CANARA ENGINEERING COLLEGE

DEPARTMENT OF INFORMATION SCIENCE

Benjanapadavu, Bantwal Taluk, Mangalore – 574219

Department of Information Science and Engineering

VISION

The Department of Information Science and Engineering strives to be a centre of learning in the field of Information Technology to produce globally competent engineers catering to the needs of the industry and society.

MISSION

- Impart technical skills in the field of Information Science & Engineering.
- Train and transform students to become technological thinkers and facilitate a quality venture which meets the industrial and societal needs.
- Encourage students to become well-rounded in their professional competencies.

PROGRAM EDUCATIONAL OBJECTIVES

1. Graduates will succeed in the field of Information Science and Engineering, professional career and higher studies.
2. Graduates will analyze the requirements of the software industries and provide novel engineering designs and efficient solutions with legal and ethical responsibility.
3. Graduates will adapt to emerging technologies, work in multidisciplinary teams with effective communication skills and leadership qualities.

PROGRAM OUTCOMES

Engineering graduates in **Information Science and Engineering**will be able to:

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and Sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** CCommunicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES

1. An ability to understand, analyze and impart the basic knowledge of Information Science and Engineering.
2. An ability to apply the programming, designing, and problem solving techniques in building/simulating the applications, solving the problems and guiding the innovative career paths to become an IT Engineer.

COURSE OBJECTIVES:

Course will enable the students to :

1 To understand the algorithms and applying the asymptotic notations to find the efficiency.

2 To utilize data structures in solving new problems.

3 To study various algorithmic design techniques.

4 To know and understand basic computability concepts and the complexity classes P, NP, and NP-Complete.

5 To Determine the time and space complexity of simple algorithms and recursively defined algorithms.

COURSE OUTCOMES (COs):

After studying the course students will be able to

1 Analyse and compare the efficiency of algorithms using asymptotic complexity.

2 Describe, apply and analyze the complexity of certain divide and conquer and decrease and conquer algorithms.

3 Describe, apply and analyze the complexity of certain greedy algorithms.

4 Describe, apply and analyze the complexity of certain dynamic programming algorithms.

5 Describe the classes P, NP, and NP-Complete and be able to prove that a certain problem is NP-Complete. Explain and apply backtracking and branch and bound techniques to deal with some hard problems.

MODULE - 1

CONTENTS

- **INTRODUCTION: WHAT IS AN ALGORITHM?**
- **ALGORITHM SPECIFICATION**
- **ANALYSIS FRAMEWORK**
- **PERFORMANCE ANALYSIS:**
    - Space Complexity
    - Time Complexity
- **ASYMPTOTIC NOTATIONS:**
    - Big-Oh Notation (O), Omega Notation (Ω), Theta Notation (Θ), Little-Oh Notation (O)
    - Mathematical Analysis of Non-Recursive and Recursive Algorithms with Examples
- **IMPORTANT PROBLEM TYPES:**
    - Sorting
    - Searching
    - String Processing
    - Graph Problems
    - Combinatorial Problems
- **FUNDAMENTAL DATA STRUCTURES:**
    - Stacks
    - Queues
    - Graphs
    - Trees
    - Sets And Dictionaries.
- **UNIVERSITY QUESTIONS.**

Objective:

Explain fundamentals of data structures and their applications essential for programming/problem solving.

Outcome:

Acquire knowledge of - Various types of data structures, operations and algorithms.

1. INTRODUCTION: WHAT IS AN ALGORITHM?

imp

1.1. Definition:

An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

In addition, all algorithms must satisfy the following criteria:

1. **Input**. Zero or more quantities are externally supplied.
2. **Output**. At least one quantity is produced.
3. **Definiteness**. Each instruction is clear and unambiguous.
4. **Finiteness**. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. **Effectiveness**. Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion3; it also must be feasible.

imp

1.2. Notion of algorithm

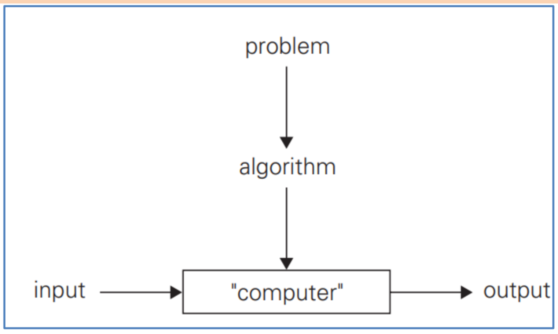


Fig. 1: Notion of Algorithm

Algorithms that are definite and effective are also called *computational procedures*. This procedure is designed to control the execution of jobs, in such a way that when no jobs are available, it does not terminate but continues in a waiting state until a new job is entered.

A *program* is the expression of an algorithm in a programming language. Sometimes words such as procedure, function, and subroutine are used synonymously for program.

The study of algorithms includes many important and active areas of research. There are four distinct areas of study one can identify:

1. **How to devise algorithms** - Creating an algorithm is an art which may never be fully automate
2. **How to validate algorithms**- Once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible legal inputs. We refer to this process as algorithm ***validation***. The purpose of the validation is to assure us that this algorithm will work correctly

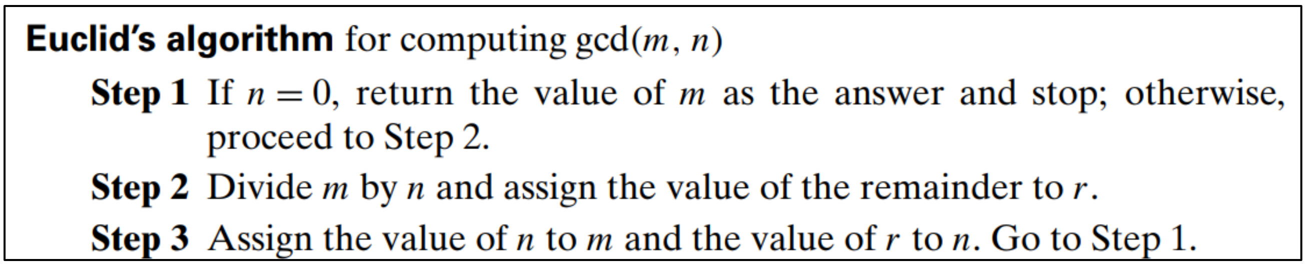
independently of the issues concerning the programming language it will eventually be written in. Once the validity of the method has been shown, a program can be written and a second phase begins. This phase is referred to as ***program proving*** or sometimes as ***program verification***.

3. **How to analyze algorithms** - This field of study is called ***analysis of algorithms***. As an algorithm is executed, it uses the computer's central processing unit (CPU)to perform operations and its memory (both immediate and auxiliary) to hold the program and data. Analysis of algorithms or performance analysis refers to the task of determining how much computing time and storage an algorithm requires.
4. **How to test a program** – Testing a program consists of two phases: debugging and profiling (or performance measurement). ***Debugging*** is the process of executing programs on sample data sets to determine whether faulty results occur and, if so, to correct them. A proof of correctness is much more valuable than a thousand tests (if that proof is correct), since it guarantees that the program will work correctly for all possible inputs. ***Profiling*** or performance measurement is the process of executing a correct program on datasets and measuring the time and space it takes to compute the results. These timing figures are useful in that they may confirm a previously done analysis and point out logical places to perform useful optimization.

Example: Recall that the greatest common divisor of two nonnegative, not-both-zero integers m and n, denoted gcd(m, n), is defined as the largest integer that divides both m and n evenly, i.e., with a remainder of zero.

gcd(m, n) = gcd(n, m mod n), where m mod n is the remainder of the division of m by n, until m mod n is equal to 0.

