People's Democratic Republic of Algeria

Ministry of Higher Education and Scientific Research

University of Oum El Bouaghi

Faculty of Exact Sciences and Natural and Life Sciences

Department of Mathematics and Computer Sciences

Course handout

Algorithmic methods

Level: Master 1 in Computer Sciences – option: distributed architectures

Dr. Zakaria Laboudi

Preface

This document is a course material concerning the subject entitled *Algorithmic Methods*, taught in the Department of Mathematics and Computer Sciences at University of Oum El Bouaghi and intended for students in the first year of the Master's cycle – option: Distributed architectures. This subject follows the courses of *Algorithms and data structures*, taught in the first and second years of the license cycle.

This course aims to present various fundamental algorithmic methods for solving many typical problems. Indeed, the resolution of problems can in many times be done using general outlines called paradigms which have the advantage of being applicable to a large number of situations. Among these methods, this course focuses on famous principles of *greedy methods*, *divide and conquer algorithms*, *dynamic programming*, *backtracking algorithms*, *probabilistic methods* and *approximation algorithms*. The objective is to give an overview of these paradigms in as simple and clear a manner as possible while exposing their theoretical foundations and elements as well as how to apply them to the resolution of various typical problems.

In order to achieve the above-stated objective, several efforts have been made to approach this work in different aspects by synthetizing relevant information based on various sources (books, course notes, websites, etc.) while respecting the official program set by the Ministry of Higher Education and Scientific Research. Nevertheless, we are aware that this work remains partial, truncated and not exhaustive. This is why we try to ensure a permanent update so as to enrich its content. Thus, we would be grateful if readers would point out any errors or provide us suggestions in this regard.

Oum El Bouaghi, June 2023

Dr. Zakaria Laboudi

Table of contents

Chapter I : Greedy methods

Chapter II : divide-and-conquer method

Chapter III : Dynamic programming

Chapter IV : Backtracking

Chapter V : Probabilistic methods

Chapter VI : Approximation algorithms

Exercices

Bibliography

Chapter I: Greedy methods

Objective:

This chapter aims to present the paradigm of ***greedy methods*** and to show some of their application aspects to solve certain typical examples. This chapter also discusses some elements of the greedy strategy as well as the theoretical underpinnings of this problem-solving paradigm.

1. Optimization problems

1.1. Definition

An optimization problem is characterized by a non-empty set of admissible solutions X and an objective function f (also called ***cost*** function) which associates to each solution $s \in X$ a value $f(s)$. Solving an optimization problem consists in finding a solution $s^* \in X$ which optimizes (i.e. which maximizes or minimizes) the value of cost function f. In this case, solution s^* is called *optimal solution*. In some cases, the optimum is not unique, but rather a set of solutions optimizing the desired objective.

1.2. Methods for solving optimization problems

Generally, methods for solving optimization problems are divided into two main families:

- **Exact methods:** they allow finding the optimal solutions. However, time complexity often increases exponentially with the growth of the solutions space size; this makes this type of resolution unsuitable for large problem instances.
- **Approximate methods:** they do not necessarily allow obtaining optimal solutions for a given problem but rather to seeking solutions close to the optimum within a reasonable execution time. They belong to two subfamilies:
  - **Methods without guarantee:** they build solutions at a lower cost without any guarantee of their quality, hoping that they will perform well. We distinguish two types of solving-methods: *heuristics*, which are used in solving specific problems and *meta-heuristics*, which designate a general framework for solving several classes of optimization problems.
  - **Methods with guarantee:** they allow building solutions whose quality is guaranteed at a lower cost: *approximation algorithms* (see chapter 6).

2. Overview on greedy algorithms

2.1. Principle of the greedy method

In many situations, solving an optimization problem involves building a solution gradually such that at each step, a certain number of choices are made. In other words, starting from an incomplete solution, the resolution process works to complete it step by step, where at each step, we deal with some of the variables over which we no longer return (i.e. by making definitive choices). In this way, ***the greedy principle*** builds a solution in an incremental manner where at each step, the most promising direction (i.e. the choice which seems the best at that time) is taken by following very simple rules. At each step, only one datum is considered without worrying about the consequences in the future, and without going back. Thus, this principle consists in making a locally optimal choice in the hope that it will lead to the globally optimal solution. By doing so, a local choice leads to an analogous sub-problem of smaller size.

Although a local choice seems the best in the short term, it nevertheless does not always lead to a globally optimal solution. In other words, local optimality does not necessarily lead to global optimality. Consequently, a greedy method generates a solution according to two possible scenarios. In the case where the short-term view always leads to an optimal solution – and this is the ideal situation, one refers to ***exact greedy algorithms***. Otherwise, one refers to ***heuristic greedy algorithms*** which only lead to sub-optimal solutions; they are used in the lack of an efficient exact algorithm. More generally, if we consider the search for a solution to a given problem as an exploration within a choice tree, the greedy method builds one and only one branch of the tree which, in the end, may correspond or not to an optimal solution. A greedy method never calls itself into question and moves as quickly as possible towards a solution. Blinded by its excessive appetite, the greedy principle does not guarantee arriving at an optimal solution, but it provides a result quickly.

2.2. General scheme

A greedy algorithm aims to find the best solution to an optimization problem, as much as the adopted strategy allows it. Each solution s is constructed using the elements of a finite set E according to different ways that depend on the problem to be solved (e.g. subset of E, permutation of elements of E, etc.). The algorithm maintains a set of successful and rejected candidates. The evaluation of the quality of each solution s is made according to an objective function $f(s)$.

The pseudo-code below presents the general scheme of the greedy method. It is based on a local criterion for selecting the elements of Set *E* to build a potentially optimal solution. To do so, it manipulates a set of abstract methods as follows:

- ***initialize:*** it builds the initial partial solution by choosing some elements from *E*.
- ***select:*** it selects the current best element of Set *E* with respect to the greedy criterion.
- ***isComplete:*** it checks whether a partial solution *sol* is a complete solution without taking into account the optimality aspect.
- ***canAdd:*** it checks whether a given element *e* can be added to a partial solution *sol* so that the result still remains a partial solution.
- ***add:*** it adds an element *e* from Set *E* to a partial solution *sol*.

```
Input : set E of the elements of the solution
Output : solution Sol
Begin
Cand.initialize (E) ; // initialize object Cand using the candidates of set E
Sol.initialize (Cand) ; // initialize object Sol using certain initial elements of Cand
While (not Sol.isComplete () AND Cand.candidateExist ()) do
    e ← Cand.select () ; // select element e according to the greedy criterion
    If (Sol.canAdd (e)) then
        Sol.add (e) ; //add element e to the partial solution Sol
    End if
    If (certain conditions that depend on the problem-solving) then
        Cand.remove (e) ; //remove e from Cand in order to be processed once
    End if
End while
Return Sol ;
End
```

As in any other general scheme, the scheme presented above shows advantages and drawbacks. Indeed, in some cases, it is even simpler. For example, when the solution sought is a permutation, in general the algorithm may be reduced to a sorting task according to the greedy criterion. In other cases, the solutions are a bit more complicated and therefore schemes that are more sophisticated are required.

2.3. Elements of the greedy strategy (proof of optimality)

A greedy algorithm produces a solution after making a sequence of choices, where at each decision point, it retains the choice that seems best at that time. Even so, by proceeding in this way, the algorithm does not always succeed in determining the optimal solution. There are two properties such that when they are checked, the problem-solving lends itself to a greedy strategy: *the greedy choice property* and *the optimal substructure property*.

(a) Greedy choice property: there always exists an optimal solution constructed by making a locally optimal first choice (a first element). In general, we demonstrate that any optimal solution contains or starts with this first greedy choice.

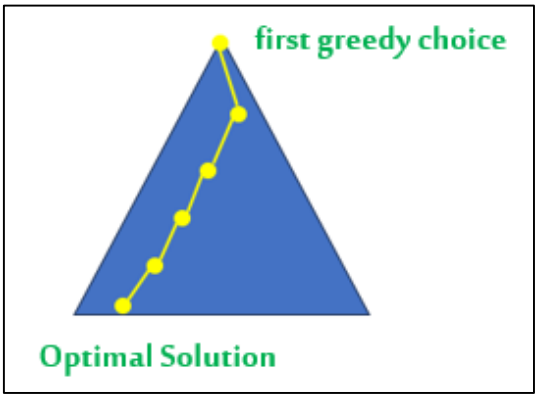


Figure 1.1. Greedy choice property.

(b) Optimal substructure property: every optimal solution contains an optimal substructure. A greedy algorithm makes a locally optimal choice and then solves the sub-problems that arise. Thus, the solving-problem progresses downwards by making successive greedy choices that iteratively reduce each instance of the problem to a smaller instance. As a result, once property (a) is proven, the optimal solution is determined by showing that it consists in the combination of the greedy first choice with an optimal solution of the underlying sub-problem.

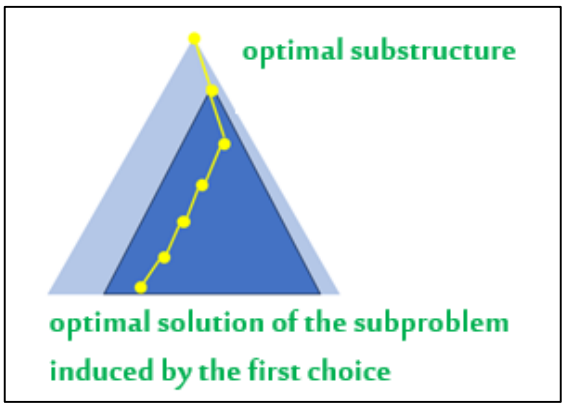


Figure 1.2. Optimal substructure property.

3. Classic examples

Now, we present typical cases of greedy methods applications. Thus, we consider a set of typical problems whose resolution helps deal with other problems either by a direct projection or by a partial adaptation.

3.1. Minimum coin change problem

This problem is formulated as designing an algorithm to give change to a customer with as few coins as possible. More precisely, given an amount to be paid and an amount given by the customer; it is asked to find the smallest combination of coins that makes up the difference. One possible solution to this problem is to use a greedy strategy as follows:

1. Sort the coins in descending order of their values.
2. For each type of coin, if the value of this coin is lower than or equal to the difference between the amount to be paid and the amount given by the customer, add a coin of this type to the solution and subtract its value from the difference.
3. Repeat from step 2 until the difference becomes zero.

- **Problem formalization**
  - **Input:** an amount M to be returned using a currency system $S = (p_1, p_2, \ldots, p_n)$ in which each type of coin p_i is characterized by a value x_i. For example, let consider the currency system $S = (p_1, p_2, p_3, p_4)$ whose values $x_{i=1..4}$ are (5 DZD, 10 DZD, 50 DZD,100 DZD).
  - **Solution:** a solution consists in calculating the numbers of coins to be returned to the customer, denoted by $k_{i=1..n}$, according to the coin types $p_{i=1..n} \in S$, respectively. We assume that for each value x_i, the number of coins is unbounded.
  - **Constraint:** sum of the returned coins must be equal to M; $M = \sum_{i=1}^{n} k_i \times p_i$.
  - **Objective function:** minimize the number of coins returned: $\sum_{i=1}^{n} k_i$.
- **Example**

We want to return an amount M to a customer with a minimum number of coins according to the following hypotheses.

M = 80.70 DZD.

S = {p_1 = 50 DZD, p_2 = 20 DZD, p_3 = 10 DZD, p_4 = 5 DZD, p_5 = 50 c, p_6 = 20 c, p_7 = 10 c}

Solution: Sol = 1 × 50 DZD + 1 × 20 DZD + 1 × 10 DZD + 1 × 50 c + 1 × 20 c where: $k_{i=1..7}$ = (1, 1, 1, 0, 1, 1, 0) and therefore $\sum_{i=1}^{n} k_i$ = 5 coins in total.

The greedy algorithm for this example goes through the following steps:

- Take coin p_i with the largest value $x_i \leq 80.70$. Therefore, the algorithm chooses p_1 whose value $x_1 = 50$ DZD so that the rest becomes 30.70.
- Likewise, take coin p_i with the largest value $x_i \leq 30.70$. Therefore, the algorithm chooses p_2 whose value $x_2 = 20$ DZD so that the rest becomes 10.70.
- And so on.

However, this greedy strategy does not guarantee optimality; let consider the following example:

$M = 80$ DZD.

$S = \{p_1 = 50 \text{ DZD}, p_2 = 40 \text{ DZD}, p_3 = 10 \text{ DZD}\}$

The adopted strategy produces solution $Sol = 1 \times 50 \text{ DZD} + 3 \times 10 \text{ DZD}$ where:

$k_{i=1..3} = (1, 0, 3)$ and thus $\sum_{i=1}^{n} k_i = 4$ coins in total.

An optimal solution could be $sol_{opt} = 2 \times 40$ DZD where:

$k_{i=1..3} = (0, 2, 0)$ thus $\sum_{i=1}^{n} k_i = 2$ coins in total.

3.2. Minimum spanning tree in a graph

The minimum spanning tree problem is a well-known problem in graph theory. It consists in finding the spanning tree of an undirected and weighted graph whose total weight is the smallest possible. Figure 1.3 represents an illustrative example of an undirected and weighted graph.

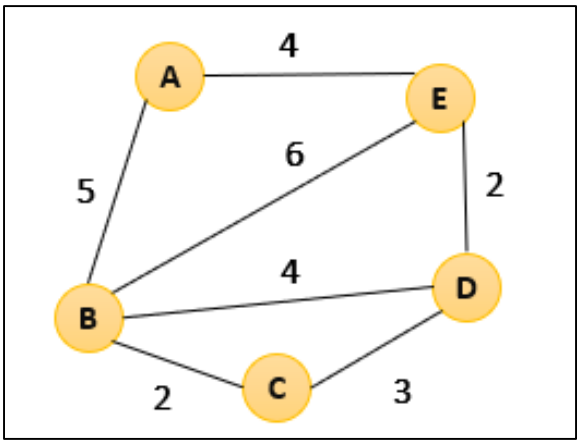


Figure 1.3. A typical undirected and weighted graph.

A spanning tree of a graph is a sub-graph which is a tree (i.e. a connected and acyclic graph) covering all the vertices of the initial graph. To find the minimum spanning tree of a given graph, there are two well-known greedy algorithms: ***Prim's algorithm*** and ***Kruskal's algorithm***.

3.2.1. Prim's algorithm

Prim's algorithm is guided by the following principle:

1. Pick an arbitrary vertex from the graph and add it to the spanning tree.
2. Find the minimum-weight edge that connects a vertex in the spanning tree to a vertex that is not yet in the spanning tree, and add it to the spanning tree.
3. Repeat from step 2 until all vertices of the graph are in the spanning tree.

The result of running this algorithm on the previous example is illustrated in Figure 1.4.

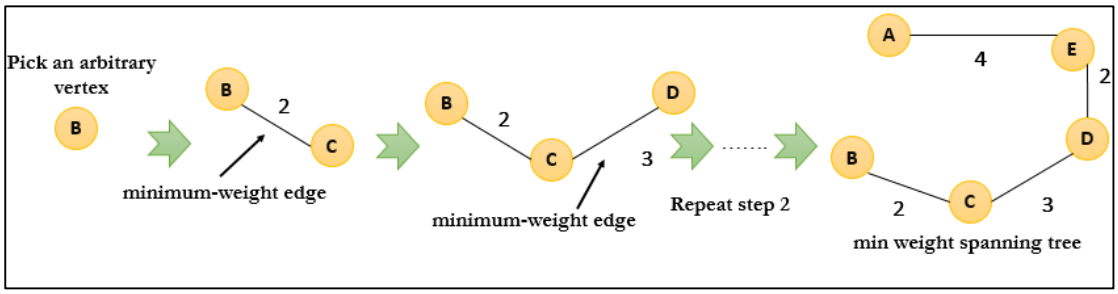


Figure 1.4. The resulting minimum spanning tree using Prim's algorithm.

The pseudo-code below describes the steps of Prim's algorithm.

Prim's algorithm

Input: $G = (X, E)$ a connected graph with positive edge weighting

Output: $T = (A, E')$ a spanning tree of minimum weight

A : Set of marked vertices

E' : Set of tree edges

Begin

Initialize E' to the empty set ;

Arbitrarily mark a vertex x such that $A = \{x\}$;

While (there is an unmarked vertex adjacent to a marked vertex) **do**

Select an unmarked vertex y adjacent to a marked vertex x such that (x, y) is the smallest weight outgoing edge ;

$E' \leftarrow E' \cup \{(x, y)\}$;

Mark y ; // $A \leftarrow A \cup \{y\}$

End while

Return $T = (A, E')$;

End

The idea of Prim's algorithm is based on the fact that at each step we choose the edge of minimum weight which connects a vertex from the spanning tree to a vertex which is not yet in the spanning tree. This edge is therefore necessarily an edge of the minimum spanning tree. By gradually adding these minimum-weight edges to the spanning tree, one finally builds the complete minimum-weight spanning tree.

3.2.2. Kruskal's algorithm

Kruskal's algorithm is another classic algorithm for solving the minimum spanning tree problem in an undirected and weighted graph. It consists of the following steps:

1. Sort all edges of the graph in ascending order of weight.
2. Loop through the sorted edges in order, and add each edge to the spanning tree if it does not lead to generating a cycle.
3. Repeat from step 2 until all edges of the graph have been traversed.

The result of running this algorithm on the previous example is illustrated in Figure 1.5.

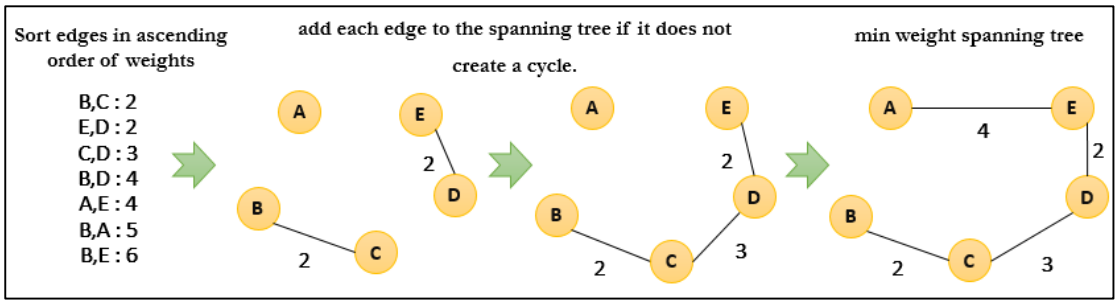


Figure 1.5. The resulting minimum spanning tree using Kruskal's algorithm.

The pseudo-code below describes the steps of Kruskal's algorithm.

Kruskal's algorithm

Input: $G = (X, E)$ a connected graph with positive edge weighting

Output: $T = (X, E')$ a spanning tree of minimum weight

Begin

sort the edges of G in ascending order of weight; // we note them as $[e_1, \ldots, e_m]$

Initialize E' to the empty set ;

For $i \leftarrow 1$ to m **do**

If ($E' \cup \{e_i\}$ does not contain cycles) **then**

$E' \leftarrow E' \cup \{e_i\}$;

End if

End for

Return $T = (X, E')$;

End

The idea of Kruskal's algorithm is based on the fact that at each step, we add to the spanning tree the edge of minimum weight which does not create a cycle. And as the edges of minimum weight are added gradually to the spanning tree, we finally build the complete minimum spanning tree.

In terms of time complexity, Kruskal's algorithm is slightly more efficient than Prim's in some cases, because it does not require calculating distances between vertices at each step. However, it requires sorting all the edges of the graph, which can be computationally expensive for large graphs.

3.3. The 0/1 knapsack problem

Informally, the knapsack problem is described as a choice among n items, those which are the most profitable, knowing that the sack has a limited capacity C. In its simplest version, namely the one-dimensional binary knapsack problem (0/1), it is asked to select a subset of items in order to maximize the total profit knowing that each item t_i is characterized by a weight w_i and a profit p_i. One way to solve that problem is to use a greedy strategy as follows:

1. Sort the items in descending order of the values of ratio p_i / w_i.
2. For each item t_i, check whether the remaining capacity of the sack is still sufficient. If so, item t_i is chosen and put in the sack. Otherwise, the algorithm moves on to the next item (i.e. item t_{i+1}).

- **Problem formulation**
  - **Input:** a set of n items $T = \{t_1, t_2, \ldots, t_n$ / each item t_i has a weight w_i and a profit $p_i\}$ and a sack of capacity C.
  - **Solution:** a solution consists in selecting a subset of items from set T. The solution is encoded as a vector $x = (x_1, x_2, ..., x_n)$ such that $x_i \in \{0, 1\}$ to indicate the absence / presence of the i^{th} item.
  - **Constraint:** the sum of the weights of the chosen items must not exceed sack capacity C: $\sum_{i=1}^{n} w_i x_i \leq C$
  - **Objective function:** maximize the total profit: $f(x) = \sum_{i=1}^{n} p_i x_i$.
- **Example**

Consider a sack of capacity $C = 10$ and a set of 4 items whose weights and profits are given in the table below.

Items (t_i)	t_1	t_2	t_3	t_4
Profits (w_i)	8	10	2.5	3
Weights (p_i)	6	7	2	2

The greedy algorithm for this example goes through the following steps:

- By sorting the items in descending order of the values of ratio p_i / w_i, we obtain:

Items (t_i)	t_4	t_2	t_1	t_3
Profits (w_i)	3	10	8	2.5
Weights (p_i)	2	7	6	2
Ratio (p_i /w_i)	1.5	1.42	1.33	1.25

- Next, the algorithm iterates over the elements of the sorted set so that it chooses items t_4 and t_2 (the total profit of selected items is 13 while their total weight is 9). The remaining items cannot be chosen as they lead to capacity overflow.

However, this greedy strategy does not guarantee the optimality. Indeed, the optimal solution includes items t_1, t_3 and t_4 in the sorted set (the total profit of selected items is 13.25 while their total weight is 10).

4. Implication of Matroid theory in greedy methods

Matroid theory covers many interesting applications of greedy methods.

4.1. Matroids

Definition. Let $M = (E, I)$ be a couple where E is a nonempty finite set of n elements and I is a nonempty family of subsets of E. M is said to be *matroid* if it satisfies the following two conditions:

1) **heredity property:** if $H \in I$ and if $F \subset H$ then $F \in I$ (we say that I is hereditary). In other words, if I contains a subset H of set E, I contains all the subsets of H. Note that the empty set is necessarily a member of I.
2) **exchange property:** if F and H are two elements of I, with $|F| < |H|$, then there exists (at least) one element $x \in H \setminus F$ such that $F \cup \{x\} \in I$.

- **Examples**

1) **Vector matroids:** let $E = \{v_1, \ldots, v_n\}$ be a set of vectors in a vector space and I be the family of all linearly independent vector subsets of E; $M = (E, I)$ is a matroid.
2) **Graph matroids (the forests of a graph):** let $G = (V, E)$ be an undirected graph and I be the family of forests of G: $F \subset I$ if and only if F is acyclic; $M = (E, I)$ is a matroid.

- **Properties**

1) Given a matroid $M = (E, I)$. For each element $x \notin F$, we say that x is an extension of $F \in I$ if $F \cup \{x\} \in I$.
2) Let F be an independent subset of a matroid M. So, we say that F is maximal if it has no extension (i.e. it is maximal in the sense of inclusion).

   Theorem. All maximal independent subsets of a matroid have the same cardinality.

   Proof. Let assume that F is a maximal independent subset of M and there is another larger independent subset H. The exchange property implies that F can be extended to an independent set $F \cup \{x\}$ for some $x \in H \setminus F$, which contradicts the assumption that F is maximal.

4.2. Greedy algorithms based on a weighted matroid

Let $M = (E, I)$ be a matroid; M is said to be weighted if there is a weighting function w of E in R^+ which assigns a strictly positive weight $w(x)$ to each element $x \in E$.

