University Center Of Barika

Course

ADVANCED ALGORITHMS

Dr. Femmam Manel

Academic year : 2024-2025

Subject Information

- **Credits:** 6
- **Coefficient:** 3
- **Evaluation Method:** 60% Exam + 40% Continuous Assessment

General Course Objective

The objective of this course is to acquire the necessary concepts to analyze problems from different domains and categories in order to find solutions that can be evaluated in terms of complexity.

Prerequisite Knowledge

Basic knowledge of algorithms.

Contents

List of Figures

List of Tables

Chapter 1

Basic Concepts in Algorithms

Unit Objectives of the Course

1. Understand the definition and importance of algorithms.
2. Analyze algorithm efficiency.
3. Explore recursion and recursive algorithms.

1.1 Introduction

Algorithms are fundamental to computer science, providing structured methods for solving problems efficiently and effectively. This chapter aims to introduce the core concepts needed to understand and design algorithms, focusing on essential areas such as computational complexity, recursion, and problem-solving strategies. Mastering these concepts is crucial for analyzing algorithms, optimizing their performance, and applying them to a wide range of computational problems.

We begin with Computational Complexity and Recursion, which are foundational topics in understanding algorithm efficiency. Computational complexity deals with evaluating algorithms based on their time and space requirements, helping us determine the most efficient solutions for different types of problems. Recursion, on the other hand, provides a powerful tool for solving problems by breaking them down into simpler sub-problems, often leading to more elegant and manageable solutions.

Next, we explore the Basics of Algorithm Analysis, where you will learn how to analyze an algorithm's performance using Big O notation and other mathematical tools. This section provides the groundwork for understanding the efficiency and feasibility of different algorithms.

The chapter also introduces Classes of Problems, categorizing problems into different types, such as P, NP, and NP-complete, to understand their inherent complexity and the types of algorithms that can solve them efficiently. Moving forward, we delve into various Problem-Solving Strategies, discussing common approaches like brute force, greedy algorithms, dynamic programming, and backtracking. These strategies offer a toolkit for approaching a wide range of computational problems, each with its strengths and applicable scenarios.F̌inally, we cover the Divide and Conquer Paradigm, a powerful problem-solving paradigm that involves breaking down a problem into smaller sub-problems, solving them independently, and combining their solutions. This approach underpins many efficient algorithms, such as merge sort, quicksort, and binary search. By the end of this chapter, you will have a comprehensive understanding of the basic concepts in algorithms, equipping you with the skills to analyze, design, and implement effective algorithms across various problem domains.

1.2 Algorithms

An algorithm is a precisely defined computational procedure that accepts one or more values as input and generates one or more values as output. An algorithm is a series of computational steps that convert the input into the output. An algorithm can be regarded as an instrument for resolving a clearly defined computational issue. The problem statement delineates the expected input/output relationship in broad terms. The algorithm delineates a particular computational method for attaining that input/output relationship.

Example: The Sorting Problem

For instance, we may need to arrange a series of numbers in nondecreasing order. This issue commonly occurs in practice and offers ample opportunity to introduce various standard design techniques and analytical tools. The sorting problem is formally defined as follows:

Problem Definition

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that:

$$a'_1 \leq a'_2 \leq \cdots \leq a'_n.$$

For example, given the input sequence $\langle 31, 41, 59, 26, 41, 58 \rangle$, a sorting algorithm returns as output the sequence $\langle 26, 31, 41, 41, 58, 59 \rangle$. Such an input sequence is called an *instance* of the sorting problem.

In general, an instance of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem. Because many programs use it as an intermediate step, sorting is a fundamental operation in computer science. As a result, we have a large number of efficient sorting algorithms at our disposal. The selection of the optimal algorithm for a specific application is contingent upon various factors, including the quantity of items to be sorted.

- The degree to which the items are partially organised,
- Potential limitations on the item values,
- The computer's architecture,
- The category of storage devices utilised (primary memory, hard drives, or magnetic tapes)

Algorithm Correctness

An algorithm is said to be *correct* if, for every input instance, it halts with the correct output. A correct algorithm solves the given computational problem.

An *incorrect* algorithm, on the other hand, might not halt on some input instances or might halt with an incorrect answer. Contrary to intuition, incorrect algorithms can sometimes be useful if we can control their error rate. For example, in Chapter 31, we will explore an algorithm with a controllable error rate for finding large prime numbers. However, in most cases, we will focus only on correct algorithms.

Algorithm Specification

An algorithm can be articulated in various forms, including:

- In plain English,
- As a computer program,
- As a hardware design.

The sole prerequisite for an algorithm specification is that it must deliver an exact description of the computational procedure to be executed. The only requirement for an algorithm specification is that it must provide a precise description of the computational procedure to be followed.

1.3 What types of issues are addressed by algorithms?

Algorithms are not limited to sorting; they are used in various fields to solve complex problems. Here are some notable examples:

- **Human Genome Project**: Gene identification and DNA mapping require sophisticated algorithms for data analysis and management, leading to significant time and cost savings.
- **Internet and Information Retrieval**: Advanced algorithms optimize data routing and enhance search engine efficiency.
- **E-commerce**: Transaction security relies on cryptographic algorithms and digital signatures.
- **Resource Optimization**: Companies, such as oil or airline firms, use linear programming to maximize profits and minimize costs.
- **Specific Problems Solved by Algorithms**:
  - **Shortest Path Search**: Finding the optimal route on a road network or the Internet.
  - **Longest Common Subsequence**: DNA sequence comparison using dynamic programming.
  - **Topological Sorting**: Efficiently organizing dependencies in a manufacturing process.
  - **Convex Hull**: Computing the smallest shape enclosing a set of points in a 2D space.

These problems have multiple possible solutions, but most are not optimal. Algorithms help select the best approach and are essential in various fields such as logistics, telecommunications, and signal processing. For example, the Fast Fourier Transform (FFT) is a key technique in data compression and large-number multiplication. Algorithms play a fundamental role in efficiently solving many practical problems, optimizing performance and resources across a wide range of applications.

1.4 Computational Complexity

It is a fundamental concept in computer science that assesses the efficiency of an algorithm concerning time and space complexity. It enables us to comprehend how an algorithm's performance scales with the magnitude of the input data. This concept is essential for selecting the most suitable algorithms for various problem types, particularly in the context of extensive datasets or time-critical applications.

Let X represents an algorithm and n denotes the size of the input data; the time and space complexity of algorithm X are the primary determinants of its efficiency.

Time Factor: Time is quantified by tallying the number of key operations, such as comparisons, within the sorting algorithm.

Space Factor: Space is quantified by assessing the maximum memory capacity demanded by the algorithm. The complexity of an algorithm $f(n)$ indicates the time and/or space requirements of the algorithm relative to n, which represents the size of the input data.

1.5 Spatial Complexity

The space complexity of an algorithm denotes the quantity of memory space necessitated by the algorithm throughout its execution. The space complexity of an algorithm is the aggregate of the following two components:

1. A designated segment necessary for the storage of specific data and variables, which remains unaffected by the magnitude of the issue. For instance, utilised simple variables and constants, program size, etc.

2. A variable part refers to the memory space allocated for variables, the size of which is contingent upon the dimensions of the problem. For instance, dynamic memory allocation and recursion stack space.

The spatial complexity $S(P)$ of any algorithm P is defined as:

$$S(P) = C + S(I)$$

where C represents the constant component and $S(I)$ denotes the variable component of the algorithm, contingent upon the characteristics of instance I.

The following is a straight forward example intended to elucidate the concept: We possess three.

Here we have three variables A, B, and C and one constant. Hence,

$$S(P) = 1 + 3$$

```
Algorithm: SUM(A, B)
Step 1 -  START
Step 2 -  C ← A + B + 10
Step 3 -  Stop
```

Figure 1.1: Algorithm Sum

1.6 Time complexity

Quantifies the duration required for an algorithm to execute in relation to the input size. Common notations utilised to articulate time complexity encompass:

- **Big O** (O): Denotes the upper limit of time complexity in the worst-case scenario. For instance, $O(n)$, $O(\log n)$, $O(n^2)$, etc.
- **Big Omega** (Ω): Denotes the lower bound of time complexity in the optimal case.
- **Big Theta** (Θ): Denotes both the upper and lower limits of time complexity, offering a precise bound for average-case situations.

1.7 asymptotic analysis

Asymptotic notation is employed to characterise the growth of functions, especially in algorithm analysis. It facilitates the comparison of algorithmic efficiency by eliminating constant factors and lower-order terms.

Asymptotic notation is defined concerning functions whose domains consist of the set of natural numbers $N = \{0, 1, 2, \dots\}$. The worst-case running time function, $T(n)$, is typically defined for integer input sizes. Nonetheless, in certain instances, it may be broadened to encompass real numbers or confined to subsets of natural numbers.

Asymptotic notation is chiefly employed to characterise:

- **Worst-case running time**: e.g., insertion sort exhibits $\Theta(n^2)$ complexity.
- **Space complexity**: the quantity of memory utilised by an algorithm.
- **General function growth** independent of algorithms.

In algorithm analysis, it is essential to delineate whether we are discussing worst-case, average-case, or best-case complexity.

1.8 Asymptotic notations

1.8.1 Big-O Notation (O)

The notation $O(n)$ is the formal way to express the upper bound of an algorithm's running time. It measures the worst-case time complexity, which represents the longest amount of time an algorithm can possibly take to complete.

$$O(g(n)) = \{f(n) \mid \exists c > 0, n_0 > 0 \text{ such that } 0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0\}$$

1.8.2 Omega Notation (Ω)

The notation $\Omega(n)$ is the formal way to express the lower bound of an algorithm's running time. It measures the best-case time complexity, which represents the best amount of time an algorithm can possibly take to complete.

$$\Omega(g(n)) = \{f(n) \mid \exists c > 0, n_0 > 0 \text{ such that } 0 \leq c \cdot g(n) \leq f(n) \text{ for all } n \geq n_0\}$$

1.8.3 Theta Notation (Θ)

The notation $\Theta(n)$ is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It provides a tight bound on the growth rate of an algorithm.

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2 > 0, n_0 > 0 \text{ such that } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ for all } n \geq n_0\}$$

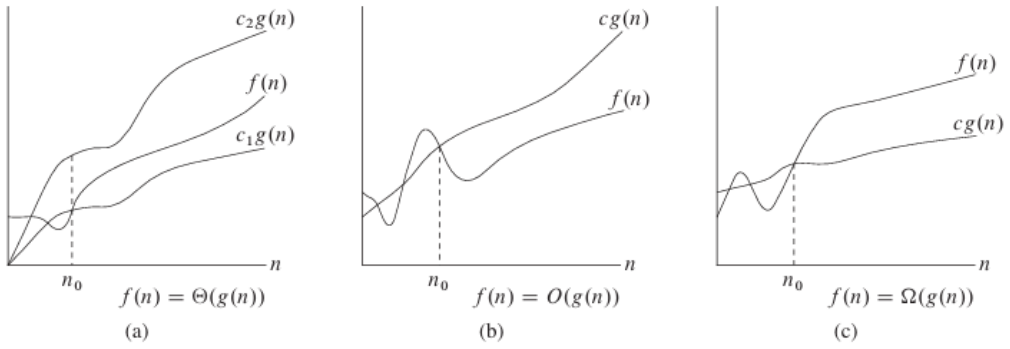


Figure 1.2: Graphic examples of the Θ, O and Ω, notations

Following is a list of some common asymptotic notations:

1.8.4 Characteristics of Algorithms

One cannot generalize a technique and call it an algorithm. A good algorithm will have these traits:

1. **Clear and unambiguous**: The algorithm must be clearly written. The process as a whole, including all of its inputs and outputs, needs to be straightforward and able to provide a singular interpretation.
2. **Well-defined inputs**: The inputs of an algorithm should be zero or more and well stated.

Complexity	Nomination
$\mathcal{O}(1)$	Constant
$\mathcal{O}(\log n)$	Logarithmic
$\mathcal{O}(n)$	Linear
$\mathcal{O}(n \log n)$	Quasi-linear
$\mathcal{O}(n^2)$	Quadratic
$\mathcal{O}(n^3)$	Cubic
$\mathcal{O}(n^{O(1)})$	Polynomial
$\mathcal{O}(2^n)$	Exponential

Table 1.1: List of some common asymptotic notations

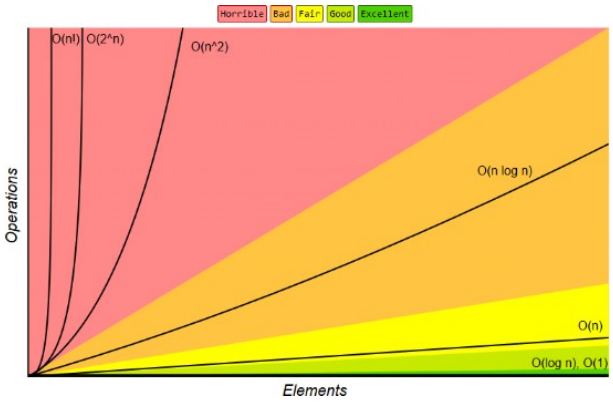


Figure 1.3: Big-O complexity chart

3. **Well-defined outputs**: An algorithm is considered successful if it produces one or more outputs that are both well-defined and consistent with the target output.
4. **Finiteness**: The algorithm must end after a certain number of steps.
5. **Practicality**: Applying the method with the resources at hand should be reasonable.
6. **Independence**: The algorithm should be independent of any specific programming language; it should have clear, concise instructions.

