U.F.R SCIENCES ET TECHNIQUES
Département d'Informatique
B.P. 1155
64013 PAU CEDEX
Téléphone secrétariat : 05.59.40.79.64
Télécopie : 05.59.40.76.54

V-Complexité des algorithmes et Classes de complexité : partie 1

I-Notion de complexité d'un algorithme
II-Complexité en temps d'un algorithme
III-Calcul de la complexité
IV-Classes de complexité

I- Notion de complexité d’un algorithme

La **théorie de la calculabilité** révèle que d'un point de vue **algorithmique**, il existe deux classes de problèmes:

- ceux qui **possèdent** une solution
- et ceux qui n'en **possèdent pas**.

La **théorie de la complexité** se concentre sur la classe des problèmes possédant une solution.

1-Problématique de la complexité

Les problèmes possédant une solution, peuvent, en général, être résolus par **plus d'un** algorithme.

Soit P, un **problème**, et :

A1, A2, A3,…..

les algorithmes exprimant sa **solution**.

Dans ce cas précis, **comment l'ingénieur choisit-il** un algorithme pour résoudre le problème P ?

Une règle générale est de prendre un algorithme qui soit facile:

- à **comprendre**,
- et à **mettre en œuvre**.

Mais lorsque l'aspect **performance** est important pour le problème P, comment choisir :

-un algorithme **efficace**

-parmi les algorithmes A1, A2, A3,….. ?

La **théorie de la complexité** répond à cette problématique en s’appuyant sur une estimation **théorique**:

-du **temps de calcul**

-et des **besoins en espace mémoire.**

2-Efficacité d’un algorithme

La théorie de la complexité a pour objet principal l'estimation de l'**efficacité** des algorithmes.

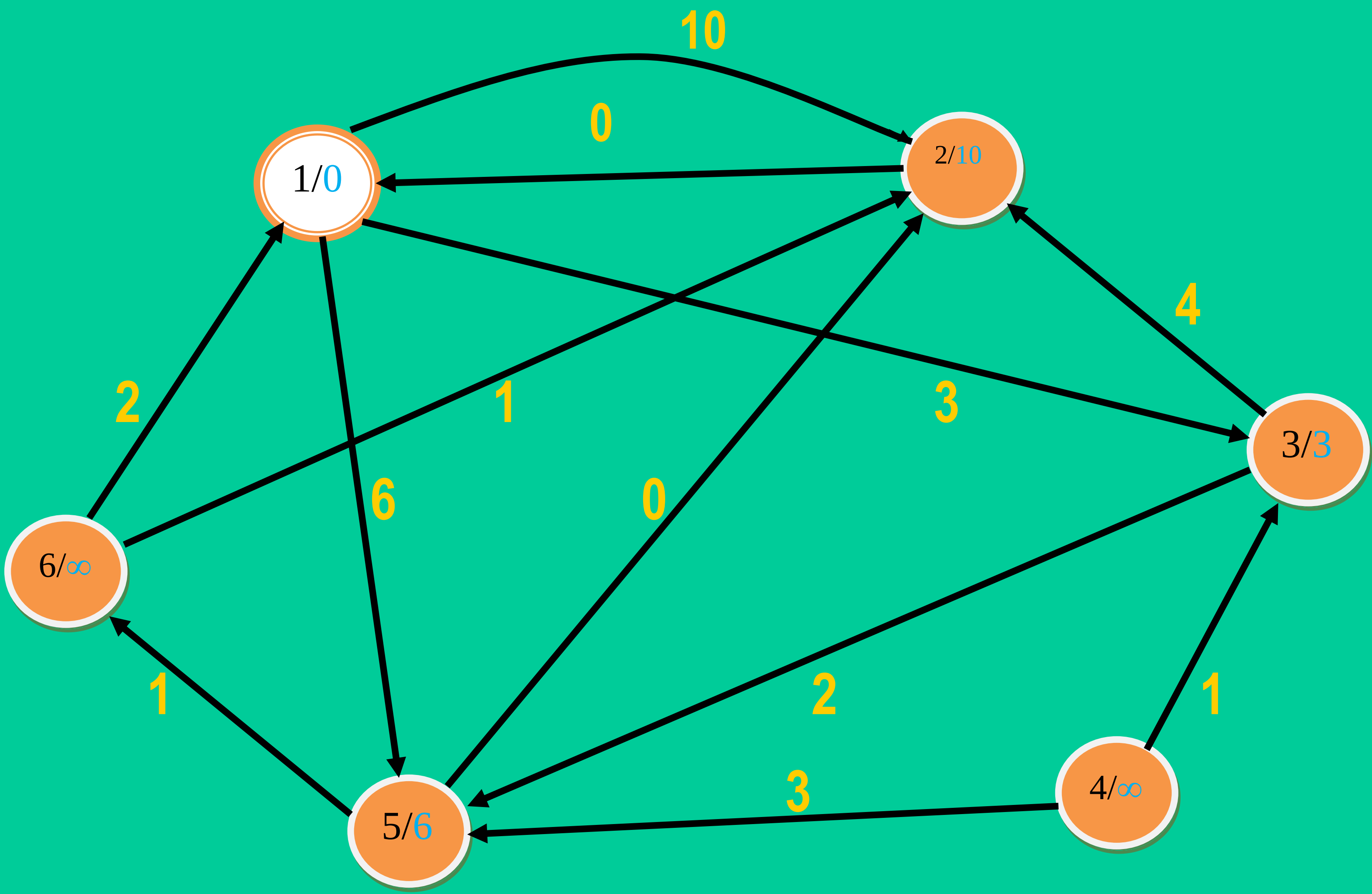
Elle soulève la question suivante:
« Entre différents algorithmes réalisant une même tâche :
- quel est le plus **rapide** ?
- et dans quelles **conditions** ? »

Dans les années 1960 et jusqu'au début des années 1970, on a découvert de nombreux algorithmes sur les **graphes**.

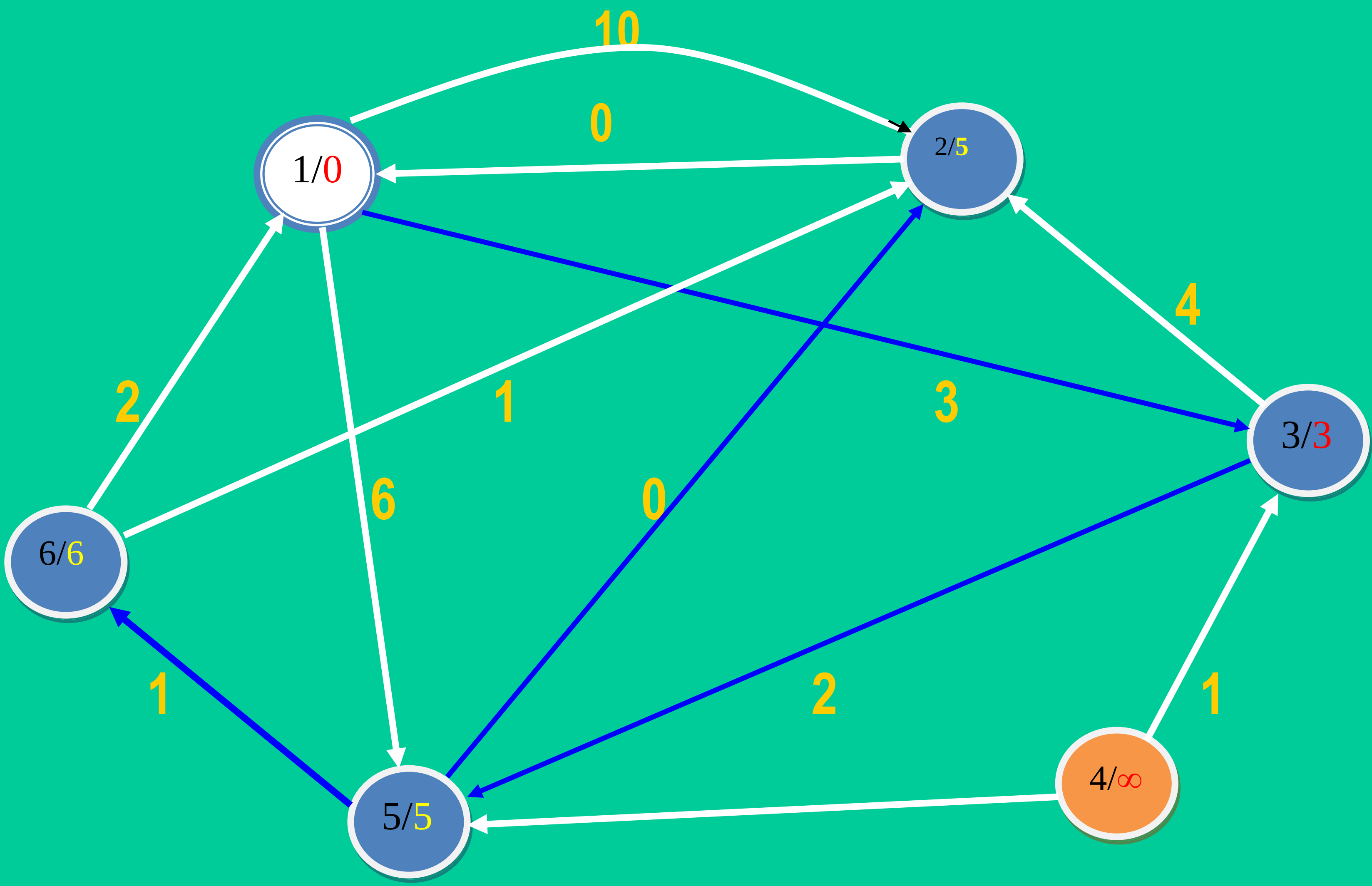
Parmi les plus appliqués en ingénierie:

- "recherche de chemin optimal": Bellman, Dijkstra,
- "couverture minimum" » »: Kruskal ou de Prim,
- "calcul du flot maximum": Ford-Fulkerson

