NEW HORIZON
COLLEGE OF ENGINEERING
New Horizon Knowledge Park, Ring Road, Marathalli
Autonomous College Permanently Affiliated to VTU, Approved by AICTE & UGC
Accredited by NAAC with 'A' Grade, Accredited by NBA

Department of Computer Science & Engineering (Data Science)

DESIGN AND ANALYSIS OF THE ALGORITHM

22CDS52

Semester: Vth AY: 2024-25

NEW HORIZON COLLEGE OF ENGINEERING

VISION

To emerge as an institute of eminence in the fields of engineering, technology and management in serving the industry and the nation by empowering students with a high degree of technical, managerial and practical competence.

MISSION

To strengthen the theoretical, practical and ethical dimensions of the learning process by fostering a culture of research and innovation among faculty members and students.

To encourage long-term interaction between the academia and industry through their involvement in the design of curriculum and its hands-on implementation.

To strengthen and mould students in professional, ethical, social and environmental dimensions by encouraging participation in co-curricular and extracurricular **activities**

QUALITY POLICY

To provide educational services of the highest quality both curricular and co-curricular to enable students integrate skills and serve the industry and society equally well at global level.

VALUES

- Academic Freedom
- Integrity Inclusiveness
- Innovation
- Professionalism
- Social Responsibility

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING (DATA SCIENCE)

PROGRAM OUTCOMES (Pos)

PO1 Engineering Knowledge: Apply knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex Computer Science and Data Science engineering problems.

PO2 Problem Analysis: Identify, formulate, review research literature and analyze complex Computer Science and Data Science engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences and engineering sciences.

PO3 Design / Development of Solutions: Design solutions for complex Computer Science and Data Science engineering problems and design system components or processes that meet specified needs with appropriate consideration for public health and safety, cultural, societal and environmental considerations.

PO4 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.

PO5 Modern tool usage: Create, select and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex Computer Science and Data Science engineering activities with an understanding of the limitations.

PO6 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 in Computer Science and Data Science Engineering.

PO7 Environment and sustainability: Understand the impact of the professional engineering solutions in Computer Science and Data Science engineering in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8 Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9 Individual and Team Work: Function effectively as an individual and as a member or leader to diverse teams, and in multidisciplinary settings.

PO10 Communication: Communicate effectively on complex Computer Science and Data Science engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective report and design documentation, make effective presentations, and give and receive clear instructions.

PO11 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.

PO12 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 (PSOs)

PSO1	Apply data analysis techniques, algorithmic expertise, and advanced modelling to effectively solve complex problems across various domains demonstrating heir capacity to derive insights and propose innovative solutions in the realm of data-driven technologies
PSO2	Collaborate proficiently with experts from diverse fields and actively engage in continuous professional growth in the domain of Computer Science & Engineering, specializing in the field of data science.

Module 1

Chapter 1:

1. What is an algorithm?

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.

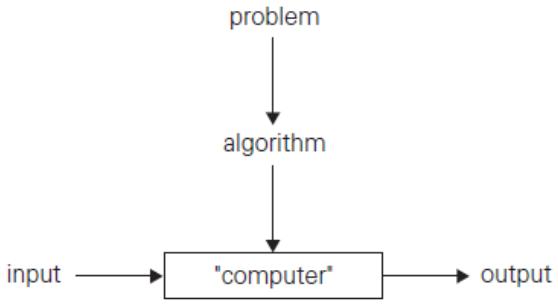


Figure 1: The notion of the algorithm.

1.1 All Algorithms Must Satisfy the Following Criteria

- The non-ambiguity requirement for each step of an algorithm cannot be compromised.
- The range of inputs for which an algorithm works has to be specified carefully.
- The same algorithm can be represented in several different ways.
- There may exist several algorithms for solving the same problem.
- Algorithms for the same problem can be based on very different ideas and can solve the problem at dramatically different speeds.

Characteristics of an algorithm:

- **Input:** Zero / more quantities are externally supplied**.**
- **Output:** At least one quantity is produced.
- **Definiteness:** Each instruction is clear and unambiguous.
- **Finiteness:** If the instructions of an algorithm are traced, then for all cases, the algorithm must terminate after a finite number of steps.
- **Efficiency:** Every instruction must be very basic and run in a short time.

1.2 Fundamentals of Algorithmic Problem Solving

Let us analyze and briefly discuss a sequence of steps one typically goes through in designing and analyzing an algorithm (Figure 1.2).

a) **Understanding the Problem:** Before designing an algorithm, thoroughly understand the problem and clarify any doubts. Familiarize yourself with common problem types and existing algorithms, considering their strengths and weaknesses. If no existing solution fits, you'll need to design your own. Clearly define the input set your algorithm must handle to avoid failures on edge cases. A correct algorithm should work for all valid inputs, not just most of the time, so don't rush this critical initial step.

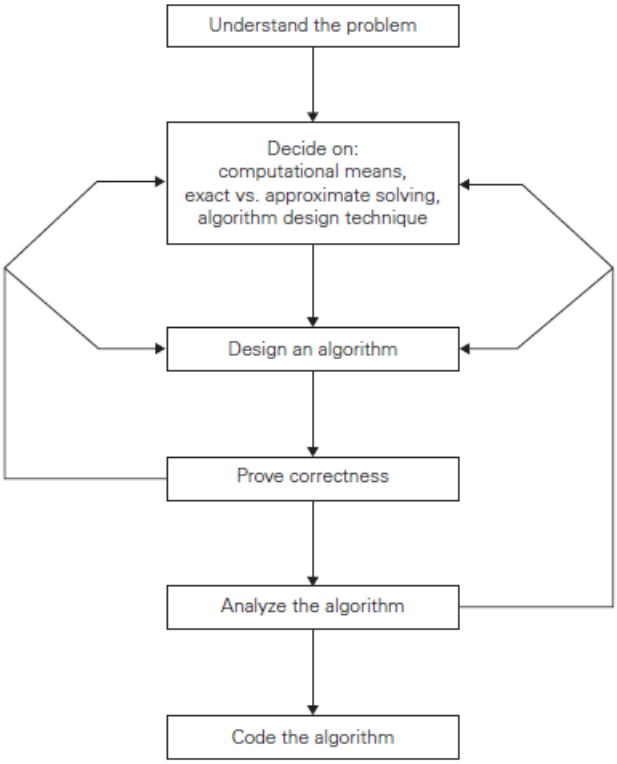


Figure 1.2: Algorithm design and analysis process.

