# UE d'Ouverture Rappels Algorithmiques

## Table des matières

**1 Introduction** **3**
1.1 Au programme . . . 3
1.2 Convention d'écriture et instructions du pseudo-code. . . . 4
1.3 Exemple d'analyse d'algorithme : Le Tri-Insertion . . . 5

**2 Diviser pour Régner** **7**
2.1 Stratégie et complexité de Diviser pour Régner . . . 7
2.2 Comportement asymptotique : notations de Landau . . . 7
2.3 Le Tri-Rapide . . . 8
2.4 Le théorème maître . . . 11
2.5 Applications arithmétiques . . . 12
2.5.1 Multiplication de deux entiers par la méthode de Karatsuba . . . 12
2.5.2 Division récursive . . . 14

**3 Programmation Dynamique** **17**
3.1 Plus Longue Sous-séquence Commune . . . 20
3.1.1 Propriété d'une plus longue sous-séquence commune . . . 21
3.1.2 Une solution récursive . . . 22
3.1.3 Calcul de la longueur d'une PLSC . . . 22
3.1.4 Construction d'une PLSC . . . 23
3.2 Le problème du sac à dos . . . 25
3.2.1 Approche directe . . . 25
3.2.2 Sous-Structure optimale . . . 26
3.2.3 Une récurrence pour trouver la valeur optimale que l'on peut transporter. . . . 26
3.2.4 Pseudo-code pour la valeur optimale . . . 26
3.2.5 Construction d'une solution optimale . . . 28

Ce cours est une initiation à quelques grands principes algorithmiques (Programmation Récursive et Programmation Dynamique) par leur mise en application dans des situations variées (numérique, graphique, géométrique,...). Cette première séance introductive est consacrée à quelques conventions et rappels (pseudo-code, comportement asymptotique), nous introduirons aussi la notion d'algorithme Diviser pour Régner, avec des applications dans le cas du problème du tri (Quicksort), de la multiplication d'entiers (Karatsuba) et de la division rapide. Afin d'analyser ces algorithmes, nous donnerons le "théorème maître" qui permet d'obtenir des bornes asymptotiques sur la plupart des fonctions rencontrées lors de l'analyse de coût des algorithmes Diviser pour Régner.

# 1 Introduction

De tout temps les hommes ont cherché des méthodes pour "systématiser" leurs raisonnements. La division ou l'algorithme d'Euclide pour trouver le PCGD de deux nombres remontent à 400 ans avant notre ère. "Systématiser", c'est assurer la reproductibilité (et donc l'apprentissage) des méthodes, c'est aussi assurer leur transmissibilité dans le temps et dans l'espace sous une forme compréhensible par tous. Afin de transmettre, il faut garantir que les méthodes proposées donnent bien ce à quoi l'on aspire (sinon on parle d'*heuristique*), à la fois dans le but de convaincre l'autre, mais aussi dans le souci constructiviste de garantir la cohérence de l'édifice des savoirs (par la certification de chacune de ses briques). Plus récemment, avec l'avènement de l'informatique, on accorda un soin tout particulier à la description formelle des algorithmes afin d'en faciliter l'exportation vers les différents langages informatiques modernes. C'est le but de l'écriture algorithmique en pseudo-code qui fera l'objet du premier préambule. Le dernier point, et ce n'est pas le moindre quand l'on pense algorithmique, consiste à trouver les méthodes les plus économiques possibles que ce soit en termes de temps, de mémoire ou de tout autre chose. Voilà donc ce que l'on entend ici par algorithmique et c'est ce que nous pouvons résumer ainsi :
l'*algorithmique* (terme qui vient du surnom Al-Khuwarizmi d'un célèbre mathématicien perse (780-850) à qui l'on doit aussi le mot algèbre) est la science qui consiste à trouver des méthodes formelles et certifiées pour résoudre de manière automatique et le plus efficacement possible des problèmes donnés.

Voilà un premier exemple applicatif : le problème du tri de listes. Nous allons détailler dans le chapitre 2 deux méthodes de tris. Le Tri-Insertion fonctionne comme le tri classique d'une main de cartes : dans ce qu'on a commencé a trié, on insère la nouvelle carte au bon endroit. Le Tri-Rapide semble plus compliqué, et nous ne retiendrons ici que les quelques données suivantes. Une première manière d'analyser un algorithme (et dans la grande majorité des cas la plus facile) c'est de regarder le pire des cas (pour le Tri-Insertion, liste triée à l'envers) Pour le pire des cas : environ 5000 comparaisons pour trier une liste de 100 entiers (pour les deux algorithmes). Pourtant le tri utilisé (quasi à chaque fois) est le Tri-Rapide (Java, Unix, ...). Pour quelle raison ? Une étude plus fine, plus difficile bien souvent, est l'analyse en moyenne : si on prend une liste aléatoire d'entiers et qu'on la classe, combien fait-on de comparaisons ? Sur 50 listes aléatoires de 100 valeurs, le Tri-Insertion a effectué environ 2452 comparaisons en moyenne et le Tri-Rapide : 642. Donc le pire des cas n'est pas toujours significatif ! !

## 1.1 Au programme

Dans ce cours, nous nous proposons, plus qu'à l'étude ou à la recherche d'algorithme particulier, de mettre en lumière deux "types" universels d'algorithmes que sont : les algorithmes de type **Diviser pour Régner**, les algorithmes de type **Programmation Dynamique**. Notre objectif étant de montrer qu'il existe au dessus du monde singulier des algorithmes, de grands principes algorithmiques communs. Pour insister sur le caractère transversal de ces principes, pour chacun d'eux nous proposerons des applications qui couvrent un champs large de l'algorithmique classique, disons pour rester catégoriel dans ses cinq ramifications suivantes : l'algorithmique numérique et matriciel,

la géométrie algorithmique, l'algorithmique des graphes, l'ordonnancement et les tris, l'algorithmique des séquences. Ce cours s'architecturera autour de cette idée principale.

Voilà quelques ouvrages utiles pour ce cours :
— *Types de données et algorithmes* : Froidevaux, Gaudel et Soria
— *Eléments d'algorithmiques* : Beauquier, Berstel et Chrétienne (téléchargeable sur la page de Berstel)
— *Introduction à l'algorithmique* : Cormen, Leiserson, Rivest, Stein

### 1.2 Convention d'écriture et instructions du pseudo-code.

Nous écrirons dans la mesure du possible les pseudo-codes sous la forme suivante :

**Algorithme 1 :** Paradigme d'écriture des pseudo-codes

**Entrées :** deux entiers $a$ et $b$.
**Sorties :** un entier.
1 ADDITION($a$, $b$)
2 $s := a + b$
3 **Retourner** $s$

Les pseudo-codes que nous écrirons auront donc un trait impératif dans le sens où nous disposerons pour les construire de l'*affectation* que l'on notera :=. Par souci de simplicité, nous utiliserons l'opérateur $\leftrightarrow$ afin d'échanger les valeurs de deux variables (on n'oublie pas que l'on a besoin d'une variable supplémentaire afin de faire cet échange). Nous utiliserons les instructions suivantes qui seront toujours écrites en caractère gras.

— La boucle avec compteur :
**pour** `variable` **allant de** `entier1` **à** `entier2`
| `instructions`
— La boucle à arrêt conditionnel :
**tant que** `condition`
| `instructions`
— L'instruction conditionnelle :
**si** `condition` **alors**
| `instructions`
**sinon**
| `instructions`
— Le retour (qui renvoie la sortie de l'algorithme) :
**retourner** `expression`
— L'arrêt forcé :
**arrêt** `commentaire`

### 1.3 Exemple d'analyse d'algorithme : Le Tri-Insertion

Nous nous intéressons ici au problème du tri d'une séquence de nombres pour montrer en quoi les méthodes employées peuvent jouer un rôle important sur les temps d'exécution. Ce sera aussi l'occasion pour nous d'introduire la notion d'algorithme de type Diviser pour Régner.

Un premier algorithme "naïf" (qui n'est pas de type diviser pour régner) appelé **Tri-Insertion** peut être décrit récursivement comme suit : L'*entrée* de l'algorithme est une séquence de nombres $L = (a_1, ..., a_n)$. La *sortie* sera une séquence constituée des mêmes nombres mais triés dans l'ordre croissant. On commence avec la séquence de départ $L'_1 = L$, supposons par récurrence que $L'_k$ soit la séquence dont les $k$ premiers éléments ont été triés dans l'ordre croissant. On obtient $L'_{k+1}$ en insérant $a_{k+1}$ à la bonne place parmi les $k$ premiers éléments de $L'_k$. On itère ce procédé, jusqu'à l'obtention de $L'_n$ qui n'est autre que la séquence triée demandée.

Pour la représentation en pseudo-code de la séquence de nombres, on utilisera un tableau, ce qui permet un accès direct aux données, mais aussi d'effectuer des opérations sur le tableau lui-même sans copie. On dit alors que le Tri-Insertion est un algorithme de tri *sur place*. Nous obtenons les trois algorithmes suivants :

On travaille sur un tableau (variable globale) $T[1..n]$ de $n$ nombres, et contenant une sentinelle : $T[0] = -\infty$.

**Algorithme 2 :** Procédure d'insertion d'un élément dans le tableau $T$

**Entrées :** $e$ est un indice entre 1 et $n-1$ ; on suppose le tableau trié entre les indices 1 et $e$.
**Sorties :** Modifie le tableau $T$ de départ, en insérant $x$ à sa place dans $T[1..e+1]$, pour obtenir le tableau $T[1..e+1]$ trié.

```
1 Insérer(x, e)
2 k := e
3 tant que T[k] > x faire
4 |   T[k+1] := T[k]
5 |   k := k − 1
6 T[k+1] := x
```

L'appel de TrierInserRec(1,$n$) provoque récursivement le tri du tableau complet.