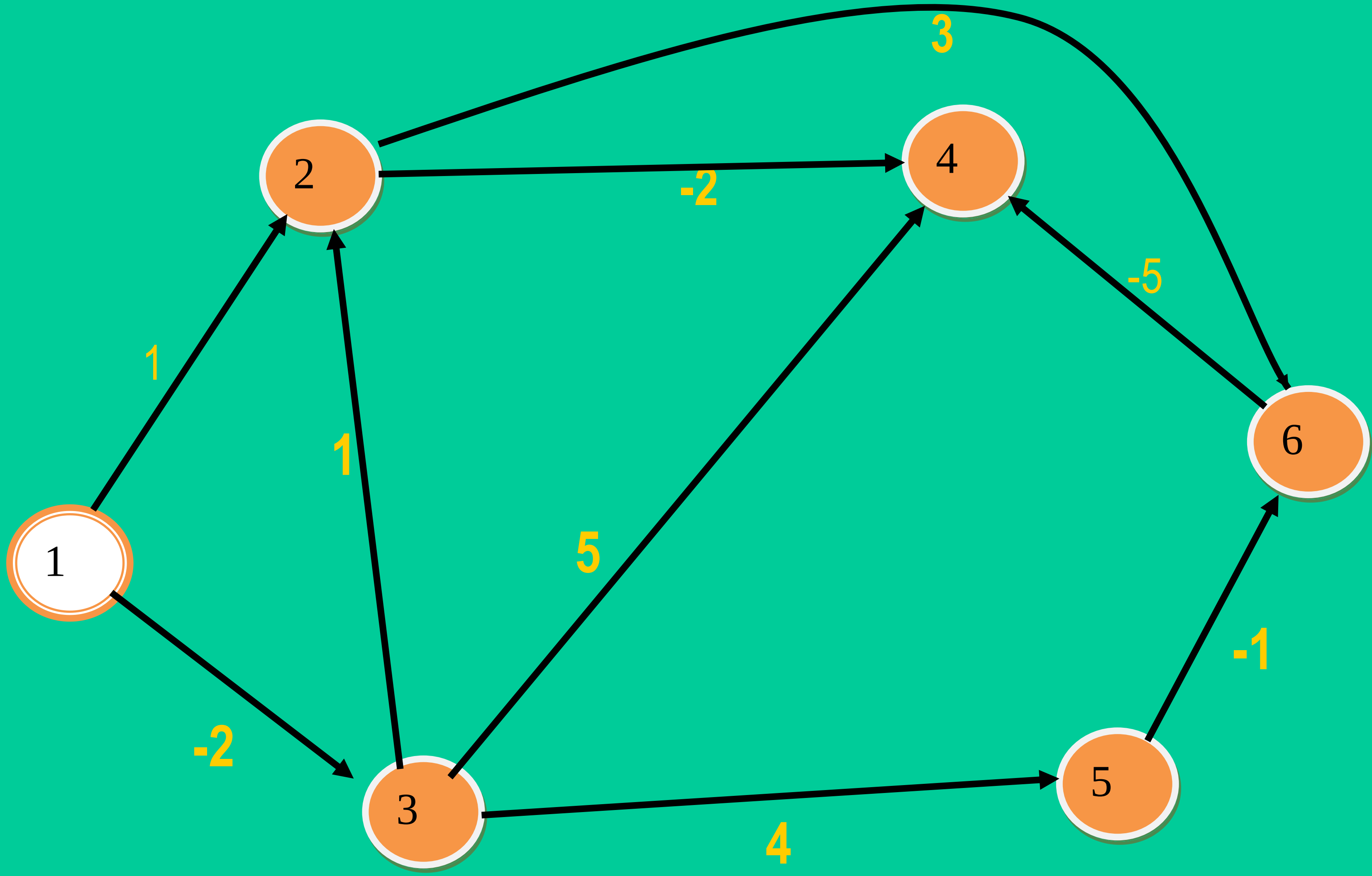
Graphe original



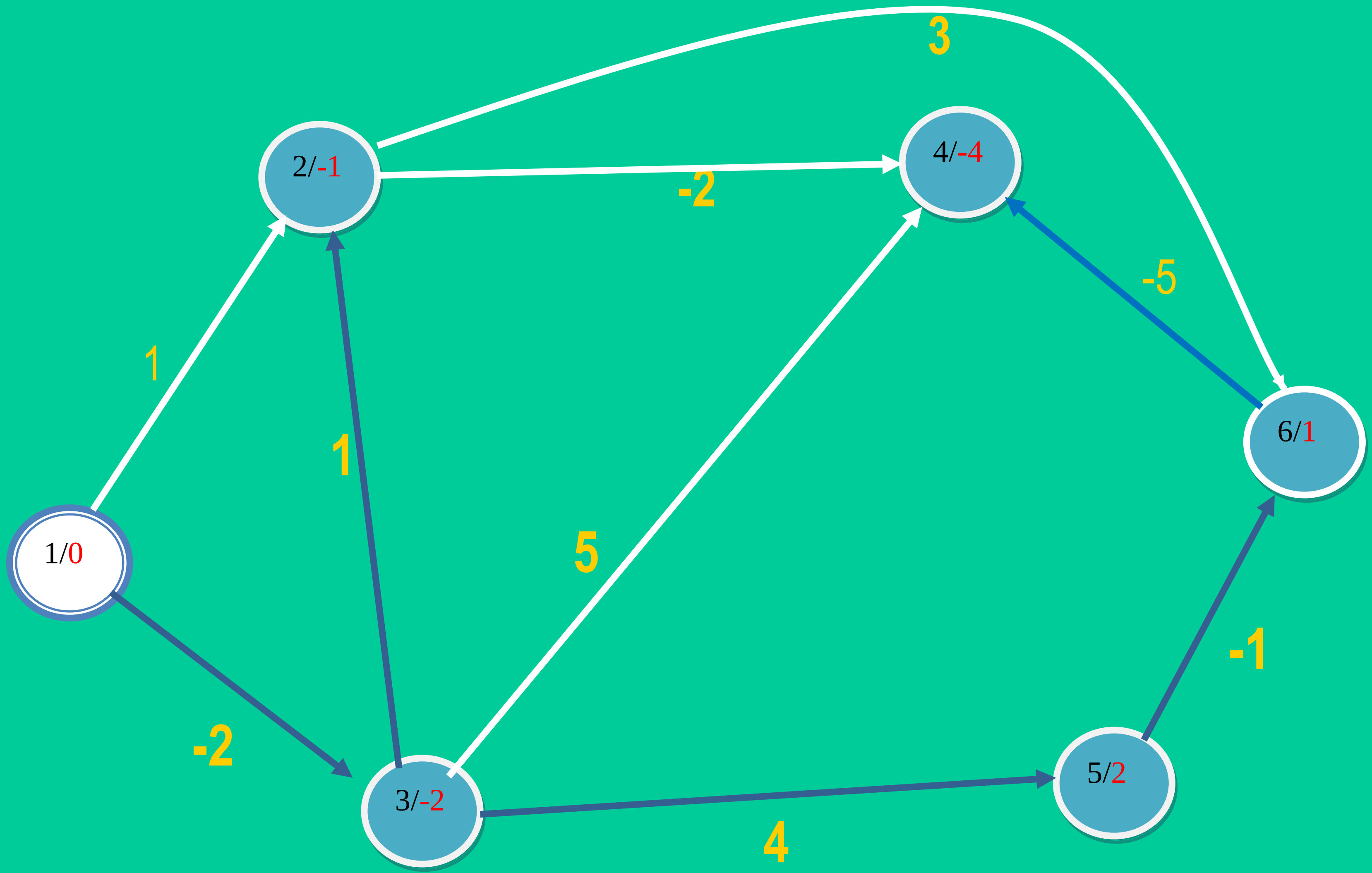
Application algorithme de Dijkstra



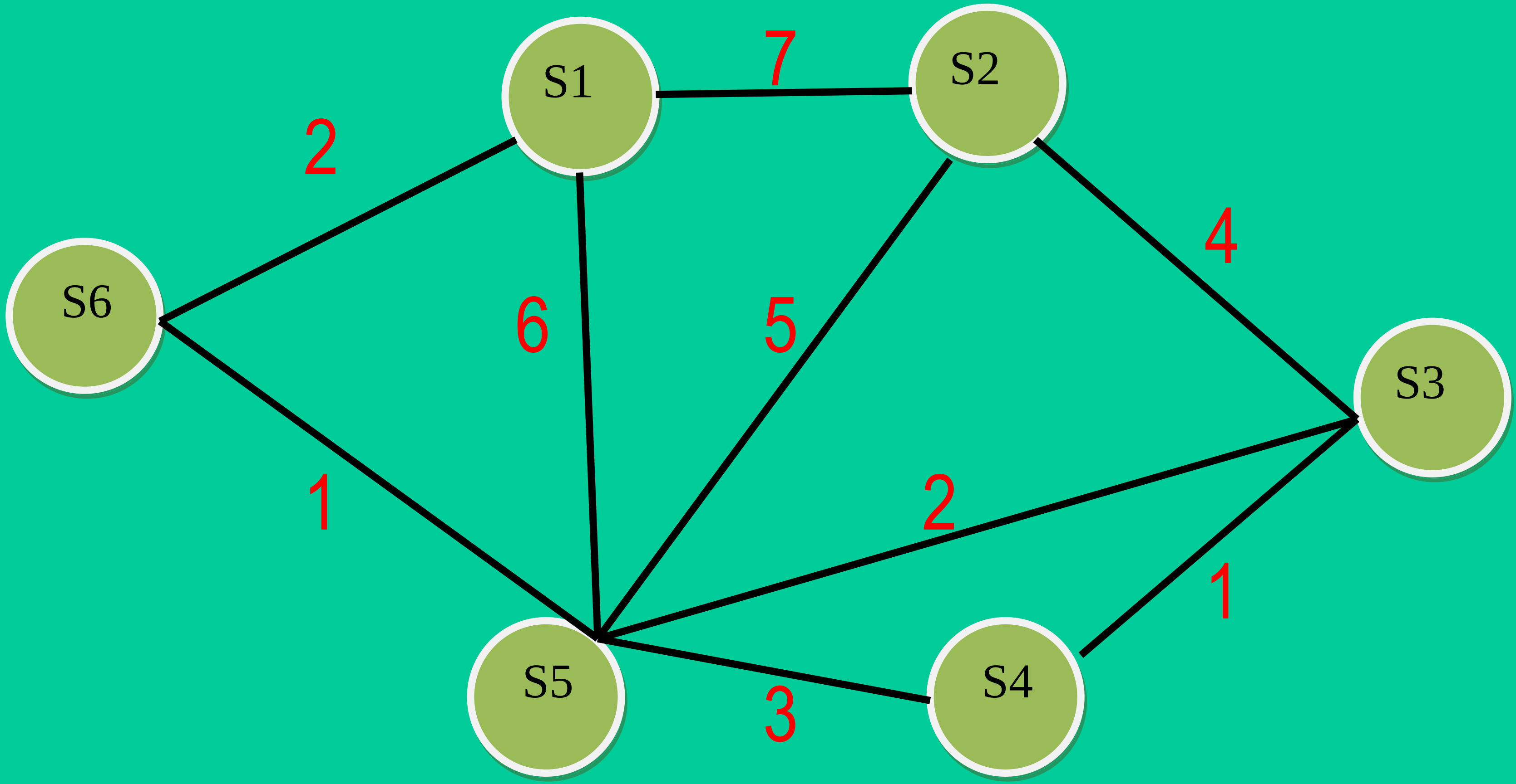
Graphe original



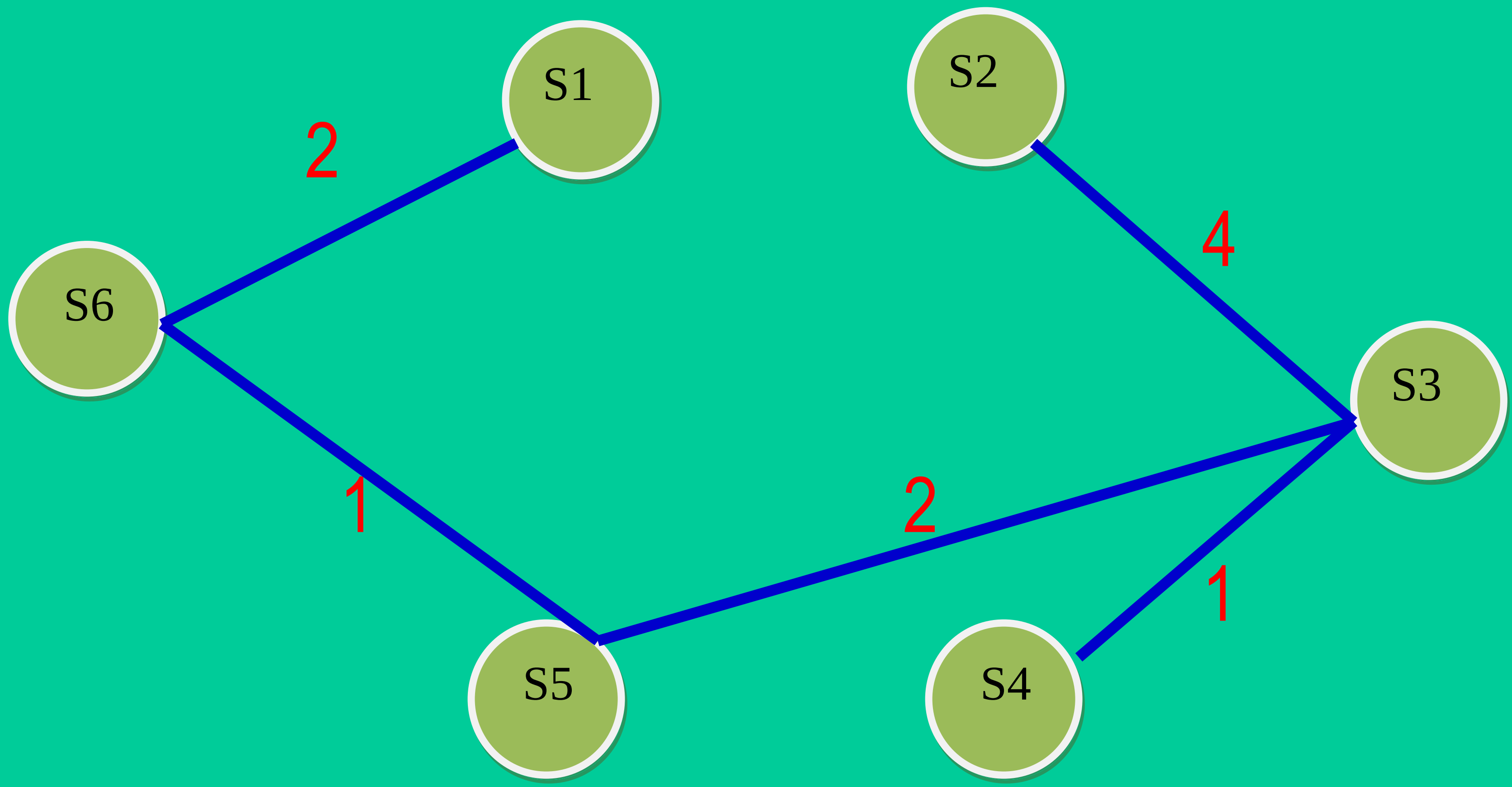
Application algorithme de Bellman



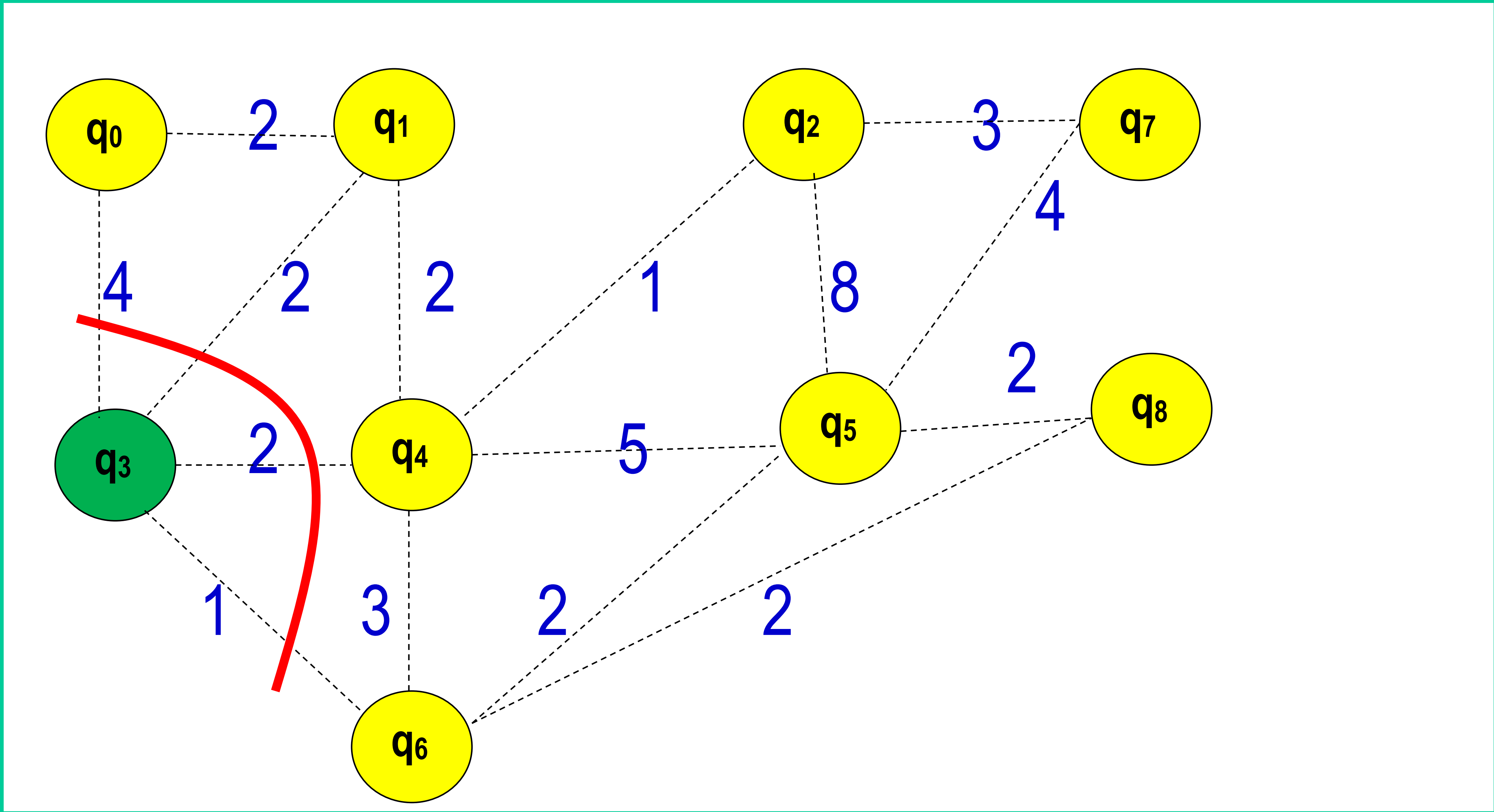
Graphe original



