Diplôme Inter-Universitaire (DIU) "Enseigner l'informatique au lycée"

Nicolas Nisse
Université Côte d'Azur, Inria, CNRS, I3S, France
nicolas.nisse@inria.fr

17 juin 2020

Résumé

Ce document présente une quatrevingtaine d'exercices destinés auxTravaux dirigés et pratiques d'Algorithmique dispensés lors du block "Algorithmique" du DIU "Enseigner l'informatique au lycée". Au passage, nous expliquons (plus ou moins informellement) toutes les notions liées à ce cours. Ce document est principalement basé sur le cours que je dispense en option informatique de classe préparatoire MPSI. Des corrections très détaillées sont données en bleu. Des remarques que je crois importantes sont en rouge.

1 Avant propos

La difficulté estimée des exercices est indiquée par le nombre de $(*)$. Bien que ce "block" de cours ne soit pas sensé parler de **complexité temporelle et spatiale** des algorithmes (qui sera abordée au "block" suivant), il est difficile de ne pas aborder ces notions (parler de **Diviser pour régner** ou de **Programmation dynamique** n'a aucun sens sans les relier aux notions de complexités temporelle ou spatiale). Elles seront donc présentes en filigrane et de façon plus ou moins informelle (on parle d'"efficacité").
Voir `http://www-sop.inria.fr/members/Nicolas.Nisse/lectures/prepa/`

Prérequis fondamental : réviser le principe du raisonnement/preuve par récurrence.

Un message (que je crois) fondamental (en fait, LE message de ce document) est que l'informatique ne se limite pas à la programmation (un informaticien ne devrait pas uniquement "pisser du code"). L'informatique (et, en tout cas, l'algorithmique) est, de mon point de vue, une branche des mathématiques. Écrire un programme revient d'abord à penser à un algorithme, à être sûr qu'il est correct (preuve mathématique) et, si possible, à l'optimiser (complexité en temps et espace). Les enseignants qui ne seraient ici "que" pour trouver des exemples d'algorithmes les trouveront, mais je trouverais dommage que vous ne trouviez que ça. **Et surtout, par pitié, expliquez à vos élèves que l'informatique est bien plus que la programmation.**

2 Variables, boucles, récursion, et correction d'algorithmes

Remarque : Dans toutes les solutions proposées, on supposera (sans le vérifier dans le code) que les arguments en entrée des algorithmes sont conformes aux prérequis. Par exemple, si un algorithme

prend un entier n en entrée, on ne testera pas que n est bien un entier (ce que l'on pourrait/devrait faire, en ajoutant une exception dans le cas contraire).

2.1 Des exemples classiques

Les trois exemples suivants sont très similaires. Plus précisément, les exercices $1, 4, 6$ ($2, 5, 7$ respectivement) sont quasi identiques.

Remarque : Les preuves des algorithmes suivants paraissent presque triviales. Le but est ici d'initier à la preuve de correction d'algorithmes et de montrer, sur des exemples simples, comment prouver formellement (ce qui n'est pas trivial) que des algorithmes sont corrects.

2.1.1 Somme des n premiers entiers

Exercice 1 *Écrire un algorithme* **itératif** *qui prend un entier $n \in \mathbb{N}$ en entrée et calcule la somme $\sum_{i=0}^{n} i$ des n premiers entiers. Plus précisément, on utilisera une boucle "pour tout" en stockant le résultat intermédiaire dans une variable.*
Prouver la correction (terminaison + invariant de boucle) de votre algorithme.

Réponse :

Algorithm 1 (Somme itérative)

Entrée un entier n.
Sortie $\sum_{i=0}^{n} i$

```
def somIter(n) :
    somIntermediaire = 0
    for i in range(n + 1) :
        somIntermediaire = somIntermediaire + i
    return somIntermediaire
```

Prouver la correction d'un algorithme signifie prouver 2 choses : qu'il termine (**terminaison**) et que le résultat obtenu est celui qu'on veut (**validité**).

Terminaison : Pour prouver formellement qu'un algorithme itératif ($\approx$ avec des boucles) termine, il faut trouver un **convergent**, i.e., une grandeur qui décroît strictement et atteint une valeur à partir de laquelle l'algorithme termine (il faut dont que la grandeur prenne ses valeurs dans un ensemble muni d'un **ordre bien fondé**). Dans l'exemple, on considère la grandeur $\phi = n + 1 - i$ qui a pour valeur $n+1$ lors de la première itération de la boucle for, et qui décroît strictement de 1 à chaque itération puisque lors d'une itération il n'y a qu'un nombre fini d'opérations à effectuer (en l'occurrence, une addition) et que la boucle for incrémente implicitement de 1 la valeur de i (notons également que i n'est pas modifié par d'autres opérations). Enfin, lorsque $\phi = 0$ (i.e., $i = n + 1$), l'algorithme sort de la boucle et retourne la valeur courante de *somIntermediaire*.

En résumé, il y a $n + 1$ itérations de la boucle for (pour un algorithme aussi simple, il n'est généralement pas nécessaire de passer par un convergent) et chacune de ces itérations réalise un

nombre fini d'opérations (ici, une opération arithmétique). Donc l'algorithme termine après $O(n)$ opérations.

Validité : Dans un algorithme itératif, pour prouver que le résultat obtenu est bien celui attendu, on passe par un **invariant de boucle**, i.e., une propriété (relation) des variables et arguments en entrée de l'algorithme dont on va prouver (généralement par récurrence) qu'elle est vérifiée après chaque itération des boucles. De cette propriété, après la dernière itération, on déduit la validité du résultat. Il n'y a pas de recette magique pour déterminer l'invariant de boucle (ou les invariants), pour le trouver il faut comprendre comment fonctionne l'algorithme et ce qu'il veut calculer (généralement déterminer l'invariant de boucle est (bien) plus difficile que de le prouver).

Ici, l'invariant que l'on va prouver est que, après la k^{me} itération de la boucle for, $somIntermediaire = \sum_{j=0}^{k-1} j$. On prouve que cette propriété est vraie par récurrence sur le nombre $k \geq 1$ d'itérations. Remarquons qu'avant la première itération, $somIntermediaire = 0$.

La première itération ($k = 1$ et $i = 0$) ajoute 0 à $somIntermediaire$, donc après la première itération, $somIntermediaire = 0 + 0 = 0 = \sum_{j=0}^{k-1} j$ et donc la propriété est vérifiée pour $k = 1$.

Supposons, par récurrence, que la propriété est vraie au rang $k-1$, i.e., après la $(k-1)^{me}$ itération de la boucle for, $somIntermediaire = \sum_{j=0}^{k-2} j$. Lors de la k^{me} itération, la variable i vaut $k-1$, et cette itération ajoute i à $somIntermediaire$. Donc, $somIntermediaire$ valait $\sum_{j=0}^{k-2} j$ (après la $(k-1)^{me}$ itération, d'après l'hypothèse de récurrence) et on lui ajoute $i = k-1$. La nouvelle valeur de $somIntermediaire$ est donc $(\sum_{j=0}^{k-2} j) + k - 1 = \sum_{j=0}^{k-1} j$, ce qui est bien la propriété voulue.

Pour prouver la terminaison, nous avons déterminé qu'il y a $n+1$ itérations de la boucle for. D'après le paragraphe précédent, au sortir de la boucle (après $n+1$ itérations), la valeur de $somIntermediaire$ est $\sum_{j=0}^{n} j$, ce qui est bien le résultat désiré. □

Exercice 2 *Écrire un algorithme* **récursif** *qui prend un entier $n \in \mathbb{N}$ en entrée et calcule la somme $\sum_{i=0}^{n} i$ des n premiers entiers.*
Prouver la correction (terminaison + preuve par récurrence) de votre algorithme.

Réponse :

Algorithm 2 (Somme récursive)

Entrée un entier n.

Sortie $\sum_{i=0}^{n} i$

```
def somRec(n) :
    if n == 0 :
        return 0
    else :
        return somRec(n − 1) + n :
```

Prouver la correction d'un algorithme signifie prouver 2 choses : qu'il termine (**terminaison**) et que le résultat obtenu est celui qu'on veut (**validité**).

Terminaison : Pour prouver formellement qu'un algorithme récursif (qui s'appelle lui-même) termine, la preuve est par récurrence, ici sur n. Plus précisément, nous allons prouver, par récurrence sur n, que $somRec(n)$ termine après n additions. Supposons que $n = 0$ (cas de base), alors l'algorithme renvoie 0 sans aucune opération arithméthique, et donc la propriété est vraie pour $n = 0$.

Supposons, par récurrence, que $somRec(n-1)$ termine après $n-1$ additions. Pour $n > 0$, l'execution de $somRec(n)$ calcule d'abord $somRec(n-1)$ (qui termine après $n-1$ additions d'après l'hypothèse de récurrence), puis ajoute n au résultat. Cette exécution termine donc après $n-1$ plus 1 additions, i.e., après n additions.

Validité : Dans un algorithme récursif, pour prouver que le résultat obtenu est bien celui attendu, on fait (une fois n'est pas coutume) une preuve par récurrence. Plus précisément, nous allons prouver, par récurrence sur n, que $somRec(n)$ renvoie $\sum_{i=0}^{n} i$. Supposons que $n = 0$ (cas de base), alors l'algorithme renvoie $0 = \sum_{i=0}^{n} i = \sum_{i=0}^{0} i$, et donc la propriété est vraie pour $n = 0$.

Supposons, par récurrence sur $n > 0$, que $somRec(n-1)$ renvoie $\sum_{i=0}^{n-1} i$. Pour $n > 0$, l'execution de $somRec(n)$ calcule d'abord $somRec(n-1)$ (qui renvoie $\sum_{i=0}^{n-1} i$ d'après l'hypothèse de récurrence), puis ajoute n au résultat. Cette exécution renvoie donc $(\sum_{i=0}^{n-1} i) + n = \sum_{i=0}^{n} i$, qui est bien le résultat espéré. □

Remarque : Nous avons prouvé que les 2 algorithmes précédents s'exécutent en réalisant $O(n)$ opérations (ici additions). Ils ont donc une **complexité temporelle** ($\approx$ on compte le nombre d'opérations "importantes" réalisées) linéaire en l'entrée n. Ils diffèrent cependant en **complexité spatiale** (qui mesure la "place" nécessaire à l'exécution de l'algorithme). Précisement, l'algorithme itératif n'utilise que 2 variables entières (i et $somIntermediaire$), alors que l'algorithme récursif doit conserver en mémoire la **pile de récursion** (pour calculer $somRec(n)$, on doit calculer $somRec(n-1)$ qui doit calculer $somRec(n-2)$ $\cdots$ qui doit calculer $somRec(0)$) ce qui requiert de conserver $O(n)$ variables (implicites)). L'algorithme itératif a donc une complexité spatiale constante (en $O(1)$) alors que l'algorithme récursif a une complexité spatiale linéaire (en $O(n)$).

De manière générale (pour d'autres problèmes), il n'est pas possible de dire qui, d'un algorithme itératif ou récursif, est le "meilleur". Intuitivement (ça n'engage que moi et n'est pas vrai pour tout problème), un algorithme récursif sera plus facile à concevoir. Il s'agira ensuite de le "dérécursiver" (en faire un algorithme itératif) pour gagner en performance (notamment pour éviter des erreurs de type "stack overflow", qui signifient que la mémoire a explosé, généralement à cause de la pile de récursion). Voir l'exercice sur la suite de Fibonacci pour un exemple plus "explicite".

Exercice 3 (on anticipe un peu sur la complexité) *Écrire un algorithme qui prend un entier $n \in \mathbb{N}$ en entrée et calcule la somme $\sum_{i=0}^{n} i$ des n premiers entiers, et est bien "meilleur" (plus "efficace") que les algorithmes précédents.*
Pourquoi pensez vous que cet algorithme est "meilleur" ?

Réponse : Rappelons que $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$. Si vous n'êtes pas convaincu, nous allons le prouver par récurrence sur n (ici, c'est des maths, pas de la programmation). Pour le cas de base ($n = 0$), on a $0 = \sum_{i=0}^{0} i = \sum_{i=0}^{n} i = \frac{n(n+1)}{2} = \frac{0*1}{2} = 0$ et donc la propriété est vraie pour $n = 0$.

Supposons, par récurrence, que la propriété est vraie pour $n - 1 \geq 0$, i.e., $\sum_{i=0}^{n-1} i = \frac{(n-1)n}{2}$. Alors, $\sum_{i=0}^{n} i = (\sum_{i=0}^{n-1} i) + n = \frac{(n-1)n}{2} + n = \frac{n(n+1)}{2}$, et donc, la propriété est vraie pour tout $n > 0$.

La preuve précédente semble sortir du chapeau puisqu'elle suppose la connaissance préalable de la solution. Nous avons présenté cette preuve puisqu'elle sert notre propos en illustrant le principe de la preuve par récurrence. Par soucis de complétude, voici deux autres preuves (peut-être) plus intuitives.

Soit $M = [a_{i,j}]_{1 \leq i \leq j \leq n}$ la matrice $n \times n$ telle que tous les éléments sur et au dessus de la diagonale valent 1 et les autres éléments valent 0, i.e., $a_{i,j} = 1$ si $j \geq i$ et $a_{i,j} = 0$ sinon.

$$M = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 & 1 & 1 \\ 0 & 1 & 1 & \cdots & 1 & 1 & 1 \\ 0 & 0 & 1 & \cdots & 1 & 1 & 1 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 1 \\ 0 & 0 & \cdots & \cdots & 0 & 1 & 1 \\ 0 & 0 & 0 & \cdots & 0 & 0 & 1 \end{bmatrix}$$

Le nombre de 1 dans M est précisément $\sum_{i=0}^{n} i$ (en sommant ligne par ligne). En ayant une vision plus "globale" (et en supposant pour simplifier que n est pair), le nombre de 1 est la moitié des "cases" ($n^2/2$) plus la moitié de la diagonale ($n/2$), i.e., $n^2/2 + n/2 = n(n+1)/2$. On a donc bien l'identité annoncée.

Pour finir, rappelons la preuve géniale du jeune Gauss (selon la légende ?) : soient $(a_0, \cdots, a_n) = (0, 1, 2, \cdots, n)$ et $(b_0, \cdots, b_n) = (n, n-1, n-2, \cdots, 1, 0)$. Alors $(a_0 + b_0, a_1 + b_1, \cdots, a_n + b_n) = (n, n, \cdots, n)$. Donc $n * (n+1) = \sum_{i=0}^{n} (a_i + b_i) = \sum_{i=0}^{n} a_i + \sum_{i=0}^{n} b_i = 2 * \sum_{i=0}^{n} i$.

On déduit de cette relation mathématique l'algorithme suivant :

Algorithm 3 (Somme efficace des n premiers entiers)

Entrée un entier n.

Sortie $\sum_{i=0}^{n} i$

```
def somBest(n) :
    return n * (n + 1)/2
```

Cet algorithme effectue donc seulement 3 opérations arithmétiques (une multiplication, une addition et une division) et n'utilise qu'une place en mémoire (celle pour stocker le résultat). Il a donc une complexité temporelle en $O(1)$ et une complexité spatiale en $O(1)$. C'est dans ce sens qu'il est meilleur que les algorithmes précédents (plus rapide et pas plus gourmand en mémoire). □

2.1.2 Factorielle

Exercice 4 *Écrire un algorithme* **itératif** *qui prend un entier $n \in \mathbb{N}$ en entrée et calcule $n!$. Plus précisément, on utilisera une boucle "pour tout" en stockant le résultat intermédiaire dans une variable.*
Prouver la correction (terminaison + invariant de boucle) de votre algorithme.

Réponse :

Algorithm 4 (Factorielle itérative)

Entrée un entier $n > 0$.

Sortie $\prod_{i=1}^{n} i = !n$

```
def factIter(n) :
    prodIntermediaire = 1
    for i in range(1, n + 1) :
        prodIntermediaire = prodIntermediaire * i
    return prodIntermediaire
```

Prouver la correction d'un algorithme signifie prouver 2 choses : qu'il termine (**terminaison**) et que le résultat obtenu est celui qu'on veut (**validité**).

Terminaison : On considère la grandeur $\phi = n + 1 - i$ qui a pour valeur n lors de la première itération de la boucle for ($i = 1$), et qui décroît strictement de 1 à chaque itération puisque lors d'une itération il n'y a qu'un nombre fini d'opérations à effectuer (en l'occurrence, une multiplication) et que la boucle for incrémente implicitement de 1 la valeur de i (notons également que i n'est pas modifié par d'autres opérations). Enfin, lorsque $\phi = 0$ (i.e., $i = n+1$), l'algorithme sort de la boucle et retourne la valeur courante de $somIntermediaire$.

En résumé, il y a n itérations de la boucle for (pour un algorithme aussi simple, il n'est généralement pas nécessaire de passer par un convergent) et chacune de ces itérations réalise un nombre fini d'opérations (ici, une opération arithmétique). Donc l'algorithme termine après $O(n)$ opérations.

Validité : Nous prouvons que l'invariant de boucle est que, après la k^{me} itération de la boucle for, $prodIntermediaire = \prod_{j=1}^{k} j$. On prouve que cette propriété est vraie par récurrence sur le nombre $k \geq 1$ d'itérations. Remarquons qu'avant la première itération, $prodIntermediaire = 1$.

La première itération ($k = 1$) multiplie $prodIntermediaire$ par $i = k = 1$, donc après la première itération, $prodIntermediaire = 1 * 1 = 1 = \prod_{j=1}^{k} j$ et donc la propriété est vérifiée pour $k = 1$.

Supposons, par récurrence, que la propriété est vraie au rang $k-1$, i.e., après la $(k-1)^{me}$ itération de la boucle for, $prodIntermediaire = \prod_{j=1}^{k-1} j = !(k-1)$. Lors de la k^{me} itération, la variable i vaut k, et cette itération multiplie $somIntermediaire$ par $i = k$. Donc, $prodIntermediaire$ valait $\prod_{j=1}^{k-1} j$ (après la $(k-1)^{me}$ itération, d'après l'hypothèse de récurrence) et on la multiplie par $i = k$. La nouvelle valeur de $prodIntermediaire$ est donc $(\prod_{j=1}^{k-1} j) * k = \prod_{j=1}^{k} j = !k$, ce qui est bien la propriété voulue.