b) **Ascertaining the Capabilities of the Computational Device:** Once you fully understand a problem, consider the capabilities of the device the algorithm will run on. Most algorithms are designed for computers resembling the von Neumann architecture, which processes one instruction at a time, called sequential algorithms. Some newer computers, however, support parallel execution, requiring parallel algorithms. While algorithm design often avoids considering specific hardware limitations, for practical applications involving complex tasks or large datasets, it's crucial to account for the computer's speed and memory. Despite modern computers

being fast, certain problems may still require careful attention to hardware constraints.

Choosing between Exact and Approximate Problem Solving: The next key decision is choosing between an exact or an approximation algorithm. Exact algorithms solve the problem completely, while approximation algorithms provide near solutions.

Algorithm Design Techniques: An ***algorithm design technique*** (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing. First, they offer guidance for creating algorithms for new problems where no satisfactory solution is yet known. While not every technique will apply to every problem, they collectively form a valuable toolkit for problem-solving. Second, algorithms are the cornerstone of computer science. Algorithm design techniques make it possible to classify algorithms according to an underlying design idea.

c) **Designing an Algorithm and Data Structures:** Designing algorithms for specific problems can be complex despite having a range of general algorithm design techniques. Sometimes, these techniques may not apply directly, requiring the combination of several methods or creative problem-solving. Choosing the right data structures is also crucial, as it can significantly impact the algorithm's efficiency. *Algorithms + Data Structures = Programs*

Pseudocode is a mixture of a natural language and programming language-like constructs.

d) **Proving an Algorithm's Correctness:** Once an algorithm is specified, its correctness must be proven by showing it produces the correct result for all valid inputs within a finite time. which can be achieved using methods like mathematical induction. While testing specific inputs helps, it cannot conclusively prove correctness, and a single failure can indicate an issue. For approximation algorithms, correctness is shown by ensuring the error remains within a predefined limit.

e) **Analyzing an Algorithm**: After correctness, the most important is efficiency. In fact, there are two kinds of algorithm efficiency: time efficiency, indicating how fast the algorithm runs, and space efficiency, indicating how much extra memory it uses. Another desirable characteristic of an algorithm is **simplicity**. **The generality** of the problem the algorithm solves and the set of inputs it accepts.

f) ***Coding an Algorithm:*** Most algorithms are ultimately destined to be implemented as computer programs. Implementing an algorithm correctly is necessary but not sufficient: you would not like to diminish your algorithm's power by an inefficient implementation. Modern compilers do provide a certain safety net in this regard, especially when they are used in their code optimization mode.

1.3 Important problem types:

Sorting: The sorting problem involves rearranging the items of a list in nondecreasing order, provided that the list items allow for such an arrangement (a relation of total ordering exists). Common sorting tasks include numbers, characters, strings, or records (e.g., student or employee data), with a specific attribute, called a *key*, guiding the sorting. Sorting is useful because it simplifies tasks like searching and ranking (e.g., search results or student GPAs), and serves as a step in various algorithms, such as geometric algorithms or data compression.

There are numerous sorting algorithms, each with different strengths. Some are simple but slow, while others are faster but more complex. Some perform better on random data, others on nearly sorted lists. No single algorithm works best in all situations, which is why research on sorting algorithms continues, despite many algorithms already achieving near-optimal performance (about $n \log_2 n$ comparisons).

Two important features of sorting algorithms are:

- Stability: A stable algorithm preserves the relative order of equal elements. This is useful, for example, when sorting by multiple criteria (e.g., alphabetically and by GPA).
- Memory usage: In-place algorithms require little extra memory, while others may need more.

Both factors influence the choice of sorting algorithms in different scenarios.

Searching: The searching problem involves finding a specific value, known as a search key, within a set or multiset. There are various search algorithms, from simple sequential search to highly efficient but limited ones like binary search. Some algorithms are better suited to certain conditions, like requiring less memory or working only with sorted arrays.

In real-world applications, especially with large databases, it's crucial to choose the right algorithm and data structure. This decision must consider not only search speed but also the

ability to efficiently handle frequent additions or deletions of data. Managing very large data sets presents unique challenges with significant implications for practical applications.

String Processing: In recent decades, the rise of applications handling nonnumerical data has increased interest in string-handling algorithms. Strings are sequences of characters from an alphabet, including text strings, bit strings, and gene sequences. Although string-processing algorithms have long been essential in computer science for language and compiling tasks, string matching—searching for a specific word in a text—has gained special attention.

Graph problems: Informally, graphs are structures made up of vertices (points) and edges (lines connecting some vertices). Graphs are valuable for both theoretical and practical applications, including transportation, communication, social networks, and project scheduling. They are also used to study the technical and social aspects of the Internet.

Common graph algorithms include:

- **Graph traversal**: determining how to reach all points in a network,
- **Shortest-path algorithms**: finding the best route between two points,
- **Topological sorting**: checking the consistency of directed graphs, like course prerequisites.