**Algorithme 3 :** Procédure récursive de Tri-Insertion

**Entrées :** $d$ et $f$ sont des indices $> 0$ dans $T$.
**Sorties :** Modifie le tableau $T$ de départ, en triant ses éléments dans l'ordre croissant.

```
1 TrierInserRec(1,n)
2 si d < f alors
3 |   TrierInserRec(d, f − 1)
4 |   Insérer(T[f], f − 1)
```

L'appel de TrierInserIter(1,$n$) provoque itérativement le tri du tableau complet.

**Algorithme 4 :** Procédure itérative de Tri-Insertion

**Entrées :** Rien.
**Sorties :** Modifie le tableau $T$ de départ, en triant ses éléments dans l'ordre croissant.

```
1 TrierInserIter()
2 pour i allant de 2 à n faire
3 |  Insérer(T[i], i − 1)
```

La seule écriture d'un pseudo-code ne suffit certes pas à valider un algorithme. Nous distinguons deux points fondamentaux à vérifier impérativement :

— s'assurer que l'algorithme se termine.
— vérifier qu'il donne toujours le bon résultat.

De plus, on ne peut concevoir l'écriture d'un algorithme sans essayer de donner une idée de son coût algorithmique.

**Analyse de l'algorithme itératif du Tri-Insertion.**

*Preuve de Terminaison :*
La boucle **tant que** de l'algorithme 2, ligne 3 s'achève toujours (elle contient au plus $e+1$ itérations). De même, la boucle **pour** de l'algorithme 4, ligne 2 s'achève toujours. Donc, l'algorithme se termine après un nombre fini d'étapes.

*Preuve de validité :*
On a un *invariant de boucle* pour la boucle **Pour** $i$ **allant de** 2 **à** $n$ de la ligne 2 qui est *"les i premiers éléments sont triés en l'ordre croissant"*. Donc à la fin de l'algorithme, c'est-à-dire quand $i$=$n$, le tableau est totalement trié.

*Analyse de la complexité dans le pire des cas* en nombre de comparaisons :
Dans le pire des cas, la boucle **tant que** fait $e+1 \rightarrow i$ comparaisons à l'étape $i$ de la boucle de la ligne 3 (algo 2). Ce qui fait en sommant, $2+3+\cdots+n$ comparaisons au total, soit $n(n+1)/2-1$ comparaisons. Or, ce cas se produit quand on cherche à trier toute séquence strictement décroissante. **Le Tri-Insertion est donc un algorithme quadratique dans le pire des cas en nombre de comparaisons**.

Nous allons maintenant proposer un autre algorithme de tri appelé **Tri-Rapide**, qui repose sur le premier principe que l'on désire mettre en exergue dans le cours, le paradigme, **"Diviser pour Régner"**.

# 2 Diviser pour Régner

## 2.1 Stratégie et complexité de Diviser pour Régner

La stratégie Diviser pour Régner consiste à scinder un problème en sous-problèmes de même nature sur des instances plus petites, à résoudre ces sous-problèmes, puis à combiner les résultats obtenus pour apporter une solution au problème posé. Il s'agit donc d'une démarche essentiellement récursive. Le paradigme "Diviser pour Régner" donne donc lieu à trois étapes à chaque niveau de récursivité :
DIVISER : Le problème est scindé en un certain nombre de sous-problèmes ;
RÉGNER : Résoudre les sous-problèmes récursivement ou, si la taille d'un sous-problème est assez réduite, le résoudre directement ;
COMBINER : Réorganiser les solutions des sous-problèmes en une solution complète du problème initial.

Supposons que l'on ait un algorithme de type Diviser pour Régner, qui suit le principe suivant : pour être résolu le problème initial sur une instance de taille $n$ est divisé en $a$ sous-problèmes sur des instances de tailles $\approx \frac{n}{b}$. Puis ces solutions sont combinées en une solution du problème de taille $n$. Il y a donc plusieurs événements qui peuvent avoir un coût :

- le coût nécessaire à diviser le problème de taille $n$ (qui dépend de la taille de l'instance). Il sera noté $D(n)$.
- le coût pour combiner les solutions des sous-problèmes en une solution du problème de taille $n$, noté $C(n)$ (dépendant lui-aussi de la taille de l'instance).
- enfin, notons $T(n)$ le temps nécéssaire pour trouver la solution de taille $n$.

Le principe récursif diviser pour régner nous amène à une récurrence du type $T(n) = aT(\frac{n}{b}) + D(n) + C(n)$. De plus on peut supposer que pour les instances de taille 1, le coup pour trouver la solution est fixé et donc constant, $T(1) = c$. Les coûts d'exécution des algorithmes de type Diviser pour Régner suivent donc toujours le même type de récurrence mathématique. Nous allons voir qu'en général on sait assez bien trouver le comportement asymptotique des fonctions $T$ définies ainsi. Avant de donner une borne pour le coût d'exécution de l'algorithme Tri-Rapide, accordons nous quelques rappels concernant les comportements asymptotiques.

## 2.2 Comportement asymptotique : notations de Landau

On note $\mathcal{F}_{\mathcal{N}}$, l'ensemble des fonctions de $\mathbb{N}$ dans $\mathbb{R}^+$ (Ce sont les suites à valeurs dans $\mathbb{R}^+$). En algorithmique, nous nous intéressons spécifiquement au comportement au voisinage de l'infini de fonctions de $\mathcal{F}_{\mathcal{N}}$, ce qui est une (bonne ?) mesure permettant de comparer l'efficacité des algorithmes. Pour ce faire, étant données deux fonctions $f$ et $g$ de $\mathcal{F}_{\mathcal{N}}$, on dira que $f$ est *dominée* par $g$ au voisinage de l'infini (ou encore que $g$ est une *borne asymptotique supérieure* de $f$) si et seulement s'il existe deux constantes strictement positives $c$ et $n_0$ telles que pour tout $n > n_0$ on a $f(n) < cg(n)$. On notera alors $f = O(g)$ ou bien $g = \Omega(f)$.

On dira que $f$ et $g$ sont *semblables* au voisinage de l'infini (ou encore que $g$ est une *borne asymptotique approchée* de $f$) si et seulement si $f = O(g)$ et $f = \Omega(g)$. On notera

alors $f = \Theta(g)$.

En particulier, étant donnée une fonction $f$, on essayera toujours de lui trouver un équivalent parmi une échelle de comparaisons simples (du type $n^a \log(n)^b$ , exponentielle, ...).

De nombreux exemples seront vus en TD.

### 2.3 Le Tri-Rapide

Le Tri-Rapide, aussi appelé "tri de Hoare" (du nom de son inventeur) ou "tri par segmentation" ou "tri des bijoutiers" ou en anglais "quicksort", est un algorithme de tri intéressant à plus d'un titre. Tout d'abord, comme le Tri-Insertion, l'algorithme Tri-Rapide opère uniquement sur la séquence à trier. En particulier, il ne crée pas de tableau auxiliaire comme c'est le cas pour d'autres tris, comme le tri-fusion par exemple. De plus, l'algorithme Tri-Rapide a un coût en nombre de comparaisons raisonnable. Nous allons montrer en effet que le nombre moyen de comparaisons effectuées pour trier une séquence "quelconque" de longueur $n$ est $\Theta(n \log n)$. Ce qui en moyenne est aussi bon que les tris optimaux (tri-fusion).

L'algorithme Tri-Rapide repose sur le principe Diviser pour Régner de la manière suivante :

Diviser : Soit $T[p..r]$ le tableau contenant la séquence d'entiers que l'on étudie. On fixe une valeur $v$ de $T$ (ici $v = T(r)$) appelée *pivot* et on réarrange le tableau $T[p..r]$ de telle sorte qu'après cela les tableaux éventuellement vides, $T[p..q-1]$, $T[q+1..r]$, contiennent respectivement tous les éléments de $T[p..r-1]$ dont les valeurs sont plus petites ou égales à $v$ et toutes les valeurs de $T[p..r-1]$ strictement plus grandes que $v$ et que $T[q] = v$.

Régner :On résout par appel récursif le problème sur les sous-tableaux $T[p..q-1]$ et $T[q+1, r]$ ;

Combiner : Il n'y a rien à faire.

Commençons par donner le pseudo-code de réarrangement$(T, p, r)$.

**Algorithme 5 :** Réarrangements

**Entrées :** $T[1..n]$ un tableau de nombres, $p$ et $r$ des entiers.
**Sorties :** un entier (la position du pivot).

```
1 RéarrangementS(T, p, r)
2 v :=T[r]
3 i := p          # i correspond à la place que devrait prendre le pivot
4 pour j allant de p à r − 1 faire
5 |   si T[j] ≤ v alors
6 |   |   T[i] ↔ T[j]
7 |   |   i := i + 1
8 T[i] ↔ T[r]
9 retourner i
```

**Analyse de l'algorithme Réarrangements**

*Preuve de Terminaison :*
Nous n'avons dans cet algorithme qu'une boucle de type compteur. Donc l'algorithme s'arrête après un nombre fini d'étapes.

*Preuve de Validité :*
Nous allons utiliser pour ce faire l'invariant de boucle (pour la boucle de la ligne 4) suivant :

1. Si $p \leq k \leq i-1$, alors $T[k] \leq v$.
2. Si $i \leq k \leq j-1$, alors $T[k] > v$.
3. Si $k = r$, alors $T[k] = v$.

