of the entire sequence is to add worst case efficiencies for each operation.
- This may result in an excessively large and unrealistic time complexity.

- Amortized analysis concentrates on finding the average complexity of a worst case sequence of operations.
- Amortized analysis considers interdependence between operations.
  - For example, if an array is sorted and only a very few new elements are added, then re-sorting this array will be must faster than sorting it for the first time.

Amortized Analysis

- **AMORTIZED ANALYSIS:**

You try to estimate an upper bound of the **total work T(n)** required for a sequence of **n** operations…

Some operations may be cheap some may be expensive. Overall, your algorithm does **T(n) of work for n operations …**

Therefore, by simple reasoning, the ***amortized cost*** of each operation is **T(n)/n**