Some graph problems, such as the **Traveling Salesman Problem (TSP)** and the **graph-colouring problem**, are computationally difficult. The TSP involves finding the shortest route through multiple cities, with applications in areas like circuit design and genetic engineering. The graph-coloring problem involves assigning the fewest colors to graph vertices, ensuring no adjacent vertices share the same color, with applications in event scheduling.

Combination problem: The traveling salesman problem and the graph coloring problem are examples of combinatorial problems, which involve finding a combinatorial object—like a permutation or subset—that meets certain constraints and may also have optimal properties, such as minimal cost. These problems are typically very challenging due to two reasons: the rapid growth in the number of possible solutions as problem size increases, and the lack of efficient algorithms to solve most of them exactly within a reasonable time frame. While some problems, like the shortest path problem, can be solved efficiently, they are rare exceptions. The difficulty of these problems is a major unresolved issue in theoretical computer science.

Geometric Problems: Geometric algorithms deal with objects like points, lines, and polygons. While the ancient Greeks developed methods for solving geometric problems using rulers and

compasses, interest in these methods faded for about 2000 years. The advent of computers revived interest in geometric algorithms, now applied in fields like computer graphics, robotics, and tomography. Modern geometric algorithms solve problems such as the closest-pair problem (finding the closest pair of points in a set) and the convex-hull problem (finding the smallest convex polygon containing all points in a set).

Numeric Problem: Numerical problems involve solving mathematical tasks such as equations, systems of equations, and integrals, which typically require approximate solutions due to the continuous nature of real numbers. In computers, real numbers are represented approximately, leading to potential round-off errors that can distort algorithm outputs after numerous operations. Over the years, sophisticated numerical algorithms have been developed and used in scientific and engineering fields. However, in recent decades, the focus of the computing industry has shifted toward business applications involving data storage, retrieval, and networking, reducing the prominence of numerical analysis. Despite this shift, basic knowledge of numerical algorithms remains important, and several classical methods are discussed in later sections.

2.1 Analysis Framework:

The general framework for analyzing the efficiency of algorithms. There are two kinds of efficiency: **time efficiency** and **space efficiency**.

- ***Time efficiency***, also called **time complexity**, indicates how fast an algorithm in question runs.
- ***Space efficiency***, also called **space complexity**, refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output.

1. **Measuring an Input's Size:** It is observed 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 where the choice of a **parameter indicating an input size does matter**. The choice of an appropriate size metric can be influenced by the operations of the algorithm in question. For example, how should we measure an input's size for a spell-checking algorithm? If the algorithm examines individual characters of its input, then we should

measure the size by the number of characters; if it works by processing words, we should count their number in the input.

We should make a special note about measuring the size of inputs for algorithms involving **properties of numbers** (e.g., checking whether a given integer n is prime). For such Algorithms, computer scientists prefer measuring size by the number ***b*** of bits in the ***n***'s binary representation: b = [log2 n] + 1. This metric usually gives a better idea about the efficiency of the algorithms in question.

2. **Units for measuring running time:** To measure 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. For example, most **sorting** algorithms work by **comparing elements** (keys) of a list being sorted with each other; for such algorithms, the basic operation is a key comparison. As another example, algorithms for **matrix multiplication** and **polynomial evaluation** require two arithmetic operations: **multiplication and addition**.

   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:

   $$T(n) = c_{op}C(n)$$

   Unless ***n*** is extremely large or very small, the formula can give a reasonable estimate of the algorithm's running time.

   It is for these reasons that the efficiency analysis framework ignores multiplicative constants and concentrates on the count's **order of growth** to within a constant multiple for large-size inputs.

3. **Order of growth**: Why is this emphasis on the count's order of growth for large input sizes? Because for large values of n, it is the function's order of growth that counts: just look at the table, which contains values of a few functions particularly important for the analysis of algorithms.

TABLE 2.1 Values (some approximate) of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

4. Worst case, best case, Average Case:

- An algorithm's efficiency is a function of a parameter indicating the size of the algorithm's input.
- But there are many algorithms for which running time depends not only on an input size but also on the specifics of a particular input.
- The 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.
- 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:* $C_{worst}(n) = n$.