For $F \subset E$: $w(F) = \sum_{x \in E} w(x)$.

- **Question:** Find an independent of maximal weight (optimal).

The following algorithm allows finding an independent set A from E of maximal weight.

Greedy algorithm

1. Sort the elements of E by decreasing weight: $w(e_1) \geq w(e_2) \geq \ldots\ldots \geq w(e_n)$;
2. $A \leftarrow \phi$;
3. **For** $i \leftarrow 1$ to $|E|$ **do**
4. **If** $(A \cup e_i \in I)$ **then**
5. $A \leftarrow A \cup \{e_i\}$;

   End if

   End for
6. **Return** A;

- **Optimality proof**

The greedy algorithm given above verifies both the greedy choice property and the optimal substructure property. We will not provide the formal proof; more details can be found in several other references. Based on these two results, the following theorem shows that the said algorithm gives an optimal solution.

Theorem. The greedy algorithm presented above gives an optimal solution.

Proof. Let e_k be the first independent element of E, i.e. the first index k of the algorithm such that $e_k \in I$.

- There is an optimal solution that contains e_k (greedy choice).
- Then, by recurrence, we show that the algorithm gives an optimal solution (optimal substructure): we restrict ourselves to a solution containing e_k, and we start again with $E' = E - \{e_k\}$ and $I' = \{X \in E'; X \cup \{e_k\} \in I\}$.
- By looking at $E' = \{e_{k+1}, \ldots, e_n\}$, the elements e_j where $j < k$ cannot be an extension of an independent.

- **Analysis of time complexity**

Let $n = |E|$; the sorting phase of the greedy algorithm takes time of $O(n \log n)$. Each execution of line 4 imposes to check whether set $F \cup \{x\}$ is independent or not. If this verification takes time $f(n)$ and the comparisons are done in constant times, the greedy algorithm takes time of $O(n \log n) + n \times f(n)$.

4.3. Typical examples

4.3.1. Scheduling on a machine

The single-processor task scheduling problem consists of a set of tasks $X = \{t_1, t_2, \ldots, t_n\}$ of duration 1, with deadlines $d_1, d_2, \ldots, d_n$: task t_i must end before deadline d_i, otherwise a penalty w_i must be paid. The objective is to find a scheduling of the tasks that minimizes the sum of the penalties.

- **Principle of the resolution algorithm:** it can be seen that the characteristics useful to the algorithm are those of the matroid, as illustrated in the following pseudo-code.

Scheduling-Algorithm

1. Sort the elements of X by decreasing weight: $w(t_1) \geq w(t_2) \geq \ldots\ldots \geq w(t_n)$;
2. $A \leftarrow \phi$;
3. **For** $i \leftarrow 1$ to $|X|$ **do**
4. **If** $(A \cup t_i \in I)$ **then**
5. $A \leftarrow A \cup \{t_i\}$;

   End if

   End for
6. **Return** $A = \{t_1, t_2, \ldots, t_k\}$;

Set A produced by the Scheduling-Algorithm is the optimal solution with a complexity of $O(n \log n)$.

- **Example**

To show the result of running this algorithm, we consider a typical set of 07 tasks whose deadlines and associated penalties are given as follows:

Tasks (t_i)	t_1	t_2	t_3	t_4	t_5	t_6	t_7
Deadlines (d_i)	4	2	4	3	1	4	6
Penalties (w_i)	7	6	5	4	3	2	1

The greedy algorithm for this configuration of tasks goes through the following steps:

1. $A = \{t_1\}$
2. $A = \{t_2, t_1\}$
3. $A = \{t_2, t_1, t_3\}$
4. $A = \{t_2, t_4, t_1, t_3\}$
5. $A = \{t_2, t_4, t_1, t_3, t_7\}$

The resulting schedule is then $A = \{t_2, t_4, t_1, t_3, t_7\}$ with a penalty of 5.

4.3.2. Vehicle rental problem

Consider a vehicle rental company that has a vehicle for which customers submit requests identified by their beginning and end dates. Let E be the set of such statements; for any $e \in E$, we denote by $beg\ (e)$ and $end\ (e)$ its beginning and end dates, respectively. The commercial policy of the company is to satisfy as many customers as possible. The problem is therefore to find a subset of E composed of compatible requests whose cardinality is maximal. Formally, the compatibility between two requests e_1 and e_2 implies that $]beg\ (e_1), end\ (e_1)[\cap]beg\ (e_2), end\ (e_2)[= \emptyset$

- **Principle of the resolution algorithm:** to find the elements of set E, it is possible to rely on a greedy algorithm whose principle is given in the pseudo-code below.

Vehicle-Rental-Algorithm (E)

Sort the elements of E in ascending order of end date i.e. $end\ (e_1) \leq \cdots \leq end\ (e_n)$

$s_1 \leftarrow e_1$; // we choose the request that ends the earliest

$k \leftarrow 1$; // k denotes the number of currently satisfied customers

$S \leftarrow \{s_1\}$;

For $i \leftarrow 2$ to n **do**

  If ($beg\ (e_i) \geq end\ (s_k)$) **then**

    $k \leftarrow k + 1$;

    $s_k \leftarrow e_i$;

    $S \leftarrow S \cup \{e_i\}$;

  End if

End for

Return $S = \{s_1, s_2, \ldots, s_k\}$ **;**

This algorithm is based on a greedy strategy because it builds set $S = \{s_1, s_2, \ldots, s_k\}$ in a sequential way, such that at each step it makes the choice of the least cost (i.e. the compatible query of earliest end-date). In addition, the characteristics of matroids are projected directly onto this algorithm. Therefore, the Vehicle-Rental-Algorithm generates Set S as the optimal solution with a complexity of $O(n \log n)$.

- **Example**

Let consider the set of requests given in the table below.

Elements of E (e_i)	e_1	e_2	e_3	e_4
Beginnings *beg* (e_i)	3	0	2	0
Ends *end* (e_i)	6	3	6	1

Sorting the requests gives $end(e_4) \leq end(e_2) \leq end(e_1) = end(e_3)$. Thus, the Vehicle-Rental-Algorithm goes through the following steps (see Figure 1.6):

- $S = \{e_4\}$;
- e_2 is incompatible with e_4, nothing occurs;
- e_1 is compatible with e_4 ; therefore, $S = \{e_1, e_4\}$;
- e_3 is incompatible with e_3, nothing occurs.
- The returned solution is $S = \{e_1, e_4\}$.

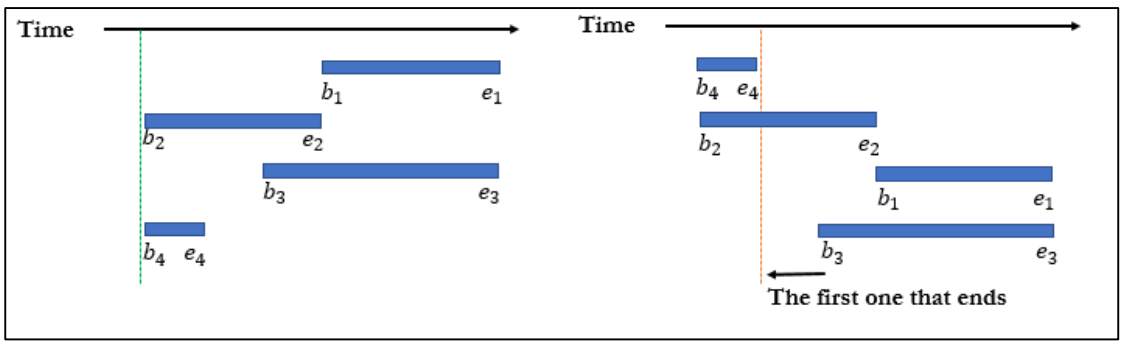


Figure 1.6. Typical execution of the greedy algorithm.

5. Advantages and drawbacks of greedy methods

Greedy algorithms have some advantages over other algorithmic methods, including:

- **Simplicity:** greedy algorithms are often easier to describe and code than other algorithms.
- **Complexity:** the search for a solution by a greedy algorithm can be considered as an exploration of a tree of choices where one and only one branch is built without going back. This reduces the cost of solving process in terms of time complexity.

- **Efficiency:** in many cases, greedy algorithms are implemented more efficiently than other algorithms. Furthermore, they are highly recommended for quickly retaining feasible solutions – even if they are not optimal.

Greedy algorithms also have some drawbacks; we cite as examples:

- **Difficult to design:** although greedy algorithms are easy to describe, the big challenge is how to do it hoping that we successfully design a good strategy.
- **Difficult to verify:** demonstrating that a given greedy algorithm is efficient or even optimal often requires a nuanced argument.
- **No guarantee of optimality:** even if greedy algorithms are often fast, on the other hand the solution they determine can be arbitrarily far from the optimal solution.

Chapter II: Divide-and-conquer method

Objective:

This chapter aims to present the ***divide-and-conquer*** paradigm and to show some of its application aspects to solve certain typical examples. It also provides a theoretical comparison with greedy methods. Finally, this chapter discusses some methods for evaluating the algorithms of this paradigm in terms of time complexity, in particular through the Master-theorem.

1. Overview on the divide-and-conquer method

1.1. Principle of the divide-and-conquer method

Originally, *divide-and-conquer* was a widespread military strategy before becoming an algorithmic design technique. Indeed, it was observed that defeating two armies of 50,000 soldiers successively was easier than facing a single army of 100,000 soldiers. Proceeding analogously, it was inspired by this strategy to design algorithms which solve complex problems by using sub-algorithms of lesser complexity. The principle consists in dividing a large problem into several analogous sub-problems. The initial problem is solved by recombining the partial results obtained by solving its parts. This process is applied recursively until it reaches basic sub-problems which are directly solved. By their very nature, recursive algorithms use this paradigm because they call themselves one or more times on a partition of the original problem, solve the sub-problems recursively and then combine the solutions to find a solution to the initial problem.

1.2. Strategy of the divide-and-conquer method

The divide-and-conquer paradigm follows a top-down approach, in order to provide solutions to problems. The solving-process consists of three stages:

1) ***Divide*** the problem by decomposing it into similar sub-problems of smaller sizes whose resolution is identical to that of the initial problem.
2) ***Conquer*** over the sub-problems through recursive solving (recursive calls). If the size of a sub-problem is quite small, its resolution is done directly (i.e. we have reached a basic case).
3) ***Combine*** the solutions of the sub-problems to construct a complete solution to the initial problem.

In order to ensure a balanced processing, it is strongly recommended that the decomposition of the problem (i.e. the "divide" stage) leads – as much as possible – to sub-problems of roughly equal size. Let's consider the example of an integer N; if N is even, we take two sub-problems of size $N/2$, otherwise we take two sub-problems of sizes $(N - 1) / 2$ and $(N + 1) / 2$, respectively. The solution to the sub-problems is obtained by applying the same process until the problem-solving becomes trivial. Generally, divide-and-conquer algorithms are applied according to two main strategies:

- The first strategy is *the recursion on data* for which the data is directly decomposed into a certain number of partitions. This gives rise to sub-problems which are solved recursively using the same function. At the end, the results obtained are combined in order to calculate the complete solution.
- The second strategy is *the recursion on results* which first carries out a preprocessing before cutting the data into partitions. Then, it recursively solves the resulting sub-problems using the same function. Finally, it combines the results obtained in order to calculate the complete solution.

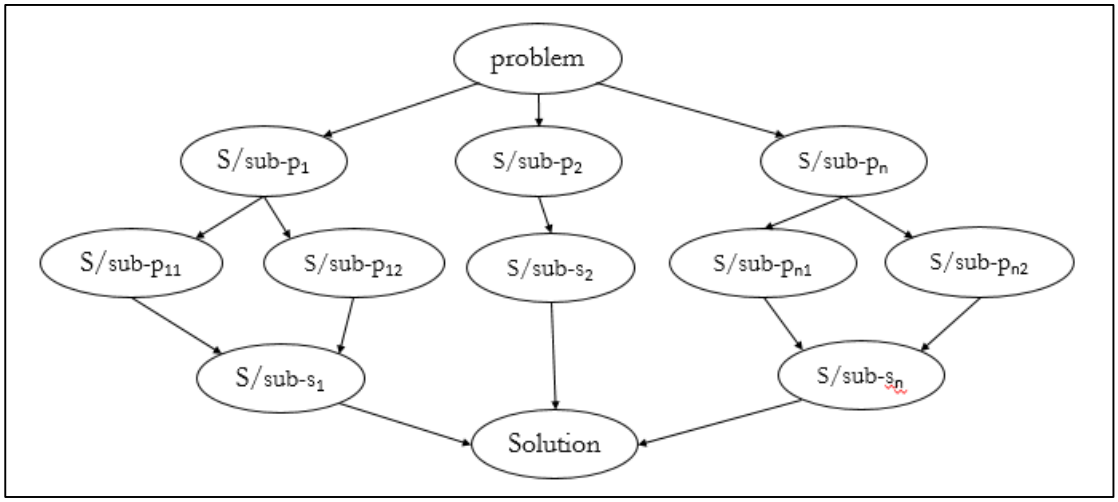


Figure 2.1. Strategy of the "divide-and-conquer" method.

1.3. General scheme

In general, a viable divide-and-conquer algorithm should:

- Efficiently divide the problem into sub-problems of balanced sizes (i.e. of approximately the same size as possible).
- Recombine the solutions of the sub-instances through an effective exploitation of the obtained partial results.
- Determine a fitting threshold at which the problem is easily solved on small instances rather than relying on recursive calls.

The general scheme of the divide-and-conquer algorithm is given as follows:

```
Function Divide-and-conquer (P : Problem) : Solution
S : Solution ;
Begin
If (‖ P ‖ is small) then
    S ← Basic-case (P) ;
Else
    (P1, P2, ..., Pk ) ← Divide (P) ;
    For i ← 1 à k do
        Si ← Conquer (Pi) ;
    End for
    S ← Combine (S1, S1, ..., Sk) ;
End if
Return S ;
End
```

2. Analysis of divide-and-conquer algorithms

2.1. General form of recursive formulas for time complexity

The time complexity of recursive algorithms is defined by means of recursive formulas as according to problem size n. In particular, for a divide-and-conquer algorithm this recursion is defined based on the three stages namely ***divide***, ***conquer*** and ***combine***:

1. If the size of the problem is small enough (basic case), $n \leq c$ for some constant c, the solution is straightforward and therefore consumes a constant time of $O(1)$.
2. Otherwise, the problem is divided into a sub-problems each of which is of size $1/b$ of the initial problem size n. Thus, the total time breaks down into three parts as follows:
   A. $D(n)$: time required to divide the problem into sub-problems.
   B. $a \times T(n/b)$: time required to solve a sub-problems.
   C. $C(n)$: time required to calculate the global solution using the partial solutions.

Thus, the recurrence relation takes the form:

$$T(n) = \begin{cases} O(1) & \text{if } n \leq c \\ a \times T(n/b) + C(n) + D(n) = a \times T(n/b) + O(n^d) \end{cases}$$

where n/b is interpreted either as $\lfloor n/b \rfloor$ or as $\lceil n/b \rceil$.

2.2. The Master-theorem

The master theorem allows solving equations written as recursive formulas related to the divide-and-conquer paradigm. Thus, we consider equation $T(n) = a \times T(n/b) + O(n^d)$, where n/b is interpreted either as $\lfloor n / b \rfloor$ or as $\lceil n / b \rceil$.

Let $\lambda = \log_b a$. There are three possible scenarios:

1) if $\lambda > d$, then $T(n) = O(n^\lambda)$;
2) if $\lambda = d$, then $T(n) = O(n^d \log n)$;
3) if $\lambda < d$, then $T(n) = O(n^d)$.

In practice, only cases 1 and 2 lead to interesting algorithmic solutions. In case 3, all the cost is concentrated in the recombination phase, which often means that there exist other more efficient solutions.

3. Divide-and-conquer algorithms *vs* greedy algorithms

Both greedy and divide-and-conquer algorithms are two of the most widely used paradigms for solving complex problems. Some differences between these methods are given in the table below:

Feature	**Greedy algorithms**	**Divide-and-conquer algorithms**
Feasibility	They make a choice that seems the best at the time hoping that it leads to the global optimum.	They make a decision at each step taking into account the current sub-problems to calculate the solutions.
Goal	They are used to find solutions to optimization problems, hoping that the optimal solutions are retained.	They are used to get solutions to various problems without necessarily working in the context of optimization.
Recursion	They are based on heuristics to make the locally optimal choice at each stage.	They are mainly based on recursive formulas so that they use the same process to solve the sub-problems.
complexity	They generally run faster.	They generally run slower.
Fashion	They compute the solutions by making choices in a serial forward fashion, never looking back or revising previous choices.	They compute the solutions in top-down by dividing the problems into smaller sub-problems that are solved independently. The solutions are obtained by combining the partial solutions of the solved sub-problems

4. Classic examples

Now, we present typical cases of applying the divide-and-conquer method to solve some computation tasks.

4.1. Binary search vs sequential search in an array

Sequential (linear) search iterates over an array or a list to decide whether a given item exists or not. The algorithm compares each element of the sequence with the sought item until either it finds it or reaches the end of the sequence. The pseudo-code below gives the steps for the sequential search for an item e in an array of integers T.

Input: sought item e and an array $T = [e_1, e_2, ..., e_n]$.
Output: index i of the first element such that $T[i] = e$ or 0 if e does not belong to T.
Begin
  For $i \leftarrow 1$ to n **do**
    If ($T[i] = e$) **then**
      Return i ;
    End if
  End for
  Return 0 ;
End

Binary search is an alternative solution to sequential search when dealing with an already sorted array. The goal is to reduce time complexity. The basic idea behind binary search is that at each step the array is divided into two equal parts. Then, the searched item is compared with the item in the middle. If the searched item is found, then the algorithm returns the corresponding cell index. Otherwise, if the searched item is lower than the median element, we make a search in the left half of the array. Finally, if the searched item is greater than the median element, we make a search in the right half of the array. In what follows, we give the steps of binary search for an item e in a sorted array $T[p..r]$.

- **Divide:** we partition array $T[p .. r]$ around the median. Thus, we obtain two sub-arrays $T_1[p .. q\text{-}1]$ et $T_2[q+1 .. r]$; q is the position of the median element.
- **Conquer:** if item e is equal to the median element (i.e. $T[q]$) or the size of the array is lower than or equal to 1, the solution is directly got (basic case). Otherwise, we recursively search in sub-arrays T_1 and T_2 using binary search algorithm.
- **Combine:** the decision about the existence of the item sought e is made based on the decisions made by considering the two sub-arrays T_1 and T_2.

The following pseudo-code shows the steps of binary search for an item in a sorted array.

```
Function binary-search (T : array [begin .. end], item e) : integer
Begin
  If (begin > end) then
     Return 0 ;
  Else
     middle ← (begin + end) / 2 ; // in order to partition array T into two parts
     If (T [middle] = e) then
        Return middle ;
     Else
        If (T [middle] > e) then // binary-search in the first part
           Return binary-search (T [begin .. middle - 1], e) ;
        Else // binary-search in the second part
           Return binary-search (T [middle +1 .. end], e) ;
        End if
     End if
  End if
End
```

- **Example**

We want to search for item $e = 38$ in the sorted array T = [3, 9, 10, 27, 38, 43, 82] using binary search. Thus, the algorithm presented above goes through the following steps:

Step 1: it compares the median element of the array (10) with value $e = 38$. As 10 is lower than 38, it searches in the right half of array T.

Step 2: likewise, it divides the right half of array T into two equal sub-parts. Then, the element at the median (38) is compared with the element sought (38). As the two elements are equal, the goal is achieved by returning position 4 in array T.

Result: the sought element (38) was found at position 4 of array T. It is shown that the time complexity of the binary search algorithm is $O(log_2 n)$.

4.2. Quick sort

Quick-sort is an algorithm for sorting arrays or lists while being designed around the divide-and-conquer paradigm. The initial data is an unsorted sequence of integers. The result is a sorted sequence of these integers. The basic idea behind the quick-sort algorithm is to choose a pivot element in the array, partition the array around that pivot, and redo

the process recursively on the resulting sub-arrays. The pivot can be chosen in different ways, by taking either the first element of the array or any element chosen at random. The steps of applying the quick-sort algorithm to a given array $T[p .. r]$ are as follows:

- **Divide:** we partition array $T[p .. r]$ around the pivot. Then, we put all the elements with small values to the left of the pivot (i.e. $T[p .. q-1]$) and all the elements with greater values to the right of the pivot (i.e. $T[q + 1 .. r]$); q is the final position of the pivot. Hence, we obtain two unsorted sub-arrays $T_1[p .. q-1]$ and $T_2[q+1 .. r]$.
- **Conquer:** each of sub-arrays $T_1[p .. q - 1]$ and $T_2[q + 1 .. r]$ are recursively sorted using the quick-sort algorithm.
- **Combine:** the two sorted sub-arrays are gathered to obtain a sorted array.

The pseudo-code below describes the steps of the quick-sort algorithm.

```
Procedure quick-sort (T : array [left .. right])
Begin
  If (left < right) then
     Initialize variables pivot, i and j to T [left], left and right, respectively;
     While (i < j) do
        i ← i + 1 ;
        While (T [i] ≤ pivot AND i < j) do
           i ← i + 1 ;
        End while
        While (T [j] > pivot) do
           j ← j - 1 ;
        End while
        If (i < j) then
           Swap (T, i, j) ;
        End if
     End while
     Swap (T, left, j - 1) ;
     quick-sort (T, left, j - 1) ; //quick-sort of the first part
     quick-sort (T, j + 1, right) ; //quick-sort of the first part
  End if
End
```

- **Example**

We want to sort an array of integers T = [38, 27, 43, 3, 9, 82, 10] using quick-sort. Thus, the algorithm presented above goes through the following steps:

Step 1: it chooses a pivot, for example the first element of array T i.e. $pivot$ = 38.

Step 2: it partitions array T around the pivot so that we put the elements smaller than 38 to the left of the pivot (i.e. T_1 = [27, 3, 9, 10]) and the elements greater than 38 to the right of the pivot (i.e. T_2 = [43, 82]).

Step 3: it recursively sorts each of sub-arrays T_1 and T_2 in order to obtain a sorted array.

At the end, we get a sorted array: [3, 9, 10, 27, 38, 43, 82]. The time complexity of quick-sort algorithm according to the worst case is shown to be $O(n^2)$.

4.3. Merge-sort

Merge-sort is another sorting algorithm based on the divide-and-conquer method. It is used to sort arrays or lists recursively. The initial data and the result are therefore the same as for the quick-sort algorithm.

The basic idea behind the merge-sort algorithm is to first divide the initial data sequence into two equal parts, then sort each of the parts separately, and finally merge the two sorted parts to obtain a sorted sequence. The efficiency of the algorithm lies in the fact that the two sorted parts are merged in linear time. The steps of applying the merge-sort algorithm to sort the elements of an array array T [1 .. n] are given as follows:

1. **Divide:** the sequence to be sorted is divided into two sub-sequences of $\frac{n}{2}$ elements: $T_1\left[1, \ldots, \frac{n}{2}\right]$ and $T_2\left[\frac{n}{2}+1, \ldots, n\right]$.
2. **Conquer:** if the array to be sorted contains at most one element, it is already sorted. In this case, the algorithm simply returns the considered array without modification (basic case). Otherwise, it recursively sorts each of the two parts T_1 and T_2 using the merge-sort algorithm.
3. **Combine:** the two sorted parts are merged into a single sorted part. To do so, the algorithm compares the elements of the two parts in ascending or descending order (depending on the sorting purposes), then it puts them in a new array so that the resulting sequence is sorted.

