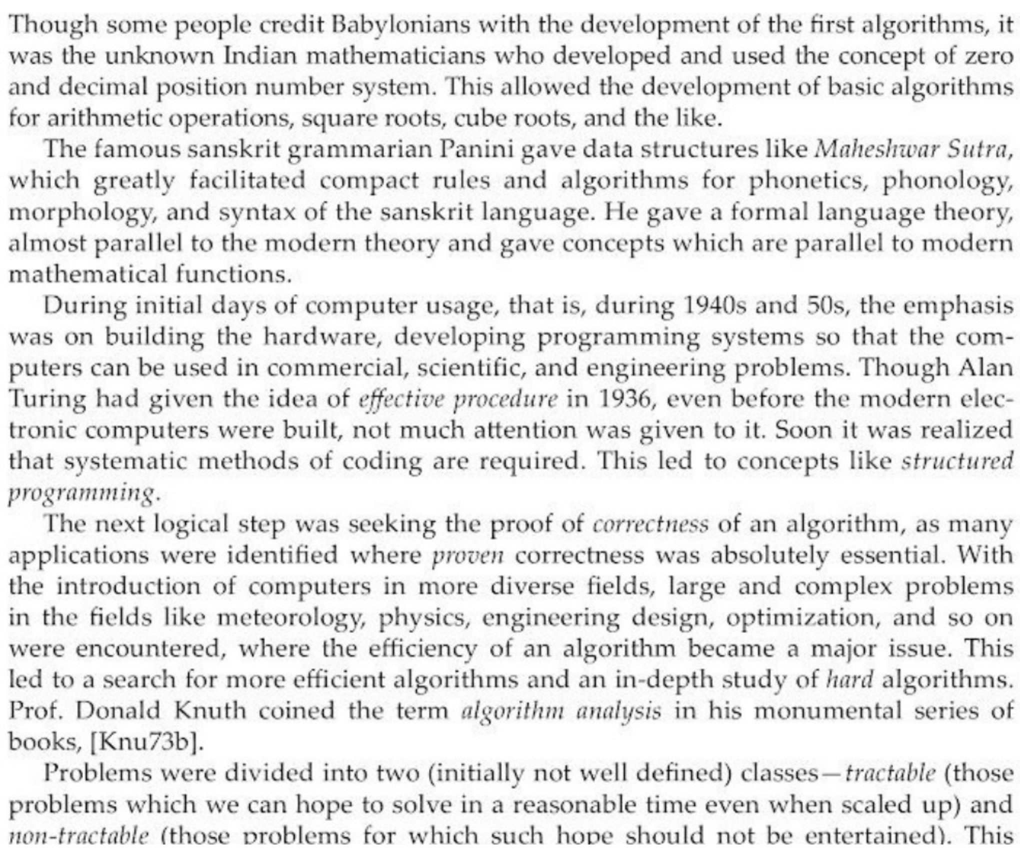
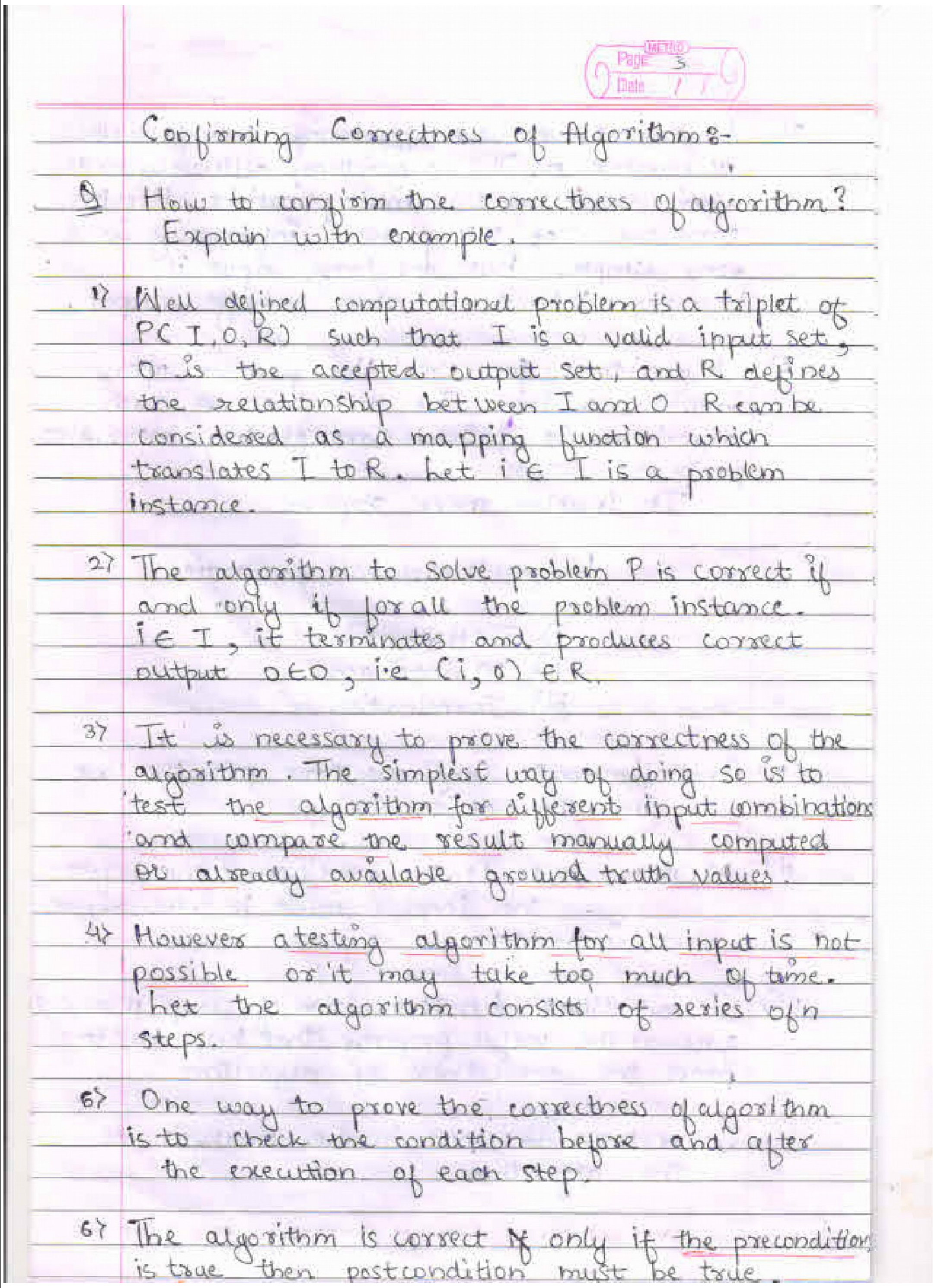
By:Prof. Akshay R. Jain
Asst. Prof.
PVG's COE, Nashik-04