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
```

Worst-case Efficiency:

- 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.

- 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 Cworst(n).
- The running time will not exceed Cworst(n), its running time on the worst-case inputs.

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.
- We determine the kind of inputs for which the count C(n) will be the smallest among all possible inputs of size n.
- The best-case inputs for sequential search are lists of size n with their first element equal to a search key accordingly, Cbest(n) = 1 for this algorithm.
- The analysis of the best-case efficiency is not nearly as important as that of the worst-case efficiency.
- The best-case efficiency deteriorates only slightly for almost-sorted arrays.

Average-case Efficiency

- To analyze the algorithm's average case efficiency, we must make some assumptions about possible inputs of size *n.*
- The standard assumptions are that
  (a) The probability of a successful search is equal to $p\ (0 \le p \le 1)$.
  (b) *The* probability of the first match occurring in the *ith position of the list is the same* for every *i.*
- 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).
- general formula yields some quite reasonable answers. For example, if $p = 1$ (the search must be successful), the average number of key comparisons made by sequential search is $(n + 1)/2$; that is, the algorithm will inspect, on average, about half of the list's elements.

- If $p = 0$ *(the search must be unsuccessful), the* average number of key comparisons will be *n because the algorithm will inspect* all *n elements on all such inputs.*

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

Performance Analysis:

There are two kinds of efficiency: **Time efficiency** and **Space efficiency**.

a) Time efficiency indicates how fast an algorithm in question runs.
b) Space efficiency deals with the extra space the algorithm requires.

In the early days of electronic computing, both resources, **time** and **space** were at a premium. Research has shown that for most problems, we can achieve much more spectacular progress at speed than in space. Therefore, we primarily concentrate on time efficiency.

Space complexity:

The total amount of computer memory required by an algorithm to complete its execution is called the **space complexity** of that algorithm. The Space required by an algorithm is the sum of the following components.

a) A **fixed** part that is independent of the input and output. This includes memory space for codes, variables, constants, and so on.
b) A **variable** part that depends on the input, output, and recursion stack. (We call these parameters instance characteristics)

Space requirement *S(P)* of an algorithm *P,* ***S(P) = c + Sp*** where ***c*** is a constant depends on the fixed part, ***Sp*** is the instance characteristics

Time complexity:

Usually, the execution time or run-time of the program is refereed as its time complexity denoted by *tp* (instance characteristics). This is the sum of the time taken to execute all

Instructions in the program. Exact estimation runtime is a complex task, as the number of instructions executed is dependent on the input data. Also, different instructions will take different times to execute. So, for the estimation of the time complexity, we count only the number of program steps. We can determine the steps needed by a program to solve a particular problem instance in two ways.

ASYMPTOTIC NOTATIONS AND ITS PROPERTIES

Asymptotic notation is a notation which is used to take meaningful statements about the efficiency of a program.

The efficiency analysis framework concentrates on the order of growth of an algorithm's basic operation count as the principal indicator of the algorithm's efficiency.

To compare and rank such orders of growth, computer scientists use three notations, they are:

- Big oh notation

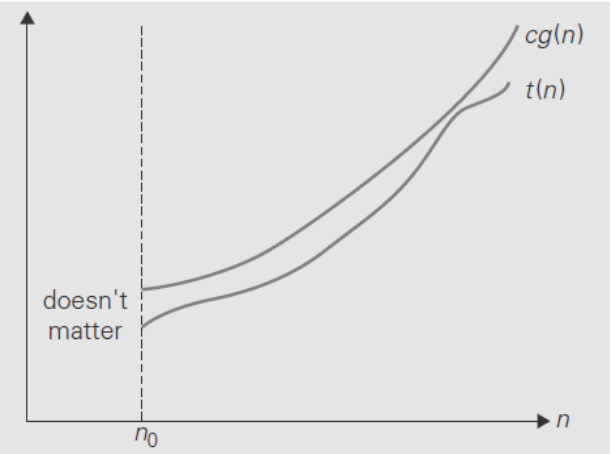
Ω - Big omega notation

Θ - Big theta notation

(i) O - Big oh notation

Definition: A function *t(n)* is said to be in *O(g(n)),* denoted $t(n) \in O(g(n))$, if t (n) is bounded above by some constant multiple of g(n) for all large n, i.e., if there exists some positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq cg(n) \text{ for all } n \geq n_0.$$



Big-oh notation: $t(n) \in O(g(n))$.

Example 1:

$$n \in O(n^2), \qquad 100n + 5 \in O(n^2), \qquad \frac{1}{2}n(n-1) \in O(n^2).$$

Example 2: Prove the assertions $100n + 5 \in O(n^2)$.

Proof: $100n + 5 \leq 100n + n$ (for all $n \geq 5$)

$= 101n$

$\leq 101n^2$ ($\because n \leq n^2$)

Since, the definition gives us a lot of freedom in choosing specific values for constants **c** and $\mathbf{n_0}$. We have c=101 and n_0=5