The following pseudo-code describes the steps of the merge-sort algorithm.

```
Function merge-sort (T : array [begin .. end]) : array [begin .. end]
Begin
  If (begin ≥ end) then
     Return T ;
  Else
     middle ← (begin + end) / 2 ; // in order to partition array T into two parts
     T1 ← merge-sort (T [begin .. middle]) ; // merge-sort of the first part
     T2 ← merge-sort (T [middle +1 .. end]) ; // merge-sort of the second part
     Return merge (T1, T2) ; // merge the two sorted parts
  End if
End
Function merge (T1 : array [b1 .. e1], T2 : array [b2 .. e2]) : array T [begin .. end]
  Initialize i, j and k to b1, b2 and 1, respectively;
  While (i ≤ e1 AND j ≤ e2) do
     If (T1 [i] > T2 [j]) then
        Copy T1 [i] into T [k] and then increment i ;
     Else
        Copy T2 [j] into T [k] and then increment j ;
     End if
     k ← k + 1 ;
  End while
  For r ← i to e1 do
     Copy T1 [r] into T [k] and then increment k ;
  End for
  For r ← j to e2 do
     Copy T2 [r] into T [k] and then increment k ;
  End for
  Return T ;
End
```

- **Example**

We want to sort an array of integers T = [38, 27, 43, 3, 9, 82, 10] using merge-sort. Thus, the algorithm presented above goes through the following steps (see Figure 2.2.):

Step 1: Array T is divided into two parts: T_1 = [38, 27, 43, 3] and T_2 = [9, 82, 10].

Step 2: We recursively sort each of the parts T_1 and T_2:

- **Processing on part 1:** we divide array T_1 into two equal parts: T_{11} = [38, 27] and T_{12} = [43, 3]. Then, we recursively sort each of the two subparts so that array T_{11} becomes [27, 38] and array T_{12} becomes [3, 43]. Finally, we merge the two sorted subparts into a single sorted part: [3, 27, 38, 43].
- **Processing on part 2:** in a similar way, array T_2 is divided into two parts: T_{21} = [9] and T_{22} = [82, 10]. Then, we recursively sort each of the two subparts so that array T_{21} remains unchanged (subpart already sorted) and array T_{22} becomes [10, 82]. Finally, we merge the two sorted subparts into a single sorted part: [9, 10, 82].

Step 3: the two parts resulting from the sorting of sub-arrays T_1 and T_2 are merged into a single sorted part: [3, 9, 10, 27, 38, 43, 82].

The final result is a sorted array T' = [3, 9, 10, 27, 38, 43, 82]. It is shown that the time complexity of the merge-sort algorithm is $O(n \log n)$.

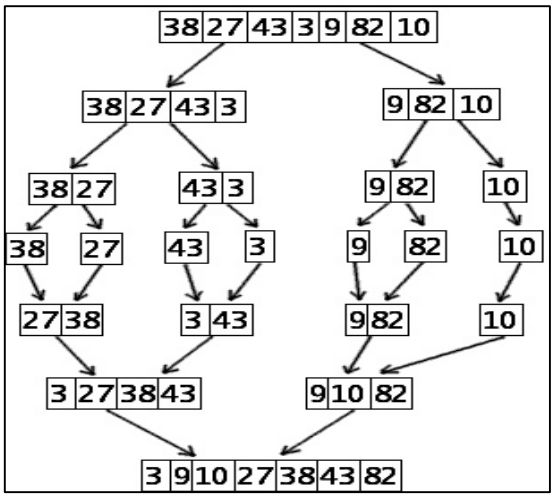


Figure 2.2. Sorting array T using merge-sort algorithm.

4.4. Strassen's algorithm for matrix multiplication

Strassen's algorithm is a divide-and-conquer matrix multiplication algorithm that is faster than the standard matrix multiplication algorithm for large matrices. The basic idea of Strassen's algorithm is to divide the matrices to be multiplied into smaller sub-matrices, in order to reduce the total number of scalar multiplications needed for the operation. More precisely, if A and B are two square matrices of size $n \times n$, Strassen's algorithm performs the multiplication operation as follows:

- **Divide:** partition each matrix into four sub-matrices of size $n/2 \times n/2$:

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

- **Conquer:** calculate the scalar multiplications $P_{i = 1 .. 7}$ and the scalar values $C_{i = 1 .. 4}$:

  $P_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$

  $P_2 = (A_{21} + A_{22}) \times B_{11}$

  $P_3 = A_{11} \times (B_{12} - B_{22})$

  $P_4 = A_{22} \times (B_{21} - B_{11})$

  $P_5 = (A_{11} + A_{12}) \times B_{22}$

  $P_6 = (A_{11} - A_{21}) \times (B_{11} + B_{12})$

  $P_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$

  $C_{11} = P_1 + P_4 - P_5 + P_7$

  $C_{12} = P_3 + P_5$

  $C_{21} = P_2 + P_4$

  $C_{22} = P_1 + P_3 - P_2 - P_6$

- **Combine:** form the resulting matrix C from sub-matrices $C_{i = 1 ..2, j = 1.. 2}$:

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

The main advantage of Strassen's algorithm is that it reduces the number of scalar multiplications needed to perform matrix multiplication from 8 to 7. Although this strategy seems non-significant, it generates a clear improvement in performance when dealing with sufficiently large matrices.

- **Example**

In the following, Strassen's algorithm is applied to the calculation of a multiplication operation of two matrices A and B of size 2×2.

Let $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$ be two square matrices. Strassen's algorithm goes through the following steps:

Step 1: divide each matrix into four sub-matrices of size 1×1:

$$A = \begin{bmatrix} A_{11} = [1] & A_{12} = [2] \\ A_{21} = [3] & A_{22} = [4] \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} B_{11} = [5] & B_{12} = [6] \\ B_{21} = [7] & B_{22} = [8] \end{bmatrix}$$

Step 2: calculate the scalar operations:

$P_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22}) = (1 + 4) \times (5 + 8) = 65$

$P_2 = (A_{21} + A_{22}) \times B_{11} = (3 + 4) \times 5 = 35$

$P_3 = A_{11} \times (B_{12} - B_{22}) = 5 \times (6 - 8) = -2$

$P_4 = A_{22} \times (B_{21} - B_{11}) = 4 \times (7 - 5) = 8$

$P_5 = (A_{11} + A_{12}) \times B_{22} = (1 + 2) \times 8 = 23$

$P_6 = (A_{11} - A_{21}) \times (B_{11} + B_{12}) = (1 - 3) \times (5 + 6) = -22$

$P_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22}) = (2 - 4) \times (7 + 8) = -30$

$C_{11} = P_1 + P_4 - P_5 + P_7 = 65 + 8 - 24 - 28 = 19$

$C_{12} = P_3 + P_5 = P_3 + P_5 = -2 + 24 = 22$

$C_{21} = P_2 + P_4 = P_2 + P_4 = 35 + 8 = 43$

$C_{22} = P_1 + P_3 - P_2 - P_6 = 65 - 2 - 35 + 22 = 50$

Step 3: combine sub-matrices $C_{i = 1..2, j = 1..2}$ to obtain the final result:

$$C = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

The result is checked by performing the standard matrix multiplication $A \times B$ to obtain:

$$C = \begin{bmatrix} 1 \times 5 + 2 \times 7 & 1 \times 6 + 2 \times 8 \\ 3 \times 5 + 4 \times 7 & 3 \times 6 + 4 \times 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

The result is identical to that obtained using Strassen's algorithm.

Note that for sufficiently large matrices, Strassen's algorithm shows higher performances than the standard matrix multiplication algorithm. Nevertheless, Strassen's algorithm is not always faster, as it has a higher constant running time.

5. Advantages and drawbacks of the "divide-and-conquer" method

Divide-and-conquer algorithms have some advantages over other methods, including:

- **Simplicity:** divide-and-conquer algorithms are often easier to describe and code than other algorithms. This is because they mainly rely on recursive formulas.
- **Possibility of parallelism:** divide-and-conquer algorithms are naturally adapted to perform parallel computation provided that the different threads run independently (e.g. the merge-sort).
- **Efficient use of cache memory:** divide-and-conquer algorithms efficiently use cache memory without occupying much space. Indeed, the sub-problems are small enough so that they can be solved in cache without using the main memory which is slower. Algorithms using cache efficiently are called *cache oblivious*.

Divide-and-conquer algorithms also have some drawbacks; let us cite as examples:

- **Difficulties induced by recursion:** divide-and-conquer algorithms naturally lend themselves to recursive writing. This may lead to two major issues: high runtime stack usage and resource overflow caused by the number of recursive calls.
- **Computational redundancy:** the recursive algorithms resulting from applying the divide-and-conquer method sometimes lead to computational redundancy with respect to the obtained sub-problems; the Fibonacci function is a good example.

Chapter III: Dynamic programming

Objective:

This chapter aims to present the ***dynamic programming*** paradigm and then to show some of its application aspects to solve several typical examples. It also provides a comparison with both greedy and divide-and-conquer methods. Finally, this chapter presents some methods for designing such algorithms.

1. Overview on dynamic programming

1.1. Dynamic programming principle

Dynamic programming is an algorithmic method that solves optimization problems by breaking them down into simpler sub-problems and then deals with these sub-problems recursively. This approach can efficiently solve complex problems by avoiding recalculating redundant sub-problems multiple times. To do so, dynamic programming involves matrix programming methods. The idea is to store the intermediate results in an array so that they are reused to solve later sub-problems. Generally speaking, dynamic programming is very suitable for solving combinatorial optimization problems, where the goal is to optimize a constrained objective function. In such cases, the solving-process maintains a set of potential solutions for which it has to find the optimal solutions (i.e. those maximizing or minimizing the value of the cost function).

1.2. When and how to use dynamic programming

Although dynamic programming is a general method of problem-solving, there is no rule to say that it can or cannot be used to solve this or that problem. In general, the possibility of using this method is a question that depends on the ability of satisfying two main properties: *the property of optimal substructure (Bellman's principle of optimality)* and *the property of superposition (overlapping) of the sub-problems.*

1) **Optimal substructure property (Bellman's principle):** any optimal solution relies itself on the combination of locally solved sub-copies in an optimal way.
2) **Property of the superposition (overlapping) of sub-problems:** once the recursive expression has been obtained, we proceed to an analysis of what happens in a naive recursive implementation. If we realize that the same problem is solved several times, we are then in the context of dynamic programming.

1.3. Dynamic programming strategy

Dynamic programming follows both top-down and bottom-up approaches to solve problems. The construction of solutions gives rise to four stages:

1) The application of Bellman's principle allows obtaining the recursive formula that defines the solution to the considered problem according to its sub-problems.
2) In dynamic programming, each sub-problem is solved only once where the result is stored in a cell of an array. The decomposition of the problem helps determine the structure of this array (which can be of dimension 1, 2, 3, etc.) according to the number of parameters involved in the recursive formula. Once the table of results has been created, its elements are initialized. This step depends on the initial conditions of the formula obtained in step 1.
3) Then, the solving-process fills in the table of results. This step consists in solving the different sub-problems using the formula obtained in step 1, following bottom-up or top-down orders. There are two approaches to populate the table of results:

i) **iterative approach:** the solving-process first initializes the cells corresponding to the basic cases. Then, the table is filled in according to a very precise order: the sub-problems are solved from smallest to largest until reaching the main problem. For each decision step, only the solutions already calculated are used so that each element is calculated once and only once.

ii) **recursive approach:** on each call to the solving-process, it looks at the table to see if the value has already been calculated. If so, it is not recalculated but rather the stored value is retrieved and used. Otherwise, it is calculated, memorized in the corresponding cell and finally used.

4) Step 3 allows only retrieving the optimal value of the cost function without specifying the intermediate values contributing to obtaining the result. In general, the detail of the solutions requires traversing the table of results starting from the final solution and reconstructing the reverse path of the calculations made to achieve there.

1.4. General scheme

In general, to design an efficient algorithm based on dynamic programming principles, some aspects should be taken into consideration: How to define the array of results as well as the boundary values? In what order should one fill it in? Where to get the answer?

There are two main ways to store the values in order to reuse sub-problems results: *Tabulation* which is bottom-up and *Memoization* which is top-down. The statements

given in the following example help make a distinction between the two methods. Ali says: "I will first learn foundations of dynamic programming, and then I will practice several exercises in order to master this programming paradigm". In turn, Sarah says: "to master the dynamic programming paradigm, I would practice several exercises but first I would have to study its foundations". Both Ali and Sarah say the same thing, the difference simply lies in the way the message is conveyed and that's exactly what tabulation and memoization do, respectively.

The general scheme of tabulation follows an iterative process, as given in the pseudo-code below.

```
Function dynamic-programming (x : problem) : solution
Begin
    Define an array T of dimension d ;
    Initialize the values of cells in array T ;
    For i1 ← beg1 to end1 do
      ⋮
        For id ← begd to endd do
            T [i1, …., id] ← expression using the cells already calculated ;
        End for
      ⋮
    End for
    Return expression using cells from array T ;
End
```

Although it solves the problem of redundant computations found in the divide-and-conquer method, dynamic programming may in turn lead to unnecessary calculations. This is due to performing bottom-up treatments which may generate values that will not be used later.

To overcome these difficulties, it is possible to rely on the technique of memoization through an effective combination between the simplicity and elegance of the divide-and-conquer paradigm (i.e. recursion) and the efficiency of dynamic programming (i.e. table of results). The idea is to use an array of sufficient size that stores the solutions of the sub-problems within a recursive function that defines the solving-process of the original problem, as shown in the following pseudo-code.

```
Global variables
T : array of d dimensions ;
function f (x1, x2, …, xd) : solution
Begin
    If (T [i1, …., id] ≠ initialization value) then
        s ← T [i1, …., id] ;
    Else
        s ← f (x'1, x'2, …, x'd) ;
        T [i1, …., id] ← s ;
    End if
    Return s ;
End
```

The elements of array T are initialized with a special value to indicate that they are not yet defined. Then, with each call to function f, we check the existence of the calculated value in array T. If the values is already calculated, we directly return the content stored in array T. Otherwise, we proceed to calculate function f, store the result in array T and finally return the value calculated on these parameters. The memoization avoids the recalculation of previously used values. Even so, this may lead to excessive use of additional memory space; therefore, even if we gain in temporal complexity, we lose in spatial complexity. The table below gives a comparison between the tabulation and memorization techniques.

	Tabulation	**Memoization**
Transition state	Difficult to think.	Easy to think.
Code	Complicated when several conditions are required.	Easy and less complicated.
Speed	Fast, as previous states are directly accessed from the table	Slow due to the number of recursive calls.
Sub-problem solving	Useful when all sub-problems must be solved at least once.	Useful in cases where some sub-problems do not need to be solved at all.
Table entries	All entries are filled one by one, starting from the first entry (basic cases).	All entries of the lookup table are not necessarily filled; the table is filled on demand.
Approach	Iterative approach.	Recursive approach.

2. Dynamic programming *vs* divide-and-conquer paradigm

The divide-and-conquer paradigm relies on breaking down a problem into identical sub-problems, solving them recursively, and then combining the partial solutions to form the solution to the initial problem. If such a decomposition always leads to independent sub-problems, this strategy is probably effective. Otherwise, if the resulting sub-problems have dependencies between them, they will have other common sub-problems. This leads the algorithm to perform extra processing since it solves certain sub-problems several times. This disadvantage is illustrated by Figure 3.1 which schematizes the execution of the calculation of the Fibbonacci sequence by the divide-and-conquer method.

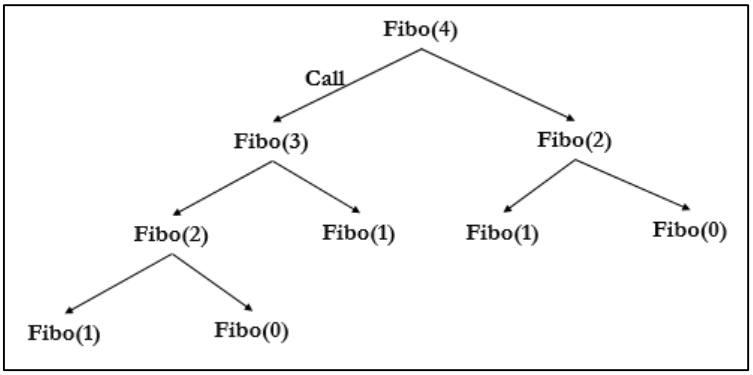


Figure 3.1. Execution scheme of Fibbonacci sequence by the divide-and-conquer method.

Dynamic programming is a design paradigm that overcomes some of the difficulties induced by the divide-and-conquer paradigm by making improvements and adaptations so that redundant computations will be computed only once. Just like in the divide-and-conquer method, this paradigm also solves a given problem based on the previous solutions obtained from the sub-problems. However, in dynamic programming, sub-problems may overlap so as to be used in solving several different sub-problems. On the other hand, in the divide-and-conquer paradigm, the sub-problems are completely independent of each other and are solved separately even if they present redundant computations. In other words, dynamic programming allows sub-problems to interact with each other, which is not the case for the divide-and-conquer method. Figure 3.2 illustrates this difference between these two methods where the root represents the problem to be solved while the descendants represent the sub-problems whose resolution is easier. In particular, the leaves represent the sub-problems corresponding to the basic cases (i.e. trivial resolution without decomposition).

The second major difference between these two methods lies in the way of carrying out the computations to solve a given problem through the recombination of the solutions of the sub-problems. In the divide-and-conquer method, treatments are always performed

from top to down, starting with solving the largest sub-problems. In contrast, treatments in dynamic programming can be carried out as well in bottom-up as in top-down depending on whether one starts with solving the smallest sub-problems before the main problem or not.

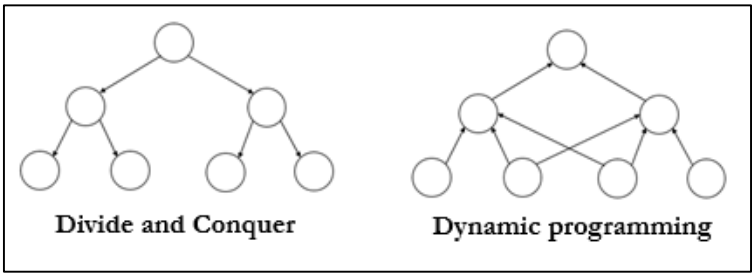


Figure 3.2. Difference between dynamic programming and divide-and-conquer methods.

3. Dynamic programming *vs* greedy algorithms

Greedy algorithms and dynamic programming are two of the most widely used paradigms for solving optimization problems. Detailed differences are given in the table below:

Feature	**Greedy algorithms**	**Dynamic programming**
Feasibility	They make a choice that seems the best at the time hoping that it leads to the global optimal solutions.	It makes a decision at each step taking into account the current problem and the solutions of solved sub-problems to calculate the optimal solution.
Optimality	Sometimes there is no guarantee to get the optimal solutions.	It is guaranteed to get the optimal solution as it considers all possible cases and then chooses the best one.
Recursion	They are based on heuristics to make the locally optimal choice at each stage.	It is based on recursive formulas that use some previously calculated states.
Space / time complexity	They are more efficient in terms of memory as they never look back. Moreover, they generally run faster.	It stores the intermediate results of solved sub-problems in a table; this may increase the space complexity. Moreover, they generally run slower.
Fashion	They compute the solutions by making choices in a serial forward fashion while never looking back or revising previous choices.	It computes the solutions in bottom-up or top-down by synthesizing them from smaller optimal sub-solutions.

4. Classic examples

Now, we present some typical cases of dynamic programming applications to solve certain computational tasks.

4.1. Calculation of shortest path in a graph (Floyd-Warshall algorithm)

Let $G = (V, E)$ be a directed graph where each edge has a non-negative length; $V = \{1, 2, \ldots, n\}$ is a set of n vertices (nodes) and E is the set of edges between the vertices. The distances between the vertices are represented as adjacency matrix M [1..n, 1..n]. To calculate the length of the shortest paths between all pairs of vertices from set V, Floyd's algorithm is used. To do so, it builds a matrix D which gives the length of the shortest path between each pair of vertices. This is an optimization problem that verifies the optimality principle: if the shortest path (optimal path) between two vertices A and B goes through an intermediate vertex C, then the portions of paths between A and C, and between C and B, must necessarily be optimal.

1. **Recursive formula:** Floyd's algorithm calculates the shortest path between each pair of vertices by using as intermediate vertices the elements of set V in order and successively. At each iteration k, matrix D gives the length of the shortest paths by involving only vertices {1, ..., k} as intermediate nodes. The recursive formula for calculating matrix D values is given as follows: $\forall\, i, j \in V$, D [i, j, k] = min (D [i, j, k - 1], D [i, k, k-1] + D [k, j, k-1]); the aim is to finally calculate D [i, j, n].
2. **Definition and initialization of the array of results:** the decomposition of the problem shows that the recursive formula is defined according to a single parameter: iteration number k. Intuitively, the array of results, denoted by D[1..n, 1..n, 1..n, 0..n], is three-dimensional (d = 3), where each element D [i, j, k] will store the shortest distance between vertices i and j involving only nodes {1, .., k}.
3. **Filling in the array of results:** the filling of the array of results is done according to the iterative approach, as shown in the pseudo-codes below.
4. **Reading the solution:** by reading any given cell D [i, j, n], we obtain an immediate answer about the shortest path between vertices i and j. However, if we want to list the intermediate vertices composing the shortest path between vertices i and j, we need to add an array N [1..n, 1..n, 1..n, 0..n] such that each cell N [i, j, k] keeps the index of the selected vertex at iteration k.

```
Floyd's algorithm
Global variables
M [1..n, 1..n, 1..n, 0..n] : array of real ; //vertices and edges of graph G = (V, E)
D [1..n, 1..n, 1..n, 0..n] : array of real ;
Procedure Floyd-Warshall ()
Begin
For i ← 1 to n do
  For j ← 1 to n do
    D [i, j, 0] ← M [i, j] ; //Assume that M [i, i] = 0 and M [i, j] = +∞ if (i, j) ∉ E
  End for
End for
For k ← 1 to n do
  For i ← 1 to n do
    For j ← 1 to n do
      D [i, j, k] ← min (D [i, j, k - 1], D [i, k, k-1] + D [k, j, k-1]) ;
    End for
  End for
End for
End
```

4.2. Calculation of the binomial coefficient

A binomial coefficient, denoted by C_k^n, is defined over each two integers $n \geq 0$ and k with $0 \leq k \leq n$, as the number of parts with k elements of a set of n elements. Here, we are interested in the recursive definition of binomial coefficients, given by the following formula:

$$C_k^n = \begin{cases} 1 & \text{if } n = k \text{ or } k = 0 \\ C_k^{n-1} + C_{k-1}^{n-1} & \end{cases}$$

It is then asked to write a function that calculates C_k^n using the principle of dynamic programming. For this purpose, we rely on the steps defined in section 4.

1. **Recursive formula:** the recursive formula is the same as the one given above.
2. **Definition and initialization of the table of results:** the decomposition of the problem shows that the recursive formula is defined according to two parameters:

n and k. Intuitively, the array of results, denoted by T [0..n, 0..k], is two-dimensional (d = 2), where each element T [i, j] will store the value C_i^j (see Figure 3.3).

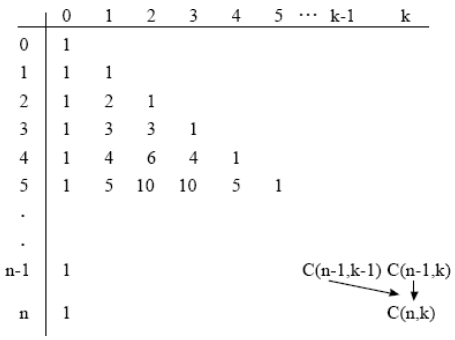


Figure 3.3. The array of results for the calculation of the binomial coefficient.

3. **Filling in the array of results:** the filling of the array of results is done according to both iterative and recursive approaches, as shown in the pseudo-codes below.
4. **Reading the solution:** by reading each cells T [n, k], we obtain an immediate answer about the value C_k^n. Now, if we want to get this value as a sum of terms, we just need to backtrack over the cells of array T based on the recursive formula.