T.E Computer (Second Semester) Examination 2018
Design & Analysis of Algorithms
(2015 Pattern)
In-Sem Exam March-2018

Time : 1 Hours **Maximum Marks : 30**

	Q1	
a	**Write a short note on the evolution of algorithm.**	**5**
Ans	 Though some people credit Babylonians with the development of the first algorithms, it was the unknown Indian mathematicians who developed and used the concept of zero and decimal position number system. This allowed the development of basic algorithms for arithmetic operations, square roots, cube roots, and the like. The famous sanskrit grammarian Panini gave data structures like *Maheshwar Sutra*, which greatly facilitated compact rules and algorithms for phonetics, phonology, morphology, and syntax of the sanskrit language. He gave a formal language theory, almost parallel to the modern theory and gave concepts which are parallel to modern mathematical functions. During initial days of computer usage, that is, during 1940s and 50s, the emphasis was on building the hardware, developing programming systems so that the computers can be used in commercial, scientific, and engineering problems. Though Alan Turing had given the idea of *effective procedure* in 1936, even before the modern electronic computers were built, not much attention was given to it. Soon it was realized that systematic methods of coding are required. This led to concepts like *structured programming*. The next logical step was seeking the proof of *correctness* of an algorithm, as many applications were identified where *proven* correctness was absolutely essential. With the introduction of computers in more diverse fields, large and complex problems in the fields like meteorology, physics, engineering design, optimization, and so on were encountered, where the efficiency of an algorithm became a major issue. This led to a search for more efficient algorithms and an in-depth study of *hard* algorithms. Prof. Donald Knuth coined the term *algorithm analysis* in his monumental series of books, [Knu73b]. Problems were divided into two (initially not well defined) classes—*tractable* (those problems which we can hope to solve in a reasonable time even when scaled up) and *non-tractable* (those problems for which such hope should not be entertained). This	
	How to confirm the correctness of Algorithm?Explain with example.	**5**
Ans		

By:Prof. Akshay R. Jain
Asst. Prof.
PVG's COE, Nashik-04



Confirming Correctness of Algorithm:-

Q How to confirm the correctness of algorithm? Explain with example.

1) Well defined computational problem is a triplet of P(I,O,R) such that I is a valid input set, O is the accepted output set, and R defines the relationship between I and O. R can be considered as a mapping function which translates I to R. Let $i \in I$ is a problem instance.

2) The algorithm to solve problem P is correct if and only if for all the problem instance. $i \in I$, it terminates and produces correct output $o \in O$, i.e $(i, o) \in R$.

3) It is necessary to prove the correctness of the algorithm. The simplest way of doing so is to test the algorithm for different input combinations and compare the result manually computed or already available ground truth values.

4) However a testing algorithm for all input is not possible or it may take too much of time. Let the algorithm consists of series of n steps.

5) One way to prove the correctness of algorithm is to check the condition before and after the execution of each step.

6) The algorithm is correct if only if the precondition is true then postcondition must be true.

MEMO Page Date: / /

7) Consider the problem of finding the factorial of number n. The algorithm definately halts after doing (n-1) multiplications. Proving correctness of this problem informally is very simple. But for large input it becomes tedious to store a huge output.

8) Algorithms for a real life problem often involves a loop. The correctness of such algorithm is proved through loop invariant property.
It involves three steps:-

Loop Invariant properties.

i) Initialization
ii) Maintenance
iii) Termination.

i) Initialization:- Conditions true prior the first interation of the loop.