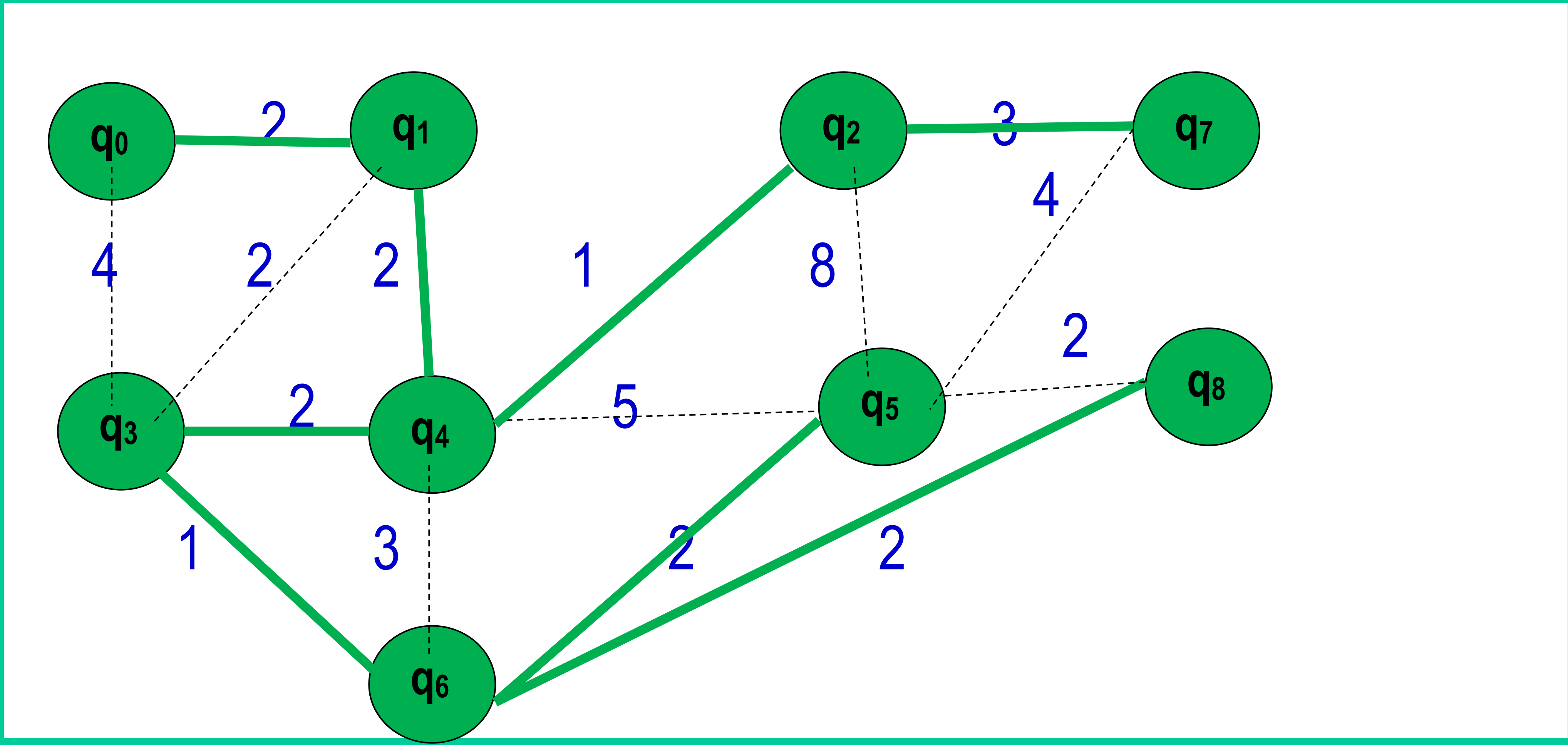
Application algorithme de Kruskal



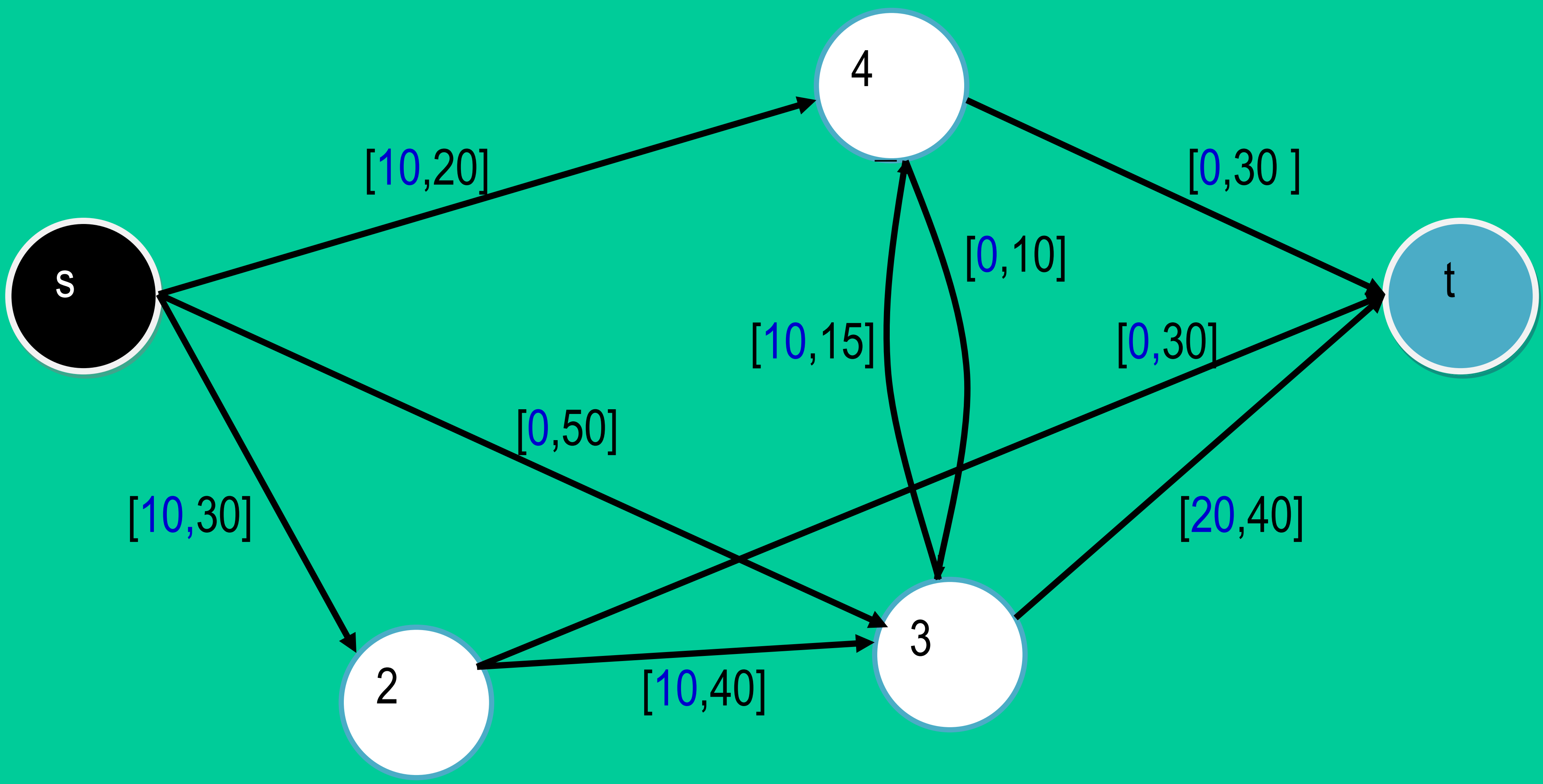
Graphe original



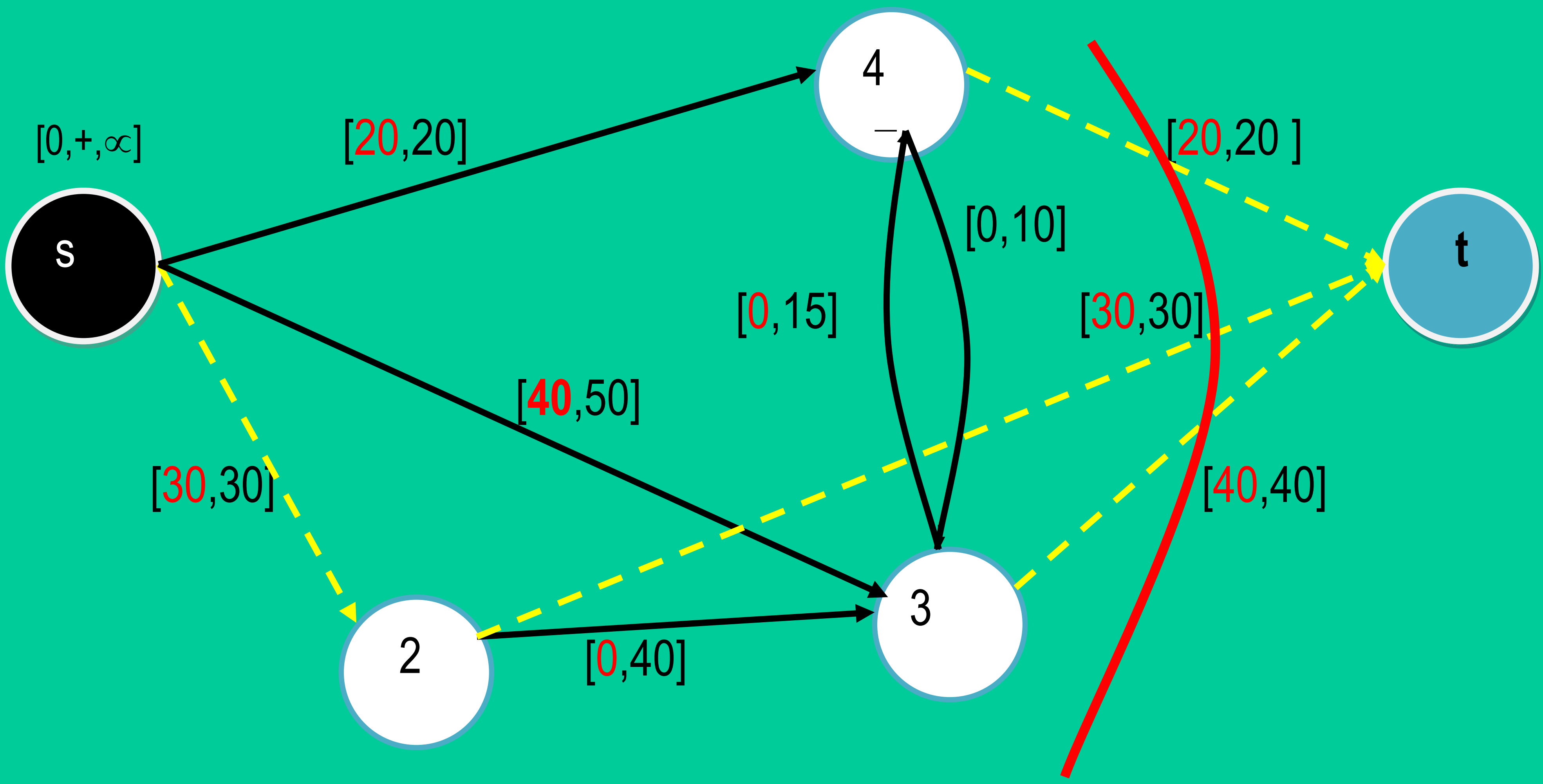
Application algorithme de Prim



Réseau transport



Application algorithme de Fulkerson



Le problème est qu'on estimait pas leur **efficacité**.