Iterative approach	**Recursive approach**

Global variables
T [0 .. n, 0 .. k] : array of integers ;
Function C (n, k : integer) : integer
Begin
//initialization
For i ← 0 à n **do**
  T [i, 0] ← 1 ;
  T [i, i] ← 1 ;
End for
For i ← 2 à n **do**
  For j ← 1 à min (k, i - 1) **do**
  T [i, j] ← T [i -1, j -1] + T [i -1, j] ;
  End for
End for
Return T [n, k] ;
End
	Global variables
	T [0 .. n, 0 .. k] : array of integers (initialized to 0 except for basic cases to 1) ;
	Function C (n, k : integer) : integer
	Begin
	If (T [n, k] ≠ 0) **then**
	  Return T [n, k] ;
	Else
	  If (k = 0 OR k = n) **then**
	  v ← 1 ;
	  Else
	  v ← C (n - 1, k - 1) + C (n - 1, k) ;
	  End if
	End if
	T [n, k] ← v ;
	Return v ;
	End

4.3. Wooden planks cutting problem

Let consider a sawmill that sells wooden planks according to their lengths: the selling price of a wooden plank of length i is p_i. When it receives as input a wooden plank of length n, it can either derive the profit / price p_n directly, or seek to cut it into k pieces to derive several sub-planks of length i_1, i_2, …., i_k (with $i_1 + i_2 + \ldots + i_k = n$) and obtain as profit the sum $p_{i1} + p_{i2} + \ldots + p_{ik}$ of the selling prices of the sub-planks. For the sawmill problem, it is asked to determine the solution that will guarantee a maximum profit for any given wooden plank of length n.

Exemple

Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

Possibilities for a wooden plank of length $i = 4$.

- No cutting: profit 9
- Cutting 1 + 3: profit 1 + 8 = 9
- Cutting 2 + 2: profit 5 + 5 = 10
- Cutting 3 + 1: profit 8 + 1 = 9
- Cutting 1+1+2: profit 1 + 1 + 5 = 7
- Cutting 1+2+1: profit 1 + 5 + 1 = 7
- Cutting 2+1+1: profit 5 + 1 + 1 = 7
- Cutting 1+1+1+1: profit 1 + 1 + 1 + 1 = 4.

Optimal solution: cutting 2+2, i.e. 2 pieces of length 2 with a total profit of 10.

1. **Recursive formula:** let r_n be the maximum profit achievable for a wooden plank of length n, with $r_0 = 0$. A possible recursive formula is: $r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$. It is obtained by considering that what we get in the end is a piece at the left end of length i and therefore at price p_i, and a rest of length $n - i$, which must be an optimal cut of a wooden plank of length $n - i$.
2. **Definition and initialization of the array of results:** the decomposition of the problem shows that the recursive formula is defined according to a single parameter: the length of a plank n. Intuitively, the array of results, denoted by $R[0..n]$, is one-dimensional ($d = 1$), where each element $R[i]$ will store the maximum profit achievable for a plank of length i.

3. **Filling in the array of results:** the filling of the array of results is done according to both iterative and recursive approaches, as shown in the pseudo-codes below.
4. **Reading the solution:** by reading any given cell $R[i]$, we obtain an immediate answer about the maximum profit achievable for a plank of length i. However, if we want to know the different cuts of each plank of length i leading to recording an optimal profit, it is necessary to add an array $S[0 .. n]$ which keeps the index of the left plank; the right part is easily deduced in the same way.

Iterative approach

```
Global variables
S [0 .. n] : array of integers ; // lengths
P [0 .. n] : array of integers ; // profits
R [0 .. n] : array of integers ; // max profits
Function Cut (n : integer) : integer
Begin
//initialization
R [0] ← 0 ;
For j ← 1 à n do
  q ← −∞ ;
  For i ← 1 à j do
    If (q < P [i] + R [j - i]) then
       q ←  P [i] + R [j - i] ;
       S [j] ←  i ;
    End if
  End for
  R [j] ← q ;
End for
Return R [n] ;
End
```

Recursive approach

```
Global variables
S [0 .. n] : array of integers ; // lengths
(initialized to −∞)
P [0 .. n] : array of integers ; // profits
R [0 .. n] : array of integers ; // max profits
Function Cut (n : integer) : integer
Begin
If (R [n] ≥ 0) then
  Return R [n] ;
Else
    if (n = 0) then
       q ← 0 ;
    Else
       q ← −∞ ;
       For i ← 1 à n do
          If (q < P [i] + Cut (n – i)) then
             q ←  P [i] + Cut (n – i) ;
             S [n] ← i ;
          End if
       End for
    End if
    R [n] ← q ;
End if
Return q ;
End
```

If we want to display the lengths of the different pieces of cutting a plank of length n (reading a solution), we just have to run the following pseudo-code:

```
While (n > 0) do
    Print ("a piece of length: ", S [n]) ;
    n ← n - S [n] ;
End while
```

4.4. Maximum path sum in a pyramid of numbers

In a pyramid of numbers, we seek to maximize the sum of the numbers crossed starting from the top of the pyramid and descending in stages one-by-one. In the example shown in Figure 3.4, the maximum corresponds to the path colored in red (3+7+4+9=23).

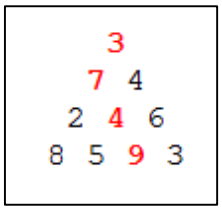


Figure 3.4. The maximum path sum in a typical pyramid of numbers.

Practically speaking, the pyramid of numbers can be modeled as a lower triangular matrix P [1..n, 1..n], n is the number of stages as given in Figure 3.5.

3	**0**	**0**	**0**
7	**4**	**0**	**0**
2	**4**	**6**	**0**
8	**5**	**9**	**3**

Figure 3.5. The lower triangular matrix related to the pyramid shown in Figure 3.4.

We denote by $S(i, j)$ the maximum sum corresponding to cell (i, j) in matrix P. Each value $S(i, j)$ depends on the value of cell (i, j) (i.e. $P[i, j]$) and the values of the maximum sum of its left and right children (i.e. $S(i+1, j)$ and $S(i+1, j+1)$). Consequently, the recursive definition of the maximum path sum for each cell (i, j) is given by the following formula:

$$S(i,j) = \begin{cases} P[i,j] & \text{when } i = n \\ P[i,j] + \max\left(S(i+1,j), S(i+1,j+1)\right) & \text{when } i < n \end{cases} \quad (1 \le i, j \le n)$$

Thus, it is asked to write a function that calculates the sum of all maximum paths $S(i, j)$, in particular $S(1, 1)$, using the principle of dynamic programming. To achieve this goal, we rely on the steps defined in section 4.

1. **Recursive formula:** the recursive formula is the same as the one given above.

2. **Definition and initialization of the table of results:** the decomposition of the problem shows that the recursive formula is defined according to two parameters: i and j. Intuitively, the array of results, denoted by T [1..n, 1..n], is two-dimensional (d = 2), where each element T [i, j] will store the value S (i, j).
3. **Filling in the array of results:** the filling of the array of results is done according to both iterative and recursive approaches, as shown in the pseudo-codes below.
4. **Reading the solution:** by reading each cells T [i, j], we get an immediate answer about the maximum value in position (i, j) in the pyramid. Now, if we want to get this value as a sum of terms, we need to add an array N [1..n, 1..n] such that each cell N [i, j] stores the column number of the cell maximizing S (i, j) while taking as value either j or j + 1.

Iterative approach	**Recursive approach**

Global variables
P [1 .. n, 1 .. n] : array of integers ;
N [1 .. n, 1 .. n] : array of integers ;
T [1 .. n, 1 .. n] : array of integers ;
Function S () : integer
Begin
//initialization
For $i \leftarrow$ 1 à n **do**
  T [n, i] $\leftarrow$ P [n, i] ;
End for
For $i \leftarrow n$ - 1 à 1 **do**
  For $j \leftarrow i$ à 1 **do**
  // we can modify this line of code to save j or j + 1 in N [i, j]
  $v \leftarrow$ max (T[i+1, j], T[i+1, j+1]) ;
  T [i, j] $\leftarrow$ P [i, j] + v ;
  End for
End for
Return T [*1*, *1*] ;
End
	Global variables
	P [1..n, 1..n] : array of integers ;
	N [1 .. n, 1 .. n] : array of integers ;
	T [1..n, 1..n] : array of integers (initialized to -1 except for basic cases i=1..n: T [n, i] $\leftarrow$ P [n, i]);
	Function S (i, j : integer) : integer
	Begin
	If (j = n + 1) **then**
	  Return 0 ;
	Else
	  If (i = n **OR** T [i, j] ≠ -1) **then**
	  Return T [i, j] ;
	  Else
	  // we can modify this line of code to save j or j + 1 in N [i, j]
	  $v \leftarrow$ max (S (i+1, j), S (i+1, j+1)) ;
	  T [i, j] $\leftarrow$ P [i, j] + v ;
	  End if
	End if
	Return T [i, j] ;
	End

If we want to display the path of the minimum sum starting from the top, we just have to run the following pseudo-code:

```
j ← N [1, 1] ;
Print ("Element: [1, 1]") ;
For i ← 1 à n - 1 do
    Print ("Element: [", (i + 1), ", ", j," ]") ;
    j ← N [i, j] ;
End for
```

5. Advantages and drawbacks of dynamic programming

Algorithms based on dynamic programming have certain advantages over other algorithmic methods, including:

- **Optimality:** algorithms based on dynamic programming always retain the optimal solutions when it is asked to solve optimization problems.
- **Dependency management between calculations:** dynamic programming better manages the dependency links between calculations. Indeed, we only need to store the elements used to solve the next sub-problems, and thus get rid of most of the old sub-problems which can hinder the progress of the algorithm.
- **Facilities offered by memoization:** the technique of memoization has several advantages such as the ease of coding and the use of a cache to obtain responses.

Algorithms based on dynamic programming also have some drawbacks; cite as examples:

- **Design difficulties:** designing dynamic programming algorithms to solve complex problems is sometimes very complicated, as it requires a precise decomposition of the problem into sub-problems and a deep understanding of the relationships between the sub-problems.
- **Storage space:** dynamic programming may require a lot of storage space to store intermediate results; this is problematic for large problems. It may also lead to performance issues because memory management may become a bottleneck for such algorithms.
- **Application limitations:** dynamic programming is not a general solving-method to all optimization problems. Some problems classes may not be suitable for dynamic programming as they lack the properties of recurrent sub-problems and well-defined transition relations.

Chapter IV: Backtracking

Objective:

This chapter aims to present the principles of the **backtracking method** as well as its use for solving constraint satisfaction problems. It also provides a theoretical comparison with dynamic programming. Next, it shows the application aspects of backtracking algorithms to solve some typical examples. Finally, this chapter presents some methods to improve the design of such algorithms.

1. Constraints satisfaction problems

- A constraint satisfaction problem (CSP) is defined by a triplet (X, D, C), where:
  - $X = \{x_1, \ldots, x_n\}$ is a finite set of variables to solve,
  - D is a function that associates to each variable x_i a domain of definition $D(x_i)$, i.e. set of values that variable x_i can take.
  - $C = \{c_1, \ldots, c_m\}$ is a finite set of constraints on variables $x_{i=1..n}$.
- We call ***state*** any assignment of values (i.e. evaluation) for some or all of variables $x_{i=1..n}$. This is a set of pairs (variable, values): $A = \{(x_i, v_i) / x_i \in X \text{ and } v_i \in D(x_i)\}$. An evaluation is said to be ***partial*** if it corresponds to a subset of variables and ***total*** (also called ***complete***) otherwise. An evaluation is said to be ***consistent*** if it shows no violation of constraints (i.e. satisfies all the constraints).
- A solution to a CSP problem is a complete and consistent evaluation. Moreover, sometimes the solution must optimize a given objective function.

- **Example:** let consider a CSP defined by a triplet (X, D, C) such that:
  - $X = \{x_1, x_2, x_3\}$ is the set of variables,
  - The domain D of the variables of set X: $D(x_1) = D(x_2) = D(x_3) = \{1, 2, 3\}$,
  - C is the set of constraints: $C = \{c_1 : x_1 = x_2 + x_3\}$.

The solutions of this problem are $A_1 = (2, 1, 1)$, $A_2 = (3, 1, 2)$ and $A_3 = (3, 2, 1)$.

2. Overview of the backtracking method

2.1. Principle of the backtracking method

Backtracking is a general algorithm that can be applied to several problems, in particular for CSPs. It is based on a systematic method that iterates over all possible configurations

of a search space. In a backtracking-based algorithm, a potential solution is usually encoded as a vector (n-tuple) $A = (x_1, x_2, \ldots, x_n)$ such that each variable $x_{i = 1 \,..\, n}$ is an element from a finite set X and defined over a set of values D (x_i). In the end, the algorithm retains one or more vectors satisfying certain criteria, linked to an optimization function in some cases. At each step, the backtracking algorithm forms a solution and checks whether there is still a chance for success or not. To this end, the algorithm starts with a given partial solution $sol = (x_1, x_2, \ldots, x_k)$ $(k \leq n)$, tries to extend it by adding another element x_{k+1} and decides whether the result is valid (i.e. potentially extendible partial solution) so that it may lead to a complete solution. If so, the algorithm recurs and continues hoping it reaches a complete solution. Otherwise, it backtracks by deleting the last element from the current partial solution and then tries another possibility for that position if possible.

Backtracking is a modified depth first search on an implicit tree of configurations, since the search for solutions is assisted by using a tree-like organization of the solutions space. The nodes are the different states and the arcs represent the transitions from one state to another. In the event that a given node cannot lead to consistent nodes (i.e. dead-end), the algorithm goes back (backtracks) to the parent nodes and proceeds to search on the next child, as shown in Figure 4.1.

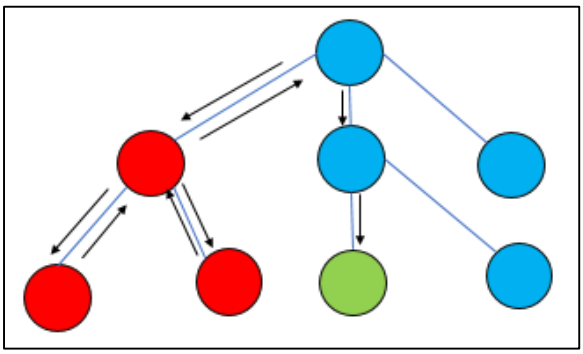


Figure 4.1. Tree of configurations (solutions space).

A backtracking algorithm does not actually need to build a tree but rather it only needs to keep track of the values in the current branch being investigated. That is why it is said that the state space tree exists implicitly in the algorithm as it is not entirely created in memory. Such a tree is called a decision tree (also known state-space tree). The root represents an initial state that precedes the search process. The nodes of any given level i of the decision tree represent the choices made to build the i^{th} component of the solution vector. A node d in a given decision tree is said to be *promising* if it is part of a partial solution that still leads to a complete solution; otherwise, it is said to be *unpromising*. Leaf nodes represent either dead-ends or complete solutions.

If the current node is promising, its child is generated by adding the next component to the current partial solution so that it (i.e. the child) undergoes some processing. In contrast, if the current node turns out to be unpromising, the algorithm backtracks to the node's parent and assigns another value to its last component. If no such option exists, the algorithm backtracks to the highest level in the tree, until reaching all complete solutions.

2.2. General scheme

In what follows, we present the most elementary recursive form of the backtracking method, in which the order of visiting the decision variables as well as the choice of their values (instantiation) are both fixed in advance.

```
Procedure Backtracking (non-instantiated-var, instantiated-var : set of variables)
Begin
   If (non-instantiated-var = ∅) then
      // we have assigned values to all decision variables successfully
      instantiated-var is a solution as it is a consistent evaluation ;
   Else
      next-decision-var ← choose the next variable from non-instantiated-var;
      For each value ∈ D (next-decision-var) do
         next-decision-var ← value ;
         If (instantiated-var ∪ {next-decision-var} is consistent) then
            Backtracking (non-instantiated-var - {next-decision-var},
            instantiated-var ∪ {next-decision-var}) ;
         End if
      End for
   End if
End
```

The backtracking procedure takes as input parameters two sets of decision variables, denoted by *non-instantiated-var* and *instantiated-var*. Variable *non-instantiated-var* keeps track of the decision variables that are not yet instantiated while variable *instantiated-var* keeps track of the decision variables that are already instantiated (i.e. with assigned values). Initially, the backtracking procedure is invoked in such a way that variable *non-instantiated-var* contains the set of all decision variables to be instantiated while variable *instantiated-var* is initialized with the empty set (i.e. *instantiated-var* = Ø). Through recursive calls, the execution scheme of the backtracking procedure oscillates

between moving forward and backtracking on previously instantiated decision variables (rollback), in order to take other paths guided by the new values (instances) assigned.

3. Backtracking *vs* dynamic programming

Backtracking and dynamic programming are two of the most widely used paradigms for solving different complex problems. Some differences are given in the table below. It should be noted that we will not provide a comparative between backtracking and greedy methods as the latter can be seen as a branch in the state tree for the former.

Feature	**Backtracking**	**Dynamic programming**
Feasibility	It builds a solution incrementally, one piece at a time while removing those elements that fail to satisfy the constraints until finding the solutions.	It makes a decision at each step taking into account the current problem and the solution to already solved sub problems to calculate the optimum.
Application	It always gets the sought solutions for CSP, optimization problems and enumerations as it cuts all possible paths.	It always gets the optimal solutions for optimization problems as it considers all possible cases and then choose the best.
Recursion	It draws the braches of the state tree by eliminating unpromising choices and proceeding to promising ones	It is based on recursive formulas that use previously calculated states (a problem is recursively defined based on other sub-problems).
Space / time complexity	It is more efficient as there is no need to store all partial results thanks to recursive calls. Nevertheless, it generally run slower.	It requires a table to store all partial results which leads to an increase in space complexity. Nevertheless, it generally runs faster.
Fashion	It computes the solutions through a systematic search in solutions spaces according to a depth-first search with any bounding function.	It computes the solutions in bottom-up or top-down by synthesizing them as smaller optimal sub solutions.

4. Classic examples

Now, we present some typical cases of backtracking algorithms applications to solve certain computation tasks.

4.1. N-Queens problem

The N-Queens problem consists in placing N chess queens on an $N \times N$ chessboard in a way that there will be no queens attacking each other. In other words, any solution requires that no two queens share the same row, column or diagonal, as shown in Figure 4.2.

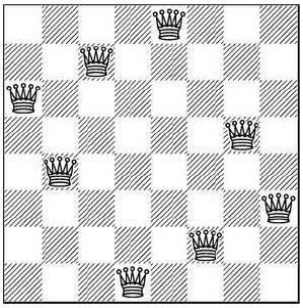


Figure 4.2. A typical solution for N-Queens problem using an 8×8 chessboard.

- **Problem formulation**

To solve this problem, we assume that each queen is fixed to a given column so that it can only change its position at rows. In the following, we specify the decision variables, their definition domain and the constraints imposed on them.

- Decision variables set $X = \{x_{i = 1.. N}\}$: each variable x_i designates whether the i^{th} queen is placed in a given row or not yet.
- Domain of definition of the decision variables: $D\ (x_{i = 1.. N}) = \{0, \ldots, N\}$. If $(x_i = 0)$ then the i^{th} queen is not yet placed; otherwise, it is placed at row with the value assigned to x_i.
- Constraints on the decision variables: we have to find all possible placements in such a way that:
  - The queens must be on different rows: $C_1 = \{\forall\ i, j \in \{0, 1, \ldots, N\}$, if $i \neq j$ then $x_i \neq x_j\}$.
  - The queens must be on different ascending diagonals: $C_2 = \{\forall\ i, j \in \{0, 1, \ldots, N\}$, if $i \neq j$ then $x_i + i \neq x_j + j\}$.
  - The queens must be on different descending diagonals: $C_3 = \{\forall\ i, j \in \{0, 1, \ldots, N\}$, if $i \neq j$ then $x_i - i \neq x_j - j\}$.

The pseudo-code below describes the steps of the corresponding backtrack algorithm, according to the general scheme presented above.

```
Algo-back-track (X : array [1 .. N] of integers, i: integer)
j : integer ;
OK : Boolean ;
Begin
        If (i = N + 1) then
                //X is a potential solution (valid queen placement)
        Else
                For j ← 1 to N do
                        X [i] ← j ;
                        OK ← check-position-constraints (X, i, j) ;
                        If (OK) then
                                Algo-back-track (X, i + 1) ;
                        End if
                End for
        End if
End
```

- **Example**

Let consider a chessboard of 4×4. The backtracking algorithm execution paths are illustrated in Figure 4.3.

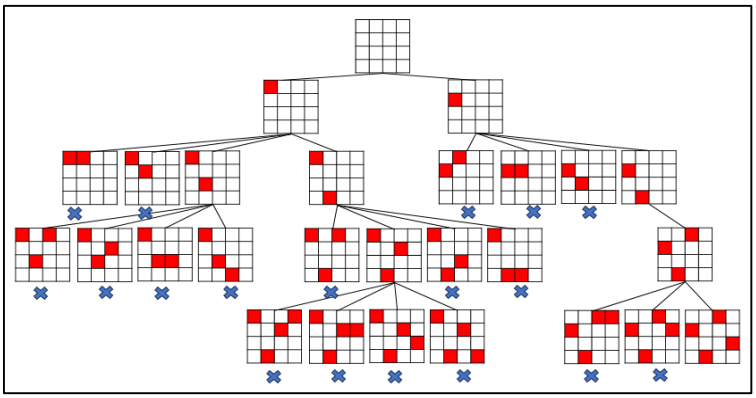


Figure 4.3. Execution paths of the backtracking algorithm on a 4×4 chessboard.

Note that:

- A_1 = (2, 4, 1, 1) is not a solution because the queens placement is complete but not consistent as constraint C_1 is not satisfied.
- A_2 = (1, 3, -, -) is a promising partial evaluation while A_3 = (1, 3, 3, -) is an unpromising partial evaluation.
- A_4 = (2, 4, 1, 3) is a solution.

4.2. Graph coloring problem

Let consider an undirected graph G and a given number m. It is asked to determine all graph coloring configurations for graph G using m colors so that no two adjacent vertices are colored with the same color. Here, coloring a graph means assigning colors to vertices.

- **Problem formulation**

This is a CSP, for which we need to specify the decision variables, their definition domain and the constraints imposed on them. Let $S = \{s_1, s_2, .., s_n\}$ be a set of n vertices, m be the maximum number of colors and M [1 .. n, 1 .. n] be the adjacency matrix for graph G.

- Decision variables set $X = \{x_{i=1..n}\}$: each variable x_i designates whether vertex s_i is colored or not yet.
- Definition domain of the decision variables: $D(x_{i=1..n}) = \{0, 1, \ldots, m\}$. If $(x_i = 0)$ then s_i is not yet colored; otherwise, it is colored with the value assigned to x_i.
- Constraint on the decision variables: $C = \{\forall s_i, s_j \in S$ if $M[i, j] = 1$ then $x_i \neq x_j\}$

The pseudo-code below describes the steps of the corresponding backtrack algorithm, according to the general scheme presented above.

```
Global variables
Constant m;
M : array [1 .. n, 1 .. n] of integers ;
Algo-back-track (X : array [1 .. n] of integers, i: integer)
i : integer ;
OK : Boolean ;
Begin
        If (i = n + 1) then //X is a potential solution (valid graph coloring)
        Else
                For j ← 1 to m do
                    X [i] ← j ;
                    OK ← check-color-consistency (X, i, j) ;
                    If (OK) then
                                Algo-back-track (X, i + 1) ;
                      End if
                End for
        End if
End
```

Example

Let G be the graph given in Figure 4.4 and $m = 4$ is the maximum number of colors.