1.8.5 Exemple : Algorithme du tri par sélection

Algorithm 1 Tri par sélection

Un tableau A de taille n Le tableau A trié en ordre croissant

for $i \leftarrow 0$ $n-2$ **do** $min \leftarrow i$

  for $j \leftarrow i+1$ $n-1$ **do**

    if $A[j] < A[min]$ **then** $min \leftarrow j$

    if $min \neq i$ **then** échanger $A[i]$ et $A[min]$

Analyse de la complexité

- La boucle externe s'exécute $(n-1)$ fois.
- La boucle interne effectue en moyenne $\frac{n}{2}$ comparaisons par itération.
- Le nombre total de comparaisons est :

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

- Donc la complexité en temps est :

$$O(n^2)$$

1.9 Definition of Recursion

Recursion is essential and crucial in the realm of algorithms, providing a distinctive approach to problem-solving. Its importance is evident in the following aspects:

1.9.1 Essential Function in Problem Resolution

Recursion offers a sophisticated and graceful method for addressing intricate issues by articulating them as simpler sub-problems. This decomposition enables programmers to construct solutions incrementally, commencing with the simplest cases and progressively tackling more complex ones.

complex situations

1.9.2 Competence in a Variety of Complex Areas

Recursion enables algorithms to embrace a "divide-and-conquer" strategy. By breaking down a problem into smaller components and solving them independently, recursive solutions often lead to clearer and more maintainable code. This versatility is particularly advantageous in scenarios where a problem can be naturally divided into smaller, solvable sub-problems.

1.10 Fundamentals of Recursion

1.10.1 Recursive Functions

Recursive functions constitute a fundamental component of recursive algorithms. These Functions invoke themselves to address smaller instances of a problem, facilitating a more elegant and succinct representation of intricate tasks. Let us examine the essential elements:

Elementary Recursive Functions

To comprehend the notion of recursive functions, examine simple examples such as computing factorials or producing Fibonacci numbers. These examples demonstrate how a function can invoke itself to decompose a problem into more manageable sub-problems.

Example: Factorial Calculation The factorial of a number n is calculated as follows:

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

1.11 Types of Recursion

Recursion can take different forms depending on how a function calls itself. Here are the main types, each illustrated with an algorithm:

A) **Simple Recursion Definition:** A function calls itself only once during execution.

Algorithm (Factorial):

```
Algorithm Factorial(n)
    if n = 0 then
        return 1
    else
        return n * Factorial(n-1)
```

B) **Multiple Recursion Definition:** A function calls itself more than once in the same execution.

Algorithm (Fibonacci):

```
Algorithm Fibonacci(n)
    if n <= 1 then
        return n
    else
        return Fibonacci(n-1) + Fibonacci(n-2)
```

C) **Mutual Recursion Definition:** Two or more functions call each other in a circular way.

Algorithm (Even/Odd Check):

```
Algorithm IsEven(n)
    if n = 0 then
        return True
    else
        return IsOdd(n-1)

Algorithm IsOdd(n)
    if n = 0 then
        return False
    else
        return IsEven(n-1)
```

D) **Nested Recursion Definition:** The argument of the recursive call is itself another recursive call.

Algorithm:

```
Algorithm Nested(n)
    if n <= 0 then
        return 0
    else
        return Nested(Nested(n-1))
```

E) **Tail Recursion Definition:** The recursive call is the last instruction of the function.

Algorithm (Sum):

```
Algorithm SumTail(n, acc)
    if n = 0 then
        return acc
    else
        return SumTail(n-1, acc+n)
```

F) **Non-tail Recursion Definition:** The recursive call is followed by additional operations after it returns.

Algorithm (Factorial):

```
Algorithm FactorialNonTail(n)
    if n = 0 then
        return 1
    else
        return n * FactorialNonTail(n-1)
```

Table 1.2: Summary of the main types of recursion

Type of recursion	**Definition**	**Example (pseudocode)**
Simple recursion	The function calls itself once during execution.	factorial(n) = n * factorial(n-1)
Multiple recursion	The function calls itself more than once in the same execution.	fibonacci(n) = fib(n-1) + fib(n-2)
Mutual recursion	Two or more functions call each other in a circular way.	even(n) → odd(n-1) ; odd(n) → even(n-1)
Nested recursion	The argument of the recursive call is itself another recursive call.	rec(n) = rec(rec(n-1))
Tail recursion	The recursive call is the last instruction executed in the function.	sum(n, acc) = sum(n-1, acc+n)
Non-tail recursion	The recursive call is followed by additional operations.	factorial(n) = n * factorial(n-1)

1.11.1 exemple: Recursive Sum of an Array

Algorithm:

```
Function Recursive_Sum(A, n):
    if n = 0 then
        return 0   // Base case: empty array
    else
        return A[n] + Recursive_Sum(A, n-1)
```

Explanation: - If the array is empty (n = 0), return 0. - Otherwise, take the last element A[n] and add it to the sum of the first n-1 elements.

1.12 Properties of Recursion

A recursive function can go into an infinite loop like an iterative loop. To avoid infinite execution of a recursive function, it must satisfy two key properties:

1. **Base criteria**: There must be at least one base case or condition such that, when this condition is met, the function stops calling itself recursively.
2. **Progressive approach**: The recursive calls should progress in such a way that each time a recursive call is made, it gets closer to meeting the base condition.

In this example, the factorial function calls itself with a smaller value until it reaches the base case ($n = 0$ or $n = 1$), preventing infinite recursion.

Illustrating How Functions Call Themselves

Breaking down the mechanics of how functions call themselves is crucial for understanding recursion. Each recursive call operates on a smaller instance of the problem, bringing the solution closer with each iteration. Visualizing the sequence of function calls can enhance comprehension.

When Should You Use Recursion?

The issue inherently supports a divide-and-conquer strategy, whereby deconstructing it into smaller sub-problems facilitates the solution.

- **Readability:** Recursive solutions may provide more comprehensible and straightforward code for certain challenges.
- **Tree-Like Structures:** Issues pertaining to tree-like structures, such as tree traversal or Fibonacci number computation, sometimes possess attractive recursive solutions.

1.12.1 Algorithm Performance Analysis

The analysis of an algorithm provides essential insights into its performance, particularly how long it will take to solve a problem with a given set of inputs. This involves estimating the time required to address a problem with N input values.

For example, we might analyze how many comparisons a sorting algorithm performs to arrange a list of N values in ascending order or determine the number of arithmetic operations needed to multiply two $N \times N$ matrices.

There are various algorithms available to solve a problem, and analyzing these algorithms equips us with the tools needed to make informed choices. For instance, by examining the analysis of two algorithms designed to find the largest of four values, we can evaluate their efficiency and select the most suitable one for the task. When exam-

Algorithm 1

```
procedure GetLargeNumber1
    largest ← a
    if b > largest then largest ← b
    if c > largest then largest ← c
    if d > largest then largest ← d
    return largest
```

ining these two algorithms, you'll notice that both perform exactly three comparisons to find the result. Although the first algorithm is easier to read and understand, both algorithms have the same level of complexity for a computer to execute. In terms of execution time, they are equivalent, but in terms of space, the first algorithm requires more due to the temporary variable named largest. While the additional space used by this variable might seem negligible, it can be significant in cases where space is a concern.

The purpose of analyzing these values is to compare the efficiency of different algorithms for solving the same problem. For this reason, comparisons are made between algorithms of the same type—such as different sorting algorithms—rather than between algorithms of different types, such as sorting versus matrix multiplication.

The space required for temporary variables may not be critical when dealing with numbers or characters, but it can be important for larger and more complex data structures. In many modern programming languages, you can define comparison operators for complex objects or records, where the space required for temporary variables might be substantial. Although time efficiency is often the primary focus, space considerations are also important when they impact performance.

1.12.2 Purpose of Algorithm Analysis

The goal of algorithm analysis is not to provide a precise formula that calculates the exact number of seconds or computer cycles an algorithm will take to execute. Such specific measurements are impractical because they depend on numerous factors, including the type of computer, its user load, the processor type, clock speed, the complexity of the

Algorithm 2

```
procedure GetLargeNumber2
  if a > b then
    if a > c then
      if a > d then return a
      else return d
    else if c > d then return c
    else return d
  else if b > c then
    if b > d then return b
    else return d
  else if c > d then return c
  else return d
```

instruction set, and the efficiency of the compiler. All these factors affect the performance of an algorithm but are not relevant to the fundamental analysis.

Instead, algorithm analysis aims to understand the relative efficiency of algorithms regardless of specific hardware or software environments. While it may be feasible to count the exact number of operations for small or simple routines, this approach is often not practical. As the input size N becomes very large, the difference between an algorithm with N+5N operations and one with N + 2500N+2500 operations becomes insignificant. Thus, analysis focuses on the growth rate of an

algorithm's resource requirements as the input size increases, rather than on precise performance metrics tied to particular computing conditions.

1.13 Problem Classes

Some issues in computer science still have no known solution; these difficulties are categorised into groups called Complexity groups. A group of issues that have similar levels of complexity is called a Complexity Class in complexity theory. Scientists may use these categories to organise issues according to the amount of room and time needed to solve and validate the answers. The theory of computation's subfield addressing the means by which an issue may be resolved.

Common resources include time and space, which refer to the amount of time it takes for the algorithm to solve a problem and the amount of memory it uses. One way to characterise the length of time it takes to validate an answer is by looking at the time complexity of the algorithm that was used to solve the issue. How much memory is needed for an algorithm to work is described by its space complexity.

1.13.1 P-Class

A problem is in **P** if it can be solved quickly, that is, in *polynomial time*. This means the time to solve it grows at a reasonable rate as the input size increases.

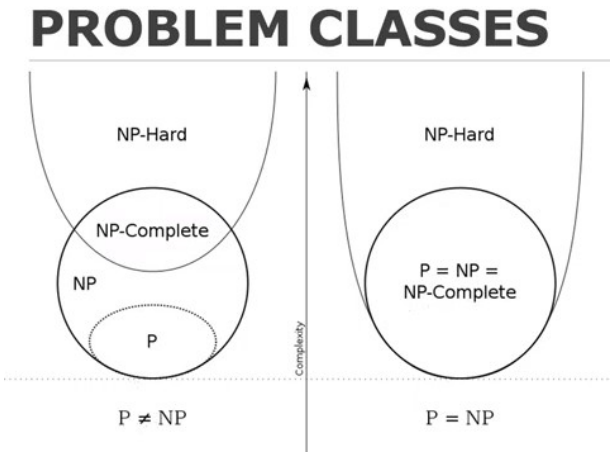


Figure 1.4: Classes of complexity

- Example: Sorting a list of n numbers can be done in $O(n \log n)$ time.
- This is considered "efficient" because even if n becomes large, the algorithm can still finish in a reasonable time.

Examples of Problems in P-Class

This class contains many problems that can be solved efficiently in polynomial time:

1. **Calculating the Greatest Common Divisor (GCD)** Using the Euclidean algorithm, the complexity is
$$O(\log \min(a, b))$$
where a and b are the input numbers.
2. **Finding a Maximum Matching in a Graph** Using the Hopcroft–Karp algorithm (for bipartite graphs), the complexity is
$$O(\sqrt{V} \cdot E)$$
where V is the number of vertices and E the number of edges.
3. **Merge Sort** A classic divide-and-conquer sorting algorithm with complexity
$$O(n \log n)$$
where n is the number of elements to be sorted.

1.13.2 NP-Class

A problem is in **NP** if, once a solution is given, we can *verify* it quickly in polynomial time, even if finding the solution is difficult.

- Example: Sudoku. Checking whether a completed Sudoku grid is correct is easy (polynomial time).
- But finding the solution from scratch may take a very long time.

.

Examples of Problems in NP-Class

The NP class contains problems for which solutions can be verified quickly, but finding the solution may take exponential time with the best-known algorithms.

1. **Boolean Satisfiability Problem (SAT)** Given a logical formula, decide whether there exists an assignment of variables that makes the formula true.

   Complexity: $O(2^n)$ in the worst case (exponential).

2. **Travelling Salesman Problem (TSP)** Given n cities and distances between them, find the shortest tour that visits each city exactly once.

   Complexity: $O(n!)$ with brute force.

   Note: There are heuristics and approximation algorithms, but no known polynomial solution.

3. **Subset Sum Problem** Given a set of integers, decide if there is a subset whose sum equals a given target.

   Complexity: $O(2^n)$ with brute force.

   Note: Dynamic programming can solve it in pseudo-polynomial time.

1.13.3 Co-NP Class

Co-NP stands for the complement of NP Class. It means if the answer to a problem in Co-NP is "No," then there is proof that can be checked in polynomial time.
Finding solutions in the **NP class** is challenging since they are being solved by a non-deterministic computer; yet, verifying these answers is trivial. In polynomial time, a Turing computer can verify NP-hard problems.

There are a lot of issues in this class that you should know how to solve:

1. The Boolean Satisfiability Problem (SAT).
2. The Hamiltonian Path Dilemma.
3. Graph Coloring Problem.

1.13.4 NP-hard Class

An NP-hard problem is at least as hard as the hardest problem in NP, and it is a class of problems such that every problem in NP reduces to NP-hard.

Characteristics :

- All NP-hard problems are not in NP. Reviewing them is a time-consuming process. This implies that verifying the correctness of a solution to an **NP-hard** issue requires a considerable amount of time. Assuming that for any issue L in NP, there is a reduction from L to A that takes polynomial time, we say that problem A is in NP-hard. One example of an NP-hard issue is:

1. The Halting Problem.
2. Boolean Formulations that are Qualified.
3. No Hamiltonian Cycle.

1.13.5 NP-complete Class

For a problem to be considered NP-complete, it must be both NP-hard and NP. The challenging issues in NP are known as NP-complete problems.
Because each issue in the NP class may be converted to an NP-complete problem in polynomial time, NP-complete problems are unique. It would be possible to solve every NP issue in polynomial time if one could solve an NP-complete problem in polynomial time. Here are a few examples of problems:

- Hamiltonian Cycle.
- Contentment.
- a vertex cover.