On se «contentait» de publier des résultats du genre:

-«cet algorithme se déroule en **6** sec,»

-« avec un tableau de **50 000** entiers en entrée, »

- « sur un ordinateur **IBM 360/91** »

- « le langage de programmation **PL/I** a été utilisé avec les optimisations standards ».

Une telle démarche rendait impossible la **comparaison** des algorithmes entre eux.

Pourquoi ?:

La «mesure» était **fortement dépendante** :

- du **processeur** utilisé,
- des temps d'accès à la **mémoire** RAM et disque,
- du **langage de programmation**,
- du **compilateur** utilisé,
- et du **système d'exploitation**.

3-Nouvelle approche de la complexité

Une approche **indépendante** des facteurs matériels est devenue nécessaire pour:

- évaluer l'**efficacité** des algorithmes
- et les **comparer** entre eux.

Donald Knuth fut un des premiers à l'appliquer de façon systématique dans «*The Art of Computer Programming*.»

La théorie de la **complexité** aborde le problème de l'estimation **théorique** de l'**efficacité** des algorithmes.

Elle soulève la problématique suivante:

« Entre différents algorithmes réalisant une même tâche :

-quel est le plus **rapide**?

-et dans quelles **conditions?** »

4-Aspects de la complexité d’un algorithme

La complexité d’un algorithme met en exergue deux aspects:

-la **complexité en espace**,

-la **complexité en temps**

La **complexité en espace** s’attache à :

- évaluer l’espace mémoire nécessaire,
- en fonction de la taille des données d’entrée.

Dans de nombreuses applications:

- robots embarqués,
- réseaux mobiles,
- traitement d'images,
- CAO/DAO,

Il est nécessaire d'étudier la complexité en espace **mémoire.**

Surtout, lorsque l'algorithme requiert de la **mémoire supplémentaire**.

La **complexité en temps** d'un algorithme fait référence au facteur **temps d'exécution** de cet algorithme.

C'est cet aspect de la complexité qui va être développé dans ce qui va suivre.

II-Complexité en temps d’un algorithme

La complexité en temps s’appuie sur une estimation
du **nombre d'opérations élémentaires** en fonction:

- -de la **taille** des donnée,
- -de la **nature** de ces données.

1-Opérations élémentaires

Les opérations élémentaires considérées sont :

-les affectations de variables,

-les tests de comparaison,

-les opérations classiques :+,*,… réalisées par l'algorithme: calculs matriciels ou des polynômes

2-Taille d’une entrée

Ce qu’on appelle **taille** d’une entrée peut varier d’un algorithme à l’autre.

Pour un algorithme de **tri**, par exemple, cette taille est mesurée par le nombre **n d’éléments à trier**.

Pour un algorithme qui résout :

- **n** équations linéaires
- à **n** inconnues,

il est normal de prendre **n** pour taille.

D'autres algorithmes pourraient utiliser :

- la **valeur** d'une entrée particulière,
- ou la **longueur** d'une liste,
- ou la **taille** d'un tableau,
- ou une **combinaison** de ces quantités.

Par la suite, on désigne par :

- **d,** la **donnée d’entrée**,
- et **n** sa **taille** .

La **nature** de la donnée **d** est un facteur souvent **déterminant** dans l’estimation de la complexité.

Par exemple, faut-il choisir :

- une **liste** ?
- ou un **arbre** ?

3-Coût relatif à une donnée

Le **coût** d'un algorithme **A :**

-pour une certaine donnée **d**

-de taille **n**

est le nombre **p** d'**opérations élémentaires** nécessaires au traitement de la **donnée** **d.**

Ce **coût** est noté provisoirement **$COUT_A(d)$**

Coût dans le pire des cas

$$\mathrm{Max}_A(n) = \max\{\mathrm{COUT}_A(d) \mid d \in D_n\}$$

D_n désigne l'ensemble de données de taille **n**.

Coût dans le meilleur des cas

$$Min_A(n) = \min \{COUT_A(d) \mid d \in D_n\}$$

Coût en moyenne

$$\mathrm{Moy}_A(n) = \sum_{d \in D_n} p(d)\,\mathrm{COUT}_A(d)$$

Où :

- **p(d)** est la probabilité d'avoir en entrée la donnée **d** parmi toutes les données de taille **n**.

Si toutes les données sont **équiprobables**, alors on a :

$$Moy_A(n) = (1/|D_n|) \sum_{d \in D_n} COUT_A(d)$$

Où $|D_n|$ désigne le cardinal de D_n.

Un aperçu du calcul du coût

Soit **t** un tableau de taille **n**.

Le tableau **t** contient des nombres entiers de **1** à **k.**

Soit **a** un entier entre **1** et **k**.

Soit **A** l'algorithme de recherche de **a** ci-après :

```
search:=proc(t,n,a)
        local i;
        for i from 1 to n do
            if t[i]=a then return(i) fi;
        od;
        return (0);
end;
```

Il définit une fonction **search** qui retourne :

-la position du premier **a** rencontré s'il existe,

-et **0** sinon.

On convient qu'une itération de la boucle **for**

-représente **une** instruction

-nécessitant pour son exécution,**1 unité** de temps

La complexité en temps de l'algorithme A peut être estimée selon l'**analyse** des cas suivants :

1-"**<u>En pire des cas</u>**":

$$\max\{COUT_A(\mathbf{t}) \mid \mathbf{t} \in D_n\} = \mathbf{n}$$

Le tableau **t** ne contient pas l'élément **a.**

2-"**<u>En meilleur des cas</u>**" :

$$\min\{COUT_A(\mathbf{t}) \mid t \in D_n\} = \mathbf{1}$$

Le tableau **t** a pour **premier** élément **a**.

3-"**<u>En moyenne</u>**":

$$(1/|D_n|) \sum_{d \in D_n} COUT_A(\mathbf{t}) = k(1-(1-1/k)^{\mathbf{n}})$$

Les nombres entiers de **1** à **k** apparaissent de manière **équiprobable**.

4-Calcul du temps d'exécution

En pratique, pour estimer la **complexité en temps,** d'un algorithme on cherche à évaluer :

-le **temps d'exécution** théorique

-pour n'**importe quelle** entrée **d** de taille **n**.

On cherche alors à définir une fonction **T** où :

T(n)

est appelée **temps d'exécution** de l'algorithme.

n est appelé **paramètre de complexité**

Dans le calcul de **T(n)** on fait abstraction des **unités**.

Temps d'exécution effectif

On estime le temps d'exécution **effectif** d'un algorithme en terme d'une expression de la forme:

$$\xi \times \mathbf{T(n)}$$

Où ξ est un **facteur constant**.

Le facteur multiplicatif ξ est introduit, également, pour faire **abstraction de la puissance** de la machine exécutant l'algorithme.

Exemple de calcul

Soit l'algorithme du **tri par sélection** :

```
(1) pour i := 1 à p-1 faire

(2)      petit := i
(3)      pour j := i+1 à p faire
(4)           si a[j] < a[petit]
(5)                alors petit := j
              finsi
         finpour
(6)      tempo := a[petit]
(7)      a[petit] := a[i]
(8)      a[i] := tempo
Finpour
```

Soit à estimer le **temps d'exécution** **T(n)** de la portion suivante :

```
(2)     petit := i
(3)     pour j := i+1 à p faire
(4)         si a[j] < a[petit]
(5)             alors petit := j
            finsi
        finpour
```

On convient de compter une **unité de temps** pour chaque exécution:

- d'une **affectation**
- ou d'un **test**.

Premier calcul:

On doit avoir:

- 1 unité pour l'affectation de **petit:** petit := i
- 1 unité pour l'**initialisation de j :** j := i+1
- 1 unité pour le **premier test de j** : j > p

D'où un sous-total de 3 unités.

<u>Deuxième calcul:</u>

Pour chaque parcours de la boucle **<u>pour</u>**, on compte:

-1 unité pour **incrémenter j**,

-1 unité pour **tester** si **j > p**

Dans le **corps** de la boucle: lignes (4) et (5):

(4)	**<u>si</u>** a[j] < a[petit]
(5)	**<u>alors</u>** petit := j

-le **test** (4) est toujours exécuté,

-mais l'**affectation** (5) n'est exécutée que si le test est vrai.

Le **corps** peut prendre donc:

-soit 1 unité,

-soit 2.

Dans le **pire des cas**, le corps prend **2** unités pour chacune de ses exécutions.

Donc à chaque itération:

-l'incrémentation de **j** et son test coûtent **2 unités**

-et son corps **2 unités**.

Comme la boucle est itérée **(p-i)** fois.

On a:

$$(p-i)(2 +2) = 4(p-i) \text{ unités.}$$

A cela, on doit ajouter les 3 unités du premier calcul:

$$4(p-i) + 3 \quad \text{unités.}$$

Le coût total est donc:

$$T(n) = 4\,(p-i) + 3.$$

La taille, notée **n** des données traitées est égale à la taille du tableau a[**i**..**p**], donc:

$$\mathbf{n = p - i + 1}$$

Le temps d'exécution recherché est donc:

$$\begin{aligned} T(\mathbf{n}) &= 4(p-i)+3 \\ &= 4(p-i+1)-1 \\ &= \mathbf{4n - 1.} \end{aligned}$$

5-Analyse asymptotique

Le calcul précédent calcule la **complexité** de façon **exacte**.

Or, **en pratique,** un tel calcul n'est:

-ni **raisonnable:** vu la quantité d'instructions de la plupart des programmes,

-ni **utile :** puisqu'il s'agit seulement de **comparer** deux algorithmes

Vers un calcul approximatif

Aussi, **trois approximations** peuvent être proposées:

1-on ne considère souvent que la **complexité au pire**,

2-on ne calcule que la **forme générale** de la complexité,

3-on ne regarde que le **comportement asymptotique** de la complexité

En théorie de complexité, c'est cette dernière approximation qui est retenue.

Elle part de l'hypothèse qu'on cherche à estimer la complexité en temps pour les données de **grande taille**.

On estime systématiquement la complexité asymptotique grâce aux notations de **Landau**.

La **notation grand O**, aussi appelée **symbole de Landau**, est utilisée pour décrire le **comportement asymptotique** des fonctions.

