ಕರ್ನಾಟಕ ರಾಜ್ಯ ಮುಕ್ತ ವಿಶ್ವವಿದ್ಯಾನಿಲಯ
ಮಾನಸಗಂಗೋತ್ರಿ, ಮೈಸೂರು – 570 006

KARNATAKA STATE OPEN UNIVERSITY
Manasagangotri Mysore - 570 006

M.Sc. Computer Science
Second Semester

ANALYSIS & DESIGN OF ALGORITHMS

Course: 7
Module: 1 - 6
MSCS-507 ANALYSIS & DESIGN OF ALGORITHMS

ಉನ್ನತ ಶಿಕ್ಷಣಕ್ಕಾಗಿ ಇರುವ ಅವಕಾಶಗಳನ್ನು ಹೆಚ್ಚಿಸುವುದಕ್ಕೆ ಮತ್ತು ಶಿಕ್ಷಣವನ್ನು ಪ್ರಜಾತಂತ್ರೀಕರಿಸುವುದಕ್ಕೆ ಮುಕ್ತ ವಿಶ್ವವಿದ್ಯಾನಿಲಯ ವ್ಯವಸ್ಥೆಯನ್ನು ಆರಂಭಿಸಲಾಗಿದೆ.

ರಾಷ್ಟ್ರೀಯ ಶಿಕ್ಷಣ ನೀತಿ 1986

The Open University System has been initiated in order to augment opportunities for higher education and as instrument of democrating education.

National Educational Policy 1986

ವಿಶ್ವ ಮಾನವ ಸಂದೇಶ

ಪ್ರತಿಯೊಂದು ಮಗುವು ಹುಟ್ಟುತ್ತಲೇ – ವಿಶ್ವಮಾನವ, ಬೆಳೆಯುತ್ತಾ ನಾವು ಅದನ್ನು 'ಅಲ್ಪ ಮಾನವ'ನನ್ನಾಗಿ ಮಾಡುತ್ತೇವೆ. ಮತ್ತೆ ಅದನ್ನು 'ವಿಶ್ವಮಾನವ'ನನ್ನಾಗಿ ಮಾಡುವುದೇ ವಿದ್ಯೆಯ ಕರ್ತವ್ಯವಾಗಬೇಕು.

ಮನುಜ ಮತ, ವಿಶ್ವ ಪಥ, ಸರ್ವೋದಯ, ಸಮನ್ವಯ, ಪೂರ್ಣದೃಷ್ಟಿ ಈ ಪಂಚಮಂತ್ರ ಇನ್ನು ಮುಂದಿನ ದೃಷ್ಟಿಯಾಗಬೇಕಾಗಿದೆ. ಅಂದರೆ, ನಮಗೆ ಇನ್ನು ಬೇಕಾದುದು ಆ ಮತ ಈ ಮತ ಅಲ್ಲ; ಮನುಜ ಮತ. ಆ ಪಥ ಈ ಪಥ ಅಲ್ಲ ; ವಿಶ್ವ ಪಥ. ಆ ಒಬ್ಬರ ಉದಯ ಮಾತ್ರವಲ್ಲ; ಸರ್ವರ ಸರ್ವಸ್ತರದ ಉದಯ. ಪರಸ್ಪರ ವಿಮುಖವಾಗಿ ಸಿಡಿದು ಹೋಗುವುದಲ್ಲ; ಸಮನ್ವಯಗೊಳ್ಳುವುದು. ಸಂಕುಚಿತ ಮತದ ಆಂಶಿಕ ದೃಷ್ಟಿ ಅಲ್ಲ; ಭೌತಿಕ ಪಾರಮಾರ್ಥಿಕ ಎಂಬ ಭಿನ್ನದೃಷ್ಟಿ ಅಲ್ಲ; ಎಲ್ಲವನ್ನು ಭಗವದ್ ದೃಷ್ಟಿಯಿಂದ ಕಾಣುವ ಪೂರ್ಣ ದೃಷ್ಟಿ.

ಕುವೆಂಪು

Gospel of Universal Man

Every Child, at birth, is the universal man. But, as it grows, we trun it into "a petty man". It should be the function of education to turn it again into the enlightened "universal man".

The Religion of Humanity, the Universal Path, the Welfare of All, Reconciliation, the Integral Vision - these ***five mantras*** should become view of the Future. In other words, what we want henceforth is not this religion or that religion, but the Religion of Humanity; not this path or that path, but the Universal Path; not the well-being of this individual or that individual, but the Welfare of All; not turning away and breaking off from one another, but reconciling and uniting in concord and harmony; and above all, not the partial view of a narrow creed, not the dual outlook of the material and the spiritual, but the Integral Vision of seeing all things with the eye of the Divine.

Kuvempu

Module 1: Introduction

Module 2: The Greedy Method

Module 3: Divide and Conquer

Module 4: Dynamic Programming

Module 5: Backtracking

Module 6: Branch and Bound

Publisher

Registrar
Karnataka State Open University
Manasagangotri, Mysore - 6.

Developed by Academic Section, KSOU, Mysore

Preface

In the current era of intelligent systems, the information technology plays a very important role in building up a complete automated society. The problems at hand have got to be tackled, on time/ on real time such that the time required to solve them with the aid of computers becomes negligible. What matters in achieving these is only the complexity of the method (algorithm) to be adapted to solve a problem. The method may achieve the intended goal either by reducing the search domain (in memory) or by the use of indexing on large memory. That is all what the tradeoff that we have to understand while designing a suitable algorithm for solving a problem. In case of existence of several ways for solving a problem, we have to look for the most efficient one among them. The efficiency of a method depends on our requirement specification. A method/algorithm which is efficient for somebody may not be efficient for others. Before, one goes for adaptation of any method, he/she has to work out the tradeoffs of all algorithms with respect to their time requirement, space requirement, correctness, accuracy, robustness, simplicity in terms of transparency etc. which are generally called the quality factors.

In this course material, we address the issues related to the quality factors of an algorithm. This material provides you an insight into the field of designing and analyzing algorithms.

The design strategies such as (i) divide and conquer, where a problem is split into many sub problems so that it can be handled so efficiently, (ii) Greedy strategy, where we always try to optimize our requirement being greedy in choosing the next best solution, (iii) Backtracking strategy, where we can look back to achieve betterment in the solution to be arrived, (iv) Branch and Bound strategy where we work with a set of constraints/functions to be met and finally (iv) Dynamic Programming strategy where we always try to minimize the search domain to achieve efficiency in algorithm execution are completely dealt in this material. Indeed, these though inside the material, are called techniques, it would be better to always call them strategies always as techniques in fact simulate what a more optimist human being does while achieving efficiency in solving a problem.

We thank everyone who helped directly or indirectly to prepare this material. Without their support this material could not have been prepared.

Dr. D.S.Guru, Dr. H.S.Nagendraswamy, Dr. Lalitha Rangarajan

Further information on the Karnataka State Open University Programmes may obtained from the University's office at Manasagangotri, Mysore-6

Printed and Published on behalf of Karnataka State Open University.

Mysore-6 by **Registrar (Administration)**

UNIT – 1

INTRODUCTION

STRUCTURE

1.0 OBJECTIVES

After studying this unit you should be able to

- Define an algorithm and its characteristics.
- Explain the performance of algorithms.
- Transform an algorithm into a program.
- Discuss the space complexity and time complexity of an algorithm.
- Give asymptotic notations for algorithms.

1.1 INTRODUCTION

Computer Science is the field where we study about how to solve a problem effectively and efficiently with the aid of computers. Solving a problem that too by the use of computers requires a thorough knowledge and understanding of the problem. The problem could be of any complex ranging from a simple problem of adding two numbers to a problem of making the computer capable of taking decisions on time in real environment, automatically by understanding the situation or environment, as if it is taken by a human being. In order to automate the task of solving a problem, one has to think of many ways of arriving at the solution. A way of arriving at a solution from the problem domain is called algorithm. Thus, one can have many algorithms for the same problem.

In case of existence of many algorithms we have to select the one which best suits our requirements through analysis of algorithms. Indeed, the design and analysis of algorithms are the two major interesting sub fields of computer science. Most of the scientists do work on these subfields just for fun. We mean to say that these two sub areas of computer science are such interesting areas. Once the most efficient algorithm is selected, it gets coded in a programming language. This essentially requires knowledge of a programming language. And finally, we go for executing the coded algorithm on a machine (Computer) of particular architecture. Thus, the field computer science broadly encompasses,

- Study on design of algorithms.
- Study on analysis of algorithms.
- Study of programming languages for coding algorithms.
- Study of machine architecture for executing algorithms.

It shall be noticed that the fundamental notion in all the above is the term algorithm. Indeed, that signifies its prominence of algorithms, in the field of computer science and thus the algorithm deserves its complete definition. Algorithm means 'process or rules for computer calculation' according to the dictionary. However, it is something beyond that definition.

1.2 DEFINE AN ALGORITHM

An algorithm, named for the ninth-century Persian mathematician al-KhowArizmi, is simply a set of rules for carrying out some calculation, either by hand or, more usually, on a machine. However, other systematic methods for calculating a result could be included. The methods we learn at school for adding, multiplying and dividing numbers are algorithms, for instance. The most famous algorithm in history dates from well before the time of the ancient Greeks: this is Euclid's algorithm for calculating the greatest common divisor of two integers. The execution of an algorithm must not normally involve any subjective decisions, nor must it call for the use of intuition or creativity. Hence a cooking recipe can be considered to be an algorithm if it describes precisely how to make a certain dish, giving exact quantities to use and detailed instructions for how long to cook it. On the other hand, if it includes such vague notions as "add salt to taste" or "cook until tender" then we would no longer call it an algorithm.

When we use an algorithm to calculate the answer to a particular problem, we usually assume that the rules will, if applied correctly, indeed give us the correct answer. A set of rules that calculate that 23 times 51 is 1170 is not generally useful in practice. However in some circumstances such *approximate algorithms* can be useful. If we want to calculate the square root of 2, for instance, no algorithm can give us an exact answer in decimal notation, since the representation of square root of 2 is infinitely long and nonrepeating. In this case, we shall be content if an algorithm can give us an answer that is as precise as we choose: 4 figures accuracy, or 10 figures, or whatever we want.

Formally, an algorithm is defined to be a sequence of steps, which if followed, accomplishes a particular task. In addition, every algorithm must satisfy the following criteria.

- Consumes zero or more inputs
- Produces at least one output

- Definiteness
- Finiteness
- Effectiveness

The definiteness property insists that each step in the algorithm is unambiguous. A step is said to be unambiguous if it is clear, in the sense that, the action specified by the step can be performed without any dilemma/confusion. The finiteness property states that the algorithm must terminate its execution after finite number of steps (or after a finite time period of execution). That is to say, it should not get into an endless process. The effectiveness property is indeed the most important property of any algorithm. There could be many algorithms for a given problem. But, one algorithm may be effective with respect to a set of constraints and the others may not be effective with respect to the same constraints. However, they may be more effective with respect to some other constraints. In fact, because of the effectiveness property associated with algorithms, finding out most optimal/effective algorithm even for already solved problem is still open for the research community for further research.

To study the effectiveness of an algorithm, we have to find out the minimum and maximum number of operations the algorithm takes to solve the desired problem. The time requirement and space requirement profiling should be done. The profiling may be relatively based on the number of inputs or the number of outputs or on the nature of input. The process of knowing about the minimum cost and the maximum cost (may be in terms of CPU time and memory locations) is called analysis of algorithm. In the subsequent sections, we present methods of analyzing an algorithm.

1.3 ANALYSIS OF ALGORITHMS

In practical situations, it may not be sufficient if an algorithm works properly and yields desired results. A single problem can be solved in many different ways, and hence it is possible to design several algorithms to perform the same job. However, when algorithms are executed (in the form of a program) it uses the computer's resources (the CPU time, the memory etc.) to perform operations and its memory to hold the program and data. An algorithm, which consumes lesser resource, is indeed a better one. Hence, the process of "analyzing the algorithm" is an indispensable component in the study of algorithms. Analysis of algorithms or performance analysis refers to the task of determining how much computing time and storage an algorithm requires to run for completion.

A straight forward method of analyzing an algorithm is by coding the algorithm and then executing it for measuring the space and time requirements on a specific computer for various data sets. This straight forward method, however, is costly, time consuming and inconvenient. Hence, alternate methods need to be evolved i.e., one should be able to arrive at the requirements going through the lines of the algorithm. To do this, we keep in mind that each line of an algorithm gets converted to one/more instructions (operations) to the computer. Hence, by counting such instructions (or operations) one can approximate the time required. Similarly the various data structures provide information about the amount of storage space necessary.