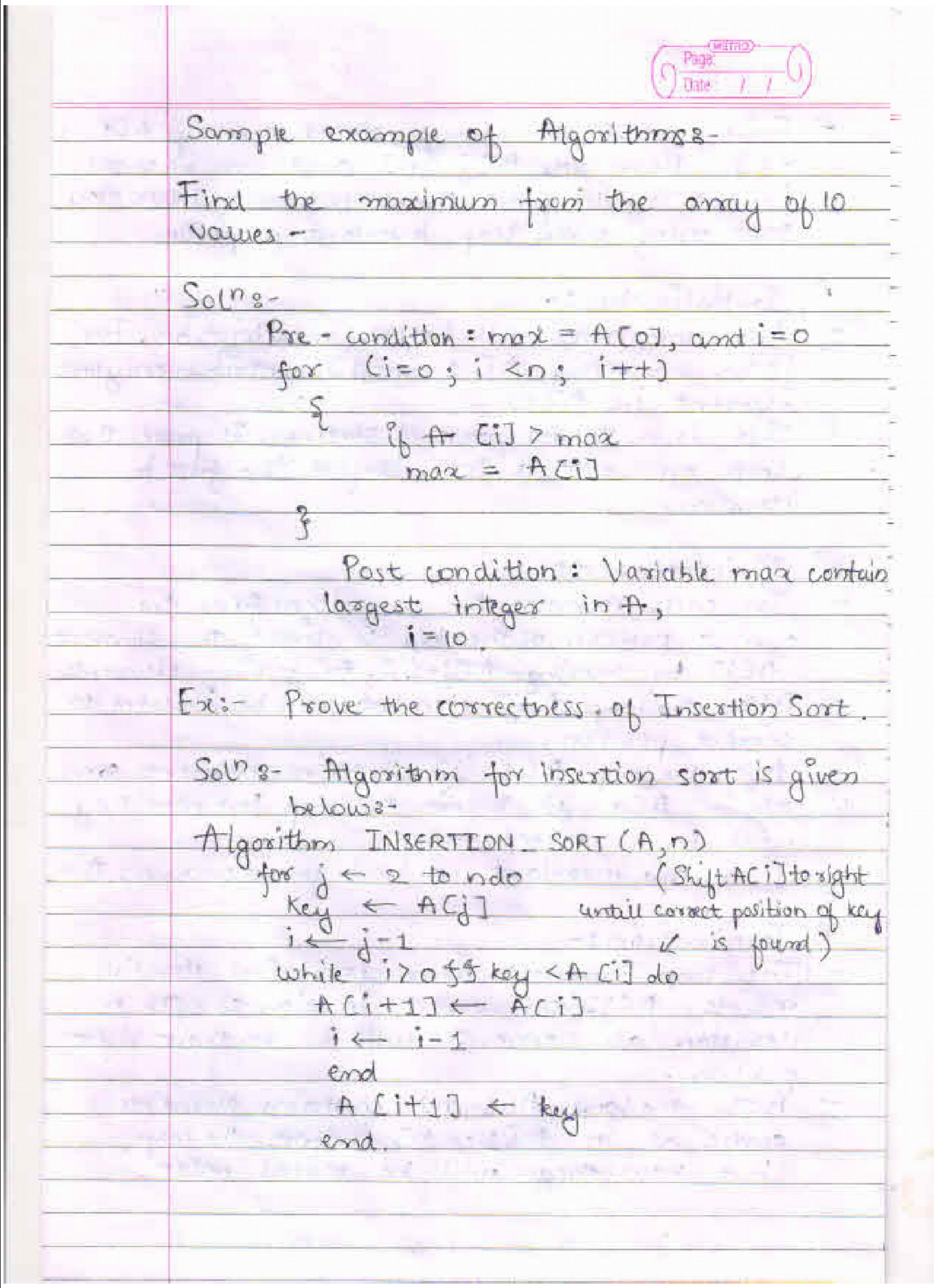
ii) Maintenance:- If the condition is true before the loop, it must be true before next interation

iii) Termination:- On termination of loop invariant gives us the useful property that helps us to prove the correctness of algorithm

Correctness does not involve analysis of the algorithm.

Invariant - Input and output does not change at the time is called Invariant

By:Prof. Akshay R. Jain
Asst. Prof.
PVG's COE, Nashik-04



Page
Date / /

Sample example of Algorithms:-

Find the maximum from the array of 10 values:-

Soln:-

```
Pre - condition : max = A[0], and i=0
for (i=0 ; i <n; i++)
  {
        if A [i] > max
        max = A[i]
  }
```

Post condition : Variable max contain largest integer in A,
i=10.

Ex:- Prove the correctness of Insertion Sort.

Soln:- Algorithm for insertion sort is given below:-

```
Algorithm INSERTION_ SORT (A,n)
  for j ← 2 to ndo                (Shift A[i] to right
  Key ← A[j]                       until correct position of key
  i ← j-1                              is found)
  while i>0 && key <A [i] do
     A[i+1] ← A[i]
     i ← i-1
     end
     A [i+1] ← key
   end.
```

By:Prof. Akshay R. Jain
Asst. Prof.
PVG's COE, Nashik-04

	OR	
	Q2	
a	**How can you measure and increase the efficiency of an algorithm.**	5
Ans	**Efficiency:** Algorithms that solve the same problem can differ enormously in their efficiency. Generally speaking, we would like to select the most efficient algorithm for solving a given problem. **Space efficiency:** Space efficiency is usually an all-or-nothing proposition; either we have enough space in our computer's memory to run a program implementing a specific algorithm, or we do not. If we have enough, we're OK; if not, we can't run the program at all. Consequently, analysis of the space requirements of a program tend to be pretty simple and straightforward. **Time efficiency:** When we talk about the efficiency of an algorithm, we usually mean the time requirements of the algorithm: how long would it take a program executing this algorithm to solve the problem? If we could afford to wait around forever (and could rely on the power company not to lose the power), it wouldn't make any difference how efficient our algorithm was in terms of time. But we can't wait forever; we need solutions in a reasonable amount of time.	
b	**Explain Characteristics of good algorithm.List out the problems solved by the algorithm.**	5
Ans	**What is an algorithm?** An algorithm is a finite set of precise instructions for performing a computation or for solving a problem. **Properties of algorithms:** ➢ **Input** from a specified set, ➢ **Output** from a specified set (solution), ➢ **Definiteness** of every step in the computation, ➢ **Correctness** of output for every possible input, ➢ **Finiteness** of the number of calculation steps, ➢ **Effectiveness** of each calculation step. **Problem solved by Algorithm:** ➢ Greedy Method ➢ Sorting ➢ Divide and Conquer ➢ Dynammic Programming	
	Q3	
a.	**Explain Functional Model.Define fetures of Functional Model.**	5
Ans	**Functional Model**	

The functional model is very close to mathematics; hence functional algorithms are easy to analyze in terms of their correctness and efficiency. A functional algorithm can serve as a specification for the development of algorithms in other models of computation.