Pour prouver la terminaison, nous avons déterminé qu'il y a n itérations de la boucle for. D'après le paragraphe précédent, au sortir de la boucle (après n itérations), la valeur retournée de $prodIntermediaire$ est $\prod_{j=1}^{n} j = !n$, ce qui est bien le résultat désiré. □

Exercice 5 *Écrire un algorithme* **récursif** *qui prend un entier $n \in \mathbb{N}$ en entrée et calcule $n!$. Prouver la correction (terminaison + preuve par récurrence) de votre algorithme.*

Réponse :

Algorithm 5 (Factorielle récursive)

Entrée un entier $n > 0$.

Sortie $\prod_{i=1}^{n} i$

```
def factRec(n) :
    if n == 1 :
        return 1
    else :
        return factRec(n − 1) ∗ n :
```

Prouver la correction d'un algorithme signifie prouver 2 choses : qu'il termine (**terminaison**) et que le résultat obtenu est celui qu'on veut (**validité**).

Terminaison : Nous allons prouver, par récurrence sur $n > 0$, que $factRec(n)$ termine après $n-1$ multiplications. Supposons que $n = 1$ (cas de base), alors l'algorithme renvoie 1 sans aucune opération arithméthique, et donc la propriété est vraie pour $n = 1$.

Supposons, par récurrence, que $factRec(n-1)$ termine après $n-2$ multiplications. Pour $n > 1$, l'execution de $factRec(n)$ calcule d'abord $factRec(n-1)$ (qui termine après $n-2$ multiplications d'après l'hypothèse de récurrence), puis multiplie le résultat par n. Cette exécution termine donc après $n-2$ plus 1 multiplications, i.e., après $n-1$ multiplications.

Validité : Nous allons prouver, par récurrence sur $n > 0$, que $factRec(n)$ renvoie $\prod_{i=1}^{n} i$. Supposons que $n = 1$ (cas de base), alors l'algorithme renvoie $1 = \prod_{i=1}^{n} i = \prod_{i=1}^{1} i$, et donc la propriété est vraie pour $n = 1$.

Supposons, par récurrence sur $n > 1$, que $factRec(n-1)$ renvoie $\prod_{i=1}^{n-1} i$. Pour $n > 1$, l'execution de $factRec(n)$ calcule d'abord $factRec(n-1)$ (qui renvoie $\prod_{i=1}^{n-1} i$ d'après l'hypothèse de récurrence), puis multiplie le résultat par n. Cette exécution renvoie donc $(\prod_{i=1}^{n-1} i) * n = \prod_{i=1}^{n} i$, qui est bien le résultat espéré. □

2.1.3 Exponentiation

Dans cette section, on interdit l'utilisation de $**$ qui calcule directement la puissance en Python (l'idée est ici de comprendre comment la fonction $**$ est calculée).

Exercice 6 *Écrire un algorithme* **itératif** *qui prend un entier $n \in \mathbb{N}$ et un réel $x \in \mathbb{R}^+$ en entrées et calcule x^n. Plus précisément, on utilisera une boucle "pour tout" en stockant le résultat intermédiaire dans une variable.*
Prouver la correction (terminaison + invariant de boucle) de votre algorithme.

Réponse :

Algorithm 6 (Exponentiation itérative)

Entrée un entier n et un réel x.
Sortie x^n

```
def expIter(n) :
    if n == 0 :
        return 1
    else
        expIntermediaire = x
        for i in range(n − 1) :
            expIntermediaire = expIntermediaire ∗ x
        return expIntermediaire
```

Prouver la correction d'un algorithme signifie prouver 2 choses : qu'il termine (**terminaison**) et que le résultat obtenu est celui qu'on veut (**validité**).

Ici, on considère deux cas selon que $n = 0$ ou $n > 0$. Si $n = 0$, l'algorithme termine et renvoie la solution voulue (cf. ce qui ce produit dans le cas où $n = 0$). Sinon ($n > 0$), la preuve est "comme d'habitude".

Terminaison : On considère la grandeur $\phi = n - 1 - i$ qui a pour valeur $n - 1$ lors de la première itération de la boucle for ($i = 0$), et qui décroît strictement de 1 à chaque itération puisque lors d'une itération il n'y a qu'un nombre fini d'opérations à effectuer (en l'occurrence, une multiplication) et que la boucle for incrémente implicitement de 1 la valeur de i (notons également que i n'est pas modifié par d'autres opérations). Enfin, lorsque $\phi = 0$ (i.e., $i = n-1$), l'algorithme sort de la boucle et retourne la valeur courante de $expIntermediaire$.

En résumé, il y a $n - 1$ itérations de la boucle for (pour un algorithme aussi simple, il n'est généralement pas nécessaire de passer par un convergent) et chacune de ces itérations réalise un nombre fini d'opérations (ici, une opération arithmétique). Donc l'algorithme termine après $O(n)$ opérations.

Validité : Nous prouvons que l'invariant de boucle est que, après la k^{me} itération de la boucle for, $expIntermediaire = x^{k+1}$. On prouve que cette propriété est vraie par récurrence sur le nombre $k \geq 1$ d'itérations. Remarquons qu'avant la première itération, $prodIntermediaire = x$.

La première itération ($k = 1$) multiplie $prodIntermediaire$ par x, donc après la première itération, $prodIntermediaire = x * x = x^2$ et donc la propriété est vérifiée pour $k = 1$.

Supposons, par récurrence, que la propriété est vraie au rang $k-1$, i.e., après la $(k-1)^{me}$ itération de la boucle for, $expIntermediaire = x^k$. Lors de la k^{me} itération, la variable $expIntermediaire$ est multipliée par x. Donc, $expIntermediaire$ valait x^k (après la $(k-1)^{me}$ itération, d'après l'hypothèse de récurrence) et on la multiplie par x. La nouvelle valeur de $expIntermediaire$ est donc $x^k * x = x^{k+1}$, ce qui est bien la propriété voulue.

Pour prouver la terminaison, nous avons déterminé qu'il y a $n - 1$ itérations de la boucle for. D'après le paragraphe précédent, au sortir de la boucle (après $n - 1$ itérations), la valeur retournée de $prodIntermediaire$ est x^n, ce qui est bien le résultat désiré. □

Exercice 7 *Écrire un algorithme* **récursif** *qui prend un entier $n \in \mathbb{N}$ et un réel $x \in \mathbb{R}^+$ en entrées et calcule x^n.*
Prouver la correction (terminaison + preuve par récurrence) de votre algorithme.

Réponse :

Algorithm 7 (Exponentiation récursive)

Entrée un réel x et un entier n.

Sortie x^n

```
def expRec(n,x) :
    if n == 0 :
        return 1
    else :
        return expRec(n − 1, x) ∗ x :
```

Prouver la correction d'un algorithme signifie prouver 2 choses : qu'il termine (**terminaison**) et que le résultat obtenu est celui qu'on veut (**validité**).

Ici, on considère deux cas selon que $n = 0$ ou $n > 0$. Si $n = 0$, l'algorithme termine et renvoie la solution voulue (cf. $x^0 = 1$, ce qui ce produit dans le cas où $n = 0$). Sinon ($n > 0$), la preuve est "comme d'habitude".

Terminaison : Nous allons prouver, par récurrence sur $n > 0$, que $expRec(x, n)$ termine après n multiplications. Supposons que $n = 1$ (cas de base), alors l'algorithme renvoie $x * expRec(x, 0) = x * 1$ qui requiert une multiplication, et donc la propriété est vraie pour $n = 1$.

Supposons, par récurrence, que $expRec(x, n-1)$ termine après $n-1$ multiplications. Pour $n > 1$, l'execution de $expRec(x, n)$ calcule d'abord $expRec(x, n-1)$ (qui termine après $n-1$ multiplications d'après l'hypothèse de récurrence), puis multiplie le résultat par x. Cette exécution termine donc après $n - 1$ plus 1 multiplications, i.e., après n multiplications.

Validité : Nous allons prouver, par récurrence sur $n > 0$, que $expRec(x, n)$ renvoie x^n. Supposons que $n = 1$ (cas de base), alors l'algorithme renvoie x, et donc la propriété est vraie pour $n = 1$.

Supposons, par récurrence sur $n \geq 1$, que $expRec(x, n-1)$ renvoie x^{n-1}. Pour $n > 1$, l'execution de $expRec(x, n)$ calcule d'abord $expRec(x, n-1)$ (qui renvoie x^{n-1} d'après l'hypothèse de récurrence), puis multiplie le résultat par x. Cette exécution renvoie donc $x^{n-1} * x = x^n$, qui est bien le résultat espéré. □

L'exercice suivant est un exemple d'algorithme suivant le paradigme **Diviser pour régner**. Plus précisément, ce paradigme d'algorithmique consiste à, pour résoudre un problème, le "diviser" en un ou des problèmes "plus petits" que l'on peut résoudre indépendamment, puis combiner leurs solutions pour obtenir une solution du problème initial. En étudiant ce type d'algorithme, on espère concevoir un algorithme plus efficace en terme de complexité temporelle ($\approx$ on veut calculer la même chose en effectuant moins "d'opérations").

Notons qu'ici, pour calculer x^n, on va se ramener à un unique sous-problème (de "taille" la moitié de la "taille" du problème initial) : le calcul de $x^{\lfloor \frac{n}{2} \rfloor}$.

Exercice 8 (*) (Exponentiation rapide) (on anticipe beaucoup sur la complexité) *En remarquant que* $x^n = (x^{\lfloor \frac{n}{2} \rfloor})^2$ *si* n *est pair et* $x^n = x * (x^{\lfloor \frac{n}{2} \rfloor})^2$ *sinon, écrire un algorithme qui calcule* x^n *plus "efficacement" que les deux algorithmes précédents.*

// $\lfloor k \rfloor \in \mathbb{N}$ *est la partie entière inférieure de* $k \in \mathbb{R}^+$.

Réponse : Prouvons d'abord la remarque de l'exercice. Si n est pair, i.e, $n = 2k$, alors $x^n = x^{2k} = (x^k)^2 = (x^{\lfloor \frac{n}{2} \rfloor})^2$. Sinon, $n = 2k + 1$, et $x^n = x^{2k+1} = x * (x^k)^2 = x * (x^{\lfloor \frac{n}{2} \rfloor})^2$. Cela permet de prouver (formellement par récurrence) que l'algorithme suivant renvoie le résultat voulu.

Algorithm 8 (Exponentiation rapide)

Entrée un réel x et un entier n.

Sortie x^n

```
def expRap(n,x) :
    if n == 0 :
        return 1
    else :
        temp = expRap((n//2),x)    (a//b : quotient de la division euclidienne de a par b)
        if (n%2) == 0 :                (Cas n est pair (% est la fonction "modulo"))
            return temp*temp
        else :                                                  (Cas n est impair)
            return x*temp*temp
```

Maintenant que vous savez prouver la correction d'un algorithme, nous insistons sur la réelle (?) difficulté, prouver sa complexité temporelle. Contrairement aux exemples précédents, la complexité temporelle (ici, le nombre de multiplications) de cet algorithme n'est pas simplement déterminée par l'entier n en entrée. Si n est pair, il faudra une multiplication ($temp * temp$) plus le nombre de multiplications pour calculer $temp = expRap(n/2, x) = expRap(\lfloor n/2 \rfloor, x)$, sinon, il faudra 2 multiplications ($x * temp * temp$) plus le nombre de multiplications pour $temp = expRap(\lfloor n/2 \rfloor, x)$.

Ici, on considère le **pire cas**, c'est-à-dire qu'on étudie le nombre maximum $c(n)$ de multiplications qu'il faut effectuer (dans le pire cas) pour calculer $expRap(n, x)$. D'après le paragraphe précédent, on a donc $c(n) = 2 + c(\lfloor n/2 \rfloor)$ et $c(0) = O(1)$.

Pour évaluer $c(n)$, on considère d'abord le cas où $n = 2^p$ (et donc $p = \log_2 n$). En posant $u_p = c(2^p)$, on voit que $u_p = c(2^p) = 2 + c(2^{p-1}) = 2 + u_{p-1}$ et donc u_p est une suite arithmétique et donc $u_p = O(p)$. D'où $c(n) = c(2^p) = u_p = O(p) = O(\log n)$.

Pour n quelconque, on pose $p \in \mathbb{N}$ tel que $2^p \leq n < 2^{p+1}$. En remarquant que $c(n)$ est croissant en n, on a donc $c(2^p) \leq c(n) \leq c(2^{p+1})$. Comme, d'après le paragraphe précédent, $c(2^p) \sim c(2^{p+1}) \sim O(p) = O(\log n)$, on en déduit que $c(n) = O(\log n)$.

L'algorithme d'exponentiation rapide calcule donc x^n en effectuant seulement un nombre logarithmique (en n) de multiplications. □

2.1.4 D'autres exemples

L'exercice 9 suit le même schéma que les précédents. L'exercice 10 un peu moins... Plus précisément, l'exercice 9 est une simple modification du problème de la somme des n premiers entiers.

Exercice 9 *Écrire un algorithme qui prend un entier $n \in \mathbb{N}$ en entrée et calcule* $\sum_{i=0}^{n} \sin^2(i)$.

On supposera que la fonction sin *de Python peut être utilisée.*

Réponse : Nous donnons deux réponses possibles.

Algorithm 9 (Somme sinus carré itérative)

Entrée un entier n.

Sortie $\sum_{i=0}^{n} \sin^2(i)$

```
def somIter(n) :
    somIntermediaire = 0
    for i in range(n + 1) :
        somIntermediaire = somIntermediaire + sin(i) * *2
    return somIntermediaire
```

Terminaison : convergent $n+1-i$, $n+1$ itérations de la boucle for qui chacune calcule un sinus, fait une multiplication et une addition.
Correction : invariant de boucle : après la k^{me} itération de la boucle for, $somIntermediaire = \sum_{j=0}^{k-1} \sin^2(j)$.

Algorithm 10 (Somme sinus carré récursive)

Entrée un entier n.

Sortie $\sum_{i=0}^{n} \sin^2(i)$

```
def somRec(n) :
    if n == 0 :
        return 0
    else :
        return somRec(n - 1) + sin(n) * *2 :
```

Preuve par récurrence sur n. □

L'exercice suivant vise à donner aux élèves une meilleure intuition de ce qu'est un logarithme.

Exercice 10 *Écrire un algorithme qui prend un entier $n \in \mathbb{N}^* \setminus \{1\}$ en entrée et calcule $\lfloor \log_2(n) \rfloor$.*
// il est bien sur interdit d'utiliser la fonction log *de Python*

Réponse :

Algorithm 11 (Logarithme en base 2)

Entrée un entier $n > 1$.

Sortie $\lfloor \log_2(n) \rfloor$

```
def logBase2(n) :
    res = 0
    aux = 1
    while aux ≤ n :
        res+ = 1
        aux = aux ∗ 2
    return res − 1
```

Terminaison : convergent $n - aux$, aux est initialisé à 1 et multiplié par 2 à chaque itération. Donc $n - aux$ est strictement décroissant et la boucle while termine lorsque $n - aux < 0$. Remarquons que le nombre d'itérations est $\lceil \log_2(n) \rceil$.
Correction : invariant de boucle : après la k^{me} itération de la boucle *while*, $res = k$, $aux = 2^k$ et $2^{k-1} \leq n$ (preuve par récurrence). Supposons que la condition de la boucle *while* devient fausse, i.e., $aux > n$, à la i^{me} itération (donc les valeurs des variables sont celles obtenues après l'itération $i - 1$). D'après l'invariant de boucle, on a $res = i - 1$, $aux = 2^{i-1}$ et $2^{i-2} \leq n$. Pour résumer, $2^{i-2} \leq n < aux = 2^{i-1}$. Donc $i - 2 \leq \log_2 n < i - 1$ (puisque $\log_2$ est une fonction croissante). L'algorithme retourne $res - 1 = i - 2 = \lfloor \log_2(n) \rfloor$.

Plus généralement :

Algorithm 12 (Logarithme en base b)

Entrée deux réels $x, b > 1$.

Sortie $\lfloor \log_b(x) \rfloor$

```
def logBaseb(x,b) :
    res = 0
    aux = 1
    while aux ≤ n :
        res+ = 1
        aux = aux ∗ b
    return res − 1
```

□

2.2 Étude de suites récursives

Cette section (probablement au delà du programme) est destinée à illustrer le lien entre algorithme récursif et suite définie par récurrence.

2.2.1 Suite de Fibonacci

On rappelle que la suite $(f_n)_{n\in\mathbb{N}}$ de Fibonacci est définie par $f_0 = 0$, $f_1 = 1$ et $f_n = f_{n-1} + f_{n-2}$ pour tout $n \geq 2$. (parfois, la suite est définie par $f_0 = 1$, $f_1 = 1$ et $f_n = f_{n-1} + f_{n-2}$, ce qui ne change quasiment rien excepté un décalage de 1 des indices).

Exercice 11 *Écrire un algorithme* **récursif** *qui prend un entier $n \in \mathbb{N}$ en entrée et calcule f_n. Prouver la correction de votre algorithme.*

Réponse :

Algorithm 13 (Suite de Fibonacci)

Entrée un entier n.