But, in real pragmatic situations, an algorithm is normally quite lengthier and involves several loops, so that 'actual count' may become unbelievably different and unanticipated. It should be noticed that the instructions themselves are of different types, (involving arithmetic operations, logical operations, simple data movement etc). Certain operations like division and multiplication take longer times than operations like addition, subtraction and data movement. Having obtained the final count, it is not really possible to decide the exact time required by the algorithm, but can be thought of as a fair approximation. Identifying the more complex and essential operations or functions and the time required for them is also a way of analyzing since the time required for simple instructions become negligible for lengthier algorithms. (Note that we are more interested in comparing an algorithm with another instead of actually evaluating them with respect to their costs. Hence, these approximations most often do not affect our final judgment). Thus, the problem of analyzing algorithms reduces to identifying the most costly instructions and summing up the time required by them. A given algorithm may work very efficiently with a few data sets, but may become sluggish with others. Hence, the choice of sufficient number of data sets, representing all possible cases becomes important, while analyzing an algorithm.

If there is more than one possible way of solving a problem, then one may think of more than one algorithm for the same problem. Hence, it is necessary to know in what domains these algorithms are applicable. Data domain is an important aspect to be known in the field of algorithms. Once we have more than one algorithm for a given problem, how do we choose the best among them? The solution is to devise some data sets and determine a performance profile for each of the algorithms. A best case data set can be obtained by having all distinct data in the set. But, it is always complex to determine a data set, which exhibits some average behavior, for all kinds of algorithms.

Analysis of algorithms is a challenging area, which needs great mathematical skills. Usage of mathematics allows us to make a quantitative judgment about the value of an algorithm. This quantitative value can be used to select the best one out of many algorithms designed to solve the same problem. In order to obtain this quantitative value, an algorithm can be analyzed at two different stages. An algorithm can be analyzed just by looking in to the algorithm i.e., without executing the algorithm. This type of analyzing is called a priori analysis. In this type of analysis, one obtains a function (of some relevant parameters), which bounds the computing time of the algorithm. That is we get a lower limit and an upper such that the computing time of the algorithm always lies in between these limits irrespective of the nature of the data sets. In the case of complex algorithm, analyzing and determining the parameters for time and space consumption, without actually executing the program is a challenging one. On the other hand the algorithm can be tested through its execution and the actual time memory required can be determined.

This way of knowing the performance of the algorithm is called a posteriori analysis. In this analysis, we obtain statistics, by running the program and hence we get the accurate cost of the algorithm's execution. When compared to a priori analysis, the a posteriori analysis can easily be comprehended. Thus, we feel it is better to understand more about the a posteriori analysis with an example.

To begin with, we note that the actual time taken for execution depends on the computer on which it is run – which we do not know at present. Also, most often, the time taken by the algorithm depends on the number of data items. It is illogical to expect the algorithm to take the same (or even similar) quantum of execution time, when it is operating with say 10 data items and again with 10,000 data items. Hence, the most important step in a priori analysis is identifying statements which consume more time. Such statements can be selected because they are complex like, division/multiplication or because they get executed many times or more often both. Based on these we can arrive at a sort of approximation to the actual execution time.

Consider the following examples:

```
a)  x ← y * z;
b)  For i ← 1 to n do
            x ← y * z;
    For end
c)  For j ← 1 to n do
    For i ← 1 to n do
            x ← y * z;
    For end
    For end
```

We do not actually know what is the time taken for multiplication, but we can assume that the example (a) takes one unit of time, (b) takes n units of time (because it gets executed n times or it's frequency count is n and the example (c) takes n^2 units of time (gets executed n * n time or it's frequency count is n2). These values of 1, n and n^2, are said be in increasing order of magnitude.

Philosophically, the above can be interpreted as follows. To travel a given distance, a plane takes negligible time, motor car takes some time, a cycle takes much longer, a person walking takes even more time. While one can clearly see that the actual time taken depends on the distance and also the actual speeds of the vehicles in question, their "orders" are fixed. Given the orders, we directly say the plane takes the least time and the walker takes the maximum time. This is a priori analysis and thus requires a lot of a priori knowledge about the data and the functionalities.

To actually find the time taken by algorithms, it is necessary to execute them actually and note the timings. However, to make things complex, the performance of an algorithm often depends on the type of the inputs and the order in which they are given. Hence, the performance of an algorithm cannot be labeled by one value, but often requires three different cases like best case performance, average case performance and worst case performance.

Despite the fact that two different algorithms to solve the same problem are represented by order notations, it is not always possible to say which among those is the best one. For instance, consider the problem of finding out the sum of first n natural numbers. Following are the two algorithms to achieve the same.

```
Algorithm: A
Input: n, an integer
Output: S, sum of first n natural numbers
Method:
    S=0;
    For i=1 to n do
        S = S + i
    For end
        Print S
Algorithm ends
```

```
Algorithm: B
Input: n, an integer
Output: S, sum of first n natural numbers
Method:
    S=0;
    S=(n*(n+1))/2;
    Print S
Algorithm ends
```

Although, both the algorithms **A** and **B** accomplish the same desired task of finding out the sum of first ***n*** natural numbers, they have their own behaviors. Let us assume that all the arithmetic operations take equal time (one unit) and let us also assume the assignment operation takes negligible time when compared to any arithmetic operations and hence can be neglected. According to this assumption the algorithm A takes, for a given ***n*** value, ***n*** unit of time ('for' loop runs n times) while the algorithm **B** takes always, exactly 3 unit of time (one addition, one multiplication and one division), irrespective of ***n*** value.

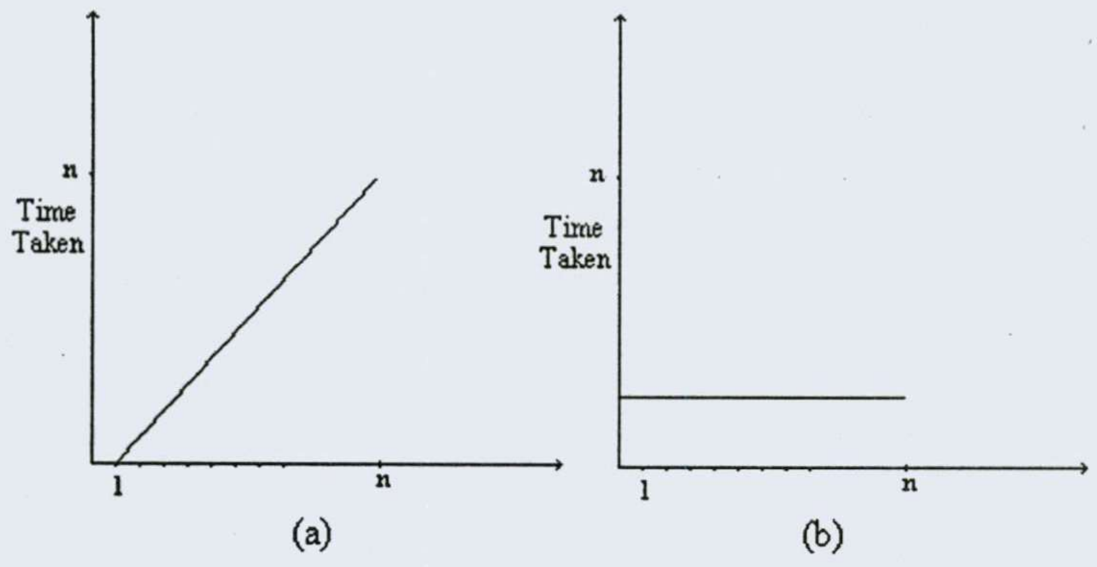

Fig. 1.1 Graph of time taken by algorithm **A** and **B**

If we plot the graph of the time taken by both the algorithms A and B, then the graph of A is linearly increasing graph(See Fig.1.1(a)) and the graph of B is a constant graph (See Fig.1.1(b)).

Therefore, it shall be noticed that the behavior (time taken by) of the algorithm A depends on the value of n and the behavior of (time taken by) the algorithm B is independent of n value. Therefore, one may feel that the algorithm B is always preferred to the algorithm A. However, it shall be noticed that the algorithm B is not as simple as the algorithm A from the point of view of understanding its functionality as one should have familiarity with the formula in case of algorithm B. That is what we say tradeoff between simplicity and efficiency.

1.4 NOTATION FOR PROGRAMS

It is important to decide how we are going to *describe* our algorithms. If we try to explain them in English, we rapidly discover that natural languages are not at all suited to this kind of thing. To avoid confusion, we shall in future specify our algorithms by giving a corresponding *program.* We assume that the reader is familiar with at least one well-structured programming language such as Pascal. However, we shall not confine ourselves strictly to any particular programming language: in this way, the essential points of an algorithm will not be obscured by relatively unimportant programming details, and it does not really matter which well-structured language the reader prefers.

A few aspects of our notation for programs deserve special attention. We use phrases in English in our programs whenever this makes for simplicity and clarity. Similarly, we use mathematical language, such as that of algebra and set theory, whenever appropriate-including symbols such as and Li introduced in Section 1.4.7. As a consequence, a single "instruction" in our programs may have to be translated into several instructions-perhaps a **while** loop-if the algorithm is to be implemented in a conventional programming language. Therefore, you should not expect to be able to run the algorithms we give directly: you will always be obliged to make the necessary effort to transcribe them into a "real" programming language. Nevertheless, this approach best serves our primary purpose, to present as clearly as possible the basic concepts underlying our algorithms.·

To simplify our programs further, we usually omit declarations of scalar quantities (integer, real, or Boolean). In cases where it matters-as in recursive **functions** and **procedures-all** variables used are implicitly understood to be *local* variables, unless the context makes it clear otherwise. In the same spirit of simplification, proliferation of **begin** and **end** statements, that plague programs written in Pascal, is avoided: the range of statements such as **if, while,** or **for,** as well as that of declarations such as **procedure, function,** or **record,** is shown by indenting the statements affected. The statement **return** marks the dynamic end of a **procedure** or a **function,** and in the latter case it also supplies the value of the **function.**

We do not declare the type of parameters in **procedures** and **functions,** nor the type of the result returned by a **function,** unless such declarations make the algorithm easier to understand. Scalar parameters are passed by value, which means they are treated as local variables within the **procedure** or **function,** unless they are declared to be **var** parameters, in which case they can be used to return a value to the calling program. In contrast, **array** parameters are passed by reference, which means that any modifications made within the **procedure** or **function** are reflected in the array actually passed in the calling statement.

Finally, we assume that the reader is familiar with the concepts of recursion, **record,** and pointer. The last two are denoted exactly as in Pascal, except for the omission of **begin** and **end** in **records.** In particular, pointers are denoted by the symbol " ↑ ".

To wrap up this section, here is a program for multiplication. Here ÷ denotes integer division: any fraction in the answer is discarded. We can compare this program to the informal English description of the same algorithm.

```
function Multiply(m, n)
    $result \leftarrow 0$
    repeat
        if m is odd then $result \leftarrow result + n$
        $m \leftarrow m \div 2$
        $m \leftarrow n + n$
    until $m = 1$
return result
```

1.5 TIME COMPLEXITY

The number of (machine) instructions which a program executes during its running time is called its **time complexity** in computer science. This number depends primarily on the size of the program's input, that is approximately on the number of the strings to be sorted (and their length) and the algorithm used. So approximately, the time complexity of the program "sort an array of n strings by minimum search" is described by the expression $c \cdot n^2$. c is a constant which depends on the programming language used, on the quality of the compiler or interpreter, on the CPU, on the size of the main memory and the access time to it, on the knowledge of the programmer, and last but not least on the algorithm itself, which may require simple but also time consuming machine instructions. (For the sake of simplicity we have drawn the factor 1/2 into c here.) So while one can make c smaller by improvement of external circumstances (and thereby often investing a lot of money), the term n^2, however, always remains unchanged.

1.6 SPACE COMPLEXITY

The better the time complexity of an algorithm is, the faster the algorithm will carry out his work in practice. Apart from time complexity, its **space complexity** is also important: This is essentially the number of memory cells which an algorithm needs. A good algorithm keeps this number as small as possible, too.

There is often a **time-space-tradeoff** involved in a problem, that is, it cannot be solved with few computing time *and* low memory consumption. One then has to make a compromise and to exchange computing time for memory consumption or vice versa, depending on which algorithm one chooses and how one parameterizes it.

1.7 ASYMPTOTIC NOTATIONS

Asymptotic notation is a method of expressing the order of magnitude of an algorithm during the a priori analysis. These order notations do not take into account all program and machine dependent factors i.e., given an algorithm, if it is realized and executed with the aid of different programming languages, then it is obvious to find different performance response for the same algorithm. In addition, if the same program is run on different computers, although the machine speeds are same, their performances may differ. But, the a priori analysis will not have these variations. There are several kinds of mathematical notations that are used in asymptotic representations.