Initialisation : Avant la première itération, $i = p$ et $j = p$. Il n'y a pas de valeur entre $p$ et $i-1$, ni de valeur entre $i$ et $j-1$, de sorte que les deux premières conditions de l'invariant de boucle sont satisfaites de manière triviale. L'affectation en ligne 2 satisfait à la troisième condition.
Par induction : il y a deux cas à considérer, selon le résultat du test en ligne 5. Quand $T[j] > v$ l'unique action faite dans la boucle est d'incrémenter $j$. Une fois $j$ incrémenté, la condition 2 est vraie pour $T[j-1]$ et tous les autres éléments restent inchangés, donc l'invariant est préservé. Quand $T[j] \leq v$, $T[i]$ et $T[j]$ sont échangés la variable $i$ est incrémenté, puis $j$ est incrémenté. Compte tenu de la permutation, on a maintenant, $T[i] \leq v$ et la condition 1 est respectée. De même, on a aussi $T[j-1] > v$, car l'élément qui a été permuté avec $T[j-1]$ est, d'après l'invariant de boucle, plus grand que $v$.
Validité finale : À la fin, $j = r$. Par conséquent, chaque élément du tableau est dans l'un des trois ensembles décrits par l'invariant et l'on a réarrangé les valeurs du tableau de telle sorte qu'il débute par les valeurs inférieures ou égales à $v$, puis l'élément $v$, puis se finit par les valeurs supérieures à $v$.

*Analyse de la Complexité* en nombre de comparaisons :
On fait une comparaison à chaque itération de la boucle de la ligne 4, soit $r - p$ comparaisons. Donc pour une instance de longueur $n$, l'algorithme Réarrangements opère $n - 1$ comparaisons.

Maintenant, il est aisé de décrire un algorithme de type Diviser pour Régner à partir de Réarrangements, en voici le pseudo-code :

**Algorithme 6 :** Tri-Rapide

**Entrées :** $T[1..n]$ un tableau de nombres, $p$ et $r$ des entiers.
**Sorties :** Rien mais le tableau $T$ est trié entre $p$ et $r$.

```
1 Tri-Rapide(T, p, r)
2 si p < r alors
3 |   q := Réarrangements(T, p, r)
4 |   Tri-Rapide(T, p, q − 1)
5 |   Tri-Rapide(T, q + 1, r)
```

Pour trier un tableau $T$, il suffit alors de faire l'appel Tri-Rapide($T$, 1, longueur($T$)).

**Analyse de l'algorithme Tri-Rapide**

*Preuve de Terminaison :*
Par induction sur la longueur de la zone $[p..r]$ à trier, l'algorithme Tri-Rapide$(T, p, r)$ avec $p = r$ ne fait aucun appel à réarrangement($T$, $p$, $r$) donc il est fini. Supposons que pour tous $r$ et $p$ tels que $r - p < k$, l'algorithme Tri-Rapide$(T, p, r)$ se termine, alors pour tout $p$, Tri-Rapide$(T, p, p + k)$ se termine aussi car il appelle récursivement trois fonctions qui par hypothèse d'induction se terminent.

*Preuve de Validité :*
Par induction sur la longueur de la zone à trier, si $p = r$, c'est trivial. Supposons que pour tous $r$ et $p$ tels que $r - p < k$, l'algorithme Tri-Rapide$(T, p, r)$ renvoie le tableau $T$ trié entre $p$ et $r$ inclus, Considérons Tri-Rapide$(T, p, r)$ avec $r - p = k$. Comme Tri-Rapide$(T, p, q - 1)$ et Tri-Rapide$(T, q + 1, r)$ sont valides par hypothèse d'induction, on a bien que tout le tableau $T$ est trié de $p$ à $r$.

*Analyse de la Complexité* en nombre de comparaisons dans le pire des cas :
Le pire des cas est obtenu lorsque le tableau est trié dans l'ordre croissant ou décroissant, en effet, on appelle a chaque fois récursivement le tri sur un tableau de taille décrémentée de 1. Soit $T(n)$ le nombre de comparaisons dans le pire des cas :

$$T(n) = n - 1 + T(n - 1).$$

Par récurrence, $T(n) = n(n - 1)/2$.

On peut être quelque peu déboussolé au regard des résultats obtenus. En effet, dans le pire des cas, l'algorithme Tri-Rapide est plutôt mauvais (quadratique). De plus, il est mauvais sur des instances déjà triées ! Pourtant, comme nous allons le montrer le nombre de comparaisons en moyenne est très bon (en $n \log n$ pour une instance de taille $n$). Aux vues des autres avantages qu'il présente, c'est un algorithme très souvent implémenté (par exemple, c'est l'algorithme implémenté dans JAVA pour les tris ,tri standard de Unix).

*Idées pour l'Analyse de la Complexité* en nombre moyen de comparaisons dans le meilleur cas :
Le meilleur cas est obtenu lorsque le tableau est coupé en sa moitié à chaque fois. L'équation de récurrence vérifie :

$$T(n) = n - 1 + T(\lfloor \frac{n-1}{2} \rfloor) + T(\lceil \frac{n-1}{2} \rceil).$$

Nous allons calculer $T(n)$ pour certaines valeurs de $n$ sous la forme $2^p - 1$. On obtient dès lors :

$$T(2^p - 1) = 2^p - 2 + 2T(2^{p-1} - 1) \qquad T(0) = 0.$$

Par récurrence, $T(2^p - 1) = p2^p - 2^{p+1} + 2$ et par conséquent, asymptotiquement, $T(n) \sim n \log n$.

Le calcul de la complexité du tri rapide en moyenne est un peu plus compliqué. Déjà, il faut définir un espace de probabilité. On va considérer l'ensemble $E_n$ de toutes les séquences de longueur $n$ dont tous les éléments sont distincts. Soit $L = (a_1, ..., a_n)$ une séquence prise au hasard (équiprobablement) dans $E_n$, la probabilité que $a_n$ soit après triage le $i$-ème élément de la séquence est clairement $1/n$, de sorte que le coût moyen en nombre de comparaisons $C_n$ vérifie : $C_0 = 0$, $C_1 = 0$ et $C_n = n-1+\frac{1}{n}\sum_{i=0}^{n-1}(C_i + C_{n-i-1})$.

Le terme générique $C_n$ peut s'écrire :

$$\begin{aligned}
C_n &= n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} C_i \\
&= n - 1 + \frac{2}{n}C_{n-1} + \frac{2}{n}\sum_{i=0}^{n-2} C_i \\
&= n - 1 + \frac{2}{n}C_{n-1} + \frac{n-1}{n}\left(n - 2 + \frac{2}{n-1}\sum_{i=0}^{n-2} C_i\right) - \frac{(n-1)(n-2)}{n} \\
&= \frac{2}{n}C_{n-1} + \frac{n-1}{n}\left(C_{n-1}\right) + \frac{2n-2}{n} \\
&= \frac{n+1}{n}C_{n-1} + \frac{2(n-1)}{n}
\end{aligned}$$

En posant $D_n = C_n/(n+1)$, la récurrence s'écrit alors :

$$D_0 = 0 \text{ et } D_n = D_{n-1} + \frac{2}{(n+1)} - \frac{2}{(n+1)\,n}.$$

On rappelle que la série $H_n = H_{n-1} + 1/n$ est la *série harmonique* qui vérifie asymptotiquement $H_n = \ln(n) + 0,577... + \Theta(1/n)$ où $0,577\ldots$ est la *constante d'Euler*. On en déduit, comme le terme $\frac{2}{(n+1)n}$ est négligeable, que $D_n = \Theta(\ln(n))$ et donc que $C_n = \Theta(n\ln(n))$.

On peut donc démontrer que le Tri-Rapide est en $\Theta(n \log n)$ en moyenne.

### 2.4 Le théorème maître

Le théorème maître permet d'avoir une idée du comportement asymptotique de la plupart des récurrences induites par les algorithmes de type Diviser pour Régner, nous le proposons sans démonstration. Pour une démonstration, voir le livre de Berstel et ses co-auteurs. On peut trouver une version plus générales (mais avec des hypothèses plus difficiles à justifier dans le livre de Cormen et ses co-auteurs.

**Théorème 1** *(Résolution de récurrences "Diviser pour Régner"). Soient $a \geq 1$ et $b > 1$ deux constantes, soient $f$ une fonction et $T$ une fonction croissante, définie sur $\mathbb{N}$ par la récurrence*

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

*où l'on interprète $n/b$ comme signifiant $\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$. $T(n)$ peut être bornée asymptotiquement de la façon suivante :*

1. *Si* $f(n) = O(n^{(\log_b a)-\epsilon})$ *pour une certaine constante* $\epsilon > 0$, *alors* $T(n) = \Theta(n^{\log_b a})$.
2. *Si* $f(n) = \Theta(n^{\log_b a})$, *alors* $T(n) = \Theta(n^{\log_b a} \log n)$.
3. *Si* $f(n) = \Omega(n^{(\log_b a)+\epsilon})$ *pour une certaine constante* $\epsilon > 0$, *et si on a asymptotiquement pour une constante* $c < 1$, $af(n/b) < cf(n)$, *alors* $T(n) = \Theta(f(n))$.

**Remarque** : Le théorème maître ne couvre pas toutes les possibilités pour la fonction $f(n)$.

**Exemples** :

— $t(n) = 2t(n/2) + \log n$ alors $t(n) = \Theta(n)$ $(h = 1/2, k = 0, q = 1)$
— $t(n) = 3t(n/2) + n \log n$ alors $t(n) = \Theta(n^{\log 3})$ $(h = 2/3, k = q = 1)$
— $t(n) = 2t(n/2) + n \log n$ alors $t(n) = \Theta(n \log^2 n)$ $(h = k = q = 1)$
— $t(n) = 5t(n/2) + (n \log n)^2$ alors $t(n) = \Theta(n^{\log 5})$ $(h = 4/5, k = q = 2)$

Soit $t$ une fonction vérifiant :

$$\begin{aligned} t(1) &= a \\ t(n) &= t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + bn, \qquad n > 1. \end{aligned}$$

Clairement $t$ est croissante, et comme $t(n) = 2t(n/2) + bn$, lorsque $n$ est une puissance de 2, on a $t(n) = \Theta(n \log n)$.
**Remarque** : Retrouver le coût dans le meilleur des cas du tri rapide à l'aide du théorème maître.