For example, gcd(60, 24) can be computed as follows: gcd(60, 24) = gcd(24, 12) = gcd(12, 0) = 12.



Euclid's algorithm for computing gcd(m, n)

Step 1 If $n = 0$, return the value of m as the answer and stop; otherwise, proceed to Step 2.

Step 2 Divide m by n and assign the value of the remainder to r.

Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.

Algorithm 1: Euclid's algorithm

2. ALGORITHM SPECIFICATIONS

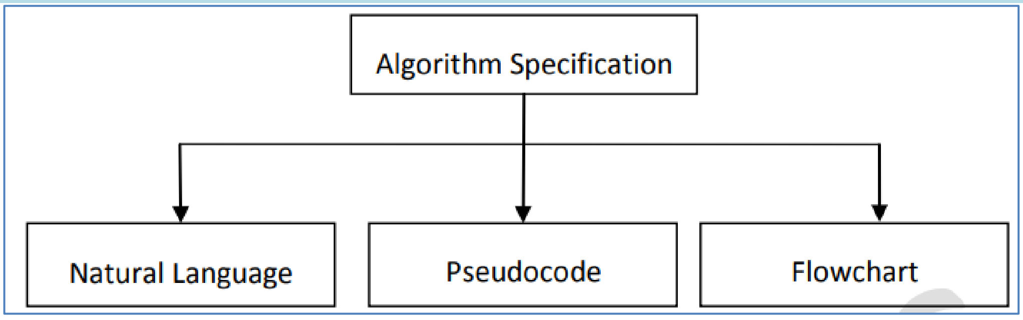


Figure 2: Algorithm Specifications

2.1. Natural Language

It is very simple and easy to specify an algorithm using natural language. But many times, specification of algorithm by using natural language is not clear and thereby we get brief specification.

Example: An algorithm to perform addition of two numbers.

Step 1: Read the first number, say a.

Step 2: Read the first number, say b.

Step 3: Add the above two numbers and store the result in c.

Step 4: Display the result from c.

a. Pseudocode

Pseudocode is a mixture of a natural language and programming language constructs. Pseudocode is usually more precise than natural language. For Assignment operation left arrow "←", for comments two slashes "//",if condition, for, while loops are used.

```
ALGORITHM  Sum(a,b)
        //Problem Description: This algorithm performs addition of two numbers
        //Input: Two integers a and b
        //Output: Addition of two integers
        c←a+b
        return c
```

Algorithm 2: Adding 2 numbers

This specification is more useful for implementation of any language.

b. Flowchart

In the earlier days of computing, the dominant method for specifying algorithms was a flowchart, this representation technique has proved to be inconvenient. Flowchart is a graphical representation of an algorithm.

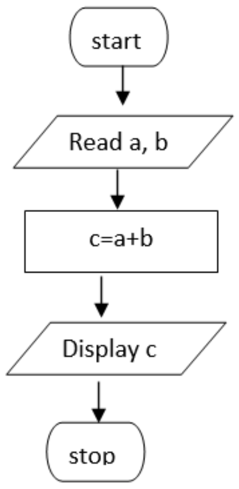


Figure 3: flowchart for adding 2 numbers

2.2. Pseudocode Conventions

Steps for writing an algorithm:

1. Comments begin with // and continue untill the end of line

   Eg: count :=count+1;//count is global ;It is initially zero.

2. Blocks are indicated with matching braces: { and } . A compound statement can be represented as a block. The body of a procedure also forms a block. Statements are delimited by ;

   ```
   Eg: for j:= 1 to n do {
           count:=count+1;
           c[i,j]:=a[i,j]+b[i,j];
           count:=count +1;
           }
   ```

3. An identifier begins with a letter. The data types of variables are not explicitly declared. The types will be clear from the context. Whether a variable is global or local to a procedure will also be evident from the context. Compound data types can be formed with records.

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

4. Assignment of values to variables is done using the assignment statement <variable>:= <expression>;

Eg: count:= count+1;

5. There are two Boolean values true and false. In order to produce these values, the logical operators and, or, and not and the relational relational operators <, <=, =, !=, >= and > are provided.

   Eg: if (j>1) then k:=i-1; else k:=n-1;

6. Elements of multidimentional arrays are accessed using [and].

   For eg: if A is a two dimentional array , the (I,j) th element of the array is denoted as A[I,j]. Array indicates start at zero.

7. The following looping statements are employed: for,while and repeat until. The while loop takes the following form.

```
While (condition) do {
                    <statement 1>
                    :
                    :
                    <statement n>
                    }
```

8. A conditional statement has the following forms:

   If < condition > then <statements>

   If <condition> then <statement 1> else <statement 2>

   Here < condition > is a Boolean expression and <statement>,<statement1>, and < statement 2> are arbitrary statements.

9. Input and output are done using the instructions read and write. No format is used to specify the size of input or output quantities.

   Eg: write (“n is even”);

10. There is only one type of procedure: ALGORITHM. An algorithm consists of a heading and a body. The heading takes the form Algorithm Name (<parameter list>)

    Example:

    The greatest common divisor(GCD) of two nonnegative integers m and n (not-both-zero), denoted gcd(m, n), is defined as the largest integer that divides both m and n evenly, i.e., with a remainder of zero.

    Euclid’s algorithm is based on applying repeatedly the equality gcd(m, n) = gcd(n, m mod n), where m mod n is the remainder of the division of m by n, until m mod n is equal to 0. Since gcd(m, 0) = m, the last value of m is also the greatest common divisor of the initial m and n. gcd(60, 24) can be computed as follows: gcd(60, 24) = gcd(24, 12) = gcd(12, 0) = 12.

Euclid's algorithm for computing gcd(m, n) in simple steps

Step 1 If n = 0, return the value of m as the answer and stop; otherwise, proceed to Step 2.

Step 2 Divide m by n and assign the value of the remainder to r.

Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.

Euclid's algorithm for computing gcd(m, n) expressed in pseudocode

```
ALGORITHM Euclid_gcd(m, n)
 //Computes gcd(m, n) by Euclid's algorithm
//Input: Two nonnegative, not-both-zero integers m and n
//Output: Greatest common divisor of m and n
while $n \neq 0$ do
        $r \leftarrow m \bmod n$
        $m \leftarrow n$
        $n \leftarrow r$
 return m
```

3. Analysis framework

3.1. Measuring an Input's Size

Let's start with the obvious observation that almost all algorithms run longer on larger inputs. For example, it takes longer to sort larger arrays, multiply larger matrices, and so on. Therefore, it is logical to investigate an algorithm's efficiency as a function of some parameter n indicating the algorithm's input size.

There are situations, of course, where the choice of a parameter indicating an input size does matter. One such example is computing the product of two $n \times n$ matrices. There are two natural measures of size for this problem. The first and more frequently used is the matrix order n. But the other natural contender is the total number of elements N in the matrices being multiplied.

3.2. Units for Measuring Running Time

The next issue concerns units for measuring an algorithm's running time. Of course, we can simply use some standard unit of time measurement—a second, or millisecond, and so on—to measure the running time of a program implementing the algorithm. There are obvious drawbacks to such an approach, however: dependence on the speed of a particular computer, dependence on the quality of a program implementing the algorithm and of the compiler used in generating the machine code, and the difficulty of clocking the actual running time of the program. Since we are after a measure of an algorithm's efficiency, we would like to have a metric that does not depend on these extraneous factors.

One possible approach is to count the number of times each of the algorithm's operations is executed. This approach is both excessively difficult and, as we shall see, usually unnecessary. The thing to do is to identify the most important operation of the algorithm, called the basic operation, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.