Example 3: Prove the assertions $100n + 5 \in O(n)$.

Proof: $100n + 5 \leq 100n + 5n$ (for all $n \geq 1$)

$= 105n$

i.e., $100n + 5 \leq 105n$

i.e., $t(n) \leq cg(n)$

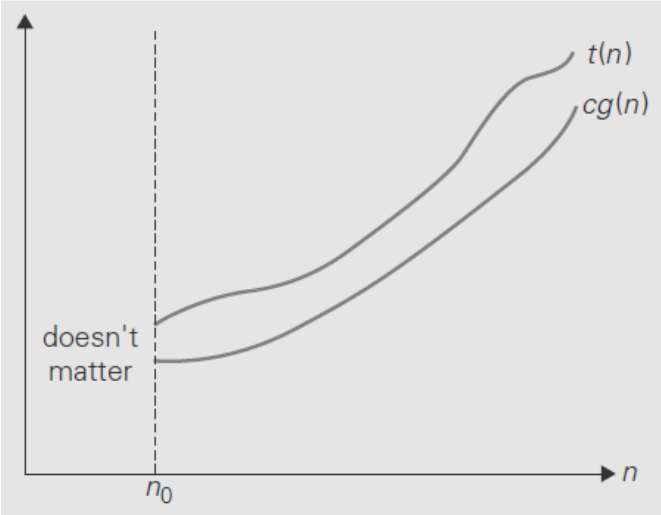
$\therefore 100n + 5 \in O(n)$ with c=105 and n_0=1

(ii) <u>Ω - Big omega notation:</u>

Definition: 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 exists some positive constant c and some nonnegative integer n0 such that

t(n) ≥ c g(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.



Big-omega notation: *t (n)* ∈ Ω *(g(n))*.

Example 4: Prove the assertions $n^3+10n^2+4n+2 \in \Omega(n^2)$.

Proof: $n^3+10n^2+4n+2 \geq n^2$ (for all $n \geq 0$)

i.e., by definition $t(n) \geq cg(n)$, where c=1 and n_0=0

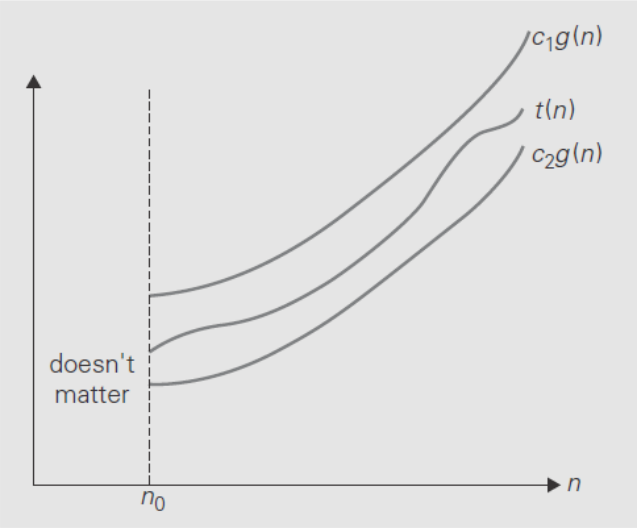
(iii) Θ - Big theta notation

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 n0 such that

$$c2g(n) \leq t(n) \leq c1g(n) \text{ for all } n \geq n0.$$

Where *t(n)* and *g(n*) are nonnegative functions defined on the set of natural numbers.

Θ = Asymptotic tight bound = Useful for average case analysis



Big-theta notation: $t(n) \in \Theta(g(n))$.

Example 5: Prove the assertions $\frac{1}{2}n(n-1) \in \Theta(n^2)$.

Proof: First prove the right inequality (the upper bound):

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

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

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \left[\frac{1}{2}n\right]\left[\frac{1}{2}n\right] \quad \text{for all } n \geq 2.$$

$$\therefore \quad \frac{1}{2}n(n-1) \geq \frac{1}{4}n^2$$

i.e., $\frac{1}{4}n^2 \leq \frac{1}{2}n(n-1) \leq \frac{1}{2}n^2$

Hence, $\frac{1}{2}n(n-1) \in \Theta(n^2)$, where c2= $\frac{1}{4}$, c1= $\frac{1}{2}$ and n_0=2

Useful Property Involving the Asymptotic Notations

The following property, in particular, is useful in analyzing algorithms that comprise two consecutively executed parts.

THEOREM: If $t1(n) \in O(g1(n))$ and $t2(n) \in O(g2(n))$, then $t1(n) + t2(n) \in O(\max\{g1(n), g2(n)\})$.(The analogous assertions are true for the Ω and Θ notations as well.)

PROOF: The proof extends to orders of growth the following simple fact about four arbitrary real numbers a1, b1, a2, b2: if $a1 \le b1$ and $a2 \le b2$, then $a1 + a2 \le 2 \max\{b1, b2\}$.

Since $t1(n) \in O(g1(n))$, there exist some positive constant c1 and some nonnegative integer n1 such that:

$$t1(n) \le c1g1(n) \text{ for all } n \ge n1.$$

Similarly, since $t2(n) \in O(g2(n))$,

$$t2(n) \le c2g2(n) \text{ for all } n \ge n2.$$

Let us denote $c3 = \max\{c1, c2\}$ and consider $n \ge \max\{n1, n2\}$ so that we can use both inequalities. Adding them yields the following:

$$\begin{aligned} t1(n) + t2(n) &\le c1g1(n) + c2g2(n) \\ &\le c3g1(n) + c3g2(n) \\ &= c3[g1(n) + g2(n)] \\ &\le c32 \max\{g1(n), g2(n)\}. \end{aligned}$$

Hence, $t1(n) + t2(n) \in O(\max\{g1(n), g2(n)\})$, with the constants c and n0 required by the definition O being $2c3 = 2 \max\{c1, c2\}$ and $\max\{n1, n2\}$, respectively.

The property implies that the algorithm's overall efficiency will be determined by the part with a higher order of growth, i.e., its least efficient part.

Therefore, $t1(n) \in O(g1(n))$ and $t2(n) \in O(g2(n))$, then $t1(n) + t2(n) \in O(\max\{g1(n), g2(n)\})$.

Basic rules of sum manipulation

$$\sum_{i=l}^{u} ca_i = c \sum_{i=l}^{u} a_i, \qquad \textbf{(R1)}$$

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i, \qquad \textbf{(R2)}$$

Summation formulas

$$\sum_{i=l}^{u} 1 = u - l + 1 \quad \text{where } l \le u \text{ are some lower and upper integer limits,} \qquad \textbf{(S1)}$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2). \qquad \textbf{(S2)}$$

Mathematical Analysis of Non-recursive Algorithms:

General Plan for Analyzing the Time Efficiency of Nonrecursive Algorithms

1. Decide on a *parameter* (or parameters) indicating an input's size.

2. Identify the algorithm's *basic operation* (in the innermost loop).

3. 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.

4. Set up a *sum* expressing the number of times the algorithm's basic operation is executed.

5. Using standard formulas and rules of sum manipulation, either find a closed-form formula for the count or, at the least, establish its *order of growth*.

Two basic rules of sum manipulation:

$$\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i,$$

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i,$$

Two summation formulas:

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

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

the formula $\sum_{i=1}^{n-1} 1 = n - 1$,
$l = 1$ and $u = n - 1$.

Example-1: To find the maximum element in the given array.

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
```