Sortie f_n

```
def Fibo(n) :
    if n <= 1 :
        if n == 0 :
            return 0
        else :
            return 1
    else :
        return Fibo(n − 1) + Fibo(n − 2) :
```

On prouve, par une récurrence forte (immédiate) sur n, que $Fibo(n)$ termine et retourne f_n. C'est clairement vrai pour $n \leq 1$. Soit $n > 1$ et supposons que c'est vrai pour tout $k < n$, alors $Fibo(n)$ calcule $Fibo(n-1)$ et $Fibo(n-2)$ qui, d'après l'hypothèse de récurrence, terminent et retournent respectivement f_{n-1} et f_{n-2}. $Fibo(n)$ retourne alors $Fibo(n-1) + Fibo(n-2) = f_{n-1} + f_{n-2} = f_n$. □

Remarque : "Normalement" (sauf si vous êtes très fort et/ou que vous avez déjà appris ça), l'algorithme que vous avez proposé à la question précédente demande beaucoup de temps pour terminer (même pour $n = 40$ ou 50) car il fait beaucoup trop de calculs (en fait, il répète les mêmes calculs plusieurs fois et la pile de récursion est très "lourde" en mémoire). La figure 1 essaie de représenter visuellement cet inconvénient en illustrant les appels récursifs effectués lors de l'appel de la fonction $Fibo(6)$. Remarquons (preuve par récurrence sur n) que l'appel de $Fibo(n)$ entraîne f_n appels récursifs. Comme on le verra à l'exercice 13, f_n est exponentiel en n, les complexités spatiale et temporelle de l'algorithme 13 sont donc très mauvaises.

Pour pallier ce problème, il faut appliquer le principe de **mémoïsation** qui demande de stocker les calculs intermédiaires pour éviter de les répéter.

Exercice 12 (*) (où on anticipe un peu sur les tableaux) *Écrire un algorithme* **récursif** *qui prend un entier $n \in \mathbb{N}$ en entrée et calcule f_n plus efficacement que le précédent algorithme. Prouver la correction de votre algorithme.*

Réponse :

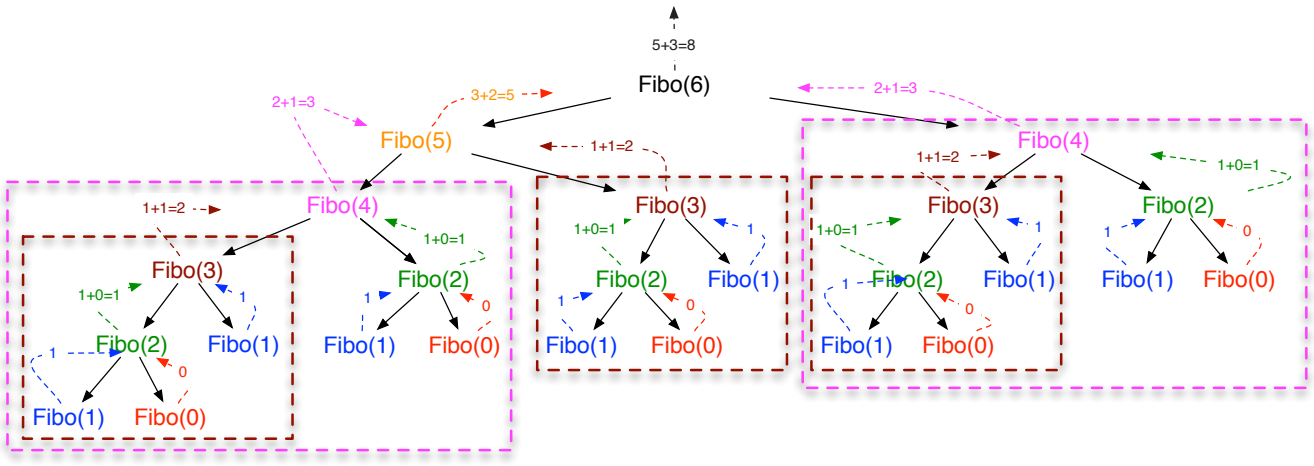


FIGURE 1 – Représentation de la pile d'exécution de l'algorithme 13 lors de l'appel de $Fibo(6)$. Les flèches en pointillés (vers le haut) représentent les valeurs retournées. Remarquons que $Fibo(0)$ (en rouge) est calculé 5 fois, $Fibo(1)$ (en bleu) est calculé 8 fois, $Fibo(2)$ (en vert) est calculé 5 fois, $Fibo(3)$ (carrés en pointillés marron) est calculé 3 fois, $Fibo(4)$ (carrés en pointillés violet) est calculé 2 fois.

Algorithm 14 (Suite de Fibonacci avec mémoïsation)

Entrée un entier n.

Sortie f_n

```
def FiboMem(n) :
    if n == 0 :
        return 0
    else :
        tab = [0 for i in range(n+1)]
        // création d'un tableau de longueur n+1, initialisé à 0, destiné à ce que tab[i] = f_i
        tab[1]=1
        for i in range(2, n + 1) :
            tab[i] = tab[i − 1] + tab[i − 2]
        return tab[n]
```

Si $n = 0$, la correction est évidente. Supposons que $n > 0$.

Terminaison : $n - 1$ itérations de la boucle for dont chacune fait une addition.

Donc la complexité temporelle est $O(n)$. De plus, en mémoire, on ne stocke qu'un tableau de longueur $O(n)$, donc la complexité spatiale est $O(n)$.

Validité : Preuve par récurrence forte sur le nombre d'itérations i de la boucle for qu'après la i^{me} itération, $tab[j] = f_j$ pour tout $j \leq i + 1$. □

L'exercice suivant est carrément hors programme, mais c'est, je crois, intéressant à connaître.

Exercice 13 (***) *Donner une formule close pour f_n (indice : ça parle du nombre d'or). En déduire un algorithme plus efficace que les précédents pour calculer f_n.*

Réponse :

La formule de Binet dit que $f_n = \frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n)$ (notons que $\frac{1+\sqrt{5}}{2}$ est le nombre d'or et que l'identité précédente peut être prouvée par récurrence sur n). Cette formule permet donc d'obtenir un algorithme plus efficace pour calculer f_n (puisque cet algorithme utilise "seulement" des puissances de n qui, en utilisant, par exemple, l'algorithme d'exponentiation rapide vu plus haut, ont une complexité temporelle $O(\log n)$).

Comme dit plus haut, la formule de Binet peut être prouvée par récurrence sur n. Nous donnons ici rapidement une preuve plus "intéressante". Notons $u(a,b)$ la "suite de Fibonacci" de premiers termes a et b, c-à-d, $u(a,b)_0 = a$, $u(a,b)_1 = b$ et $u(a,b)_n = u(a,b)_{n-1} + u(a,b)_{n-2}$ pour tout $n \geq 2$. Au détriment d'un petit décalage d'indice par rapport à la définition précédente, $u(1,1)$ est la suite de Fibonacci "classique". L'ensemble $\mathcal{F} = \{u(a,b) \mid a,b \in \mathbb{R}\}$ est un $\mathbb{R}$-espace vectoriel en posant $x \cdot u(a,b) = u(xa,xb)$ pour tout $x \in \mathbb{R}$, et $u(a,b) + u(c,d) = u(a+c,b+d)$. Le nombre d'or $\phi = \frac{1+\sqrt{5}}{2}$ est la solution positive de l'équation $\phi^2 = \phi + 1$, et $\bar{\phi} = \frac{1-\sqrt{5}}{2}$ est la racine négative de ce polynôme. Il est facile de montrer que $u(1,\phi)_n = \phi^n$ et que $u(1,\bar{\phi})_n = \bar{\phi}^n$ pour tout $n \geq 0$ (suite géométrique). Finalement, en résolvant un système de deux équations à deux inconnues, $u(1,1) = \frac{1}{\sqrt{5}}u(1,\phi) - \frac{1}{\sqrt{5}}u(1,\bar{\phi})$.

Algorithm 15 (Suite de Fibonacci avec formule de Binet)

Entrée un entier n.

Sortie f_n

```
def FiboBinet(n) :
    racine5 = 5**(1/2)
    phi = (1 + racine5)/2
    phi' = (1 - racine5)/2
    return ((phi**n) - (phi'**n))/racine5
```

Comparez les résultats obtenus par les fonctions $FiboMem(n)$ et $FiboBinet(n)$. On observe une différence qui est due aux erreurs d'arrondis inhérentes aux calculs de puissances (de réels codés comme des $float$) effectués par l'algorithme $FiboBinet(n)$. □

2.2.2 Tours de Hanoï

Dans ce jeu, on dispose de 3 piquets notés A, B et C. Il y a également $n \in \mathbb{N}$ disques percés, tous de diamètres différents. Initialement, tous les disques sont sur le piquet A, rangés par diamètre croissant (le plus grand disque est en dessous, et le plus petit disque au dessus).

Le but est de déplacer tous les disques sur le piquet C avec les contraintes suivantes.

Chaque mouvement consiste à choisir 2 piquets, disons X et Y, prendre le disque le plus haut sur le piquet X, et le déplacer au sommet du piquet Y (on ne déplace qu'un seul disque à la fois).

On impose de plus la contrainte qu'on ne peut jamais placer un disque sur un disque de diamètre inférieur.

Exercice 14 (**) *Écrire un algorithme qui prend un entier $n \in \mathbb{N}$ en entrée et calcule le nombre minimum de mouvements pour résoudre le problème des tours de Hanoï avec n disques.*
Prouver la correction de votre algorithme.

Réponse : Nous prouvons ci-dessous que le nombre minimum de mouvements pour résoudre le problème des tours de Hanoï avec n disques est $2^n - 1$. Nous donnons un algorithme récursif qui "colle" avec la stratégie optimale décrite ci-dessous.

Algorithm 16 (Meilleure stratégie pour le problème des tours de Hanoï)

Entrée un entier $n > 0$.

Sortie Nombre minimum $2^n - 1$ de mouvements pour résoudre le problème des tours de Hanoï avec n disques.

```
def Hanoi(n) :
    if n == 1 :
        return 1
    else :
        return 2 * Hanoi(n − 1) + 1
```

Correction : Prouvons par récurrence sur n que l'algorithme précédent termine et retourne $2^n - 1$. En effet, $Hanoi(1)$ retourne 1 et, si $n > 1$, $Hanoi(n-1)$ retourne (par récurrence) $2^{n-1} - 1$ et donc $Hanoi(n)$ retourne $2 * Hanoi(n-1) - 1 = 2 * (2^{n-1} - 1) + 1 = 2^n - 1$.

Prouvons maintenant que le nombre minimum de mouvements pour résoudre le problème des tours de Hanoï avec n disques est $2^n - 1$. La preuve est par récurrence sur n. Pour $n = 1$, le résultat est évident puisqu'il suffit d'un mouvement (déplacer l'unique disque du piquet A vers le piquet C et ce mouvement est bien sûr nécessaire). Pour $n > 1$, une stratégie consiste en déplacer la tour des $n-1$ plus petits disques du piquet A au piquet B en $2^{n-1} - 1$ mouvements (possible par induction), puis déplacer le plus grand disque du piquet A au piquet C (un mouvement) et enfin déplacer la tour des $n-1$ plus petits disques du piquet B au piquet C en $2^{n-1} - 1$ mouvements (possible par induction). Cette stratégie récursive utilise $2 * (2^{n-1} - 1) + 1 = 2^n - 1$ mouvements et est représentée sur la figure 2.

Pour montrer (sans trop de détails) que $2^n - 1$ mouvements sont nécessaires, considérons n'importe quelle stratégie et considérons son dernier mouvement qui amène le plus grand disque vers le piquet C (mouvement tel qu'on ne déplace plus jamais le plus grand disque). Avant ce mouvement, il a fallu déplacer la tour des $n-1$ plus petits disques vers un autre piquet (ce qui, par l'hypothèse de récurrence, a requis au moins $2^{n-1} - 1$ mouvements). Après ce mouvement, il faut déplacer la tour des $n-1$ plus petits disques vers le piquet C (ce qui, par l'hypothèse de récurrence, requiert au moins $2^{n-1} - 1$ mouvements). Toute stratégie utilise donc au moins $2 * (2^{n-1} - 1) + 1 = 2^n - 1$ mouvements. □

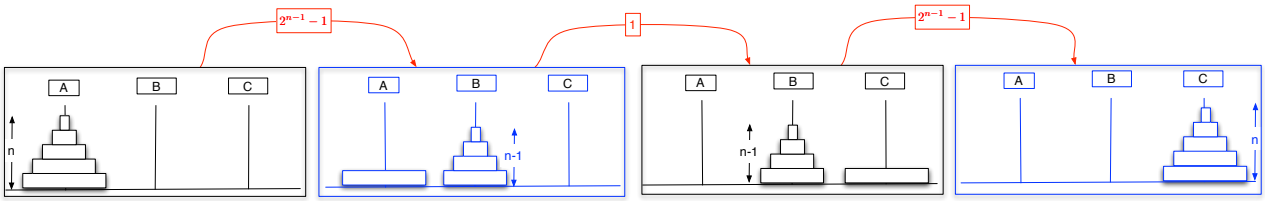


FIGURE 2 – Illustration d'une stratégie optimale pour résoudre le problème des tours de Hanoï.

2.3 Algorithmes surprise

Algorithm 17 (? ?)

Entrée deux entiers a et b, $a \geq b$.

Sortie ? ?

1. — Soit la variable x initialisée à a
   — Soit la variable y initialisée à b
2. Tant que $y > 0$ faire
   — Soit la variable *temp* initialisée à y
   — faire $y \leftarrow x \mod y$. // *étudier les spécifications de la fonction* mod *de Python*
   — faire $x \leftarrow temp$.
3. Renvoyer x.

Exercice 15 (**) *Que calcule l'algorithme précédent ? Prouvez le (terminaison + invariant de boucle).*

Réponse : Cet algorithme est celui d'Euclide qui détermine le plus grand commun diviseur $pgcd(a, b)$ de a et b (vous avez surement deviné ce résultat par vos tests successifs et/ou en étudiant le comportement de l'algorithme).
Terminaison : un convergent est y dont la valeur (entière) décroit strictement à chaque itération (puisque $0 \leq x \mod y < y$ pour tous entiers x, y). Donc, il y a au plus $O(b)$ itérations de la boucle *while*.
Validité : Prouvons, par récurrence sur $i \geq 1$, qu'avant la i^{me} itération de la boucle *while*, $pgcd(x, y) = pgcd(a, b)$. C'est clairement le cas pour $i = 1$ puisqu'avant la première itération, $a = x$ et $b = y$. Soient (a_i, b_i) les valeurs des variables a et b avant la $(i + 1)^{me}$ itération de la boucle while pour tout $i \geq 1$. Par récurrence, $pgcd(a_i, b_i) = pgcd(x, y)$. Comme $pgcd(a_i, b_i) = pgcd(b_i, a_i \mod b_i)$ (en effet, si $a = ka', b = kb'$ et $ka' = a = qb + r = qkb' + r$ (avec $r < b$), alors $r = k(a' - qb')$ est bien divisible par k, et on peut montrer de la même façon que tout diviseur commun de a et r est aussi un diviseur de b), on en déduit que $pgcd(x, y) = pgcd(a_i, b_i) = pgcd(a_{i+1}, b_{i+1})$ où $a_{i+1} = b_i$ et $b_{i+1} = a_i \mod b_i$ sont les nouvelles valeurs des variables a et b avant la $(i + 2)^{me}$ itération de la boucle *while*.

Pour conclure, l'algorithme se termine lors de l'itération i telle que $b_i = 0$ et renvoie $a_i = pgcd(a_i, 0) = pgcd(a_i, b_i) = pgcd(a, b)$ (la dernière égalité vient de l'invariant de boucle). □

Algorithm 18 (Fonction f [McCarthy 1970])

Paramètres 3 entiers a, b et c.

Entrée un entier $n > 0$.

Sortie ??

1. Si $n > a$
   Renvoyer $n - b$
2. Sinon
   Renvoyer $f(f(n + c))$.

Exercice 16 *Cet algorithme termine-t-il si $a = 100$, $b = 10$ et $c = 11$? Pour quelles valeurs des paramètres a, b, c l'algorithme termine-t-il ?*

Réponse : Par récurrence ("inversée", i.e., sur $a - n$ si $n \leq a$), si $a = 100$, $b = 10$ et $c = 11$, retourne $n - 10$ si $n > 100$ et 91 sinon, et donc l'algorithme termine pour ce choix des paramètres. La seconde question est laissée à la sagacité du lecteur. □

Algorithm 19 (Collatz/Syracuse)

Entrée un entier $n > 0$.

Sortie ??

1. Soit la variable x initialisée à n
2. Tant que $x \neq 1$ faire
   — Si x est pair, faire $x \leftarrow x/2$ *// on peut utiliser la fonction* mod *(ou %) de Python*
   — Sinon, faire $x \leftarrow 3 * x + 1$.
3. Renvoyer x.

Exercice 17 (Question fourbe) *Testez cet algorithme (observez la séquence des valeurs de x). Que retourne cet algorithme ? Cet algorithme est-il correct ?*

Réponse : Jusqu'à présent, les preuves de terminaison ont été (presque) triviales. Cet exercice veut montrer que c'est cependant nécessaire/important de s'en préoccuper.

Si vous avez codé l'algorithme précédent et fait des tests, vous avez (presque sûrement) vu que l'algorithme termine toujours (quel que soit l'entier n en entrée) et retourne 1. C'est un problème ouvert de recherche (conjecture de Collatz/Syracuse) de savoir si c'est toujours le cas (pour tout entier n). En d'autres termes, personne (jusqu'à présent) n'a pu prouver que cet algorithme termine dans tous les cas. Ainsi, je ne sais pas si cet algorithme est correct (existe t'il un entier n tel que cet algorithme appliqué à n ne termine pas (i.e., ne "tombe" jamais sur 1) ? ? ?). □

3 La même chose, mais avec des tableaux

Dans cette section, on présente différentes façons de travailler avec des tableaux, mais également des exemples d'utilisations des tableaux (à quoi ça sert ?).

Rappelons que dans un tableau de n éléments, ceux-ci sont indicés de 0 à $n-1$!! En effet, le premier élément d'un tableau de longueur n est celui d'indice 0, et son dernier élément est celui d'indice $n-1$.

Dans cette section, nous ne rappellerons pas systématiquement de prouver la correction des algorithmes, mais il est important que vous vous posiez toujours cette question (et y répondiez).