1.14 Problem-Solving Strategies

1.14.1 Divide and Conquer

This strategy involves breaking down a complex problem into smaller subproblems, solving each subproblem independently, and then combining their solutions to solve the original problem.

Example: The Merge Sort algorithm. This algorithm divides the array to be sorted into two halves, recursively sorts each half, and then merges the two sorted halves.

1.14.2 Dynamic Programming

This strategy solves overlapping subproblems by storing their solutions to avoid recalculating them. It is particularly useful when subproblems overlap.

Example: The Edit Distance algorithm, which calculates the minimum number of operations required to transform one string into another. The solution uses a table to store intermediate results.

1.14.3 Greedy Algorithms

The goal of developing an algorithm is to find the best possible answer to a problem. Decisions are taken inside the solution domain in the greedy algorithm technique. Because greed takes precedence, the most convenient option that seems to provide the best answer is selected. In their pursuit of a locally optimal solution, greedy algorithms may end up with globally optimal results. In most cases, nevertheless, greedy algorithms will not give solutions that are optimal on a global scale.

Example: Kruskal's algorithm for finding the Minimum Spanning Tree of a weighted graph. At each step, it selects the edge with the lowest weight that does not form a cycle.

1.14.4 Backtracking

This strategy systematically explores all possible solutions and backtracks when a partial solution does not lead to a valid overall solution.

Example: The Eight Queens Problem on a chessboard, where eight queens are placed so that no two queens threaten each other. Backtracking is used to undo an incorrect placement.

1.14.5 Branch and Bound

This strategy divides the solution space into smaller subspaces and uses bounds to avoid exploring subspaces that cannot contain an optimal solution.

Example: The Knapsack Problem. By branching possible solutions and using an upper bound on the maximum possible value, branches that cannot improve the current solution are eliminated.

1.14.6 Heuristics

Uses methods based on empirical rules or assumptions to find "good enough" solutions when exact solutions are too costly to compute.

Example: The A* algorithm for the shortest path in a graph, which uses a heuristic (such as Manhattan distance) to guide the search toward the solution.

1.14.7 Trial and Error Methods

Trying different possible solutions iteratively until a correct solution is found.

Example: Solving a logic puzzle by testing different combinations until a solution is reached.

1.14.8 Analogical Reasoning

Solving a problem by finding similarities with an already solved problem and adapting the solution of the latter to the current problem.

Example: Using the Bubble Sort algorithm as an analogy to understand other more complex sorting algorithms.

1.14.9 Linear Programming

Solving optimization problems where the constraints and objective function are linear.

Example: The Assignment Problem, where the goal is to minimize the cost of assigning tasks to agents.

1.15 Divide and Conquer Paradigm

The Divide and Conquer algorithm is a way to solve problems by breaking them up into smaller, easier-to-handle pieces and then fixing each piece on its own. separately, and then putting the two answers together to solve the original problem. It is a mathematical method that is used a lot in math and computer science.

1.15.1 Divide and Conquer Algorithm Definition

This method entails decomposing a bigger problem into smaller subproblems, resolving them separately, and subsequently amalgamating their solutions to address the original problem. The fundamental concept is recursively partitioning the problem into smaller subproblems until they become enough straightforward to be resolved immediately. Upon acquiring the solutions to the subproblems, they are thereafter amalgamated to yield the comprehensive solution.

1.15.2 Stages of Divide and Conquer Algorithm

An algorithm can be categorized into three phases: **Divide**, **Conquer**, and **Merge**.

Divide

- Deconstruct the initial problem into smaller subproblems.
- Every subproblem must encapsulate a segment of the comprehensive issue.
- The objective is to partition the problem until further subdivision is unfeasible.

Conquer

- Address each of the smaller subproblems independently.
- When a subproblem is sufficiently short, commonly known as the "base case," we resolve it immediately without additional recursion.
- The objective is to independently identify solutions for these subproblems.

Merge

- Integrate the subproblems to provide the comprehensive answer to the entire problem.
- After resolving the smaller subproblems, we iteratively amalgamate their solutions to derive the solution for the bigger problem.
- The objective is to devise a solution for the initial problem by integrating the outcomes from the subproblems.

When the subproblems are large enough to solve recursively, we call that the **recursive case**. Once the subproblems become small enough that we no longer recurse, we say that the recursion “bottoms out” and that we have reached the **base case**.

When the subproblems are sufficiently substantial to be addressed recursively, we refer to that as the **recursive case** case. When the subproblems diminish to a size that precludes further recursion, we state that the recursion "bottoms out" and that we have attained the **base case**.

In addition to solving smaller versions of the main issue, we may also be faced with subproblems that are somewhat different. Solving such subproblems is something we think about doing during the combine stage.

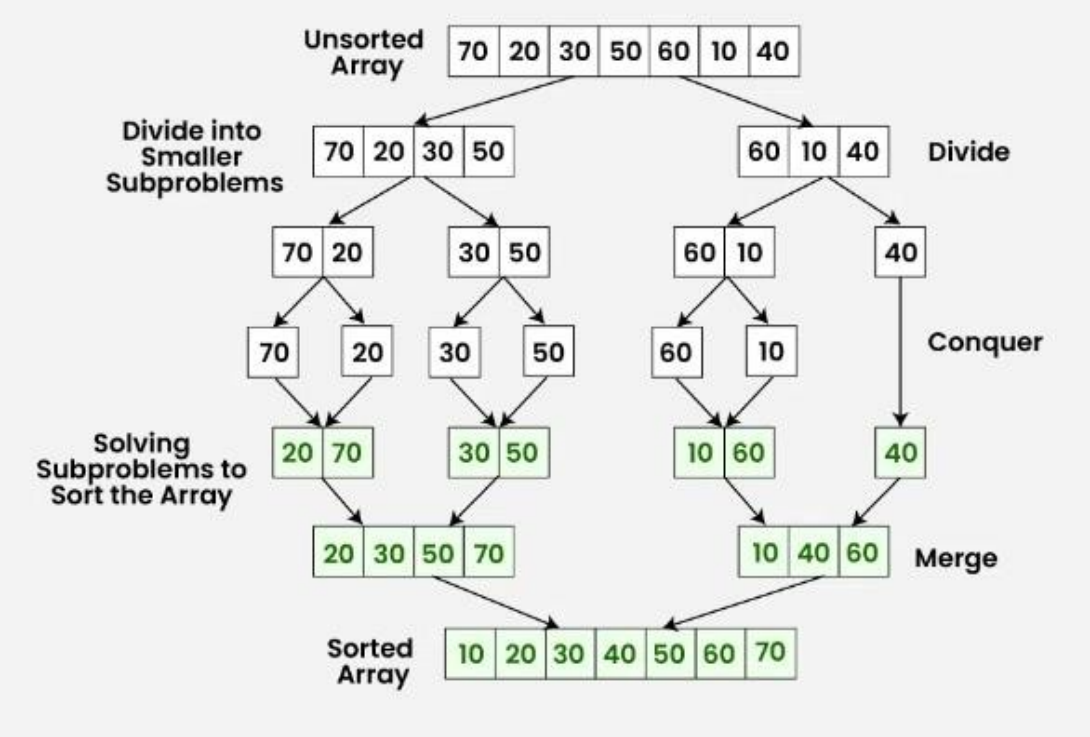


Figure 1.5: Working of divide & Conquer algorithm

Next, we will examine two approaches for multiplying matrices with dimensions $n \times n$:

- Compared to the simple approach of multiplying square matrices, the time complexity of running one is $O(n^3)$.
- Alternatively, there is **Strassen's algorithm**, which, while not as efficient as the direct approach, runs in $O(n^{2.81})$ time and eventually beats it.

1.15.3 Illustrating the Behavior of Merge Sort

Merge Sort is works in three main steps:

1. **Divide:** Split the problem into smaller subproblems.
2. **Conquer:** Solve each subproblem (recursively).
3. **Combine:** Merge the solutions of subproblems into the final answer.

Step-by-step Example:

Suppose we want to sort the array:

[38, 27, 43, 3, 9, 82, 10]

- First, we divide it into two parts: [38, 27, 43, 3] and [9, 82, 10].
- Then we divide each part again until we get single elements: [38], [27], [43], [3], [9], [82], [10].
- Now we start merging them in sorted order:

    - Merge [38] and [27] → [27, 38]
    - Merge [43] and [3] → [3, 43]
    - Merge [9] and [82] → [9, 82]
  - Continue merging:
    - Merge [27, 38] and [3, 43] → [3, 27, 38, 43]
    - Merge [9, 82] and [10] → [9, 10, 82]
  - Finally, merge the two halves: [3, 27, 38, 43] and [9, 10, 82] → [3, 9, 10, 27, 38, 43, 82].

Result: The array is now completely sorted.

Complexity Analysis

- **Time Complexity:** $O(n \log n)$ in best, average, and worst case. (This is because we always split into halves $\log n$ times, and each merge step takes $O(n)$ time.

Student-friendly summary: Merge Sort is efficient because it always divides the problem in half and then recombines the results. Even if the input is already sorted or completely unsorted, Merge Sort will take the same predictable time $O(n \log n)$.

.

1.15.4 Recurrences

In order to naturally characterise the running times of divide-and-conquer algorithms, recurrences are an integral part of the divide-and-conquer paradigm. When a function's value on smaller inputs is described by an inequality or equation, we say that the function is recursive.

The worst-case execution time of the `MERGE-SORT` procedure, denoted as $T(n)$, follows the recurrence relation:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

where $O(n)$ represents the time required for merging the subarrays.

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1; \\ 2T(n/2) + \Theta(n), & \text{if } n > 1. \end{cases}$$

whose solution we claimed to be $T(n) = \Theta(n \log n)$.

There are three approaches to solving recurrences, namely to find the solution's asymptotic Θ or O bounds:

Solving Recurrences for Merge Sort

There are three main approaches to solving recurrence relations. The recurrence for Merge Sort is:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

1. **Substitution (or Iterative Method)**
   Expand the recurrence step by step:

   $$T(n) = 2\Big(2T\left(\frac{n}{4}\right) + O\left(\frac{n}{2}\right)\Big) + O(n)$$

   $$= 4T\left(\frac{n}{4}\right) + 2O\left(\frac{n}{2}\right) + O(n)$$

   Continue expansion until the base case $T(1)$ is reached. The sum of the costs per level is $O(n)$, and the number of levels is $\log n$. Hence:

   $$T(n) = O(n \log n)$$

2. **Recursion Tree Method**
   Build a tree representing the recursive calls:

   - Level 0: cost $= n$
   - Level 1: two subproblems, each of cost $n/2$, total $= n$
   - Level 2: four subproblems, each of cost $n/4$, total $= n$

   Each level contributes $O(n)$. The depth of the tree is $\log n$. Thus:

   $$T(n) = O(n \log n)$$

3. **Master Theorem**
   Compare the recurrence with the general form:

   $$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

   Here, $a = 2$, $b = 2$, and $f(n) = O(n)$.
   Compute:

   $$n^{\log_b a} = n^{\log_2 2} = n^1 = n$$

   Since $f(n) = \Theta(n)$, we are in Case 2 of the Master Theorem:

   $$T(n) = \Theta(n \log n)$$

All three methods confirm that the time complexity of Merge Sort is:

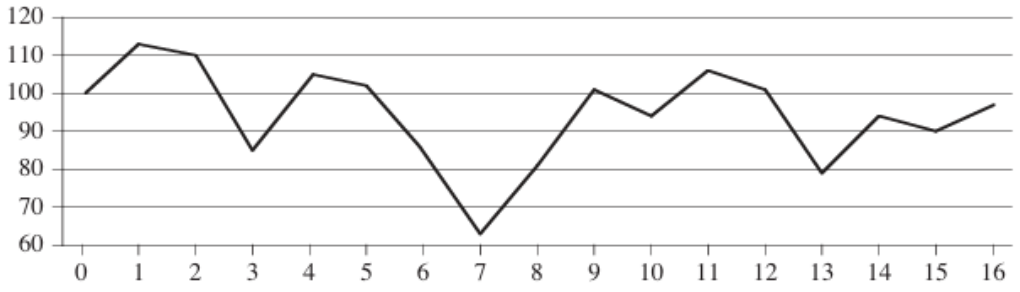
$$T(n) = \Theta(n \log n)$$

1.15.5 The Maximum-Subarray Problem

So, here's the deal: the Volatile Chemical Corporation is offering you the chance to invest. The volatile chemical corporation's stock price is just as unpredictable as the substances it manufactures. You are only permitted to purchase one unit of stock at a time and then sell it at a later date, provided that you do not buy or sell before the trading day ends. In order to make up for this limitation, you are permitted to discover the future value of the stock. The aim is to make as much money as possible.

In Figure 1.6, we can see the stock price movement over a period of 17 days. After day 0, when the price per share is \$100, you are free to purchase the shares whenever you choose. Obviously, if you want to maximise your profit, you should "buy low, sell high" — that is, purchase at the lowest possible price and then sell at the highest possible price.

Unfortunately, you might not be able to buy at the lowest price and then sell at the highest price within a given period. In Figure 1.6, the lowest price occurs after day 7, which occurs after the highest price, after day 1. You might think that you can always



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

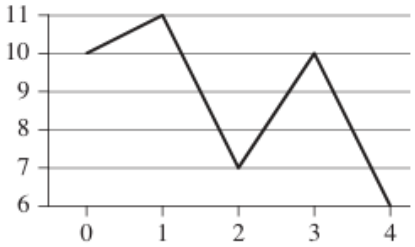
Figure 1.6: Stock Price Evolution of Volatile Chemical Corporation Over 17 Days

maximize profit by either buying at the lowest price or selling at the highest price. For example, in Figure 1.6, we would maximize profit by buying at the lowest price, after day 7. If this strategy always worked, then it would be easy to determine how to maximize profit: find the highest and lowest prices, and then work left from the highest price to find the lowest prior price, work right from the lowest price to find the highest later price, and take the pair with the greater difference. Figure 1.7 shows a simple counterexample. demonstrating Demonstrating that maximum profit can be achieved without buying at the lowest price or selling at the highest price. section

To solve the **maximum-subarray problem** using the *divide-and-conquer* technique, we consider an array $A[low \ldots high]$. The divide-and-conquer approach suggests dividing this subarray into two subarrays of approximately equal size. That is, we determine the midpoint mid and consider the subarrays $A[low \ldots mid]$ and $A[mid + 1 \ldots high]$.