Definition: f(n) = O(g(n)) (read as "f of n equals big oh of g of n"), if and only if there exist two positive, integer constants c and n_0 such that
ABS(f(n)) $\leq$ C*ABS(g(n)) for all n $\geq n_0$

In other words, suppose we are determining the computing time, f(n) of some algorithm where n may be the number of inputs to the algorithm, or the number of outputs, or their sum or any other relevant parameter. Since f(n) is machine dependent (it depends on which computer we are working on). An a priori analysis cannot determine f(n), the actual complexity as described earlier. However, it can determine a g(n) such that f(n)=O(g(n)). An algorithm is said to have a computing time O(g(n)) (of the order of g(n)), if the resulting times of running the algorithm on some computer with the same type of data but for increasing values of n, will always be less than some constant times |g(n)|. We use some polynomial of n, which acts as an upper limit, and we can be sure that the algorithm does not take more than the time prescribed by the upper limit. For instance, let us consider the following algorithm,

```
Algorithm: Sum
Input: n, number of values to be added
        A, array of n elements
Output: S, sum of n elements in A
Method :
        (1)  S=0;
        (2)  For i= 1 to n do
        (3)  S= S + A[ i ];
             For end
        (4)  Output S;
Algorithm ends.
```

In the above algorithm, statement (1) is executed 1 time, statement (2) is executed n+1 times, statement (3) is executed n times, and statement (4) is executed 1 time. Thus, the total time taken is 2n+3.

In order to represent the time complexity of the above algorithm as f(n)=O(n), it is required to find the integer constants c and n0, which satisfy the above definition of O notation. i.e., an algorithm with the time complexity 2n + 3 obtained from a priori analysis can be represented as O(n) because $2n + 3 \leq 3n$ for all $n \geq 3$ here c=3 and $n_0 = 3$.

Some more examples:

The function $3n + 2 = O(n)$ as $3n + 2 \leq 4n$ for all $n \geq 2$.

$3n + 3 = O(n)$ as $3n + 3 \leq 4n$ for all $n \geq 3$.

$10n^2 + 4n + 2 = O(n^2)$ as $10n^2 + 4n + 2 \leq 11n^2$ for all $n \geq 5$.

$6 * 2^n + n^2 = O(n^2)$ as $6 * 2^n + n^2 \leq 7 * 2^n$ for all $n \geq 4$.

The most commonly encountered complexities are O(1), O(log n), O(n), O(n log n), $O(n^2)$, $O(n^3)$ and $O(2^n)$. Algorithms of higher powers of n are seldom solvable by simple methods. O(1) means a computing time that is constant. O(n) is called linear, $O(n^2)$ is called quadratic, $O(n^3)$ is called cubic and $O(2^n)$ is called exponential. The commonly used complexities can thus be arranged in an increasing order of complexity as follows.

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

If we substitute different values of n and plot the growth of these functions, it becomes obvious that at lower values of n, there is not much difference between them. But as n increases, the values of the higher powers grow much faster than the lower ones and hence the difference increases. For example at n = 2, 3, 4, …, 9 the values of 2^n happens to be lesser than n^3 but once $n \geq 10$, 2^n shows a drastic growth.

The O - notation discussed so far is the most popular of the asymptotic notations and is used to define the upper bound of the performance of an algorithm also referred to as the worst case performance of an algorithm. But it is not the only complexity we have. Sometimes, we may wish to determine the lower bound of an algorithm i.e., the least value, the complexity of an algorithm can take. This is denoted by Ω (omega).

Definition: $f(n) = \Omega(g(n))$ (read as "f of n equals omega of g of n) if and only if there exist positive non-zero constants C and n_0, such that for all $ABS(f(n)) \geq C*ABS(g(n))$ for all $n \geq n_0$.

Some Examples: The function $3n + 2 = \Omega(n)$ as $3n + 2 \geq 3n$ for $n \geq 1$.

$3n + 3 = \Omega\ (n)$ as $3n + 3 \geq 3n$ for $n \geq 1$.

$10n^2 + 4n + 2 = \Omega(n^2)$ as $10n^2 + 4n + 2 \geq n^2$ for $n^3 \geq 1$.

$10n^2 + 4n + 2 = \Omega(n)$ as $10n^2 + 4n + 2 \geq n$ for $n \geq 1$.

$10n^2 + 4n + 2 = \Omega(1)$ as $10n^2 + 4n + 2 \geq 1$ for $n \geq 1$.

In some cases both the upper and lower bounds of an algorithm can be the same. Such a situation is described by the θ-notation.

Definition: $f(n) = \theta(g(n))$ if and only if there exist positive constants C_1, C_2 and n_0 such that for all $n > n_0$, $C_1 |g(n)| \le f(n) \le C_2 |g(n)|$

Some Examples: $3n + 2 = \theta(n)$ as $3n + 2 \ge 3n$ for all $n \ge 2$ and $3n + 2 \le 4n$ for all $n \ge 2$, so $C_1 = 3$ and $C_2 = 4$ and $n_0 = 2$.

1.8 SUMMARY

In this unit, we have introduced algorithms. A glimpse of all the phases we should go through when we study an algorithm and its variations was given. In the study of algorithms, the process of designing algorithms, validating algorithms, analyzing algorithms, coding the designed algorithms, verifying, debugging and studying the time involved for execution were presented. All in all, the basic idea behind the analysis of algorithms is given in this unit.

1.9 KEYWORDS

1) Algorithm
2) Space complexity
3) Time complexity
4) Asymptotic notation

1.10 QUESTIONS FOR SELF STUDY

1) What is an algorithm? Explain its characteristics?
2) What is meant by analysis of algorithms? Why is it needed?
3) What is meant by space complexity and time complexity of an algorithm? Explain.
4) What is meant by asymptotic notation? Explain.

1.11 EXCERCISES

1) Design algorithms for the following and determine the frequency counts for all statements in the devised algorithms and express their complexities with the help of asymptotic notations.
    a) To test whether the three numbers represent the sides of a right angle triangle.
    b) To test whether a given point p(x, y) lies on x-axis or y-axis or in I/II/III/IV quadrant.
    c) To compute the area of a circle of a given circumference
    d) To locate a specific word in a dictionary.
    e) To find the first n prime numbers.
    f) To add two matrices.
    g) To multiply two matrices.
    h) To search an element for its presence/absence in a given list of random data elements without sorting the list.
    i) To find the minimum and maximum in the given list of data elements without sorting the list.

1.12 REFERENCES

1) Fundamentals of Algorithmics: Gilles Brassard and Paul Bratley, Prentice Hall Englewood Cliffs, New Jersey 07632.
2) Sartaj Sahni, 2000, Data structures, Algorithms and Applications in C++, McGraw Hill International Edition.
3) Goodman And Hedetniemi, 1987, Introduction to the Design and Analysis of Algorithms, Mcgraw Hill International Editions.

UNIT- 2

PRACTICAL COMPLEXITIES AND PERFORMANCE MEASUREMENT

STRUCTURE

2.0 OBJECTIVES

After studying this unit you should be able to

- Explain how test algorithms.
- Measure the performance of an algorithm.
- Determine how the time requirements vary as the instance characteristics change.

2.1 INTRODUCTION

Performance measurement is the process of executing a correct program on different data sets to measure the time and space that it takes to compute the results. Complexity of a program is generally some function of the instance characteristics.

2.2 HOW TO TEST ALGORITHMS?

The ultimate test is performed to ensure that the program developed on the basis of the designed algorithm, runs satisfactorily. Testing a program involves two phases viz., debugging and profiling. Debugging is the process of executing a program with sample datasets to determine if the results obtained are satisfactory. When unsatisfactory results are generated, suitable changes are to be incorporated in the program to get the desired results. However, it is pointed out that "debugging can only indicate the presence of errors but not their absence" i.e., a program that yields unsatisfactory results on a sample data set is definitely faulty, but on the other hand a program producing the desirable results on one/more data sets need not be correct. In order to actually prove that a program is perfect, a process of "proving" is necessary wherein the program is analytically proved to be correct and in such cases, it is bound to yield perfect results for all possible sets of data.

On the other hand, profiling or performance measurement is the process of executing a correct program on different data sets to measure the time and space that it takes to compute the results. that several different programs may do a given job satisfactorily. But often, especially when large data sets are being operated upon, the amount of space and the computation time required become important. In fact, a major portion of the study of algorithms pertains to the study of time and space requirements of algorithms. The following section discusses the actual way of measuring the performance of an algorithm.

2.3 PROFILING (Performance Measurement)

This is the final stage of algorithm evaluation. A question to be answered when the program is ready for execution, (after the algorithm has been devised, made a priori analysis of that, coded into a program debugged and compiled) is how do we actually evaluate the time taken by the program? Obviously, the time required to read the input data or give the output should not be taken into account. If somebody is keying in the input data through the keyboard or if data is being read from an input device, the speed of operation is dependent on the speed of the device, but not on the speed of the algorithm. So, we have to exclude that time while evaluating the programs. Similarly, the time to write out the output to any device should also be excluded. Almost all systems provide a facility to measure the elapsed system time by using stime() or other similar functions. These can be inserted at appropriate places in the program and they act as stop clock measurement. For example, the system time can be noted down just after all the inputs have been read. Another reading can be taken just before the output operations start. The difference between the two readings is the actual time of run of the program. If multiple inputs and outputs are there, the counting operations should be included at suitable places to exclude the I/O operations.

It is not enough if this is done for one data set. Normally various data sets are chosen and the performance is measured as explained above. A plot of data size n v/s the actual time can be drawn which gives an insight into the performance of the algorithm.

The entire procedure explained above is called "profiling". However, unfortunately, the times provided by the system clock are not always dependable. Most often, they are only indicative in nature and should not be taken as an accurate measurement. Especially when the time durations involved are of the order of 1-2 milliseconds, the figures tend to vary often between one run and the other, even with the same program and all same input values.

Irrespective of what we have seen here and in the subsequent discussions, devising algorithms is both an art and a science. As a science part, one can study certain standard methods (as we do in this course) but there is also an individual style of programming which comes only by practice.

2.4 PRACTICAL COMPLEXITIES

We have seen that the time complexity of a program is generally some function of the instance characteristics. This function is very useful in determining how the time requirements vary as the instance characteristics change. We can also use the complexity function to compare two programs P and Q that perform the same task. Assume that program P has complexity θ(n) and that program Q has complexity $\theta(n^2)$. We can assert that, program P is faster than program Q is for sufficiently large n. To see the validity of this assertion, observe that the actual computing time of P is bounded from above by cn for some constant c and for all n n > nl, while that of Q is bounded from below by dn2 for some constant d and all n, $n \geq n_2$. Since $cn \leq dn^2$ for $n \geq c/d$, program *P* is faster than program Q whenever $n \geq \max\{n_1, n_2, c/d)$.

One should always be cautiously aware of the presence of the phrase *sufficiently large* in the as assertion of the preceding discussion. When deciding which of the two programs to use, we must know whether the n we are dealing with is, in fact, sufficiently large. If program P actually runs in 10^6n milliseconds while program Q runs in n^2 milliseconds and if we always have $n \leq 10^6$, then program Q is the one to use.

To get a feel for how the various functions grow with n, you should study figures 2.1 and 2.2. These figures show that 2^n grows very rapidly with n. In fact, if a program needs 2^n steps for execution, then when n = 40, the number of steps needed is approximately $1.1*10^{12}$. On a computer performing 1.000.000,000 steps per second, this program would require about 18.3 minutes. If n = 50, the same program would run for about 13 days on this computer. When n = 60, about 310.56 years will be required to execute the program, and when n = 100, about $4*10^{13}$ years will be needed. We can conclude that the utility of programs with exponential complexity is limited to small n (typically n < 40).

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65,536
5	32	160	1024	32,768	4,294,967,296

Figure 2.1 values of various functions

Programs that have a complexity that is a high-degree polynomial are also of limited utility. For example, if a program needs n^{10} steps, then our 1.000,000,000 steps per second computer needs 10 seconds when n = 10; 3171 years when n = 100; and 3.17 + 1013 years when n = 1000. If the program's complexity had been n^3 steps instead, then the computer would need 1 second when n = 1000, 110.67 minutes when n = 10,000 and 11.57 days when n = 100,000.