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 algorithms for loop.

- There are two operations in the loop's body:

  The comparison**: A[i]> maxval.**

  The assignment: **maxval←A[i].**

- Since the comparison is executed on each repetition of the loop and the assignment is not. Consider the comparison to be the algorithm's basic operation.
- 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

  Let us denote C (n), the number of times this comparison is executed.

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.

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

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

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

Example-2: To check whether all the elements in the given array are distinct.

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
```

The natural measure of the input's size here is again n, the number of elements in the array.

- Since the innermost loop contains a single operation (the comparison of two elements), we should consider it as the algorithm's basic operation.

- The number of element comparisons depends not only on *n.*
- limit our investigation to the worst case only.
- 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.
- The innermost loop reveals that there are two kinds of worst-case 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.

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

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

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

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

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

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

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

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

substitution: $$\therefore \sum_{i=0}^{n} i = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

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

$$= (n-1)\left[(n-1) - \frac{(n-2)}{2}\right]$$

$$= (n-1)\left[\frac{2(n-1)-(n-2)}{2}\right]$$
$$= (n-1)\left[\frac{2n-2-n+2}{2}\right]$$
$$= (n-1)\left[\frac{n}{2}\right]$$
$$= \frac{n^2}{2} - \frac{n}{2}$$
$$= \theta\left(\frac{n^2}{2}\right)$$
$$= \theta\left(\frac{1}{2}\left(n^2\right)\right)$$

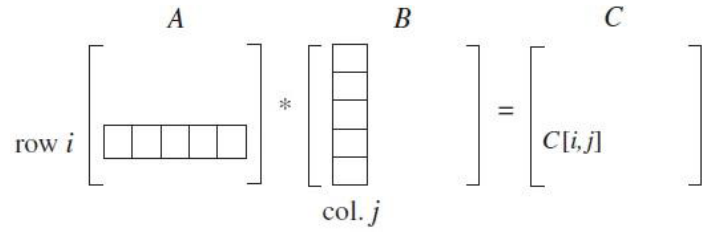
$$= \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2).$$

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

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

Example-3: To perform 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 $n \times n$ matrix whose elements are computed as the scalar (dot) products of the rows of matrix A and the columns of matrix B:



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
```

- We measure an input's size by matrix order n.

- 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.
- 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.

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

- The total number of multiplications M(n) is expressed by the following triple sum:

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

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

$$= \sum_{i=0}^{n-1} n^2 \qquad = n^3.$$

EXAMPLE 4: The following algorithm finds the number of binary digits in the binary representation of a positive decimal integer.

ALGORITHM *Binary*(n)

```
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
count ← 1
while n > 1 do
    count ← count + 1
    n ← ⌊n/2⌋
return count
```

- The most frequently executed operation here is not inside the while loop but rather the comparison n > 1 that determines whether the loop's body will be executed.
- Since the number of times the comparison will be executed is larger than the number of repetitions of the loop's body by exactly 1.

- The fact that the loop variable takes on only a few values between its lower and upper limits.
- Use an alternative way of computing the number of times the loop is executed.
- Since the value of n is about halved on each repetition of the loop.

T(n) ϵ Θlog2 *n*.

Analysis of Recursive Algorithms:

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

- Decide on a parameter (or parameters) indicating an input's size.
- Identify the algorithm's basic operation.
- 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.
- Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
- Solve the recurrence or, at least, ascertain the order of growth of its solution.

EXAMPLE 1: Compute the factorial function F(n) = n! for an arbitrary nonnegative integer n. Since $n! = 1 \cdot \ldots \cdot (n-1) \cdot n = (n-1)! \cdot n$, for n ≥ 1 and 0!= 1 by definition, we can compute $F(n) = F(n-1) \cdot n$ 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
```

Algorithm analysis

- For simplicity, we consider *n* itself as an indicator of this algorithm's input size. i.e. 1.
- The basic operation of the algorithm is multiplication, whose number of executions we denote *M(n)*. Since the function *F(n)* is computed according to the formula $F(n) = F(n-1) \cdot n$ for $n > 0$.
- The number of multiplications *M(n)* needed to compute it must satisfy the equality

M(n − 1) multiplications are spent to compute *F(n − 1),* and one more multiplication is needed to multiply the result by *n.*

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0$$

To compute F(n-1) — To multiply F(n-1) by n

Recurrence relations

The last equation defines the sequence M(n) that we need to find. This equation defines M(n) not explicitly, i.e., as a function of n, but implicitly as a function of its value at another point,

namely n − 1. Such equations are called *recurrence relations* or *recurrences*.

Solve the recurrence relation $M(n) = M(n-1) + 1,$ i.e., to find an explicit formula for M*(n)* in terms of *n* only.

To determine a solution uniquely, we need an initial condition that tells us the value with which the sequence starts. We can obtain this value by inspecting the condition that makes the algorithm stop its recursive calls:

if $n = 0$, return 1.

This tells us two things. First, since the calls stop when n = 0, the smallest value of n for which this algorithm is executed and hence M(n) defined is 0. Second, by inspecting the pseudocode's exiting line, we can see that when n = 0, the algorithm performs no multiplications.

$M(0) = 0.$

the calls stop when $n = 0$ — no multiplications when $n = 0$

Thus, the recurrence relation and initial condition for the algorithm's number of multiplications

M(n):

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0,$$
$$M(0) = 0 \quad \text{for } n = 0.$$

Method of backward substitutions

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

Therefore $M(n)=n$

Tower of Hanoi puzzle.

In this puzzle, there are ***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 smaller one. The problem has an elegant recursive solution, which is illustrated in Figure.

- If n = 1, we move the single disk directly from the source peg to the destination peg.
- 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).

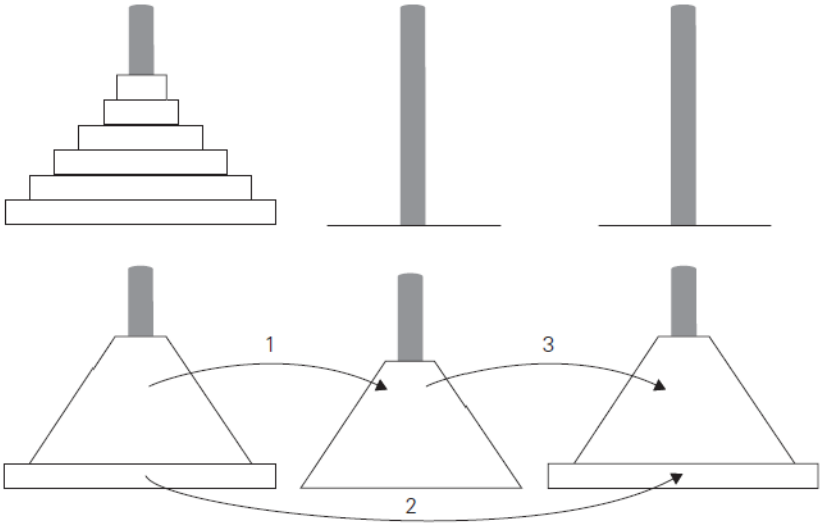


Figure: Recursive solution to the Tower of Hanoi puzzle.