As a rule, it is not difficult to identify the basic operation of an algorithm: it is usually the most time-consuming operation in the algorithm's innermost loop. As a rule, it is not difficult to identify the basic operation of an algorithm: it is usually the most time-consuming operation in the algorithm's innermost loop. Here is an important application. Let c_{op} be the execution time of an algorithm's basic operation on a particular computer, and let C(n) be the number of times this operation needs to be executed for this algorithm. Then we can estimate the running time T (n) of a program implementing this algorithm on that computer by the formula $T(n) \approx c_{op}C(n)$.

3.3. Orders of Growth

Order of growth is a function that grows depends on the program. Why this emphasis on the count's order of growth for large input sizes? A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones. When we have to compute, for example, the greatest common divisor of two small numbers, it is not immediately clear how much more efficient Euclid's algorithm is compared to the other two algorithms or even why we should care which of them is faster and by how much. It is only when we have to find the greatest common divisor of two large numbers that the difference in algorithm efficiencies becomes both clear and important. For large values of n, it is the function's order of growth that counts.

3.4. Worst-Case, Best-Case, and Average-Case Efficiencies

There are many algorithms for which running time depends not only on an input size but also on the specifics of a particular input. Consider, as an example, sequential search. This is a straightforward algorithm that searches for a given item (some search key K) in a list of n elements by checking successive elements of the list until either a match with the search key is found or the list is exhausted.

```
ALGORITHM  SequentialSearch(A[0..n − 1], K)
    //Searches for a given value in a given array by sequential search
    //Input: An array A[0..n − 1] and a search key K
    //Output: The index of the first element in A that matches K
    //          or −1 if there are no matching elements
    i ← 0
    while i < n and A[i] ≠ K do
        i ← i + 1
    if i < n return i
    else return −1
```

Algorithm 3: sequential search

- ***Worst case efficiency***

Clearly, the running time of this algorithm can be quite different for the same list size n. In the worst case, when there are no matching elements or the first matching element happens to be the last one on the list, the algorithm makes the largest number of key comparisons among all possible inputs of size n:

$$\boxed{\mathbf{C_{worst}(n) = n.}}$$

The worst-case efficiency of an algorithm is its efficiency for the worst-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the longest among all possible inputs of that size.

The way to determine the worst-case efficiency of an algorithm is, in principle, quite straightforward: analyze the algorithm to see what kind of inputs yield the largest value of the basic operation's count C(n) among all possible inputs of size n and then compute this worst-case value $C_{worst}(n)$. (For sequential search, the answer was obvious. The methods for handling less trivial situations are explained in subsequent sections of this chapter.) Clearly, the worst-case analysis provides very important information about an algorithm's efficiency by bounding its running time from above.

- ***Best case efficiency***

The best-case efficiency of an algorithm is its efficiency for the best-case input of size n, which is an input (or inputs) of size n for which the algorithm runs the fastest among all possible inputs of that size.

Accordingly, we can analyze the best-case efficiency as follows. First, we determine the kind of inputs for which the count C(n) will be the smallest among all possible inputs of size n. Then we ascertain the value of C(n) on these most convenient inputs. For example, the best-case inputs for sequential search are lists of size n with their first element equal to a search key; accordingly,

$$\boxed{C_{best}(n) = 1}$$

- ***Average case efficiency***

It should be clear from our discussion, however, that neither the worst-case analysis nor its best-case counterpart yields the necessary information about an algorithm's behavior on a "typical" or "random" input. This is the information that the average-case efficiency seeks to provide. To analyze the algorithm's averagecase efficiency, we must make some assumptions about possible inputs of size n.

Let's consider again sequential search. The standard assumptions are that (a) the probability of a successful search is equal to p ($0 \le p \le 1$) and (b) the probability of the first match occurring in the ith position of the list is the same for every i. Under these assumptions—the validity of which is usually difficult to verify, their reasonableness notwithstanding—we can find the average number of key comparisons Cavg(n) as follows. In the case of a successful search, the probability of the first match occurring in the ith position of the list is p/n for every i, and the number of comparisons made by the algorithm in such a situation is obviously i. In the case of an unsuccessful search, the number of comparisons will be n with the probability of such a search being (1 − p). Therefore

$$\begin{aligned} C_{avg}(n) &= [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1-p) \\ &= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1-p) \\ &= \frac{p}{n}\frac{n(n+1)}{2} + n(1-p) = \frac{p(n+1)}{2} + n(1-p). \end{aligned}$$

4. Performance Analysis

There are many criteria upon which we can judge an algorithm. For instance:

1. Does it do what we want it to do?
2. Does it work correctly according to the original specifications of the task?
3. Is there documentation that describes how to use it and how it works?
4. Are procedures created in such a way that they perform logical subfunctions?
5. Is the code readable?

Performance evaluation can be loosely divided into two major phases:

1. A priori estimates (Performance Analysis)
2. A posteriori testing (Performance measurement)

imp

4.1. Space Complexity

The space complexity of an algorithm is the amount of memory it needs to run to completion. The space needed by each of these algorithms is seen to be the sum of the following components:

1. *A fixed part* that is independent of the characteristics of the inputs and outputs. This part typically includes the instruction space, space for simple variables and fixed-size component variables, space for constants.
2. *A variable part* that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by reference variables and the recursion stack space.

The space requirement S(P) of any algorithm P may therefore be written as S(P)= c + Sp (instance characteristics), where c is a constant.

Example - 1:

Consider the following algorithm:

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

Algorithm 4: computes the problem

Making the assumption that one word is adequate to store the values of each of a, b, c, and the result, we see that the space needed by abc is independent of the instance characteristics. Consequently, Sp(instance characteristics) = 0. Since each a, b and c carries one word each, total space required for variables is 3.

S(P) = c + Sp = 3+0 = 3 words

Example – 2:

Consider the following algorithm:

```
Algorithm Sum(a, n)
{
    $s := 0.0$;
    for $i := 1$ to $n$ do
        $s := s + a[i]$;
    return $s$;
}
```

Algorithm 5: Iterative function for sum

The problem instances are characterized by n, the number of elements to be summed. The space needed by n is one word, since it is of type integer. The space needed by a is the space needed by variables of type array of floating point numbers. This is at least n words, since a must be large enough to hold the n elements to be summed. So, we obtain $S_{sum}(n) >= (n + 3)$ (n for a[ ] , one each for n, i, and s).

imp

4.2. Time Complexity

The time complexity of an algorithm is the amount of computer time it needs to run to completion. The time T (P) taken by a program P is the sum of the compile time and the run (or execution) time. The compile time does not depend on the instance characteristics. Also, we may assume that a compiled program will be run several times without recompilation. Consequently, we concern ourselves with just the run time of a program. This run time is denoted by tp.

A program step is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics

The number of steps any program statement is assigned depends on the kind of statement. For example, comments count as zero steps; an assignmnent statement which does not involve any calls to other algorithms is counted as one step; in an iterative statement such as the for, while, and repeat-until statements, we consider the step counts only for the control part of the statement. The control parts for for and while statements have the following forms:

for $i := \langle expr \rangle$ **to** $\langle expr1 \rangle$ **do**

while $(\langle expr \rangle)$ **do**

Determining the no. of steps:

- We introduce a new variable, count, into the program.
- This is a global variable with initial value 0.
- Statements to increment count by the appropriate amount are introduced into the program.
- This is done so that each time a statement in the original program is executed, count is incremented by the step count of that statement.
- Example:

```
Algorithm Sum(a, n)
{
    s := 0.0;
    count := count + 1; // count is global; it is initially zero.
    for i := 1 to n do
    {
        count := count + 1; // For for
        s := s + a[i]; count := count + 1; // For assignment
    }
    count := count + 1; // For last time of for
    count := count + 1; // For the return
    return s;
}
```

Algorithm 6: Algorithm 4 count statement added

```
Algorithm Sum(a, n)
{
    for i := 1 to n do count := count + 2;
    count := count + 3;
}
```

Algorithm 7: Simplified version of algorithm 5.