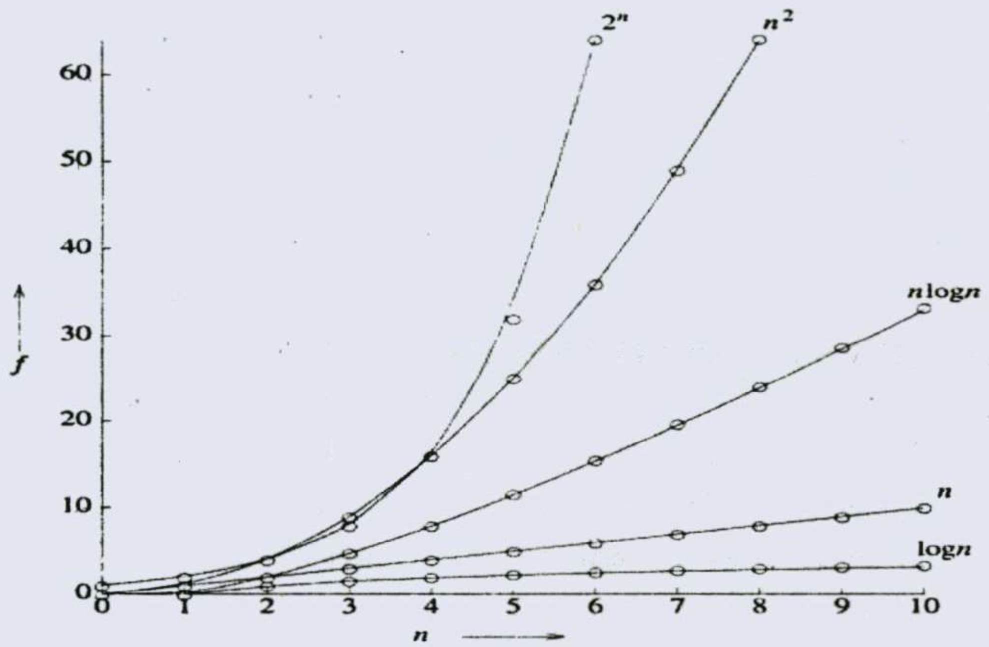

Figure 2.2 Plot of various functions

Figure 2.2 above gives the time that a 1,000,000,000 instructions per second computer needs to execute a program of complexity f (n) instructions. One should note that currently only the fastest computers can execute about 1.000,000,000 instructions per second. From a practical standpoint, it is evident that for reasonably large n only programs of small complexity are feasible.

2.5 SUMMARY

In this unit, we have introduced algorithms. A glimpse of all the phases we should go through when we study an algorithm and its variations was given. In the study of algorithms, the process of designing algorithms, validating algorithms, analyzing algorithms, coding the designed algorithms, verifying, debugging and studying the time involved for execution were presented. All in all, the basic idea behind the analysis of algorithms is given in this unit.

2.6 KEYWORDS

1) Algorithm
2) Space complexity
3) Time complexity
4) Asymptotic notation
5) Profiling

2.7 QUESTIONS FOR SELF STUDY

1) What is an algorithm? Explain its characteristics?
2) How to test algorithms? Why is it needed?
3) Explain Practical complexities of algorithms?
4) What is meant by profiling? Explain.

2.8 EXCERCISES

1) How do we actually evaluate the time taken by the program?
2) Discuss about the Practical complexities considering few example functions.

2.9 REFERENCES

1) Fundamentals of Algorithmics: Gilles Brassard and Paul Bratley, Prentice Hall Englewood Cliffs, New Jersey 07632.
2) Sartaj Sahni, 2000, Data structures, Algorithms and Applications in C++, McGraw Hill International Edition.
3) Goodman And Hedetniemi, 1987, Introduction to the Design and Analysis of Algorithms, Mcgraw Hill International Editions.

UNIT- 3

ANALYZING CONTROL STRUCTURES, USING A BAROMETER, SUPPLEMENTARY EXAMPLES

STRUCTURE

3.0 OBJECTIVES

After studying this unit you should be able to

- Analyze Control Structures.
- Analyze algorithms using Barometer.
- Evaluate recursive algorithms.

3.1 INTRODUCTION

When you have several different algorithms to solve the same problem, you have to decide which one is best suited for your application. An essential tool for this purpose is the *analysis of algorithms*. Only after you have determined the efficiency of the various algorithms you will be able to make a well-informed decision. But there is no magic formula for analyzing the efficiency of algorithms. It is largely a matter of judgement, intuition and experience. Nevertheless, there are some basic techniques that are often useful, such as knowing how to deal with control structures and recurrence equations. This unit covers the most commonly used techniques and illustrates them with examples.

3.2 ANALYZING CONTROL STRUCTURES

The analysis of algorithms usually proceeds from the inside out. First, we determine the time required by individual instructions (this time is often bounded by a constant); then we combine these times according to the control structures that combine the instructions in the program. Some control structures such as sequencing - putting one instruction after another - are easy to analyze whereas others such as while loops are more subtle. In this unit, we give general principles that are useful in analyses involving the most frequently encountered control structures, as well as examples of the application of these principles.

Sequencing

Let P_1 and P_2 be two fragments of an algorithm. They may be single instructions or complicated sub algorithms. Let t_1 and t_2 be the times taken by P_1 and P_2, respectively. These times may depend on various parameters, such as the instance size. The **sequencing rule** says that the time required to compute "$P_1; P_2$ ", that is first P_1 and then P_2, is simply $t_1 + t_2$. By the maximum rule, this time is in θ (max (t_1, t_2)). Despite its simplicity, applying this rule is sometimes less obvious than it may appear. For example, it could happen that one of the parameters that control t_2 depends on the result of the computation performed by P_1. Thus, the analysis of " P_1; P_2" cannot always be performed by considering P_1 and P_2 independently.

"For" loops

For loops are the easiest loops to analyse. Consider the following loop.

```
for $i \leftarrow 1$ to $m$ do $P(i)$
```

Here and throughout the book, we adopt the convention that when $m = 0$ this is not an error; it simply means that the controlled statement $P(i)$ is not executed at all. Suppose this loop is part of a larger algorithm, working on an instance of size n. (Be careful not to confuse m and n.) The easiest case is when the time taken by $P(i)$ does not actually depend on i, although it could depend on the instance size or, more generally, on the instance itself. Let t denote the time required to compute $P(i)$. In this case, the obvious analysis of the loop is that $P(i)$ is performed m times, each time at a cost of t, and thus the total time required by the loop is simply $l = mt$. Although this approach is usually adequate, there is a potential pitfall: we did not take account of the time needed for *loop control.* After all, **for** loop is shorthand for something like the following **while** loop.

```
$i \leftarrow 1$
while $i < m$ do
    $P(i)$
    $i \leftarrow i + 1$
```

In most situations, it is reasonable to count at unit cost the test $i < m$, the instructions $i \leftarrow 1$ and $i \leftarrow i + 1$, and the sequencing operations **(go to)** implicit in the **while** loop. Let c be an upper bound on the time required by each of these operations. The time l taken by the loop is thus bounded above by

$l \leq$	c	for $i \leftarrow 1$
	$+ (m + 1) c$	for the tests $i \leq m$
	$+ mt$	for the executions of $P(i)$
	$+ mc$	for the executions of $i \leftarrow i + 1$
	$+ mc$	for the sequencing operations
$\leq$	$(t + 3c) m + 2c$.	

Moreover this time is clearly bounded below by mt. If c is negligible compared to t, our previous estimate that f is roughly equal to mt was therefore justified, except for one crucial case: $4l \approx mt$ is completely wrong when $m = 0$ (it is even worse if m is negative!).

Resist the temptation to say that the time taken by the loop is in $\theta(mt)$ on the pretext that the θ notation is only asked to be effective beyond some threshold such as $m > 1$. The problem with this argument is that if we are in fact analyzing the entire algorithm rather than simply the **for** loop, the threshold implied by the θ notation concerns *n,* the instance size, rather than *m,* the number of times we go round the loop, and $m = 0$ could happen for arbitrarily large values of *n*. On the other hand, provided *t* is bounded *below* by some constant (which is always the case in practice), and provided there exists a threshold no such that $m > 1$ whenever $n \geq n_o$.

The analysis of for loops is more interesting when the time $t(i)$ required for *P(i)* varies as a function of *i*. (In general, the time required for *P(i)* could depend not only on *i* but also on the instance size *n* or even on the instance itself.) If we neglect the time taken by the loop control, which is usually adequate provided $m > 1$, the same **for** loop

```
for i ← 1 to $m$ do $P(i)$
```

takes a time given not by a multiplication but rather by a sum: it is $\sum_{i=1}^{m} t(i)$. We illustrate the analysis of **for** loops with a simple algorithm for computing the Fibonacci sequence as shown below.

```
function Fibiter(n)
    i ← 1; j ← 0
    for k ← 1 to n do  j ← i + j
                       i ← j - i
    return j
```

If we count all arithmetic operations at unit cost, the instructions inside the **for** loop take constant time. Let the time taken by these instructions be bounded above by some constant c. Not taking loop control into account, the time taken by the for loop is bounded above by *n* times this constant: nc. Since the instructions before and after the loop take negligible time, we conclude that the algorithm takes a time in *O(n)*. Similar reasoning yields that this time is also in *O(n),* hence it is in *O((n)*. We know that it is not reasonable to count the additions involved in the computation of the Fibonacci sequence at unit cost unless *n* is very small. Therefore, we should take account of the fact that an instruction as simple as *"j - i + j "* is increasingly expensive each time round the loop. It is easy to program long-integer additions and subtractions so that the time needed to add or subtract two integers is in the exact order of the number of figures in the larger operand. To determine the time taken by the k^{th} trip round the loop, we need to know the length of the integers involved. We can prove by mathematical induction that the values of i and j at the end of the k-th iteration are f_{k-1} and f_k, respectively. This is precisely why the algorithm works: it returns the value of j at the end of the n^{th} iteration, which is therefore f, as required. Moreover, the Moivre's formula tells us that the size of f_k is in $\theta(k)$. Therefore, the k^{th} iteration takes a time $\theta\ (k - 1) + \theta\ (k)$, which is the same as $\theta(k)$. Let c be a constant such that this time is bounded above by ck for all $k > 1$. If we neglect the time required for the loop control and for the instructions before and after the loop, we conclude that the time taken by the algorithm is bounded above by

$$\sum_{k=1}^{n} ck = c \sum_{k=1}^{n} k = c\,\frac{n\,(n+1)}{2} \in O(n^2).$$

Similar reasoning yields that this time is in $\Omega(n^2)$, and therefore it is in $\theta(n^2)$. Thus it makes a crucial difference in the analysis of *Fibrec* whether or not we count arithmetic operations at unit cost.

The analysis of *for* loops that start at a value other than 1 or proceed by larger steps should be obvious at this point. Consider the following loop for example.

```
for $i \leftarrow 5$ to $m$ step 2 do $P(i)$
```

Here, $P(i)$ is executed $((m - 5) \div 2) + 1$ times provided $m \geq 3$. (For a **for** loop to make sense, the endpoint should always be at least as large as the starting point *minus* the step).

Recursive calls

The analysis of recursive algorithms is usually straightforward, at least up to a point. Simple inspection of the algorithm often gives rise to a recurrence equation that "mimics" the flow of control in the algorithm. Once the recurrence equation has been obtained, some general techniques can be applied to transform the equation into simpler nonrecursive asymptotic notation. As an example, consider the problem of computing the Fibonacci sequence with the recursive algorithm *Fibrec*.

```
function Fibrec($n$)
  if $n < 2$ then return $n$
  else return Fibrec($n-1$) + Fibrec($n-2$)
```

Let *T(n)* be the time taken by a call on *Fibrec(n)*. If $n < 2$, the algorithm simply returns *n*, which takes some constant time *a*. Otherwise, most of the work is spent in the two recursive calls, which take time *T(n -1)* and *T(n* - 2), respectively. Moreover, one addition involving f_{n-1} and f_{n-2} (which are the values returned by the recursive calls) must be performed, as well as the control of the recursion and the test "if $n < 2$". Let *h(n)* stand for the work involved in this addition and control, that is the time required by a call on *Fibrec (n)* ignoring the time spent inside the two recursive calls. By definition of *T(n)* and *h(n)*, we obtain the following recurrence.

$$T(n) = \begin{cases} a & \text{if } n = 0 \text{ or } n = 1 \\ T(n-1) + T(n-2) + h(n) & \text{otherwise} \end{cases}$$

If we count the additions at unit cost, *h(n)* is bounded by a constant and we conclude that *Fibrec(n)* takes a time exponential in *n*. This is *double* exponential in the size of the instance since the *value* of *n* is exponential in the *size* of *n*.

If we do not count the additions at unit cost, *h(n)* is no longer bounded by a constant. Instead *h(n)* is dominated by the time required for the addition of f_{n-1} and f_{n-2} for sufficiently large *n*. We know that this addition takes a time in the exact order of *n*. Therefore $h(n) \in \theta(n)$. Surprisingly, the result is the same regardless of whether *h (n)* is constant or linear: it is still the case that $T(n) \in \theta(fn)$. In conclusion, *Fibrec(n)* takes a time exponential in *n* whether or not we count additions at unit cost! The only difference lies in the multiplicative constant hidden in the θ notation.

"While" and "repeat" loops

While and repeat loops are usually harder to analyze than for loops because there is no obvious a priori way to know how many times we shall have to go round the loop. The standard technique for analyzing these loops is to find a function of the variables involved whose value decreases each time around. To conclude that the loop will eventually terminate, it suffices to show that this value must be a positive integer. (You cannot keep decreasing an integer indefinitely.) To determine how many times the loop is repeated, however, we need to understand better how the value of this function decreases.