```
Algorithm TowersOfHanoi(n, x, y, z)
// Move the top n disks from tower x to tower y.
{
    if (n ≥ 1) then
    {
        TowersOfHanoi(n − 1, x, z, y);
        write ("move top disk from tower", x,
        "to top of tower", y);
        TowersOfHanoi(n − 1, z, y, x);
    }
}
```

Computation of Number of Moves

The number of moves ***M(n)*** depends only on n. The recurrence equation is

$$M(n) = M(n-1) + 1 + M(n-1) \quad \text{for } n > 1.$$

We have the following recurrence relation for the number of moves M(n):

$$M(n) = 2M(n-1) + 1 \quad \text{for } n > 1$$
$$M(1) = 1.$$

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

$$M(n) = 2M(n-1) + 1$$

sub. $M(n-1) = 2M(n-2) + 1$

$$= 2[2M(n-2) + 1] + 1$$
$$= 2^2 M(n-2) + 2 + 1$$

sub. $M(n-2) = 2M(n-3) + 1$

$$= 2^2 [2M(n-3) + 1] + 2 + 1$$
$$= 2^3 M(n-3) + 2^2 + 2 + 1.$$

sub. $M(n-3) = 2M(n-4) + 1$

$$= 2^4 M(n-4) + 2^3 + 2^2 + 2 + 1$$

- Generally, after i substitutions.

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

- Since the initial condition is specified for **n = 1**, which is achieved for **i = n − 1.**

$$M(n) = 2^{n-1} M(n - (n - 1)) + 2^{n-1} - 1$$
$$= 2^{n-1} M(1) + 2^{n-1} - 1$$
$$= 2^{n-1} + 2^{n-1} - 1$$
$$= 2^n - 1$$
$$\boldsymbol{C(n) = 2^n - 1}$$

Number of binary digits in the binary representation of a positive decimal integer to count bits of a decimal number in its binary representation.

ALGORITHM *BinRec*(*n*)
//Input: A positive decimal integer *n*
//Output: The number of binary digits in *n*'s binary representation
if $n = 1$ **return** 1
else return $BinRec(\lfloor n/2 \rfloor) + 1$

- Let us set up a recurrence and an initial condition for the number of additions **A(n)** made by the algorithm.
- The number of additions made in computing **BinRec(n/2)** is **A(n/2)**, plus one more addition is made by the algorithm to increase the returned value by 1.
- The recurrence relation

$$\mathbf{A(n) = A(n/2) + 1 \text{ for } n > 1.}$$

- Since the recursive calls end when n is equal to 1 and no additions are made,
- The initial condition is **A(1) = 0**.
- The presence of n/2 in the function's argument makes the method of backward substitutions stumble on values of n that are not powers of 2.

The standard approach to solving such a recurrence is to **solve it only for n = 2k** and then take advantage of the theorem called the **smoothness rule**, which

claims that under very broad assumptions, the order of growth observed for n = 2k gives a correct answer about the order of growth for all values of n.

Now backward substitutions encounter no problems:

$$
\begin{aligned}
A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1 \\
&= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1 \\
&= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \cdots \\
&\cdots \\
&= A(2^{k-i}) + i \\
&\cdots \\
&= A(2^{k-k}) + k.
\end{aligned}
$$

Thus, we end up with

$$A(2^k) = A(1) + k = k,$$

or, after returning to the original variable $n = 2^k$ and hence $k = \log_2 n$,

$$A(n) = \log_2 n \in \Theta(\log n).$$

Chapter 2:

Brute force design technique.

Selection sort, sequential search, string matching algorithm with complexity Analysis.

"Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved".

The ―force‖ implied by the strategy's definition is that of a computer and not that of one's intellect. ―Just do it!‖ would be another way to describe the prescription of the brute-force approach. And often, the brute-force strategy is indeed the one that is easiest to apply.

As an example, consider the exponentiation problem: compute and for a nonzero number and a nonnegative integer n. Although this problem might seem trivial, it provides a useful vehicle for illustrating several algorithm design strategies, including the brute force. (Also note that computing a mod m for some large integers is a principal component of a leading encryption algorithm.) By the definition of exponentiation,

$$a^n = a * \ldots * a \quad // \text{ N times.}$$

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 Ai:

$$\underbrace{A_0 \le A_1 \le \cdots \le A_{i-1}}_{\text{in their final positions}} \;\big|\; \underbrace{A_i, \ldots, A_{min}, \ldots, A_{n-1}}_{\text{the last } n-i \text{ elements}}$$

After n − 1 passes, the list is sorted.

Here is the 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]
```

The analysis of the 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

Example of sorting with selection sort: Each line corresponds to one iteration of the algorithm, i.e., a pass through the list's tail to the right of the vertical bar; an element in bold indicates the smallest element found. Elements to the left of the vertical bar are in their final positions and are not considered in this and subsequent iterations.

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

Thus, selection sort is a Θ (n2) algorithm on all inputs. Note, however, that the number of key swaps is only Θ (n), or, more precisely, n – 1.