## 2.5 Applications arithmétiques

### 2.5.1 Multiplication de deux entiers par la méthode de Karatsuba

Dans cette section, nous évaluerons le coût des algorithmes en nombre de multiplications élémentaires effectuées (multiplications de nombres à un chiffre). On remarque que pour le produit de deux nombres de $n$ chiffres en utilisant la méthode naïve (celle apprise à l'école), on fait $n^2$ multiplications élémentaires, le coût $T$ en calcul d'une multiplication de deux nombres à $n$ chiffres est donc $T(n) = \Theta(n^2)$.

L'algorithme de Karatsuba est une application du principe "diviser pour régner" qui permet d'améliorer le coût des multiplications des grands nombres. Le principe en est assez simple :

Soient $a$ et $b$ deux nombres positifs écrits en base 2 de $n = 2k$ chiffres, on peut écrire $a = (a_1 \times 2^k + a_0)$ et $b = (b_1 \times 2^k + b_0)$ avec $a_0, b_0, a_1, b_1$ des nombres binaires à $k$ chiffres. On remarque alors que le calcul de $(a_1 \times 2^k + a_0)(b_1 \times 2^k + b_0) = a_1 b_1 \times 2^{2k} + (a_1 b_0 + a_0 b_1) \times 2^k + a_0 b_0$ ne nécessite pas les quatre produits $a_1 b_1$, $a_1 b_0$, $a_0 b_1$ et $a_0 b_0$, mais peut en fait être effectué seulement avec les trois produits $a_1 b_1$, $a_0 b_0$ et $(|a_1 - a_0|)(|b_1 - b_0|)$ en regroupant les calculs sous la forme suivante (on note $sgn(x)$ le signe de $x$) :

$(a_1 \times 2^k + a_0)(b_1 \times 2^k + b_0) = a_1 b_1 \times 2^{2k} + (a_1 b_1 + a_0 b_0 - sgn(a_1 - a_0) sgn(b_1 - b_0)(|a_1 - a_0|)(|b_1 - b_0|)) \times 2^k + a_0 b_0$

Ce que l'on peut écrire ainsi :
DIVISER : On décompose $a$ et $b$ des nombres de $2k$ chiffres en $a = (a_1 \times 2^k + a_0)$ et $b = (b_1 \times 2^k + b_0)$ où $a_0, b_0, a_1, b_1$ sont des nombres à $k$ chiffres.
RÉGNER : On résout par appel récursif le problème pour $a_0b_0$, $a_1b_1$ et $(|a_1 - a_0|)(|b_1 - b_0|)$.
COMBINER : On obtient le produit $ab$ en faisant $(a_1 \times 2^k + a_0)(b_1 \times 2^k + b_0) = a_1b_1 \times 2^{2k} + (a_1b_1 + a_0b_0 - sgn(a_1 - a_0)sgn(b_1 - b_0)(|a_1 - a_0|)(|b_1 - b_0|)) \times 2^k + a_0b_0$.

On admet que les opérations $/2^k$ (division entière par une puissance de 2) et $\bmod 2^k$ (reste de la division entière par une puissance de 2) ne nécessitent pas de multiplication. En fait, ces opérations se font en machine en temps constants et sont négligeables. D'où le pseudo-code suivant :

**Algorithme 7 :** Karatsuba

**Entrées :** deux entiers positifs.
**Sorties :** un entier.

```
1  Karatsuba(a, b)
2  k := max(⌊log2 a⌋, ⌊log2 b⌋) + 1
3  si k ≤ 1 alors
4      retourner ab
5  sinon
6      k := ⌊n/2⌋
7      a0 := a mod 2^k
8      a1 := a/2^k
9      b0 := b mod 2^k
10     b1 := b/2^k
11     x := Karatsuba(a0, b0)
12     y := Karatsuba(a1, b1)
13     z := Karatsuba(((|a1 − a0|), (|b1 − b0|))
14     retourner y × 2^2k + (x + y − sgn(a1 − a0)sgn(b1 − b0)z) × 2^k + x
```

### Analyse de l'algorithme Karatsuba

*Preuve de Terminaison :*
Par induction sur $k = \max(\lfloor \log_2 a \rfloor, \lfloor \log_2 b \rfloor) + 1$. Si $k = 1$ alors, on ne fait qu'une multiplication, donc l'algorithme est fini. Supposons que pour tout $k < n_0$, l'algorithme Karatsuba$(a, b)$ se termine, alors pour tous $a$ et $b$ avec $n_0 = \max(\lfloor \log_2 a \rfloor, \lfloor \log_2 b \rfloor) + 1$, Karatsuba$(a, b)$ se termine aussi car il appelle recursivement trois copies de Karatsuba sur des instances plus petites (qui par hypothèse d'induction se terminent).

*Preuve de Validité :*
Par induction sur $k = \max(\lfloor \log_2 a \rfloor, \lfloor \log_2 b \rfloor) + 1$. Si $k = 1$ alors, Karatsuba$(a, b)$ renvoie $ab$. Supposons que pour tout $k < n_0$, l'algorithme Karatsuba$(a, b)$ renvoie $ab$, alors pour tous $a$ et $b$ avec $n_0 = \max(\lfloor \log_2 a \rfloor, \lfloor \log_2 b \rfloor) + 1$, Karatsuba$(a, b) = y \times 2^{2k} + (x + y - sgn(a_1 - a_0)sgn(b_1 - b_0)z) \times 2^k + x$ qui vaut $ab$ car par récurrence $x = a_0b_0$, $y = a_1b_1$ et $z = (|a_1 - a_0|)(|b_1 - b_0|)$.

*Analyse de la Complexité* en nombre de multiplications élémentaires :
Notons $T(n)$ le nombre de multiplications élémentaires nécessaires pour multiplier deux nombres de $n$ chiffres. Le principe Diviser pour Régner permet de donner la formule de récurrence pour $T$ suivante : $T(1) = 1$ et $T(n) = 3T(n/2)$. On en déduit que $T(n) = \Theta(n^{\frac{\ln(3)}{\ln(2)}})$. Or $\frac{\ln(3)}{\ln(2)} \approx 1.585$, ce qui bien mieux que le $n^2$ de la multiplication naïve.

### 2.5.2 Division récursive

Le problème consiste à trouver le quotient et le reste de la division d'un nombre de $n+m$ chiffres en base 2 par un nombre de $n$ chiffres (en base 2). La division naïve que l'on apprend à l'école nécessite $O(mn)$ opérations élémentaires sur des chiffres (disons multiplications et divisions). On va ici proposer un algorithme plus rapide, qui repose sur le principe diviser pour régner. Commençons par une remarque de bon sens, les chiffres de poids fort du quotient dépendent en majeure partie des chiffres de poids fort du dividende et du diviseur.

**Algorithme 8 :** Div-Réc

**Entrées :** deux entiers positifs respectivement de $n+m$ et $n$ chiffres en base 2 avec $m \leq n$.

**Sorties :** le quotient et le reste de la division de $A$ par $B$.

```
1  Div-Réc(A, B)
2  si m < 2 alors
3      retourner Div-Naïve(A, B)
4  k := ⌊m/2⌋
5  B_1 := B/2^k
6  B_0 := B mod 2^k
7  (Q_1, R_1) := Div-Réc(A/2^{2k}, B_1)
8  A' := 2^{2k} R_1 + A mod 2^{2k} − 2^k Q_1 B_0
9  tant que A' < 0 faire
10     Q_1 := Q_1 − 1
11     A' := A' + 2^k B
12 (Q_0, R_0) := Div-Réc(A'/2^k, B_1)
13 A'' := 2^k R_0 + A' mod 2^k − Q_0 B_0
14 tant que A'' < 0 faire
15     Q_0 := Q_0 − 1
16     A'' := A'' + B
17 Q := 2^k Q_1 + Q_0
18 retourner (Q, A'')
```

**Analyse de l'algorithme Div-Réc**

*Preuve de Terminaison :*
Par induction sur $m$. Si $m < 2$, l'algorithme retourne un résultat. Supposons que pour tout $m < n_0$, l'algorithme Div-Réc$(A, B)$ se termine, alors pour tous $A$ et $B$ avec $m = n_0$, Div-Réc$(A, B)$ se termine aussi car il appelle récursivement deux copies de Div-Réc sur des instances plus petites qui par hypothèse d'induction se terminent.