In the functional model of computation every problem is viewed as an evaluation of a function. The solution to a given problem is specified by a complete and unambiguous functional description.

A good model of computation must have the following facilities.

1. **Primitive expressions** which represent the simplest objects with which the model is concerned.
2. **Methods of combination** which specify how the primitive expressions can be combined with one another to obtain compound expressions.
3. **Methods of abstraction** which specify how the compound objects can be named and manipulated as units.

Features of Functional Model

We now introduce the following features of the functional model:

1. The primitive expressions
2. Definition of one function in terms of another (substitution)
3. Definition of functions using conditionals
4. Inductive definition of functions

The Primitive Expressions:-The basic primitives of the functional model are *constants*, *variables*, and *functions*.

Elements of the sets $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$ are the constants. Also, the elements of the set $\mathbb{B}$ = {*true*, *false*} are constants. There can be other kind of constants, which will be introduced as and when needed.

Variables are identifiers which refer to data objects (constants). We use identifiers like *n*, *a*, *b*, *x* and so on, to refer to various data elements used in our functional algorithms. The variables are bound to values (constants) in the same way as in normal algebra. Thus, the declarations $x = 5$ and $y = true$ bind the variables *x* and *y* to the values 5 and *true*, respectively.

The primitive functions of the type

$f : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$

and

$f : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$

which we shall assume to be available in our functional model are addition (+), subtraction (–), and multiplication (*). We shall also assume the availability of the *div* and *mod* functions of the type

$f : \mathbb{N} \times \mathbb{P} \to \mathbb{N}$.

The functional mapping symbolism like $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is also called the **signature** of that function. Note that if $a \in \mathbb{N}$ and $b \in \mathbb{P}$ and $a = q * b + r$ for some integers $q \in \mathbb{N}$ and $0 \le r < b$ then $div(a, b) = q$ and $mod(a, b) = r$. The division function

$/ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$

will be assumed to be valid only for real numbers. In addition to these, we will assume the relational functions =, ≤, <, ≥, >, and ≠ which are of the type

$f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}$

or

$f : \mathbb{R} \times \mathbb{R} \to \mathbb{B}$

depending on the context.

Also, the following functions are assumed to be available.

1. $\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ (**and**),
2. $\vee : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ (**or**), and
3. $\neg : \mathbb{B} \to \mathbb{B}$ (**not**).

Substitution of Functions:- We shall now give a few examples of the definition of one function in terms of another and the evaluation of such functions through substitution.

EX 1]Finding the square of a natural number. We can directly specify the function square, which is of the type

$square : \mathbb{N} \to \mathbb{N}$

in terms of the standard multiplication function:

$* : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ as

$square(n) = n * n$

Here, we assume that we can substitute one function for another, provided both of them return an item of the same type. To evaluate, say, *square*(5), we have to thus evaluate 5 * 5.

Hence, we can build more complex functions from simpler ones. As an example, let

By:Prof. Akshay R. Jain
Asst. Prof.
PVG's COE, Nashik-04

us define a function to compute $x2 + y2$.

Definition of Functions using Conditionals:-

Finding the larger of two numbers. Let us define a function

$max : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

The domain set for this function is the Cartesian product of natural numbers representing a pair, and the range is the set $\mathbb{N}$. Thus the function accepts a pair of natural numbers as its input, and gives a single natural number as its output. We define this function as

$$max(a, b) = \begin{cases} a & \text{if } a \geq b \\ b & \text{otherwise} \end{cases}$$

While defining the function *max*, we have assumed that two natural numbers can be compared using the ≥ function to determine which is larger. The basic primitive used in this case is ***if-then-else***. Thus if $a \geq b$, the function returns a as the output, else it returns b. Note that for every pair of natural numbers as its input, *max* returns a unique number as the output and hence it adheres to the definition of a function.

Inductive Definition of Functions:-

All the examples we have seen so far, are of functions which can be evaluated by substitutions or evaluation of conditions. Let us now consider functions as inductively defined computational processes.

We consider an example of an inductively defined functional algorithm for computing the GCD (Greatest Common Divisor) of two positive integers.

EX1]Computing the GCD of two numbers. We can define the function

$gcd : \mathbb{P} \times \mathbb{P} \to \mathbb{P}$ as

$$gcd(a, b) = \begin{cases} a & \text{if } a = b \\ gcd(a-b, b) & \text{if } a > b \\ gcd(a, b-a) & \text{if } b > a \end{cases}$$

It is a function because, for every pair of positive integers as input, it gives a positive integer as the output. It is also a finite computational process, because given any two positive integers as input, the description unambiguously tells us how to compute the solution and the process terminates after a finite number of steps.

For example, for the specific case of computing *gcd*(18, 12), we have

gcd(18, 12) = *gcd*(12, 6) = *gcd*(6, 6) = 6.

b. **Explain Recursive and Iterative process with example.** **5**

Ans

Recursive Processes:-

Recursive computational processes are characterized by a chain of *deferred* operations. As an example, we will consider an algorithm for computing the factorial of an integer *n*(*n*!).

EX] Factorial Computation.

Given *n* ≥ 0, compute the factorial of *n*(*n*!).