3.1 Manipulons des tableaux

Exercice 18 (Échange) *Écrire un algorithme Echange qui prend en entrées un tableau tab de longueur n et deux entiers $0 \leq i \leq j < n$, et échange dans tab les éléments d'indice i et j. Par exemple,* [2] *Echange*$([7,9,2,7,4,6],1,4)$ *doit modifier le tableau en* $[7,4,2,7,9,6]$.

// Notons que l'algorithme peut ne RIEN retourner.

Réponse :

Algorithm 20 (Echange)

Entrée un tableau *tab* et deux indices $0 \leq i, j < len(tab)$.

Sortie Rien

```
def Echange(tab,i,j) :
    aux = tab[i]
    tab[i]=tab[j]
    tab[j]=aux
```

□

Remarque : En Python, il est possible de répondre à la question sans variable auxiliaire. Ce n'est cependant pas possible dans beaucoup de langages de programmation et il est important de comprendre cette distinction. De mon point de vue, il faut éluder cette possibilité proposée par Python (d'ailleurs, je n'en donne pas l'implémentation) puisque comprendre l'intérêt d'utiliser une variable auxiliaire pour la question précédente est pédagogiquement fondamental.

Exercice 19 (Recherche de maximum) *Écrire un algorithme qui prend en entrée un tableau d'entiers* **distincts** *et retourne l'indice et la valeur de l'entier maximum contenu dans le tableau.*

Réponse :

Algorithm 21 (IndiceMax)

Entrée un tableau *tab* d'entiers distincts.

Sortie indice et valeur de l'entier maximum contenu dans le tableau

```
def IndiceMax(tab) :
    indice_courant = 0
    max_courant = tab[0]
    for i in range(len(tab)) :
        if tab[i] > max_courant :
            indice_courant = i
            max_courant = tab[i]
    return indice_courant, max_courant
```

Terminaison : $len(tab)$ itérations d'une boucle for avec un nombre fini d'opérations.

Validité : Invariant de boucle : Après la i^{me} itération $indice\_courant, max\_courant$ sont respectivement l'indice et la valeur de l'entier maximum se trouvant entre les indices 0 et i dans *tab*. □

Exercice 20 (Recherche séquentielle dans un tableau) *Écrire un algorithme qui prend en entrées un tableau tab et un élément x et qui retourne le plus petit indice i tel que $tab[i] = x$, et retourne -1 si le tableau ne contient pas la valeur x.*

Réponse :

Algorithm 22 (Recherche séquentielle)

Entrée un tableau *tab* et un élément x.

Sortie le plus petit indice i tel que $tab[i] = x$, et retourne -1 si le tableau ne contient pas la valeur x

```
def RechSeq(tab,x) :
    compteur = 0
    n = len(tab)
    while compteur < n and tab[compteur] ≠ x :
        compteur += 1
    If compteur == n :
        return -1
    else :
        return compteur
```

Terminaison : au plus n itérations de la boucle *while*.

Validité : Invariant de boucle : Après la i^{me} itération, l'élément x n'occupe aucune position d'indice $\leq i$ dans *tab*. □

La question suivante vise à formaliser l'algorithme que nous utilisons tous pour chercher un mot dans un dictionnaire (du moins pour les élèves qui se souviennent ce qu'est un dictionnaire :( ). C'est aussi un exemple d'algorithme du type **Diviser pour régner**.

Exercice 21 (*) (Recherche dichotomique dans un tableau trié) *Supposons maintenant que le tableau tab en entrée est* **trié** *par ordre croissant, i.e., $tab[i] \leq tab[i+1]$ pour tout $0 \leq i < n-1$ avec n la longueur du tableau.*

Écrire un algorithme (plus "efficace" que le précédent) qui prend en entrées un tableau tab et un élément x et qui retourne le plus petit indice i tel que $tab[i] = x$, et retourne -1 si le tableau ne contient pas la valeur x.

Réponse :

Algorithm 23 (Recherche Dichotomique, fonction auxiliaire)

Entrée un tableau *tab* trié, un élément x et deux indices $0 \leq a \leq b < len(tab)$.

Sortie le plus petit indice $a \leq i \leq b$ tel que $tab[i] = x$, et retourne -1 si le tableau ne contient pas la valeur x entre les indices a et b.

```
def RechDichAux(tab,x,a,b) :
    if a == b :
        if tab[a] == x :
            return a
        else
            return -1
    else :
        if x ≤ tab[(a + b)/2] :
            return RechDichAux(tab, x, a, (a + b)//2)
        else
            return RechDichAux(tab, x, (a + b)//2 + 1, b)
```

Preuves (de terminaison et validité) par récurrence sur $b - a$.

Algorithm 24 (Recherche Dichotomique)

Entrée un tableau *tab* trié, un élément x.

Sortie le plus petit indice $0 \leq i < len(tab)$ tel que $tab[i] = x$, et retourne -1 si le tableau ne contient pas la valeur x.

```
def RechDich(tab,x) :
    return RechDichAux(tab, x, 0, len(tab))
```

□

La question précédente veut démontrer d'une part l'intérêt de trier *a priori* des données pour une recherche plus efficace (imaginez un dictionnaire (ou un ordinateur, ou une base de données...) où les mots (données) seraient listé(e)s dans un ordre quelconque...)) et d'autre part l'intérêt d'optimiser l'algorithme à utiliser en fonction des caractéristiques de l'entrée.

3.2 Applications des tableaux : représentations de polynômes

Vous avez appris différents types de données : entiers (int), réels ($\approx$ float), tableaux, listes... Comment manipuler des polynômes ? Ici, nous illustrons une façon d'utiliser des tableaux pour décrire des polynômes (ce n'est pas la seule, et la manière de représenter des polynômes (tableaux des coefficients, série de Fourier...) dépend de l'utilisation. Ici, nous visons un but pédagogique).

Considérons un polynôme $P[X] = \sum_{i=0}^{n} a_i X^i$ de degré n (i.e., $a_n \neq 0$). Une façon de le représenter (qu'on adopte ici) est par le tableau $[a_0, \cdots, a_n]$ de ses coefficients. Réciproquement, un tableau $[b_0, \cdots, b_n]$ (avec $b_n \neq 0$) représentera le polynôme $\sum_{i=0}^{n} b_i X^i$ de degré n (notons que le tableau correspondant est de longueur $n+1$).

3.2.1 Évaluation de polynôme

Exercice 22 (Évaluation d'un polynôme) *Écrire un algorithme qui prend en entrées un tableau tab de longueur $n+1$ (représentant un polynôme $P[X] = \sum_{i=0}^{n} tab[i]X^i$) et un réel $x \in \mathbb{R}$ et qui retourne $P(x)$.* *// on interdit l'utilisation de ** qui calcule directement la puissance en Python*

Réponse :

Algorithm 25 (Evaluation polynôme)

Entrée un tableau *tab* de $n+1$ réels et un $x \in \mathbb{R}$.

Sortie $\sum_{i=0}^{n} tab[i] * x^i$.

```
def EvalPol(tab,x) :
    res = 0
    for i in range(len(tab)) :
        puiss = 1
        for j in range(i) :
            puiss = puiss * x
        res = res + tab[i] * puiss
    return res
```

Terminaison : 2 boucles *for* imbriquées, le nombre total d'itérations de la seconde boucle est $\sum_{i=0}^{n} i = O(n^2)$ (avec n la taille du tableau).

Terminaison : Il faut prouver 2 invariants de boucle (par récurrence sur i et j) : après la j^{me} itération de la seconde boucle lors de la i^{me} itération de la première, on a $puiss = x^j$ et $res = \sum_{k=0}^{i-1} tab[k] * x^k$; et après la i^{me} itération de la première boucle $res = \sum_{k=0}^{i} tab[k] * x^k$. □

Notons que l'algorithme précédent est volontairement mauvais (on calcule plein de fois les mêmes choses).

Algorithm 26 (Méthode de Horner)

Entrée Un polynôme $P[X] = \sum_{0 \leq i \leq n} a_i X^i$ (tableau $[a_0, \cdots, a_n]$) et un réel $x \in \mathbb{R}$.

Sortie ??

1. Soit $v = a_n$
2. Pour i allant de 1 à n faire
   $v \leftarrow v * x + a_{n-i}$
3. Renvoyer v.

Exercice 23 (*) *Que retourne l'algorithme précédent ? Prouvez le. Comparez l'"efficacité" de cet algorithme avec celui que vous avez proposé à la question précédente.*

Réponse : Terminaison : Une boucle for avec n itérations faisant chacune une addition et une multiplication.

Validité : l'invariant de boucle est : après la i^{me} itération, $v = \sum_{k=0}^{i} a_{n-k} x^{i-k}$. Ainsi, après la n^{me} itération, $v = \sum_{k=0}^{n} a_{n-k} x^{n-k} = \sum_{k=0}^{n} a_k x^k$. Contrairement à l'algorithme précédent qui effectuait $\Theta(n^2)$ opérations, la méthode de Horner n'en exécute que $2n$. □

Remarque : si, à l'exercice 21, vous avez proposé l'algorithme suivant, bravo ! Il est bien meilleur que celui proposé dans la correction de l'exercice 21, mais un chouilla moins bon que l'algorithme de Horner puisque l'algorithme suivant réalise $3n$ opérations (1 addition et 2 multiplications pour chaque itération de la boucle for).

Algorithm 27 (Evaluation polynôme, version 2)

Entrée un tableau tab de $n+1$ réels et un $x \in \mathbb{R}$.

Sortie $\sum_{i=0}^{n} tab[i] * x^i$.

```
def EvalPol2(tab,x) :
    res = 0
    puiss = 1
    for i in range(len(tab)) :
        res = res + tab[i] * puiss
        puiss = puiss * x
    return res
```

3.2.2 Multiplication de polynômes

Cette section est au delà du programme de lycée. Nous en parlons puisqu'elle est au programme d'option info de math. sup. et qu'il est donc sûrement (peut-être) intéressant de la connaître. En particulier, nous donnons un exemple classique de la méthode **diviser pour régner**.

Le problème est le suivant. Étant donnés deux polynômes P et Q (donnés par les tableaux de leurs coefficients), comment calculer le polynôme PQ, i.e., le tableau des coefficients du polynôme produit de P et Q ?

Pour simplifier, on supposera que P et Q sont de même degré n.

Exercice 24 *Soient deux polynômes $P[X] = \sum_{0\leq i\leq n} a_i X^i$ et $Q[X] = \sum_{0\leq i\leq n} b_i X^i$. Remarquez que $PQ[X] = \sum_{i=0}^{2n} (\sum_{k=0}^{i} a_k b_{i-k}) X^i$. En déduire un algorithme qui, étant donnés deux tableaux $[a_i]_{0\leq i\leq n}$ et $[b_i]_{0\leq i\leq n}$ représentants les polynômes P et Q, calcule le tableau représentant le polynôme PQ.*

Remarque/aide : dans l'exercice précédent, on pourra commencer par "compléter" les polynômes P et Q en leur ajoutant des coefficients $a_{n+1}, \cdots, a_{2n}$ (resp., $b_{n+1}, \cdots, b_{2n}$ tous nuls.

En deux mots, déterminer la **complexité temporelle** d'un algorithme consiste à calculer l'ordre de grandeur (asymptotique) du nombre d'opérations élémentaires (ici, on comptera le nombre de multiplications de 2 nombres) en fonction de la "taille" de l'entrée (ici, il s'agit du degré n des polynômes). La complexité de l'algorithme précédent est $O(n^2)$. Le but de ce qui suit est la conception d'un algorithme plus "efficace" (très informellement, "plus rapide"), i.e., avec une meilleure complexité.

Exercice 25 *Soient deux polynômes $P[X] = \sum_{0\leq i\leq n} a_i X^i$ et $Q[X] = \sum_{0\leq i\leq n} b_i X^i$. Écrire un algorithme qui, étant donnés deux tableaux $[a_i]_{0\leq i\leq n}$ et $[b_i]_{0\leq i\leq n}$ représentants les polynômes P et Q, calcule le tableau représentant le polynôme $P+Q$ (resp., $P-Q$).*

Exercice 26 *Soit un polynôme $P[X] = \sum_{0\leq i\leq n} a_i X^i$. Écrire un algorithme qui, étant donnés le tableau $[a_i]_{0\leq i\leq n}$ représentant le polynôme P, et un entier $k \in \mathbb{N}$, calcule le tableau représentant le polynôme $X^k * P[X]$.*

On rappelle (division euclidienne de polynômes) que pour tous polynômes $P[X]$ de degré n et $Q[X]$ de degré $k \leq n$, il existe deux uniques polynômes $A[X]$ et $B[X]$ avec B de degré $< k$ tels que $P[X] = A[X]Q[X] + B[X]$. Pour simplifier, dans la suite, on suppose que le degré de P et Q est une puissance de 2, i.e., $n = 2^p$.

Exercice 27 *Soit un polynôme $P[X] = \sum_{0 \leq i \leq n} a_i X^i$. Écrire un algorithme qui, étant donné le tableau $[a_i]_{0 \leq i \leq n}$ représentant le polynôme P, calcule les tableaux représentant les polynômes A et B tels que $P[X] = X^{n/2}A[X] + B[X]$, avec B de degré $< n/2$.*

Exercice 28 (***) (Méthode de Karatsuba (1962)) *Soient deux polynômes $P(X) = \sum_{0 \leq i \leq n} a_i X^i$ et $Q(X) = \sum_{0 \leq i \leq n} b_i X^i$. Soient A, B, C et D (B et D de degré $< n/2$) les polynômes tels que $P[X] = X^{n/2}A[X] + B[X]$ et $Q[X] = X^{n/2}C[X] + D[X]$. En remarquant (prouvez le) que*

$$PQ[X] = AC[X] * X^n + (AC[X] + BD[X] - (A[X] - B[X])(C[X] - D[X]))X^{n/2} + BD[X],$$

écrire un algorithme qui, étant donnés deux tableaux $[a_i]_{0 \leq i \leq n}$ et $[b_i]_{0 \leq i \leq n}$ représentants les polynômes P et Q, calcule le tableau représentant le polynôme PQ.

Prouver que sa complexité est $O(n^{\log_2 3})$.

Intuitivement, l'algorithme précédent calcule le produit de 2 polynômes de degré n en se ramenant au calcul de 3 produits de polynômes de degré $n/2$.

Notons qu'il existe des algorithmes encore plus efficaces, par exemple en se servant des séries de Fourier, on peut concevoir un algorithme avec une complexité en $O(n \log n)$.

3.3 Applications des tableaux : représentations de matrices

Encore une fois, nous allons au delà du programme de lycée...

Soient $n, m \in \mathbb{N}$. Une matrice $n \times m$ est un *tableau* de $n * m$ éléments (ici des entiers relatifs), indicés par deux indices $1 \leq i \leq n$ et $1 \leq j \leq m$. On note $A = [a_{i,j}]_{i \leq n, j \leq m}$ où $a_{i,j}$ est l'élément de la *ligne* i et de la *colonne* j. **Concrètement, une matrice sera représentée par un tableau de tableaux (typiquement, un tableau de lignes où chaque ligne est elle-même représentée par un tableau).**

Par exemple, la matrice $A = [i + \frac{1}{j^2}]_{i \leq 3, j \leq 4} = \begin{bmatrix} 2 & \frac{5}{4} & \frac{10}{9} & \frac{17}{16} \\ 3 & \frac{9}{4} & \frac{19}{9} & \frac{33}{16} \\ 4 & \frac{13}{4} & \frac{28}{9} & \frac{49}{16} \end{bmatrix}$. Soit $B = \begin{bmatrix} -4 & \frac{1}{4} & 0 & \frac{17}{16} \\ 3 & 2 & \frac{1}{9} & -\frac{15}{16} \\ 6 & \frac{13}{4} & \frac{1}{9} & \frac{49}{16} \end{bmatrix}$.

La somme de deux matrices $A = [a_{i,j}]_{i \leq n, j \leq m}$ et $B = [b_{i,j}]_{i \leq n, j \leq m}$ est la matrice $C = A + B = [a_{i,j} + b_{i,j}]_{i \leq n, j \leq m}$. La différence de deux matrices est définie de façon similaire. Par exemple, $A - B = \begin{bmatrix} 6 & 1 & \frac{10}{9} & 0 \\ 0 & \frac{1}{4} & 2 & 3 \\ -2 & 0 & 3 & 0 \end{bmatrix}$.

Le produit de deux matrices $A = [a_{i,j}]_{i \leq n, j \leq m}$ et $B = [b_{i,j}]_{i \leq m, j \leq p}$ résulte en la matrice $A * B = C = [c_{i,j}]_{i \leq n, j \leq p}$ définie par $c_{i,j} = \sum_{1 \leq k \leq m} a_{i,k} b_{k,j}$. Notons que la matrice $A * B$ n'est définie que si les dimensions sont compatibles, c'est-à-dire que le nombre de colonnes de A doit être égal au nombre de lignes de B [1].

1. Bien que cette définition puisse paraître étrange au premier abord, elle devient naturelle lorsque l'on considère une matrice comme une application dans un espace vectoriel, par exemple une rotation dans le plan, et que la multiplication correspond à la composition de deux applications.

$$\begin{bmatrix} & & a_{1,j} & & \\ & & a_{2,j} & & \\ & & \dots & & \\ & & a_{i,j} & & \\ & & \dots & & \\ & & a_{i,m} & & \end{bmatrix} = A$$

$$B = \begin{bmatrix} & & & & & \\ b_{i,1} & b_{i,2} & \dots & b_{i,j} & \dots & b_{i,m} \\ & & & & & \end{bmatrix} \quad \begin{bmatrix} & & \dots & & \\ & & \dots & & \\ .. & .. & .. & c_{i,j} = \sum_{1 \leq k \leq m} a_{i,k} b_{k,j} & \\ & & & & \end{bmatrix} = B * A$$

Exercice 29 *Soient deux matrices A et B carrées $n \times n$, représentées par des tableaux de tableaux (tableaux de lignes). En utilisant directement la définition du produit de matrices, écrire un algorithme qui calcule le tableau de tableaux représentant la matrice AB.*

A priori, votre algorithme précédent a une complexité en $O(n^3)$ (où l'on mesure le nombre d'opérations arithmétiques). Le but de ce qui suit est de proposé un algorithme plus efficace.