*Preuve de Validité :*
Par induction sur $m$. Si $m < 2$, l'algorithme retourne le bon résultat. Supposons que pour tout $m < n_0$, l'algorithme Div-Réc$(A, B)$ renvoie le reste et le quotient de $A$ par $B$, alors dans ce cas ligne 7, $Q_1$ et $R_1$ sont bien le quotient et le reste de $\lfloor A/2^{2k} \rfloor$ par $B_1$ et ligne 12, $Q_0$ et $R_0$ sont bien le quotient et le reste de $\lfloor A'/2^k \rfloor$ par $B_1$. Posons, $A_1 = A/2^{2k} = B_1Q_1 + R_1$, on obtient $A' := 2^{2k}(A_1 - B_1Q_1) + A \bmod 2^{2k} - 2^kQ_1B_0$. Or, $A = 2^{2k}(A_1) + A \bmod 2^{2k}$ et $2^kQ_1B = 2^{2k}Q_1B_1 + 2^kQ_1B_0$. Donc, ligne 8, $A' = A - 2^kQ_1B$. Maintenant, $A' < 2^kB$. En effet, $A_1 = Q_1B_1 + R_1$ et en utilisant $A - 2^{2k} + 1 \leq 2^{2k}A_1$ et $2^kB_1 \leq B$, on obtient $A - 2^{2k} + 1 \leq 2^{2k}R_1 + 2^kQ_1B$. Soit encore $A - 2^kQ_1B \leq 2^{2k}(R_1 + 1) - 1$. Or, $2^k(R_1 + 1) \leq 2^kB_1 \leq B$. Ceci montre bien que $A' < 2^kB$. Donc, ligne 12, $A'$ est le reste de $A$ par $2^kB$. Par le même raisonnement, on montre que ligne 13, $A'' = A' - Q_0B$ et $A'' < B$. Donc, ligne 17, $A''$ est le reste de $A$ divisé par $B$. Le couple $(Q, A'')$ est bien le résultat que l'on veut.

*Analyse de la Complexité* en nombre d'opérations élémentaires :
Notons $D(m, n)$ le nombre d'opérations élémentaires. Posons $m' = m/2$. On a donc $D(m, n) = D(2m', n) = 2D(m', n - m') + 2M(m') + O(n)$ où $M(m')$ désigne le coût d'une multiplication de deux nombres de $m'$ chiffres. En fait, les seules opérations ayant un coût sont les deux multiplications $Q_1B_0$ ligne 8 et $Q_0B_0$ ligne 13. Or, $B_0$ a $m'$ chiffres et $Q_1$ et $Q_0$ ont au plus $m' + 1$ chiffres, d'où le terme $2M(m')$ et un nombre fixe d'additions de nombres d'au plus $n + m'$ chiffres dans les deux boucles tant que (on peut voir que les boucles tant que ne font pas plus de 3 passes.). Cela donne un terme en $O(n)$, le reste découle des deux appels récursifs. En effet, $B_1$ a $n - m'$ chiffres et $A/2^{2k}$ et $A'/2^k$ ont $n$ chiffres. Cela donne le terme $2D(m', n - m')$. En particulier, si l'on divise un nombre de $2n$ chiffres par un nombre de $n$ chiffres, on obtient la récurrence suivante : $D(n, n) = D(\lfloor n/2 \rfloor, \lceil n/2 \rceil) + 2M(n) + O(n)$. En utilisant Karatsuba pour calculer les multiplications $M(n) = O(n^{\frac{\ln(3)}{\ln(2)}})$ et le théorème maître, on trouve $D(n, n) = O(n^{\frac{\ln(3)}{\ln(2)}})$.

Trace de l'algorithme sur un exemple : $A =_{10} 45$; $B =_{10} 1010$.
Commençons par convertir $A$ et $B$ en base 2 : $A =_2 101101$; $B =_2 10$.

| Div-Réc(101101,1010) | | |
|---|---|---|
| | $k := 1$ | |
| | $B_1 := 101$ | |
| | $B_0 := 0$ | |
| | $(Q_1, R_1) :=$ Div-Réc(1011,101) | |
| | | $m := 1$ |
| | | Div-Naïve(1011,101) |
| | | retourner $(10, 1)$ |
| | $A' := 100 + 01 - 0 = 101$ | |
| | $(Q_0, R_0) :=$ Div-Réc(10,101) | |
| | | $m := -1$ |
| | | Div-Naïve(10,101) |
| | | retourner $(0, 101)$ |
| | $A'' := 100 + 1 - 0 = 101$ | |
| | $Q := 100$ | |
| | retourner $(100, 101)$ | |

# 3 Programmation Dynamique

Dans cette partie du cours, nous abordons les algorithmes de type Programmation Dynamique. Nous illustrons ce type de programmation par un premier exemple introductif concernant le calcul du $n$-ième nombre de Fibonacci.

La programmation dynamique, comme le principe Diviser pour Régner, est un principe algorithmique reposant sur une approche récursive des problèmes. Certains problèmes, quand on les scinde, font appel plusieurs fois aux mêmes sous-problèmes, ou à des sous-problèmes qui ne sont pas indépendants les uns des autres. Dans ce cas, les méthodes Diviser pour Régner ne sont pas aussi efficaces. En voici un exemple très simple. La suite de Fibonacci est définie récursivement par $F_0 = 0$, $F_1 = 1$ et $F_{n+2} = F_{n+1} + F_n$ pour $n \geq 0$. Supposons que l'on cherche à calculer le $n$-ième terme de cette suite. Si on utilise directement le principe Diviser pour Régner suivant :
DIVISER : On remarque que pour calculer $F_n$, il suffit de savoir calculer $F_{n-1}$ et $F_{n-2}$.
RÉGNER : On résout le problème récursivement pour $F_{n-1}$ et $F_{n-2}$ si l'indice est supérieur à 2, sinon on remplace directement.
COMBINER : On additionne $F_{n-1}$ et $F_{n-2}$.

On obtient le pseudo-code ci-dessous :

**Algorithme 9 :** Fibonacci

**Entrées :** Un entier positif $n$.
**Sorties :** Le $n$-ième nombre de Fibonacci.

```
1 Fibo(n)
2 si n < 2 alors
3     retourner n
4 sinon
5     retourner Fibo(n − 1)+Fibo(n − 2)
```

**Analyse de l'algorithme Fibo :**
*Preuve de Terminaison :*
Immédiat.
*Preuve de Validité :*
Immédiat.
*Analyse de la Complexité* en nombre d'additions :
On a clairement la relation de récurrence suivante pour le nombre d'additions : $\mathcal{T}(0) = 0$, $\mathcal{T}(1) = 0$ et $\mathcal{T}(n) = \mathcal{T}(n-1) + \mathcal{T}(n-2) + 1$. Un calcul rapide montre que $\mathcal{T}(n) = \Theta(\phi^n)$ avec $\phi = (1 + \sqrt{5})/2$.

Cette approche se révèle trop couteuse, car on calcule plusieurs fois les mêmes sous-problèmes comme le montre le déroulement de l'algorithme pour trouver $F_5$ :
1er niveau de récursion : Pour trouver $F_5$, on calcule $F_4$ et $F_3$
2ème niveau : Pour trouver $F_4$, on calcule $F_3$ et $F_2$ et pour trouver $F_3$, on calcule $F_2$.
3ème niveau : Pour trouver $F_3$, on calcule $F_2$

En conclusion, pour trouver $F_5$, on a calculé 1 fois $F_4$, 2 fois $F_3$, 3 fois $F_2$. Un algorithme Diviser pour Régner fait plus de travail que nécessaire, en résolvant plusieurs fois des sous-problèmes identiques. Alors qu'évidemment, il aurait suffit de les calculer chacun une fois ! La programmation dynamique permet de pallier ce problème. Pour ce faire, un algorithme de type Programmation Dynamique résout chaque sous-problème une seule fois et mémorise sa réponse dans un tableau, évitant ainsi le recalcul de la solution chaque fois que le sous-problème est rencontré de nouveau.

Voici le pseudo-code d'un algorithme de type Programmation Dynanique pour trouver la valeur du $n$-ième nombre de Fibonacci. Attention, il faut faire l'initialisation du tableau en dehors de la fonction récursive.

**Algorithme 10 :** Fibo-Prog-Réc

**Entrées :** Un entier positif $n$.
**Sorties :** Le $n$-ième nombre de Fibonacci.

```
1 Initialisation(T, n)            ♯ Créer un tableau T de longueur n initialisé à -1
2 T[0] :=0 ; T[1] :=1

3 Fibo-Prog-Réc(n)
4 si n < 2 alors
5 |  retourner n
6 si T[n − 1] = −1 alors
7 |  T[n − 1] := Fibo-Prog-Réc(n − 1)      ♯ il est inutile de tester T[n − 2] = −1
8 T[n] := T[n − 1] + T[n − 2]
9 retourner T[n]
```

**Analyse de l'algorithme Fibo-Prog-Réc :**

*Preuve de Terminaison :*
Fibo-Prog-Réc($n$) s'arrête quand $n$ vaut 0 ou 1. Comme Fibo-Prog-Réc($n$) appelle uniquement (de manière directe) Fibo-Prog-Réc($n-1$). On en déduit que si Fibo-Prog-Réc($n-1$) s'arrête, alors l'algorithme Fibo-Prog-Réc($n$) s'arrête aussi. Par induction, on a la preuve de terminaison.

*Preuve de Validité :*
Par induction, montrons qu'à la sortie de Fibo-Prog-Réc($n$), pour tout $0 \leq i \leq n$, on a $T[i] = F_i$ et pour $j > n$, on a $T[j]$ =Nil. C'est vrai quand $n$ vaut 1. Supposons qu'à la sortie de Fibo-Prog-Réc($n$) $T$ soit bien rempli jusqu'à la case $n$ et pour $j > n$, on a $T[j]$ =Nil, alors quand on lance Fibo-Prog-Réc($n+1$), à la ligne 8, on fait l'appel Fibo-Prog-Réc($n$) et donc le tableau $T$ est alors rempli jusqu'à $n$, enfin ligne 9, on remplit correctement la case $n+1$. Donc à la sortie de Fibo-Prog-Réc($n+1$) le tableau est bien rempli jusqu'à $n+1$ et vide après. Par récurrence, la validité de l'algorithme s'ensuit.