On the basis of the inductive definition of *n*!, we can define a **functional algorithm**, that is, an algorithm which is expected to be invoked by another, higher level algorithm—for computing *factorial*(*n*), which is of the type,

$factorial : \mathbb{N} \rightarrow \mathbb{N}$ as

$$factorial(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times factorial(n-1) & \text{otherwise} \end{cases}$$

Here *factorial*(*n*) is the function name and the description after the = sign is the body of the function. It is instructive to examine the computational process underlying this definition. The computational process, in special case of *n* = 5, looks as follows:

factorial(5) = (5 × *factorial*(4)) = (5 × (4 × *factorial*(3))) = (5 × (4 × (3 × *factorial*(2))))

= (5 × (4 × (3 × (2 × *factorial*(1))))) = (5 × (4 × (3 × (2 × (1 × *factorial*(0))))))

= (5 × (4 × (3 × (2 × (1 × 1))))) = (5 × (4 × (3 × (2 × 1)))) = (5 × (4 × (3 × 2)))

= (5 × (4 × 6)) = (5 × 24) = 120

A computation such as this is characterized by a *growing* (*recursive descent*) and a *shrinking* (*recursive ascent*) process. In the growing phase each call to the function is replaced by its body which in turn contains a call to the same function with different values of the formal arguments. According to the inductive definition, computations can be done by the actual multiplications which would be postponed till the base case of *factorial*(0). This results in a growing process. Once the base value is available, the actual multiplications can be carried out resulting in a

	shrinking process. Computational processes which are characterized by such deferred computations are called **recursive**. This is not to be confused with the notion of a recursive procedure which refers to the syntactic fact that the procedure is described in terms of itself. Such a recursive procedure may actually execute iteratively, for example, in tail-recursion, while in case of recursive computational process, deferred computation is emphasised. **Iterative Processes:-** The underlying computational process for the special case of $\mathit{factorial}(5)$ looks as follows: $\mathit{factorial}(5) = \mathit{fact\_iter}(5, 1, 0) = \mathit{fact\_iter}(5, 1, 1) = \mathit{fact\_iter}(5, 2, 2) = \mathit{fact\_iter}(5, 6, 3)$ $= \mathit{fact\_iter}(5, 24, 4)$ $= \mathit{fact\_iter}(5, 120, 5)$ $= 120$ Contrast this with the recursive process for computing $\mathit{factorial}(n)$. The recursive process is characterized by a growing and shrinking process due to deferred computations. In the growing process, the multiplicative constants 5, 4, 3, 2 and 1 are stacked up before the results of $\mathit{factorial}(0)$, $\mathit{factorial}(1)$, $\mathit{factorial}(2)$, $\mathit{factorial}(3)$ and $\mathit{factorial}(4)$ become available. In the shrinking process, the actual multiplications $n * \mathit{factorial}(n - 1)$ are carried out to obtain $\mathit{factorial}(n)$ successively.	
	OR	
	Q4	
a	**Explain Tail recursion with suitable Example**	**5**
Ans	how such inefficiencies can be removed by describing alternative algorithms for these problems using **tail-recursion** which lead to **iterative** computational processes. The crucial idea in iterative algorithms is to represent the **state** of computation at each stage in terms of auxiliary variables so as to obtain the final result from the final state of these variables. We may think of the state of a computation as a collection of instantaneous values of certain entities. As an example of an iterative algorithm described through state changes, let us consider the problem of computation of $\mathit{factorial}(n)$ again and design an iterative algorithm for the problem. **EX 1] Iterative computation of factorial.** We maintain the state of the factorial computation in terms of three auxiliary	

variables *f*, *c*, and *m*. We start with the initial values *f* = *f*0 when *c* = *c*0, and successively increment the value of the counter *c* by 1, while maintaining at every stage, the following condition as an *invariant* about the state of the computation:

$$(c_0 \le c \le m) \wedge \left(f = f_0 * \prod_{i=c_0+1}^{c} i \right) \wedge \left(f_0 * \prod_{i=c_0+1}^{m} i = f * \prod_{i=c+1}^{m} i \right)$$

Then, when *c* = *m* we can obtain $f = f_0 * \prod_{i=c_0+1}^{m}$ as the final result. This is the same as *factorial*(*n*) if the initial values are *m* = *n*, *c*0 = 0, and *f*0 = 1 respectively. The resulting algorithm is described here.

factorial(*n*) = *fact_iter*(*n*, 1, 0),

where the auxiliary function *fact_iter* : $\mathbb{N} \times \mathbb{P} \times \mathbb{N} \rightarrow \mathbb{P}$ is defined as

$$fact\_iter(m, f, c) = \begin{cases} f & \text{if } c = m \\ fact\_iter(m, f * (c+1), c+1) & \text{otherwise} \end{cases}$$

Note that the invariant condition (which is a boolean function of the state of the system described in terms of the variables *f* and *c*) holds true every time the function *fact_iter* is invoked.

The function description of *fact_iter* is called tail-recursive because the “otherwise” clause in its description is a simple recursive call to the function itself. Contrast this with the “otherwise” clause of the recursive factorial which is given as *n* * *factorial*(*n* – 1) and involves the recursive call with the multiplication operation. A tail-recursive definition such as this leads to a computational process different from that of the recursive version for the same problem. The underlying computational process for the special case of *factorial*(5) looks as follows:

factorial(5) = *fact_iter*(5, 1, 0) = *fact_iter*(5, 1, 1) = *fact_iter*(5, 2, 2) = *fact_iter*(5, 6, 3)

= *fact_iter*(5, 24, 4)

= *fact_iter*(5, 120, 5)

= 120

Contrast this with the recursive process for computing *factorial*(*n*). The recursive process is characterized by a growing and shrinking process due to deferred computations. In the growing process, the multiplicative constants 5, 4, 3, 2 and 1 are stacked up before the results of *factorial*(0), *factorial*(1), *factorial*(2), *factorial*(3) and *factorial*(4) become available. In the shrinking process, the actual multiplications *n* * *factorial*(*n* – 1) are carried out to obtain *factorial*(*n*)

	successively.	
b	**Obtain a set of optimal Huffman codes for the messages (M1...M6) with relative frequencies(q1...q6)=(5,9,12,13,16,45) draw the decode tree for this set of codes?**	**5**