- A contiguous subarray $A[i \ldots j]$ of $A[low \ldots high]$ must lie in exactly one of the following three cases:

  1. Entirely in $A[low \ldots mid]$, so that $low \leq i \leq j \leq mid$.



Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	−4	3	−4

Figure 1.7: Example Demonstrating That Maximum Profit Does Not Always Align with Lowest or Highest Price

2. Entirely in $A[mid + 1 \ldots high]$, so that $mid < i \leq j \leq high$.
3. Crossing the midpoint, so that $low \leq i \leq mid < j \leq high$.

Since a **maximum subarray** of $A[low \ldots high]$ must have the greatest sum among all subarrays in these three cases, we solve the problem recursively:

- Find the maximum subarrays of $A[low \ldots mid]$ and $A[mid + 1 \ldots high]$ recursively, since these are smaller instances of the same problem.
- Find the maximum subarray that crosses the midpoint.

Once these three cases are solved, the maximum subarray of $A[low \ldots high]$ is the one with the greatest sum among them.

Strassen's Algorithm for Matrix Multiplication

If you have seen matrices before, then you probably know how to multiply them. If $A = [a_{ij}]$ and $B = [b_{ij}]$ are square $n \times n$ matrices, then in the product $C = AB$, we define the entry c_{ij}, for $i, j = 1, 2, \ldots, n$, by:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \tag{1.1}$$

We must compute n^2 matrix entries, and each is the sum of n values. The following procedure takes $n \times n$ matrices A and B and multiplies them, returning their $n \times n$ product C. We assume that each matrix has an attribute `rows`, giving the number of rows in the matrix.

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          C[i,j] = 0
6          for k = 1 to n
7              C[i,j] = C[i,j] + A[i,k] * B[k,j]
8  return C
```

The `SQUARE-MATRIX-MULTIPLY` procedure works as follows. The for-loop of lines 3–7 computes the entries of each row i, and within a given row i, the for-loop of lines 4–7 computes each of the entries c_{ij}, for each column j. Line 5 initializes c_{ij} to 0 as we start computing the sum given in equation (1), and each iteration of the for-loop of lines 6–7 adds in one more term of equation (1).

Because each of the triply nested for-loops runs exactly n iterations, and each execution of line 7 takes constant time, the `SQUARE-MATRIX-MULTIPLY` procedure takes $O(n^3)$ time.

You might at first think that any matrix multiplication algorithm must take $O(n^3)$ time, since the natural definition of matrix multiplication requires that many multiplications. You would be incorrect, however: we have a way to multiply matrices in $o(n^3)$ time. In this section, we shall see Strassen's remarkable recursive algorithm for multiplying $n \times n$ matrices. It runs in $O(n^{\log_2 7})$ time, which we shall show in Section 4.5. Since $\log_2 7$ lies between 2.80 and 2.81, Strassen's algorithm runs in $O(n^{2.81})$ time, which is asymptotically better than the simple `SQUARE-MATRIX-MULTIPLY` procedure.

A Simple Divide-and-Conquer Algorithm

To keep things simple, when we use a divide-and-conquer algorithm to compute the matrix product $C = AB$, we assume that n is an exact power of 2 in each of the $n \times n$ matrices. We make this assumption because in each divide step, we will divide $n \times n$ matrices into four $\frac{n}{2} \times \frac{n}{2}$ matrices, and by assuming that n is an exact power of 2, we are guaranteed that as long as $n \geq 2$, the dimension $\frac{n}{2}$ is an integer.

Suppose that we partition each of A, B, and C into four $\frac{n}{2} \times \frac{n}{2}$ matrices:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

so that we rewrite the equation $C = AB$ as:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Equation (4.10) corresponds to the four equations:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \tag{4.11}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22} \tag{4.12}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \tag{4.13}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22} \tag{4.14}$$

Each of these four equations specifies two multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices and the addition of their $\frac{n}{2} \times \frac{n}{2}$ products. We can use these equations to create a straightforward, recursive, divide-and-conquer algorithm.

1.15.6 Complexity Analysis of Divide and Conquer Algorithm

The time complexity of a Divide and Conquer algorithm follows the recurrence relation:

$$T(n) = aT(n/b) + f(n)$$

where:

- n = input size
- a = amount of recursion-level subproblems
- n/b =sizes of all subproblems (with the assumption that they are all of the same size)
- $f(n)$ = cost of all operations performed outside of the recursive calls, including all operations used to partition the issue and combine its solutions.

1.15.7 Examples of Divide and Conquer Algorithms

The following computer algorithms are based on the divide-and-conquer programming approach:

1. Merge Sort
2. Quick Sort
3. Binary Search
4. Strassen's Matrix Multiplication
5. Closest Pair (Points)

There are various ways to solve any computer problem, but the mentioned algorithms are good examples of the divide and conquer approach.

Chapter 2

Dynamic Programming

Unit Objectives of the Course

1. Learn the concepts of overlapping subproblems and optimal substructure, which are essential for designing efficient algorithms.
2. Able to formulate recurrence relations, structure solutions using memoization tables, and optimize time and space complexity.

2.1 Introduction

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. Dynamic programming is both an algorithmic paradigm and a mathematical optimization technique. It was developed by Richard Bellman in the 1950s and has since been applied in various fields, such as aerospace engineering and economics. The essence of dynamic programming lies in simplifying complex problems by dividing them into smaller, more manageable sub-problems, which are then solved recursively.

While not all decision problems can be decomposed this way, those that involve decisions over multiple time points often exhibit this recursive structure. In computer science, a problem is said to have an optimal substructure if it can be solved optimally by breaking it down into sub-problems and then finding the optimal solutions to those sub- problems. When sub-problems are nested within larger problems in a way that allows for dynamic programming, there is a specific relationship between the value of the larger problem and its sub- problems.

This relationship is commonly known as the Bellman equation in optimization literature.

2.2 Definition of Dynamic Programming (AD)

One strategy in mathematics and computer science for solving difficult issues is **dynamic programming**, which involves dividing the problem into smaller, more manageable parts. To cut down on unnecessary calculations, it solves each subproblem once and stores the answers, ultimately leading to better answers for many different kinds of issues.

Separate subproblems are created, solved recursively, and then combined to answer the main issue; this technique is known as **divide-and-conquer**, as discussed in previous chapters. On the other hand, dynamic programming comes into play when there is overlap between the subproblems, or when subproblems share subsubproblems.

By addressing the same common subproblems over and over again, a divide-and-conquer algorithm in this setting wastes time. Instead of having to recalculate the solution for each subsubproblem, a dynamic programming approach can solve them at once and store the results in a table. When solving optimization issues, we usually use dynamic programming.

There are several potential answers to such issues. Every possible answer has a value, and our goal is to discover the one with the best value, whether that's the lowest or the highest. We refer to this kind of answer as an **ideal solution**, rather than the optimal one, since there could be several alternatives that all reach the optimal value.

Four Steps of Dynamic Programming

There is a four-step process that is followed while creating an algorithm for dynamic programming:

1. Outline the components of a perfect answer.
2. Find the value of the best answer iteratively.
3. Determine the worth of the best solution, usually from the ground up.
4. Build the best possible answer using the calculated data.

An issue may be solved using dynamic programming by following steps 1-3. We may skip step 4 if we only require the optimum solution's value and not the solution itself. In order to conveniently build an ideal solution, we may save extra information during step 3 when we actually conduct step 4.

2.3 Mechanism of Dynamic Programming (DP)

1. **Identify Subproblems:** Break down the main problem into smaller, overlapping subproblems that are easier to solve.
2. **Store Solutions:** Solve each subproblem once and save its solution in a table or array to reuse when needed.
3. **Build Up Solutions:** Use the stored solutions of smaller subproblems to construct the solution to the original problem.

4. **Avoid Redundancy:** By caching the solutions, DP prevents solving the same subproblem multiple times, thus minimizing redundant computations and enhancing efficiency.

2.4 Comparison of Dynamic Programming with Divide and Conquer

The divide and conquer strategy solves complex problems by breaking them into smaller, more manageable subproblems that can be solved independently. In contrast, dynamic programming efficiently handles overlapping subproblems by leveraging previously computed solutions to build up an optimal solution. While divide and conquer is a form of recursive programming, dynamic programming is typically implemented in a non-recursive manner.

With divide and conquer, subproblems are independent, meaning each subproblem is solved without reference to others, which often results in more computational effort since the solutions are not reused. This independence makes divide and conquer algorithms, such as merge sort, quicksort, and binary search, more time-consuming compared to dynamic programming methods. In dynamic programming, solutions to subproblems are interdependent, allowing the reuse of previously solved subproblems, which makes it more efficient and faster. Examples of dynamic programming applications include matrix chain multiplication and binary search tree optimization.

Dynamic programming, as the term suggests, involves a dynamic approach that adjusts based on the feasibility of solving subproblems. It is not a straightforward step-by-step process; if a solution to a particular subproblem is not feasible, then the solution to the overall problem is also unattainable. Thus, in dynamic programming, the interdependency of subproblems is crucial for solving the main problem.

The key difference between divide and conquer and dynamic programming lies in their focus: divide and conquer works on solving the entire problem by addressing each subproblem independently, whereas dynamic programming focuses on solving interrelated subproblems to achieve an overall solution efficiently.

Exemple : Computation of the Binomial Coefficient

$$C(n,k) \stackrel{\text{def}}{\equiv} \binom{n}{k} \stackrel{\text{def}}{\equiv} \frac{n!}{k!(n-k)!}$$

Without performing multiplicati

$$(a+b)^n = \sum_{i=0}^{n} C(n,i)a^i b^{n-i}$$

It is well known that C(n, k) satisfies the following recurrence: ons. The coefficient C(n, k) is called the binomial coefficient, as it appears in the binomial formula

$$(a+b)^n = \sum_{i=0}^{n} C(n,i)a^i b^{n-i}$$

It is well known that C(n, k) satisfies the following recurrence:

$$C(n,k) = C(n-1,k-1) + C(n-1,k) \quad \text{pour } n > k > 0$$
$$\text{avec } C(n,0) = C(n,n) = 1$$

To avoid calculating $C(i,j)$ multiple times, we compute each value of $C(i,j)$ only once,

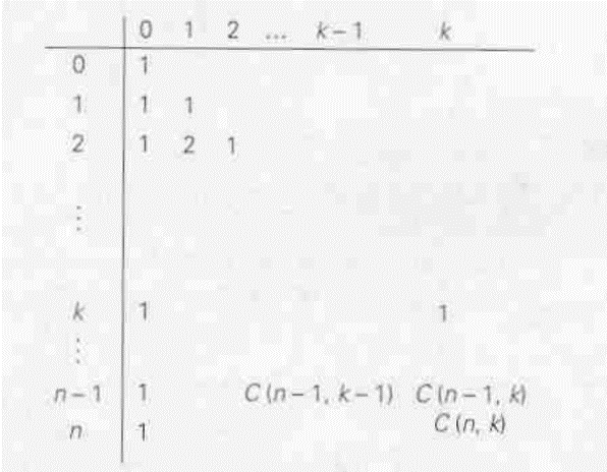


Figure 2.1: Pascal's Triangle

from bottom to top, for $0 \leq i \leq n$, $0 \leq j \leq k$. We compute the table $C(i,j)$ row by row, from left to right, starting with row $i = 0$ (and ending with $i = n$). This is Pascal's triangle.

Here is the algorithm `Binomial(n,k)` constructing this table $C(i,j)$.

ALGORITHM *Binomial(n, k)*

// Computes $C(n, k)$ by the dynamic programming algorithm
Input: A pair of nonnegative integers $n \geq k \geq 0$
Output: The value of $C(n, k)$