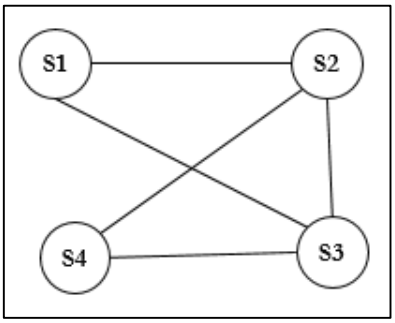


Figure 4.4. A typical graph with four vertices and five edges.

Note that:

- $A_1 = (2, 4, 1, 0)$ is a partial evaluation as x_4 does not refer to any color.
- $A_2 = (1, 3, 2, 2)$ is not a solution because it is not a consistent evaluation as the constraint on adjacent vertices coloring is not satisfied.
- $A_3 = (2, 4, 1, 3)$ and $A_4 = (2, 1, 2, 3)$ are potential solutions with respect to the constraints on adjacent vertices coloring. Even so, it is possible to compare between the different solutions according to the total number of colors used, thus keeping those colored using a smaller number.

4.3. Modified knapsack problem (inspired from the original formulation)

In this version of the knapsack problem, it is asked to purchase a subset from a set of n items with given values and weights. Each item i is worth v_i and weighs w_i. The aim is to maximize the number of purchased items while taking into account that the maximum capacity of the container is W and the purchase budget is B.

- **Problem formulation**

This is a CSP that requires optimizing an objective function (the number of purchased items). Therefore, we need to specify the decision variables to solve, their domain of definition and the constraints imposed on them, in addition to the objective function to be optimized.

- Decision variables set $X = \{x_{i = 1 \,..\, n}\}$: each variable x_i designates whether item i is chosen to be purchased and therefore put in the container or not.
- Domain of definition of the decision variables: $D\ (x_{i = 1..\, n}) = \{0, 1\}$; if $(x_i = 1)$ then item i is purchased and put in the container and quite the opposite for $x_i = 0$.
- Constraints on the decision variables: we have to maximize the number of purchased items:

$$f(x) = \sum_{i=1}^{n} x_i$$

Subject to:

$$C_1: \sum_{i=1}^{n} w_i \times x_i \leq C$$

$$C_2: \sum_{i=1}^{n} v_i \times x_i \leq B$$

The pseudo-code below describes the steps of the corresponding backtrack algorithm, according to the general scheme presented above. We note here that it is possible to make some code optimizations in order to improve its overall complexity (e.g. adaptive computing of the price and weight for partial evaluations).

```
Global variables
nb-max : initialized to 0 ;
X-max : array [1 .. n] of integers ;
Algo-back-track (X : array [1 .. n] of integers, i: integer)
weight, price, nb, j : integer ;
Begin
    If (i = n + 1) then //X is a potential solution but not necessarily optimal
        nb ← ∑_{j=1}^{n} x_j ;
        If (nb > nb-max) then
            nb-max ← nb ;
            X-max ← X ; //X is better than current optimal solution.
        End if
    Else
        For j ← 0 to 1 do
            X [i] ← j ;
            weight ← ∑_{j=1}^{i} w_j x_j ;
            price ← ∑_{j=1}^{i} v_j x_j ;
            If (weight ≤ W AND price ≤ B) then
                Algo-back-track (X, i + 1) ;
            End if
        End for
    End if
End
```

- **Example**

Consider a container of a maximum capacity $W = 10$, a budget $B = 20$ and a set of 4 items whose values and weights are given in the table below.

Item id	1	2	3	4
Weight	5	3	3	2
Value	7	5	8	7

Note that:

- $A_1 = (0, 1, 1, 0)$ is a potential solution but not optimal since it includes only two items.
- $A_2 = (1, 1, 1, 0)$ is not a solution because it is not a consistent evaluation as the constraint on weight is not satisfied.
- $A_3 = (1, 1, 0, 1)$, $A_4 = (1, 0, 1, 1)$ and $A_5 = (0, 1, 1, 1)$ are optimal solutions with respect to the number of contained items; it is possible to store them in a list. Even so, it is possible to compare between these solutions by involving the total price and / or the total weight of the chosen items, thus keeping the solutions with the smallest total price and / or total weight.

4.4. Subset-sum problem

The subset sum problem is an important decision problem in complexity and cryptology fields. The problem is described as follows: let S be a set of n integers: $S = \{s_1, s_2, .., s_n\}$. The goal is to find all subsets of set S whose elements sum is equal to a given integer d.

- **Problem formulation**

This is a CSP; thus, we need to specify the decision variables, their definition domain and the constraints imposed on them.

- Decision variables set $X = \{x_{i = 1 .. n}\}$: each variable x_i designates whether element s_i is considered for addition or not.
- Domain of definition of the decision variables: $D(x_{i = 1.. n}) = \{0, 1\}$; if $(x_i = 1)$ then element s_i will be considered for addition and quite the opposite for $x_i = 0$.
- Constraint on the decision variables: $C = \{C_1: \sum_{i = 1}^{n} s_i \times x_i = d\}$.

The pseudo-code below describes the steps of the corresponding backtrack algorithm, according to the general scheme presented above.

```
Global variables
Constant d ; S : Set of n integers ;
Algo-back-track (X : array [1 .. n] of integers, i : integer)
s, j : integer ;
Begin
    If (i = n + 1) then //X is a potential solution
            s ← ∑_{j=1}^{n} s_j x_j ;
            If (s = d) then //X is a solution
            End if
    Else
        For j ← 0 to 1 do
            X [i] ← j ;
            s ← ∑_{j=1}^{i} s_j x_j ;
            If (s ≤ d) then
                Algo-back-track (X, i + 1) ;
            End if
        End for
    End if
End
```

- **Example**

Let consider a set $S = \{2, 4, 6, 10, 12\}$ and given integer $d = 12$. Note that:

- $A_1 = (1, 0, 0, 0, 1)$ is not a solution because it is not a consistent evaluation as the sum of elements is not equal $d = 12$.
- $A_2 = (1, 0, 0, 1, 0)$ and $A_3 = (1, 1, 1, 0, 0)$ are solutions. Even so, it is possible to compare between the different solutions according to the cardinal of the resulting subsets, thus keeping those with the smallest cardinal.

5. Improving the basic scheme of backtracking algorithms

Although backtracking easily iterates through all subsets or permutations of a set, their efficiency requires pruning dead or redundant branches whenever it is possible. This is due to the fact that CSP are often NP-complete. The general performances of the backtracking procedure depends mainly on:

- Problem formulation as well as the resulting possible branches through which the backtracking algorithm goes.

- Representation of solutions regarding the order in which the nodes making up the branches of solutions are visited (orders of the variables and values assignment).

Next, we show how to make improvements to the backtracking search process by involving two techniques: ***anticipation*** and ***heuristics***.

5.1. Anticipation

To improve the algorithm presented in section 2.2, one solution is to anticipate the consequences of the partial evaluations under construction on the domains of the variables which do not yet take values (empty variables). Indeed, we can check whether an empty variable x_i no longer has a value in its domain $D(x_i)$ so that it leads to a locally consistent state (i.e. the current partial evaluation remains still consistent). If so, there is no need to continue developing this branch, and therefore go back immediately to explore other possibilities. One way to implement this principle is to filter, at each stage of the search process, the domains of the unaffected variables by removing the "locally inconsistent" values. Depending on the number of empty variables, filtering can be performed at different levels of local consistency, which reduces more or less the domains of the variables, but which also takes more or less time.

In summary, the principle of anticipation consists in modifying the backtracking algorithm presented in Section 3, by simply adding a filtering step each time a value is assigned to a variable, which detects and thus avoids conflicting assignments as early as possible.

5.2. Heuristics

The algorithm presented in Section 2.2 chooses, at each step, the next variable to instantiate among the set of variables that are not yet instantiated; then, once the variable is chosen, it tries to instantiate it according to its domain values. Thus, it does not say anything about the order in which the variables should be instantiated, nor about the order in which the values should be assigned to the variables. These two orders help considerably change the efficiency of backtracking algorithms. Indeed, let imagine that, at each step, we have the advice of a "know-it-all – oracle" who tells us which value to choose without ever making a mistake; in this case, the solution would be found without ever turning back. Unfortunately, satisfying a CSP on finite domains is generally an NP-complete problem; thus, it is more than unlikely that this 100% reliable oracle could never be "programmed". To deal with this issue, it is possible to rely on heuristics so as to determine the order in which the variables and the values should be considered. A heuristic is an unsystematic rule (in the sense that it is not 100% reliable) which gives us indications on the direction to take in the state-tree. The heuristics concerning the order of instantiation of values

depend generally on the considered problem and consequently are difficult to generalize. On the other hand, there exist some heuristics to order the instantiation of variables which very often helps speed up the search process. The general idea is to instantiate the most ***"critical"*** variables first, i.e. those which are involved in many constraints and/or which only take very few values. The order of instantiation of variables can be:

- ***static,*** if it is fixed before starting the search. For example, the variables can be ordered according to the number of constraints relating to them. The idea is to instantiate the most constrained variables first, i.e. those which are involved in a large number of constraints.
- **or dynamic,** if the next variable to instantiate is chosen dynamically at each step of the search. For example, the ***"first-fail"*** heuristic consists in choosing, at each step, the variable whose domain has the smallest number of values locally consistent with the current partial evaluation. This heuristic is often combined with anticipation algorithms to filter the domains of the variables by keeping only the values that satisfy a certain level of local consistency.

6. Advantages and drawbacks of the backtracking method

Backtracking algorithms have some advantages, including:

- **Simplicity:** backtracking algorithms are easier to describe and code than other algorithms.
- **Efficiency:** the search for solutions by a backtracking algorithm is used to explore a tree of choices in an adaptive way without the need to build it completely. This is because the branches are built and destroyed as a result of moving forward and moving backward (rollbacks), respectively.
- **Performance:** backtracking algorithms allow systematic checking of all potential evaluations of the problem. This makes it possible to retain all potential solutions, including the optimal solutions in the case of optimization problems.

Backtracking algorithms also have some drawbacks; we cite as examples:

- **Algorithmic complexity:** the search for a solution by a backtracking algorithm is used to explore a tree of choices where it must sometimes be completely traversed. This would highly increase the process cost in terms of time complexity, depending on problems size.
- **Difficulties induced by recursion:** backtracking algorithms often lend themselves to recursive writing. This may lead to high runtime stack usage and resource overflow caused by recursive calls.

Chapter V: Probabilistic methods

Objective:

This chapter aims to present the principles, elements and theoretical basis of ***probabilistic methods (randomized algorithms)***. In particular, it emphasize the generation of pseud-random numbers due to their role in the design of randomized algorithms. This chapter also discusses their categories as well as their implication in solving some typical examples.

1. Deterministic algorithms *vs* probabilistic algorithms

An algorithm is said to be *deterministic* if the computation process always produces the same output each time it receives the same input (i.e. the same data set). This is because the algorithm always goes through the same sequence of states. In other words, only the example to be solved and the description of the algorithm (i.e. the set of instructions) determine the sequence of calculations performed, without resorting to other external factors such as random variables.

Nevertheless, by only relying on determinism, certain restrictions or even constraints can be imposed. Indeed, for some situations, one must relax the requirements by limiting oneself to admitting approximate results. In this case, the execution of the algorithm involves probabilistic choices guided by random selections (e.g. heads or tails of coin toss). Hence, we talk about a probabilistic algorithm (non-deterministic or even stochastic) whose execution uses a source of randomness by involving data obtained at random (random variables). As a result, the computation process produces different outputs each time it receives the same input due to the fact that it goes through various sequences of states.

In many situations, probabilistic algorithms are useful and can even be accurate. Let's take the example of a box containing n bricks of which approximately 10% are of size 1×2. Hence, we want to get a brick of size 1×2 by hand. By using a deterministic algorithm, we check the bricks one by one until finding a sought brick. Indeed, with a bit of luck, it may happen that no brick of size 1×2 is got on the first attempts, thus testing almost all the bricks before managing to find the desired one. The time complexity of such an operation is $O(n)$. By relying on probabilistic algorithms such as the algorithms based on Las Vegas or Monte Carlo approaches, it is possible to get a good answer with a higher probability.

2. Random and pseudo-random number generation

The simulation of a stochastic model requires the existence of a source of "random" numbers in order to generate the values taken by the random variables involved in the definition of the model. These numbers are generally generated according to a sequence of random numbers U1, U2, … playing the role of *uniform independent identically distributed random variables* over the interval [0, 1]. Such numbers can be obtained by physical processes such as the lottery wheel and the lighting at irregular intervals of a disc divided into 10 isometric sectors and numbered from 0 to 9.

However, in practice, we usually deal with pseudo-random numbers to designate random numbers, which in fact play a similar role to real sequences of identically distributed independent random variables.

2.1. Uniform distribution of numbers

Let a and b be two real numbers. The function *uniform* (a, b) returns a real number x chosen randomly, uniformly and independently, such that $a \leq x < b$.

Over integers i and j, *uniform* (i, j) returns a random integer k such that $i \leq k \leq j$, usually with probability $1/(j - i + 1)$.

The link between ***uniform* (*integers*)** and ***uniform* (*reals*)** is that $uniform_{integers}(i, j) = \lfloor uniform_{reals}(i, j + 1) \rfloor$.

On a non-empty finite set X, *uniform* (X) returns a randomly chosen element from X, usually with probability $1 / |X|$.

2.2. Algorithmic pseudo-random number generators

Pseudo-random numbers refer to random numbers that are able of being generated as a random-looking sequence from a *seed* (known also as *germ*). A sequence of numbers is said to be pseudo-random if it is generated deterministically but appears to have been generated randomly. A pseudo-random number generator is an algorithm implemented by a function which returns a new random numeric value on each call. The sequence of values returned must have good statistical properties so that it can be considered as a sequence of independent random variables with uniform distribution in a given interval.

Generally speaking, most algorithmic methods for generating pseudo-random numbers are characterized by a (S, f, g), where:

- S is a finite set,
- f is an application from S to itself,
- g is a function from S over [0, 1].

The generator works by iterating over function f from an initial value $s_0 \in S$, called seed or germ, chosen by the user. Sometimes, the seed has to satisfy some constraints in addition to being an element of set S. By noting $(x_n)_{n \geq 0}$ as the list of successive values (between 0 and 1) returned by the generator, we have: $x_n = g(f \circ \ldots \circ f(s_0))$.

In practice, the generator keeps in memory only its current state $s_n \in S$, initialized by s_0, and updated during each new call by means of the recursive formula $s_n = f(s_{n-1})$; the value returned is actually equal to $g(s_n)$. Thus, the choice of seed entirely determines the sequence of the pseudo-random numbers produced by the generator. Once the seed is chosen, nothing random remains in the functioning of the generator whose outputs are nevertheless supposed to mimic independent random variables with a uniform law on [0,1].

2.3. Examples of pseudo-random number generators

2.3.1. The Von Neumann method

Proposed by Von Neumann in 1946, this pseudo-random number generator is known as the middle-square method. It is considered as the first method for automatically generating pseudo-random numbers. Today, it is very little used or even unused as several more efficient techniques have emerged; thus, it is presented only for historical interest. The principle of this generator is simple: choose a number, square it and finally take the digits in the middle to generate a pseudo-random number. The result serves as a seed for generating the next pseudo-random number. The pseudo-code below summarizes the steps of this method.

Von Neumann's method for generating pseudo-random numbers

1. $k \leftarrow$ initialize the seed of the generator with a random number of n digits ;
2. $r \leftarrow k^2$;
3. **If** (number of digits of $r < 2 \times n$) **then**
   Pad with leading zeros until the number of digits of r becomes $2 \times n$;
   End if
4. $r' \leftarrow$ extract the n middle digits of r ;
5. $k \leftarrow r'$; // r' serves as a seed for generating the next pseudo-random number
6. Start from Step 2.

- **Example**

Consider the number 1111, hence $n = 4$.

1. $(1111)^2 = 01234321$

2. We extract the n middle digits: 2343, which represents the output of the generator.
3. $(2343)^2 = 05489649$
4. We extract the n middle digits: 4896 (the next pseudo-random number).
5. $(4896)^2 = 23970816$
6. We extract the n middle digits: 9708.
7. … and so on.

Starting from seed 1111, the following sequence of pseudo-random numbers is generated: 2343, 4896, 9708, 2452, 0123, 0151, 0228, 0519, 2693, 2522, 3604, 9888, 7725, 6756, …
Although this method is simple in its writing, the period of the middle square is small. In addition, the outputs may sometimes produce dead-ends thus constituting an absorbing state of the algorithm (e.g. the sequence 0000).

2.3.2. Fibbonacci-based method

This method uses the Fibonacci sequence modulo M (the maximum desired value), according to the recursive formula: $x_n = (x_{n-1} + x_{n-2}) \bmod M$; each term of the sequence is the sum of the two terms which precede it modulo a given number M. It is therefore an additive congruence. The value of M is fixed beforehand; likewise, the initial values x_0 and x_1 are given as input so that they constitute a seed for generating the pseudo-random numbers which follow.

- **Example**

Starting from numbers $x_0 = 1$ and $x_1 = 2$, and $M = 100$ as modulo, the resulting sequence of pseudo-random numbers is as follows: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 44, 33, 77, 10, 87, 97, 84, 81, 65, 46, 11, 57, 68, 25, 93, 18, 11, 29, 40, …..
Although this generator is very simple to implement and consumes few resources, it nevertheless shows a strong correlation between successive values.

2.3.3. The congruent linear method

It represents the most widely used algorithm for generating pseudo-random numbers; its principle is based on a very simple recursive formula: $x_{n+1} = (a \times x_n + c)$ modulo m. This formula also allows making a jump of step k between the terms, as follows:
$x_{n+k} = (a \times k \times x_n + (a^k - 1) \times c \,/\, (a - 1))$ modulo m

- m : the modulo defining the largest value supported by this system ($m > 0$)
- a : the multiplier ($0 \leq a < m$)
- c : the jump (the step) ($0 \leq c < m$)
- x_0 **:** the starting value which actually represents the generator seed ($0 \leq x_0 < m$).

- **Example**

By using a seed $x_0 = 1$, a multiplier $a = 2$, a jump $c = 3$ and a modulo $m = 100$, the resulting sequence of pseudo-random numbers is as follows: 1, 5, 13, 29, 61, 25, 53, 9, 21, 45, 93, 89, 81, 65, 33, 69, 41, 85, 73, 49, 1, 5, 13, 29, …

2.4. Interests of algorithmic sources

Algorithmic methods for generating pseudo-random numbers are interesting for several reasons, among which we cite:

- **repeatability:** once the seed has been chosen by the user (and stored), it is possible to fully reproduce the sequence of pseudo-random numbers generated. This is very important, for example, to check the results obtained by simulation or to debug simulation codes.
- **ease of use:** in most implementations, the generation of pseudo-random numbers is fast and without any limitation with respect to the number of calls to the generator used.
- **standardization:** using a standardized procedure for generating pseudo-random numbers allows having a reliable information on the quality and performance of the contributions made by various researchers and users, in the form of scientific publications and documentation.

2.5. Application fields of pseudo-random numbers

Pseudo-random numbers are used in various applications fields, among which we cite:

- **Confidentiality of exchanges on wireless networks:** confidentiality during exchanges through a wireless network is a key aspect for protecting the security of the data circulating. In this context, most protection mechanisms are based on defining encryption keys of different lengths. These keys are specified at access point and client level so that they create pseudo-random numbers used to encrypt the transmitted data, and therefore ensure their confidentiality.
- **Encryption systems:** cryptanalysis ensures a high level of reliability regarding the encryption methods adopted, in particular stream ciphers. The latter consists in adding - bit by bit - to the clear message a pseudo-random binary sequence of the same length.
- **Simulation based on queuing theory:** queuing theory is a mathematical theory in the field of probability, which studies the optimal solutions for managing queuing (queues). The incoming flows and service mechanisms are usually simulated using pseudo-random numbers.

3. Categories of probabilistic algorithms

In classical algorithms, we are often interested in two important aspects:

- **The correctness:** does the algorithm return the correct result?
- **Termination and complexity:** does the algorithm always terminate and in how many operations with respect to the inputs size?

In probabilistic algorithms (randomize algorithms), these aspects are probabilized. Probabilization of the results affects the correctness of the algorithm in such a way that the complexity is maintained with respect to the corresponding deterministic algorithm (the execution time is the same for both algorithms) but not the result. In contrast, the probabilisation of the termination affects the execution time in such a way that the correctness of the result is maintained with respect to the corresponding deterministic algorithm (the result is the same for both algorithms) but not the execution time. In the following, four categories of randomized algorithms are presented along with explanatory examples on their application to problem solving. In fact, this classification is a controversial topic as some researchers and scholars claim that there are four classes (numerical algorithms, Sherwood algorithms, Monte Carlo algorithms and Las Vegas algorithms) while others consider only two (Monte Carlo algorithms and Las Vegas algorithms) while attaching the other classes to these two classes.

3.1. Numerical algorithms

The answer of the algorithm is always approximate. However, its precision is all the better on average as the time available to the algorithm is large. This class of algorithms is used to approximate the solutions of numerical problems (e.g. calculating π, numerical integration, etc.).

- **Example: throwing darts at a square target (calculation of π)**

The experiment consists of throwing n darts at a square target and counting the number k of those falling inside the circle inscribed in this square. Let r denote the radius of the circle (see Figure 5.1).

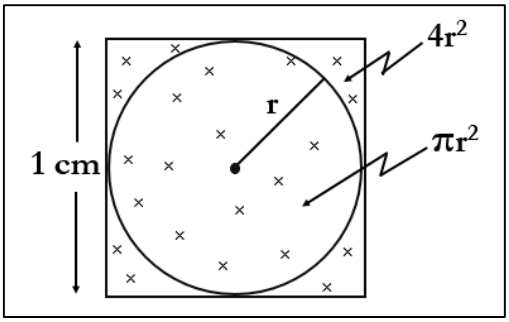


Figure 5.1. Square target of dimension 1× 1 cm.

If we randomly and uniformly choose a point p in the square, we ask ourselves what is the probability that point p is in the circle. The answer is obvious: $\pi r^2 / 4r^2 = \pi / 4$.
Now, we choose n points randomly, uniformly and independently in the square, such that the number of points in the circle is denoted by k, hence $E(k) = n\pi / 4$. According to the weak law of large numbers, we have: $\lim_{n\to\infty} |k - E(k)| \geq \varepsilon$. It follows that: $\pi \approx 4k / n$. This experiment is simulated by the algorithm given below. The returned approximation of π is all the better as n is large.

```
Function Darts (n : integer) : real.
Begin
k ← 0 ;
For i ← 1 to n do
   //Throw a dart
   x ← uniform (0, 1) ;
   y ← uniform (0, 1) ;
   //Check if it is in the circle
  If (x² + y² ≤ 1) then
      k ← k + 1 ;
  End if
End for
Return 4×k / n ;
End
```

3.2. Sherwood's algorithms

These algorithms always return an exact answer. For such an algorithm, there exists a deterministic algorithm already known to solve the problem treated but which is much faster on average than in the worst case. Thus, the use of randomness aims to reduce this difference between good and bad cases.