An alternative approach to the analysis of while loops consist of treating them like recursive algorithms. The analysis of **repeat** loops is carried out similarly.

We shall study *binary search* algorithm, which illustrates perfectly the analysis of while loops. The purpose of binary search is to find an element x in an array T [1 .. n] that is in nondecreasing order. Assume for simplicity that x is guaranteed to appear at least once in T. We require to find an integer i such that $1 < i < n$ and $T\,[i] = x$. The basic idea behind binary search is to compare x with the element y in the middle of T. The search is over if $x = y$; it can be confined to the upper half of the array if $x > y$; otherwise, it is sufficient to search the lower half. We obtain the following algorithm.

```
function Binary_Search($T[1..n]$, $x$)
  $i \leftarrow 1$; $j \leftarrow n$;
  while $i < j$ do
     $k \leftarrow (i+j) \div 2$;
     case $x < T[k]$: $j \leftarrow k - 1$;
     case $x = T[k]$: $i, j \leftarrow k$ {return $k$};
     case $x > T[k]$: $i \leftarrow k + 1$;
  return $i$
```

Recall that to analyze the running time of a **while** loop, we must find a function of the variables involved whose value decreases each time round the loop. In this case, it is natural to consider $j - i + 1$, which we shall call d. Thus d represents the number of elements of T still under consideration. Initially, $d = n$. The loop terminates when $i \geq j$, which is equivalent to $d \leq 1$. Each time round the loop, there are three possibilities: either j is set to k -1, i is set to $k + 1$, or both i and j are set to k. Let d and $\hat{d}$ stand respectively for the value of $j - i + 1$ before and after the iteration under consideration. We use i, j, $\hat{\imath}$ and $\hat{\jmath}$ similarly. If $x < T[k]$, the instruction "$j \leftarrow k - 1$" is executed and thus , $\hat{\imath} = i$ and $\hat{\jmath} = [(i + j) \div 2] - 1$. Therefore,

$$\hat{d} = \hat{\jmath} - \hat{\imath} + 1 = j - (i + j) \div 2 \leq j - (i + j - 1)/2 = d/2$$

Similarly, if $x > T[k]$, the instruction "$i \leftarrow k + 1$" is executed and thus

$$\hat{\imath} = [(i + j) \div 2] + 1 \text{ and } \hat{\jmath} = j$$

$$\hat{d} = \hat{\jmath} - \hat{\imath} + 1 = j - (i + j) \div 2 \leq j - (i + j - 1)/2 = d/2$$

Finally, if $x = T[k]$, then i and j are set to the same value and thus $\hat{d} = 1$;; but d was at least 2 since otherwise the loop would not have been reentered. We conclude that $\hat{d} \leq d/2$ whichever case happens, which means that the value of d is at least halved each time round the loop. Since we stop when $d \leq 1$, the process must eventually stop, but how much time does it take?

To determine an upper bound on the running time of binary search, let d_l denote the value of $j - i + 1$ at the end of the l^{th} trip round the loop for $l \geq 1$ and let $d_o = n$. Since $d_l - 1$ is the value of $j - i + 1$ before starting the l^{th} iteration, we have proved that $d_l \leq d_{l-1}/2$ for all $l \geq 1$. It follows immediately by mathematical induction that $d_l \leq n / 2^l$. But the loop terminates when $d \leq 1$, which happens at the latest when $l = [\lg n]$. We conclude that the loop is entered at most $[\lg n]$ times. Since each trip round the loop takes constant time, binary search takes a time in $O(\log n)$. Similar reasoning yields a matching lower bound of $\Omega(\log n)$ in the worst case, and thus binary search takes a time in $\theta(\log n)$. This is true even though our algorithm can go much faster in the best case, when x is situated precisely in the middle of the array.

3.3 USING A BAROMETER

The analysis of many algorithms is significantly simplified when one instruction or one test-can be singled out as *barometer.* A barometer instruction is one that is executed at least as often as any other instruction in the algorithm. (There is no harm if some instructions are executed up to a constant number of times more often than the barometer since their contribution is absorbed in the asymptotic notation). Provided the time taken by each instruction is bounded by a constant, the time taken by the entire algorithm is in the exact order of the number of times that the barometer instruction is executed.

This is useful because it allows us to neglect the exact times taken by each instruction. In particular, it avoids the need to introduce constants such as those bounding the time taken by various elementary operations, which are meaningless since they depend on the implementation, and they are discarded when the final result is expressed in terms of asymptotic notation. For example, consider the analysis of *Fibiter* algorithm when we count all arithmetic operations at unit cost. We saw that the algorithm takes a time bounded above by cn for some meaningless constant c, and therefore that it takes a time in $\theta(n)$. It would have been simpler to say that the instruction $j \leftarrow i + j$ can be taken as barometer, that this instruction is obviously executed exactly n times, and therefore the algorithm takes a time in $\theta(n)$. Selection sorting will provide a more convincing example of the usefulness of barometer instructions in the next section.

When an algorithm involves several nested loops, any instruction of the innermost loop can usually be used as barometer. However, this should be done carefully because there are cases where it is necessary to take account of the implicit loop control. This happens typically when some of the loops are executed zero times, because such loops do take time even though they entail no executions of the barometer instruction. If this happens too often, the number of times the barometer instruction is executed can be dwarfed by the number of times empty loops are entered-and therefore it was an error to consider it as a barometer. Consider for instance pigeon-hole sorting. Here we generalize the algorithm to handle the case where the elements to be sorted are integers known to lie between 1 and s rather than between 1 and 10000. Recall that $T[1. .n]$ is the array to be sorted and $U[1. .s]$ is an array constructed so that $U[k]$ gives the number of times integer k appears in T. The final phase of the algorithm rebuilds T in nondecreasing order as follows from the information available in U.

```
$i \leftarrow 0$
for $k \leftarrow 1$ to $s$ do
   while $U[k] \neq 0$ do
      $i \leftarrow i + 1$ ⟸
      $T[i] \leftarrow k$
      $U[k] \leftarrow U[k] - 1$
```

To analyze the time required by this process, we use "$U [k]$" to denote the value *originally* stored in $U[k]$ since all these values are set to 0 during the process. It is tempting to choose any of the instructions in the inner loop as a barometer. For each value of k, these instructions are executed $U[k]$ times. The total number of times they are executed is therefore $\sum_{k=1}^{s} U[k]$. But this sum is equal to n, the number of integers to sort, since the sum of the number of times that each element appears gives the total number of elements. If indeed these instructions could serve as a barometer, we would conclude that this process takes a time in the exact order of *n.* A simple example is sufficient to convince us that this is not necessarily the case. Suppose $U[k] = 1$ when k is a perfect square and $U[k] = 0$ otherwise. This would correspond to sorting an array T containing exactly once each perfect square between 1 and n^2, using $s = n^2$ pigeon-holes. In this case, the process clearly takes a time in $\Omega(n^2)$ since the outer loop is executed s times. Therefore, it cannot be that the time taken is in $\theta(n)$. This proves that the choice of the instructions in the inner loop as a barometer was incorrect. The problem arises because we can only neglect the time spent initializing and controlling loops provided we make sure to include something even if the loop is executed zero times.

The correct and detailed analysis of the process is as follows. Let a be the time needed for the test $U[k] \neq 0$ each time round the inner loop and let b be the time taken by one execution of the instructions in the inner loop, including the implicit sequencing operation to go back to the test at the beginning of the loop. To execute the inner loop completely for a given value of k takes a time $t_k = (1 + U[k])\, a + U[k]\, b$, where we add 1 to $U[k]$ before multiplying by a to take account of the fact that the test is performed each time round the loop *and* one more time to determine that the loop has been completed. The crucial thing is that this time is not zero even when $U[k] = 0$. The complete process takes a time $c + \sum_{k=1}^{s} (d + t_k)$ where c and d are new constants to take account of the time needed to initialize and control the outer loop, respectively. When simplified, this expression yields $c + (a + d)\, s + (a + b)\, n$. We conclude that the process takes a time in $\theta(n + s)$. Thus the time depends on two independent parameters n and *s;* it cannot be expressed as a function of just one of them. It is easy to see that the initialization phase of pigeon-hole sorting also takes a time in $\theta\ (n + s)$, unless *virtual initialization* is used in which case a time in $\theta(n)$ suffices for that phase. In any case, this sorting technique takes a time in $\theta(n + s)$ in total to sort n integers between 1 and s. If you prefer, the maximum rule can be invoked to state that this time is in $\theta(\max\,(n, s))$. Hence, pigeon-hole sorting is worthwhile but only provided s is small enough compared to n. For instance, if we are interested in the time required as a function only of the number of elements to sort, this technique succeeds in astonishing linear time if $s \in O(n)$ but it chugs along in quadratic time when $s \in \theta(n^2)$.

Despite the above, the use of a barometer is appropriate to analyze pigeon-hole sorting. Our problem was that we did not choose the proper barometer. Instead of the instructions *inside* the inner loop, we should have used the inner-loop *test* " $U[k] \neq 0$ " as a barometer. Indeed, no instructions in the process are executed more times than this test is performed, which is the definition of a barometer. It is easy to show that this test is performed exactly $n + s$ times, and therefore the correct conclusion about the running time of the process follows immediately without need to introduce meaningless constants.

In conclusion, the use of a barometer is a handy tool to simplify the analysis of many algorithms, but this technique should be used with care.

3.3 SUPPLEMENTARY EXAMPLES

In this section, we study several additional examples of analyses of algorithms involving loops, recursion, and the use of barometers.

Selection sort

Let's consider a selection sorting technique as shown below, which is a good example for the analysis of *nested* loops.

```
procedure select(T[1..n])
    for i ← 1 to n − 1 do
        minj ← i; minx ← T[i]
        for j ← i + 1 to n do
            if T[j] < minx then minj ← j
                                 minx ← T[j]
        T[minj] ← T[i]
        T[i] ← minx
```

Although the time spent by each trip round the inner loop is not constant, it takes longer time when ***T[j] < minx*** and is bounded above by some constant **c** (that takes the loop control into account). For each value of *i*, the instructions in the inner loop are executed **$n - (i + 1) + 1 = n - i$** times, and therefore the time taken by the inner loop is **$t(i) \leq (n - i)c$.**

The time taken for the ***i***-th trip round the outer loop is bounded above by $b + t(i)$ for an appropriate constant ***b*** that takes account of the elementary operations before and after the inner loop and of the loop control for the outer loop. Therefore, the total time spent by the algorithm is bounded above by

$$\begin{aligned}\sum_{i=1}^{n-1} b + (n-i)c &= \sum_{i=1}^{n-1} (b+cn) - c\sum_{i=1}^{n-1} i \\ &= (n-1)(b+cn) - cn(n-1)/2 \\ &= \frac{1}{2}cn^2 + \left(b - \frac{1}{2}c\right)n - b,\end{aligned}$$

which is in $o(n^2)$. Similar reasoning shows that this time is also in $\Omega(n^2)$ in all cases, and therefore selection sort takes a time in $\theta(n^2)$ to sort *n* items.

The above argument can be simplified, obviating the need to introduce explicit constants such as *b* and *c*, once we are comfortable with the notion of a barometer instruction. Here, it is natural to take the innermost test "if $T[j] < minx$" as a barometer and count the exact number of times it is executed. This is a good measure of the total running time of the algorithm because none of the loops can be executed zero times (in which case loop control could have been more time consuming than our barometer). The number of times that the test is executed is easily seen to be

$$\begin{aligned}\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 1 &= \sum_{i=1}^{n-1}(n-i) \\ &= \sum_{k=1}^{n-1} k = n(n-1)/2.\end{aligned}$$

Thus the number of times the barometer instruction is executed is in $\theta(n^2)$, which automatically gives the running time of the algorithm itself.

Insertion Sort

Let's consider one more sorting technique called Insertion Sort for analysis. The procedure for insertion sorting is as shown below.

```
procedure insert(T[1..n])
    for i ← 2 to n do
        x ← T[i]; j ← i − 1
        while j > 0 and x < T[j] do T[j + 1] ← T[j]
                                    j ← j − 1
        T[j + 1] ← x
```

Unlike selection sorting, the time taken to sort n items by insertion depends significantly on the original order of the elements. Here, we analyze this algorithm in the worst case. To analyze the running time of this algorithm, we choose as barometer the number of times the **while** loop condition ($j > 0$ and $x < T[j]$) is tested.

Suppose for a moment that i is fixed. Let $x = T[i]$, as in the algorithm. The worst case arises when x is less than $T[j]$ for every j between 1 and $i - 1$, since in this case we have to compare x to $T[i-1]$, $T[i-2]$,..., $T[1]$ before we leave the **while** loop because $j = 0$. Thus the **while** loop test is performed i times in the worst case. This worst case happens for every value of i from 2 to n when the array is initially sorted into descending order. The barometer test is thus performed $\sum_{i=2}^{n} i = n(n+1)/2 - 1$ times in total, which is in $\theta(n^2)$. This shows that insertion sort also takes a time in $\theta(n^2)$ to sort n items in the worst case.