Rappelons qu'il est possible de définir une matrice *par blocs*. Soient quatre matrices $U = [u_{i,j}]_{i \leq \ell, j \leq k}, V = [v_{i,j}]_{i \leq m-\ell, j \leq n-k}, W = [w_{i,j}]_{i \leq \ell, j \leq k}, R = [r_{i,j}]_{i \leq m-\ell, j \leq n-k}$.

La matrice $X = [x_{i,j}]_{i \leq m, j \leq n}$, notée $X = \begin{bmatrix} U & V \\ W & R \end{bmatrix}$, est définie par

— $x_{i,j} = u_{i,j}$ si $i \leq \ell$ et $j \leq k$,
— $x_{i,j} = v_{i,j}$ si $i \leq \ell$ et $k < j \leq n$,
— $x_{i,j} = w_{i,j}$ si $\ell < i \leq m$ et $j \leq k$, et
— $x_{i,j} = r_{i,j}$ si $\ell < i \leq m$ et $k < j \leq n$.

Par exemple, $\begin{bmatrix} A & B \\ A-B & 0 \end{bmatrix} = \begin{bmatrix} 2 & \frac{5}{4} & \frac{10}{9} & \frac{17}{16} & -4 & \frac{1}{4} & 0 & \frac{17}{16} \\ 3 & \frac{9}{4} & \frac{19}{9} & \frac{33}{16} & 3 & 2 & \frac{1}{9} & -\frac{15}{16} \\ 4 & \frac{13}{4} & \frac{28}{9} & \frac{49}{16} & 6 & \frac{13}{4} & \frac{1}{9} & \frac{49}{16} \\ 6 & 1 & \frac{10}{9} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{4} & 2 & 3 & 0 & 0 & 0 & 0 \\ -2 & 0 & 3 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$

Notons que le bloc "0" correspond à la matrice dont tous les éléments sont nuls et dont les dimensions sont implicitement définies par les dimensions de $A - B$ et B.

Exercice 30 *Soient 8 matrices $A = [a_{i,j}]_{i \leq n, j \leq m}$, $B = [b_{i,j}]_{i \leq n, j \leq q}$, $C = [c_{i,j}]_{i \leq \ell, j \leq m}$, $D = [d_{i,j}]_{i \leq \ell, j \leq q}$, $U = [u_{i,j}]_{i \leq m, j \leq p}$, $V = [v_{i,j}]_{i \leq m, j \leq t}$, $W = [w_{i,j}]_{i \leq q, j \leq p}$ et $R = [r_{i,j}]_{i \leq q, j \leq t}$.*

Soit $X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$ et $B = \begin{bmatrix} U & V \\ W & R \end{bmatrix}$. Prouver que $X * Y = \begin{bmatrix} A*U + B*W & A*V + B*R \\ C*U + D*W & C*V + D*R \end{bmatrix}$.

3.3.1 Algorithme de Strassen

Soit $p \in \mathbb{N}^*$. Soient A, B, C, D, E, F, G et H huit matrices $2^{p-1} \times 2^{p-1}$. Soient $M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$ et $N = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$.

Soient $P_1 = A * (F - H)$, $P_2 = (A + B) * H$, $P_3 = (C + D) * E$, $P_4 = D * (G - E)$, $P_5 = (A + D) * (E + H)$, $P_6 = (B - D) * (G + H)$ et $P_7 = (A - C) * (E + F)$.

Exercice 31 *Quelles sont les dimensions de $M, N, M * N$ et P_i, pour tout $i \leq 7$?*

Prouver que $M * N = \begin{bmatrix} -P_2 + P_4 + P_5 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 - P_3 + P_5 - P_7 \end{bmatrix}$

Les matrices P_i, $i \leq 7$, étant données, donner le nombre d'opérations élémentaires (additions et soustractions de nombres) pour calculer $M * N$.

Exercice 32 *Soit $c(p)$ le nombre d'opérations élémentaires (multiplications, additions et soustractions) pour calculer le produit de deux matrices $2^p \times 2^p$. Exprimer $c(p)$ en fonction de $c(p-1)$.*

Exercice 33 *Prouver que $c(p) = O(7^p) = O(n^{\log_2 7})$ et comparer à l'algorithme naïf proposé plus haut (Exercice 28).*

Intuitivement, l'algorithme précédent calcule le produit de 2 matrices de côté n en se ramenant au calcul de 7 produits de matrices de côté $n/2$. Notons que $\log_2 7 > 2.8$.

Comme calculer le produit de 2 matrices de côté n requiert de calculer n^2 coefficients, il n'est pas possible de le faire avec une compexité $o(n^2)$. Le meilleur algorithme (à ma connaissance) pour réaliser un tel produit est celui de François Le Gall (2014), qui a une complexité de $O(n^{2.3728639})$. C'est un problème de recherche ouvert important de savoir si il est possible de faire mieux.

4 Trions un peu

Dans cette section, on se donne un tableau d'entiers et le but est de retourner un tableau avec ces mêmes entiers dans l'ordre, i.e., triés par ordre croissant (ou décroissant, mais ici on considèrera l'ordre croissant). D'un point de vue personnel, enseigner les algorithmes de tri est assez rébarbatif (et l'impression est la même du point de vue élève) car on a l'impression de répéter plusieurs fois la même chose sur un sujet *a priori* peu passionnant. Pourtant, trier des données est une première étape incontournable de la majeure partie des problèmes que l'on rencontre (que ce soit dans un ordinateur ou dans la vie de tous les jours (imaginez un dictionnaire qui ne respecte pas l'ordre alphabétique)). D'autre part, pédagogiquement parlant, les algorithmes de tri permettent d'aborder de nombreuses notions importantes en algorithmique : compléxité temporelle, diviser pour régner (tri fusion/merge sort), structure de données, arbres et diviser pour régner (tri par tas/heap sort), complexité en moyenne (tri rapide/quick sort), complexité en espace (tri sur place ou non)... Bref, c'est chiant mais inévitable/très important/formateur...

Donc, dans ce qui suit, nous allons étudier différents algorithmes qui prennent en entrée un tableau d'entiers et trient ces entiers. Nous imposons la contrainte supplémentaire (sauf pour le tri-fusion) que le tri doit être effectué **sur place**, c'est-à-dire qu'aucun tableau auxiliaire ne doit être créé.

Exercice 34 *Imaginez et codez un algorithme qui prend en entrée un tableau t d'entiers et modifie le tableau de façon à ce qu'à la fin le tableau t contienne les mêmes éléments mais triés dans l'ordre croissant (i.e., $t[0] \leq t[1] \leq \cdots \leq t[n]$).*

Pour vous aider, essayer de formaliser la manière dont vous triez des cartes lorsqu'elles sont dans votre main. Notez aussi que la fonction Echange (Exercice 17) vous sera très utile.

Ce qui suit présente et compare différents algorithmes de tri. **Il est donc important que vous essayiez de répondre à l'exercice précédent, en proposant votre propre solution, avant de lire la suite.**

Comme nous allons le voir, l'idée de présenter divers algorithmes de tri est de les comparer. Pour cela, nous étudions leur complexité temporelle définie par le nombre de comparaisons (d'entiers deux à deux) qu'ils doivent réaliser pour résoudre le problème.

Contrairement aux problèmes précédents (par exemple, chaque algorithme pour calculer le produit de 2 polynômes réalisait le même nombre d'opérations arithmétiques quels que soient les 2 polynômes), le nombre de comparaisons réalisées pour trier un tableau dépend du tableau initial (testez cette assertion avec votre algorithme, en considérant par exemple un tableau initialement déjà trié, ou un tableau où les éléments sont initialement dans l'ordre décroissant). On en vient donc à la notion de **complexité en pire cas**, qui mesure le nombre **maximum** d'opérations élémentaires (en fonction de la taille de l'entrée) réalisées par un algorithme quelle que soit l'instance.

4.1 Tri par Sélection

Le tri (croissant) par sélection d'un tableau t de n éléments consiste à rechercher le plus grand élément de t, le permuter (l'échanger) avec l'élément situé en fin de tableau, et à itérer le traitement avec un élément de moins, jusqu'à ce qu'il n'y ait plus qu'un seul élément. Par exemple, si le tableau initial est :

$t =$	[| 5;	1;	12;	3;	24;	8;	10;	2;	14;	7;	2;	9;	4;	17 |]

Il devient après le premier échange :

$t =$	[| 5;	1;	12;	3;	17;	8;	10;	2;	14;	7;	2;	9;	4;	24 |]

L'algorithme est alors réitéré sur le tableau en omettant son dernier élément, et ainsi de suite.

Exercice 35 *Écrire une fonction $TrouveIndiceMax$ qui prend en entrée un tableau t de n entiers et un entier $0 \leq i < n$ et retourne l'indice d'un élément de valeur maximum parmi les $i+1$ premiers éléments du tableau. Combien de comparaisons réalise cette fonction dans le pire cas ?*

Réponse :

Algorithm 28 (TrouveIndiceMax)

Entrée un tableau *tab* d'entiers et un indice $i \leq len(tab)$.

Sortie indice de l'entier maximum contenu entre les indices 0 et i du tableau

```
def TrouveIndiceMax(tab,i) :
    indice_courant = 0
    max_courant = tab[0]
    for j in range(i + 1) :
        if tab[j] > max_courant :
            indice_courant = j
            max_courant = tab[j]
    return indice_courant
```

Terminaison : $O(i)$ itérations d'une boucle for avec un nombre fini d'opérations (entre autre, une comparaison) à chaque itération.
Validité : Invariant de boucle : Après la j^{me} itération $indice\_courant, max\_courant$ sont respectivement l'indice et la valeur de l'entier maximum se trouvant entre les indices 0 et j dans tab. □

Soit la fonction récursive $TriSelection\_Aux$ définie par :

On supposera que $0 \leq i < n$ *où* n *est la taille de* t.

Algorithm 29 (TriSelection_Aux)

Entrée un tableau tab de n éléments et un entier $0 \leq i < n$.

Sortie ??

1. Si $i = 0$, ne rien faire
2. Sinon
   - Soit $k = TrouveIndiceMax(tab, i)$
   - $Echange(tab, i, k)$
   - TriSelection_Aux$(tab, i-1)$

Exercice 36 *Soit* $t = [|3; 6; 2; 7; 3|]$. *Donner la valeur de* t *après l'exécution de* `TriSelection_Aux(t,2)`. *Expliquer en une phrase ce que fait la fonction* `TriSelection_Aux`.

Réponse : $t = [|2; 3; 6; 7; 3|]$.
l'algorithme trie les $i+1$ premiers éléments du tableau en recherchant tout d'abord le plus grand élément entre les indices 0 et i, en mettant cet élément à la position d'incide i (en échangeant avec l'ancien élément à la position i), et enfin en triant récursivement les i premiers éléments. □

Exercice 37 *Prouver que le nombre de comparaisons réalisées par* `TriSelection_Aux t i` *est égal à* $O(i)$ *plus le nombre de comparaisons de* `TriSelection_Aux t (i-1)`.
En déduire la complexité de `TriSelection_Aux t i` *(en fonction de* i*).*

Réponse : La complexité de $TriSelection\_Aux(t, i)$ est donc $\sum_{k=0}^{i} k = O(i^2)$. □

Exercice 38 *En utilisant la fonction* `TriSelection_Aux`*, écrire une fonction* $TriSelection$ *qui prend un tableau* t *d'entiers en entrée et le trie (ordre croissant).*
Déduire sa complexité de la question précédente.

Réponse :

Algorithm 30 (Tri Selection)

Entrée un tableau *tab* d'entiers.

Sortie trie les éléments du tableau

def TriSelect(tab) :

TriSelection_Aux($tab, len(tab)$)

Complexité en $O(n^2)$ comparaisons d'après la question précédente. □

4.2 Tri par insertion

Le tri (croissant) par insertion d'un tableau t de n éléments consiste à rechercher itérativement la place d'insertion de l'élément d'indice i dans les éléments d'indice 0 à i sachant que les éléments d'indice 0 à $i-1$ sont déjà triés. Une fois cette place déterminée, on procède par échanges successifs pour que les éléments d'indice 0 à i soient triés. **Rappelons qu'on ne veut pas créer de tableaux intermédiaires, mais trier "sur place".**

On commence en considérant que l'élément d'indice 0 est, à lui tout seul, un tableau trié, et on procède ainsi pour insérer les éléments d'indice 1 à $n-1$.

Exercice 39 *Ecrire une fonction posInser qui recherche dans un tableau t trié la place d'insertion d'un nouvel élément de valeur x.*

Réponse :

Algorithm 31 (Position d'insertion)

Entrée un tableau *tab* d'entiers trié dans l'ordre croissant et un entier $x \in \mathbb{N}$.

Sortie l'indice où x doit être inséré.

def posInser(tab,x) :

$compteur = 0$

while $compteur < len(tab)$ and $tab[compteur] \leq x$:

$compteur += 1$

return *compteur*

□

Exercice 40 *Ecrire une fonction decale qui prend un tableau t de taille n et deux indices $i < j < n$ en entrée, et décale d'une position vers la droite (incrémente d'un la position) tout les éléments du tableau en entrée entre les deux indices i et $j-1$ et met l'élément d'indice j en position i.*

On pourra utiliser la fonction `Echange` *de la question* 1

Réponse :

Algorithm 32 (Décale)

Entrée un tableau tab et deux indices $0 \leq i < j < len(tab)$.

Sortie décale (vers la "gauche") les éléments entre les indices i et $j-1$, et met $tab[j]$ à la position i.

def decale(tab,i,j) :

  for k in $range(j-i)$:

    $Echange(tab, j-k, j-k-1)$

□

Exercice 41 *Écrire une fonction triInsertion qui trie par insertion le tableau passé en entrée.*

Réponse :

On a besoin d'une variante légèrement modifiée de $posInser$

Algorithm 33 (Position d'insertion Auxiliaire)

Entrée un entier $i < len(tab)$, un tableau tab d'entiers dont les i premiers éléments sont triés dans l'ordre croissant et un entier $x \in \mathbb{N}$.

Sortie l'indice $\leq i$ où x doit être inséré dans le préfixe déjà trié.

def posInser_aux(tab,x,i) :

  $compteur = 0$

  while $compteur < i$ and $tab[compteur] \leq x$:

    $compteur+ = 1$

  return $compteur$

Algorithm 34 (Tri Insertion)

Entrée un tableau tab.

Sortie trie le tableau, par insertion, dans l'ordre croissant.

def triInser(tab) :

  for i in $range(1, len(tab))$:

    $k = posInser\_aux(tab, tab[i], i)$

    $decale(tab, k, i)$

□

Exercice 42 *Donner (en justifiant) les complexités des fonctions decale et triInsertion.*

Réponse : La fonction $decale(tab, k, i)$ fait $O(i-k)$ échanges, ce qui, dans le pire cas, fait $O(i)$ échange. La fonction $posInser\_aux$ fait dans le pire cas $O(i)$ comparaisons. Le nombre de comparaisons effectuées par $triInser$ est donc $\sum i = 0^n O(i) = O(n^2)$.

Remarquez que $triInser$ fait le plus de comparaisons (i.e., $\Theta(n^2)$) si le tableau initial est déjà trié... □

4.3 Tri à bulles (bubble sort)

Ici, nous vous proposons d'observer le comportement d'un algorithme (itératif, avec une boucle "tant que") afin de deviner sa définition.

L'algorithme à définir prend un tableau d'entiers en entrée et trie (sur place) ses éléments. Nous décrivons l'évolution du tableau $t = [5, 2, 4, 6, 3, 9]$ en entrée.

évolution	t
tableau initial	$[5, 2, 4, 6, 3, 9]$
itération 1	$[2, 5, 4, 6, 3, 9]$
de la boucle	$[2, 4, 5, 6, 3, 9]$
"tant que"	$[2, 4, 5, 3, 6, 9]$
itération 2	$[2, 4, 3, 5, 6, 9]$
itération 3	$[2, 3, 4, 5, 6, 9]$
itération 4	$[2, 3, 4, 5, 6, 9]$

Exercice 43 (*) *Écrire l'algorithme de tri à bulles permettant de réaliser le comportement observé ci-dessus. Quelle est sa complexité ?*

Réponse :

Algorithm 35 (Tri à bulles)

Entrée un tableau *tab*.

Sortie trie le tableau.

```
def bubbleSort(tab) :
    continue = true
    while continue :
        continue = false
        for i in range(1, len(tab)) :
            if tab[i − 1] > tab[i] :
                Echange(tab, i − 1, i)
                continue = true
```

Terminaison : Soit $t_0 = (tab[0], \cdots, tab[n-1])$ (avec n la longueur du tableau) défini par les valeurs initiales du tableau. Soit $t_j = (tab[0], \cdots, tab[n-1])$ avec les valeurs du tableau après l'itération

j de la boucle *while*. On prouve par récurrence sur j que la suite $(t_j)_{j\geq 0}$ décroît strictement pour l'ordre lexicographique (ordre bien fondé) jusqu'à ce que $t_j = t_{j+1}$ auquel cas $continue = false$ et l'algorithme termine.

Validité : si t_j n'est pas trié (après la j^{me} itération), alors $continue = true$ après la $(j+1)^{me}$ itération, et donc l'algorithme ne termine pas tant que le tableau n'est pas trié.

Complexité : $O(n^2)$ comparaisons. Nous pouvons montrer par récurrence qu'après la j^{me} itération (boucle *while*), les j derniers éléments du tableau t_j sont à leur place. (Autrement dit, chacun des j derniers éléments est plus grand que n'importe quel élément parmi les $n-j$ premiers éléments de t_j ET ces j derniers éléments sont dans l'ordre croissant.)