When the statements to increment count are introduced into Algorithm 5, the result is Algorithm 6. The change in the value of count by the time this program terminates is the number of steps executed by Algorithm 5.

Since we are interested in determining only the change in the value of count, Algorithm 5 may be simplified to Algorithm 6. It is easy to see that in the for loop, the value of count will increase by a, total of 2n. If count is zero to start with, then it will be 2n + 3 on termination. So, each invocation of Sum (Algorithm 5) executes a total of $2n$ +3 Steps.

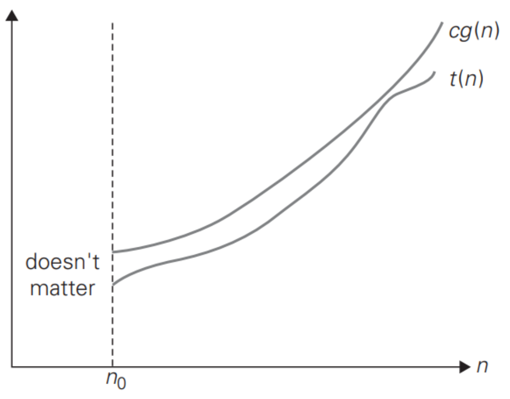
One of the instance characteristics that is frequently used in the literature is the *input size*. The input size of any instance of a problem is defined to be the number of words (or the number of elements) needed to describe that instance.

5. Asymptotic Notations and Basic Efficiency Classes

imp

5.1. O-notation

A function t (n) is said to be in O(g(n)), denoted t (n) ∈ O(g(n)), if t (n) is bounded above by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that t (n) ≤ cg(n) for all n ≥ n0.



Example

Let us formally prove one of the assertions made in the introduction: $100n + 5 \in O(n^2)$.

Indeed, $100n + 5 \leq 100n + n$ (for all $n \geq 5$) $= 101n \leq 101n^2$.

Thus, as values of the constants c and n_0 required by the definition, we can take 101 and 5, respectively. Note that the definition gives us a lot of freedom in choosing specific values for constants c and n_0.

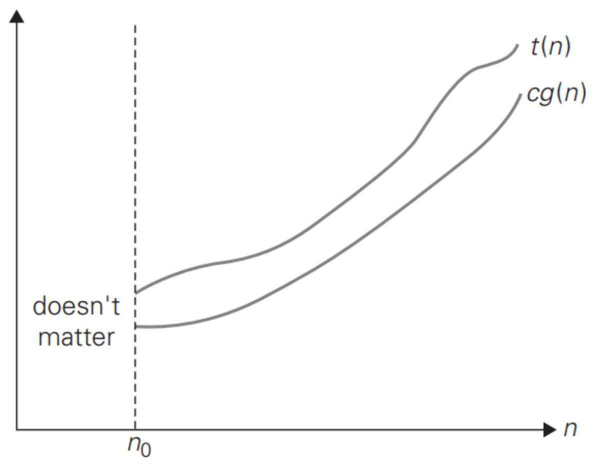
For example, we could also reason that $100n + 5 \leq 100n + 5n$ (for all $n \geq 1$) $= 105n$ to complete the proof with c = 105 and $n_0 = 1$. 54

imp

5.2. Ω-notation

A function t (n) is said to be in Ω(g(n)), denoted t (n) ∈ Ω(g(n)), if t (n) is bounded below by some positive constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n0 such that t (n) ≥ cg(n) for all n ≥ n0.

Here is an example of the formal proof that n3 ∈ Ω (n2): n3 ≥ n2 for all n ≥ 0, i.e., we can select c = 1 and n0 = 0



imp

5.3. Θ-notation

A function t (n) is said to be in Θ(g(n)), denoted t (n) ∈ Θ (g(n)), if t (n) is bounded both above and below by some positive constant multiples of g(n) for all large n, i.e., if there exist some positive constants c1 and c2 and some nonnegative integer n_0 such that $c_2g(n) \le t(n) \le c_1g(n)$ for all $n \ge n_0$.

For example, let us prove that [n(n − 1)]/2 ∈ Θ (n2). First, we prove the right inequality (the upper bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \le \frac{1}{2}n^2 \quad \text{for all } n \ge 0.$$

Second, we prove the left inequality (the lower bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \ge \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n \text{ (for all } n \ge 2) = \frac{1}{4}n^2.$$

Hence, we can select $c_2 = \frac{1}{4}$, $c_1 = \frac{1}{2}$, and $n_0 = 2$.

imp

5.4. Example problems

1. Example - 1

Let $g(n) = n^2$, what is the relation b/n n^3, $\frac{1}{2}(n)(n-1)$ & $100n+5$ using Ω notation ?

a) n^3 & n^2

$n^3 > n^2$

$\therefore n^3 \in \Omega(n^2)$

b) $\frac{1}{2}(n)(n-1)$ & n^2

$\Rightarrow \frac{n}{2}(n-1)$ & n^2

$\Rightarrow \frac{n^2}{2} - \frac{n}{2}$ & n^2

$\downarrow$ $\downarrow$

$\rightarrow$ order is n^2 order is n^2.

$\Rightarrow$ order of $\frac{1}{2}n(n-1)$ & n^2 are same

$\therefore$ $\frac{1}{2}(n)(n-1) \in \Omega(n^2)$

2. **Example - 2**

Ex. 2 - Let $f(n) = 100n + 5$ Express $f(n)$ using O.

Sol^n

$f(n) = 100n + 5$

Note Inorder to express $f(n)$ using $O(n)$, It should satisfy the following constraints

$f(n) \leq c \cdot g(n)$ for $n \geq n_0$.

Replace 5 with n, $\Rightarrow$ $100n + n$, call it as $c \cdot g(n)$

$c \cdot g(n) = 100n + n$ for $n = 5$

$c \cdot g(n) = 101n$ for $n = 5$

$\therefore$ It satisfies the following Constraint

$f(n) \leq c \cdot g(n)$ for $n \geq n_0$

$\downarrow$ $\downarrow$ $\downarrow$ $\downarrow$

$100n + 5 \leq 101 \cdot n$ for $n \geq 5$

From the above equation $c = 101$, $g(n) = n$, $n_0 = 5$

$\therefore$ $f(n) \in O(g(n))$

$\Rightarrow$ $\underline{\underline{f(n) \in O(n)}}$

3. **Example - 3**

Let $f(n) = 10n^3 + 8$. Express $f(n)$ using Big-Oh (O)

$f(n) = 10n^3 + 8$

Replace 8 with $n^3 \Rightarrow 10n^3 + n^3$, call it as $c \cdot g(n)$

$c \cdot g(n) = 10n^3 + n^3$ for $n = 2$ $(n^3 = 8 \Rightarrow n = 2)$

Now following constraint is satisfied

$f(n) \le c \cdot g(n)$ for $n \ge n_0$

$10n^3 + 8 \le 11 \cdot n^3$ for $n \ge 2$

$\therefore c = 11, g(n) = n^3, n_0 = 2$

$f(n) \in O(g(n))$

$\Rightarrow \underline{f(n) \in O(n^3)}$

4. Example - 4

Let $f(n) = 100n + 5$. Express $f(n)$ using Θ.