3.4 SUMMARY

In this unit, we discussed the techniques to analyze the complexity and performance of algorithms. We analyzed the most frequently encountered control structures such as sequencing, while loops, for loops and recursive calls to study the complexity and performance of algorithms. We learnt that the use of a barometer is a handy tool to simplify the analysis of many algorithms, but this technique should be used with care. Additional examples are considered to analyze the algorithms involving loops, recursion, and the use of barometers.

3.5 KEYWORDS

1) Control structures
2) Sequencing
3) For loops
4) While loops
5) Recursive calls
6) Barometer
7) Insertion sort
8) Selection sort
9) Time complexity

3.6 QUESTIONS FOR SELF STUDY

1) Explain various loop structures to be considered for analyzing algorithms.
2) How do you analyze recursive algorithms? Explain with an example.
3) How do you determine an upper bound on the running time of binary search algorithm? Explain.
4) What is a barometer instruction? How is it useful to analyze an algorithm? Explain with an example.

3.7 EXCERCISES

1) Analyze while loops considering Binary search algorithm
2) Analyze the time complexity of selection sort algorithm.
3) Analyze the time complexity of insertion sort algorithm.

3.8 REFERENCES

4) Fundamentals of Algorithmics: Gilles Brassard and Paul Bratley, Prentice Hall Englewood Cliffs, New Jersey 07632.

5) Sartaj Sahni, 2000, Data structures, Algorithms and Applications in C++, McGraw Hill International Edition.

6) Goodman And Hedetniemi, 1987, Introduction to the Design and Analysis of Algorithms, Mcgraw Hill International Editions.

UNIT- 4

AVERAGE CASE ANALYSIS, AMORTIZED ANALYSIS, SOLVING RECURRENCES

STRUCTURE

4.0 OBJECTIVES

After studying this unit you should be able to

- Analyze average case analysis of algorithms.
- Determine the time complexity of algorithms using amortization scheme.
- Solve certain class of recurrences using characteristic equation.

4.1 AVERAGE-CASE ANALYSIS

We saw that insertion sort takes quadratic time in the *worst* case. On the other hand, it is easy to show that it succeeds in linear time in the *best* case. It is natural to wonder about its efficiency *on the average*. For the question to make sense, we must be precise about the meaning of "on the average". This requires us to assume an a priori probability distribution on the instances that our algorithm may be asked to solve. The conclusion of an average-case analysis may depend crucially on this assumption, and such analysis may be misleading if in fact our assumption turns out not to correspond with the reality of the application that uses the algorithm. Most of the time, average-case analyses are performed under the more or less realistic assumption that all instances of any given size are equally likely. For sorting problems, it is simpler to assume also that all the elements to be sorted are distinct.

Suppose we have n distinct elements to sort by insertion and all $n!$ permutations of these elements are equally likely. To determine the time taken on average by the algorithm, we could add the times required to sort each of the possible permutations, and then divide by $n!$ the answer thus obtained. An alternative approach, easier in this case, is to analyze directly the time required by the algorithm. For any i, $2 \leq i \leq n$, consider the subarray $T[1..i]$. The *partial rank* of $T[i]$ is defined as the position it would occupy if the subarray were sorted. For example, the partial rank of $T[4]$ in [3,6,2,5,1,7,4] is 3 because $T[1..4]$ once sorted is [2,3,5,6]. Clearly, the partial rank of $T[i]$ does not depend on the order of the elements in subarray $T[1..i\text{-}1]$. It is easy to show that if all $n!$ permutations of $T[1..n]$ are equally likely, then the partial rank of $T[i]$ is equally likely to take any value between 1 and i, independently for all values of i.

Suppose now that i is fixed, $2 \leq i \leq n$, and that we are about to enter the **while** loop. Inspection of the algorithm shows that subarray $T[1..i\text{-}1]$contains the same elements as before the algorithm was called, although they are now in sorted order, and $T[i]$ still has its original value since it has not yet been moved. Therefore, the partial rank of $T[i]$ is equally likely to be any value between 1 and i. Let k be this partial rank. We choose again as barometer the number of times the **while** loop condition ($j>0$ and $x<T[j]$) is tested. By definition of partial rank, and since $T[1..i-1]$ is sorted, this test is performed exactly $i-k+1$ times. Because each value of k between 1 and i has probability $1/i$ of occurring, the average number of times the barometer test is performed for any given value of i is

$$c_i = \frac{1}{i} \sum_{k=1}^{i} (i-k+1) = \frac{i+1}{2}$$

These events are independent for different values of i. The total average number of times the barometer test is performed when sorting n items is therefore

$$\sum_{i=2}^{n} c_i = \sum_{i=2}^{n} \frac{i+1}{2} = \frac{(n-1)(n+4)}{4}$$

which is in $\theta(n^2)$. We conclude that insertion sorting makes on the average about half as many comparisons as in the worst case, but this number is still quadratic.

4.2 AMORTIZED ANALYSIS

Worst-case analysis is sometimes overly pessimistic. Consider for instance a process P that has side effects, which means that P modifies the value of global variables. As a result of side effects, two successive identical calls on P could take a substantially different amount of time. The easy thing to do when analyzing an algorithm that uses P as subalgorithm would be to analyze P in the worst case, and assume the worst happens each time P is called. This approach yields a correct answer assuming we are satisfied with an analysis in O notation but the answer could be pessimistic. Consider for instance the following loop.

```
for $i \leftarrow 1$ to $n$ do $P$
```

If P takes a time in $\theta(\log n)$ in the worst case, it is correct to conclude that the loop takes a time in $O(n \log n)$, but it may be that it is much faster *even in the worst case.* This could happen if P cannot take a long time ($\Omega(\log n)$) unless it has been called many times previously, each time at small cost. It could be for instance that P takes constant time on the average, in which case the entire loop would be performed in linear time.

Rather than taking the average over all possible inputs, which requires an assumption on the probability distribution of instances, we take the average over *successive* calls. Here the times taken by the various calls are highly dependent. To prevent confusion, we shall say in this context that each call on P takes *amortized* constant time rather than saying that it takes constant time on the average.

Saying that a process takes amortized constant time means that there exists a constant c such that for any positive n and any sequence of n calls on the process, the total time for those calls is bounded above by cn. Therefore, excessive time is allowed for one call only if very short times have been registered previously, *not* merely if further calls would go quickly. Indeed, if a call were allowed to be expensive on the ground that it prepares for much quicker later calls, the expense would be wasted should that call be the final one.

Consider for instance the time needed to get a cup of coffee in a common coffee room. Most of the time, you simply walk into the room, grab the pot, and pour coffee into your cup. Perhaps you spill a few drops on the table. Once in a while, however, you have the bad luck to find the pot empty, and you must start a fresh brew, which is considerably more time-consuming. While you are at it, you may as well clean up the table. Thus, the algorithm for getting a cup of coffee takes substantial time in the worst case, yet it is quick in the amortized sense because a long time is required only after several cups have been obtained quickly. (For this analogy to work properly, we must assume somewhat unrealistically that the pot is full when the first person walks in; otherwise the very first cup would consume too much time.)

A classic example of this behavior in computer science concerns storage allocation with occasional need for "garbage collection". A simpler example concerns updating a binary counter. Suppose we wish to manage the counter as an array of bits representing the value of the counter in binary notation: array $C[1..m]$ represents $\sum_{j=1}^{m} 2^{m-j}\, C[j]$. For instance, array [0,1,1,0,1,1] represents 27. Since such a counter can only count up to $2^m - 1$, we shall assume that we are happy to count modulo 2^m. Here is the algorithm for adding 1 to the counter.

```
procedure count(C[1..m])
    {This procedure assumes $m \geq 1$
     and $C[j] \in \{0,1\}$ for each $1 \leq j \leq m$}
    $j \leftarrow m + 1$
    repeat
        $j \leftarrow j - 1$
        $C[j] \leftarrow 1 - C[j]$
    until $C[j] = 1$ or $j = 1$
```

Called on our example [0,1,1,0,1,1], the array becomes [0,1,1,0,1,0] the first time round the loop, [0,1,1,0,0,0] the second time, and [0,1,1,1,0,0] the third time (which indeed represents the value 28 in binary); the loop then terminates with $j = 4$ since $C[4]$ is now equal to 1. Clearly, the algorithm's worst case occurs when $C[j] = 1$ for all j, in which case it goes round the loop m times. Therefore, n calls on *count* starting from an all-zero array take total time in $O(nm)$. But do they take a time in $\theta(nm)$? The answer is negative, as we are about to show that *count* takes constant amortized time. This implies that our n calls on *count* collectively take a time in $\theta(n)$, with a hidden constant that does not depend on m. In particular, counting from 0 to $n = 2^m - 1$ can be achieved in a time linear in n, whereas careless worst-case analysis of *count* would yield the correct but pessimistic conclusion that it takes a time in $O(n \log n)$.

There are two main techniques to establish amortized analysis results: the potential function approach and the accounting trick. Both techniques apply best to analyze the number of times a barometer instruction is executed.

Potential Functions

Suppose the process to be analyzed modifies a database and its efficiency each time it is called depends on the current state of that database. We associate with the database a notion of "cleanliness", known as the *potential function* of the database. Calls on the process are allowed to take more time than average provided they clean up the database. Conversely, quick calls are allowed to mess it up. This is precisely what happens in the coffee room! The analogy holds even further: the faster you fill up your cup, the more likely you will spill coffee, which in turn means that it will take longer when the time comes to clean up. Similarly, the faster the process goes when it goes fast, the more it messes up the database, which in turn requires more time when cleaning up becomes necessary.

Formally, we introduce an integer-valued potential function Φ of the state of the database. Larger values of Φ correspond to dirtier states. Let ϕ_0 be the value of Φ on the initial state; it represents our standard of cleanliness. Let ϕ_i be the value of Φ on the database after the i^{th} call on the process, and let t_i be the time needed by that call (or the number of times the barometer instruction is performed). We define the *amortized time* taken by that call as

$$\hat{t}_i = t_i + \phi_i - \phi_{i-1}.$$

Thus, $\hat{t}_i$ is the actual time required to carry out the i^{th} call on the process plus the increase in potential caused by that call. It is sometimes better to think of it as the actual time minus the decrease in potential, as this shows that operations that clean up the database will be allowed to run longer without incurring a penalty in terms of their amortized time.

Let T_n denote the total time required for the first n calls on the process, and denote the total *amortized* time by $\hat{T}_n$.

$$\begin{aligned}\hat{T}_n &= \sum_{i=1}^{n} \hat{t}_i = \sum_{i=1}^{n}(t_i + \phi_i - \phi_{i-1}) \\ &= \sum_{i=1}^{n} t_i + \sum_{i=1}^{n} \phi_i - \sum_{i=1}^{n} \phi_{i-1} \\ &= T_n + \phi_n + \phi_{n-1} + \cdots + \phi_1 \\ &\qquad - \phi_{n-1} - \cdots - \phi_1 - \phi_0 \\ &= T_n + \phi_n - \phi_0\end{aligned}$$

Therefore

$$T_n = \hat{T}_n - (\phi_n - \phi_0).$$

The significance of this is that $T_n \le \hat{T}_n$ holds for all n provided ϕ_n never becomes smaller than ϕ_0. In other words, the total amortized time is always an upper bound on the total actual time needed to perform a sequence of operations, as long as the database is never allowed to become "cleaner" than it was initially. This approach is interesting when the actual time varies significantly from one call to the next, whereas the amortized time is nearly invariant. For instance, a sequence of operations takes linear time when the amortized time per operation is constant, regardless of the actual time required for each operation. The challenge in applying this technique is to figure out the proper potential function. We illustrate this with our example of the binary counter. A call on *count* is increasingly expensive as the rightmost zero in the counter is farther to the left. Therefore the potential function that immediately comes to mind would be *m* minus the largest *j* such that $C[j]=0$. It turns out, however, that this choice of potential function is not appropriate because a single operation can mess up the counter arbitrarily (adding 1 to the counter representing $2^k -2$ causes this potential function to jump from 0 to k). Fortunately, a simpler potential function works well: define $\Phi(C)$as the number of bits equal to 1 in *C*. Clearly, our condition that the potential never be allowed to decrease below the initial potential holds since the initial potential is zero.

What is the amortized cost of adding 1 to the counter, in terms of the number of times we go round the loop? There are three cases to consider.