- **Example: search for the k^{th} smallest element of an array of n elements.**

The strategy of the probabilistic algorithm for finding the k^{th} smallest element of an array T of n elements consists in randomly choosing a pivot among the elements of array T, as illustrated in the following pseudo-code.

```
Function Sherwood-selection (T [1 .. n]: array, k: integer): element from T
Begin
  //Return the k^th smallest element of T ; we assume that 1 ≤ k ≤ n
  i ← 1 ;
  j ← n ;
  While (i < j) do
     m ← T [uniform (i, j)] ;
     //pivot around m: after this operation, the elements of T [i ..u -1] are all lower than
     m, those of T [u .. v] are equal to m, and those of T [v+1 .. j] are greater than m
     Partition (T, i, j, m, u, v) ;
     If (k < u) then
        j ← u - 1 ;
     Else
        If (k > v) then
           i ← v + 1 ;
        Else
           i ← k ;
           j ← k ;
        End if
     End if
  End while
  Return T [i] ;
End
```

3.3. Monte Carlo algorithms

These algorithms always return an answer but not always right, i.e. they always give an answer that is not always accurate. This is why it is very difficult to determine whether the answer obtained is correct or not. The probability of success of these algorithms in terms of correct answer is all the better as the time available to them is large.

- **Example: majority table problem**

Let T [1 .. n] be an array of n elements. An element x is said to be *majority* in array T if and only if the number of elements whose values are equal to x is greater than n / 2. Similarly, array T is said to be *majority* if it contains a majority element. The probabilistic

algorithm for performing this task is given below. This algorithm is 1/2 correct and true-biased with $X = \{T / T \text{ is not majority}\}$.

```
Function majority (T [1 .. n]: array): Boolean
Begin
    //draw an item x at random
    i ← uniform (1, n) ;
    x ← T [i] ;
    // count the number k of elements equal to x
    k ← 0 ;
    For j ← 1 to n do
        If (T [j] = x) then
            k ← k + 1 ;
        End if
    End for
    Return (k > n/2);
End
```

A call to function majority on array T leads to two possible results. In the case where the function return true, array T is actually a majority (i.e. function majority is true-biased). Otherwise, we cannot draw a conclusion but probability (x is in the minority | T is majority) = $(1 - p) < 1/2$.

3.4. Las Vegas Algorithms

They never return an incorrect answer but they may not find an answer. Randomness is thus restricted to the internal control of the calculation and in no way affects the result. These algorithms can also solve some problems for which no efficient deterministic algorithms are known.

The main element of a probabilistic nature attached to such an algorithm is typically its execution time. The usual definition of a Las Vegas algorithm is that only the expectation of the computation time is finite; the execution time is random. Therefore, the answer is always correct and probably fast. The probability of success of such an algorithm is all the better as the time available is large. The probability of a failure can be made arbitrarily small (close to 0) by repeating the algorithm often enough.

- **Example: N-Queens problem**

The strategy of the adopted probabilistic algorithm consists in placing the queens randomly on successive lines while making sure that the queens already placed are not in conflict with each other. With the following algorithm, if variable *success* = *true* then array *trial* [1 .. *nb-queens*] contains the solution, as illustrated in the pseudo-code below.

```
Algorithm place-queens-LV (nb-queens: integer)
trial, ok : array [0 .. nb-queens] of integer ;
cols, diag-45, diag-135 : set ;
nb : integer ;
Begin
    success ← true ;
    initialize arrays trial and ok with zeros ;
    initialize sets cols, diag-45 and diag-135 to ∅
    For k ← 1 to nb-queens do
        nb ← 0 ;
        For j ← 1 to nb-queens do
            If ((j ∉ cols) and ((j - k) ∉ diag-45) and ((j + k) ∉ diag-135) then
                nb ← nb + 1 ;
                ok [nb] ← j ;
            End If
        End for
        If (nb > 0) then
            j ← ok [uniform (1, nb)] ;
            cols.add (j) ;
            diag-45.add (j - k) ;
            diag-135.add (j + k) ;
            trial [k] = j ;
        else
            success ← false ; break ;
        End if
    End for
    Return [success, trial] ;
End
```

4. Advantages and drawbacks of randomized algorithms

Randomized algorithms have some advantages over deterministic algorithms, including:

- **Simplicity:** randomized algorithms are known for their simplicity to understand and implement. Moreover, many deterministic algorithms are easily convertible into randomized algorithms.
- **Performance:** randomized algorithms are very efficient compared to deterministic algorithms. Indeed, for many problems, a randomized algorithm is the simplest, the fastest, or even both.
- **Time and space complexity:** several randomized algorithms use little execution time and space compared to deterministic algorithms. This is because they exhibit superior asymptotic bounds, which makes their complexity better than that of the corresponding deterministic algorithms.

Randomized algorithms also have some drawbacks; we cite as examples:

- **Reliability:** this is an important issue in many applications, as not all randomized algorithms always give correct answers. Moreover, some randomized algorithms may not terminate. Therefore, reliability concerns should be handled carefully.
- **Quality:** the quality of randomized algorithms depends heavily on the quality of the random number generator used as part of the algorithm.
- **Lack of design paradigm:** unlike other paradigms, randomized algorithms do not rely on a single design principle. Hence, it would be better if we came to think of randomized algorithms as those designed using a set of clear principles.

Chapter VI: Approximation algorithms

Objective:

This chapter aims to give an overview of ***approximation algorithms*** while highlighting their principles, elements and theoretical foundation. It also discusses their application aspects for solving several typical examples.

1. Solving NP-complete optimization problems

An optimization problem is a problem for which there exists a set of potential solutions among which one must select the optimal solution. Optimization problems can be either a maximization or a minimization of a cost function (see the formal definition of an optimization problem in Section 1 of Chapter 1).

The difficulty of solving an optimization problem depends on its complexity class. In particular, problems belonging to the NP-complete class are the most difficult. This is due to the fact that NP-complete problems do not admit algorithms that run in polynomial time, in order to find solutions. Indeed, although any solution of an NP-complete problem can be verified quickly (in polynomial time), to date, there is no efficient method to find such a solution. A classic example of NP-complete problems is the traveling salesman problem, whose goal is to determine the shortest paths to visit a large number of cities.

Depending on the instance size of a given optimization problem, there are three main solving-methods:

- **The exact resolution:** although this option allows finding the optimal solution to the problem treated, this is generally done while knowing a priori that the cost is most likely exponential in time. An example of such an approach is the exhaustive search through brute force algorithms.
- **Heuristic resolution:** according to which we build a solution at a lower cost, hoping that it shows good performances. However, there is no guarantee on the quality of the result obtained. An example of such a solving-approach is greedy algorithms.
- **Guaranteed resolution:** a solution is built at a lower cost so that its quality can be measured and is therefore guaranteed: *approximation algorithms* that we present and discuss throughout this chapter.

2. Overview on approximation algorithms

2.1. Basic idea of approximation algorithms

Approximation algorithms deal with NP-complete optimization problems. An approximation algorithm does not guarantee retaining the optimal solution but rather it allows getting a solution as close as possible to the optimum in a reasonable time. Some of the features of approximation algorithms are summarized as follows:

- They guarantee to run in polynomial time though they do not guarantee getting the most effective solutions.
- They are used to get an answer near the optimal solution. In addition, the quality of the retained solutions can be measured.
- They guarantee to seek out high accuracy and top quality solutions.

Polynomial complexity and bounded approximation factor are two desired properties for NP-problems. On the one hand, we can always calculate an optimal solution by enumerating all possible solutions but, unfortunately, the computation cost is often exponential in time in accordance with the growth of problems size; this has relatively little interest in practice. On the other hand, being able to say something about what the algorithm produces and guaranteeing an approximation factor regardless of the instances size are also two important criteria for approaching the optimal solution. This is because it is always possible to quickly calculate a solution arbitrarily far from the optimum (for example, by drawing it at random). It is important to differentiate between an approximation algorithm and a heuristic, especially since they both provide approximate solutions. The former is a polynomial time algorithm with some degree of guarantee while the latter does not necessarily satisfy these two features.

2.2. Notations

In what follows, we use the notations given below to provide definitions and formulas.

- ***Π*:** an NP-complete optimization problem
- ***I*:** an instance of problem *Π*.
- ***A* (*Π*, *I*):** the solution obtained for instance *I* of problem *Π* using algorithm *A*.
- ***OPT* (*Π*, *I*):** the optimal solution for instance *I* of problem *Π*.
- ***COST* (*SOL* (*Π*, *I*)):** returns the objective function value for a given solution *SOL* to instance *I* of problem *Π*.
- ***LB* (*Π*, *I*):** a lower bound on *COST* (*OPT* (*Π*, *I*)) for instance *I* of minimization problem *Π*.

- ***UB* (Π, *I*):** an upper bound on *COST* (*OPT* (Π, *I*)) for instance *I* of maximization problem Π.

2.3. Performance ratio (approximation factor)

Let *A* be an algorithm that deals with a given problem Π. *A* is a λ-approximation if:

1. It runs in polynomial time in accordance with instances size.
2. It always produces a solution which is, at worst, λ times worse than *OPT* (Π, *I*).

For a minimization problem Π, we have: $\forall I$, *COST* (*A* (Π, *I*)) $\leq \lambda \times$ *COST* (*OPT* (Π, *I*))

For a maximization problem Π, we have: $\forall I$, *COST* (*A* (Π, *I*)) $\geq \frac{1}{\lambda} \times$ *COST* (*OPT* (Π, *I*))

λ is called *approximation factor* ($\lambda \geq 1$ and $\lambda = 1$ for optimal algorithms).

We say that a given approximation factor λ is non-improvable if for each $\varepsilon > 0$, there exists no instance *I* of problem Π such that *COST* (*A* (Π, *I*)) $\leq (\lambda - \varepsilon) \times$ *COST* (*OPT* (Π, *I*)).

- **Example**

Consider a minimization problem Π and an algorithm *A* to solve it. By considering three instances *I*, *I'* and *I''*, the table below provides the optimal solutions, the solutions retained by algorithm *A* as well as the approximation factors λ.

Instance	Optimal solution	Retained solutions by *A*	Approximation factor λ
I	3	6	2
I'	8	12	1.5
I''	6	8	1.33

- **Question:** how to compare with an optimal solution that - by definition - we do not know how to calculate in a reasonable time?

Overall, it is often difficult to prove that a given algorithm results in a good approximation factor and thus produces a good output. However, what is more important in particular is that the optimum should not be much better. In fact, it is not always possible to figure out what the optimum should be. In this case, we have to prove lower/upper bounds on the optimal solution. Most often, these bounds are obtained from the problem structure but sometimes the solving-method helps as well.

Let *I* be an instance of a problem Π and *A* be an algorithm. We have:

$\forall I$, *COST*(*A* (Π, *I*)) $\leq \lambda \times$ *LB*(Π, *I*) $\Rightarrow$ ($\forall I$, *COST*(*A* (Π, *I*)) $\leq \lambda \times$*COST*(*OPT* (Π, *I*)) $\wedge$ Algorithm *A* is an approximation algorithm of factor λ) [for a minimization problem Π].

$\forall I$, *COST*(*A* (Π, *I*)) $\leq \frac{1}{\lambda} \times$ *UB*(Π, *I*) $\Rightarrow$ ($\forall I$, *COST* (*A* (Π, *I*)) $\leq \frac{1}{\lambda} \times$*COST*(*OPT* ($\Pi$, *I*)) $\wedge$ Algorithm *A* is an approximation algorithm of factor λ) [for a maximization problem Π].

2.4. Approximation schemes

Let A be an approximation algorithm for problem Π and ε, $c \in \mathbb{R}^+$ be two constants.

Algorithm A is ***polynomial time*** if its time complexity is polynomial in the number of inputs. In contrast, an algorithm is considered as ***pseudo-polynomial time*** (PPT) if its worst-case time complexity is polynomial in the numeric value of input (e.g. counting frequencies of all elements in an array). If problem Π can be solved by a pseudo-polynomial time algorithm A, then it is called ***weakly NP-complete*** (solving the knapsack problem using dynamic programming is a good example); otherwise, it is called ***strongly NP-complete***, unless complexity class P = complexity class NP.

Algorithm A is said to be ***polynomial-time approximation scheme*** (PTAS) if for any given instance I of problem Π, approximation factor $\lambda = (1+\varepsilon)$ for minimization and $\lambda = (1 - \varepsilon)$ for maximization. ε is a parameter that denotes the upper or lower bound of the quality of $A(\Pi, I)$ relative to $OPT(\Pi, I)$. Thus, the difference between $COST(OPT(\Pi, I))$ and $COST(A(\Pi, I))$ must not exceed ε for all possible instances of problem Π. By assigning a value to ε, algorithm A must run in polynomial time $poly(|I|)$.

Likewise, algorithm A is said to be ***PTAS with absolute performance guarantee*** if for any given instance I of problem Π, $[COST(A(\Pi, I)) - COST(OPT(\Pi, I))] \leq c$ for minimization and $[COST(OPT(\Pi, I)) - COST(A(\Pi, I))] \leq c$ for maximization. This means that $A(\Pi, I)$ is at most c worse than $OPT(\Pi, I)$, in terms of objective function value (the cost function).

Finally, although a PTAS algorithm runs in polynomial time in accordance with n (size of a given instance I of problem Π), the time complexity may grow exponentially with respect to ε. For instance, approximation algorithms running in $O(n^{\frac{1}{\varepsilon}})$ or $O(2^{\frac{1}{\varepsilon}})$ are still PTASs. To address this issue, ***fully polynomial time approximation scheme*** (FPTAS) is used to study a class of PTAS algorithms that run in polynomial time according to both n and $\frac{1}{\varepsilon}$ $poly(n, \frac{1}{\varepsilon})$, e.g. $O(\frac{n^{\alpha}}{\varepsilon^{\beta}})$ with $\alpha, \beta \geq 1$. Intuitively, this is a guarantee that an increase of problem size n or an increase of approximation quality $\frac{1}{\varepsilon}$ does not affect the runtime more than polynomially.

2.5. Classification of approximation algorithms

Depending on approximation factor equation, approximation algorithms can be classified into (see Figure 6.1):

- **PPTA** = optimization problems admitting a pseudo-polynomial time algorithm.

- **APX** = optimization problems admitting an approximation algorithm running in polynomial time with a constant performance ratio.
- **PTA** = optimization problems admitting a PTAS
- **FPTA** = optimization problems admitting an FPTAS

Unless complexity class P = complexity class NP, it holds that FPTA ⊆ PTA ⊆ APX.

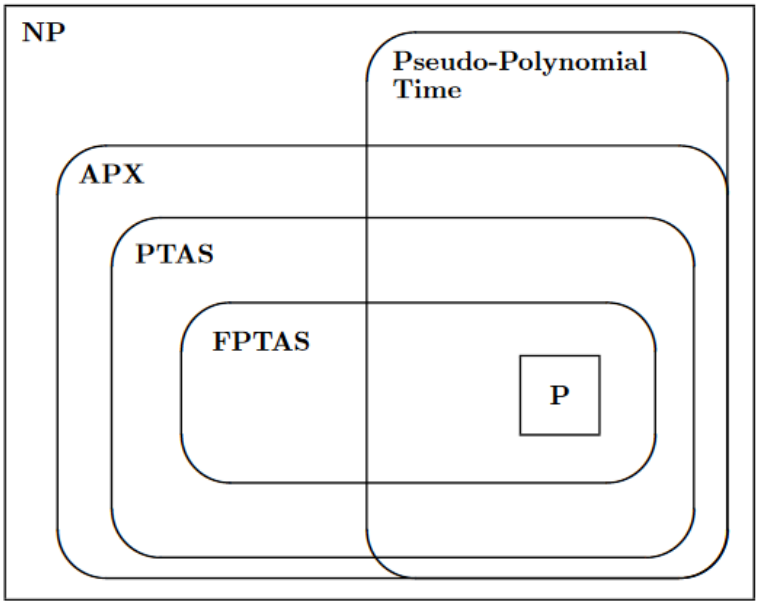


Figure 6.1. Classification of approximation algorithms.

2.6. Hardness of approximation

Unless complexity class P = complexity class NP, many computational problems are not only difficult to solve but also difficult to approximate for any given approximation factor. They are said to be *non-approximable*; the traveling salesman problem is a good example. Consider a problem Π and an instance I with the goal of proving that problem Π is $\alpha(|I|)$ hard to approximate ($|I|$ represents the size of instance I). Hence, we use two well-known methods: *gap-introducing reduction* and *gap-preserving reduction*.

- **Gap-introducing reduction**

It aims at reducing an NP-complete decision problem Π' to problem Π. Let Π' be a decision problem and Π be a minimization problem (similar for maximization). A reduction h from problem Π' to problem Π is said to be *gap-introducing* if:

1. It transforms each instance I' of problem Π' to an instance $I = h(I')$ of problem Π, in polynomial time.
2. There exist functions f and α such that:

   If instance I' is a "yes instance" of problem Π' then $OPT(\Pi, I) \leq f(I)$

If instance I' is a "no instance" of problem Π' then $OPT(\Pi, I) > \alpha(|I|) \times f(I)$

Theorem (proof not included). If problem Π' is NP-complete then problem Π cannot be approximated with a factor α.

- **Gap-preserving reduction**

It reduces a problem Π' that is hard to approximate to problem Π. Let Π' and Π be minimization problems (similar for maximization). A reduction h from problem Π' to problem Π is said to be *gap-preserving* if:

1. It transforms each instance I' of problem Π' to an instance $I = h(I')$ of problem Π, in polynomial time.
2. There exist functions f, f', α, β such that:

   $OPT(\Pi', I') \leq f'(I') \Rightarrow OPT(\Pi, I) \leq f(I)$

   $OPT(\Pi', I') > \beta(|I'|) \times f'(I') \Rightarrow OPT(\Pi, I) > \alpha(|I|) \times f(I)$

Theorem (proof not included). If problem Π' is non-approximable with a factor β then problem Π cannot be approximated with a factor α unless P = NP.

Generally speaking, proving the hardness of approximation results requires considerable knowledge of the characteristics of problems. Even so, there are a few broad methods used, among which we cite:

1. Use already existing hardness results and gap reductions to get hardness results for new problems.
2. Amplification: some problems exhibit self-reducibility even in the sense of approximation. This allows using a given hardness factor to something larger.
3. Use probabilistically checkable proofs (PCPs) and other sophisticated tools such as the parallel repetition theorem which establish gap reduction for some basic CSPs (constrain satisfaction problems).

3. Classic examples

In what follows, we show some examples of approximation algorithms application to solve typical problems while measuring the quality of the solutions retained.

3.1. Graph-coloring problem

The graph-coloring problem consists in determining the smallest number of colors needed to color the vertices of a graph, such that the vertices of each edge cannot have the same color. A coloring of the vertices of a given graph $G = (V, E)$ can be seen as a function $c : V \rightarrow \mathbb{N}$, such that $c(u) \neq c(v)$ if u and v are connected by an edge in G. The pseudo-code below determines a coloring which is not necessarily minimal.

For each $u \in V$ **do**

Color vertex u with the smallest color not appearing on an adjacent vertices ;

End for

Let us apply this algorithm to graph $G = (V, E)$ where $V = \{1, 2, \ldots, 2n\}$ such that each odd vertex i is connected to all even vertices j except i+1. Figure 6.2 illustrates a typical graph with $n = 4$.

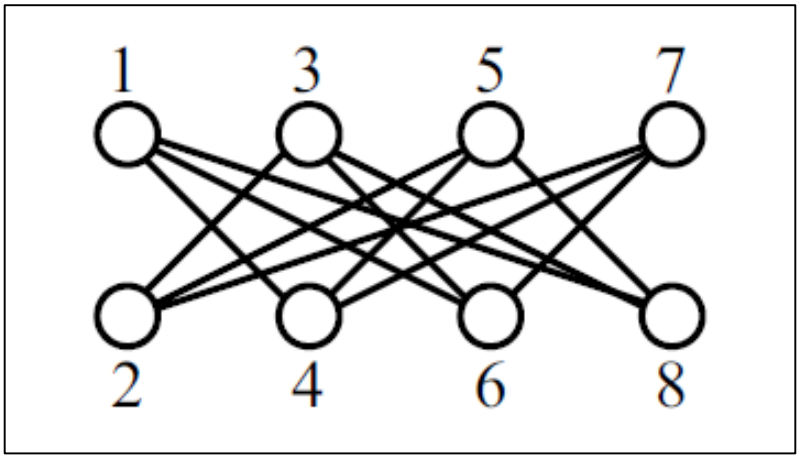


Figure 6.2. A typical a graph with $2n$ nodes ($n = 4$).

The adopted algorithm uses n colors whereas two colors are enough. If we denote by $OPT(G)$ the minimum number of colors necessary and by $A(G)$ the number of colors used by the above algorithm, then: $\lambda = \frac{A(G)}{\mathrm{OPT}(G)} = \frac{n}{2}$

3.2. Traveling salesman problem

3.2.1. Approximation algorithm for the determination of Hamilton cycles

In this version of the traveling salesman problem, we are interested in determining a Hamilton cycle (i.e. passing exactly once through each vertex) of minimum length in a complete graph G. Suppose that the distance matrix satisfies the triangular inequality, i.e. $distance(x, z) \leq distance(x, y) + distance(y, z)$ for any triplet of vertices x, y, z. A non-optimal solution can be determined using the algorithm illustrated below (see Figure 6.3 that shows a typical example of a complete graph with five vertices).

1. Determine a minimum cost tree in G.
2. Determine a cycle that passes exactly twice through each edge of the tree.
3. Shorten this cycle if necessary to pass through each vertex only once.

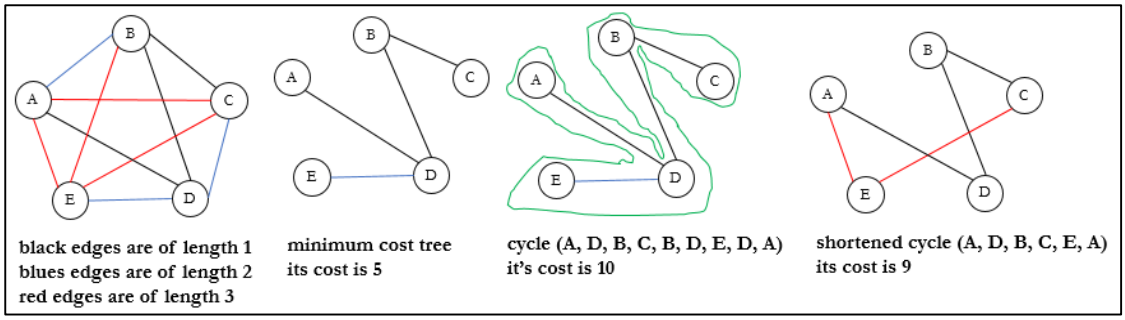


Figure 6.3. Steps for the determination of Hamilton cycle of minimum length.

We denote by $OPT\ (G)$ the length of the smallest Hamilton cycle in G and by $A\ (G)$ the length of the cycle produced by the above algorithm. It is proven that, as the minimum cost tree has a cost lower than or equal to $OPT\ (G)$: $\lambda = \frac{A\ (G)}{OPT\ (G)} \leq 2$

3.2.2. Non-approximation results in the traveling salesman problem

Let $G\ (V, E)$ be a complete graph and w: $E \rightarrow \mathbb{N}$ be a weight function. Thus, it is asked to find a tour that has minimum total weight which is in fact an NP-hard problem.

Theorem. According to this definition, the traveling salesman problem does not admit an approximation algorithm of any approximation factor α in polynomial time in n (number of vertices), unless P = NP.

To prove the approximation hardness, we rely on a gap-introducing reduction from Hamilton cycle (a cycle using each vertex only ones) due to the fact that the determination of whether a given graph G' has a Hamilton cycle or not is NP-complete. The idea is to use the proof by absurd starting from the assumption that there is an α-approximation algorithm A, as shown in Figure 6.4.