The constraint to be satisfied is,

$c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$ for $n \ge n_0$

$100 \cdot n \le 100n + 5 \le 105 \cdot n$ for $n \ge 1$

$\therefore c_1 = 100, c_2 = 105, g(n) = n ; n_0 = 1$

$\therefore f(n) \in \Theta(g(n))$

$\Rightarrow \underline{f(n) \in \Theta(n)}$

5. Example - 5

Let $f(n) = 10n^3 + 5$. Express $f(n)$ using Big Theta (Θ)

Assume 5 as n^3.

$\Rightarrow 10n^3 + 5 \le 10n^3 + n^3$ for $n^3 \ge 5$

$\Rightarrow 10n^3 + 5 \le 11n^3$ for $n^3 \ge 5$

$\Rightarrow 5 \le n^3$

$\Rightarrow n^3 \ge 5$

$\Rightarrow n \ge 2$

The constraint to be satisfied is

$c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$ for $n \ge n_0$

$10 \cdot n^3 \le 10n^3 + 5 \le 11 \cdot n^3$ for $n \ge 2$.

$\therefore c_1 = 10, c_2 = 11, g(n) = n^3, n_0 = 2$.

$\therefore f(n) \in \Theta(g(n)) \Rightarrow \underline{f(n) \in \Theta(n^3)}$

imp

6. Property of asymptotic notation

Prove that If t1(n) ∈ O(g1(n)) and t2(n) ∈ O(g2(n)), then t1(n) + t2(n) ∈ O(max{g1(n), g2(n)}).

THEOREM If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

PROOF The proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, a_2, b_2: if $a_1 \le b_1$ and $a_2 \le b_2$, then $a_1 + a_2 \le 2\max\{b_1, b_2\}$.

Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some nonnegative integer n_1 such that

$$t_1(n) \le c_1 g_1(n) \quad \text{for all } n \ge n_1.$$

Similarly, since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \le c_2 g_2(n) \quad \text{for all } n \ge n_2.$$

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \ge \max\{n_1, n_2\}$ so that we can use both inequalities. Adding them yields the following:

$$\begin{aligned} t_1(n) + t_2(n) &\le c_1 g_1(n) + c_2 g_2(n) \\ &\le c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)] \\ &\le c_3 2\max\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$, with the constants c and n_0 required by the O definition being $2c_3 = 2\max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively. ■

So what does this property imply for an algorithm that comprises two consecutively executed parts? It implies that the algorithm's overall efficiency is determined by the part with a higher order of growth, i.e., its least efficient part:

$$\left.\begin{array}{|c|}\hline t_1(n) \in O(g_1(n)) \\ \hline t_2(n) \in O(g_2(n)) \\ \hline \end{array}\right\} \quad t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

imp

7. Mathematical Analysis of Non-recursive Algorithms

7.1. General Plan

6. Decide on a parameter indicating an input's size.
7. Identify the algorithm's basic operation.

8. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
9. Set up a sum expressing the number of times the algorithm's basic operation is executed
10. Using standard formulas and rules of sum manipulation, either find a closed form formula for the count or, at the very least, establish its order of growth.

Note: summation formulas

$$\sum_{i=l}^{u} 1 = u - l + 1 \quad \text{where } l \leq u \text{ are some lower and upper integer limits}$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2).$$

7.2. Largest element in a list of n numbers

Consider the problem of finding the value of the largest element in a list of n numbers. For simplicity, we assume that the list is implemented as an array. The following is pseudocode of a standard algorithm for solving the problem.

```
ALGORITHM  MaxElement(A[0..n − 1])
    //Determines the value of the largest element in a given array
    //Input: An array A[0..n − 1] of real numbers
    //Output: The value of the largest element in A
    maxval ← A[0]
    for i ← 1 to n − 1 do
        if A[i] > maxval
            maxval ← A[i]
    return maxval
```

Algorithm 8: MaxElements

Analysis:

1. The obvious measure of an input's size here is the number of elements in the array, i.e., n. The operations that are going to be executed most often are in the algorithm's for loop.
2. There are two opherations in the loop's body: the comparison A[i] > maxval and the assignment maxval ← A[i]. Since the comparison is executed on each repetition of the loop and the assignment is not, we should consider the comparison to be the algorithm's basic operation. Note that the number of comparisons will be the same for all arrays of size n; therefore, in terms of this metric, there is no need to distinguish among the worst, average, and best cases here.
3. Let us denote C(n) the number of times this comparison is executed and try to find a formula expressing it as a function of size n. The algorithm makes one comparison on each execution of

the loop, which is repeated for each value of the loop's variable i within the bounds 1 and n − 1, inclusive. Therefore, we get the following sum for C(n):

$$C(n) = \sum_{i=1}^{n-1} 1.$$

4. This is an easy sum to compute because it is nothing other than 1 repeated n − 1 times. Thus,

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

5. Express C(n) using asymptotic notation. Since there are no other parameters to be considered, the total no. of times the basic operation is executed remains same both in best and worst cases.

$$c_1 g(n) \le t(n) \le c_2 g(n) \text{ for all } n \ge n_0$$

$$n/2 \le n-1 \le 2n \text{ for all } n \ge 2$$

where $c_1 = ½$, $c_2 = 2$, $g(n) = n$, $t(n) = n-1$, $n_0 = 2$

so, by the definition,

C(n) € Θ(g(n))

➔<u>C(n) € Θ(n)</u>

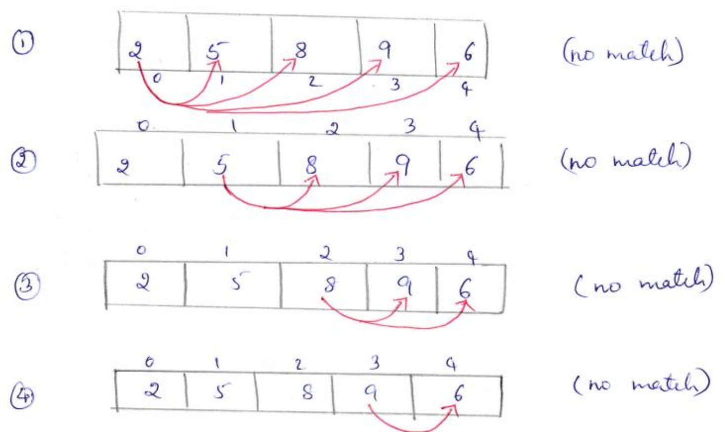
7.3. Element uniqueness problem

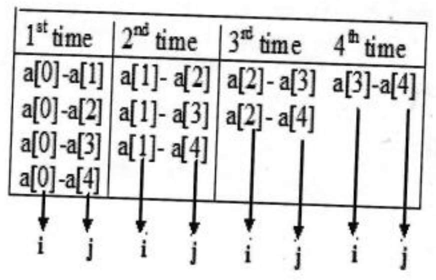
Consider the element uniqueness problem: check whether all the elements in a given array of n elements are distinct. This problem can be solved by the following straightforward algorithm.

```
ALGORITHM UniqueElements(A[0..n − 1])
    //Determines whether all the elements in a given array are distinct
    //Input: An array A[0..n − 1]
    //Output: Returns "true" if all the elements in A are distinct
    //          and "false" otherwise
    for i ← 0 to n − 2 do
        for j ← i + 1 to n − 1 do
            if A[i] = A[j] return false
    return true
```

Algorithm 9: Unique Elements

Design:





1st time	2nd time	3rd time	4th time
a[0]-a[1]	a[1]- a[2]	a[2]- a[3]	a[3]-a[4]
a[0]-a[2]	a[1]- a[3]	a[2]- a[4]	
a[0]-a[3]	a[1]- a[4]		
a[0]-a[4]			
i j	i j	i j	i j