- If the counter represents an even integer, we go round the loop once only as we flip the rightmost bit from 0 to 1. As a result, there is one more bit set to 1 than there was before. Therefore, the actual cost is 1 trip round the loop, and the increase in potential is also 1. By definition, the amortized cost of the operation is $1 + 1 = 2$.
- If all the bits in the counter are equal to 1, we go round the loop *m* times, flipping all those bits to 0. As a result, the potential drops from m to 0. Therefore, the amortized cost is $m - m = 0$.
- In all other cases, each time we go round the loop we decrease the potential by 1 since we flip a bit from 1 to 0, except for the last trip round the loop when we increase the potential by 1 since we flip a bit from 0 to 1. Thus, if we go round the loop *k* times, we decrease the potential $k - 1$ times and we increase it once, for a net decrease of $k - 2$. Therefore, the amortized cost is $k - (k - 2) = 2$.

In conclusion, the amortized cost of adding 1 to a binary counter is always exactly equivalent to going round the loop twice, except that it costs nothing when the counter cycles back to zero. Since the actual cost of a sequence of operations is never more than the amortized cost, this proves that the total number of times we go round the loop when incrementing a counter n times in succession is at most $2n$ provided the counter was initially set to zero.

The Accounting Trick

This technique can be thought of as a restatement of the potential function approach, yet it is easier to apply in some contexts. Suppose you have already guessed an upper bound T on the time spent in the amortized sense whenever process P is called, and you wish to prove that your intuition was correct (T may depend on various parameters, such as the instance size). To use the accounting trick, you must set up a virtual bank account, which initially contains zero tokens. Each time P is called, an allowance of T tokens is deposited in the account; each time the barometer instruction is executed, you must pay for it by spending one token from the account. The golden rule is never to allow the account to become overdrawn. This insures that long operations are permitted only if sufficiently many quick ones have already taken place. Therefore, it suffices to show that the golden rule is obeyed to conclude that the actual time taken by any sequence of operations never exceeds its amortized time, and in particular any sequence of s operations takes a time that is at most Ts.

To analyze our example of a binary counter, we allocate two tokens for each call on *count* (this is our initial guess) and we spend one token each time *count* goes round its loop. The key insight again concerns the number of bits set to 1 in the counter. We leave it for the reader to verify that each call on *count* increases (decreases) the amount available in the bank account precisely by the increase (decrease) it causes in the number of bits set to 1 in the counter (unless the counter cycles back to zero, in which case less tokens are spent). In other words, if there were i bits set to 1 in the counter before the call and $j > 0$ bits afterwards, the number of tokens available in the bank account once the call is completed has increased by $j - i$ (counting a negative increase as a decrease). Consequently, the number of tokens in the account is always exactly equal to the number of bits currently set to 1 in the counter (unless the counter has cycled, in which case there are more tokens in the account). This proves that the account is never overdrawn since the number of bits set to 1 cannot be negative, and therefore each call on *count* costs at most two tokens in the amortized sense.

4.3 SOLVING RECURRENCES

The indispensable last step when analyzing an algorithm is often to solve a recurrence equation. With a little experience and intuition most recurrences can be solved by intelligent guesswork. However, there exists a powerful technique that can be used to solve certain classes of recurrence almost automatically. This is the main topic of this section: the technique of the *characteristic equation*.

Intelligent Guesswork

This approach generally proceeds in four stages: calculate the first few values of the recurrence, look for regularity, guess a suitable general form, and finally prove by mathematical induction (perhaps constructive induction) that this form is correct. Consider the following recurrence.

$$T(n)=\begin{cases}0 & \text{if } n=0\\ 3T(n\div 2)+n & \text{otherwise}\end{cases} \tag{4.4}$$

One of the first lessons experience will teach you if you try solving recurrences is that discontinuous functions such as the floor function (implicit in $n \div 2$) are hard to analyze. Our first step is to replace $n \div 2$ with the better-behaved "$n/2$" with a suitable restriction on the set of values of n that we consider initially. It is tempting to restrict n to being even since in that case $n \div 2 = n/2$, but recursively dividing an even number by 2 may produce an odd number larger than 1. Therefore, it is a better idea to restrict n to being an exact power of 2. Once this special case is handled, the general case follows painlessly in asymptotic notation.

First, we tabulate the value of the recurrence on the first few powers of 2.

n	1	2	4	8	16	32
$T(n)$	1	5	19	65	211	665

Each term in this table but the first is computed from the previous term. For instance, $T(16) = 3 \times T(8) + 16 = 3 \times 65 + 16 = 211$. But is this table useful? There is certainly no obvious pattern in this sequence! What regularity is there to look for? The solution becomes apparent if we keep more "history" about the value of $T(n)$. Instead of writing $T(2) = 5$, it is more useful to write $T(2) = 3 \times 1 + 2$.

Then,

$$T(4) = 3 \times T(2) + 4 = 3 \times (3 \times 1 + 2) + 4 = 3^2 \times 1 + 3 \times 2 + 4.$$

We continue in this way, writing n as an explicit power of 2.

n	$T(n)$
1	1
2	$3 \times 1 + 2$
2^2	$3^2 \times 1 + 3 \times 2 + 2^2$
2^3	$3^3 \times 1 + 3^2 \times 2 + 3 \times 2^2 + 2^3$
2^4	$3^4 \times 1 + 3^3 \times 2 + 3^2 \times 2^2 + 3 \times 2^3 + 2^4$
2^5	$3^5 \times 1 + 3^4 \times 2 + 3^3 \times 2^2 + 3^2 \times 2^3 + 3 \times 2^4 + 2^5$

The pattern is now obvious.

$$\begin{aligned} T(2^k) &= 3^k 2^0 + 3^{k-1} 2^1 + 3^{k-2} 2^2 + \cdots 1\, 2^{k-1} + 3^0 2^k \\ &= \sum_{i=0}^{k} 3^{k-i} 2^i = 3^k \sum_{i=0}^{k} (2/3)^i \\ &= 3^k \times (1 - (2/3)^{k+1}) / (1 - 2/3) \\ &= 3^{k+1} - 2^{k+1} \end{aligned} \tag{4.5}$$

It is easy to check this formula against our earlier tabulation. By induction (not *mathematical* induction), we are now convinced that the above equation is correct.

With hindsight, the Equation (4.5) could have been guessed with just a little more intuition. For this it would have been enough to tabulate the value of $T(n) + in$ for small values of i, such as $-2 \leq i \leq 2$.

n	1	2	4	8	16	32
$T(n)-2n$	-1	1	11	49	179	601
$T(n)-n$	0	3	15	57	195	633
$T(n)$	1	5	19	65	211	665
$T(n)+n$	2	7	23	73	227	697
$T(n)+2n$	3	9	27	81	243	729

This time, it is immediately apparent that $T(n)+2n$ is an exact power of 3, from which the Equation (4.5) is readily derived.

What happens when n is not a power of 2? Solving recurrence 4.4 exactly is rather difficult. Fortunately, this is unnecessary if we are happy to obtain the answer in asymptotic notation. For this, it is convenient to rewrite Equation 4.5 in terms of $T(n)$ rather than in terms of $T(2^k)$. Since $n = 2k$ it follows that $k = \lg n$.
Therefore

$$T(n) = T(2^{\lg n}) = 3^{1+\lg n} - 2^{1+\lg n}$$

Using the fact that $3^{\lg n} = n^{\lg 3}$ it follows that $T(n) = 3n^{\lg 3} - 2n$ (4.6)

when n is a power of 2. Using conditional asymptotic notation, we conclude that $T(n) \in \Theta(n^{\lg 3} \mid n$ is a power of 2). Since $T(n)$ is a nondecreasing function (a fact easily proven by mathematical induction) and $n^{\lg 3}$ is a smooth function and $T(n) \in \Theta(n^{\lg 3})$ unconditionally.

Homogeneous Recurrences

We begin our study of the *technique of the characteristic equation* with the resolution of homogeneous linear recurrences with constant coefficients, that is recurrences of the form

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0 \tag{4.7}$$

where the t_i are the values we are looking for. In addition to Equation 4.7, the values of t_i on k values of i (usually $0 \le i \le k - 1$ or $1 \le i \le k$) are needed to determine the sequence. These *initial conditions* will be considered later. Until then, Equation 4.7 typically has infinitely many solutions. This recurrence is

- *linear* because it does not contain terms of the form $t_{n-i}t_{n-j}$, t^2_{n-i}, and so on;
- *homogeneous* because the linear combination of the t_{n-i} is equal to zero; and
- *with constant coefficients* because the a_i are constants.
- Consider for instance our now familiar recurrence for the Fibonacci sequence.

Consider for instance our now familiar recurrence for the Fibonacci sequence.

$$f_n = f_{n-1} + f_{n-2}$$

This recurrence easily fits the mould of Equation 4.7 after obvious rewriting.

$$f_n - f_{n-1} - f_{n-2} = 0$$

Therefore, the Fibonacci sequence corresponds to a homogeneous linear recurrence with constant coefficients with $k = 2$, $a_0 = 1$ and $a_1, = a_2 = -1$.

Before we even start to look for solutions to Equation 4.7, it is interesting to note that any linear combination of solutions is itself a solution. In other words, if f_n, and g_n satisfy Equation 4.7, $\sum_{i=0}^{k} a_i f_{n-i} = 0$ and similarly for g_n, and if we set $t_n = cf_n + dg_n$ for arbitrary constants c and d, then t_n, is also a solution to Equation 4.7. This is true because

$$\begin{aligned}
&a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} \\
&= a_0(cf_n + dg_n) + a_1(cf_{n-1} + dg_{n-1}) + \cdots + a_k(cf_{n-k} + dg_{n-k}) \\
&= c\,(a_0 f_n + a_1 f_{n-1} + \cdots + a_k f_{n-k}) + d\,(a_0 g_n + a_1 g_{n-1} + \cdots + a_k g_{n-k}) \\
&= c \times 0 + d \times 0 = 0.
\end{aligned}$$

This rule generalizes to linear combinations of any number of solutions.

Trying to solve a few easy examples of recurrences of the form of Equation 4.7 (*not* the Fibonacci sequence) by intelligent guesswork suggests looking for solutions of the form

$$t_n = x^n$$

where x is a constant as yet unknown. If we try this guessed solution in Equation 4.7, we obtain

$$a_0 x^n + a_1 x^{n-1} + \cdots + a_k x^{n-k} = 0.$$

This equation is satisfied if x = 0, a trivial solution of no interest. Otherwise, the equation is satisfied if and only if

$$a_0 x^k + a_1 x^{k-1} + \cdots + a_k = 0.$$

This equation of degree k in x is called the *characteristic equation* of the recurrence 4.7 and

$$p(x) = a_0 x^k + a_1 x^{k-1} + \cdots + a_k$$

is called its *characteristic polynomial.*

Recall that the fundamental theorem of algebra states that any polynomial $p(x)$ of degree k has exactly k roots (not necessarily distinct), which means that it can be factorized as a product of k monomials

$$p(x) = \prod_{i=1}^{k} (x - r_i)$$

where the r_i may be complex numbers. Moreover, these r_i are the only solutions of the equation $p(x) = 0$.

Consider any root r_i of the characteristic polynomial. Since $p(r_i) = 0$ it follows that $x = r_i$ is a solution to the characteristic equation and therefore r_i^n is a solution to the recurrence. Since any linear combination of solutions is also a solution, we conclude that

$$t_n = \sum_{i=1}^{k} c_i r_i^n \tag{4.8}$$

satisfies the recurrence for any choice of constants $C_1, C_2, \ldots, C_k$. The remarkable fact, which we do not prove here, is that Equation 4.7 has *only* solutions of this form *provided all the r_i are distinct.* In this case, the k constants can be determined from k initial conditions by solving a system of k linear equations in k unknowns.

Example: (Fibonacci) Consider the recurrence

$$f_n = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1 \\ f_{n-1} + f_{n-2} & \text{otherwise} \end{cases}$$

First we rewrite this recurrence to fit the mould of Equation 4.7.

$$f_n - f_{n-1} - f_{n-2} = 0$$

The characteristic polynomial is

$$x^2 - x - 1$$

whose roots are

$$r_1 = \frac{1+\sqrt{5}}{2} \text{ and } r_2 = \frac{1-\sqrt{5}}{2}.$$

The general solution is therefore of the form

$$f_n = c_1 r_1^n + c_2 r_2^n. \tag{4.9}$$

It remains to use the initial conditions to determine the constants c_1 and c_2. When $n = 0$, Equation 4.9 yields $f_0 = c_1 + c_2$. But we know that $f_0 = 0$. Therefore, $c_1 + c_2 = 0$. Similarly, when $n = 1$, Equation 4.9 together with the second initial condition tell us that $f_1 = c_1 r_1 + c_2 r_2 = 1$. Remembering that the values of r_1 and r_2 are known, this gives us two linear equations in the two unknowns c_1 and c_2.

$$\begin{array}{rcrcl} c_1 & + & c_2 & = & 0 \\ r_1 c_1 & + & r_2 c_2 & = & 1 \end{array}$$

Solving these equations, we obtain

$$c_1 = \frac{1}{\sqrt{5}} \text{ and } c_2 = -\frac{1}{\sqrt{5}}.$$

Thus

$$f_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right],$$

which is de Moivre's famous formula for the Fibonacci sequence. Notice how much easier the technique of the characteristic equation is than the approach by constructive induction. It is also more precise since all we were able to discover with constructive induction was that "f_n grows exponentially in a number close to ϕ"; now we have an exact formula.