- Pour $j = 1$, il faut montrer qu'après la boucle for, l'élément le plus grand est en dernière position. Nous montrons par récurrence qu'après la i^{me} itération de la boucle for, $tab[i]$ contient le maximum parmi les $i+1$ premiers éléments de t_j (les éléments d'indice entre 0 et i).
  - Clairement vrai pour $i = 1$.
  - Supposons que cela est vrai pour $i < n-1$. Le maximum parmi les $i+2$ premiers éléments est soit le maximum parmi les $i+1$ premiers éléments ($tab[i]$ par hypothèse de récurrence) ou $tab[i+1]$. Dans le premier cas, il faut échanger $tab[i]$ et $tab[i+1]$.
- Supposons qu'après la j^{me} itération (boucle *while*), les j derniers éléments du tableau t_j sont à leur place. Premièrement, la boucle for ne changera pas les j derniers éléments par hypothèse. Montrons alors qu'après la boucle for, le maximum parmi les $n-j$ premiers éléments est dans $tab[n-j-2]$. Même preuve que précédemment avec le tableau constitué des $n-j$ premiers éléments. Donc après la $(j+1)^{me}$ itération (boucle *while*), les $j+1$ derniers éléments du tableau t_{j+1} sont à leur place.

Pour conclure, nous avons prouvé que le nombre d'itérations de la boucle *while* est n dans le pire des cas. Donc la complexité est $O(n^2)$. □

Remarque : bien que l'algorithme de tri à bulles puisse demander $\Theta(n^2)$ comparaisons (par exemple si le tableau initial est trié dans l'ordre décroissant), il reste efficace en pratique si le tableau initial est "presque" trié.

On m'a posé la question de savoir si l'algorithme de tri à bulles pouvait être codé avec une seule boucle. La question sous-jacente était de savoir cet algorithme avait une complexité en $O(n^2)$ parce qu'il était implémenté avec deux boucles imbriquées (une boucle for dans une boucle while dans l'implémentation proposée). L'implémentation ci-dessous répond par la négative à cette question puisqu'elle ne contient qu'une boucle mais que sa complexité reste néanmoins $O(n^2)$.

Algorithm 36 (Tri à bulles 2)

Entrée un tableau *tab*.

Sortie trie le tableau.

```
def bubbleSort2(tab) :
    continue = true
    i = 0
    while continue or i > 0 :
        if i = 0 :
            continue = false
        if i < len(tab) − 1 :
            if tab[i] > tab[i + 1] :
                Echange(tab, i, i + 1)
                continue = true
            i = i + 1
        else
            i = 0
```

4.4 Tri fusion (merge sort)

L'algorithme de tri fusion est l'archétype du principe **diviser pour régner**. Plutôt que de trier directement notre tableau, on le partitionne/divise en 2 sous-tableaux disjoints (la partie droite et la partie gauche), on trie (récursivement) chacun des deux sous-tableaux, puis on fusionne ces deux sous tableaux triés en le tableau initial trié.

Intuitivement, l'algorithme précédent se ramène au tri de 2 tableaux de taille $n/2$ pour trier un tableau de taille n.

Pour simplifier, dans ce qui suit, n est supposé être une puissance de 2 (donc toute division par 2 retournera un entier).

Exercice 44 (**) *Soit un tableau t de n éléments tel que $t[0] \leq t[1] \leq \cdots \leq t[n/2-1]$ et $t[n/2] \leq t[n/2+1] \leq \cdots \leq t[n-1]$ (les moitiés "droite" et "gauche" de t sont déjà triées).*

Écrire un algorithme qui prend t en entrée et le trie avec une complexité $O(n)$ (nombre de comparaisons).

Réponse :

On fait une variante (un peu) plus générale de la fonction demandée pour pouvoir s'en servir plus loin.

Algorithm 37 (Fusion)

Entrée un tableau tab et trois entiers $0 \leq a < b < c < len(tab)$ et on suppose que $tab[a] \leq \cdots tab[b]$ et $tab[b+1] \leq \cdots \leq tab[c]$ (ces deux "parties" sont chacune déjà triées).

Sortie trie les éléments de tab entre ses indices a et c.

```
def fusion(tab, a, b, c)
    pointeur1 = a
    pointeur2 = b + 1
    temp = [0 for i in range(c − a + 1)] :
    for i in range(c − a + 1) :
        if pointeur1 > b :
            temp[i] = tab[pointeur2]
            pointeur2 + = 1
        elif pointeur2 > c :
            temp[i] = tab[pointeur1]
            pointeur1 + = 1
        elif tab[pointeur1] ≤ tab[pointeur2] :
            temp[i] = tab[pointeur1]
            pointeur1 + = 1
        else :
            temp[i] = tab[pointeur2]
            pointeur2 + = 1
    for i in range(c − a + 1) :
        tab[i + a] = temp[i].
```

Terminaison : 2 boucles for consécutives avec $c - a + 1$ itérations chacune et un nombre constant de comparaisons à chaque itération. Donc complexité en $O(c - a)$ comparaisons

Validité : L'invariant de la première boucle est qu'après sa i^{me} itération, les i premières "cases" de temp sont occupées par les i plus petits éléments (de tab entre les indices a et c), et dans l'ordre. □

Remarque : Notons que lorsque l'on manipule des tableaux, la fonction précédente requiert (si on veut la même complexité en temps) la création d'un tableau auxiliaire (et donc la complexité en espace augmente). Avec des listes, on peut implémenter cette fonction sans perte d'espace ni de temps.

Exercice 45 *En utilisant la fonction précédente, écrire un algorithme* **récursif** *qui prend en entrées un tableau t de taille n et 2 entiers $0 \leq i \leq j < n$ et trie (ordre croissant) les éléments entre les indices i et j du tableau t.*

Réponse :

Algorithm 38 (Tri Fusion auxiliaire)

Entrée un tableau tab et deux indices $0 \leq i \leq j < len(tab)$.

Sortie trie récursivement les éléments entre les positions i et j.

```
def triFusion_aux(tab,i,j) :
    if (j − i) ≤ 1 :
        if tab[i] > tab[j] :
            Echange(tab, i, j)
    else :
        triFusion_aux(tab,i,(i+j)//2)
        triFusion_aux(tab,(i+j)//2+1,j)
        fusion(tab, i, (i + j)//2, j)
```

Preuve par récurrence sur $j - i \geq 0$. □

Exercice 46 *En déduire un algorithme qui prend un tableau t en entrée et trie ses éléments par ordre croissant.*

Réponse :

Algorithm 39 (Tri Fusion)

Entrée un tableau tab.

Sortie trie récursivement le tableau.

```
def triFusion(tab) :
    triFusion_aux(tab,0,len(tab)-1)
```

Preuve directe d'après l'exercice précédent. □

Exercice 47 (**) *Prouver que cet algorithme a une complexité (exécute dans le pire cas un nombre de comparaisons) $O(n \log n)$.*

Réponse : Soit $c(n)$ le nombre de comparaisons (en pire cas) réalisées par $triFusion$ pour trier un tableau de n éléments.

Clairement $c(1) = O(1)$ (cas de base dans la fonction $triFusion\_Aux$).

La fonction $triFusion$ s'appelle récursivement sur 2 tableaux de longueurs respectives $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$. Enfin, elle applique la méthode $fusion$ dont on a vu qu'elle fait $O(n)$ comparaison. Donc $c(n) = c(\lfloor n/2 \rfloor) + c(\lceil n/2 \rceil) + O(n)$.

Pour résoudre cette récurrence, on suppose tout d'abord que $n = 2^p$ pour $p \in \mathbb{N}$. Notons $u_p = c(2^p) = c(n)$.

Une simple réécriture de la formule de récurrence donne : $u_0 = O(1)$ et $u_p = 2u_{p-1} + O(2^p)$.

Ici, l'astuce est de poser $v_p = \frac{u_p}{2^p}$ pour tout $p \in \mathbb{N}$. On en déduit (en divisant la formule de récurrence de u_p par 2^p) que $v_p = v_{p-1} + O(1)$. Ainsi, $v_p = O(p)$ (suite arithmétique), et donc $u_p = O(p2^p)$, et donc $c(n) = c(2^p) = u_p = O(n \log n)$ (en effet, notez que $n = 2^p$ et donc $p = \log_2(n)$).

Pour n quelconque, on pose $p \in \mathbb{N}$ tel que $2^p \leq n < 2^{p+1}$. En remarquant que $c(n)$ est croissant en n, on a donc $c(2^p) \leq c(n) \leq c(2^{p+1})$. Comme, d'après le paragraphe précédent, $c(2^p) \sim c(2^{p+1}) \sim O(p2^p) = O(n \log n)$, on en déduit que $c(n) = O(n \log n)$. □

Exercice 48 (***) *Prouver que tout algorithme pour trier un tableau de n entiers requiert $\Omega(n \log n)$ comparaisons.*

Réponse : Considérons un tableau *tab* contenant les entiers de 1 à n. Rappelons qu'il existe $n! \approx n^n$ permutations (bijections) $\sigma : [1, n] \to [1, n]$. Rappelons également que, si $1 \leq i < j \leq n$ sont fixés, le nombre de permutations σ telles que $\sigma(i) < \sigma(j)$ est $n!/2$.

Trier ce tableau revient en fait à déterminer l'unique permutation $\sigma : [1, n] \to [1, n]$, telle que $[\sigma(tab[1]), \sigma(tab[2]), \cdots, \sigma(tab[n])] = [1, \cdots, n]$.

Quel que soit l'algorithme de tri (qui n'utilise que des comparaisons et ne "triche" pas comme l'algorithme ci-dessous), chaque comparaison de deux éléments $tab[i]$ $tab[j]$ pour $0 \leq i < j \leq n$, divise le nombre de permutations candidates par 2. Pour qu'un algorithme soit valide, il ne doit rester qu'une permutation candidate après la dernière (la k^{me}) comparaison. Ainsi, il est nécessaire que $n!/2^k \leq 1$, i.e. (à la louche) que $n^n/2^k \leq 1$, et donc, en passant au log, que $k \geq n \log n$. □

Remarque : Si on connaît la valeur M d'un élément maximum du tableau *tab* à trier, un autre algorithme peut être le suivant : on crée un tableau T de longueur $M + 1$, dont tous les éléments sont initialisés à 0. On parcourt séquentiellement *tab* et, pour toute valeur i trouvée dans le tableau, on incrémente de 1 la valeur de $T[i]$. Finalement, on parcourt séquentiellement T pour récupérer les valeurs dans l'ordre. La complexité est alors $O(\max\{n, M\})$. On peut donc battre la borne inférieure si $M = o(n \log n)$!! C'est cependant, entre autre, au détriment de l'espace mémoire utilisé puisqu'il faut construire un tableau de taille $\Theta(M)$. En général, tout problème demande de faire un compromis entre complexités en temps et en espace.

Ainsi, l'algorithme de tri fusion est théoriquement (dans le pire cas) optimal. En pratique, il existe cependant de meilleurs algorithmes que nous présentons ci-dessous.

4.5 Tri rapide (quick sort)

Le principe du quick sort est de partitionner le tableau à trier (s'il a au moins deux éléments) en trois sous-tableaux, le premier comprenant tous les éléments inférieurs ou égaux à un élément fixé (l'élément pivot), le deuxième ne contenant qu'un seul élément (l'élément pivot) et le troisième contenant tous les éléments supérieurs ou égaux à l'élément pivot. Puis l'algorithme quick sort est appliqué **récursivement** sur les premier et troisième sous-tableau (l'élément pivot, lui, est à sa place définitive). **Ici encore, on triera "sur place" (sans créer d'autres tableaux)**

Soient les fonctions *partition* où `partition(t,i,j)` opère sur le sous-tableau de t des éléments d'indice allant de i à j (inclus), prend `t[i]` comme élément pivot, déplace les éléments supérieurs ou égaux au pivot en fin du sous-tableau (i.e., après le pivot), positionne le pivot à sa place définitive et retourne l'indice de cette place

et *quicksort* qui trie le tableau t entre les indices i et j (inclus).

On considère de plus le tableau t (de 14 éléments) suivant :

$t =$	[7;	1;	12;	3;	24;	8;	10;	6;	14;	7;	2;	9;	4;	17]

Par exemple, l'appel à `partition(t,0,13)` retourne 5, l'indice de la nouvelle place du pivot (le pivot est dans ce cas l'élément $t[0]$ de valeur 7), et après l'appel à `partition(t,0,13)`, une valeur possible du tableau est

$t =$	[1;	3;	6;	2;	4;	7;	12;	24;	8;	10;	14;	7;	9;	17]

Exercice 49 *Avec quels paramètres doit-on appeler* `quicksort` *pour continuer le tri de t juste après cette partition ? Notez qu'il y a 2 appels à réaliser*

Réponse : *quicksort*$(t, 0, 4)$ et *quicksort*$(t, 6, 13)$ □

Exercice 50 *Écrire la fonction* `quicksort`. *(en supposant que la fonction* `partition` *est définie)*

Réponse : On commence par une fonction auxiliaire.

Algorithm 40 (Tri Rapide auxiliaire)

Entrée un tableau *tab* et deux indices i et j.

Sortie trie récursivement les éléments entre les positions i et j.

```
def triRapide_aux(tab,i,j) :
    if (j − i) > 0 :
        k = partition(t, i, j)
        triRapide_aux(tab,i,k-1)
        triRapide_aux(tab,k+1,j)
```

Preuve par récurrence sur $j - i$ (notez que $j - i$ peut être négatif), en vérifiant que les prérequis de la fonction *partition* sont staisfaits, puis en remarquant, qu'après l'exécution de *partition*(t, i, j), $tab[i]$ a été déplacé à la position k, et les éléments $< tab[i]$ sont aux positions entre i et $k - 1$ (sauf si $k - 1 < i$ auquel cas, $tab[i]$ était le plus petit élément entre les indices i et j), et les éléments $\geq tab[i]$ sont aux positions entre $k + 1$ et j (sauf si $k + 1 > j$ auquel cas, $tab[i]$ était le plus grand élément entre les indices i et j).

Algorithm 41 (Tri Rapide)

Entrée un tableau *tab*.

Sortie trie récursivement le tableau.

```
def quicksort(tab) :
    triRapide_aux(tab,0,len(tab)-1)
```

□

Pour la question suivante, on pourra utiliser deux parcours, l'un allant du début du sous-tableau vers la fin et l'autre allant de la fin vers le début. Le parcours *montant* sera suspendu quand un élément sera supérieur au pivot ; le parcours *descendant* sera suspendu quand un élément sera inférieur au pivot. Les éléments seront alors échangés et les parcours reprendront jusqu'à ce qu'ils se rejoignent. La place du pivot sera alors déterminée, le pivot y sera mis, et sa place retournée.

Exercice 51 (***) *Écrire la fonction* `partition`.

Réponse :

Algorithm 42 (Partition avec pivot)

Entrée un tableau tab et deux indices $0 \leq i < j < len(tab)$.

Sortie déplace l'élément $tab[i]$ en une certaine position $i \leq k \leq j$ et déplace tous les autres éléments entre les indices i et j de telle façon que ceux inférieurs (resp., supérieurs) à $tab[i]$ sont placés entre les positions i et $k-1$ (resp., entre $k+1$ et j). Renvoie l'indice k.

```
def partition(tab,i,j) :
    up = i + 1
    down = j
    k = i
    while up ≤ down :
        if tab[k] > tab[up] :
            Echange(tab, k, up)
            k += 1
            up += 1
        else :
            Echange(tab, up, down)
            down -= 1
    return k
```

Terminaison : $down - up$ diminue de 1 à chaque itération de la boucle *while* et celle-ci s'arrête lorsque $down - up < 0$. Donc, $O(j-i)$ itérations.

Validité : Après chaque itération de la boucle *while*, on a $tab[k]$ vaut la valeur initiale de $tab[i]$, $k = up - 1$, pour tout $i \leq j < k$, $tab[j] < tab[k]$, et pour tout $j > down$, $tab[j] \geq tab[k]$. □

Exercice 52 *Donner et expliquer la complexité (en pire cas) de ce tri.*

Réponse : Soit $c(n)$ le nombre de comparaisons (en pire cas) réalisées par *triRapide* pour trier un tableau de n éléments.

Prouvons par récurrence sur n que $c(n) = O(n^2)$.

L'algorithme *triRapide* commence par calculer $k = partition(t, 0, n-1)$ qui effectue $\Theta(n)$ comparaisons (cf. exercice précédent). Puis applique triRapide_aux(tab,i,k-1) et triRapide_aux(tab,k+1,n-1) qui, par récurrence exécutent respectivement $O(k^2)$ et $O((n-k)^2)$ comparaisons. Donc, $c(n) =$

$O(n + k^2 + (n-k)^2)$ comparaisons. En effectuant une simple analyse de fonction (de variable k), $n + k^2 + (n-k)^2$ est maximum pour $k = n/2$ et ce cas donne donc $c(n) = O(n^2)$.

Notez que l'algorithme *triRapide* fait effectivement $\Theta(n^2)$ comparaisons si le tableau est initialement trié. □

Note de "culture générale" : la **complexité en moyenne** de l'algorithme quick sort est $O(n \log n)$. C'est-à-dire que, bien qu'il existe des tableaux pour lesquels l'algorithme de tri rapide requiert $\Theta(n^2)$ comparaisons (donnez un exemple), si l'on fait la moyenne du nombre de comparaisons effectuées par l'algorithme pour "tous les tableaux de longueurs n", nous obtenons un meilleur résultat : $O(n \log n)$. Nous le démontrons ci-dessous. Dans la suite on considère des tableaux qui correspondent aux **permutations** de $[|1, n|]$. C'est-à-dire, un tableau de longueur n contient des éléments, deux-à-deux distincts, dans $[|1, n|]$. Soit σ_n l'ensemble de ces tableaux.

Exercice 53 *Calculer le cardinal de* σ_n.

Soit $T \in \sigma_n$ un tableau. On note $\mathcal{C}(T)$ le nombre de comparaisons d'entiers réalisées par *quickSort*.

Jusqu'à présent, on s'intéressait à la complexité en pire cas, c'est-à-dire au $\max_{T\in\sigma_n} \mathcal{C}(T)$ (le tableau qui nécessite le plus de comparaisons).

Le but de cet exercice est de calculer la complexité en moyenne définie par $C_n = \frac{1}{|\sigma_n|} \sum_{T\in\sigma_n} \mathcal{C}(T)$.