Analysis:

1. The natural measure of the input’s size here is again n, the number of elements in the array.
2. Since the innermost loop contains a single operation (the comparison of two elements), we should consider it as the algorithm’s basic operation.
3. The number of element comparisons depends not only on n but also on whether there are equal elements in the array and, if there are, which array positions they occupy. We will limit our investigation to the worst case only. By definition, the worst case input is an array for which the number of element comparisons Cworst(n) is the largest among all arrays of size n.
4. An inspection of the innermost loop reveals that there are two kinds of worst-case inputs—inputs for which the algorithm does not exit the loop prematurely: arrays with no equal elements and arrays in which the last two elements are the only pair of equal elements. For such inputs, one comparison is made for each repetition of the innermost loop, i.e., for each value of the loop variable j between its limits i + 1 and n − 1; this is repeated for each value of the outer loop, i.e., for each value of the loop variable i between its limits 0 and n − 2. Accordingly, we get

$$C_{worst}(n) = \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2}[(n-1)-(i+1)+1] = \sum_{i=0}^{n-2}(n-1-i)$$

$$= \sum_{i=0}^{n-2}(n-1) - \sum_{i=0}^{n-2} i = (n-1)\sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2).$$

We also could have computed the sum $\sum_{i=0}^{n-2}(n-1-i)$ faster as follows:

$$\sum_{i=0}^{n-2}(n-1-i) = (n-1)+(n-2)+\cdots+1 = \frac{(n-1)n}{2},$$

where the last equality is obtained by applying summation formula (S2). Note that this result was perfectly predictable: in the worst case, the algorithm needs to compare all n(n − 1)/2 distinct pairs of its n elements.

5. Total no. of basic operation executed is $f(n) = \frac{n(n-1)}{2}$

$$t(n) \le cg(n) \;\; \textit{for all } n \ge n_0$$

$$\frac{n(n-1)}{2} \le n^2 \;\; \textit{for all } n \ge 0$$

where c = 1, g(n) = n^2, n_0 = 0

so, by the definition,

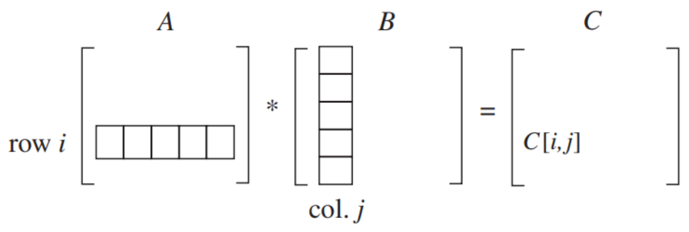
f(n) € O(g(n))

➔ f(n) € O(n^2)

imp

7.4. Matrix Multiplication

Given two n × n matrices A and B, find the time efficiency of the definition-based algorithm for computing their product C = AB. By definition, C is an n × n matrix whose elements are computed as the scalar (dot) products of the rows of matrix A and the columns of matrix B:



where $C[i, j] = A[i, 0]B[0, j] + \cdots + A[i, k]B[k, j] + \cdots + A[i, n-1]B[n-1, j]$ for every pair of indices $0 \le i,\ j \le n-1$.

```
ALGORITHM  MatrixMultiplication(A[0..n − 1, 0..n − 1], B[0..n − 1, 0..n − 1])
    //Multiplies two square matrices of order n by the definition-based algorithm
    //Input: Two n × n matrices A and B
    //Output: Matrix C = AB
    for i ← 0 to n − 1 do
        for j ← 0 to n − 1 do
            C[i, j] ← 0.0
            for k ← 0 to n − 1 do
                C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
    return C
```

1. We measure an input's size by matrix order n.
2. There are two arithmetical operations in the innermost loop here—multiplication and addition—that, in principle, can compete for designation as the algorithm's basic operation. Actually, we do

not have to choose between them, because on each repetition of the innermost loop each of the two is executed exactly once. So by counting one we automatically count the other.

3. Let us set up a sum for the total number of multiplications M(n) executed by the algorithm. (Since this count depends only on the size of the input matrices, we do not have to investigate the worst-case, average-case, and best-case efficiencies separately.) Obviously, there is just one multiplication executed on each repetition of the algorithm's innermost loop, which is governed by the variable k ranging from the lower bound 0 to the upper bound n − 1. Therefore, the number of multiplications made for every pair of specific values of variables i and j is

$$\sum_{k=0}^{n-1} 1,$$

and the total number of multiplications M(n) is expressed by the following triple sum

$$M(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} 1$$

$$M(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

4. Inorder to represent $f(n)=n^3$, using asymptotic notation Θ,

$$c_1 g(n) \le t(n) \le c_2 g(n) \text{ for all } n \ge n_0$$

$$n^3/2 \le n^3 \le 2n^3 \text{ for all } n \ge 0$$

where $c_1 = ½$, $c_2 = 2$, $g(n) = n^3$, $n_0 = 0$

so, by the definition,

M(n) € Θ(g(n))

➔<u>M(n) € Θ(n^3)</u>

imp

8. Mathematical Analysis of Recursive Algorithms

8.1. General Plan for Analyzing the Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or, at least, ascertain the order of growth of its solution.

imp

8.2. Factorial of n numbers

Compute the factorial function F (n) = n! for an arbitrary nonnegative integer n.

Since n!= 1 . ... . (n − 1) . n = (n − 1)!. n for n ≥ 1

and 0!= 1 by definition, we can compute F (n) = F (n − 1) . n with the following recursive algorithm.

```
ALGORITHM  F(n)
    //Computes n! recursively
    //Input: A nonnegative integer n
    //Output: The value of n!
    if $n = 0$ return 1
    else return $F(n-1) * n$
```

Ex: To find factorial of 4 [i.e, 4!]

$4! = 4 * 3!$

$3! = 3 * 2!$

$2! = 2 * 1!$

$1! = 1 * 0!$

$n! = n * (n-1)!$ when $n \neq 0$

$\boxed{0! = 1}$

$n! = 1$ when $n = 0$

Hence factorial of n can be written as

$$fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * fact(n-1) & \text{if } n > 0 \end{cases}$$

Analysis:

1. Input size: n
2. Basic operation: Multiplication operation
3. The total no. of multiplications can be obtained using the following recurrence relation:

$t(n) = 0$, if $n = 0$. $\Rightarrow t(0) = 0$.

$t(n) = 1 + t(n-1)$ if $n > 0$

4. Solving recurrence relation:

$$\begin{aligned} t(n) &= 1 + t(n-1) & t(n-1) &= 1 + t(n-2) \\ &= 1 + [1 + t(n-2)] \\ &= 2 + t(n-2) & t(n-2) &= 1 + t(n-3) \\ &= 2 + [1 + t(n-3)] \\ &= 3 + t(n-3) \\ &= 4 + t(n-4) \\ &\vdots \end{aligned}$$

$t(n) = i + t(n-i)$

Inorder to get initial condition, make $i = n$

$$\begin{aligned} \therefore t(n) &= n + t(n-n) \\ &= n + t(0) \\ &= n + 0 \end{aligned}$$

So t(n) = n

5. Inorder to represent t(n)=n, using asymptotic notation Θ,

$$c_1 g(n) \leq t\ (n) \leq c_2 g(n) \text{ for all } n \geq n_0$$

$$n/2 \leq n \leq 2n \text{ for all } n \geq 0$$

where $c_1 = ½$, $c_2 = 2$, g(n) = n, $n_0 = 0$

so, by the definition,

M(n) € Θ(g(n))

➔<u>M(n) € Θ(n)</u>

imp

8.3. Tower of hanoi

The Tower of Hanoi puzzle. In this puzzle, we (or mythical monks, if you do not like to move disks) have n disks of different sizes that can slide onto any of three pegs. Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top. The goal is to move all the disks to the third peg, using the second one as an auxiliary, if necessary. We can move only one disk at a time, and it is forbidden to place a larger disk on top of a smaller one. The problem has an elegant recursive solution, which is illustrated in Figure . To move n > 1 disks from peg 1 to peg 3 (with peg 2 as auxiliary), we first move recursively n − 1 disks from peg 1 to peg 2 (with peg 3 as auxiliary), then move the largest disk directly from peg 1 to peg 3, and, finally, move recursively n − 1 disks from peg 2 to peg 3 (using peg 1 as auxiliary). Of course, if n = 1, we simply move the single disk directly from the source peg to the destination peg.