Par exemple, avec une telle notation, on peut estimer que:

- la **complexité** de l'algorithme de Dijkstra
- est en **$O(n^2)$** .

```java
/**
 * Implementation de l'algorithme de Dijkstra.
*/
public class Dijkstra implements Pathfinding {
	private int[][] graph;
	private int[] distanceFromStart;
	private boolean[] activesNodes;
	private int dim;
	private int[] precedences;
	private void activeAdjacents(final int node)
      {
		int distanceTo;
		for (int to = 0; to < this.dim; to++)
			if (this.isAdjacent(node, to) && (distanceTo = this.distanceFromStart[node] +

		      this.graph[node][to])
		   < this.distanceFromStart[to])this.activeNode(node, to, distanceTo);
	}
```

```java
private void activeNode(final int from, final int node, final int distance) {
	this.distanceFromStart[node] = distance;
	this.precedences[node] = from;
	this.activesNodes[node] = true;
}

private List<Integer> buildPath(final int end) {
	final List<Integer> path = new ArrayList<Integer>();
	path.add(end);
	// utilisation d'une boucle do-while pour conserver le point de depart
	// et d'arrivee dans la liste même lorsque le point de depart correspond
	// au point d'arrivee
	int position = end;
	do {
		path.add(0, this.precedences[position]);
		position = path.get(0);
	} while (this.distanceFromStart[position] != 0);

	return path;
}

@Override
```

```java
public List<Integer> getPath(final int[][] graph, final int start, final int end) {
	return this.getPath(graph, new int[] { start }, new int[] { end });
}

@Override
public List<Integer> getPath(final int[][] graph, final int start, final int[] ends) {
	return this.getPath(graph, new int[] { start }, ends);
}

@Override
public List<Integer> getPath(final int[][] graph, final int[] starts, final int[] ends) {
	Arrays.sort(ends);

	// initialisation des variables necessaires a la resolution du probleme
	this.init(graph, starts);

	// calcul des distances par rapport au point de depart et recuperation
	// du point d'arrivee
	final int end = this.processDistances(ends);

	return (end != -1) ? this.buildPath(end) : null;
}

private void init(final int[][] graph, final int[] start)
```

```
{
        this.graph = graph;
        this.dim = graph.length;
        this.activesNodes = new boolean[this.dim];

        this.precedences = new int[this.dim];
        Arrays.fill(this.precedences, -1);

        this.distanceFromStart = new int[this.dim];
        Arrays.fill(this.distanceFromStart, Integer.MAX_VALUE);

        for (final int value : start)
            this.activeNode(value, value, 0);
    }

    private boolean isAdjacent(final int from, final int to) {
        return this.graph[from][to] >= 0;
    }

    private int processDistances(final int[] ends) {
        // selectionne le prochain noeud a analyser (noeud courant)
        final int next = this.selectNextNode();
        if (next == -1)
            return -1;
```

```
            if (Arrays.binarySearch(ends, next) >= 0)
                return next;
            // active les prochains noeuds a analyser a partir du noeud courant
            this.activeAdjacents(next);
            // desactive le noeud courant
            this.activesNodes[next] = false;
            // appel recursif de la methode pour traiter le prochain noeud
            return this.processDistances(ends);
        }
        private int selectNextNode() {
            int nextNode = -1;
            for (int node = 0; node < this.dim; node++)
                if (this.activesNodes[node] && (nextNode == -1 || this.distanceFromStart[node] <
                                                this.distanceFromStart[nextNode]))
                    nextNode = node;

            return nextNode;
        }
}
```

On peut, également, estimer que:

- la complexité de l'algorithme de **Bellman Ford**
- est en **O(m x n)** ou simplement en **O(n^3)**

```
import java.io.*;
import java.util.*;
public class BellmanFord {
    LinkedList<Edge> edges;
    int d[];
    int n,e,s;
    final int INFINITY=999;
    private static class Edge  {
        int u,v,w;
        public Edge(int a, int b, int c)     {
            u=a;
            v=b;
            w=c;
        }
    }
```

```java
BellmanFord() throws IOException {
    int item;
    edges = new LinkedList<Edge>();
    BufferedReader inp = new BufferedReader (new InputStreamReader(System.in));

    System.out.print("Entrer n ");
    n = Integer.parseInt(inp.readLine());

    System.out.println("Matrice cout");
    for(int i=0;i<n;i++) {
        for(int j=0;j<n;j++)   {
            item = Integer.parseInt(inp.readLine());
            if(item != 0)
                edges.add(new Edge(i,j,item));
        }
    }
    e = edges.size();
    d = new int[n];

    System.out.print("Entrer la source");
    s = Integer.parseInt(inp.readLine());
}
```

```
void relax() {
    int i,j;
    for(i=0;i<n;++i)
        d[i]=INFINITY;

    d[s] = 0;
    for (i = 0; i < n - 1; ++i) {
        for (j = 0; j < e; ++j) { //ici, calcul de chemin optimal
            if (d[edges.get(j).u] + edges.get(j).w < d[edges.get(j).v]) {
                d[edges.get(j).v] = d[edges.get(j).u] + edges.get(j).w;
            }
         }
     }
}

boolean cycle() {
    int j;
    for (j = 0; j < e; ++j)
        if (d[edges.get(j).u] + edges.get(j).w < d[edges.get(j).v])
             return false;
    return true;
}
```

```
    public static void main(String args[]) throws IOException   {
        BellmanFord r = new BellmanFord();
        r.relax();
        if(r.cycle()) {
            for(int i=0;i<r.n;i++)
                System.out.println(r.s+" ==> "+r.d[i]);
        } else {
            System.out.println("Présence de cycle absorbant ");
        }
    }
}
```

Qu’est-ce que la complexité asymptotique ?

La **complexité asymptotique** indique :

-avec quelle **rapidité**

-une fonction **croit** .

La lettre **O** est utilisée parce que la courbe de la croissance d'une fonction est aussi appelée "**ordre**"

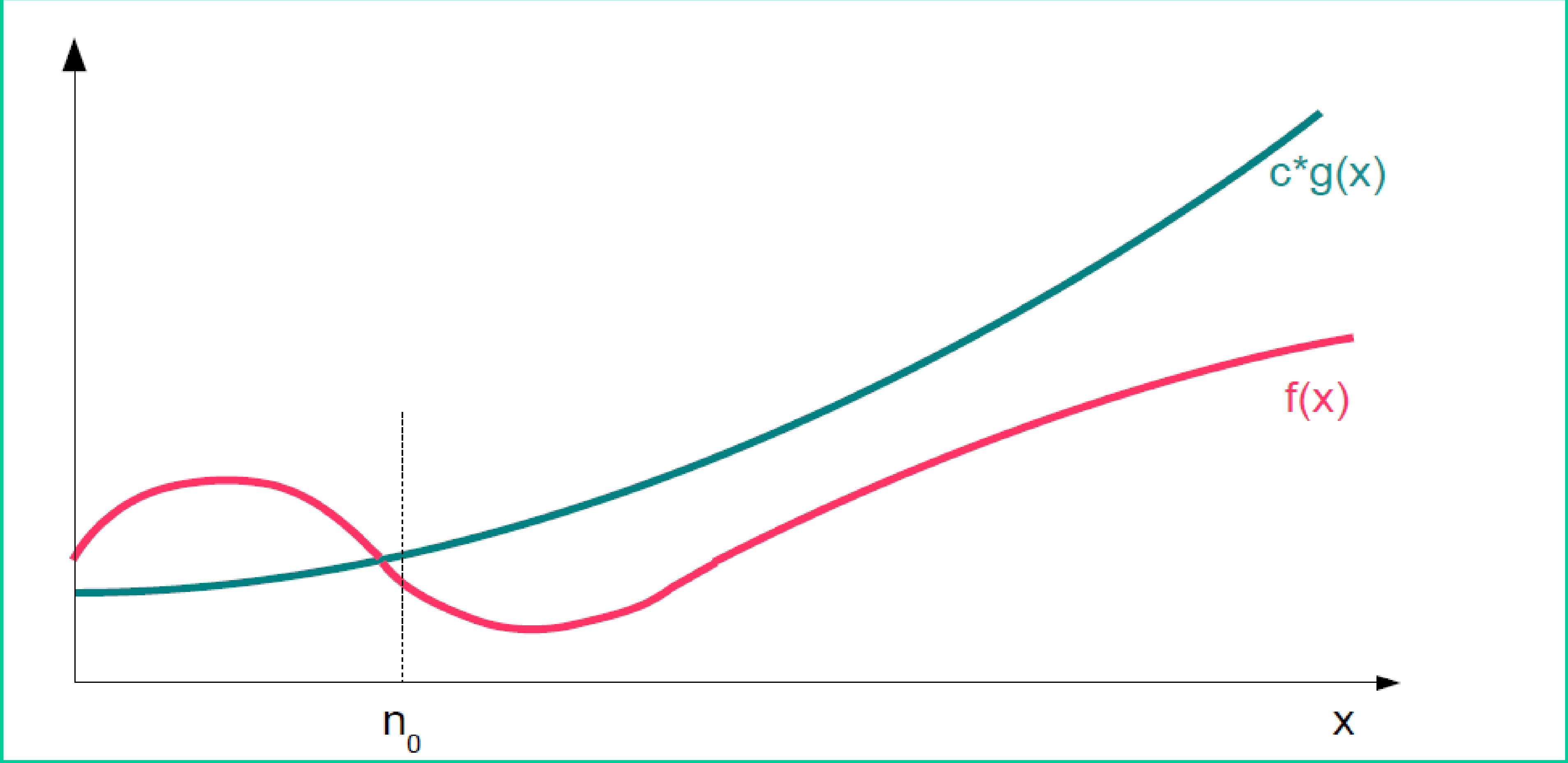
Informellement, cette notion vient de deux idées simples basées sur:

- la "domination" d'une fonction **f**
- par une autre fonction **g**.

Ce qui signifie que g "croit" plus vite que f

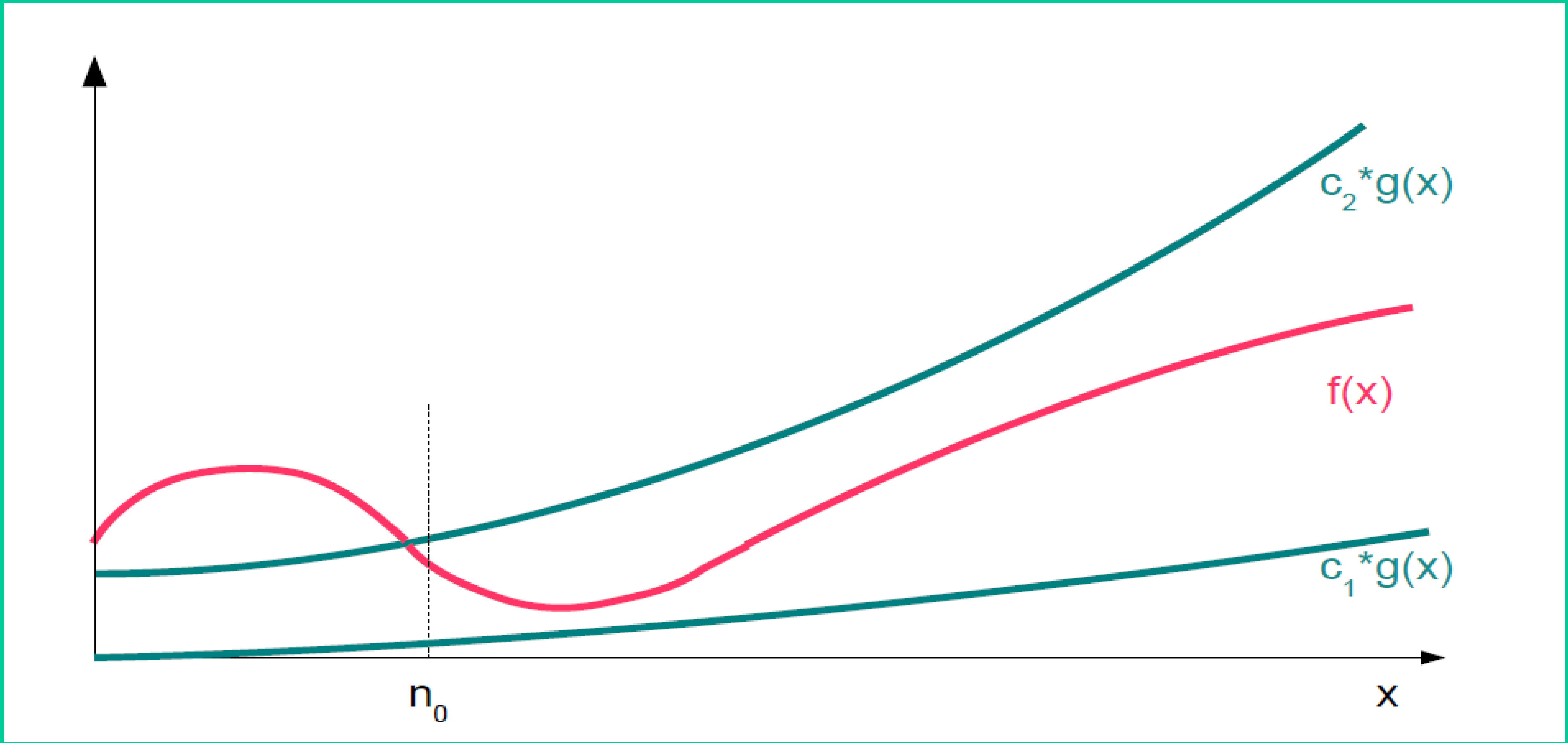
On note:

$$f(x) = O(g(x))$$



On note aussi:

$$f(x) = \Theta(g(x))$$



idée 1

Evaluer la complexité l'algorithme sur des **données de grande taille**.

Le nombre **n** désignant cette taille est '**grand**': théoriquement :

$$n \to +\infty$$

Conséquences

Par exemple, on obtient, dans un premier temps, les simplifications suivantes :

$3n^3 + 2n^2$ "se simplifie en" :

$3n^3 + 2n^2 = O(3n^3)$

$5n^2 + 2n + 4$ "se simplifie en"

$5n^2 + 2n + 4 = O(5n^2)$

$2n + 6$ "se simplifie en" :

$2n + 6 = O(2n)$

idée 2

Eliminer les **constantes multiplicatrices**.

Pourquoi ?:
en effet, deux ordinateurs de puissances distinctes diffèrent en temps d'exécution par une **constante multiplicatrice**.