```
for i ← 0 to n do
    for j ← 0 to min(i, k) do
        if j = 0 or j = i then
            C[i, j] ← 1
        else
            C[i, j] ← C[i - 1, j - 1] + C[i - 1, j]
return C[n, k]
```

2.5 Analyse de l'algorithme *Binomial(n, k)*

We use addition as the basic operation. The execution time $T(n, k)$ is therefore the number of additions performed by *Binomial(n, k)* to obtain the value of the coefficient $C(n, k)$. It is the same in the worst and best cases since we have only one instance per pair (n, k). Each calculation of

$$C(i, j) = C(i - 1, j - 1) + C(i - 1, j)$$

requires 1 addition. The number of additions is given by the number of entries (i, j) that exist with $1 \leq i \leq n$ and $1 \leq j \leq k$. Rows i from 1 to k form a triangle with $\frac{k(k-1)}{2}$ entries. Rows i from $k + 1$ to n form a rectangle with $(n - k)k$ entries. Therefore, we have:

$$T(n, k) = \frac{k(k-1)}{2} + (n - k)k = nk - \frac{k^2}{2} - \frac{k}{2}$$

This is much more efficient than *BinomRec(n, k)*.
Therefore, we have:

$$T(n, k) \leq nk$$

- We also have:

$$T(n, k) = nk - \frac{k^2}{2} - \frac{k}{2} \leq nk - \frac{nk}{2} - \frac{n}{2}\frac{k}{2} = \frac{nk}{4}$$

- Thus:

$$\frac{nk}{4} \leq T(n, k) \leq nk$$

- Hence,

$$T(n, k) \in O(nk)$$

- The space used by *Binomial(n, k)* is also $O(nk)$.
- However, it is not necessary to store the entire table.
- It is enough to use a vector of k elements for the current row and update this vector from left to right.
- In this version, the space used will only be $O(k)$.

2.5.1 The Knapsack Problem

Problem Description: The 0-1 Knapsack Problem is a fundamental optimization problem. We are given:

- A set of n items, each with a *weight* w_i and a *value* v_i.
- A knapsack (bag) with a maximum capacity W.

The goal is to choose a subset of items such that:

- The total weight does not exceed W,
- The total value is maximized.

It is called the **binary** or **0-1** problem because for each item i:

$$x_i = \begin{cases} 1 & \text{if item } i \text{ is included in the knapsack} \\ 0 & \text{if item } i \text{ is not included} \end{cases}$$

Mathematical Formulation:

$$\max \sum_{i=1}^{n} v_i x_i \quad \text{subject to} \quad \sum_{i=1}^{n} w_i x_i \leq W, \quad x_i \in \{0, 1\}$$

Illustrative Example:
Suppose we have 4 items and a knapsack of capacity $W = 7$.

Item	Weight w_i	Value v_i
1	1	1
2	3	4
3	4	5
4	5	7

- If we take items (2,3): total weight $= 3 + 4 = 7$ and total value $= 4 + 5 = 9$.
- If we take items (4): total weight $= 5$ and total value $= 7$.
- If we take items (1,2,3): total weight $= 1 + 3 + 4 = 8$ (not allowed since it exceeds W).

Optimal Solution: Take items (2,3) with maximum value 9. We have n objects (items) with respective weights $w_1, \ldots, w_n$, and respective values $v_1, \ldots, v_n$.

- Each weight w_i is a non-negative integer, and the values v_i are real numbers.
- We have a knapsack that can support a maximum total weight W.
- W is a non-negative integer.
- The goal is to find the subset of objects with the maximum value whose weight does not exceed W (the capacity of the knapsack).
- This is a relevant problem for a thief.
- The brute force algorithm for this problem consists of enumerating each of the 2^n possible subsets and, for each subset, calculating the sum of the values of each item when the sum of the weights does not exceed W.
- The execution time of this algorithm is therefore $O(2^n)$.

The first step in designing a dynamic programming algorithm is to express the solution of an instance in terms of the solution of a smaller instance.

- Consider an instance consisting of the first i items, $0 \leq i \leq n$,
- with weights $w_1, \ldots, w_i$ and values $v_1, \ldots, v_i$,
- and a knapsack of capacity j, $0 \leq j \leq W$.
- Let $V[i, j]$ be the value of the optimal solution for this instance.
- Thus, $V[i, j]$ is the value of the subset with the maximum value among the first i items whose total weight does not exceed j.
- There are two possible types of subsets of the first i items whose total weight does not exceed j:
  - Those that include item i,
  - Those that do not include item i.

  Among the subsets (of the first i items) that do not include item i, the value of the optimal subset is:

  $$V[i-1, j]$$

  Among the subsets that include item i (then $j - w_i \geq 0$), the value of the optimal subset is:

  $$v_i + V[i-1, j-w_i]$$

- The maximum value of the optimal subset of the first i items is therefore the maximum of these two values.
- We thus obtain the following recurrence:

$$V[i,j] = \begin{cases} \max\{V[i-1,j], v_i + V[i-1, j-w_i]\} & \text{si } j - w_i \geq 0, \\ V[i-1,j] & \text{si } j - w_i < 0. \end{cases}$$

We also have the following initial conditions:

$$V[0,j] = 0 \quad \text{pour } j \geq 0,$$
$$V[i,0] = 0 \quad \text{pour } i \geq 0.$$

To find the value $V[n,W]$ of the optimal solution for the initial problem, it is sufficient

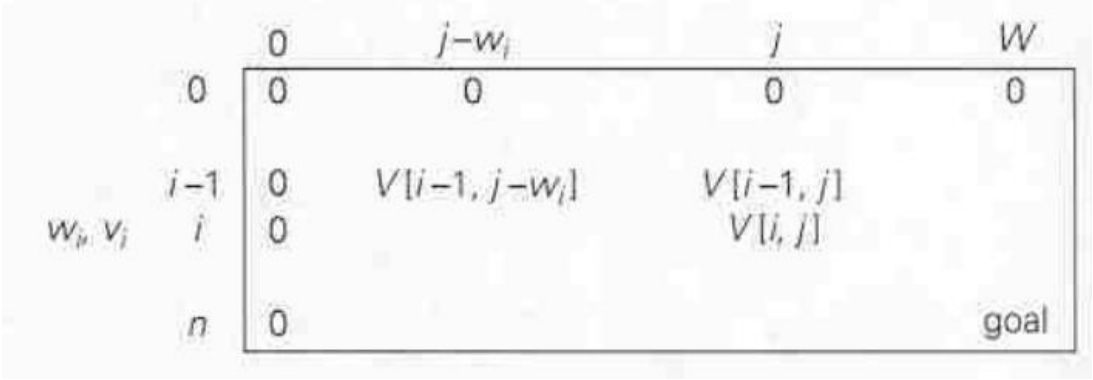


Figure 2.2: Table for solving the knapsack problem by dynamic program

to construct the table $V[i,j]$ from top to bottom using the previous recurrence:

2.5.2 DP Knapsack Algorithm

Algorithm 2 DPKnapsack$(w[1..n], v[1..n], W)$

```
// Solves the knapsack problem by dynamic programming (bottom up)
// Input: Arrays w[1..n] and v[1..n] of weights and values of n items,
//        knapsack capacity W
// Output: Table V[0..n, 0..W] that contains the value of an optimal
//        subset in V[n, W] and from which the items of an optimal
//        subset can be found
for i ← 0 to n do
    V[i, 0] ← 0
end for
for j ← 1 to W do
    V[0, j] ← 0
end for
for i ← 1 to n do
    for j ← 1 to W do
        if j − w[i] ≥ 0 then
            V[i, j] ← max{V[i − 1, j], v[i] + V[i − 1, j − w[i]]}
        else
            V[i, j] ← V[i − 1, j]
        end if
    end for
end for
return V[n, W], V =0
```

2.6 Rod cutting

Serling Enterprises buys long steel rods and cuts them into shorter rods, which are then sold. Each cut is free, and the goal is to determine the best way to cut up a rod to maximize revenue. Given a rod of length n inches and a price table p_i for $i = 1, 2, \ldots, n$, the objective is to determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

2.6.1 Example

Consider a rod of length $n = 4$. The different ways to cut it are:

- No cuts: $r_4 = p_4$
- One cut: $(2, 2) \Rightarrow r_4 = p_2 + p_2$
- Other decompositions: $(1, 3), (3, 1)$

If $p_2 + p_2 = 10$ is the highest, then this is the optimal revenue.

2.6.2 Mathematical Formulation

The maximum revenue for a rod of length n can be formulated as:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1) \tag{2.1}$$

where p_n corresponds to selling the rod without cutting, and the other terms correspond to making an initial cut at length i and recursively solving for r_{n-i}.

2.7 Optimal Substructure

If an optimal solution cuts the rod into k pieces, then the decomposition:

$$n = i_1 + i_2 + \cdots + i_k \tag{2.2}$$

provides maximum revenue:

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k} \tag{2.3}$$

This confirms that the problem exhibits **optimal substructure**, meaning optimal solutions incorporate optimal solutions to subproblems. Using dynamic programming, we can solve the rod-cutting problem efficiently by computing and storing the maximum revenue for each subproblem iteratively or recursively with memoization.

Chapter 3

Heuristic Methods

Unit Objectives of the Course

1. Introduce students to the principles of heuristic methods and their importance in solving complex problems
2. Train students to assess the performance of heuristic methods in terms of computation time, solution quality, and robustness, while considering their limitations

3.1 Introduction

In combinatorial optimization, a heuristic is an approximate algorithm that allows identifying at least one feasible solution quickly in polynomial time, though not necessarily optimal. The use of a heuristic is effective in calculating an approximate solution to a problem, thus speeding up the exact resolution process.Heuristics are techniques that may be used to obtain approximate solutions when traditional approaches don't work or to solve issues quicker when classic methods aren't working. To decide which parts of the search space to investigate and which to go over, heuristics are used in the context of local search algorithms. This may greatly improve the search's efficiency and efficacy, especially in complicated or big domains.

Generally, a heuristic is designed for a specific problem, relying on its own structure without providing any guarantee on the quality of the computed solution.

3.2 Heuristics

3.2.1 Definitions

A heuristic is a technique that improves the efficiency of a search process, possibly at the expense of accuracy or optimality of the solution.

For optimization problems (NP-complete), where finding an exact (optimal) solution is difficult (exponential cost), one may settle for:

- A satisfactory solution given by a heuristic with lower cost.

Heuristics are simple empirical rules that are not based on scientific analysis, and are different from algorithms. They are based on:

- Experience and already obtained results,
- Analogy to optimize future searches.

⇒ The solution is not optimal but rather an approximate one.

3.3 Local Search Methods

One heuristic approach to solve optimisation issues that are computationally challenging is local search in computer science. When there are several possible solutions to a problem, local search can be applied to discover the one that maximises some criterion. By implementing local modifications, local search algorithms iteratively traverse the space of potential solutions (the search space) in pursuit of an optimal solution or until a predetermined time limit has passed.

3.3.1 Working of a Local Search Algorithm

The basic working principle of a local search algorithm involves the following steps:

- **Initialization:** The first step is to produce an initial solution, which can be done either randomly or via a heuristic.

- **Evaluation:** Use an objective function or a fitness measure to assess the quality of the original answer. How near the answer is to the target result may be measured by this function.
- **Neighbor Generation:**Make small adjustments to the present solution to get a group of solutions that are nearby. These alterations are commonly known as "moves."
- **Selection:** Choose one of the neighboring solutions based on a criterion, such as the improvement in the objective function value. This step determines the direction in which the search proceeds.
- **Termination:** Continue the process iteratively, moving to the selected neighboring solution, and repeating steps 2 to 4 until a termination condition is met. This condition could be a maximum number of iterations, reaching a predefined threshold, or finding a satisfactory solution.

Several local search algorithms are commonly used in AI and optimization problems. Let's explore a few of them:

3.3.2 Hill Climbing