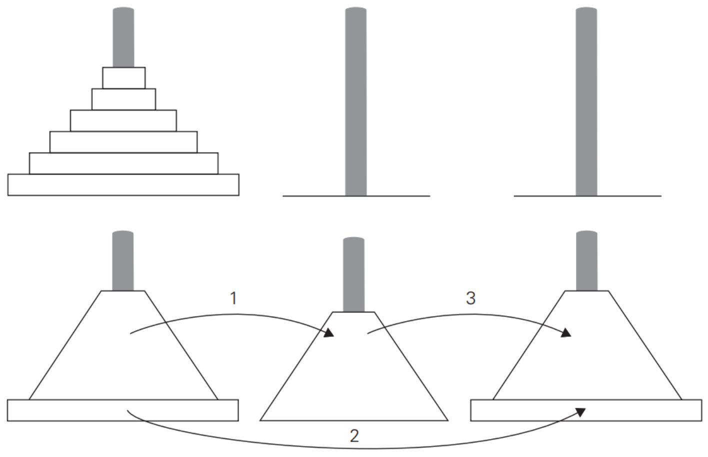


FIGURE 2.4 Recursive solution to the Tower of Hanoi puzzle.

```
Algorithm
//Input :- n : total no. of disks to be moved
//output :- All n disks should be available in the
            destination

if (n==1)
        move disk-1 from source to Destination
        return
end if
// Move n-1 disks from A to C using B as temp
    TowerofHanoi (n-1, source, dest, temp)
// move n-1 disks from B to C using A as temp
    TowerofHanoi (n-1, temp, source, dest).
```

Analysis:

1. Input size: n
2. Basic operation: movement of disks
3. Total no. of disks movements can be obtained using recurrence relation:

$$\boxed{M(n) = \begin{cases} 1 & \text{if } n=1 \\ \underbrace{M(n-1)}_{\text{No. of disk movement from src to temp}} + \underbrace{1}_{\text{No. of disk movement from src to Destination}} + \underbrace{M(n-1)}_{\text{No. of disk movement from temp to dest.}} & \text{otherwise.} \end{cases}}$$

4. With the obvious initial condition M(1) = 1, we have the following recurrence relation for the number of moves M(n): M(n) = 2M(n − 1) + 1 for n > 1,

   M(1) = 1

5. We solve this recurrence by the same method of backward substitutions:

$$\begin{aligned} M(n) &= 2M(n-1)+1 && \text{sub. } M(n-1) = 2M(n-2)+1 \\ &= 2[2M(n-2)+1]+1 = 2^2M(n-2)+2+1 && \text{sub. } M(n-2) = 2M(n-3)+1 \\ &= 2^2[2M(n-3)+1]+2+1 = 2^3M(n-3)+2^2+2+1. \end{aligned}$$

   The pattern of the first three sums on the left suggests that the next one will be

$2^4M(n-4)+2^3+2^2+2+1$, and generally, after i substitutions, we get

$$M(n)=2^iM(n-i)+2^{i-1}+2^{i-2}+\cdots+2+1=2^iM(n-i)+2^i-1.$$

Since the initial condition is specified for $n=1$, which is achieved for $i=n-1$, we get the following formula for the solution to recurrence (2.3):

$$\begin{aligned}M(n)&=2^{n-1}M(n-(n-1))+2^{n-1}-1\\&=2^{n-1}M(1)+2^{n-1}-1=2^{n-1}+2^{n-1}-1=2^n-1.\end{aligned}$$

$c_1\cdot g(n)\le f(n)\le c_2\cdot g(n)$ for $n\ge n_0$

$\frac{2^n}{2}\le 2^n-1\le 2\cdot 2^n$ for $n\ge 1$

where $c_1=\frac{1}{2}$, $c_2=2$, $g(n)=2^n$, $n_0=1$

$\therefore\ M(n)\in\Theta(g(n))$

$\Rightarrow M(n)\in\Theta(2^n)$

imp

8.4. Selection sort

We start selection sort by scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list. Then we scan the list, starting with the second element, to find the smallest among the last n − 1 elements and exchange it with the second element, putting the second smallest element in its final position. Generally, on the ith pass through the list, which we number from 0 to n − 2, the algorithm searches for the smallest item among the last n − i elements and swaps it with A_i:

$$A_0\le A_1\le\cdots\le A_{i-1}\ \big|\ A_i,\ldots,A_{min},\ldots,A_{n-1}$$

in their final positions — the last $n-i$ elements

After n − 1 passes, the list is sorted. Here is pseudocode of this algorithm, which, for simplicity, assumes that the list is implemented as an array:

```
ALGORITHM  SelectionSort(A[0..n − 1])
    //Sorts a given array by selection sort
    //Input: An array A[0..n − 1] of orderable elements
    //Output: Array A[0..n − 1] sorted in nondecreasing order
    for i ← 0 to n − 2 do
        min ← i
        for j ← i + 1 to n − 1 do
            if A[j] < A[min]  min ← j
        swap A[i] and A[min]
```

As an example, the action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated in Figure 3.1. The analysis of selection sort is straightforward. The input size is given by the number of elements n; the basic operation is the key comparison A[j] < A[min]. The number of times it is executed depends only on the array size and is given by the following sum:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i).$$

```
| 89   45   68   90   29   34   17
  17 | 45   68   90   29   34   89
  17   29 | 68   90   45   34   89
  17   29   34 | 90   45   68   89
  17   29   34   45 | 90   68   89
  17   29   34   45   68 | 90   89
  17   29   34   45   68   89 | 90
```

Since we have already encountered the last sum in analyzing the algorithm of Example 2 in Section 2.3, you should be able to compute it now on your own. Whether you compute this sum by distributing the summation symbol or by immediately getting the sum of decreasing integers, the answer, of course, must be the same:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}.$$

Thus, selection sort is a $\Theta(n^2)$ algorithm on all inputs. Note, however, that the number of key swaps is only $\Theta(n)$, or, more precisely, $n-1$ (one for each repetition of the i loop). This property distinguishes selection sort positively from many other sorting algorithms.

imp

8.5. Bubble Sort

Another brute-force application to the sorting problem is to compare adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, we end up “bubbling up” the largest element to the last position on the list. The next pass bubbles up the second largest element, and so on, until after n − 1 passes the list is sorted. Pass i (0 ≤ i ≤ n − 2) of bubble sort can be represented by the following diagram

Algorithm:

```
ALGORITHM  BubbleSort(A[0..n − 1])
    //Sorts a given array by bubble sort
    //Input: An array A[0..n − 1] of orderable elements
    //Output: Array A[0..n − 1] sorted in nondecreasing order
    for i ← 0 to n − 2 do
        for j ← 0 to n − 2 − i do
            if A[j + 1] < A[j]  swap A[j] and A[j + 1]
```

Design:

89 $\overset{?}{\leftrightarrow}$ 45 68 90 29 34 17
45 89 $\overset{?}{\leftrightarrow}$ 68 90 29 34 17
45 68 89 $\overset{?}{\leftrightarrow}$ 90 $\overset{?}{\leftrightarrow}$ 29 34 17
45 68 89 29 90 $\overset{?}{\leftrightarrow}$ 34 17
45 68 89 29 34 90 $\overset{?}{\leftrightarrow}$ 17
45 68 89 29 34 17 | 90

45 $\overset{?}{\leftrightarrow}$ 68 $\overset{?}{\leftrightarrow}$ 89 $\overset{?}{\leftrightarrow}$ 29 34 17 | 90
45 68 29 89 $\overset{?}{\leftrightarrow}$ 34 17 | 90
45 68 29 34 89 $\overset{?}{\leftrightarrow}$ 17 | 90
45 68 29 34 17 | 89 90

etc.