Conséquences

Par exemple, on obtient, dans un deuxième temps, les simplifications suivantes :

$3n^3 + 2n^2 = O(n^3)$

$5n^2 + 2n + 4 = O(n^2)$

$2n + 6 = O(n)$

$4 = O(1)$ (car $4 = 4n^0$)

Comment comparer l'efficacité des algorithmes ?

L'idée de base est donc de pouvoir **comparer** la complexité de deux algorithmes.

Ainsi, un algorithme en $O(n^{m})$ est plus **efficace** qu'un algorithme en $O(n^{p})$ si:

$$m < p.$$

Pour l'exemple précédent, la portion d'algorithme considérée :

-prend un temps de **(4n-1)**

-pour un tableau de taille **n**

Alors, d'après la notation de **Landeau**, on peut écrire:

$$\mathbf{T(n) = 4n - 1}$$
$$\mathbf{= O(n)}$$

Comment lire la notation O ?

La notation :

$$T(n) = O(n)$$

se lit : "le coût en temps, engendré par l'algorithme **«est en»** **O**(n) ".

Sémantique de la notation "grand O"

Soit **T(n)** le temps d'exécution d'un algorithme, mesuré en fonction de la taille **n** de l'entrée.

On peut d'ores et déjà assurer les hypothèses suivantes:

$$\mathbf{n} \geq 0 \quad \text{et} \quad \mathbf{T(n)} \geq 0$$

Maintenant, soit **f** une fonction définie dans $\mathbb{N}$.

De façon **informelle**, on dira que:

$$\mathbf{T(n) = O(f(n))}$$

si **T(n)** :

-est égal à **au plus** une constante multipliée par **f**(**n**) :

$$\mathbf{T(n)} \leq \mathbf{c}\ f(\mathbf{n})$$

-à **partir de** certaines valeurs de **n** telles que : **n**> **N**

Formellement, on dit que:

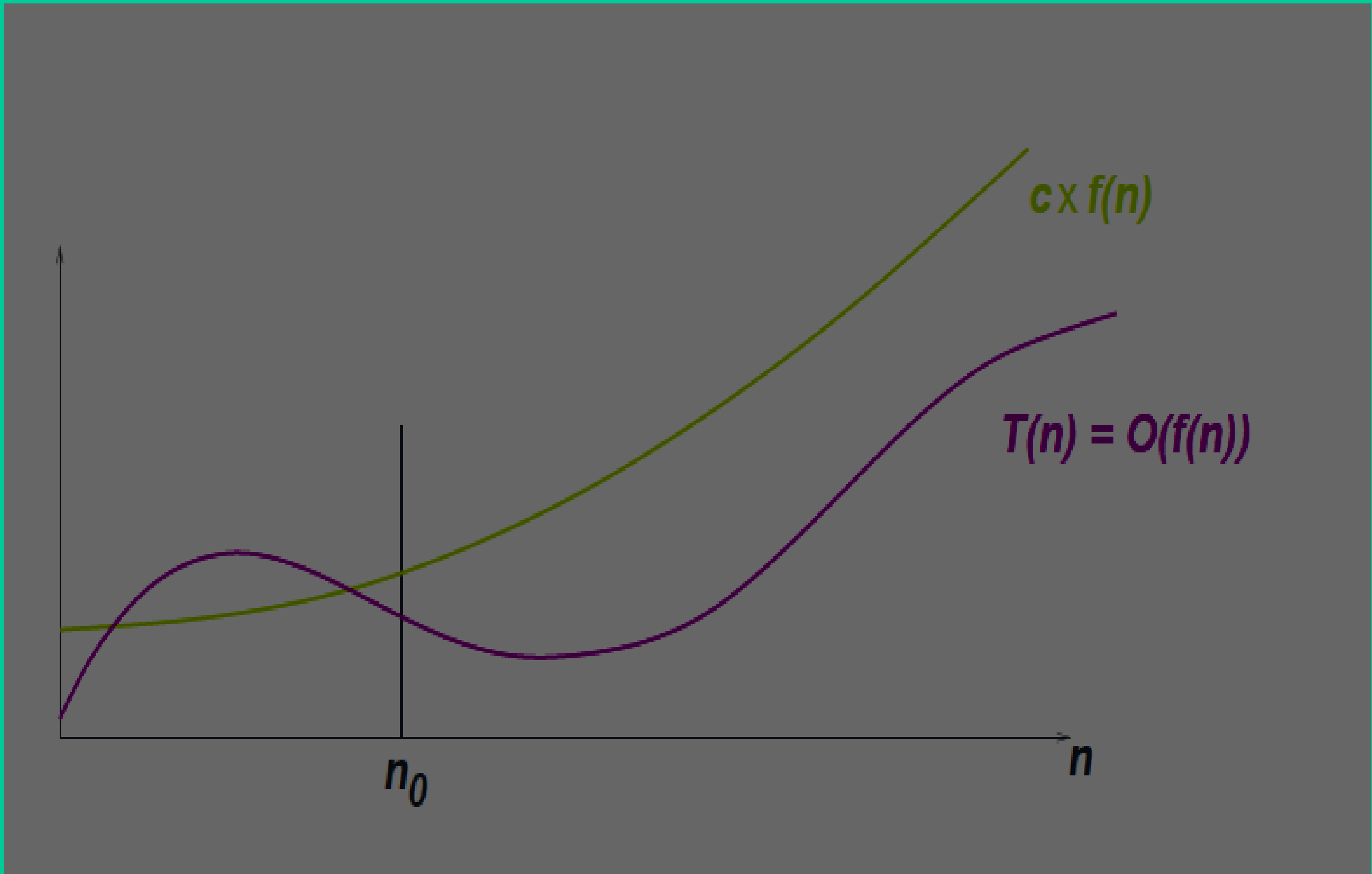
$$T(n) = O(f(n))$$

si :

$$\exists n_0 \in \mathbb{N}, \exists c \in R^{+*} \bullet$$
$$\forall n \in \mathbb{N} \bullet n \geq n_0 \Rightarrow T(n) \leq c\, f(n)$$

f(n) caractérise le comportement **asymptotique** (c'est à dire quand $n \to \infty$) de **T(n).**

On peut l'illustrer comme suit:



Comment appliquer la définition ?

Pour montrer que :

$$T(n) = O(f(n))$$

pour T et f données, il faut:

1-choisir un $\mathbf{n_0}$ et un **c**

2-et prouver que :

$$\forall\, n \in \mathbb{N} \bullet n \geq n_0 \Rightarrow T(n) \leq cf(n),$$

Exemple

Soit un algorithme dont le temps d'exécution est :

$$T(n) = (n+1)^2 + 2$$

Montrons que :

$$T(n) = O(n^2)$$

On dira alors que T(n) est **quadratique**.

1-Choisissons :

$$c = 4$$

$$\mathbf{n_0} = 5$$

dans la définition précédente.

2-On doit prouver que pour **n**≥ 5, on a:

$$(n+1)^2 + 2 \leq 4n^2$$

Comme:

$$(n+1)^2+2 = n^2 + 2n + 3.$$

Tant que $n \geq 5$, on a :

$$n \leq n^2 \quad \text{et } 3 \leq n^2.$$

Ainsi :

$$T(n) = n^2+2n+3$$
$$\leq n^2+2n^2+ n^2 = 4\,\mathbf{n^2}$$

Cependant, on ne peut choisir :

$$n_0=0$$

avec n'importe quel c.

En effet, avec n = 0, on aurait à montrer que:

$$T(0) = (0+1)^2 \leq c0^2$$

En d'autre termes l'inégalité fausse:

$$1 \leq 0$$

Cela n'est pas grave, puisque **il suffit juste de choisir :**

-un c

-et un n_0

qui fonctionnent.

Il peut sembler étrange que bien que :

$$(n+1)^2 \geq n^2,$$

on puisse toujours écrire:

$$(n+1)^2 = O(n^2).$$

En effet posons :

$$n_0 = 1 \text{ et } c = 4.$$

On doit avoir pour $n \geq 1$:

$$(n+1)^2 \leq 4\, n^2$$

En effet, on a:

$$(n+1)^2 = n^2+2n+1$$

$$n^2+2n+1 \leq n^2+2\, n^2+ n^2 = 4\, n^2$$

En fait, on peut également écrire que :

$$(n+1)^2 = O(k\, n^2\ )$$

où k ($\geq$ 0) est une fraction.

Par exemple:

$$(n+1)^2 \ = \ O(n^2/100)$$

Posons $n_0 = 1$ et $c = 400$.

On doit avoir:

$$(n+1)^2 = n^2+2n+1 \leq n^2+2\, n^2+ n^2 = 400(n^2/100)$$

Principes de base de simplification

Règle a

"Les facteurs constants ne sont pas importants"

Pour toute constante $\mathbf{p} > 0$ et pour toute fonction $\mathbf{T(n)}$:

$$\mathbf{T(n)} = O(\mathbf{pT(n)}).$$

Posons :

$\mathbf{n_0} = 0$ et $\mathbf{c} = 1/\mathbf{p}$.

Puisque $\mathbf{cp} = 1$, alors :

$$T(n) \leq \mathbf{c}\,\mathbf{p}\,T(n)$$
$$T(n) \leq \mathbf{c}\,(\mathbf{p}\,T(n))$$

Donc : $T(n) = O(pT(n))$.

Lorsque :

$$T(n) = O(f(n))$$

Alors :

$$T(n) = O(\mathbf{p}f(n))$$

que **p** soit un **grand nombre** ou une **très petite fraction**, dès lors que **p** >0.

D'après la définition, pour une constante c_1 et pour tous les $n \geq n_0$, on a :

$$T(n) \leq c_1 f(n)$$

Si on choisit :

$$c = c_1/p,$$

on voit que:

$$T(n) \leq \mathbf{cp}\ f(n) \quad \text{ou} \quad T(n) \leq \mathbf{c}\ \mathbf{(p}\ f(n))$$

Donc: $T(n) = O(\mathbf{p}f(n)$

Exemple

Comme exemple de la règle (a), on peut voir que :

$$2n^3 = O(0{,}001n^3).$$

En effet, soit $n_0=0$ et $c = 2/0{,}001 = 2000$.

Il est alors clair que :

$$2n^3 \leq 2000(0{,}001n^3) = 2n^3, \text{ pour } n \geq 0.$$

Règle b

"Les termes d'ordre inférieur sont négligeables"

On suppose que T(n) a une forme polynomiale :

$$T(n) = \mathbf{a_k} n^k + a_{k-1} n^{k-1} + \ldots + a_2 n^2 + a_1 n + a_0$$

le coefficient de tête $\mathbf{a_k}$ étant **positif**.

On a alors:

$$\mathbf{T(n) = O(n^k)}$$

On peut écarter tous les termes hormis celui avec l'exposant le plus élevé :

$$a_k n^k$$

Grâce à la règle (a), on ignore la constante correspondante $\mathbf{a_k}$ que l'on remplace par **1**.

On peut en conclure que:

$$T(n) = O(n^k).$$

Posons :

$n_0 = 1$

et c comme étant la somme de tous les coefficients positifs parmi les a_i, $0 \leq i \leq k$.

Si $a_j \leq 0$, alors $a_j n^j \leq 0$.

Si $a_j > 0$, alors $a_j n^j \leq a_j n^k$, pour $j < k$

Donc, on a: $T(n) \leq c\, n^k$

Ou : $T(n) = O(n^k)$.

Exemple

Comme exemple de la règle (b), considérons le polynôme suivant:

$$T(n) = 3n^5 + 10n^4 - 4n^3 + n + 1$$

Le terme d'ordre supérieur est n^5 et nous affirmons que:

$$T(n) = O(n^5).$$

Pour le montrer, posons :

- $\mathbf{n_0} = 1$
- et **c** égal à la somme des coefficients positifs:

$c = 3+ 10+1 +1 =15$

Pour $n \geq 1$, on a l'inégalité suivante :

$$3n^5 + 10n^4 - 4n^3 + n + 1 \leq 3n^5 + 10n^5 + n^5 + n^5$$

Comme: $3n^5 + 10n^5 + n^5 + n^5 = 15n^5$

On en conclut que :

$$T(n) = O(n^5).$$

La règle **(b)** s'applique à n'importe quelle somme d'expressions.

En effet, quand de manière générale, on a :

$$T(n) = g(n) + h(n)$$

Avec :

$$(h(n)/g(n)) \rightarrow 0 \text{ quand } n \rightarrow +\infty$$

On dit que h(n) **"croît moins vite"** que g(n).

Ou g(n) domine h(n)

Alors $h(n)$ peut être **"**négligée**"** devant $g(n)$

Ce qui donne:

$$T(n) = O(g(n)).$$

Exemple

Soit:

$$T(n) = 2^n + n^3$$

Il est notoire que :

-tout polynôme: ici n^3

-**croît moins vite** que toute exponentielle: ici 2^n.

On peut donc négliger n^3 et conclure que :

$$T(n) = O(2^n).$$

Pour le montrer formellement, posons :

$$n_0 = 10 \text{ et } c = 2$$

ce qui nous amène à prouver que pour $n \geq 10$, on a :

$$2^n + n^3 \leq 2 \times 2^n$$

En soustrayant 2^n de chaque côté, on voit qu'il suffit de montrer que:

-pour $n \geq 10$,

-on a $n^3 \leq 2^n$.