4.4 SUMMARY

In this unit, average case analysis of sorting algorithm is analyzed. We observed that insertion sorting makes on the average about half as many comparisons as in the worst case and this number is quadratic. The two main techniques to establish amortized analysis results: the potential function approach and the accounting trick are discussed.

Both techniques apply best to analyze the number of times a barometer instruction is executed. The indispensable step when analyzing an algorithm is often to solve a recurrence equation. With a little experience and intuition most recurrences can be solved by intelligent guesswork. The characteristic equation is a powerful technique that can be used to solve certain classes of recurrence almost automatically.

4.5 KEYWORDS

1) Average Case Analysis
2) Amortized Analysis
3) Recurrence
4) Accounting trick
5) Characteristic equation
6) Characteristic polynomial

4.6 QUESTIONS FOR SELF STUDY

1) Explain average case analysis of algorithms considering suitable examples.
2) What is meant by amortized analysis? Explain with an example.
3) Explain the potential function approach to analyze algorithms.
4) How to solve certain class of recurrences using characteristic equation?
5) Explain intelligent guesswork approach to solve recurrences.
6) What are homogeneous linear recurrences? Explain.

4.7 EXCERCISES

1) Determine the complexity of sorting algorithms using amortization scheme.
2) Obtain the general solution for a Fibonacci sequence using characteristic equation approach.

4.8 REFERENCES

1) Fundamentals of Algorithmics: Gilles Brassard and Paul Bratley, Prentice Hall Englewood Cliffs, New Jersey 07632.

2) Sartaj Sahni, 2000, Data structures, Algorithms and Applications in C++, McGraw Hill International Edition.

3) Goodman And Hedetniemi, 1987, Introduction to the Design and Analysis of Algorithms, Mcgraw Hill International Editions.

UNIT – 5

THE GREEDY METHOD

STRUCTURE

5.0 OBJECTIVES

After studying this unit you should be able to

- Elucidate the optimal solution to a problem.
- Explain the greedy method to construct an optimal solution to a problem.
- Design greedy algorithm for a problem.
- State the complexity of an algorithm.

5.1 INTRODUCTION

Greedy method is a method of choosing a subset of a dataset as the solution set that result in some profit. Consider a problem having *n* inputs. We are required to obtain a solution which is a series of subsets that satisfy some constraints or conditions. Any subset, which satisfies these constraints, is called a feasible solution. It is required to obtain a feasible solution that maximizes or minimizes an objective function. This feasible solution finally obtained is called optimal solution. The concept is called Greedy because at each stage we choose the "best" available solution i.e., we are "greedy" about the output.

In greedy strategy, one can devise an algorithm that works in stages, considering one input at a time and at each stage, a decision is taken on whether the data chosen results with an optimal solution or not. If the inclusion of a particular data, results with an optimal solution, then the data is added into the partial solution set. On the other hand, if the inclusion of that data results with infeasible solution then the data is eliminated from the solution set.

Stated in simple terms, the greedy algorithm suggests that we should be "greedy" about the intermediate solution i.e., if at any intermediate stage *k* different options are available to us, choose an option which "maximizes" the output.

The general algorithm for the greedy method is

- Choose an element *e* belonging to the input dataset D.
- Check whether *e* can be included into the solution set S, If yes solution set is Union(S, *e*)
- Continue until s is filled up or D is exhausted whichever is earlier.

Sometimes the problem under greedy strategy could be to select a subset out of given *n* inputs, and sometimes it could be to reorder the n data in some optimal sequence. The control abstraction for the subset paradigm is as follows:

```
Algorithm: GREEDY
Input: A, a set containing n inputs
Output: S, the solution subset
Method:
S = { }
    For i = 1 to n do
            x = SELECT (A(i))
            If(FEASIBLE (S, x))
            S = UNION(S, x)
    If end
    For end
return (S)
Algorithm ends
```

SELECT selects the best possible solution (or input) from the available inputs and includes it in the solution. If it is feasible (in some cases, the constraints may not allow us to include it in the solution, even if it produces the best results), then it is appended to the partially built solution. The whole process is repeated till all the options are exhausted.

In the next few sections, we look into some of the applications of the Greedy method.

5.2 THE PROBLEM OF OPTIMAL STORAGE ON TAPES

We know, when large quantities of data or program need to be backed up, the best way is to store them on tapes. For our discussion, let us presume that they are programs. However, they can be any other data also. (Though with the advent of high capacity CDs, the importance of tapes as storage media can be said to have reduced, but they have not vanished altogether).

Consider n programs that are to be stored on a tape of length L. Each program P_i is of length l_i where i lies between 1 and n. All programs can be stored on the tape iff the sum of the lengths of the programs is at most L. It is assumed that, whenever a program is to be retrieved the tape is initially positioned at the start end. That is, since the tape is a sequential device, to access the i^{th} program, we will have to go through all the previous (i-1) programs before we get to i, i.e., to go to a useful program (of primary interest), we have to go through a number of useless (for the time being) programs. Since we do not know in what order the programs are going to be retrieved, after accessing each program, we come back to the initial position.

Let t_i be the time required for retrieving program ii where programs are stored in the order $\{P_1, P_2, P_3, \ldots, P_n\}$. Thus to get to the i^{th} program, the required time is proportional to

$$t_i = \sum_{k=1}^{i} l_k$$

If all programs are accessed with equal probability over a period of time, then the average retrieval time (mean retrieval time)

$$\text{MRT} = \frac{1}{n}\sum_{k=1}^{n} t_k$$

To minimize this average retrieval time, we have to store the programs on the tape in some optimal sequence.

Before trying to devise the algorithm, let us look at the commonsense part of it. The first program (program no. 1) will have to be passed through, the maximum number of times, as irrespective of the program that we are interested in, we have to pass this. Similarly, the second program has to be passed through in all cases, except when we are interested in the first program. Likewise it can be concluded that the first, second, … , n^{th} programs are scanned/passed through with decreasing frequency. Hence, it is better to store the shortest program at the beginning. Due to the fact that the first program is passed/scanned, maximum number of times, if we store the smallest one, it adds the smallest possible amount to the time factor.

Similarly the second program will have to be the second smallest and so on. Precisely, the programs to be stored on the tape are to be arranged according to the increasing lengths. The smallest program which is passed/scanned through the maximum number of times, adds less over head. The largest of programs is at the end of the tape, but will hardly be passed through, except when we are actually interested in the same.

From the above discussion one can observe that the MRT can be minimized if the programs are stored in an increasing order i.e., $l_1 \leq l_2 \leq l_3 \ldots \leq l_n$. Hence the ordering defined minimizes the retrieval time. Here, the solution set obtained is not a subset of data but is the data set itself in a different sequence.

Now we take a numerical example to see whether this works.

Illustration

Assume that 3 files are given. Let the length of files A, B and C be 7, 3 and 5 units respectively. All these three files are to be stored on to a tape S in some sequence that reduces the average retrieval time.

The table shows the retrieval time for all possible orders.

Order of recording	Retrieval time	MRT
ABC	7+(7+3)+(7+3+5)=32	$\frac{32}{3} = 10.667$
ACB	7+(7+5)+(7+5+3)=34	$\frac{34}{3} = 11.333$
BAC	3+(3+7)+(3+7+5)=28	$\frac{28}{3} = 9.333$
BCA	3+(3+5)+(3+5+7)=26	$\frac{26}{3} = 8.333$
CAB	5+(5+7)+(5+7+3)=32	$\frac{32}{3} = 10.667$
CBA	5+(5+3)+(5+3+7)=28	$\frac{28}{3} = 9.333$

Though this discussion looks fairly straight forward enough not to need an algorithm, it provides us certain insights. In this case, we are trying to optimize on the time of accessing, which is nothing but the lengths of programs to be traversed. Hence, we try to optimize ón the length of the part of the scan we scan at each stage - i.e. we are "greedy" about the time spent in retrieving a file of our interest. We actually build the list in stages. At each stage of storing the programs choose the least costly of available options - i.e. the program of shortest length. Though the program is simple enough, we write a small algorithm to implement the same.

```
Algorithm: TAPE STORAGE
Input : n, the number of programs
        $l_1, l_2, l_3, \ldots, l_n$, the lengths of the programs to be stored
Output: minimum mean retrieval time yielding sequence
Method :
Sort the list in the non decreasing order of lengths
For i = 1 to n do
     Choose the ith program from the sorted list and store it on the tape
   For end
Algorithm ends
```

The procedure can also be used to append programs on more than one tape. The first tape gets the smallest set of the programs (in terms of their lengths) and so on.

Complexity

The greedy method simply requires us to store the programs in nondecreasing order of their lengths. This ordering can be done in $O(n \log n)$ time using efficient sorting algorithm like heap sort.

5.3 A THIRSTY BABY

Assume there is a thirsty, but smart, baby, who has access to a glass of water, a carton of milk, etc., a total of n different kinds of liquids. Let a_i be the amount of ounces in which the i^{th} liquid is available. Based on her experience, she also assigns certain satisfying factor, s_i, to the i^{th} liquid. If the baby needs to drink t ounces of liquid, how much of each liquid should she drink?

Let x_i, $1 \le i \le n$, be the amount of the i^{th} liquid the baby will drink. The solution for this thirsty baby problem is obtained by finding real numbers x_i, $1 \le i \le n$, that maximize $\sum_{i=1}^{n} s_i x_i$, subject to the constraints that $\sum_{i=1}^{n} x_i = t$ and for all $1 \le i \le n$, $0 \le x_i \le a_i$. We notice that if $\sum_{i=1}^{n} a_i < t$, then this instance will not be solvable.

A specification

Input: n, t, s_i, a_i, $1 \le i \le n$. n is an integer, and the rest are positive reals.

Output: If $\sum_{i=1}^{n} a_i \ge t$, output is a set of real numbers x_i, $1 \le i \le n$, such that $\sum_{i=1}^{n} s_i x_i$, is maximum, $\sum_{i=1}^{n} x_i = t$, and for all $1 \le i \le n$, $0 \le x_i \le a_i$.

In this case, the constraints are $\sum_{i=1}^{n} x_i = t$, and for all $1 \le i \le n$, $0 \le x_i \le a_i$, and the optimization function is $\sum_{i=1}^{n} s_i x_i$.

Every set of xi that satisfies the constraints is a feasible solution, and if it further maximizes $\sum_{i=1}^{n} s_i x_i$ is an optimal solution.

5.4 LOADING PROBLEM

A large ship is to be loaded with containers of cargos. Different containers, although of equal size, will have different weights. Let w_i be the weight of the i^{th} container, $1 \le i \le n$, and the capacity of the ship is c, we want to find out how could we load the ship with the maximum number of containers.

Let $x_i \in \{0, 1\}$. If $x_i = 1$, we will load the i^{th} container, otherwise, we will not load it. We wish to assign values to x_i's is such that $\sum_{i=1}^{n} x_i \le C$, and $x_i \in \{0, 1\}$. The optimization function is $\sum_{i=1}^{n} x_i$.

5.5 CHANGE MAKING

A child buys a candy bar at less than one buck and gives a $1 bill to the cashier, who wants to make a change using the fewest number of coins. The cashier constructs the change in stages, in each of which a coin is added to the change.

The greedy criterion is as follows: At each stage, increase the total amount as much as possible. To ensure the feasibility, such amount in no stage should exceed the desired change. For example, if the desired change is 67 cents. The first two stages will add in two quarters. The next one adds a dime, and following one will add a nickel, and the last two will finish off with two pennies.

5.6 MACHINE SCHEDULING

We are given an infinite supply of machines, and *n* tasks to be performed in those machines. Each task has a start time, s_i, and finish time, t_i. The period $[s_i, t_i]$ is called the processing interval of task *i*. Two tasks *i* and *j* might overlap, e.g., [1, 4] overlaps with [2, 4], but not with [4, 7].

A feasible assignment is an assignment in which no machine is given two overlapped tasks. An optimal assignment is a feasible one that uses fewest numbers of machines.

We line up tasks in nondecreasing order of s_i's, and call a machine *old*, if it has been assigned a task, otherwise, call it *new*. A greedy strategy could be the following: At each stage, if an old machine becomes available by the start time of a task, assign the task to this machine; otherwise, assign it to a new one.

Example: Given seven tasks, their start time, as well as their finish time as follow:

task	a	b	c	d	e	f	g
start	0	3	4	9	7	1	6
finish	2	7	7	11	4	5	8