(complete the iteration)

Passes	1st Pass	2nd Pass	3rd Pass	4th pass	5th pass	6th pass
n = 7 Comparing	0 – 1 1 – 2 2 – 3 3 – 4 4 – 5 5 – 6	0 – 1 1 – 2 2 – 3 3 – 4 4 – 5	0 – 1 1 – 2 2 – 3 3 – 4	0 – 1 1 – 2 2 – 3	0 – 1 1 – 2	0 – 1
	j = 0 to 5	j = 0 to 4	j = 0 to 3	j = 0 to 2	j = 0 to 1	j = 0 to 0

⇒ 0 to 7 - 2
⇒ 0 to n - 2
⇒ 0 to n - (1+1)
⇒ 0 to n - i - 1

Analysis:

```
for i ← 0 to n-2
    for j ← 0 to n-i-2
        if (a[j+1] < a[j]) swap.
```

$$f(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-i-2} 1$$

$$= \sum_{i=0}^{n-2} [n-i-2-0+1]$$

$$= \sum_{i=0}^{n-2} n-i-1$$

$$= (n-0-1) + (n-1-1) + \cdots + (n-(n-2)-1)$$

$$= (n-1) + (n-2) + \cdots + 2 + 1$$

$$f(n) = \frac{n(n-1)}{2}$$

Express $f(n)$ using Θ notation, So constraint to be satisfied is

$$c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n) \quad \text{for } n \ge n_0.$$

$$\frac{1}{4} n^2 \le \frac{n(n-1)}{2} \le n^2 \quad \text{for } n \ge 2.$$

$$\therefore c_1 = \frac{1}{4},\ c_2 = 1,\ g(n) = n^2,\ n_0 = 2.$$

$$\therefore f(n) \in \Theta(g(n))$$

$$\Rightarrow \underline{f(n) \in \Theta(n^2)}$$

imp

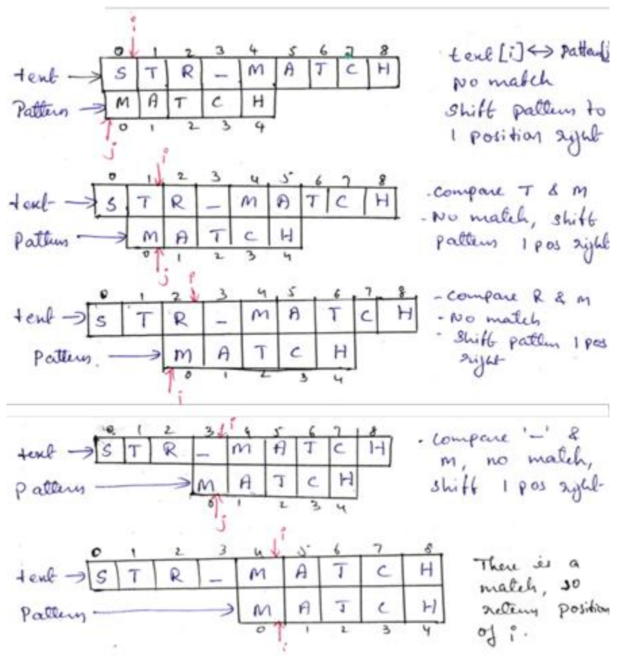
8.6. Brute-Force String Matching

Recall the string-matching problem introduced in Section 1.3: given a string of n characters called the text and a string of m characters (m ≤ n) called the pattern, find a substring of the text that matches the pattern. To put it more precisely, we want to find i—the index of the leftmost character of the first matching substring in the text—such that ti = p0,...,ti+j = pj ,...,ti+m−1 = pm−1

$$\begin{array}{ccccccccccc} t_0 & \dots & t_i & \dots & t_{i+j} & \dots & t_{i+m-1} & & \dots & t_{n-1} & \text{text } T \\ & & \updownarrow & & \updownarrow & & \updownarrow & & & & \\ & & p_0 & \dots & p_j & \dots & p_{m-1} & & \text{pattern } P & & \end{array}$$

If matches other than the first one need to be found, a string-matching algorithm can simply continue working until the entire text is exhausted. A brute-force algorithm for the string-matching problem is quite obvious: align the pattern against the first m characters of the text and start matching the corresponding pairs of characters from left to right until either all the m pairs of the characters match (then the algorithm can stop) or a mismatching pair is encountered. In the latter case, shift the pattern one position to the right and resume the character comparisons, starting again with the first character of the pattern and its counterpart in the text. Note that the last position in the text that can still be a beginning of a matching substring is n − m(provided the text positions are indexed from 0 to n − 1). Beyond that position, there are not enough characters to match the entire pattern; hence, the algorithm need not make any comparisons there.

Design:


text
Pattern
S T R _ M A T C H
M A T C H
text[i] ↔ pattern[j]
No match
Shift pattern to 1 position right
compare T & M
No match, shift pattern 1 pos right
compare R & M
No match
Shift pattern 1 pos right
compare '_' & M, no match, shift 1 pos right
There is a match, so return position of i.

```
ALGORITHM BruteForceStringMatch(T[0..n − 1], P[0..m − 1])
    //Implements brute-force string matching
    //Input: An array T[0..n − 1] of n characters representing a text and
    //         an array P[0..m − 1] of m characters representing a pattern
    //Output: The index of the first character in the text that starts a
    //         matching substring or −1 if the search is unsuccessful
    for i ← 0 to n − m do
        j ← 0
        while j < m and P[j] = T[i + j] do
            j ← j + 1
        if j = m return i
    return −1
```

Analysis:

```
for i ← 0 to n − m do
    j ← 0
    while j < m
```

The above code can also be written as,

```
for i ← 0 to n − m do
    for j ← 0 to m − 1 do
        :  // operation
```

$$f(n) = \sum_{i=0}^{n-m} \sum_{j=0}^{m-1} 1$$

$$= \sum_{i=0}^{n-m} [m - 1 - 0 + 1] = \sum_{i=0}^{n-m} m = m \sum_{i=0}^{n-m} 1$$

$$= m [n - m - 0 + 1] = m [n - m + 1]$$

$$f(n) = \underline{mn - m^2 + m}$$

Inorder to represent it using Θ notation, It should follow the following constraint,

$$c_1 g(n) \le f(n) \le c_2 \cdot g(n) \quad \text{for } n \ge n_0 .$$

$$\frac{mn}{2} \le mn - m^2 + m \le 2mn \quad \text{for } m \ge 0, n \ge 0$$

$$\therefore c_1 = \frac{1}{2}, \quad c_2 = 2, \quad g(n) = mn, \quad m_0 = 0, \quad n_0 = 0$$

$$\therefore f(n) \in \Theta(g(n))$$

$$\Rightarrow f(n) \in \Theta(mn)$$