Pour n=10, on a:

$$2^n = 2^{10} = 1024 \text{ et } n^3 = 10^3 = 1000$$

d'où :

$$n^3 \leq 2^n \text{ pour } n = 10.$$

Chaque fois qu'on ajoute 1 à n :

- 2^n **double**,
- tandis que n^3 est **multiplié par $(n+1)^3/n^3$** qui est inférieure à 2 quand $n \geq 10$.

Ainsi, quand **n** dépasse 10, n^3 devient progressivement inférieur à 2^n.

On en conclut que :

$$n^3 \leq 2^n \text{ pour tout } n \geq 10$$

et donc que:

$$2^n+n^3 = O(2^n).$$

Calcul des expressions en O

En appliquant systématiquement les règles (**a)** et (**b)**, on peut **simplifier sensiblement** l'expression du temps d'exécution **T(n).**

On va voir combien il est important de réaliser de telles simplifications lorsqu'on **analyse des algorithmes**.

Transitivité

Une proprpriété très utile pour la simplification est la **transitivité** de la relation :

«est en grand O de» .

Cela signifie que :

si $f(n) = O(g(n))$

et $g(n) = O(h(n))$

alors $f(n) = O(h(n))$.

Exemple

$$T(n) = 3n^5+10n^4 - 4n^3+n+1$$

$$T(n) = O(n^5).$$

D'après la règle (a), on a :

$$n^5 = O(0{,}01n^5).$$

Grâce à la **loi de transitivité** pour les expressions en O, on a:

$$T(n) = \mathbf{O}(0{,}01n^5).$$

Notation grand O dans les expressions

Du point de vue strictement mathématique, la seule façon correcte d'utiliser une expression en grand O est de la placer après les mots « est en », comme dans:

« $2n^2$ est en $O(n^2)$ ».

Et noter :

« $2n^2 = O(n^2)$ ».

Cependant, on prend aussi la liberté d'utiliser les expressions en O comme des opérandes d'opérateurs arithmétiques :

Comme dans l'expression:

$$O(n)+O(n^2).$$

Une expression en **O** utilisée de cette manière, doit être interprétée comme une **fonction**.

Par exemple, $O(n) + O(n^2)$ doit être interprétée comme la somme :

- d'une fonction linéaire,
- et d'une fonction quadratique.

Règle de sommation

Il existe une technique générale qui permet de combiner deux expressions en grand O.

Supposons qu'un algorithme soit constitué de deux parties :

-l'une a un coût en $O(n^2)$

-et l'autre un coût en $O(n^3)$.

On peut les **additionner** pour obtenir le coût d'exécution de la totalité de l'algorithme.

On utilise la **règle de sommation** qui suit.

Supposons que :

- $T_1(n) = O(f_1(n))$
- $T_2(n) = O(f_2(n))$.

Par ailleurs, supposons que :

- la croissance de f_2
- ne soit pas plus rapide que celle de f_1

c'est-à-dire:

$$f_2(n) = O(f_1(n)).$$

On peut alors conclure que:

$$T_1(n) + T_2(n) = O(f_1(n)).$$

Exemple

Considérons l'algorithme qui transforme une matrice a [1..n,1..n] en une matrice identité.

(1) lire (n)
(2) **pour** i := 1 **à** n **faire**
(3)   **pour** j := 1 **à** n **faire**
(4)   a [i, j] := 0
    finpour
    finpour
(5) **pour** i := 1 **à** n **faire**
(6)   a [i, i] := 1
    Finpour

Les lignes (2) à (4) placent la valeur 0 dans toutes les éléments de a.

Les lignes (5) et (6) placent des 1 dans toutes les positions diagonales entre **a**[1, 1] et **a**[n, n].

Le résultat est une matrice identité **a** avec la propriété :

$$a \times m = m \times a$$

pour toute matrice m[1..n, 1..n].

L’action (1) qui lit **n** prend un coût en O(1) : une quantité de temps constante, indépendante de n.

L’action d’affectation (6) prend aussi un temps en O(1)

Comme on emprunte les lignes (5) et (6) exactement **n** fois, on aboutit à un coût total en O(n) pour cette boucle.

De même, l’affectation à la ligne (4) a un coût en O(1).

On emprunte **n** fois les lignes (3) et (4), pour un coût total en O(n).

On emprunte **n** fois la boucle externe aux lignes (2) à (4), avec un coût en O(n) à chaque itération, pour un coût total en O(n^2).

Ainsi, le coût en temps de l'algorithme est en:

$$O(1)+O(n^2)+O(n),$$

De manière plus formelle :

- $T_1(n) = O(1)$,
- $T_2(n) = O(n^2)$,
- $T_3(n) = O(n)$.

Il faut donc trouver une **borne supérieure** pour:

$$T_1(n) + T_2(n) + T_3(n)$$

afin de déduire le temps consommé par l'algorithme.

Comme la constante **1** est en $O(n^2)$, on peut appliquer la règle de sommation pour conclure que :

$$(T_1(n) + T_2(n)) = O(n^2).$$

Puis, comme **n** est aussi en O(**n**^{2}), on peut appliquer la règle de sommation à :

$$(T_1(n) + T_2(n))$$

et $$T_3(n)$$

pour conclure que:

$$T_1(n) + T_2(n) + T_3(n) = O(n^2).$$

Autrement dit, l'algorithme a un coût en O(n^2).

III-Calcul de la complexité

Nous allons apprendre à calculer pour les **formes algorithmes de base**.

-leur coût en temps **T(n)**

-et surtout, à déduire une **borne** O pour ce coût.

On s'efforcera de rechercher des **bornes** :

- **simples**

-et **approchées**.

On considérera les principales **formes algorithmiques** utilisées dans la plupart des langages d'implémentation, à savoir :

-les **séquences** d'instructions simples,
-les **blocs** d'instructions,
-la boucle **<u>pour</u>**,
-la répétitive **<u>tant que</u>** ou **<u>répéter</u>**.

Notion d' «ordre de grandeur»

Soit les fonctions f et g.

O,n dit que f est **dominée** par g et on note :

$$f = O(g)$$

lorsque l'on a:

$$\exists n_0, \exists c > 0, \quad \forall n \geq n_0, \left| f(n) \right| \leq cg(n)$$

On dit f est **du même ordre de grandeur** que g et l'on note :

$$f = \Theta(g)$$

lorsque l’on a:

$$f=O(g) \quad \text{et} \quad g=O(f).$$

Notion d’équivalence

On dit que f est **négligeable** devant g et on note:

$f = o(g)$ «f est en petit o»

lorsque:

$f(n)/g(n) \rightarrow 0$ quand $n \rightarrow +\propto$

On dit que f est **équivalente** à g lorsque :

$f(n)/g(n) \rightarrow 1$ quand $n \rightarrow +\propto$

Propriétés utiles de O

1-**Réflexivité**:

$$f = O(f)$$

2-**Transitivité**:

$$\text{si } f = O(g) \text{ et } g = O(h) \quad \Rightarrow f = O(h)$$

3-Produit par un scalaire positif:

$$p > 0 \;\Rightarrow O(p.f) = O(f)$$

4-Somme de fonctions:

$$O(f) + O(g)=O(\max\{f,g\})$$

5-Produit de fonctions:

$$O(f) . O(g)=O(f.g)$$

1-Complexité d'une instruction simple

Le premier principe dont nous avons besoin est que :

- -une **affectation**,
- -une **lecture** / **écriture**
- -une **comparaison** avec des expressions simples

a un coût en temps en O(1).

O(1) signifie que ce coût est une quantité de temps constante.

Exemple

Soit l'algorithme de tri par sélection:

(1) **pour** i := 1 **à** n-1 **faire**
(2) petit := i
(3) **pour** j := i+1 **à** n **faire**
(4) **si** a[j] < a[petit]
(5) **alors** petit := j
finsi
finpour
(6) tempo := a[petit]
(7) a[petit] := a[i]
(8) a[i] := tempo
finpour

Les affectations des lignes 2, 5, 6, 7 et 8 engendrent toutes un coût en **O(1).**

```
(1) pour i := 1 à n-1 faire
(2)      petit := i
(3)      pour j := i+1 à n faire
(4)          si a[j] < a[petit]
(5)              alors petit := j
             finsi
         finpour
(6)      tempo := a[petit]
(7)      a[petit] := a[i]
(8)      a[i] := temp
    finpour
```

2-Coût d’une séquence d’instructions

Une séquence instructions dont le coût d’exécution de chacune est en O(1), engendre un coût en O(1).

Selon la **règle de sommation**, la somme d’un nombre quelconque constant de O(1) donne O(1).

$$O(1) + O(1) + \ldots + O(1) = O(1)$$

Exemple

Soit la séquence T_{68} formée des lignes 6 à 8 ci-dessous.

```
(1)  pour i := 1 à n-1 faire
(2)        petit := i
(3)        pour j := i+1 à n faire
(4)              si a[j] < a[petit]
(5)                    alors petit := j
           finsi
       finpour
(6)        tempo := a[petit]          O(1)
(7)        a[petit] := a[i]           O(1)
(8)        a[i] := temp               O(1)
   finpour
```

Chacune des instructions a un coût en O(1).

Alors:

$$T_{68} = O(1) + O(1) + O(1) = O(1)$$

On n’a pas inclus la ligne (5) dans le bloc, car elle fait partie de l’instruction **<u>si</u>** de la ligne (4).

En effet, les lignes (6) à (8) sont parfois exécutées sans que la ligne (5) le soit.

3-Coût d'une boucle pour

1-Cas simple:

pour i $\leftarrow$1 **à** n **faire**	n = nombre répétitions
s	Ts = O(1)
finpour	

Ts=O(1).

T(n) = n.Ts

= n.O(1)

= O(n)

1-Cas général:

Pour calculer le coût d'une boucle pour, il faut obtenir :

-une **borne supérieure** pour le coût

-engendré par l'exécution **d'une seule itération.**

Ce coût en temps d'une boucle **pour** s'analyse en:

-coût d'**incrémentation** de l'indice: i++,

-coût de **comparaison** de l'indice avec la borne supérieure : i> n

-coût engendré par le **corps** du **pour** : S

Le coût de test et d'incrémentation est en O(1).

Hormis le cas où le corps de la boucle **est vide**, ces O(1) peuvent être éliminés par la règle de sommation.

Dans le cas où le coût engendré par le corps de la boucle est **le même** à chaque itération, on peut multiplier :

- la **borne supérieure** grand O du corps,
- par le **nombre d'itérations** exécutées.

A ce coût, il faut additionner ensuite :

-le coût, en O(1), servant à **initialiser** l'indice de la boucle : i :=1

-et celui, en O(1), de la **première comparaison** de l'indice avec la sa borne supérieure : i > n

A moins d'exécuter la boucle zéro fois, chacun de ces deux coûts est un terme:

-d'ordre inférieur

-qu'on peut éliminer par la règle de sommation.

Exemple

Soit la boucle **pour** des lignes 3 et 4 du code suivant :

```
(2) pour i := 1 à n faire
(3)       pour j := 1 à n faire
(4)             a[i, j] := 0
finpour
finpour
```

Le corps de la boucle pour (ligne 4) a un coût en O(1).
La boucle sera exécutée **n** fois

Comme le corps du **<u>pour</u>** est en O(1), on peut négliger:

-le coût de l'incrémentation de j,

-et celui de la comparaison de j avec n,

car tous deux également en O(1).

Ainsi, le coût d'exécution des lignes 3 et 4 est:

$$T_{34} = n \times O(1) = O(n).$$

De la même manière, on peut borner le coût d'exécution du **pour** extérieure T_{24} (indice i): lignes 2 à 4.

```
(2) pour i := 1 à n faire
(3)     pour j := 1 à n faire
(4)         a[i, j] := 0
    finpour
finpour
```

On a calculé le coût de la boucle intérieure T_{34} (indice j):

$$T_{34} = O(n).$$

Pour le <u>**pour**</u> extérieure, on peut négliger le coût en O(1) engendré, à chaque itération, par:

- l'incrémentation de **i**
- le test de **i > n**.

Ce qui amène à un coût en O(n).

L'initialisation **i := 1** et les (n+1) tests de la condition **i > n** ont un temps en O(1) et peuvent être négligés.

Enfin, on remarque que :

-on emprunte la boucle extérieure **n** fois,

-avec à chaque itération un coût en O(n).

Ce qui donne un coût total en:

$$T_{\mathbf{24}} = n.\ O(n) = O(n^2).$$

Exemple

Considérons la boucle **pour** des lignes 3 à 5:

```
(1)  pour i := 1 à n-1 faire
(2)      petit := i
(3)      pour j := i+1 à n faire
(4)          si a[j] < a[petit]
(5)              alors petit := j
         finsi
     finpour
(6)      tempo := a[petit]
(7)      a[petit] := a[i]
(8)      a[i] := tempo
   finpour
```

Le corps est une instruction **<u>si</u>**.