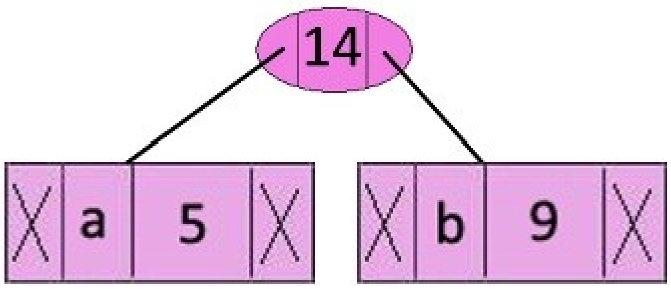
Ans

Let us understand the algorithm with an example:

character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

Step 1. Build a min heap that contains 6 nodes where each node represents root of a tree with single node.

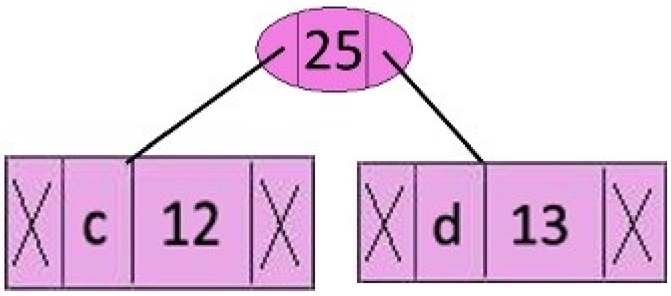
Step 2 Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14.



Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

character	Frequency
c	12
d	13
Internal Node	14
e	16
f	45

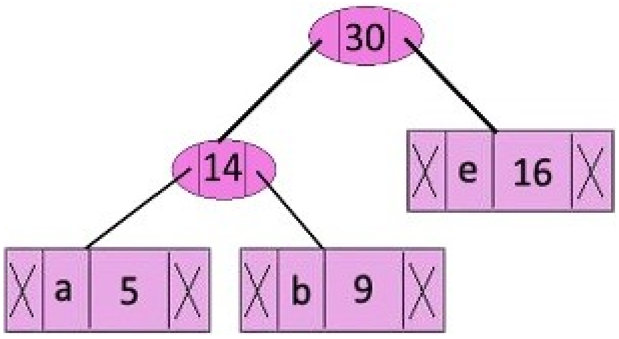
Step 3: Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25



Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes.

character	Frequency
Internal Node	14
e	16
Internal Node	25
f	45

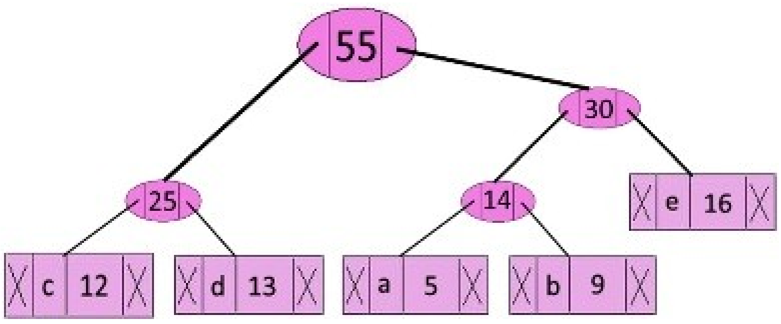
Step 4: Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30



Now min heap contains 3 nodes.

character	Frequency
Internal Node	25
Internal Node	30
f	45

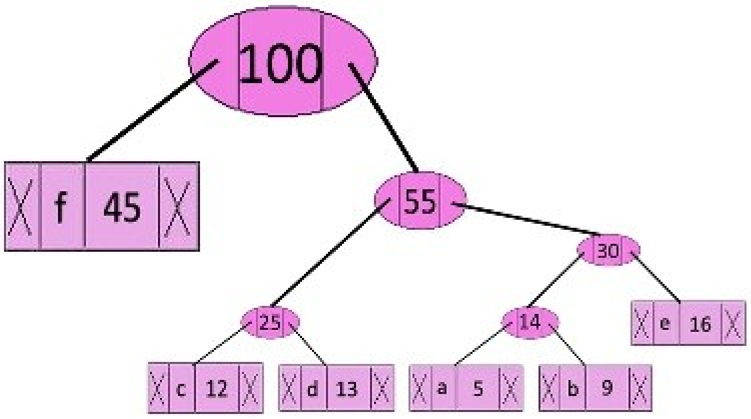
Step 5: Extract two minimum frequency nodes. Add a new internal node with frequency
25 + 30 =
55



Now min heap contains 2 nodes.

character	Frequency
f	45
Internal Node	55

Step 6: Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100



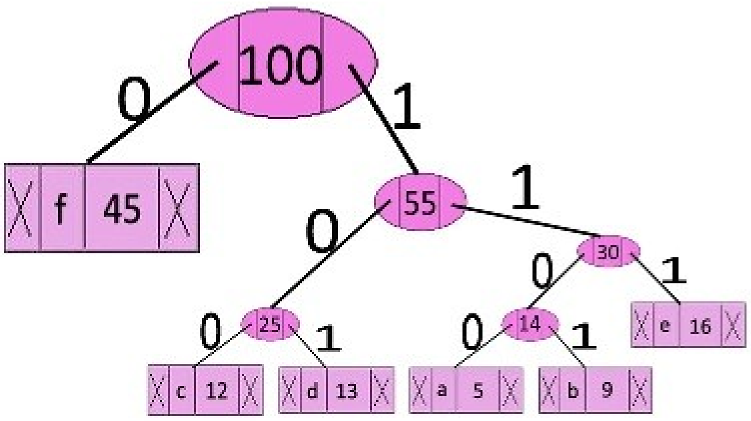
Now min heap contains only one node.

character	Frequency
Internal Node	100

Since the heap contains only one node, the algorithm stops here.