Artificial intelligence (AI) uses **hill climbing**, a simple optimization process, to find the optimal solution to a problem. As a member of the local search algorithm family, it finds frequent usage in optimization issues involving the identification of the optimal solution from a collection of alternatives.

Starting with a basic solution, the algorithm repeatedly tweaks it to make it better in Hill Climbing. A heuristic function assesses the present solution's quality and uses it to direct these modifications. When all possible movements have been exhausted, the process stops because it has hit a **local maximum**.

There are several variants of Hill Climbing:

- **Steepest Ascent Hill Climbing**: The algorithm considers every conceivable change to the present solution and selects the one that yields the best result.
- **First-Choice Hill Climbing**: Regardless of how good a move is, it is accepted at random if it improves the solution.
- **Simulated Annealing**: A probabilistic variant of hill climbing that sometimes accepts poorer movements to avoid local maxima.

Hill Climbing is widely used in **schedule optimization**, **route planning**, and **resource allocation**. However, it has some disadvantages, such as a lack of variety in the search space and an inclination to get stuck in local maxima. To overcome these issues and improve search results, Hill Climbing is sometimes combined with other methods like simulated annealing or genetic algorithms.

Functioning of the Hill Climbing Algorithm

Beginning at a starting point, like the base of a hill, the Hill Climbing Algorithm in artificial intelligence iteratively explores neighbouring solutions. Each iteration of the algorithm is like a mountaineer weighing the pros and cons of a new step as it is evaluated in relation to some objective function.
This function guarantees advancement by guiding the algorithm towards the top. An excellent example would be an app that helps users solve mazes. Here, the algorithm's actions represent strategic moves made within the maze with the goal of finding the shortest path to the exit. Like a mountain climber deciding which step will bring them closer to the top of a hill, the algorithm considers each possible step and determines which one is most effective in getting them closer to the exit.

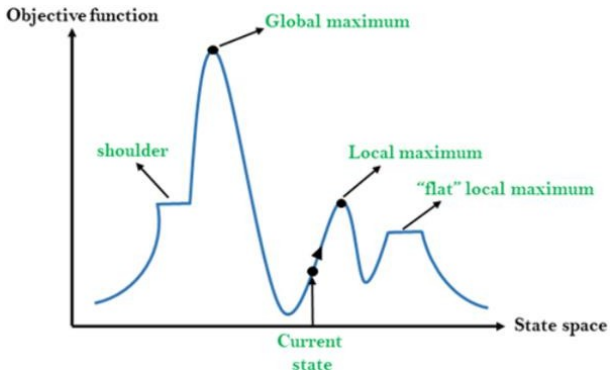


Figure 3.1: State space diagram for Hill climbing

Characteristics of the Hill Climbing Algorithm

Key characteristics of the Hill Climbing Algorithm are:

- **Generate and Test Approach:** The algorithm generates neighboring solutions and evaluates their quality, always seeking an improvement in the solution space.
- **Greedy Local Search:** It uses a straightforward strategy, prioritizing immediate gains that offer local improvements.
- **No Backtracking:** Unlike some algorithms, Hill Climbing does not revisit previous choices, continuously advancing toward an optimal solution.

Types of Hill Climbing Algorithm

The Hill Climbing Algorithm presents itself in various forms, each suitable for specific scenarios:

Simple Hill Climbing This version evaluates neighboring solutions and selects the first one that improves the current state. For example, optimizing delivery routes might pick the first alternate route that shortens delivery time, even if it's not optimal.

Algorithm:

1. Start with an initial state.

2. Check if the initial state is the goal. If so, return success and exit.
3. Enter a loop to search for a better state continuously:
    - Select a neighboring state within the loop by applying an operator to the current state.
    - Evaluate this new state:
        - If it's the goal state, return success and exit.
        - If it's better than the current state, update the current state to this new state.
        - If it's not better, discard it and continue the loop.
4. End the process if no better state is found and the goal isn't achieved.

Climbing Hill with the Steepest Ascent

This variant assesses all neighboring solutions, choosing the one with the most significant improvement. In allocating resources, for instance, it evaluates all possible distributions to identify the most efficient one.

Algorithm:

This version takes a look at all the nearby options and picks the one that's significantly better. For example, when deciding how to distribute resources, it considers every conceivable distribution and chooses the most efficient one.

Algorithm Steps

1. Assess the starting point. If it is the objective, return success; otherwise, make it the current condition.
2. Repeat until a solution is found or no further improvement is possible:
    - Set **BEST_SUCCESSOR** as the best possible improvement from the current state.
    - Evaluate each new state resulting from applying an operator to the current state:
        - If it meets the goal, return success.
        - If it is better than **BEST_SUCCESSOR**, update **BEST_SUCCESSOR**.
    - If **BEST_SUCCESSOR** is an improvement, update the current state.
3. If no solution is found and no further improvement is possible, terminate the program.

3.3.3 Local Beam Search

Local beam search is a parallelised variation of hill climbing, particularly developed to address the issue of entrapment in local optima. Local beam search starts with numerous initial solutions, sustaining a constant quantity (the "beam width") concurrently. The algorithm examines the neighbours of all these solutions and identifies the optimal solutions among them.

How Beam Search Works?

- **Beginning:** Begin by generating a number of possible initial solutions.
- **Assessment:** Check the merit of each preliminary answer.
- **Neighbor Generation:** Generate neighboring solutions for all the current solutions.
- **Selection:** Choose the top solutions based on the improvement in the objective function.
- **Termination:** Continue iterating until a termination condition is met.

Local beam search effectively avoids local optima because it maintains diversity in the solutions it explores. However, it requires more memory to store multiple solutions in memory simultaneously.

Features of Beam Search

- **Width of the Beam (W):** This value tells the program how many nodes to look at at each level. The beam width W has a direct effect on the search's spread and the number of nodes that are looked at.
- **Branching Factor (B):** If B represents the branching factor, the algorithm assesses $W \times B$ nodes at each level but advances only W for further growth.
- **Completeness and Optimality:** The limitations of beam search, caused by a constrained beam width, might hinder its capacity to identify the ideal solution by excluding potentially superior pathways.
- **Memory Efficiency:** The beam width limits the memory needed for the search, making beam search appropriate for resource-limited settings.

Example: Examine a search tree in which W equals 2 and B equals 3. At each level, only two nodes (black nodes) are chosen for continued growth based on their heuristic values.

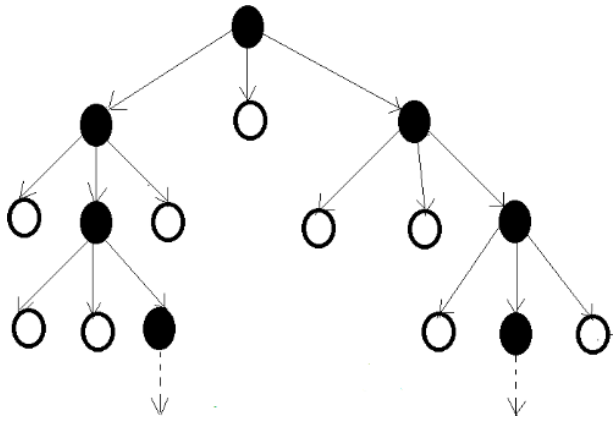


Figure 3.2: Illustration of Beam Search in a Search Tree

3.4 The A* Algorithm

The A* algorithm was designed so that the first solution found is one of the best. It was first proposed in 1968 by Peter E. Hart, Nils John Nilsson, and Bertram Raphael. It is an extension of Dijkstra's algorithm (1959).

A* is a heuristic dynamic programming algorithm that generally provides an approximate solution. It is a pathfinding algorithm in a graph from a start node to a goal node.

At each node, it uses a heuristic evaluation function to estimate the best path passing through it, and then visits nodes in order of this heuristic value.

Advantages:

- Simple algorithm,
- No preprocessing required,
- Uses little memory.

3.4.1 Properties

1. Guarantees to always find the shortest path to a goal (admissible algorithm).
2. If A* uses a heuristic that never overestimates the distance (or cost) to the goal, then A* is admissible.
3. A heuristic that makes A* admissible is called an **admissible heuristic**.
4. If the heuristic always returns zero (never an overestimate), A* behaves like Dijkstra's algorithm and always finds the optimal solution.
5. The best heuristic (though often impractical to compute) is the true minimal distance (or real cost) to the goal. A practical admissible heuristic is the *straight-line distance* (Euclidean distance) to the goal.
6. A* never examines more nodes than any other admissible search algorithm, provided the other algorithm does not use a more accurate heuristic. Thus, A* is the most efficient algorithm guaranteeing the shortest path.

3.4.2 Algorithm Description

A* builds two sets of nodes:

- The **Open list**: nodes to be examined,
- The **Closed list**: nodes already examined and belonging to the solution path.

Steps:

1. Start from the initial node $x = x_0$.
2. Explore all successors (neighbors) x' of x.

3. If x' is not in Open or Closed lists, add it to Open.
4. If x' is already in one of the lists and the new cost is smaller, update its cost. If $x' \in$ Closed, move it back to Open.
5. Select the node x in Open with the smallest evaluation function value

$$f(x) = g(x) + h(x)$$

where $g(x)$ is the cost from start to x, and $h(x)$ is the heuristic estimate from x to the goal.
6. Move x from Open to Closed.
7. Repeat until goal node is reached or Open is empty (no solution).

Usage: This algorithm is widely used due to its speed (e.g., in video games, robotics, and navigation systems).

3.5 Meta-heuristics

Optimization algorithms, generally stochastic in nature, combining several heuristic approaches.

The interest and application areas of heuristic methods (A* Algorithm, Simulated Annealing, and Genetic Algorithms) lie in optimization problems of the form:

$$\min_{x \in X} \mathbf{f(x)}$$
$$\begin{cases} \text{sous des contraintes} \\ \mathbf{g(x)} \leq \mathbf{b} \end{cases}$$

Figure 3.3: Optimization function

The function f may be a multi-objective optimization and both the objective function f and the constraints g are nonlinear.

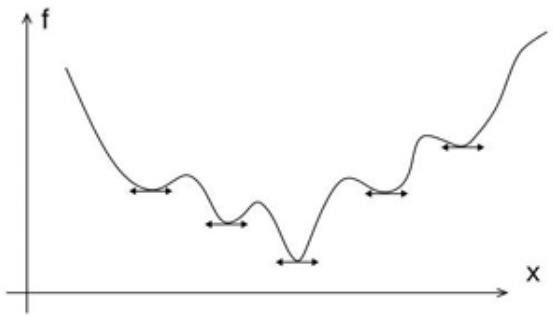


Figure 3.4: Example of optimization function

Problem: Several possible local minima ⇒ classical nonlinear optimization methods are costly and unable to capture the global solution.

Solution: Meta-heuristic methods, with the ability to escape from a local minimum and move towards a global minimum.

3.5.1 Tabu Search

Tabu Search Algorithms (TSA) are metaheuristic algorithms loosely related to evolutionary computing. They can address NP-hard problems, particularly combinatorial optimization problems. TSA reduce critical regions in the search space by applying this method. Various diversification and intensification techniques can be applied, depending on the specific nature of the problem. Different types of solutions, both within and outside the set, can be used, although better results are typically obtained by focusing on solutions within the desired set. TSA use short- term, long-term, and intermediate memory to facilitate diversification and intensification.

Algorithm 3 Tabu Search Algorithm