On constate que :

-la ligne (4) effectue un test : coût en O(1),

-la ligne (5), quand elle est exécutée, a un coût en O(1).

Donc l’exécution du corps de la boucle se fait en O(1), que la ligne (5) soit exécutée ou non.

L’incrémentation et le test de la boucle ajoutent un coût en O(1).

Le temps total d’une seule **itération** se résume donc à O(1).

Il faut maintenant calculer le nombre d’itérations de la boucle.

Le nombre d’itérations est calculé par la formule suivante :

« limite supérieure - limite inférieure + 1 »

Ce qui donne :

$$n – (i+1) + 1 = n- i$$

comme nombre d’itérations.

En toute rigueur, la formule précédente ne s’applique que pour $i \leq n$.

Par ailleurs, on peut voir à la ligne 1:

(1) **pour** i := 1 **à** n-1 **faire**

que $i < n-1$ ou $n-i > 1$

Donc (n-i) ne peut pas être nul.

Le temps passé dans la boucle est donc:

$$(n-i) \times O(1) = O(n-i)$$

On n'a pas à ajouter O(1) pour l'initialisation de j, puisqu'on a établi que (n-i) ne peut être nul.

Si on n'avait pas vu que $(n-i)$ était positif, on aurait dû écrire que le coût est en :

$$O(\max(1, n-i)).$$

Donc le coût est tout simplement en $O(n)$.

4-Les instructions conditionnelles

On rappelle qu'instruction conditionnelle **si alors sinon** est constituée de trois parties :

si **condition**
  alors **partie_alors**
  sinon **partie_sinon**
  finsi

A moins de comporter des **appels** de fonctions, la **condition** s’exécute avec un coût en O(1).

En effet, elle ne peut nécessiter qu’un nombre constant :

- d’opérations arithmétiques,
- d’accès aux données,
- de comparaisons de valeurs
- et d’opérations logiques

Supposons que :

-la **partie_alors** engendre un coût en O(f(n))

- la **partie_sinon**, un coût en O(g(n)).

Supposons aussi que :

-si la **partie_sinon** peut faire défaut c'est-à-dire g(n) = 0,

-la **partie_alors** n'est **pas un bloc vide**.

<u>Cas 1</u> :

$$f(n) = O(g(n))$$

Alors on peut prendre:

$$T(n) = O(g(n))$$

comme coût global d'exécution de la conditionnelle.

En effet :

– on peut négliger le $O(1)$ de la condition,

– si la **partie-sinon** est exécutée, on sait déjà que son coût d'exécution est en $O(g(n))$

– si la **partie-alors** est exécutée, le coût d'exécution sera toujours en $O(g(n))$ car :

$$f(n) = O(g(n)).$$

Cas 2 :

$$g(n) = O(f(n))$$

Alors on peut prendre:

$$T(n) = O(f(n))$$

comme coût global d'exécution de la conditionnelle.

On remarquera que quand la **partie-sinon** fait défaut, comme c'est souvent le cas :

$$g(n) = 0$$

son coût est à coup sûr en $O(f(n))$.

Cas 3 :

C’est le cas délicat est celui où :

- ni f
- ni g

ne sont grand O l’une de l’autre.

On sait que une borne supérieure sûre pour le coût d’exécution sera plus grande que f(n) et g(n).

Nous devons donc écrire:

$$T(n) = O(\max(f(n), g(n))).$$

le coût d'exécution de la conditionnelle.

Exemple

```
(1)si a[1, 1] = 0
(2)     alors  pour i := 1 à n faire
(3)                 pour j := 1 à n faire
(4)                 a[i, j] := 0
                    finpour
               finpour
(5)     sinon pour i := 1 à n faire
(6)            a[i, i] := 1
               finpour
```

Le coût d'exécution T_{24} (lignes 2 à 4) est :

$$T_{24} = O(n^2).$$

(2) **alors** **pour** i := 1 **à** n **faire**
(3) **pour** j := 1 **à** n **faire**
(4) a[i, j] := 0
finpour

Celui de T_{56} (lignes 5 et 6) :

$$T_{56} = O(n).$$

(5) **sinon pour** i := 1 **à** n **faire**
(6) a[i, i] := 1
finpour

Donc, ici :

$$f(n) = n^2 \quad \text{et} \quad g(n) = n$$

Alors :

$$T(n) = O(n^2)$$

C’est le résultat de l’hypothèse la pire, à savoir que :

- la condition (1) est vraie
- la **partie_alors** sera exécutée.

5. Les blocs

Une séquence **d'instructions simples**:

- affectation,
- lecture/écriture
- test…

génère **toujours** un coût en **O(1).**

Mais on peut aussi avoir à faire à des séquences d'instructions comportant des **instructions complexes**:

Il peut s'agir :

- d'**instructions conditionnelles**
- ou des **boucles**.

Une telle **séquence** d'instructions simples et complexes est appelée un **bloc**.

Soit le bloc B:

Début

I_1 ; $T_1 = O(f_1)$

I_2 ; $T_2 = O(f_2)$

… …

I_p ; $T_p = O(f_p)$

Fin

Son coût T_B est:

$$T_B = T_1 + T_2 + \ldots + T_p$$

1-Le coût en temps d'un **bloc** est calculé en prenant :

-la **somme** des coûts en grand O

-de **chacune** de ses instructions.

$$T_B = O(f_1) + O(f_2) + \ldots + O(f_p)$$

Ensuite, on utiliser la **règle de sommation** :

$$T_B = O(\max\{ f_1, f_2,\ldots,f_p \})$$

pour éliminer tous les termes de la somme…. sauf un.

Exemple

Soit l’algorithme:

```
(1)  pour i := 1 à n-1 faire
(2)        petit := i
(3)        pour j := i+1 à n faire
(4)              si a[j] < a[petit]
(5)                    alors petit := j
             finsi
         finpour
(6)        tempo := a[petit]
(7)        a[petit] := a[i]
(8)        a[i] := tempo
     finpour
```

On peut voir le corps de la boucle extérieure T_{28} : lignes 2 à 8.

```
(2)     petit := i
(3)     pour j := i+1 à n faire
(4)         si a[j] < a[petit]
(5)             alors petit := j
          finsi
        finpour
(6)     tempo := a[petit]
(7)     a[petit] := a[i]
(8)     a[i] := tempo
```

C'est un bloc qui est composé de cinq instructions :

- l'affectation de la ligne (2),
- la boucle **pour** des lignes (3), (4) et (5),
- l'affectation de la ligne (6),
- l'affectation de la ligne (7),
- l'affectation de la ligne (8).

On peut noter que les lignes 4 et 5 ne sont **pas visibles** au niveau de ce bloc.

Elles sont cachées à l'intérieur d'une instruction plus grande: la boucle **pour** des lignes (3) à (5).

```
(3)    pour j := i+1 à n faire
(4)        si a[j] < a[petit]
(5)            alors petit := j
```

On sait que les quatre instructions d'affectation prennent chacune un coût en O(1).

Le coût d'exécution de la boucle T_{35} des lignes 3 à 5 est :

$$T_{35} = O(n\text{-}i).$$

Le coût d'exécution du bloc est donc :

$$T(n) = O(1) + O(n\text{-}i) + O(1) + O(1) + O(1)$$

Comme:

$$1 \leq n\text{-}i$$

on peut éliminer tous les O(1) en appliquant la règle de sommation.

Le bloc tout entier a donc un coût total :

$$T(n) = O(n\text{-}i).$$

Notons que **i** est l'indice de la boucle <u>**pour**</u> extérieure: il **varie** donc à l'intérieur de cette boucle.

Donc O(n-i) ne peut être considéré comme le coût d'exécution de toutes les itérations de cette boucle.

Par ailleurs, on peut voir dans la ligne 1 que $i \geq 1$

(1) **pour** i := 1 **à** n-1 **faire**

$$i \geq 1 \quad \Rightarrow \quad n-i \leq \; n-1 \; \Rightarrow n-i = O(n-1)$$

D’après la loi de **transitivité**, O(n-1) est aussi le coût de chaque itération de la boucle extérieure.

Mieux, grâce à la règle (b) on peut simplifier O(n-1) en O(n) .

Déterminons maintenant le nombre de fois que l’on emprunte la boucle extérieure.

Comme i varie de 1 à n-1, on emprunte cette boucle **(n-1)** fois.

En appliquant la propriété du produit par une constante positive:

$$T(n) = (n-1).\ O(n) = O((n-1).n)$$
$$= O(n^2-n).$$

En appliquant une fois encore la règle (b) on obtient pour l'algorithme de tri par sélection le coût total T(n) :

$$T(n) = O(n^2)$$

On peut donc dire que le coût d'exécution du le tri par sélection est **quadratique**.

Exemple

Soit la portion d'algorithme suivante :

h ← 1	c1 = O(1)
tant que h ≤ n **faire**	c2 = O(1)
h ← 2*h	c3 = O(1)
fintantque	

Nombre d'itérations : $\log_2(n)$.

$$T(n) = \log2(n).\ O(1)$$
$$= O(\log_2 (n))$$

5. Les boucles tant que et répéter

L’analyse d’une boucle **tant que** ou **répéter** ressemble à celle d’une boucle **pour**.

Cependant, le nombre de parcours d’une boucle **tant que** ou **répéter** n’est pas connu a priori.

De ce fait une partie de l'analyse doit être consacrée à déterminer :

-**une borne supérieure**

-pour le nombre d'itérations de la boucle.

Ensuite, il faut déterminer le coût pour le temps d'exécution d'une **seule itération**.

Pour ce faire, on examine le **corps** de la boucle pour déterminer son coût d'exécution.

On ajoute un temps en O(1) pour prendre en compte le test de la condition après l'exécution du corps.

Mais à moins que le corps ne fasse défaut, on peut négliger ce terme.

On obtient alors le coût d'exécution de la boucle en multipliant :

-une **borne supérieure** pour le nombre d'itérations

-par le coût d'une seule itération.

Pour la boucle <u>**tant que**</u>, il faut ajouter le coût nécessaire au test de la condition la première fois, avant d'entrer dans le corps.

Mais ce terme en O(1) peut normalement être négligé.

Exemple

Considérons le fragment d'algorithme recherche linéaire:

(1) i := 1
(2) **<u>tant que</u>** x $\neq$ a[i] **<u>faire</u>**
(3) i := i+1
<u>fintantque</u>

Les deux instructions d'affectation 1 et 3 ont un coût en O(1).

```
(1) i := 1
(2) tant que x ≠ a[i] faire
(3) i := i+1
      fintantque
```

La boucle **tant que** peut être exécutée jusqu'à **n** fois au maximum, car on suppose que l'un des éléments du tableau est x.

Comme le corps de la boucle est en $O(1)$, le coût du **tant que** est en $O(n)$.

D'après la règle de sommation, le temps total $T(n)$ du fragment est donc:

$$
\begin{aligned}
T(n) &= O(1) + O(n) \\
&= O(n)
\end{aligned}
$$

6-Complexité d'un algorithme récursif

Cas général

fonction FunctionRecursive (n)	T(n)
si (n > 1) **alors**	O(1)
FunctionRecursive(n/2) ;	T(n/2)
Traitement(n) ;	C(n)
FunctionRecursive(n/2)	T(n/2)

$$T(n) = 2 * T(n/2) + C(n)$$

Si $C(n) = 1$ alors

$$T(n) = k \times n$$
$$T(n) = O(n)$$

Si $C(n) = n$ alors

$$T(n) = k \times n \times \log_2 n$$
$$T(n) = O(n \times \log_2 n)$$

Cas simple

```
fonction fact (n: Naturel) : Naturel          T(n)
début
    si n = 0 alors                            C1 = O(1)
            retourner 1
            sinon
            retourner n * fact (n-1)          C2+ T(n-1)
    finsi
fin
```

Soit $T(n)$ le coût engendré par fact(n)

$$T(0) = C1$$
$$= O(1)$$

Pour $n>0$, on a :

$$T(n) = C1 + C2 + T(n-1)$$
$$\Rightarrow$$
$$T(n) = nC2 + (n + 1)C1 = n(C1 + C2) + C1$$

En posant $C = 2.\max\{C1, C2\}$, on a:

$$T(n) \leq C.n + C1$$

Finalement:

$$T(n) = O(n)$$

Cas du «tri par fusion»:

fonction **TriParFusion**(S, n)

  si (n ≤ 1) **alors** — **O(1)**

    renvoyer S

  décomposer S en S1 et S2, — **n**

  S1 :=TriParFusion(S1), — **$T(\lceil n/2 \rceil)$**

  S2 :=TriParFusion(S2), — **$T(\lfloor n/2 \rfloor)$**

  S :=**fusion**(S1, S2), — **n**

  renvoyer S

Si n=1 alors :

$$T(1) = 1$$

Si n>1 alors :

$$T(n) = 1 + n + 2 \times T(n/2) + n$$

$$T(n) = 2\,n + 1 + 2\,T(n/2)$$

$$T(n) = \ 2\mathbf{n\ log\ n} + 2n - 1$$

$$T(n) = \ 2\mathbf{n\ log\ n} + 2n - 1$$

$$T(n) = \ = O(\mathbf{n\ log\ n})$$