Steps to print codes from Huffman Tree:

Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



The codes are as follows:

character	code-word
f	0
c	100
d	101
a	1100
b	1101
e	111

Q5

a **What is maent by divide and conquer strategy? Name few problems that can be solved using divide and conquer?** **5**

Ans

Divide and conquer is a design strategy which is well known to breaking down efficiency barriers. When the method applies, it often leads to a large improvement in time complexity. For example, from O (n 2) to O (n log n) to sort the elements.
Divide and conquer strategy is as follows: divide the problem instance into two or more smaller instances of the same problem, solve the smaller instances recursively, and assemble the solutions to form a solution of the original instance. The recursion stops when an instance is reached which is too small to divide. When dividing the instance, one can either use whatever division comes most easily to hand or invest time in making the division carefully so that the assembly is simplified.
Divide and conquer algorithm consists of two parts:
Divide:
Conquer :
Divide the problem into a number of sub problems. The sub problems are solved recursively.
The solution to the original problem is then formed from the solutions to the sub problems (patching together the answers).
Traditionally, routines in which the text contains at least two recursive calls are called divide and conquer algorithms, while routines whose text contains only one recursive call are not. Divide–and–conquer is a very powerful use of recursion
Ex)
Quick Sort
Merg Sort
Binary Search Tree

b **Explain Basic Steps of Genetic Algorithm.** **5**

Genetic Algorithms*

The *Genetic Algorithm* is a model of global optimization through machine learning which derives its behaviour from a metaphor of the processes of evolution in the nature.

This is done by the creation, within a computer system, of a *Population* of *Individuals* represented by *Chromosomes*, which are tokens for certain characteristics.

The chromosomes are essentially a set of character strings that are analogous to the base-4 chromosomes in our own DNA. The individuals in the population then go through a process of *evolution*.

In nature, it is seen that encoding for the genetic information (*Genome*) is done in a way that admits asexual reproduction (such as by budding), which generally results in offsprings that are genetically identical to the parent.

Sexual reproduction allows the creation of genetically different offsprings that are of the same general class (species).

Roughly, what happens at the molecular level is that a pair of chromosomes exchange chunks of genetic information.

This is the *recombination* operation, which is generally referred to as *Crossover* because of the way the genetic material crosses over from one chromosome to another.

The crossover operation happens in an environment where the selection of who gets to mate is a function of the fitness of the individual, that is, how good the individual is at competing in its environment.

Some genetic algorithms use a simple function of the fitness measure to select individuals based on probability to undergo genetic operations such as crossover or asexual reproduction (the propagation of genetic material is unaltered).

This is called **fitness-proportionate selection**. Other implementations use a model in which certain randomly selected individuals in a subgroup compete and the fittest is selected.

This is called *tournament* selection. The two processes that most contribute to evolution are crossover and fitness based selection/reproduction.

As it turns out, there are mathematical proofs that indicate that the process of *fitness-proportionate* reproduction is, in fact, near optimal in some sense.

Mutation also plays a role in this process, although its importance continues to be a matter of debate (some refer to it as a background operator, while others view it as playing dominant role in the evolutionary process).

It cannot be stressed that the genetic algorithm (as a simulation of a genetic process) is *not a random search* for a solution to a problem (highly fit individual). The genetic algorithm uses stochastic processes, but the result is distinctly non-

	random (better than random). Genetic algorithms are used for a number of different application areas. An example of this would be multidimensional optimization problem in which the character string of the chromosome can be used to encode the values for the different parameters being optimized.	