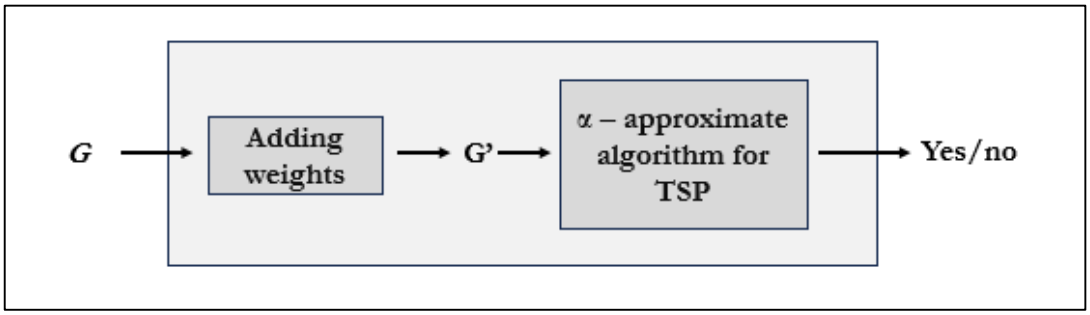


Figure 6.4. Steps for proving the non-approximation of the traveling salesman problem.

Let's build a polynomial algorithm A deciding Hamilton cycles.

Adding weights: based on graph $G\ (V, E)$, we define $G'\ (V, E)$ as a complete graph such that $w\ (e) = 1$ if $e \in E$, otherwise $w\ (e) = 1 + \alpha \times n$ ($n = |V|$) with $\alpha \geq 0$.

- If graph G' admits an optimal tour of cost n (i.e. $OPT\ (G') = n$) then graph G admits a Hamilton cycle.

– If graph G' admits a tour of cost $\leq \alpha \times n$, then graph G admits a tour of cost n. Likewise, if graph G' admits a tour of cost $> \alpha \times n$, then graph G does not admit a tour of cost n.

Clearly, unless P = NP, algorithm A cannot be a polynomial algorithm, and therefore cannot be an α-approximation.

3.3. Vertex cover problem

This problem consists in modeling and solving a system for which it is asked to place a minimum number of guards to monitor all the corridors. To this end, one generally involve graph theory where a cover of vertices must be calculated.

Let $G = (V, E)$ be a graph. A vertex cover is a subset $C \subseteq V$ such that for each $(u, v) \in E$, $u \in C$ or $v \in C$ (see Figure 6.5).

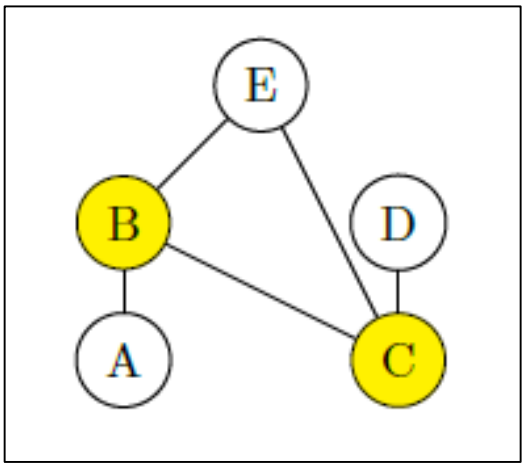


Figure 6.5. A typical graph illustrating a vertex cover.

Solving this problem can be done through the naive algorithm below; Figure 6.6 gives the result of its execution on a typical graph.

Naïve algorithm

$C \leftarrow \emptyset$;

$E' \leftarrow E$;

While $(E' \neq \emptyset)$ **do**

$e \leftarrow$ the first element of set $V \setminus C$;

$E' \leftarrow E' \cup \{\text{edges } a = (u, v) \ / \ u = e \text{ or } v = e \text{ with } a \notin E'\}$;

$C \leftarrow C \cup \{e\}$;

End while

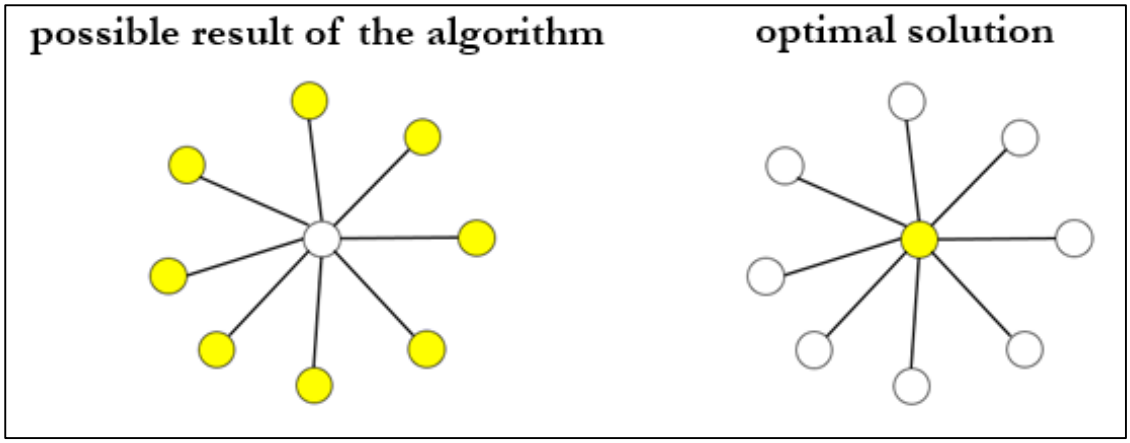


Figure 6.6. Execution result of the naïve algorithm on a typical graph.

Now, we proceed to an approximation algorithm, in order to solve this problem. This algorithm is based on the concept of coupling in graph theory. Thus, we first introduce some definitions that would help to better understand the adopted approximation algorithm. Let $G = (V, E)$ be a graph.

- A set $M \subseteq E$ is a matching if for any vertex $u \in V$, there exists at most one edge $(a, b) \in M$ with $u = a$ or $u = b$. One can easily show that $|M| \leq |C|$, for any matching M and any vertex cover C.
- A matching M is maximum if for each matching M' we have $|M'| \leq |M|$.
- A maximal matching for inclusion is a matching M such that for each $M' \subseteq E$, $M \subseteq M'$ implies that M' is not a matching. It is demonstrated that a maximum matching for cardinal is maximum for inclusion.

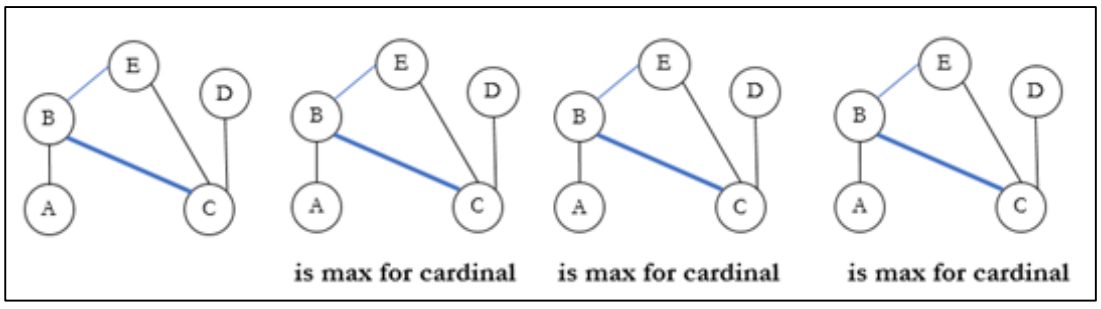


Figure 6.7. Typical examples of maximal matching for inclusion.

The approximation algorithm is described by the following pseudo-code:

Approximation algorithm for vertex cover

$M \leftarrow$ a maximal matching for inclusion ;

$C \leftarrow$ {vertices of the edges of M} ;

Return C ;

It is proven that the result C returned by the approximation algorithm is a vertex cover. It is also proven that this algorithm admits an approximation of factor $\lambda = 2$, as illustrated in the graphs illustrated in Figure 6.8.

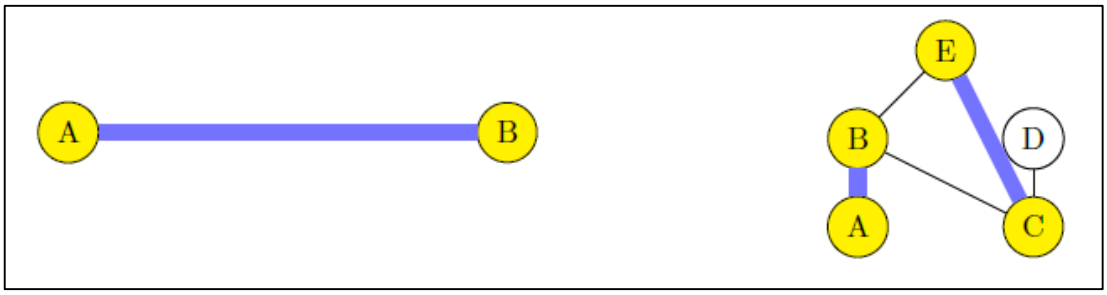


Figure 6.8. Typical examples of vertex cover.

3.4. Bin packing problem

Let $L = \{l_1, l_2, \ldots, l_n\}$ be a list of n items of sizes $[s_1, s_2, \ldots, s_n]$ respectively in such a way that $0 < s_{i=1..n} \leq 1$. We also assume an infinite supply of bins such that the size of each unit is 1. The aim is then to pack the items of list L using as few bins as possible. This is an NP-complete optimization problem. There are two versions to this problem depending on how the elements arrive and are placed in bins.

- **Online version:** this version assumes that the items arrive sequentially one by one in an unknown order. Thus, each item must be put in a bin before considering the next item.
- **Offline version:** all items of list L are given in advance.

The online version of this problem would be more difficult since it does not always lead to the optimal solution. For instance, let consider a list L of $n = 2 \times m$ items: m small items of size $0.5 - \varepsilon$ and m large items of size $0.5 + \varepsilon$. The optimal solution consists in packing the items as pairs (small item, large item), thus leading to use m bins. However, in the online version of the problem, there is no knowledge about the future items as well as the way in which they will arrive; this makes it difficult to build an optimal solution. Indeed, with a bit of luck, the small items may arrive before the larger ones. In this case, we will need about of $3/2\ m$ bins to pack all items according to their arrival (the small items are packed before the large ones). In what follows, we present simple approximation algorithms for both online and offline versions of the problem. To measure the performance of a given approximation algorithm A, we denote by $A(L)$ and $OPT(L)$ the number of bins used when algorithm A is applied to list L and the optimal number for list L, respectively.

3.4.1. Approximation algorithms for the online version of the problem

We consider three well-known online algorithms in the literature that use at most twice the optimal number of bins. These algorithms use the following general scheme.

```
While (not end of items arrival) do
    //carry out a capacity check of the bins already open
    If (the new item can be packed in one of the bins already open) then
        1. put it in one of these bins;
    else
        2. open a new bin to pack the new item;
    End if
End while
```

The considered algorithms differ in the criterion used to choose the open bin for packing the new item in step 1, as follows:

- **Next-fit strategy (NF):** the NF strategy always keeps the last used bin open to check whether it is suitable for packing the new item or not. If so, the new item is packed, which only results in a decrease in the remaining capacity of the bin. Otherwise, a new bin is opened while the last one is closed. Note that NF can be generalized to w-NF by keeping the w last bins open (sliding window of size w through the bins used). NF is simple as it requires a linear time $O(n)$ and a bounded space $O(1)$ (it only keeps a single open bin in memory). Moreover, it is proven that NF admits an approximation of factor $\lambda = \frac{NF(L)}{\mathrm{OPT}(L)} \leq 2$ (i.e. 2-approximate). Indeed, if k is the optimal number of bins, then NF never uses more than $2 \times k$ bins. In particular, there exist sequences that force NF to use $2 \times k - 2$ bins.

  Example: let $L = \{l_1, l_2, l_3, l_4, l_5, l_6, l_7\}$ be a list of 7 items whose sizes are respectively [0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8]. The NF algorithm uses 5 bins to pack the items as follows: $bin_1(l_1, l_2)$, $bin_2(l_3)$, $bin_3(l_4, l_5)$, $bin_4(l_6)$ and $bin_5(l_7)$.

- **First-fit strategy (FF):** this strategy can be seen as an improvement of the NF algorithm by keeping all used bins open in the order in which they were opened, until the packing task is finished. Thus, FF scans the bins in an attempt to pack the new item in the first bin that fits. Otherwise, a new bin is opened in order to pack the new item. The FF requires $O(n^2)$ time which is reduced to $O(n\ Log\ n)$ using proposer data structures (self-balancing binary search trees) and $O(n)$ for space complexity (it keeps all bins in memory). It is proven that FF admits an

approximation of factor $\lambda = \frac{FF(L)}{\text{OPT (L)}} \leq 1.7$. Indeed, if k is the optimal number of bins, then FF never uses more than $1.7 \times k$ bins.

Example: let $L = \{l_1, l_2, l_3, l_4, l_5, l_6, l_7\}$ be a list of 7 items whose sizes are respectively [0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8]. The FF algorithm uses 4 bins to pack the items as follows: $bin_1(l_1, l_2, l_5)$, $bin_2(l_3, l_6)$, $bin_3(l_4)$ and $bin_4(l_7)$.

- **Best-fit strategy (BF):** this strategy also keeps all used bins open until the packing task is finished. The BF scans the bins in an attempt to pack the new item in the tightest spot (i.e. the bin with a remaining capacity close as much as possible to the item size). Otherwise, a new bin is opened in order to pack the new item. Similarly to FF algorithm, the BF requires $O(n^2)$ time that can be reduced to $O(n\ Log\ n)$ using self-balancing binary search trees and $O(n)$ for space complexity (it keeps all bins in memory). It is proven that BF admits an approximation of factor $\lambda = \frac{BF(L)}{\text{OPT (L)}} \leq 1.7$. If k is the optimal number of bins, then BF never uses more than $1.7 \times k$ bins.

  Example: let $L = \{l_1, l_2, l_3, l_4, l_5, l_6, l_7\}$ be a list of 7 items whose sizes are respectively [0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8]. The BF algorithm uses 4 bins to pack the items as follows: $bin_1(l_1, l_2, l_5)$, $bin_2(l_3)$, $bin_3(l_4, l_6)$ and $bin_4(l_7)$.

3.4.2. Approximation algorithms for the offline version of the problem

By considering that all items of list L are known in advance, it is expected to do better. Exhaustive enumeration (e.g. through backtracking algorithms) is the best way to find the optimal solution. However, there is no efficient algorithm that solves this problem in a polynomial time due to NP-completeness. One way to overcome some of the difficulties caused by online algorithms is to sort the input sequence of items before packing them in bins, as given in the pseudo-code below.

1. Sort list L according to a descending order of items size.
2. Apply an online algorithm on the sorted list L.

Next, we present the algorithms discussed in the previous section (i.e. NF, FF and BF) in the context of the offline version of bin packing problem, as follows:

- **Next-fit-decreasing (NFD):** the NFD first sorts list L according to a descending order of items sizes and then call the NF algorithm. It is proven that NFD admits an approximation of factor λ slightly less than 1.7 in the worst case.
- **First-fit-decreasing (FFD):** the FFD first sorts list L according to a descending

order of items sizes and then call the FF algorithm. It is proven that FFD admits an approximation of factor $\text{FFD}(L) = \frac{\text{OPT}(L) \times 11 + 6}{9}$.

- **Best-fit-decreasing (BFD):** the BFD first sorts list L according to a descending order of items sizes and then call the BF algorithm. It is proven that BFD admits an approximation of factor $\text{FFD}(L) \leq \frac{\text{OPT}(L) \times 6}{5} + 1$.

3.5. 0/1 Knapsack problem

Consider a knapsack that can hold up to a maximum weight $W \in \mathbb{N}$ and a set of n objects $X = \{x_1, ..., x_n\}$, each associated with a $weight(x_i) \in \mathbb{N}$ and a $value(x_i) \in \mathbb{N}$. For each subset $S \subseteq X$, we define $weight(S) = \sum_{x \in S} weight(x)$ and $value(S) = \sum_{x \in S} value(x)$. The goal is to find a subset $S \subseteq X$ that maximizes objective function $value(S)$ without exceeding the knapsack capacity W (i.e. $weight(S) \leq W$). It should be noted that the greedy algorithm shown in Section 4.3 in chapter 1 represents a 2-approximation algorithm. Thus, in the following, we present only the (1-ε)-approximation algorithm.

The knapsack problem admits a PPT algorithm based on dynamic programming with time complexity $O(n \times V^*)$; n is the number of objects in set X while V^* is their maximum value (i.e. $value(X)$). Note that $n \times V_{max}$ is a bound on the optimal solution; such that $V_{max} = \max_{i = 1..n} value(x_i)$. As a result, the time complexity is $O(n^2 \times V_{max})$.

We define matrix $M[1..n+1, 1..V^*+1]$; for each $0 \leq i \leq n$ and $0 \leq v \leq V^*$, the corresponding recursive formula is given as follows:

$$M[i, v] = \begin{cases} +\infty & \text{if } i = 0 \text{ or } v = 0 \\ M[i\text{-}1, v] & \text{if } v < value(x_i) \\ min(M[i\text{-}1, v], weight(x_i) + M[i\text{-}1, v\text{-}value(x_i)]) & \text{otherwise} \end{cases}$$

An entry $M[i, v]$ is the minimum weight of objects from subset $\{x_1, ..., x_v\}$ such that the value is exactly v. In the case where there is no such subset, $M[i, v] = +\infty$. The optimal solution $OPT(M) = \max_{0 \leq v \leq V^*} \{p / M[n, v] \leq W\}$.

The recursive formula shows that each entry $M[i, v]$ only depends on previous entries (i.e. it is calculated in bottom-up) which means that $OPT(M)$ is determined afterwards.

However, V^* can be arbitrarily large which means that the runtime is polynomial as long as V^* is polynomial in n. Therefore, we need for an FPTAS algorithm that runs in polynomial time in n while being independent of V^*. To do so, we rely on the adopted PPT that serves

as a subroutine for the approximation algorithm. The idea consists in downscaling the profits to polynomial size as required by the error parameter ε, as given in the pseudo-code below. It is proven that the time complexity of the FPTAS is $O\left(\frac{n^3}{\varepsilon}\right)$.

Function approximation-knapsack (ε: real): subset of set X
Begin
$k \leftarrow \varepsilon \times V_{max} / n$;
Redefine function *value* so that *value* $(x_{i=1..n})$ becomes $\lfloor value\ (x_{i=1..n}) / k \rfloor$;
Use dynamic programming to get solution S' for the new instance of the problem;
Return S';
End

4. Advantages and drawbacks of approximation algorithms

Approximation algorithms have some advantages, including:

- **Problem analysis:** the design of approximation algorithms requires analyzing problems which makes it possible: to highlight the variations in difficulty between them, to distinguish the easy cases of problems from those difficult and to help design effective and practical heuristics.
- **Time complexity:** approximation algorithms run in polynomial time, which significantly reduces the time complexity compared to the corresponding exact algorithms that generally run in exponential time.
- **Guarantee:** although approximation algorithms do not allow necessarily finding the optimal solutions, the approximation factor λ is nevertheless a proven guarantee since the solution is at most λ times worse than the optimal solution.

Approximation algorithms also have some drawbacks; we cite as examples:

- **Difficulty of approximation:** the determination of a good approximation is often carried out by making several proofs, which requires a strong mathematical background and formal demonstrations.
- **Complexity estimation:** unlike other algorithmic paradigms (e.g. integer programming), there is often no incremental/continuous tradeoff between running time and solution quality.
- **Approximation quality:** for some problems, the approximation factor still remains large while for others, it is impossible to admit an approximation scheme.

Exercises set 1 (Greedy algorithms)

Exercise 1: shortest superstring problem

Let $S = \{s_1, s_2, \ldots, s_n\}$ be a set of n strings such that no element s_i is a substring of another s_j. It is asked to devise a greedy algorithm that attempts to find the shortest superstring containing each element $s_i \in S$ as a substring.

Example

Let consider S = {"TCADOG", "GTAAGT", "DOGTA", "TTCA", "AGTCTTC"}. Thus, str_1 = "TCADOGTAAGTCTTCA" and str_2 = "TTCADOGTAGTAAGTCTTC" are two potential solutions. However, str_1 is better as it is shorter than str_2.

Exercise 2: job-sequencing problem with deadlines

Let consider a uniprocessor machine and a set of n tasks $X = \{t_1, t_2, \ldots, t_n\}$ with deadlines $d_1, d_2, \ldots, d_n$. Each task t_i takes one unit and it cannot run beyond its deadline (i.e. it must end before deadline d_i). When a given task ends before its deadline, it earns a profit p_i. The objective is to try to find a task scheduling that maximizes the sum of profits.

Example

Let consider a system composed of 10 tasks, as follows:

Tasks (t_i)	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
Deadlines (d_i)	9	2	5	7	4	2	5	7	4	3
Profits (p_i)	15	2	18	1	25	20	8	10	12	5

A possible jobs schedule is as follows: t_7, t_6, t_9, t_5, t_3, t_4, t_8, t_1; the total profit earned is 109.

Exercise 3: activity selection problem

Let $A = \{a_1, a_2, \ldots, a_n\}$ be a set of n activities such that each activity is characterized by its starting and finishing time. The objective is to try to find the maximum number of activities performed by a single person assuming that a person can only work on a single activity at a time.

Example

Let consider a set of 11 activities such that the following pairs give their starting and finishing time: a_1 (1, 4), a_2 (3, 5), a_3 (0, 6), a_4 (5, 7), a_5 (3, 8), a_6 (5, 9), a_7 (6, 10), a_8 (8, 11), a_9 (8, 12), a_{10} (2, 13), a_{11} (12, 14).

A potential solution is to select: a_1 (1, 4), a_4 (5, 7), a_8 (8, 11), a_{11} (12, 14).

Exercise 4: bookshelves problem

A library is planning to redo its bookshelves. It includes a collection of books $b_{i = 1..n}$. Each book b_i has a width w_i and a height h_i. The books should be stored on shelves of length L. Let assume that the books should be arranged in an unspecified order by considering only their heights and widths. The objective is to write two greedy algorithms that return the number of required bookshelves for the arrangement of books in order to:

1. minimize the number of shelves necessary while caring only about books widths.
2. minimize the books clutter which is defined as the sum of the heights of the largest book in each shelf used (without worrying about books widths).

Example

Let consider a set of 8 books to be arranged in bookshelves of length $L = 10$. The following pairs give their heights and widths, respectively: b_1 (25, 3), b_2 (26, 2), b_3 (27, 3), b_4 (30, 2), b_5 (30, 1), b_6 (25, 4), b_7 (29, 3), b_8 (24, 1). The order of books that minimizes the number of required bookshelves would be: b_6, b_1, b_3, b_7, b_2, b_4, b_5, b_8 (2 bookshelves) while the order that minimizes the books clutter would be: b_4, b_5, b_7, b_3, b_8, b_2, b_1, b_6 (2 bookshelves and a books clutter of 56).

Exercise 5: train platforms problem

Consider a station for which the arrival and departure of trains are scheduled. Thus, it is asked to write a greedy algorithm that attempts to determine the minimum number of platforms so that there will be no delays in trains' arrivals.

Example: let consider trains arrival = {2:00, 2:10, 3:00, 3:20, 3:50} and departure = {2:30, 3:40, 3:20, 4:30, 4:00}. The minimum number of platforms needed is 2 (see the table below)

Event	**Time**	**Platform id**
Arrival	2:00	1
Arrival	2:10	2
Departure	2:30	1
Arrival	3:00	1
Departure	3:20	1
Arrival	3:20	1
Departure	3:40	2
Arrival	3:50	2
Departure	4:00	2
Departure	4:30	1

Exercises set 2 (Divide-and-conquer)

Exercise 1: minimum-maximum in an array

Let consider an array A of n integers. Using the divide-and-conquer paradigm, write:

1. an algorithm to find the index of the largest element in A. Calculate its time complexity.
2. an algorithm that performs the simultaneous search for the smallest and largest elements in array A (it returns their indices). Calculate its time complexity.