Pour cela, pour tout $1 \leq k \leq n$, soit $\sigma_n^k \subseteq \sigma_n$ l'ensemble des tableaux dont le premier élément est l'entier k.

Exercice 54 *Prouver que* $|\sigma_n^k| = |\sigma_n|/n$.

Finalement, soit $C_n(k) = \frac{1}{|\sigma_n^k|} \sum_{T\in\sigma_n^k} \mathcal{C}(T)$ la complexité en moyenne de *quickSort* dans l'ensemble σ_n^k.

Exercice 55 *Prouver que* $C_n = \frac{1}{n} \sum_{1\leq k\leq n} C_n(k)$.

Exercice 56 *Prouver que* $C_n(k) = n + 1 + C_{k-1} + C_{n-k}$.

Exercice 57 *En déduire que* $C_n = n + 1 + \frac{2}{n} \sum_{1\leq k<n} C_k$

On rappelle que $H_n = \sum_{1\leq k\leq n} \frac{1}{k} \sim \log n$

Exercice 58 *Prouver que* $\frac{C_n}{n+1} = 2H_{n+1} + O(1)$.

indice : calculer d'abord $(n+1)C_{n+1} - nC_n$ *pour prouver que* $\frac{C_{n+1}}{n+2} = \frac{C_n}{n+1} + \frac{4}{n+2} - \frac{2}{n+1}$

Exercice 59 *Conclusion ?*

4.6 Tri par tas (heap sort)

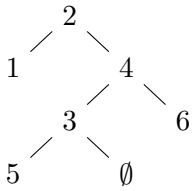
4.6.1 Définitions préliminaires

Définition 1 *Un* arbre binaire *est défini récursivement :*

- $A = \emptyset$ *est l'arbre vide.*
- $A = (u, k, v)$ *est un arbre constitué d'un nœud k (appelé racine de A) de valeur k (dans la suite on confond un nœud et sa valeur), et de deux arbres u et v. L'arbre u est appelé sous-arbre gauche de A, et l'arbre v est appelé sous-arbre droit de A.*

Le nombre de nœuds (non vide) d'un arbre est appelé sa taille.

Par exemple $((\emptyset, 1, \emptyset), 2, (((\emptyset, 5, \emptyset), 3, \emptyset), 4, (\emptyset, 6, \emptyset)))$ est un arbre de taille 6 que l'on pourra représenter ainsi :



Définition 2 *Soit $A = (g, r, d)$ un arbre binaire non vide.*

La racine de g (si $g \neq \emptyset$) et la racine de d (si $d \neq \emptyset$) sont les enfants de la racine r de A. Réciproquement, r est le parent des racines de g et d.

La racine de g (si $g \neq \emptyset$) est l'enfant gauche de r et la racine de d (si $d \neq \emptyset$) l'enfant droit de r. Noter que tout nœud d'un arbre binaire a $0, 1$ ou 2 enfants.

Un nœud sans enfant est appelé une feuille de A.

Dans l'exemple précédent, $(\emptyset, 1, \emptyset)$ est le sous-arbre gauche (réduit au nœud 1), la racine est le nœud 2 et $(((\emptyset, 5, \emptyset), 3, \emptyset), 4, (\emptyset, 6, \emptyset))$ est le sous arbre droit. Le nœud 4 a deux enfants 3 et 6. Le nœud 3 a un unique enfant 5. Les feuilles sont les nœuds 1, 5 et 6.

Définition 3 *Si n est un nœud d'un arbre binaire A, le nombre minimum de nœuds (n exclu) à traverser pour aller de n à la racine de A est appelé la hauteur de n (c'est la distance de n à la racine de A). La racine est le seul nœud de hauteur 0.*

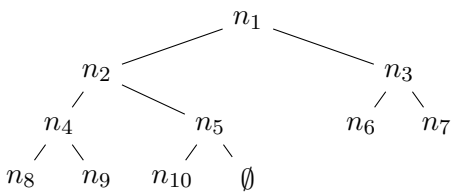
La hauteur maximum d'une feuille de A est la hauteur de A, notée par $h(A)$.

Pour $0 \leq p \leq h(A)$, l'ensemble des nœuds de même hauteur p est appelé le niveau p de A.

Dans l'exemple précédent, la feuille 1 est de hauteur 1. Le nœud 3 est de hauteur 2. La hauteur de l'arbre est 3 (puisque la feuille 5 atteint cette hauteur). Le niveau 1 est constitué des nœuds 1 et 4, le niveau 2 est composé de 3 et 6, etc.

Définition 4 *Un arbre binaire presque complet (ABPC) est un arbre binaire A dans lequel tout nœud de niveau $\leq h(A) - 2$ a exactement deux enfants (en particulier, les feuilles sont de niveau $h(A) - 1$ ou $h(A)$) et où toutes les feuilles du niveau $h(A)$ sont "à gauche" dans la représentation.*

Par exemple l'arbre suivant est un ABPC :



Il dispose de quatre niveaux. Le niveau 0 qui contient n_1. Le niveau 1 qui contient n_2 et n_3. Le niveau 2 qui contient n_4, n_5, n_6 et n_7. Le niveau 4 qui contient n_8, n_9 et n_{10}.

Exercice 60 *Soit A un ABPC de taille n et de hauteur h.*
Pour tout $0 \leq k < h$, déterminer le nombre de nœuds au niveau k.
Donner une borne supérieure (en fonction de n) du nombre de nœuds au niveau h.
En déduire un encadrement de n en fonction de h.
Conclure que $h = \Theta(\log n)$.

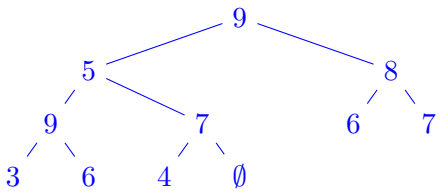
Réponse : Par récurrence sur $k \geq 0$, le nombre de nœuds au niveau k est 2^k.
Pour la même raison, le nombre de nœuds au niveau h est compris entre 1 et 2^h.
On en déduit que $\sum_{i=0}^{h-1} 2^i < n \leq \sum_{i=0}^{h} 2^i$.
Donc $2^h \leq n \leq 2^{h+1} - 1$ et $h = \Theta(\log n)$. □

Il est possible de représenter un ABPC par un tableau en stockant successivement les nœuds du niveau 0, les nœuds du niveau 1, les nœuds du niveau n jusqu'à finir par les feuilles.

Dans le cas de l'ABPC précédent on obtient le tableau $[n1; n2; n3; n4; n5; n6; n7; n8; n9; n10]$

Exercice 61 *Dessiner l'arbre représenté par le tableau suivant : $t = [9; 5; 8; 9; 7; 6; 7; 3; 6; 4]$*

Réponse :



□

Définition 5 *Un nœud (u, k, v) est dominant si k est supérieur ou égal à la valeur de la racine de u et à la valeur de la racine de v (on considère que la racine d'un arbre vide est de valeur $-\infty$).*
Un tas est un ABPC dont tous les nœuds sont dominants.

Exercice 62 *Indiquer les nœuds dominants dans l'arbre obtenu à la question précédente.*

Réponse : Tous les nœuds sauf celui étiqueté par 5. □

4.6.2 Construction d'un tas

Ainsi, la position d'un nœud dans un ABPC représenté par un tableau dépend uniquement de son indice dans le tableau et, possiblement, de la talle du tableau. Les fonctions demandées dans les Questions 22 et 23 sont donc indépendantes des valeurs des nœuds (et donc ne demandent pas de connaître le tableau en entrée).

Exercice 63 *Écrire les fonctions (et donner leur complexité) $indiceEnfantG, indiceEnfantD$ et $indiceParent$ telles que $indiceEnfantG(i)$ (resp. $indiceEnfantD(i)$ prend un indice i, et renvoie l'indice de l'enfant gauche (resp. droit) du nœud d'indice i dans un ABPC représenté par un tableau, et $indiceParent(i)$ renvoie l'indice du parent du nœud d'indice i.*

Notez que l'on suppose ici que les enfants existent.

Réponse : En faisant un dessin, on se convainc rapidement que les enfants de l'élément d'indice i ont pour indices $2i+1$ et $2i+2$ respectivement. Nous le prouvons dans le paragraphe suivant bien que la preuve ne soit pas demandée dans la question.

Soit un indice i et h l'entier tel que $(\sum_{j=0}^{h-1} 2^j) - 1 < i \leq (\sum_{j=0}^{h} 2^j) - 1$. Avec le même raisonnement qu'à la question 59, on en déduit que l'élément d'indice i est au niveau h (le "-1" vient de ce que les indices commencent à 0). Le dernier élément du niveau h est donc celui d'indice $(\sum_{j=0}^{h} 2^j) - 1$. Les éléments du niveau h d'indice inférieur ou égal à i sont donc ceux d'indice $k + (\sum_{j=0}^{h-1} 2^j) - 1$ pour $k = 1, \cdots, i - ((\sum_{j=0}^{h-1} 2^j) - 1)$. Pour tout $1 \leq k \leq i - ((\sum_{j=0}^{h-1} 2^j) - 1)$, l'élément d'indice $(\sum_{j=0}^{h-1} 2^j) + k - 1$ (au niveau h) a deux enfants d'indices $(\sum_{j=0}^{h} 2^j) + 2(k-1)$ et $(\sum_{j=0}^{h} 2^j) + 2(k-1) + 1$ (preuve formelle par récurrence sur k, faîtes un dessin pour vous en convaincre). En particulier, l'élément d'indice i (i.e., pour $k = i - ((\sum_{j=0}^{h-1} 2^j) - 1)$) a donc deux enfants d'indices $(\sum_{j=0}^{h} 2^j) + 2((i - ((\sum_{j=0}^{h-1} 2^j) - 1)) - 1) = 2i + 1$ et $(\sum_{j=0}^{h} 2^j) + 2((i - ((\sum_{j=0}^{h-1} 2^j) - 1)) - 1) + 1 = 2i + 2$.

On déduit de ce qui précède :

Algorithm 43 (Indice Enfant Gauche)

```
def indiceEnfantG(i) :
    return 2 * i + 1
```

Algorithm 44 (Indice Enfant Droite)

```
def indiceEnfantD(i) :
    return 2 * i + 2
```

La fonction suivante est volontairement écrite de façon succincte (et donc incompréhensible au premier abord) pour vous laisser réfléchir un peu...

Algorithm 45 (Parent)

```
def Parent(i) :
    return (i − (1 − (i  mod 2)))//2
```

□

Exercice 64 *Écrire les fonctions (et donner leur complexité)* $estFeuille, estParent1$ *et* $estParent2$ *où* $estFeuille(n,i)$ *renvoie* `vrai` *si et seulement si* i *est l'indice d'une feuille d'un ABPC de taille* n*, et* $estParent1(n,i)$ *(resp.* $estParent2(n,i)$*) renvoie* `vrai` *si et seulement si* i *est l'indice d'un nœud ayant un unique enfant (resp. deux enfants) dans un ABPC de taille* n*.*

Réponse :

L'algorithme suivant suit l'idée qu'un élément est une feuille si et seulement si il n'a pas d'enfants, autrement dit, si le plus petit indice de son enfant potentiel (son enfant gauche) est au delà de l'indice maximum de tous les éléments, i.e, $n-1$ (le "-1" vient encore du fait que les indices commencent à 0).

Algorithm 46 (estFeuille)

```
def estFeuille(n, i) :
    return indiceEnfantG(i) > n − 1
```

Apparté. L'algorithme suivant est également correct (et très (trop) souvent proposé par les élèves).

Algorithm 47 (estFeuille)

```
def estFeuille(n, i) :
    if indiceEnfantG(i) > n − 1 :
        return true
    else
        return false
```

Cet algorithme montre une mécompréhension importante du fait qu'un test (une comparaison) est une opération à part entière (cela ne concerne pas que Python) qui renvoie une valeur booléene qui peut/doit être utilisée. Tel que je le décris, cela paraît anecdotique (je n'ai pas encore trouvé

d'exemple frappant), mais c'est réellement fondamental (d'un point de vue algorithmique) de comprendre la distinction entre les 2 algorithmes précédents et de comprendre que l'algorithme 46 est "le bon".

Algorithm 48 (estParent1)

```
def estParent1(n, i) :
    return indiceEnfantG(i) = n − 1
```

Algorithm 49 (estParent2)

```
def estParent2(n, i) :
    return indiceEnfantG(i) < n − 1
```

□

Exercice 65 *Écrire la fonction estDominant où estDominant*(t, i) *retourne* `vrai` *si et seulement si le nœud d'indice i est dominant dans la représentation t sous forme de tableau d'un ABPC.*

Quelle est l'ordre de grandeur du nombre d'opérations par rapport à la taille de l'ABPC ?

Réponse :

Algorithm 50 (sommet Dominant)

```
def estDominant(t, i) :
    n = len(t)
    if estFeuille(n, i) :
        return true
    elif estParent1(n, i) :
        return t[i] ≥ t[indiceEnfantG(i)]
        else :
            return (t[i] ≥ t[indiceEnfantG(i)]) and (t[i] ≥ t[indiceEnfantD(i)])
```

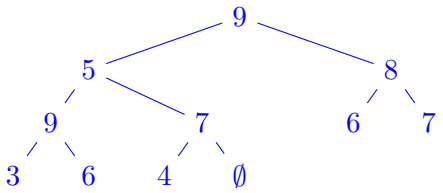
estDominant(t, i) fait un nombre constant de tests et les fonctions auxiliaires appelées par *estDominant*(t, i) (*estFeuille*(n, i), *estParent*1(n, i), *indiceEnfantG*(i), *indiceEnfantD*(i)]) ont une complexité $O(1)$ (voir les exercices précédents), et donc *estDominant* a une complexité temporelle $O(1)$. □

Considérons la fonction $retablirTas(t, i)$ qui opère sur le sous-arbre de t (ABPC sous forme de tableau) à partir de l'indice i pour en faire un tas, **avec comme hypothèse que les enfants de i, s'ils existent, sont déjà des tas**.

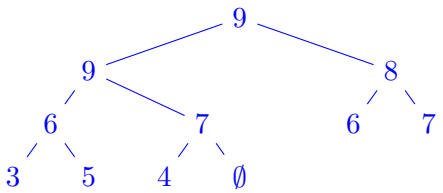
Par exemple, si `t = [| 9; 5; 8; 9; 7; 6; 7; 3; 6; 4 |]` l'appel à $retablirTas(t, 1)$ transformera l'arbre en `[|9; 9; 8; 6; 7; 6; 7; 3; 5; 4|]`.

Exercice 66 *Dessiner l'arbre représenté par le tableau t ci-dessus une fois transformé.*
Décrire les opérations effectuées par la fonction `retablirTas` *et l'écrire.*
Quelle est l'ordre de grandeur du nombre d'opérations par rapport à la taille de l'ABPC ?

Réponse : Voici l'arbre avant changement



Après changement, il devient



La méthode $retablirTas(t, 1)$ considère l'élément d'indice 1 (et ici, de valeur 5). Notons cet élément x. La méthode vérifie d'abord si x est dominant. Si oui, alors le sous arbre enraciné en x est un tas (puisque ses sous-arbres sont des tas). Si non, l'algorithme considère le plus grand enfant de x. Notons y cet élément (ici de valeur 9). Les éléments x et y sont échangés (notons qu'après cet échange, l'élément y est dominant) et la méthode est appelée récursivement sur la nouvelle position de l'élément x. Ici, x n'est toujours pas dominant et est échangé avec l'élémet de valeur 6. Finalement, x est dominant (puisque feuille) et l'algorithme termine.

Algorithm 51 (retablirTas)

def $retablirTas(t, i)$:
  if $not(estDominant(t, i))$
    if $estParent1(len(t), i)$:
      $Echange(t, i, indiceEnfantG(i))$
    elif $t[indiceEnfantG(i)] \geq t[indiceEnfantD(i)]$:
      $Echange(t, i, indiceEnfantG(i))$
      $retablirTas(t, indiceEnfantG(i))$
    else :
      $Echange(t, i, indiceEnfantD(i))$
      $retablirTas(t, indiceEnfantD(i))$

Terminaison : À chaque appel récursif, l'indice de l'élément considéré augmente strictement (tout en restant inférieur à la taille du tableau). L'algorithme termine quand l'élément considéré devient dominant ce qui arrive forcément lorsque l'élément devient une feuille. La profondeur de l'élément considéré augmente à chaque appel, donc l'algorithme termine en temps $O(\log n)$ (majorant de la profondeur de l'arbre d'après une question précédente).

Validité : Par récurrence sur le nombre d'appel récursif, montrez qu'à chaque appel, le seul sommet du sous-arbre initial a ne potentiellement pas être dominant est le sommet considéré à l'appel courant. □

Soit la fonction : *construireTas* qui transforme l'ABPC donné en entrée pour en faire un tas. Par exemple si `t = [| 1; 2; 3; 4; 5; 6; 7; 8; 9; 10 |]` l'appel à `construireTas t` transformera l'arbre en `[| 10; 9; 7; 8; 5; 6; 3; 1; 4; 2 |]`.

Exercice 67 *Ecrire la fonction* `construireTas`.

Quelle est l'ordre de grandeur du nombre d'opérations par rapport à la taille de l'ABPC ?

Réponse :

Les feuilles sont déjà des tas (prouvez que, dans un ABPC de taille n, les feuilles sont les sommets d'indices $n/2+1$ à $n-1$). On ne traite donc que les sommets non feuilles en tranformant successivement les sous-arbres (des indices les plus grands des sommets non feuille vers la racine) en tas.

Algorithm 52 (construire Tas)

```
def construireTas(t) :
    for i in range(len(t)//2 + 1) :
        retablirTas(t, len(t)//2 − i)
```

La complexité est $O(n \log n)$ puisqu'on appelle $O(n)$ fois la fonction rétablir tas. On pourrait en fait le faire en temps $O(n)$. □

4.6.3 Tri par tas

Le tri par tas consiste à interpréter le tableau comme un arbre binaire presque complet, à le transformer en tas, puis itérativement, à permuter la racine de l'arbre avec la dernière feuille, puis, à reconstituer le tas avec un élément de moins et ce jusqu'à ce qu'il n'y ait plus qu'un élément à traiter.