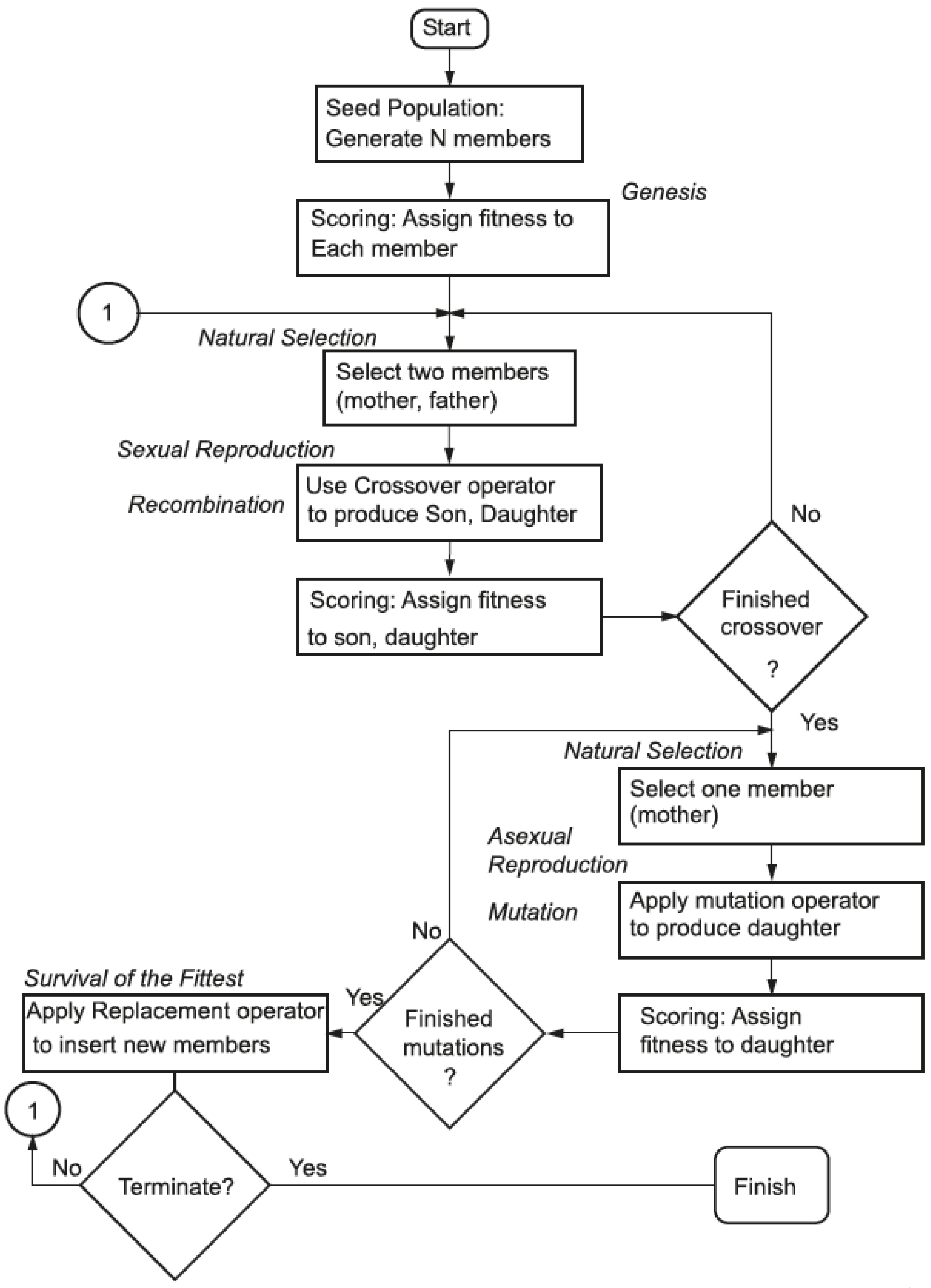

Start
Seed Population: Generate N members
Genesis
Scoring: Assign fitness to Each member
1
Natural Selection
Select two members (mother, father)
Sexual Reproduction
Recombination
Use Crossover operator to produce Son, Daughter
No
Scoring: Assign fitness to son, daughter
Finished crossover ?
Yes
Natural Selection
Select one member (mother)
Asexual Reproduction
Mutation
Apply mutation operator to produce daughter
No
Survival of the Fittest
Yes
Apply Replacement operator to insert new members
Finished mutations ?
Scoring: Assign fitness to daughter
1
No
Terminate?
Yes
Finish

Fig.1.Flow chart of genetic algorithm

OR

Q6

a	**Consider a knapsack instance n=3.(w1,w2,w3)=(2,3,4),(p1,p2,p3)=(1,2,5) and M=6. Find optimal solution using dynammic programming.**	5

Ans

Initially, f o (x) = 0, for all x and f i (x) = - infinity if x < 0.
F n (M) = max {f n-1 (M), f n-1 (M - w n) + p n }
F 3 (6) = max (f 2 (6), f 2 (6 – 4) + 5} = max {f 2 (6), f 2 (2) + 5}
F 2 (6) = max (f 1 (6), f 1 (6 – 3) + 2} = max {f 1 (6), f 1 (3) + 2}
F 1 (6) = max (f 0 (6), f 0 (6 – 2) + 1} = max {0, 0 + 1} = 1
F 1 (3) = max (f 0 (3), f 0 (3 – 2) + 1} = max {0, 0 + 1} = 1
Therefore, F 2 (6) = max (1, 1 + 2} = 3
F 2 (2) = max (f 1 (2), f 1 (2 – 3) + 2} = max {f 1 (2), - ∞ + 2}
F 1 (2) = max (f 0 (2), f 0 (2 – 2) + 1} = max {0, 0 + 1} = 1
F 2 (2) = max {1, - ∞ + 2} = 1
Finally, f 3 (6) = max {3, 1 + 5} = 6

b	**Write a short note on tabu search.**	5

Ans

Tabu Search (TS)

- The basic concept of Tabu Search is "a meta-heuristic superimposed on another heuristic". The overall approach is to avoid entrainment in cycles by forbidding or penalising moves which take the solution, in the next iteration, to points in the solution space previously visited (hence the term "tabu" or "taboo").
- The TS is fairly new. The method is actively undergoing research and is continuing to evolve and improve.
- The Tabu method was partly motivated by the observation that human behaviour appears to operate with a random element that leads to inconsistent behaviour given similar circumstances.
- As Glover points out, the resulting tendency to deviate from a charted course, might be regretted as a source of error but can also prove to be a source of gain. The Tabu method operates in this way with the exception that new courses are not chosen randomly.
- Instead the Tabu search proceeds according to the supposition that there is no point in accepting a new (poor) solution unless it is to avoid a path already investigated.
- This insures that new regions of a problem solution space will be investigated in, with the goal of avoiding local minima and ultimately finding the desired solution.
- The Tabu search begins by proceeding to a local minima. To avoid retracing

the steps used, the method records recent moves in one or more *Tabu lists*.

- The original intent of the list was not to prevent a previous move from being repeated, but rather to insure it was not reversed. The Tabu lists are historical in nature and form the Tabu search memory. The role of the memory can change as the algorithm proceeds.
- At initialisation the goal is to make a coarse examination of the solution space, known as **diversification**, but as candidate locations are identified the search is more focused to produce local optimal solutions in a process of **intensification**.
- In many cases, the difference between the various implementations of the Tabu method are related to the size, variability, and adaptability of the Tabu memory to a particular problem domain.