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 ($0 \le i \le n - 2$) of bubble sort can be represented by the following diagram:

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid \underbrace{A_{n-i} \le \cdots \le A_{n-1}}_{\text{in their final positions}}$$

Here is pseudocode of this 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 $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow 0$ **to** $n-2-i$ **do**
    if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

The action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated as an example. The number of key comparisons for the bubble-sort version given above is the same for all arrays of size n; it is obtained by a sum that is almost identical to the sum for selection sort.

89 ?↔ 45 68 90 29 34 17
45 89 ?↔ 68 90 29 34 17
45 68 89 ?↔ 90 ?↔ 29 34 17
45 68 89 29 90 ?↔ 34 17
45 68 89 29 34 90 ?↔ 17
45 68 89 29 34 17 | 90

45 ?↔ 68 ?↔ 89 ?↔ 29 34 17 | 90
45 68 29 89 ?↔ 34 17 | 90
45 68 29 34 89 ?↔ 17 | 90
45 68 29 34 17 | 89 90

etc.

First two passes of bubble sort on the list 89, 45, 68, 90, 29, 34, and 17. A new line is shown after a swap of two elements is done. The elements to the right of the vertical bar are in their final positions and are not considered in subsequent iterations of the algorithm.

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

The number of key swaps, however, depends on the input. In the worst case of decreasing arrays, it is the same as the number of key comparisons:

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

Sequential Search and Brute-Force Method.

Here we discuss two applications of this strategy to the problem of searching. The first deals with the canonical problem of searching for an item of a given value in a given list. The second is different in that it deals with the string-matching problem.

Sequential Search:

The algorithm simply compares successive elements of a given list with a given search key until either a match is encountered (successful search) or the list is exhausted without finding a match (unsuccessful search).

A simple extra trick is often employed in implementing sequential search: if we append the search key to the end of the list, the search for the key will have to be successful, and therefore we can eliminate the end of list check altogether. Here is pseudocode of this enhanced version.

ALGORITHM *SequentialSearch2*$(A[0..n], K)$

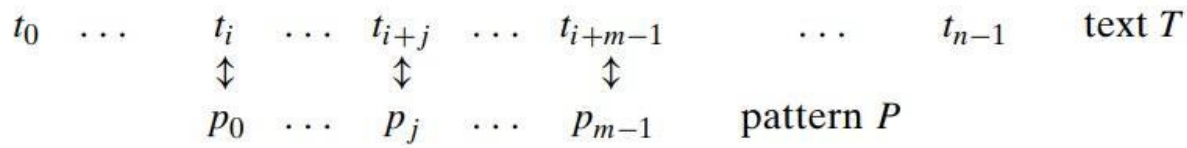
```
//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in A[0..n − 1] whose value is
//        equal to K or −1 if no such element is found
A[n] ← K
i ← 0
while A[i] ≠ K do
        i ← i + 1
if i < n return i
else return −1
```

Sequential search provides an excellent illustration of the brute-force approach, with its characteristic strength (simplicity) and weakness (inferior efficiency).

Brute-Force String Matching:

Recall the string-matching problem introduced 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:



If matches other than the first one needs to be found, a string-matching algorithm can simply continue working until the entire text is exhausted.

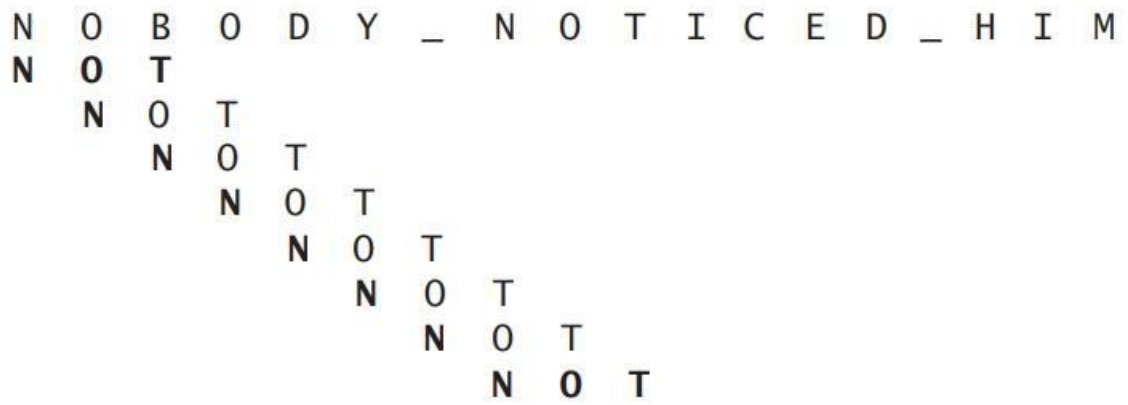
A brute-force algorithm for the string-matching problem is quite obvious: align the pattern against the first m characters of the text and start matching the corresponding pairs of characters from left to right until either all the m pairs of the characters match (then the algorithm can stop) or a mismatching pair is encountered.

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
```

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).

The worst case is much worse: the algorithm may have to make all m comparisons before shifting the pattern, and this can happen for each of the n − m + 1 tries. , in the worst case, the algorithm makes m (n − m + 1) character comparisons, which puts it in the O (nm) class.



For a typical word search in a natural language text, however, we should expect that most shifts would happen after very few comparisons (check the example again).

Q1: what are the differences between Big Oh (O), omega (Ω), and theta (θ)?

Q2. Is there a difference between algorithm, pseudocode, and program? Explain.

Q3. Define algorithm. Discuss how to analyze the algorithm.

Q4. Explain recursive and non-recursive algorithms with examples. Or, With an example, explain how recurrence equations are solved.

Q5. What are the fundamental steps to solve an algorithm? Explain. Or Describe in detail about the steps in analyzing and coding an algorithm

Q6.What are the fundamental steps to solve an algorithm? Or What are the steps for solving an efficient algorithm?