*Analyse de la Complexité* en nombre d'additions :
Fibo-Prog-Réc($n+1$) fait une addition ligne 9 et appelle Fibo-Prog-Réc($n$) ligne 8. Donc le nombre d'additions vérifie : $\mathcal{T}(0) = 0$, $\mathcal{T}(1) = 0$ et $\mathcal{T}(n+1) = \mathcal{T}(n) + 1$. Ceci donne $\mathcal{T}(n) = \Theta(n)$.

On priviligiera donc cette approche quand la solution d'un problème sur une instance de taille $n$ s'exprime en fonction de solutions du même problème sur des instances de taille inférieure à $n$ et qu'une implémentation récursive du type diviser pour Régner conduit à rappeler de nombreuses fois les mêmes sous-problèmes.

Il y a en fait deux possibilités pour implémenter un algorithme de type Programmation Dynamique qui dépendent de la manière utilisée pour remplir le tableau (itérative ou récursive) :

— Une version récursive, appelée aussi *memoizing*, pour laquelle à chaque appel, on regarde dans le tableau si la valeur a déjà été calculée. Si c'est le cas, on récupère la valeur mémorisée. sinon, on la calcule et on la stocke. Cette méthode permet de ne pas avoir à connaître à l'avance les valeurs à calculer. C'est celle utilisée pour le pseudo-code de Fibo-Prog-Réc.

— Une version itérative où l'on initialise les cases correspondant aux cas de base. Puis on remplit le tableau selon un ordre permettant à chaque nouveau calcul de n'utiliser que les solutions déjà calculées.

Cette dernière donne pour le calcul du $n$-ième nombre de Fibonacci, le pseudo-code suivant :

**Algorithme 11 :** Fibo-Prog-It

**Entrées :** Un entier positif $n$.
**Sorties :** Le $n$-ième nombre de Fibonacci.

```
1 Fibo-Prog-It(n)
2 Créer un tableau T de longueur n initialisé à Nil
3 T[0] := 0
4 T[1] := 1
5 pour i allant de 2 à n faire
6 |  T[i] := T[i − 1] + T[i − 2]
7 retourner T[n]
```

On remarque que dans ce cas très simple, il est même possible de ne pas créer de tableau en faisant :

**Algorithme 12 :** Fibo-Prog-It2

**Entrées :** Un entier positif $n$.
**Sorties :** Le $n$-ième nombre de Fibonacci.

```
1  Fibo-Prog-It2(n)
2  si n < 2 alors
3      retourner n
4  i := 0
5  j := 1
6  pour k allant de 2 à n faire
7      temp := j
8      j := j + i
9      i := temp
10 retourner j
```

**Analyse de l'algorithme Fibo-Prog-It2 :**

*Preuve de Terminaison :*
Immédiat.

*Preuve de Validité :*
Considérons l'invariant de boucle sur $k$ suivant : $i$ vaut $F_{k-2}$ et $j$ vaut $F_{k-1}$. Quand on entre pour la première fois dans la boucle c'est vrai. Maintenant, supposons qu'au début de la $k$-ième itération, on a $i = F_{k-2}$ et $j = F_{k-1}$, alors, ligne 8, $j$ devient $F_k$ et $i$ prend la valeur de *temp* qui est $j = F_{k-1}$. Donc l'invariance est préservée. A la sortie de la boucle $j$ vaut donc $F_n$, ce qui est le résultat attendu.

*Analyse de la Complexité* en nombre d'additions :
Fibo-Prog-It2($n$) fait une addition à chaque itération de la boucle sur $k$, soit $n-1$ additions. Ceci donne donc $\mathcal{T}(n) = \Theta(n)$.

### 3.1 Plus Longue Sous-séquence Commune

Le problème de la plus longue sous-séquence commune consiste, étant données deux séquences $X = (x_1, x_2, ..., x_m)$ et $Y = (y_1, y_2, ..., y_n)$, à trouver une sous-séquence (suite extraite, les éléments ne sont pas forcément consécutifs) commune à $X$ et $Y$ de longueur maximale. Nous allons montrer que le problème de la plus longue sous-séquence commune, que l'on notera en abrégé PLSC, peut être résolu efficacement grâce à la Programmation Dynamique.

Exemple : $X = 0123456789$ et $Y = 27465$ alors $PLSC = 246$.

### 3.1.1 Propriété d'une plus longue sous-séquence commune

On pourrait essayer de résoudre le problème de la PLSC en énumérant toutes les sous-séquences de $X$ en ordre décroissant suivant leur longueur et en les testant pour savoir si elles sont aussi des sous-séquences de $Y$, et en s'arrêtant à la première sous-séquence qui vérifie le test. Chaque sous-séquence de $X$ correspond par sa fonction indicatrice à un sous-ensemble des indices $1, 2, ..., m$ de $X$. Il existe donc $2^m$ sous-séquences de $X$, de sorte que cette approche demande un traitement en temps exponentiel dans le pire des cas (c'est-à-dire si la PLSC est de longueur 1), ce qui rend cette technique inexploitable pour de longues séquences. Or, le problème de la PLSC possède une propriété qui va nous permettre de suivre un principe de Programmation Dynamique. Etant donnée une séquence $X = (x_1, x_2, \ldots, x_m)$, on définit le $i$-ème préfixe de $X$, pour $i = 0, 1, \ldots, m$, par $X_i = (x_1, x_2, \ldots, x_i)$. Par exemple, si $X = (A, B, C, B, D, A, B)$, alors $X_4 = (A, B, C, B)$ et $X_0$ représente la séquence vide.

**Lemme 1** *Soient deux séquences $X = (x_1, x_2, \ldots, x_m)$ et $Y = (y_1, y_2, \ldots, y_n)$, et soit $Z = (z_1, z_2, \ldots, z_k)$ une PLSC de $X$ et $Y$.*

1. *Si $x_m = y_n$, alors $z_k = x_m = y_n$ et $Z_{k-1}$ est une PLSC de $X_{m-1}$ et $Y_{n-1}$.*
2. *Si $x_m \neq y_n$ et $z_k \neq x_m$ alors $Z$ est une PLSC de $X_{m-1}$ et $Y$.*
3. *Si $x_m \neq y_n$ et $z_k \neq y_n$ alors $Z$ est une PLSC de $X$ et $Y_{n-1}$.*

**Preuve.**

1. Si $z_k \neq x_m$, on peut concaténer $x_m = y_n$ à $Z$ pour obtenir une sous-séquence commune de $X$ et $Y$ de longueur $k+1$, ce qui contredit l'hypothèse selon laquelle $Z$ est une PLSC de $X$ et $Y$. On a donc forcément $z_k = x_m = y_n$. Or, le préfixe $Z_{k-1}$ est une sous-séquence commune de longueur $(k-1)$ de $X_{m-1}$ et $Y_{n-1}$. On va montrer que c'est une PLSC. Supposons, en raisonnant par l'absurde, qu'il existe une sous-séquence commune $W$ de $X_{m-1}$ et $Y_{n-1}$ de longueur plus grande que $k-1$. Alors, la concaténation de $x_m = y_n$ à $W$ produit une sous-séquence commune à $X$ et $Y$ dont la longueur est plus grande que $k$, ce qui est contradictoire avec la maximalité de $k$.
2. Si $z_k \neq x_m$, alors $Z$ est une sous-séquence commune de $X_{m-1}$ et $Y$. S'il existait une sous-séquence commune $W$ de $X_{m-1}$ et $Y$ de taille supérieure à $k$, alors $W$ serait aussi une sous-séquence commune de $X$ et $Y$ ce qui contredit l'hypothèse selon laquelle $Z$ est une PLSC de $X$ et $Y$.
3. Identique au cas 2.

□

La caractérisation du lemme 1 montre qu'une PLSC de deux séquences contient une PLSC pour des préfixes respectifs $X'$ et $Y'$ des deux séquences $X$ et $Y$. Nous dirons que le problème de la PLSC possède la propriété de *sous-structure optimale*. C'est-à-dire que l'on peut construire une solution optimale à partir de sous-solutions optimales.

### 3.1.2 Une solution récursive

Grâce au lemme 1 nous pouvons construire récursivement une solution comme suit. Si $x_m = y_n$, on cherche une PLSC de $X_{m-1}$ et $Y_{n-1}$. La concaténation de $x_m = y_n$ à cette PLSC engendre une PLSC de $X$ et $Y$. Si $x_m \neq y_n$, alors on doit alors résoudre deux sous-problèmes : trouver une PLSC de $X_{m-1}$ et $Y$, et trouver une PLSC de $X$ et $Y_{n-1}$. La plus grande des deux PLSC est une PLSC de $X$ et $Y$. Comme ces cas épuisent toutes les possibilités, on sait que l'une des solutions optimales des sous-problèmes doit servir dans une PLSC de $X$ et $Y$.

Appelons $c[i, j]$ la longueur d'une PLSC des séquences $X_i$ et $Y_j$. Si $i = 0$ ou $j = 0$, l'une des séquences a une longueur nulle, et donc la PLSC est de longueur nulle. La sous-structure optimale du problème de la PLSC débouche sur la formule récursive :

$$c[i,j] = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0, \\ c[i-1, j-1] + 1 & \text{si } i > 0 \text{ et } j > 0 \text{ et } x_i = y_j, \\ \max\{c[i, j-1], c[i-1, j]\} & \text{si } i > 0 \text{ et } j > 0 \text{ et } x_i \neq y_j. \end{cases}$$

On remarque que, dans cette définition récursive, que la condition $x_i = y_j$ restreint le nombre de sous-problèmes à considérer. En effet, on considère alors le sous-problème consistant à trouver la PLSC de $X_{m-1}$ et $Y_{n-1}$ et dans ce cas, on n'a donc pas besoin de trouver une PLSC de $X_{m-1}$ et $Y$ (ni de PLSC de $X$ et $Y_{n-1}$). Il y a donc des sous-problèmes qu'il n'est pas nécessaire de traiter pour répondre au problème initial.