Exercise 2: modified binary search in a sorted array

Let T [1..n] be an array of distinct integers sorted in ascending order, some of which may be negative. Using the divide-and-conquer paradigm, write an algorithm that returns an index i such that $T[i] = i$, assuming that such an index exists. Calculate its time complexity.

Exercise 3: median in two sorted arrays

Let A [1..n] and B [1..n] be two arrays sorted in ascending order. We seek to find the median element of these two arrays (element which has as many strict greater elements as lower or equal elements), using the divide-and-conquer paradigm. Calculate its time complexity.

Exercise 4: inversions in an array

Consider an array of n positive integers $A = [a_1, a_2, \ldots, a_n]$. We say that pair (i, j) is an inversion of A if $(i < j)$ and $(a_i > a_j)$. For example, array $A = [2, 6, 3, 1, 5]$ has 5 inversions: (1, 4), (2, 3), (2, 4), (2, 5) and (3, 4).

1. Write a naive algorithm to determine the number of inversions of array A. Calculate its time complexity.
2. By relying on the divide-and-conquer paradigm, write an algorithm to determine the number of inversions of array A. Calculate its time complexity.

Exercise 5: integer power

I) Let A and n be two integers such that $n \geq 0$. We want to calculate the value of power A^n.

1. Write a simple recursive function to calculate A^n. Calculate its time complexity.
2. We define the value of power A^n as follows:

$$A^n = \begin{cases} 1 \text{ if } n = 0 \\ A^n = A^{n/2} \times A^{n/2} & \text{when } n \text{ is even } (n = 2 \times k) \\ A^{2k+1} = A \times A^{n/2} \times A^{n/2} & \text{when } n \text{ is odd } (n = 2 \times k + 1) \end{cases}$$

Use *the divide-and-conquer* paradigm to write a recursive function that performs the requested calculation. Calculate its time complexity.

3. Now, the value of power A^n is defined as follows:

$$A^n = \begin{cases} 1 \text{ if } n = 0 \\ A^{2k} = (A^2)^k & \text{when } n \text{ is even } (n = 2\times k) \\ A^{2k+1} = A\times(A^2)^k & \text{when } n \text{ is odd } (n = 2\times k + 1) \end{cases}$$

Use *the divide-and-conquer* paradigm to write a recursive function that performs the requested calculation. Calculate its time complexity.

II) Let F be a function that depends on an integer $n \geq 0$ and an integer constant $v_0 > 0$, defined as follows:

```
Function F (n : integer) : integer
Begin
        If (n = 0) then
                Return (v0) ;
        Else
                Return F (n - 1) × F (n - 1) × …. × F (n - 1) ;
                                  └──────────────────────┘
                                          v0 times
        End if
End
```

1. Give the result of execution of function F for:

   $n = 3$ and $v_0 = 2$

   $n = 3$ and $v_0 = 3$

2. What does function F calculate? A demonstration is requested.

3. Determine the number of multiplications $m(n)$ performed by function $F(n)$.

4. Calculate the time complexity of function $F(n)$.

5. Show how it would be possible to improve the time complexity. Give the time complexity of the proposed solution.

Exercises set 3 (Dynamic programming)

Remark.

For all the exercises in this set, the aim is to solve problems using dynamic programming. Thus, it is asked to give:

1. The recursive formula leading to efficient problem solving.
2. The structure of the array of results as well as the initial values of its cells.
3. The pseudo-code to fill in the array of results.

Exercise 1: Fibonacci sequence

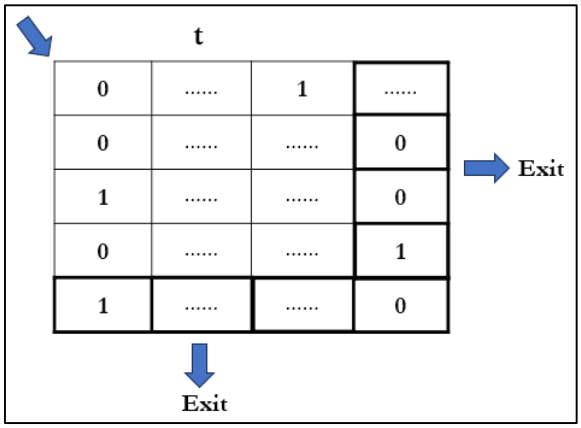
The Fibonacci sequence is defined by the following recursive formula:

$$F_n = \begin{cases} F_0 = 1 \\ F_1 = 1 \\ F_{n-1} + F_{n-2} \quad \text{when } n > 1 \end{cases}$$

The aim is to calculate the n^{th} term of the Fibonacci sequence.

Exercise 2: maze problem

A maze is modeled by a matrix t of size $n \times p$ containing only 0s and 1s, as shown in the figure below. The 0s represent empty spaces while the 1s represent inaccessible cells. Moves can be *down*, *right* and *across the diagonal*. The objective is to find the shortest path to get out.



Exercise 3: edit distance calculation (Levenshtein distance)

The edit distance aims to measure the similarity between two strings. A classic use-case of edit distance is when a search engine returns the same result for two entered keywords (e.g. "***dynamic***" and "***dymanic***"). Three elementary operations on words are considered:

- **substitution:** one letter is replaced by another,
- **insertion:** a new letter is added,
- **deletion:** a letter is deleted.

The edit distance between two words U and V is then the minimal number of operations to transform U into V. For instance, on the word ***decay***, if we substitute ***y*** for ***d*** and insert ***e*** after ***d***, we get the word ***decade***. We can demonstrate that this number of operations is minimal and that the distance from ***decay*** to ***decade*** is therefore 2: one substitution and one insertion. Thus, it is asked to provide a solution to this problem.

Exercise 4: largest independent set in a graph

Let $G = (V, E)$ be a directed graph where $V = \{v_1, v_2, \ldots, v_n\}$ is a set of n vertices (nodes) and E is the set of edges between the vertices such that each edge has a non-negative length. A subset $X \subseteq V$ is said to be *independent* if, for any pair of vertices $x, y \in X$, $(x, y) \notin E$ (i.e. no pair of vertices in X share an edge). Therefore, we are interested in finding an independent set of maximum size using dynamic programming.

Exercise 5: longest palindromic subsequence

Let $S = c_1c_2 \ldots c_n$ be a sequence of n characters. It is asked to find the length of the longest palindromic subsequence in sequence S.

Example

Let S_1 = WEEKSFORWEEKS and S_2 = MMAMCMCAM be two characters sequences.

The longest palindromic subsequence of S_1 is 5. There are more than palindromic subsequences of length 5 (e.g. EEWEE, EEREE, etc.).

Regarding sequence S_2, the longest palindromic subsequence is 7 corresponding to MAMCMAM. Subsequences MMMMM and MMCMM in sequence S_2 are also palindromic but not the longest ones.

Exercises set 4 (Backtracking)

Remark.

For all the exercises in this set, the aim is to solve CSPs using backtracking algorithms according to the basic scheme. Thus, it is asked to give:

1. The problem formulation by specifying the decision variables, their domains of definition and the constraints imposed on them.
2. The pseudo-code describing the steps of the backtracking algorithm.

Exercise 1: all possible strings made by placing spaces

Let S be a string for which it is asked to generate all possible strings made by placing zero or one space between its characters.

Example

Consider S = "xyz", the set of all possible strings is: {"xyz", "xy z", "x yz", "x y z"}.

Exercise 2: tug of war

Let S be a set of n integers. We seek to divide set S into two subsets of size $n/2$ each so that the absolute value of the difference of the elements sum of each subset is as minimum as possible. Note that in the case where n is odd, one subset is of size $(n - 1)/2$ while the other subset is of size $(n + 1)/2$.

Example

Consider S = {3, 4, 5, -3, 100, 1, 89, 54, 23, 20} where n = 10. Note that subsets S_1 = {4, 100, 1, 23, 20} and S_2 = {3, 5, -3, 89, 54} represent an optimal solution as they are both of size 5 and the sum of their elements is the same (148 in both cases).

Exercise 3: Maximum number possible using at most *K* swaps

Let M and K two positive integers. We seek to find the maximum integer possible by using at most K swap operations on its digits.

Example

- For M = 6945 and K = 1, the output is 9645 (swap 9 with 6).
- For M = 5488 and K = 2, the output is 8854 (swap 8 with 4 so number becomes 5884, then swap 5 with 8 so number becomes 8854).
- For M = 96541 and K = 1, the output is 96541 (no swap is required).

Exercise 4: knight tour problem

Let consider a knight that is placed on the first block of an empty board. The knight moves according to chess rules so that it must visit each square exactly once. Thus, we seek to find all solutions such that for each solution, we keep track of all movements through which the knight goes.

Example

The table below shows a typical path that a knight follows to cover all cells of a chessboard of 8 × 8. Numbers in the cells indicate the order by which they were visited by the knight.

0	59	38	33	30	17	8	63
37	34	31	60	9	62	29	16
58	1	36	39	32	27	18	7
35	48	41	26	61	10	15	28
42	57	2	49	40	23	6	19
47	50	45	54	25	20	11	14
56	43	52	3	22	13	24	5
51	46	55	44	53	4	21	12

Exercise 5: Sudoku

Let consider a 2D array of size 9×9 which is partially filled. The goal is to assign digits (from 1 to 9) to the empty cells so that each row, column, and sub-grid of size 3×3 contains exactly one instance of the digits from 1 to 9.

Example

	2		7			1	9	
4			1	9	5			
	9			6	4	7		
	5			7		6		
2		6				5		8
		9		5			4	
		8	3	4			6	
			5	2	9			1
	3	2			1		5	

Initial state

6	2	5	7	3	8	1	9	4
4	8	7	1	9	5	3	2	6
3	9	1	2	6	4	7	8	5
8	5	3	4	7	2	6	1	9
2	4	6	9	1	3	5	7	8
1	7	9	8	5	6	2	4	3
5	1	8	3	4	7	9	6	2
7	6	4	5	2	9	8	3	1
9	3	2	6	8	1	4	5	7

Final state (correct solution)

Exercises set 5 (Probabilistic methods)

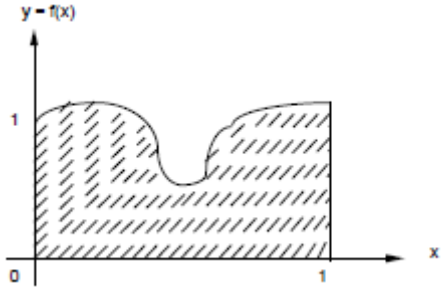
Exercise 1: fair and biased random values

I) Let ***foo()*** be an algorithmic function that represents a biased coin such that when called, it returns 0 or 1 with 60% and 40% probabilities, respectively. Based only on function foo(),write a new function ***new-foo-50()*** that returns 0 or 1 with 50% probability each.

II) Now, let assume that ***foo()*** represents a fair coin (i.e. when called, it returns 0 or 1 with equal probability of 50% each). Based only on function foo(), write a new function ***new-foo-25-75()*** that returns 0 or 1 with 25% and 75% probabilities, respectively.

Exercise 2: numerical integration

Let f: [0, 1] → [0, 1] be a continuous function. Therefore, we want to estimate the value of $\int_0^1 f(x)\,dx$ which is actually a measure of the area under the curve $y = f(x)$, as shown in the figure below. Write a pseudo-code of a randomized algorithm that returns the estimated value of this numerical integration.



Exercise 3: randomness on array elements and indices

I) Let $A = [a_1, a_2, \ldots, a_n]$ be an array of n elements. It is asked to write a pseudo-code of an algorithm that generates a random permutation of the elements of array A (i.e. randomize array A). This problem is also known as ***"shuffle a deck of cards"***. In this context, the term ***"shuffle"*** implies that each permutation of the elements of array A should be equally likely.

Example

Let consider $A = [1, 2, 3, 4, 5, 6, 7, 8]$. One possible output would be: [8, 7, 4, 6, 3, 1, 2, 5].

II) Let $A = [a_1, a_2, \ldots, a_n]$ be an array of n elements. It is asked to write a pseudo-code of a randomized algorithm that finds the most occurring element of array A and returns any one of its indices randomly with equal probability.

Example

Let A = [-1, 4, 9, 7, 7, 2, 7, 3, 0, 9, 6, 5, 7, 8, 9]. Element with maximum frequency is: at index 7, **OR** at Index 4, **OR** at index 5, **OR** at index 13. All these outputs have equal probability.

Exercise 4: search in a compact sorted list

A compact list is a list whose elements are compressed into an array. For this purpose, we use two arrays V [1..n] to store the elements of the list and P [1..n] to keep track of the chaining pointers between the elements of the list in addition to an integer variable t to locate the head of the list. The keys of the list elements are sorted in ascending order without necessarily being sorted in array V, i.e. $V[t] < V[P[t]] < V[P[P[t]]] < \ldots$ etc., but not necessarily $V[1] < V[2] < V[3] < \ldots$ etc.

Write a randomized function ***position*** **(*x*, *CL*)** which returns the position of an element x in a compact sorted list CL in array V. Note that binary search cannot be applied since the middle of the list cannot be located.

Example

The table below represents the compact sorted list CL whose elements are: (1, 2, 3, 5, 8, 13, 21) with t = 4 (the head of the list). Position (13, CL) = 3, Position (2, CL) = 1.

i	1	2	3	4	5	6	7
***V* [*i*]**	2	3	13	1	5	21	8
***P* [*i*]**	2	5	6	1	7	0	3

Exercise 5: calculation of the median value in an array

Let $A = [a_1, a_2, \ldots, a_n]$ be an array of n elements. Write a pseudo-code of a randomized algorithm that calculates the median value in array A (i.e. element at position $n/2$ if array A is sorted), using the Monte Carlo method.

Example

Let consider A = [12, 3, 8, 13, 5, 18, 9, 11, 1, 12, 6, 17, 11]. The median value is 11; its position is 11 if array A is sorted.

Exercises set 6 (Approximation algorithms)

Remark.

For all the exercises in this set, we deal with NP optimization problems. Therefore, we want to write approximation algorithms guaranteeing good approximation factors.

Exercise 1: set-covering problem

Let $A = \{e_1, e_2, \ldots, e_n\}$ be a set of n elements and $S = \{s_1, s_2, \ldots, s_m\}$ be a collection of m subsets of A such that each subset s_i has a given cost c_i. The objective is to find a sub-collection of S with a minimum cost while covering all elements of A.

Example

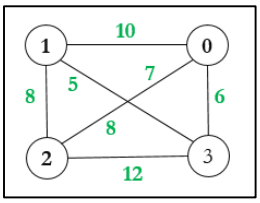
Let $A = \{1, 2, 3, 4, 5\}$ and $S = \{s_1, s_2, s_3\}$ such that: $s_1 = \{1, 3, 4\}$ with cost $c_1 = 5$, $s_2 = \{2, 5\}$ with cost $c_2 = 10$ and $s_3 = \{1, 2, 3, 4\}$ with cost $c_3 = 3$. There are two possible sub-collections covering the elements of A: $\{s_1, s_2\}$ with total cost 15 and $\{s_2, s_3\}$ with total cost 13. Thus, sub-collection $\{s_2, s_3\}$ has a minimum cost 13.

Exercise 2: *k*-centers problem

Let $G = (V, E)$ be a complete undirected graph where $V = \{v_1, v_2, \ldots, v_n\}$ is a set of n vertices (nodes) and E is the set of edges between the vertices such that each edge has a non-negative length (distance between associated vertices). The objective is to select a subset $X \subseteq V$ of k vertices to place warehouses in such a way that the maximum distance of a vertex to a warehouse is minimized.

Example

Consider $V = \{0, 1, 2, 3\}$; the edges and distances between vertices are given in the figure below. Thus, it is asked to place 2 warehouses among these 4 vertices so that the maximum distance of a vertex to a warehouse is minimized. The two warehouses should be placed in vertices 2 and 3. In this case, the maximum distance of a vertex from a warehouse is 6.



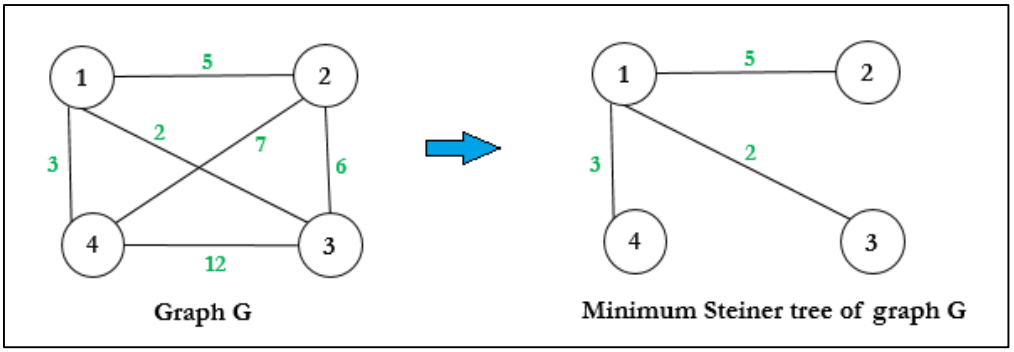
Exercise 3: minimum Steiner tree of a graph

Let $G = (V, E)$ be a connected, weighted and undirected graph, and $R \subseteq V$ be a subset of required vertices (terminals).

The goal is find a minimum Steiner tree of graph G which is a tree of minimum weight while containing all vertices of set R (it may or may not contain the remaining vertices).

Example

Let consider the graph given in the figure below where $R = \{2, 3, 4\}$ is a subset of required vertices. The minimum Steiner tree includes the edges set $E' = \{(1, 2), (1, 3), (1, 4)\}$.



Exercise 4: maximum subset sum

Let $S = \{v_1, v_2, .., v_n\}$ be a set of n positive integers. The goal is to find a subset of set S whose elements sum is as large as possible without exceeding a given integer d (the optimal solution is the sum which approaches d as much as possible).

Example

Let $S = \{2, 3, 5, 7\}$ and $d = 11$.

One possible solution is subset $S_1 = \{2, 7\}$ with elements sum equal to 9. There are two optimal solutions with elements sum equal to 10: $S_{opt1} = \{3, 7\}$ and $S_{opt2} = \{2, 3, 5\}$.

Exercise 5: load balancing

Let $M = \{m_1, m_2, .., m_k\}$ and $T = \{j_1, j_2, .., j_n\}$ be a set of k machines and n jobs such that the processing time of each job j_i is t_i. The goal is to assign each job to a machine such that the makespan is minimized. The makespan is defined as the largest total processing time of a machine.

Example

Let consider a set $M = \{m_1, m_2, m_3\}$ composed of three machines. The table below gives the considered jobs and their processing times.

Jobs (j_i)	j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8	j_9	j_{10}	j_{11}	j_{12}
Processing times (t_i)	3	4	2	3	1	6	4	4	3	2	1	5

The optimal solution consists in assigning jobs (j_1, j_2, j_3, j_4, j_5) to m_1, jobs (j_6, j_7, j_9) to m_2 and jobs ($j_8, j_{10}, j_{11}, j_{12}$) to m_3 with a makespan of 13.

References

Remark. All liks were consulted on july 20, 2023.

1. Amiar Lotfi. Méthodes algorithmiques. Course handout. University of Oum El Bouaghi.

2. Various online articles published for GeeksforGeeks (a computer science portal for geeks developed by Sandeep Jain in 2009) and available at: https://www.geeksforgeeks.org/.

3. Various online articles published for Wikepedia and available at: https://www.wikipedia.org/.

4. Karine Zampieri, Stéphane Rivière, Béatrice Amerein-Soltner. Algorithmes gloutons (2018). Université des sciences en ligne. Chapter available at: https://ressources.unisciel.fr/algoprog/s35techn/emodules/gl00macours1/res/gl00cours-texte-xxx.pdf.

5. Pierre Béjian. MATROIDES ET ALGORITHMES GLOUTONS : UNE INTRODUCTION (2003). Chapter available at: http://pauillac.inria.fr/~quercia/documents-info/Luminy-2003/beijan/matroide_papier.pdf

6. Souad Aroussi. Algorithmes gloutons (2013). University of Blida. Chapter available at: https://sites.google.com/a/esi.dz/informatiqueblida/

7. Ana Busic. Conception d'algorithmes et applications. Chapter available at: www.di.ens.fr/~busic/cours/LI325/slidesCAAC8_1213.pdf

8. Coding Freak. Top 7 Greedy Algorithm Problems (2018). Online article published for Techie Delight and available at: https://medium.com/techie-delight/top-7-greedy-algorithm-problems-3885feaf9430.

9. Sylvie Hamel. Algorithmes diviser-pour-régner. University of Montréal. Chapter available at: http://www.iro.umontreal.ca/~hamelsyl/AlgosDiviser_pour_regner.pdf.

10. Karine Zampieri, Stéphane Rivière, Béatrice Amerein-Soltner. Paradigme diviser pour régner (2018). Université des sciences en ligne. Chapter available at: https://ressources.unisciel.fr/algoprog/s35techn/emodules/dr00macours1/res/dr00cours-texte-xxx.pdf.

11. R. Lelouche. DIVISER POUR RÉGNER. Chapter available at: http://www2.ift.ulaval.ca/~bherer/Transparents_Ch07.pdf

12. Karine Zampieri, Stéphane Rivière, Béatrice Amerein-Soltner. Programmation dynamique (2018). Université des sciences en ligne. Chapter available at: https://ressources.unisciel.fr/algoprog/s35techn/emodules/dy00macours1/res/dy00cours-texte-xxx.pdf.

13. Manuel Lafond. Algorithmes et structures de données. Course notes, University of Sherbrooke.

14. R. Lelouche. PROGRAMMATION DYNAMIQUE. Chapter available at: http://www2.ift.ulaval.ca/~bherer/Transparents_Ch08.pdf.

15. DESIGN AND ANALYSIS OF ALGORITHMS (Lecture notes B.TECH III YEAR). MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY (Autonomous Institution – UGC, Govt. of India)

16. Steven Skiena. Backtracking. Department of Computer Science, State University of New York Stony Brook, NY 11794–4400.

17. Soni Upadhyay. What is Backtracking Algorithm? Types, Examples & its Application (updated 2023). Online article published and available at: https://www.simplilearn.com/tutorials/data-structure-tutorial/backtracking-algorithm

18. Christine SOLNON. Résolution de CSPs. Online chapter available at: https://perso.liris.cnrs.fr/christine.solnon/Site-PPC/session3/e-miage-ppc-sess3.htm#grd3

19. James Aspnes. Notes on Randomized Algorithms (2023). Course notes availble at: http://www.cs.yale.edu/homes/aspnes/classes/469/notes.pdf

20. Robert Cori. Algorithmes probabilistes. Online presentation available at: http://denif.ens-lyon.fr/data/concept_analyse_algo_x/2007/cours/cours7.pdf

21. R. Lelouche. ALGORITHMES PROBABILISTE. Chapter available at: http://www2.ift.ulaval.ca/~bherer/Transparents_Ch10.pdf.

22. François Schwarzentruber. Algorithmes d'approximation (2021). Chapter availble at: http://people.irisa.fr/Francois.Schwarzentruber/algo2/03approximation.pdf.

23. Frédéric Vivien. Algorithmes d'approximation. Online course available at: https://www.youtube.com/watch?v=g-NLgxhCW-A

24. S. Le Digabel. Algorithmes d'approximation (2018). Ecole Polytechnique de Montréal. Chapter available at: https://www.gerad.ca/Sebastien.Le.Digabel/MTH6311/9_algos_approx.pdf

25. Subhash Suri. Approximation Algorithms (2021). Chapter available at: https://sites.cs.ucsb.edu/~suri/cs130b/BinPacking

26. Cyril Gavoille. Analyse d'Algorithme (2022). Course notes available at: https://dept-info.labri.fr/~gavoille/UE-AA/cours.pdf

27. Aris Pagourtzis. Approximation Algorithms. Online presentation available at: https://courses.corelab.ntua.gr/pluginfile.php/2328/course/section/362/approximation.pdf