```
1: begin TS
2: TS_list = {}
3: S = initial solution
4: S* = S
5: repeat
6:     Find the best admissible solution S_1 in the neighborhood of S
7:     if f(S_1) > f(S*) then
8:         S* = S_1
9:     end if
10:    S = S_1
11:    Update TS_list
12: until Stopping criterion is met
13: End
```

Travelling Salesman Problem

The **Travelling Salesman Problem (TSP)** is one of the most famous problems in computational mathematics for optimization research. It is popular and simple to explain and understand, yet it is highly challenging to find an optimal solution. TSP is classified as an **NP-hard** problem, which makes it very difficult to solve all possible instances within a minimum execution time.

TSP is defined as the problem of calculating the optimal cost route in a tree or graph (undirected graph) from a source or starting node, visiting all other nodes exactly once (except the starting node), and then returning to the starting node.

TSP can be categorized into two types:

- **Asymmetric TSP:** The cost of traveling between two cities depends on the direction. For example, the cost of traveling from city X to city Y may be different from the cost of traveling from city Y to city X.

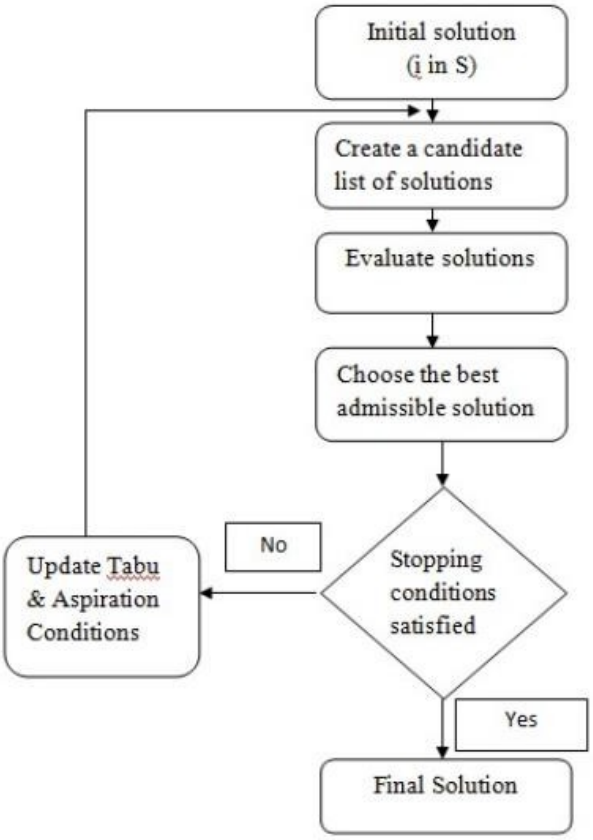


Figure 3.5: Flow chart of TSA

- **Symmetric TSP:** The cost of a link or edge is independent of the direction. The cost of traveling from city X to city Y is the same as traveling from city Y to city X.

Figure 3.6 presents an example of a symmetric TSP graph with ten nodes. The numbers on the edges represent the weights connecting the nodes.

Initial Solution

Figure 3.6 shows the initial solution where the starting node is 1. By applying the **Travelling Salesman Problem (TSP)** algorithm, the travelling process starts at node 1 and ensures that every node is visited exactly once, except for the starting node. The visiting process is designed in such a way that an optimal cost is found without forming any cycles (except for the return to the starting node). By applying local search, we can determine the longest path with an optimal cost.

After this step, **Tabu Search Algorithm (TSA)** is applied to find the optimal solution, as illustrated in Figure 3.8. In the example below, we first use a **greedy algorithm** to determine the longest traveled path to obtain the initial solution. The effectiveness of finding the optimal solution for TSP increases, as shown in Figure 3.7.

Initial solution for Symmetric TSP: Based on the heuristic tail solution, the initial solution has a cost of 81. This will be used as the input for the **Tabu Search Algorithm (TSA)**.

Tabu Search Process:

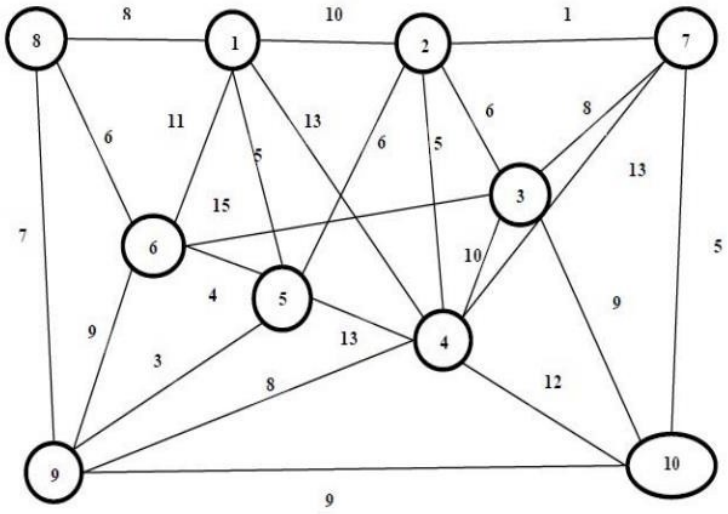


Figure 3.6: Example of a symmetric TSP graph with ten nodes.

- At the beginning, the edge (1,2) is selected for the *swap move* mechanism.
- Initially, the *tabu list* is empty. Two random edges are selected, and there are no restrictions on exchanging them.
- In the first stage, moves are performed without restrictions, and the best candidate list is selected by swapping edges.
- Afterward, two new edges are added, and the best possible solution is determined with some restrictions.
- The best *tabu list* contains nodes (1,2) and (4,5), which are swapped. These two edges are deleted and replaced with (1,4) and (2,5).
- After removing these edges, the new tour length is reduced to 77, as illustrated in Figure 3.8.

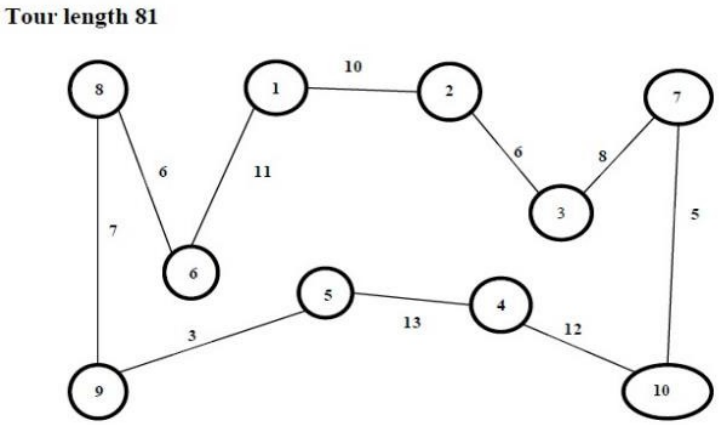


Figure 3.7: Solution by greed algorithm

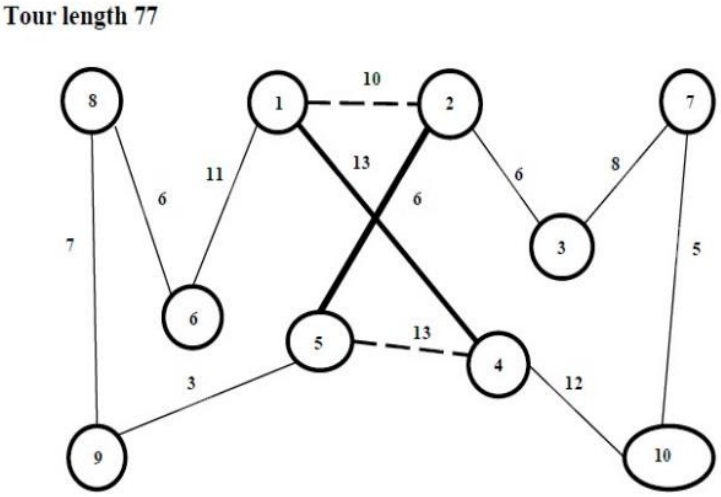


Figure 3.8: First iteration of Tabu search

If both the old and new solutions work, or if the new solution doesn't, the process continues with two more edges. Otherwise, the old edges are kept. With a few limitations, however, it provides superior solutions, and the best ones will be modified to reflect the new constraints. The iteration is repeated whenever stopping criteria is satisfied. When

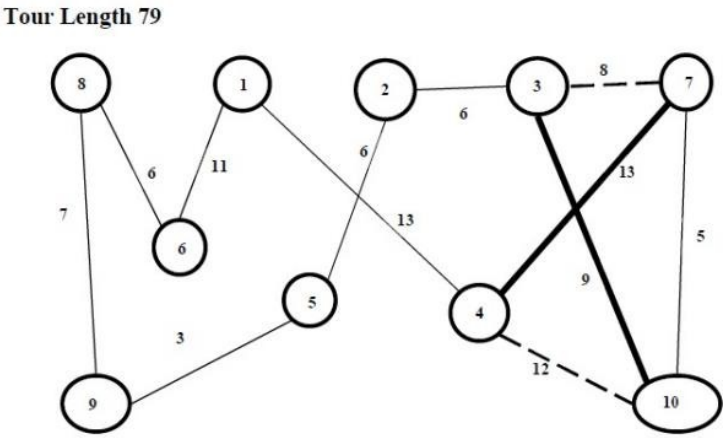


Figure 3.9: Second iteration of Tabu search

storing criteria are satisfied, the iteration is fixed before the process starts. In this method, there is no backtracking process. In the example, Figure 3.9 and Figure 3.10, the TSP considers the best solution as 77.

Complexity Analysis

The temporal complexity of Tabu Search is contingent upon the specific problem addressed and the extent of the search space, resulting in the absence of a universal calculation method.

- The worst-case time complexity of a Tabu Search problem is commonly expressed as $O(2^n)$, where n is the size of the search space. However, the magnitude of the problem can be exponential.

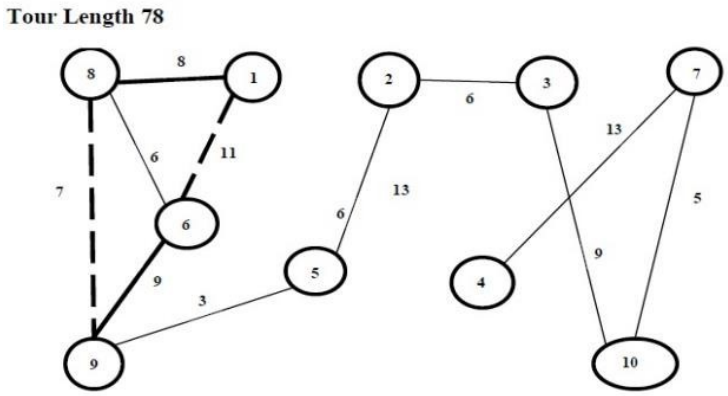


Figure 3.10: Third iteration of Tabu search

- Tabu Search's space complexity is frequently significantly lower than its worst-case time complexity, although this is contingent upon the size of the search space. The search space size is represented as $O(k \cdot n)$, with n indicating the size of the search space and k representing the size of the Tabu List or the number of candidate solutions evaluated and retained in memory.

3.5.2 Variable Neighborhood Search (VNS)

Variable Neighborhood Search (VNS) is a metaheuristic method for solving a set of combinatorial optimization and global optimization problems. It explores distant neighborhoods of the current incumbent solution and moves from there to a new one if and only if it improves the current solution. If an improvement was made, the local search method is applied repeatedly to get from solutions in the neighborhood to local optima. Variable Neighborhood Search (VNS) was designed for approximating solutions of discrete and continuous optimization problems. According to these, it aims at solving linear program problems, integer program problems, mixed integer program problems, nonlinear program problems, etc.

VNS is a recent metaheuristic that systematically exploits the idea of neighborhood changes to both reach local minima and escape the valleys that contain them.

So, the VNS metaheuristic is made up of these main parts:

- Defining a neighbourhood of the present answer;
- Making changes to the neighbourhood;
- There are three types of operations: local search, shaking (a way to change the current answer)
- updating the current solution.

A number of metaheuristics, or ways to build heuristics, build on this idea so that it doesn't get stuck in a local optimal. Gene Search, Simulated Annealing, and Tabu Search are the most well-known ones.

The following ideas are what VNS is based on:

1. What is a local minimum for one neighbourhood structure might not be a local minimum for another neighbourhood structure.

2. A global minimum is the smallest value that can be found in all possible neighbourhood structures.

3. For many problems, the local minima for one or more neighbourhoods are pretty close to each other.

To get to a local minimum, VNS uses a decline method that starts at a certain point. After that, it looks at a number of different specified areas around this answer. Whenever the local is run, one or more places in the present neighbourhood are used as starting points.

As you go down the slope, it stops at a local minimum. The search goes straight to the new local minimum if it's better than the old one. In this way, VNS is not like Simulated Annealing or Tabu Search, which are trajectory-following methods that let you make moves in the same area that don't improve things. The general ideas behind VNS and its extensions are simple and don't need many or any factors, which is different from many other metaheuristics. So, VNS not only gives very good solutions, often in easier ways than other methods, but it also explains why those solutions work so well, which can lead to practices that are more effective and complex.

Genetic Algorithms (GA)

Genetic Algorithms form a very interesting family of optimization algorithms. They were first developed by John Holland at the University of Michigan ("Adaptation in Natural and Artificial Systems", 1975).

3.5.3 Definitions

- **Individual:** a potential solution to the problem (an element of the search space).
- **Genotype or chromosome:** another way of saying "individual".
- **Gene:** a chromosome is composed of genes. In binary encoding, a gene is either 0 or 1.
- **Phenotype:** each genotype represents a potential solution to an optimization problem. The value of this potential solution is called the phenotype.

3.5.4 Principle of Genetic Algorithms

Each potential solution of a problem is encoded as a "chromosome". The set of chromosomes or "individuals" forms the "population", which evolves over time. A "generation" represents the state of the population at time t.

The population evolves across generations according to selection, crossover, and mutation rules, called **genetic operators** in computing.

In a basic GA, there are three main operators:

Genetic Operators

A. Selection: The selection operator gives "better" individuals a higher chance to participate in the next generation, according to a certain criterion called **fitness**.

B. Crossover: The goal of crossover is to combine chromosomes from parents with good traits to generate better offspring. It mixes genes from two parent chromosomes to produce two children. The population size remains constant.

- **Single-point crossover:** the genomes of both parents are cut at a single location, usually randomly chosen. The fragments are recombined to form children (see Figure 3.11).
- **Uniform crossover:** a random binary mask of the same length as the chromosome genome is used. If bit i of the mask is 1, child 1 takes bit i from parent 1 and child 2 from parent 2; if 0, the reverse is done.

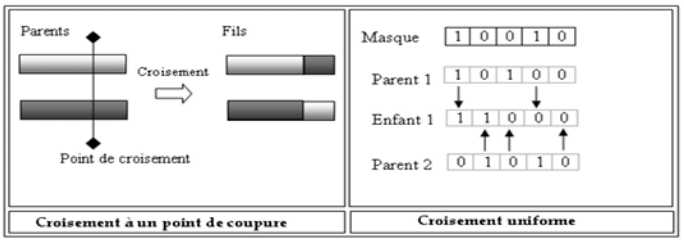


Figure 3.11: Crossover operator

C. Mutation: Modifies the genes of the children's chromosomes.

Each individual has a "fitness" value according to the problem. After crossover and mutation, a new generation is constructed by keeping individuals with a particular fitness property until convergence toward an optimal solution.

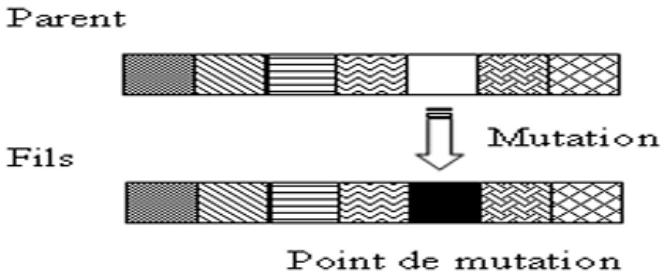


Figure 3.12: Mutation operator

3.5.5 Optimization Process

1. **Initialization:** generate a set of initial solutions.
2. **Update phase:**
   - **Evolution:**

  - Selection: choose individuals with probability proportional to their quality.
  - Reproduction: generate new individuals from this list using genetic operators.
  - Replacement: remove individuals with a probability inversely proportional to their quality.
- Update the best solution found.
- Repeat until the number of generations $\leq$ predetermined value.

3.5.6 Properties and Notes

- The objective function f can be nonlinear.
- Computation time is often significant.
- Implementation can be challenging.
- Usually provides an approximate solution.
- May encounter problems with local extrema.

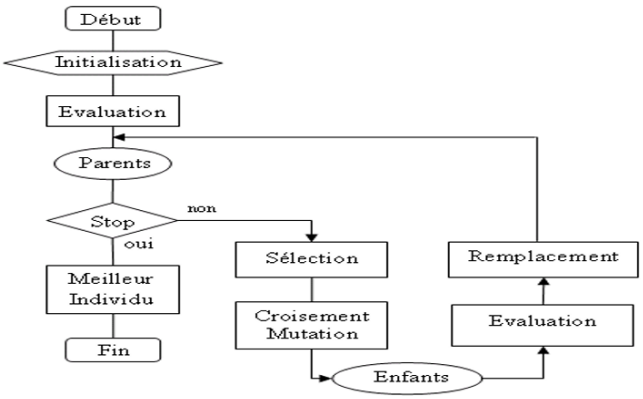


Figure 3.13: Basic principle of a Genetic Algorithm

Chapter 4

Stochastic programming

Unit Objectives of the Course

1. To equip them with advanced techniques for managing and optimizing systems under uncertainty and variability.
2. This enables them to design more robust and adaptive models capable of handling unexpected conditions in applications such as planning, forecasting, and machine learning.

4.1 Introduction

When there are unknown factors in a mathematical optimisation problem with stochastic programming, there are many possible answers. For this, a model might have to be put through a number of steps, and each step might be affected by a different set of factors. Mathematicians can use this to help them make decisions, divide up resources, and do other similar tasks. It is also studied in schools, where experts are working on making new random programming models that work better in the real world.
It's possible for optimisation problems to get very complicated. When the form is easier, all the factors are known, so they can be put through an equation to find the best answer. Many times, this isn't possible when parameters aren't completely clear and unknown factors could change the result. Probability distributions help stochastic programmers figure out the range of factors and use them in the solution.
There are common cases that can be found in mathematical models of nature events. For instance, when butterflies lay eggs, they want the best chance that the eggs will hatch and grow into larvae, which will then turn into adult butterflies. A model of random programming can help us figure out the best set of choices the butterfly could make. Predation, changes in weather, and other things that stop eggs from hatching or kill larvae before they become adults are examples of variables. To solve the problem in the best way, the scientist can follow a set of steps.
At each step, choices can either limit or expand the choices at the next step. To find the best answer, stochastic programming needs to be able to adapt to different situations while still requiring choices to be made in a certain way so that they can be numbered in a math problem. The level of difficulty may vary depending on the type of problem; some are easy and only have two steps, while others may have more than that. It's possible to find the best answer for each step and think about how it will affect decisions further down the line.

4.2 Stochastic Programming

In the field of mathematical optimisation, stochastic programming is a way to model optimisation problems that have unknown outcomes. It is an op to have a fuzzy program.

an optimisation problem where some or all of the problem factors are unknown but the probability distributions are known. If you compare this method to deterministic optimisation, all the problem parameters are taken to be known accurately. In stochastic programming, the goal is to make a choice that both meets some specific criteria picked by the person making the choice and takes into account the fact that the problem parameters are unclear. Because many choices in the real world are based on unknowns, stochastic programming has been used in many fields, from banking to transportation to energy efficiency.