### 3.1.3 Calcul de la longueur d'une PLSC

En se rapportant à la définition récursive des $c[i, j]$, on peut faire appel au principe de la Programmation Dynamique pour calculer les solutions de manière itérative. La procédure Longueur-PLSC prend deux séquences $X = (x_1, x_2, \ldots, x_m)$ et $Y = (y_1, y_2, ..., y_n)$ en entrée. Elle stocke les valeurs $c[i, j]$ dans un tableau $c[0..m, 0..n]$ dont les éléments sont calculés dans l'ordre des lignes. (Autrement dit, la première ligne de $c$ est remplie de la gauche vers la droite, puis la deuxième ligne, etc.) Elle gère aussi un tableau $b[1..m, 1..n]$ pour simplifier la construction d'une solution optimale. Intuitivement, $b[i, j]$ pointe vers l'élément du tableau qui correspond à la solution optimale du sous-problème choisie pendant le calcul de $c[i, j]$. A la sortie de l'algorithme $c[m, n]$ contient la longueur

d'une PLSC de $X$ et $Y$.

**Algorithme 13 :** Longueur-PLSC

**Entrées :** $X$ et $Y$ deux tableaux contenant respectivement $x_1, x_2, \ldots, x_m$ et $y_1, y_2, ..., y_n$

**Sorties :** Rien mais on a rempli les tableaux $c$ et $b$

```
1  Longueur-PLSC(X, Y)
2  m := longueur(X)
3  n := longueur(Y)
4  pour i allant de 1 à m faire
5      c[i, 0] := 0
6  pour j allant de 1 à n faire
7      c[0, j] := 0
8  pour i allant de 1 à m faire
9      pour j allant de 1 à n faire
10         si x[i] = y[j] alors
11             c[i, j] := c[i − 1, j − 1] + 1
12             b[i, j] := " ↙ "
13         sinon
14             si c[i − 1, j] ≥ c[i, j − 1] alors
15                 c[i, j] := c[i − 1, j]
16                 b[i, j] := " ↓ "
17             sinon
18                 c[i, j] := c[i, j − 1]
19                 b[i, j] := " ← "
```

**Analyse de l'algorithme Longueur-PLSC :**

*Preuve de Terminaison :*
Immédiat.

*Preuve de Validité :*
Elle découle naturellement de la définition inductive de $c[i, j]$.
*Analyse de la Complexité* en nombre de comparaisons entre un élément de $X$ et un élément de $Y$ :
Le nombre de comparaisons faites est $mn$, puisque l'on fait une comparaison à chaque itération des boucles imbriquées lignes 8-9. On a donc un algorithme en $\Theta(mn)$.

### 3.1.4 Construction d'une PLSC

Le tableau $b$ construit par Longueur-PLSC peut servir à retrouver rapidement une PLSC de $X = (x_1, x_2, \ldots, x_m)$ et $Y = y_1, y_2, \ldots, y_n)$. On commence tout simplement en $b[m, n]$ et on se déplace dans le tableau en suivant les flèches. Chaque fois que nous rencontrons une $\swarrow$ dans l'élément $b[i, j]$, on sait que $x_i = y_j$ appartient à la PLSC. Cette méthode permet de retrouver les éléments de la PLSC dans l'ordre inverse. La

procédure récursive suivante imprime une PLSC de $X$ et $Y$ dans le bon ordre. L'appel initial est Afficher-PLSC($b, X$, longueur[$X$], longueur[$Y$]). La procédure prend un temps $O(m+n)$, puisqu'au moins l'un des deux indices $i$ ou $j$ est décrémenté à chaque étape de la récursivité.

**Algorithme 14 :** Afficher-PLSC

**Entrées :** Le tableau $b$, le tableau $X$ et deux entiers $i$, $j$
**Sorties :** Rien mais imprime une PLSC

```
1  Afficher-PLSC(b, X, i, j)
2  si i = 0 ou j = 0 alors
3      Stop
4  si b[i, j] = " ↙ " alors
5      Afficher-PLSC(b, X, i − 1, j − 1)
6      afficher x[i]
7  sinon
8      si b[i, j] = " ↓ " alors
9          Afficher-PLSC(b, X, i − 1, j)
10     sinon
11         Afficher-PLSC(b, X, i, j − 1)
```

Trace de l'algorithme avec X = 'ABCDE' et Y = 'EAACFE'.

```
i =  1  j =  1  et  0  >=  0
i =  1  j =  2  et  A  ==  A
i =  1  j =  3  et  A  ==  A
i =  1  j =  4  et  0  <  1
i =  1  j =  5  et  0  <  1
i =  1  j =  6  et  0  <  1
i =  2  j =  1  et  0  >=  0
i =  2  j =  2  et  1  >=  0
i =  2  j =  3  et  1  >=  1
i =  2  j =  4  et  1  >=  1
i =  2  j =  5  et  1  >=  1
i =  2  j =  6  et  1  >=  1
i =  3  j =  1  et  0  >=  0
i =  3  j =  2  et  1  >=  0
i =  3  j =  3  et  1  >=  1
i =  3  j =  4  et  C  ==  C
i =  3  j =  5  et  1  <  2
i =  3  j =  6  et  1  <  2
i =  4  j =  1  et  0  >=  0
i =  4  j =  2  et  1  >=  0
i =  4  j =  3  et  1  >=  1
i =  4  j =  4  et  2  >=  1
i =  4  j =  5  et  2  >=  2
```

```
i =  4  j =  6  et  2  >=  2
i =  5  j =  1  et  E  ==  E
i =  5  j =  2  et  1  >=  1
i =  5  j =  3  et  1  >=  1
i =  5  j =  4  et  2  >=  1
i =  5  j =  5  et  2  >=  2
i =  5  j =  6  et  E  ==  E
```

Matrice $c$ : (la matrice se lit de haut en bas)

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 2 & 2 & 2 \\ 0 & 0 & 1 & 1 & 2 & 2 & 2 \\ 0 & 0 & 1 & 1 & 2 & 2 & 3 \end{pmatrix}$$

Matrice $b$ : (la matrice se lit de haut en bas)

$$\begin{pmatrix} . & . & . & . & . & . & . \\ . & \uparrow & \nwarrow & \nwarrow & \leftarrow & \leftarrow & \leftarrow \\ . & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\ . & \uparrow & \uparrow & \uparrow & \nwarrow & \leftarrow & \leftarrow \\ . & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\ . & \nwarrow & \uparrow & \uparrow & \uparrow & \uparrow & \nwarrow \end{pmatrix}$$

## 3.2 Le problème du sac à dos

Nous allons ici illustrer le principe de la Programmation Dynamique par un algorithme qui résout le problème dit du sac à dos. Ce problème peut être décrit de la manière suivante. On dispose de $n$ objets qui ont chacun une valeur de $v_i$ euros et un poids de $p_i$ kg et d'un sac à dos qui ne peut pas supporter un poids total supérieur à $C$ kg. Quels objets doit-on prendre pour maximiser la valeur totale que l'on peut transporter dans le sac à dos ? En termes plus mathématiques, le problème peut donc être posé ainsi : Etant donnés $C$ un nombre entier positif et $v$ et $p$ deux séquences de $n$ nombres entiers positifs, trouver un sous-ensemble $I$ de $\{1, ..., n\}$ tel que $\sum_{i \in I} p_i \leq C$ et pour lequel $\sum_{i \in I} v_i$ est maximal.

### 3.2.1 Approche directe

Une approche directe du problème pourrait consister à regarder tous les sous-ensembles $I$ d'indices, de calculer, pour chacun d'eux, $\sum_{i \in I} p_i$ et parmi ceux qui sont valides (ceux tels que $\sum_{i \in I} p_i \leq C$) prendre celui pour lequel $\sum_{i \in I} v_i$ est maximal. Mais cette démarche naïve n'est pas algorithmiquement convaincante car il y a $2^n$ sous-ensembles de $\{1, ..., n\}$ ce qui nous amènerait à faire un nombre exponentiel d'opérations pour trouver une solution.

### 3.2.2 Sous-Structure optimale

Encore une fois, une amélioration vient de la structure particulière que possède une solution optimale. Supposons que l'on ait une solution optimale $I_{max}$. On peut distinguer deux cas :

— soit $n \in I_{max}$ et dans ce cas, $I_{max}(C')\backslash\{n\}$ est une solution optimale pour le problème où le sac à dos a pour contenance maximale $C' = C - p_n$ et où l'on cherche à le remplir avec les $n-1$ premiers objets (de poids respectifs $p_1, ..., p_{n-1}$ et de valeurs $v_1, ..., v_{n-1}$). En effet, si on avait une solution meilleure $I'$ pour ce dernier problème, le sous-ensemble d'indices $I' \bigcup \{n\}$ serait une solution qui contredirait l'optimalité de la solution $I_{max}$.

— soit $n \notin I_{max}$ et dans ce cas, $I_{max}(C)$ est une solution optimale pour le problème où le sac à dos a pour contenance maximale $C$ et où l'on cherche à le remplir avec les $n-1$ premiers objets. Car comme précédement, si on avait une solution meilleure $I'$ pour ce dernier problème, le sous-ensemble d'indices $I'$ serait une solution qui contredirait l'optimalité de la solution $I_{max}$.

On remarque donc que les solutions optimales du problème du sac à dos contiennent des solutions optimales pour le problème du sac à dos sur des instances plus petites.

### 3.2.3 Une récurrence pour trouver la valeur optimale que l'on peut transporter.

Soit $p = (p_1, ..., p_n)$ et $v = (v_1, ..., v_n)$ deux séquences de $n$ nombres entiers positifs. Nous notons pour tous $i \leq n$ et $C \geq 0$, $P_i(C)$ le problème de sac à dos dont les données initiales sont $C$ pour la contenance maximale du sac à dos et $(p_1, ..., p_i)$ et $(v_1, ..., v_i)$ pour les deux suites respectivement des poids et des valeurs des $i$ objets du problème. De plus, on note $I_i(C)$ un sous-ensemble d'indices qui permet d'obtenir la solution optimale du problème $P_i(C)$ et on note $M_i(C)$ la valeur maximale transportable $M_i(C) = \sum_{i \in I_i(C)} v_i$. On regarde maintenant la structure de $I_i(C)$ : soit $i \in I_i(C)$ et dans ce cas, $I_i(C)\backslash\{i\}$ est une solution optimale pour le problème $P_{i-1}(C - p_i)$, soit $i \notin I_i(C)$ et dans ce cas $I_i(C)\backslash\{i\}$ est une solution optimale pour $P_{i-1}(C)$. Comme on ne sait pas à l'avance si $i$ appartient ou non à $I_i(C)$, on est obligé de regarder quelle est la meilleure des deux solutions. On a donc que :

$$M_i(C) = \max\{M_{i-1}(C), M_{i-1}(C - p_i) + v_i\}.$$

### 3.2.4 Pseudo-code pour la valeur optimale

On va commencer par calculer le coût optimal par une approche itérative. L'entrée est deux tableaux $P[1..n]$ et $V[1..n]$ contenant respectivement les valeurs $p_1, ..., p_n$ et $v_1, ..., v_n$ et une valeur $C$ représentant la contenance maximale du sac à dos. La procédure utilise un tableau auxiliaire $M[0..n, 0..C]$ pour mémoriser les valeurs $M_i(C)$ et un tableau $X[1..n, 1..C]$ dont l'entrée $(i, c)$ contient un booléen qui indique si $i$ appartient

ou non à une solution optimale du problème $P_i(c)$.

**Algorithme 15 :** ValeurMax

**Entrées :** Deux tableaux d'entiers positifs $P[1..n]$ et $V[1..n]$ et $C$ un nb entier positif.

**Sorties :** Rien

```
1  ValeurMax(P, V, C)
2  n := longueur[P]
3  CréerTab(M, n)
4  pour c allant de 0 à C faire
5      M[0, c] := 0
6  pour i allant de 1 à n faire
7      M[i, 0] := 0
8  pour c allant de 1 à C faire
9      pour i allant de 1 à n faire
10         q1 := M[i − 1, c]
11         si c − P[i] < 0 alors
12             q2 := −1
13         sinon
14             q2 := M[i − 1, c − P[i]] + V[i]
15         si q1 < q2 alors
16             M[i, c] := q2
17             X[i, c] := vrai
18         sinon
19             M[i, c] := q1
20             X[i, c] := faux
```

**Analyse de l'algorithme ValeurMax :**

*Preuve de Terminaison :*
Evident, il n'y a que des boucles prédéfinies.

*Preuve de Validité :*
L'algorithme commence aux lignes 3-7 par l'initialisation des cas de base. S'il n'y a pas d'objet à prendre ($i = 0$), la valeur maximale transportable est évidement 0 quelque soit la contenance du sac. Donc $M[0, c] := 0$, pour $c = 0, 1, \ldots, C$. De même si le sac a une contenance nulle, on ne peut prendre aucun objet. Donc $M[i, 0] := 0$, pour $i = 1, \ldots, n$. On utilise ensuite lignes 8-20 la récurrence pour calculer $M[i, c]$ pour $i = 1, \ldots, n$ et $c = 1, ..., B$. On remarque que, quand l'algorithme en est au calcul de $M[i, c]$, il a déjà calculé les valeurs $M[i-1, c]$ et $M[i-1, c-p_i]$ si $i > 0$ et $c - p_i \geq 0$. Si $c - p_i < 0$, cela veut dire que l'objet de poids $p_i$ est trop lourd pour être rajouté dans le sac. Dans ce cas, on interdit alors la prise de cet objet ($M[i, c] = M[i-1, c]$ car $q_1 > q_2$ dans tous les cas). Quand on accède à la ligne 16, on a $M[i-1, c-p_i] > M[i-1, c]$. Ceci veut dire que l'on peut prendre le $i$-ième objet dans le sac pour construire une solution optimale.

*Analyse de la Complexité* en nombre de comparaisons :
Le nombre de comparaisons est $2nC$ (deux comparaisons à chaque passage dans la double boucle ligne 8-20). La procédure ValeurMax fait donc $\Theta(nC)$ comparaisons. L'algorithme nécessite un espace de stockage $\Theta(nC)$ pour le tableau $M$. ValeurMax est donc beaucoup plus efficace que la méthode en temps exponentiel consistant à énumérer tous les sous-ensembles d'indices possibles et à tester chacun d'eux. Néanmoins l'algorithme du sac à dos est ce que l'on appelle un algorithme pseudo-polynomial. En fait, la valeur $C$ peut être stockée en machine sur $\lceil \log_2 C \rceil$ bits et donc le nombre de comparaisons $nC$ est exponentiel par rapport à la taille mémoire pour stocker $C$. On peut montrer de plus que le problème du sac à dos est un problème NP-complet.

### 3.2.5 Construction d'une solution optimale

Bien que ValeurMax détermine la valeur maximale que peut contenir un sac à dos, elle ne rend pas le sous-ensemble d'indices qui permet d'obtenir la valeur maximale. Le tableau $X$ construit par ValeurMax peut servir à retrouver rapidement une solution optimale pour le problème $P_n(C)$. On commence tout simplement en $X[n, C]$ et on se déplace dans le tableau grâce aux booléens de $X$. Chaque fois que nous rencontrons $X[i, c] = faux$, on sait que $i$ n'appartient pas à une solution optimale. Dans ce cas, on sait qu'une solution optimale pour le problème $P_{i-1}(c)$ est une solution optimale pour le problème $P_i(c)$. Si l'on rencontre $X[i, c] = vrai$, alors une solution optimale pour le problème $P_{i-1}(c - P[i])$ peut être étendue à une solution optimale pour le problème $P_i(c)$ en lui rajoutant $i$.

Ceci permet d'écrire le pseudo-code récursif suivant :

**Algorithme 16 :** AfficherIndice

**Entrées :** les tableaux $X$ et $P$ et deux entiers $i$, $C$
**Sorties :** Rien mais affiche une solution optimale

```
1 AfficherIndice(X, P, i, C)
2 si i = 0 alors
3 |  Stop
4 si X[i, j] = faux alors
5 |  AfficherIndice(X, P, i − 1, C)
6 sinon
7 |  AfficherIndice(X, P, i − 1, C − P[i])
8 |  afficher(i)
```

Nous proposons maintenant une version récursive de l'algorithme. Elle est composée de deux parties, une partie initialisation (Mémorisation-ValeurMax) et une partie

construction récursive (Récupérer-ValeurMax).

**Algorithme 17 :** Mémorisation-ValeurMax

**Entrées :** Deux tableaux d'entiers positifs $P[1..n]$ et $V[1..n]$ et $C$ un nombre entier positif.

**Sorties :** La valeur optimale que l'on peut transporter.

```
1 Mémorisation-ValeurMax(P, V, C)
2 n := longueur[P]
3 Créer un tableau d'entiers M[0..n, 0..C]
4 Créer un tableau de booléens X[1..n, 1..C]
5 pour i allant de 1 à n faire
6 |   pour c allant de 1 à C faire
7 |   |   M[i, c] := −1
8 |   |   X[i, c] := faux
9 retourner Récupérer-ValeurMax(P, V, n, C)
```

**Algorithme 18 :** Récupérer-ValeurMax

**Entrées :** Deux tableaux d'entiers positifs $P[1..n]$ et $V[1..n]$ et deux nombres entiers positifs $i$ et $C$.

**Sorties :** Un entier indiquant la valeur maximale que peut transporter le sac de contenance $C$ pour des objets pris parmis les $i$ premiers

```
1 Récupérer-ValeurMax(P, V, i, C)
2 si M[i, C] ≥ 0 alors
3 |   retourner M[i, C]
4 sinon
5 |   si i = 0 ou C = 0 alors
6 |   |   M[i, j] := 0
7 |   sinon
8 |   |   q1 :=Récupérer-ValeurMax(P, V, i − 1, C)
9 |   |   si C − P[i] < 0 alors
10|   |   |   q2 := −1
11|   |   sinon
12|   |   |   q2 :=Récupérer-ValeurMax(P, V, i − 1, C − P[i]) + V[i]
13|   |   si q1 < q2 alors
14|   |   |   M[i, C] := q2
15|   |   |   X[i, C] := vrai
16|   |   sinon
17|   |   |   M[i, C] := q1
18|   |   |   X[i, C] := faux
19 retourner M[i, C]
```

**Analyse de l'algorithme :**

*Preuve de Terminaison :*

Il suffit de remarquer que l'on appelle récursivement Récupérer-ValeurMax sur des instances où $i$ a diminué de 1 et que l'algorithme se termine quand $i = 0$.

*Preuve de Validité :*
Immédiat par définition récursive de $M[i, C]$.
*Analyse de la Complexité* en nombre de comparaisons :
Il est facile de montrer que l'on fait moins de $nC$ comparaisons car on remplit au plus une fois chaque case du tableau $M$. Mais dans cette version récursive le tableau $M$ n'est pas nécessairement entièrement calculé. Le calcul de la complexité en moyenne est très ardu et sort du cadre de ce cours.