Mathematical Programming, alternatively known as Optimization, focuses on decision-making processes by formulating and solving mathematical models to find the best possible decisions. Stochastic Programming, a subset of Mathematical Programming, extends this concept to decision-making under uncertainty. In Stochastic Programming, the decision models incorporate random parameters to account for uncertainty in the problem's parameters, effectively addressing situations where some factors are not known with certainty. This approach can be seen as Mathematical Programming with the addition of randomness in its parameters to better handle real-world scenarios where uncertainty plays a significant role.

4.2.1 Stochastic Programming Methods

Scenario-Based Methods

- **Description:** These methods use scenarios to represent different possible realizations of random variables. The idea is to generate a set of representative scenarios and then solve an optimization problem based on these scenarios.
- **Example:** Sample Average Approximation (SAA). This method involves creating a sample of scenarios and solving the optimization problem on this sample to approximate the solution of the full stochastic problem.

Stochastic Integer Programming

- **Description:** This method is used for problems where some of the decision variables must be integers. It incorporates uncertainty into optimization problems where discrete variables play a significant role.
- **Example:** Supply chain management where certain decisions, such as the number of vehicles or warehouses, must be integer values.

Chance Constrained Programming

- **Description:** This approach deals with constraints that must be satisfied with a given probability. Instead of ensuring constraints are met in all scenarios, it allows for some level of constraint violation.

- **Example:** If a constraint requires storage capacity not to be exceeded, chance constrained programming allows exceeding this capacity with a specified probability.

Stochastic Dynamic Programming

- **Description:** This method is used for problems where decisions are made in multiple stages, and each future decision may be influenced by random realizations up to that point. Stochastic dynamic programming solves these problems using decision principles at each stage.
- **Example:** Production planning where production decisions need to be adjusted based on uncertain future demands.

Markov Decision Process

- **Description:** Markov Decision Processes (MDPs) are a class of stochastic decision problems where the future state of the system depends only on the current state and chosen action, not on past states. They are used to model and solve sequential decision problems.
- **Example:** Inventory management where ordering decisions depend on the current inventory level and future demand probabilities.

Benders Decomposition

- **Description:** Benders decomposition is a decomposition method for solving optimization problems with decision variables divided into two groups, where one group is optimized conditionally on the solution of the other group. It is often used for stochastic problems with linear and continuous constraints.
- **Example:** Benders decomposition can be used for resource allocation problems with capacity constraints where the main decision variables are decomposed to facilitate solution.

4.2.2 Two-Stage Problem Definition

A two-stage stochastic programming problem is an optimization problem where decisions are made in two stages.

First Stage

In this initial stage, decisions are made before the uncertainty is revealed. These decisions are often referred to as "here-and-now" decisions and are typically based on the available information at that time. The goal is to choose these decisions to optimize an objective function while considering the uncertainty that will be revealed later.

Second Stage

After the uncertainty is revealed, additional decisions are made to respond to the realized outcomes. These decisions, known as "wait-and-see" decisions, are based on the actual realization of the uncertain parameters. The objective here is to minimize the cost or maximize the benefit considering both the first-stage decisions and the new information.

4.2.3 Mathematical Formulation

A two-stage stochastic programming problem can be formulated as follows:

$$\min_{x \in X} \{g(x) = f(x) + E_\xi[Q(x, \xi)]\}$$

where $Q(x, \xi)$ is the optimal value of the second-stage problem

$$\min_y \{q(y, \xi) \mid T(\xi)x + W(\xi)y = h(\xi)\}.$$

The classical two-stage linear stochastic programming problems can be formulated as

$$\min_{x \in R^n} g(x) = c^T x + E_\xi[Q(x, \xi)]$$

subject to

$$Ax = b$$

$$x \geq 0$$

where $Q(x, \xi)$ is the optimal value of the second-stage problem

$$\min_{y \in R^m} q(\xi)^T y$$

subject to

$$T(\xi)x + W(\xi)y = h(\xi)$$

$$y \geq 0$$

n such formulation, $x \in R^n$ is the first-stage decision variable vector, $y \in R^m$ is the second-stage decision variable vector, and $\xi \in (q, T, W, h)$ contains the data of the second-stage problem.

At the first stage, we have to make a "here-and-now" decision x before the realization of the uncertain data ξ, viewed as a random vector, is known. At the second stage, after a realization of ξ becomes available, we optimize our behavior by solving an appropriate optimization problem.

At the first stage, we optimize (minimize in the above formulation) the cost $c^T x$ of the first-stage decision plus the expected cost of the (optimal) second-stage decision. We can view the second-stage problem simply as an optimization problem, which describes our supposedly optimal behavior when the uncertain data is revealed, or we can consider its solution as a recourse action.

The term Wy compensates for a possible inconsistency of the system $Tx \leq h$ and $q^T y$ is the cost of this recourse action.

The considered two-stage problem is linear because the objective functions and the constraints are linear. Conceptually, this is not essential, and one can consider more general two-stage stochastic programs. For example, if the first-stage problem is integer, one could add integrality constraints to the first-stage problem so that the feasible set is discrete. Non-linear objectives and constraints could also be incorporated if needed.

4.3 The Farmer's Problem

On 500 acres of land, a farmer grows corn, wheat, and sugar beets. During the off-season, he wants to choose how much land to give each crop.

- For cow feed, at least 200 tonnes of wheat and 240 tonnes of maize are required.
- Those that aren't grown on a farm can be bought from a wholesaler.
- Any extra grain that isn't needed to feed cattle can be sold for:
  - \$170 per tonne of wheat
  - \$150 per tonne of maize
- The seller charges 40% more for the grain, leading to prices of:
  - \$238 per tonne of wheat
  - \$210 per tonne of maize
- You can sell up to 6,000 tonnes of sugar beets for \$36 per tonne, and any extra can be sold for \$10 per tonne.

Crop results are hard to predict because they depend on the weather during the growing season. There are three possible outcomes: *good*, *fair*, and *bad*. All three are equally likely to happen. . (In this data, only the yields are scenario-dependent, while in reality, the purchase prices and sales revenues from grain would be higher in years with poor yield, etc.) A farmer has 500 acres of land and must decide how much land to

Scenario	**Wheat yield (tons/acre)**	**Corn yield (tons/acre)**	**Beet yield (tons/acre)**
1. Good	3	3.6	24
2. Fair	2.5	3	20
3. Bad	2	2.4	16

Table 4.1: Crop Yield under Different Scenarios

allocate for wheat, corn, and sugar beets. The decision needs to take into account:

- **Demand:** 200 tons of wheat and 240 tons of corn are required for cattle feed.

- **Sales and Purchases:**
  - Any extra wheat and corn can be sold.
  - If the required amount is not met, the farmer can purchase from a wholesaler at a higher price.
- **Sugar Beets:** Up to 6000 tons can be sold at $36 per ton, and any additional amount is sold at $10 per ton.

Uncertainty

Crop yields are uncertain and depend on three scenarios: "good," "fair," and "bad," with equal probabilities. The farmer needs to decide how much land to allocate without knowing which yield scenario will occur.

Mathematical Formulation

In two-stage stochastic programming, the first stage involves decisions made before knowing the uncertainty, and the second stage adjusts the decisions based on the actual scenario that occurs.

$$Z = \min c^T x + \sum_{k=1}^{K} P_k q_k^T y_k \tag{4.1}$$

Subject to:

$$T_k x + W y_k = h_k, \quad k = 1, \ldots, K; \tag{4.2}$$

$$x \in X \tag{4.3}$$

Decision Variables

First-stage:

$$x_1 = \text{acres of land raised for wheat}$$
$$x_2 = \text{acres of land that was grown with maize}$$
$$x_3 = \text{acres of land cropped with beets}$$

In the first stage, the farmer decides how to allocate the 500 acres among the three crops, considering future uncertainty about the yields.

Second-stage: These decisions are made after uncertainty is revealed (after the actual yields are known, depending on the scenario—"good," "fair," or "bad"):

$$
\begin{aligned}
w_1 &= \text{tons of wheat sold} \\
w_2 &= \text{tons of corn sold} \\
w_3 &= \text{tons of beets sold at \$36/T} \\
w_4 &= \text{tons of beets sold at \$10/T} \\
y_1 &= \text{tons of wheat purchased} \\
y_2 &= \text{tons of corn purchased}
\end{aligned}
$$

$$
\begin{aligned}
\min \quad & 150x_1 + 230x_2 + 260x_3 + 3\sum_{K=1}^{3} Q_K(x) && (4.4)\\
\text{s.t.} \quad & x_1 + x_2 + x_3 \leq 500 && (4.5)\\
& x_j \geq 0, \quad j = 1, 2, 3 && (4.6)
\end{aligned}
$$

where $Q_K(x)$ is the optimal solution of the second-stage (recourse) problem after the scenario has been determined, given that the first-stage variables x have been selected.

Recourse

Scenario 1 (Good)

$$
\begin{aligned}
Q_1(x) = \min \quad & -170w_1 - 150w_2 - 36w_3 - 10w_4 + 238y_1 + 210y_2 && (4.7)\\
\text{s.t.} \quad & y_1 - w_1 \geq 200 - 3x_1 && (4.8)\\
& y_2 - w_2 \geq 240 - 3.6x_2 && (4.9)\\
& w_3 + w_4 \leq 24x_3 && (4.10)\\
& y_1 \geq 0, \quad y_2 \geq 0, \quad w_1 \geq 0, \quad w_2 \geq 0, \quad 0 \leq w_3 \leq 6000, \quad w_4 \geq 0 && (4.11)
\end{aligned}
$$

Scenario 2 (Fair)

$$
\begin{aligned}
Q_2(x) = \min \quad & -170w_1 - 150w_2 - 36w_3 - 10w_4 + 238y_1 + 210y_2 && (4.12)\\
\text{s.t.} \quad & y_1 - w_1 \geq 200 - 2.5x_1 && (4.13)\\
& y_2 - w_2 \geq 240 - 3x_2 && (4.14)\\
& w_3 + w_4 \leq 20x_3 && (4.15)\\
& y_1 \geq 0, \quad y_2 \geq 0, \quad w_1 \geq 0, \quad w_2 \geq 0, \quad 0 \leq w_3 \leq 6000, \quad w_4 \geq 0 && (4.16)
\end{aligned}
$$

Scenario 3 (Bad)

$$
\begin{aligned}
Q_3(x) = \min \quad & -170w_1 - 150w_2 - 36w_3 - 10w_4 + 238y_1 + 210y_2 && (4.17)\\
\text{s.t.} \quad & y_1 - w_1 \geq 200 - 2x_1 && (4.18)\\
& y_2 - w_2 \geq 240 - 2.4x_2 && (4.19)\\
& w_3 + w_4 \leq 16x_3 && (4.20)\\
& y_1 \geq 0, \quad y_2 \geq 0, \quad w_1 \geq 0, \quad w_2 \geq 0, \quad 0 \leq w_3 \leq 6000, \quad w_4 \geq 0 && (4.21)
\end{aligned}
$$

The application of stochastic programming to the farmer's problem provides a robust method for decision-making under uncertainty, allowing for more efficient resource management and better adaptation to variations in crop yields.

Stochastic programming is a powerful approach for solving optimization problems under uncertainty. By incorporating random variables and probabilistic distributions into decision-making models, it allows for effective risk management and the ability to account for future uncertainties. Unlike deterministic methods, stochastic programming provides more robust and realistic solutions, making it applicable in fields such as finance, resource management, energy, and industrial production.

However, it often requires sophisticated computational techniques, such as decomposition algorithms, and can be more complex to implement. In conclusion, stochastic programming is an essential tool for decision-making in uncertain environments, although it involves a trade-off between solution accuracy and computational complexity.

Bibliography

[1] BeyondVerse. *Understanding Recursion: A Key Concept in Algorithms*. Pearson Education, 2023.

[2] BeyondVerse Follow. Understanding recursion: A key concept in algorithms, 2023. Consulté le 21 mars 2025.

[3] Dr. LOUNNAS Bilal. *A LGOR I THMS ANA LYS I S*. Université Mohamed Boudiaf, Msila, Algeria, 2024. Polycopié de cours, Master 1 Informatique.

[4] John R Birge and Francois Louveaux. *Introduction to stochastic programming*. Springer Science & Business Media, 2011.

[5] Sébastien Giguère. *Algorithmes d'apprentissage automatique pour la conception de composés pharmaceutiques et de vaccins*. PhD thesis, Université Laval, 2015.

[6] Pierre Hansen, Nenad Mladenović, and Jose A Moreno Perez. Variable neighbourhood search: methods and applications. *Annals of Operations Research*, 175:367–407, 2010.

[7] Himanshi Singh. What is hill climbing algorithm in ai, 2024. Consulté le 20 mars 2025.

[8] Donald Ervin Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

[9] Charles E Leiserson et al. Introduction to algorithms. *CS 312 Lecture 23*, 26, 1999.

[10] José Andrés Moreno Pérez, Nenad Mladenović, Belén Melián Batista, and Ignacio J García del Amo. Variable neighbourhood search. *Metaheuristic Procedures for Training Neutral Networks*, pages 71–86, 2006.

[11] Vishnu Kumar Prajapati, Mayank Jain, and Lokesh Chouhan. Tabu search algorithm (tsa): A comprehensive survey. In *2020 3rd International Conference on Emerging Technologies in Computer Engineering: Machine Learning and Internet of Things (ICETCE)*, pages 1–8. IEEE, 2020.