Exercice 68 *écrire la fonction heapsort qui réalise cet algorithme et trie le tableau donné en entrée.*

Quelle est l'ordre de grandeur du nombre d'opérations par rapport à la taille du tableau ?

Réponse : Nous revisitons certaines fonctions précédentes (en ajoutant un paramètre d'entrée), ce qui nous permettra de les utiliser.

La fonction $estDominant2(t, i, n)$ décide si l'indice $i < n$ est dominant dans l'ABPC représenté par les n premiers éléments du tableau t (t pourrait avoir plus de n éléments).

Algorithm 53 (sommet Dominant 2)

```
def estDominant2(t, i, n) :
    if estFeuille(n, i) :
        return true
    elif estParent1(n, i) :
        return t[i] ≥ t[indiceEnfantG(i)]
        else :
            return (t[i] ≥ t[indiceEnfantG(i)]) and (t[i] ≥ t[indiceEnfantD(i)])
```

De même, on adapte la fonction $retablirTas$ pour qu'elle ne travaille que sur les n premiers éléments du tableau (avec possible $n <$ à la taille du tableau).

Algorithm 54 (retablirTas 2)

```
def retablirTas2(t, i, n) :
    if not(estDominant2(t, i, n))
        if estParent1(n, i) :
            Echange(t, i, indiceEnfantG(i))
        elif t[indiceEnfantG(i)] ≥ t[indiceEnfantD(i)] :
            Echange(t, i, indiceEnfantG(i))
            retablirTas2(t, indiceEnfantG(i), n)
        else :
            Echange(t, i, indiceEnfantD(i))
            retablirTas2(t, indiceEnfantD(i), n)
```

Algorithm 55 (Tri par Tas)

```
def triParTas(t) :
    construireTas(t)
    n = len(t)
    for i in range(n − 1) :
        Echange(t, 0, n − i − 1)
        retablirTas2(t, 0, n − i − 2)
```

Terminaison : $triParTas(t)$ fait appel à $construireTas(t)$ (temps $O(n \log n)$) puis fait $O(n)$ appels à $retablirTas2$. Soit une complexité totale $O(n \log n)$. Le tri par tas a donc la même complexité temporelle que le tri fusion, mais fait le tri "sur place". En ce sens, il est préféré au tri fusion.

Validité : Par récurrence sur i, après la i^{me} itération, les i plus grands éléments du tableau sont à leur place, de plus, l'ABPC représenté par les $n-i-1$ premiers éléments du tableau est "presque" un tas : précisément seule la racine n'est pas dominante. □

5 Programmation dynamique

Algorithme qui calcule une solution en combinant les solutions de sous-problèmes. Cela diffère des algorithmes de type diviser pour régner par le fait que les sous-problèmes considérés ne sont pas nécessairement indépendants.

5.1 Problèmes de rendu de monnaie

Notion de problème d'optimisation : On cherche une solution **réalisable** (si elle existe) et qui soit **optimale** pour une certaine fonction objectif.

Dans le **problème de rendu de monnaie**, on veut rendre une certaine somme (x) en utilisant le moins de pièces possible (fonction objectif) parmi un ensemble de "types" de pièces dont on dispose (le système $(c_1, c_2, \cdots, c_n)$).

Par exemple, supposons qu'il faut rendre 78 euros et que l'on dispose de pièces de $1, 2, 5, 10$ et 50 euros (autant de pièces de chaque sorte que l'on veut). Il est possible de rendre 78 pièces de 1 euro, ou 7 pièces de 10 euros, 4 pièces de 2 euros et 4 pièces de 1 euro, ou une pièce de 50, 2 pièces de 10, et un pièce de $5, 2$ et 1 euro. Dans la dernière solution, on ne rend que 6 pièces au total alors que la seconde demande 15 pièces.

Remarque. Dans une solution S (pour rendre x) dont la plus grande pièce a pour valeur y, il y a au moins $\lceil x/y \rceil$ pièces.

Ainsi, dans l'exemple précédent, toute solution qui ne contient pas de pièce de 50 a au moins 8 pièces. Comme on dispose d'une solution avec 6 pièces, on sait qu'il faut au moins rendre une pièce de 50 pour espérer une solution optimale (avec le moins de pièces possible).

Plus formellement, le problème de rendu de monnaie est défini ainsi :

Problème de rendu de monnaie

Entrée : $x \in \mathbb{N}$ et $\mathcal{S} = (c_1, c_2, \cdots, c_n) \in \mathbb{N}^n$

Sortie : $(k_1, \cdots, k_n) \in \mathbb{N}^n$ tels que $\sum_{i \leq n} k_i c_i = x$ et $\sum_{i \leq n} k_i$ est minimum.

Exercice 69 *Donner un exemple d'instance qui n'admet pas de solution réalisable. Qui admet plusieurs solutions optimales.*

Réponse : Exemple d'instance sans solution réalisable : $x = 17$ et $\mathcal{S} = (2, 7)$.
Exemple d'instance avec plusieurs solutions réalisables : $x = 15$ et $\mathcal{S} = (1, 5, 7)$. Les deux solutions optimales (avec trois pièces) sont $(1, 0, 2)$ et $(0, 3, 0)$. Dans la première solution, nous avons une pièce de 1 et deux pièces de 7. Dans la deuxième solution, nous avons trois pièces de 5. Il existe encore d'autres solutions réalisables (par exemple avec 15 pièces de 1) qui sont réalisables (valides) mais pas optimales...) □

On supposera toujours que $c_1 = 1 < c_2 < \cdots < c_n$. On note $OPT(x)$ la valeur $\sum_{i \leq n} k_i$ d'une solution minimum.
Remarque. On suppose que $c_1 = 1$ pour qu'il existe toujours une solution réalisable (prouvez le).

Une famille importante d'algorithmes est celles des **algorithmes gloutons**. Je ne sais pas donner simplement de définition formelle de tels algorithmes. Intuitivement, ces algorithmes suivent le principe de toujours chercher des optima locaux. C'est-à-dire, à chaque étape de l'algorithme, l'action effectuée est une action qui semble la meilleure localement (à cette étape). Ces algorithmes ont l'avantage qu'ils sont généralement très efficaces en temps. Cependant, ils ne sont pas toujours optimaux.

Par exemple, pour se rendre à une destination (disons qu'elle est au nord), un algorithme de routage glouton est de suivre, à chaque carrefour, la route qui semble mener vers la destination (par exemple, on va vers la route qui se rapproche de la direction Nord). Cependant, il peut arriver qu'on finit par arriver devant un mur qui nous force à faire un grand détour, situation que l'on aurait pu éviter si notre algorithme avait eu une vision globale de la carte.

Dans le cas du problème de rendu de monnaie, l'algorithme glouton naturel est de commencer par rendre la plus grande pièce possible, puis la plus grande pièce possible pour rendre ce qu'il reste à rendre, etc.

Exercice 70 *Écrire un algorithme* **glouton** *qui consiste à itérativement (ou récursivement) rendre la plus grande pièce possible.*

Réponse : Les algorithmes proposés prennent en entrées une valeur x (à rendre) et un tableau c (représentant le système de monnaie). Il renvoie un tableau t de longueur n (le nombre de types de pièce possibles) tel que $t[i]$ est le nombre de pièces de type $i-1$ à rendre.

Algorithm 56 (RenduMonnaieIter)

```
def renduMonnaieIter(x, c) :
    n = len(c)
    pieceCourante = n − 1
    res = [0 for i in range(n)]
    resteARendre = x
    while resteARendre > 0
        ℓ = resteARendre//c[pieceCourante]
        res[pieceCourante] = ℓ
        resteARendre = resteARendre − ℓ ∗ c[pieceCourante]
        pieceCourante − = 1
    return res
```

Pour la version récursive, on définit une fonction auxiliaire $RenduMonnaieRecAux$ telle que $renduMonnaieRecAux(x, c, i)$ renvoie une liste $(0 < k_j, \cdots, k_{i-1})$ pour rendre x en utilisant les i plus petites pièces du système c.

Algorithm 57 (RenduMonnaie récursif auxiliaire)

```
def renduMonnaieRecAux(x, c, i) :
    if x == 0 :
        return []
    else :
        ℓ = x//c[i]
        res = renduMonnaieRecAux(x − (ℓ ∗ c[i]), c, i − 1)
        res.append(ℓ)
        return res
```

Preuve par récurrence sur i.

Apparté. L'algorithme suivant est une proposition que j'ai faîte en TD mais ne fonctionne pas car j'ai fait une erreur de débutant (en Python) ! ! !

Algorithm 58 (RenduMonnaie récursif auxiliaire version eronnée)

```
def renduMonnaieRecAuxBad(x, c, i) :
    if x == 0 :
        return []
    else :
        ℓ = x//c[i]
        return renduMonnaieRecAuxBad(x − (ℓ ∗ c[i]), c, i − 1).append(ℓ)
```

En effet, la fonction $x.append(y)$ modifie x mais ne retourne rien ! ! ! (tester et comprendre en quoi diffèrent les algorithmes 55 et 56 et pourquoi l'algorithme 55 est correct et pas celui 56...)

Algorithm 59 (RenduMonnaie récursif)

```
def renduMonnaieRec(x, c) :
    renduMonnaieRecAux(x, c, len(c) − 1)
```

□

Remarque. Soit $(k_1, \cdots, k_n)$ la solution calculée par $Glouton(x)$. $\forall 1 \leq i \leq n$, $k_i = \lfloor \frac{x - \sum_{j=i+1}^{n} k_j c_j}{c_i} \rfloor$.

Exercice 71 *Montrer que l'algorithme glouton n'est pas toujours optimal.*
Montrer que l'algorithme glouton est optimal pour $n \leq 2$.

Réponse : Si $x = 6$ et le système de monnaie est $c = (1, 3, 4)$, l'algorithme glouton renvoie une pièce de 4 et 2 pièces de 1 pour rendre $x = 6$, alors que 2 pièces de 3 sont suffisantes.

Soit $c = (c_1, c_2)$ un système de deux types de pièces ($c_1 < c_2$). Soit une solution réalisable (k_1, k_2) pour rendre une valeur x, i.e., $x = k_1c_1 + k_2c_2$. Donc $x = (k_1 + k_2)c_1 + k_2(c_2 - c_1)$. Comme x, c_1, c_2 sont des paramètres constants, minimiser $k_1 + k_2$ revient donc à maximiser k_2. Donc l'algorithme glouton (dont maximiser k_2 est l'objectif) est optimal. □

Remarque : Un système $(c_1, \cdots, c_n)$ de monnaie est dit *canonique* si l'algorithme glouton renvoie toujours une solution optimale. Aucune caractérisation des systèmes canoniques n'est connue (c'est un problème ouvert). Cependant, contrairement à ce que j'ai dit en TD (désolé), il existe un algorithme "relativement efficace" (du à Kozen et Zaks, en 1994) pour décider si un système est canonique. Si un système $c = (c_1, \cdots, c_n)$ n'est pas canonique, alors il existe (au moins) une valeur x, $c_3 + 1 < x < c_n + c_{n-1}$, telle que l'algorithme glouton "se trompe" (ne renvoie pas une solution optimale) pour x. Il suffit donc de tester l'algorithme glouton et de comparer la solution obtenue avec l'optimale (nous allons maintenant voir comment calculer une solution optimale) pour ce nombre fini de valeurs x.

Rappelons que $OPT(x)$ dénote la valeur d'une solution optimale, i.e., le nombre minimum de pièces (dans le système considéré) à donner pour rendre la valeur x.

Exercice 72 (*) *Soit $x > 0$. Montrer que $OPT(x) = 1 + \min_{1 \leq i \leq n, x - c_i \geq 0} OPT(x - c_i)$*

Réponse : La preuve est par doubles inégalités.

Prouvons d'abord que $OPT(x) \geq 1 + \min_{1 \leq i \leq n, x - c_i \geq 0} OPT(x - c_i)$. Intuitivement, on part d'une solution optimale pour rendre x et on en déduit une solution (pas forcément optimale) pour rendre une valeur $x - c_j$ pour un certain j.

Soit $(k_1, \cdots, k_n)$ une solution optimale pour x, i.e., $\sum_{i=1}^{n} k_i c_i = x$ et $\sum_{i=1}^{n} k_i = OPT(x)$. Comme $x > 0$, il existe $1 \leq j \leq n$ tel que $k_j > 0$. Posons $k' = (k'_1, \cdots, k'_n)$ tel que $k'_i = k_i$ si $i \neq j$ et $k'_j = k_j - 1 \geq 0$. Puisque $\sum_{i=1}^{n} k'_i c_i = x - c_j$, k' est une solution réalisable pour rendre $x - c_j$. Ainsi, $\sum_{i=1}^{n} k'_i \geq OPT(x - c_j)$. On en déduit $OPT(x) = \sum_{i=1}^{n} k_i = 1 + \sum_{i=1}^{n} k'_i \geq 1 + OPT(x - c_j) \geq 1 + \min_{1 \leq i \leq n, x - c_i \geq 0} OPT(x - c_i)$.

Prouvons maintenant que $OPT(x) \leq 1 + \min_{1 \leq i \leq n, x - c_i \geq 0} OPT(x - c_i)$. Soit $1 \leq j \leq n$ tel que $OPT(x - c_j) = \min_{1 \leq i \leq n, x - c_i \geq 0} OPT(x - c_i)$. Intuitivement, on part d'une solution optimale pour rendre $x - c_j$ et on en déduit une solution (pas forcément optimale) pour rendre la valeur x.

Soit $(k'_1, \cdots, k'_n)$ une solution optimale pour $x - c_j$, i.e., $\sum_{i=1}^{n} k'_i c_i = x - c_j$ et $\sum_{i=1}^{n} k'_i = OPT(x - c_j)$. Posons $k = (k_1, \cdots, k_n)$ tel que $k_i = k'_i$ si $i \neq j$ et $k_j = k'_j + 1 \geq 0$. Puisque $\sum_{i=1}^{n} k_i c_i = x$, k est une solution réalisable pour rendre x. Ainsi, $\sum_{i=1}^{n} k_i \geq OPT(x)$. On en déduit $OPT(x) \leq \sum_{i=1}^{n} k_i = 1 + \sum_{i=1}^{n} k'_i = 1 + OPT(x - c_i) = 1 + \min_{1 \leq i \leq n, x - c_i \geq 0} OPT(x - c_i)$. □

Rappelons que le paradigme "diviser pour régner" consiste à, pour résoudre un problème, le "diviser" en des problèmes "plus petits" que l'on peut résoudre indépendamment, puis combiner leurs solutions pour obtenir une solution du problème initial. Le paradigme de **programmation dynamique** généralise "diviser pour régner" en ce sens qu'il consiste à, pour résoudre un problème, le

"diviser" en des problèmes "plus petits", pas nécessairement indépendamment, puis combiner leurs solutions pour obtenir une solution du problème initial. La notion de programmation dynamique a été définie par Bellman (1950). Le terme "programmation" n'est pas à prendre dans son sens usuel mais doit être compris comme "planification". On "planifie" le calcul de notre "gros problème" en préparant d'abord les calculs des sous-problèmes.

Exercice 73 (*) *Déduire de la question précédente un algorithme qui prend en entrées un tableau* $[c_1, c_2, \cdots, c_n]$ *(tel que* $c_1 = 1 < c_2 < \cdots < c_n$*) et un entier* $x \in \mathbb{N}$ *et calcule* $OPT(x)$ *ainsi qu'une solution optimale.*

Réponse : Commençons par un algorithme qui ne renvoie que $OPT(x)$ (mais pas une solution optimale). En effet, cet algorithme "contient" toute la difficulté algorithmique du problème. L'algorithme 62 (qui répond à la question en renvoyant également une solution optimale) n'est pas plus difficile algorithmiquement, mais est plus difficile à lire puisqu'il manipule plus de variables.

Algorithm 60 (Rendu de Monnaie par programmation dynamique)

Entrée un tableau c d'entiers tel que $c[0] = 1 < c[2] < \cdots < c[len(c) - 1]$ et un entier $x \in \mathbb{N}$

Sortie $OPT(x)$.

```
def renduMonnaieProgDyn(x, c) :
    n = len(c)
    res = [x for i in range(x + 1)]
    res[0] = 0
    for i in range(1, x + 1) :
        for j in range(n) :
            if i − c[j] ≥ 0 :
                res[i] = min(res[i − c[j]] + 1, res[i])
    return res[x]
```

Terminaison : deux boucles *for* imbriquées, donc termine avec une complexité en $O(xn)$.

Validité : Après l'itération j (de la boucle interne) de l'itération i (de la boucle principale), on prouve par récurrence les propriétés suivantes. Pour tout $0 \leq k < i$, $res[k] = OPT(k)$. Pour tout $i < k \leq x$, $res[k] \leq OPT(k)$. Finalement (par récurrence sur j), $res[i] = 1 + \min_{1 \leq q \leq j, x - c_q \geq 0} OPT(x - c_q)$. On en déduit, qu'après l'itération n (de la boucle interne) de l'itération i (de la boucle principale), $res[i] = 1 + \min_{1 \leq q \leq n, x - c_q \geq 0} OPT(x - c_q)$. D'après l'exercice précédent, $res[i] = OPT(i)$.

Continuons en améliorant d'abord un peu la complexité de l'algorithme précédent en "tronquant" la boucle *for* interne, en la remplaçant par une boucle *while*.

Algorithm 61 (Rendu de Monnaie par programmation dynamique)

Entrée un tableau c d'entiers tel que $c[0] = 1 < c[2] < \cdots < c[len(c) - 1]$ et un entier $x \in \mathbb{N}$

Sortie $OPT(x)$.

```
def renduMonnaieProgDyn(x, c) :
    n = len(c)
    res = [x for i in range(x + 1)]
    res[0] = 0
    for i in range(1, x + 1) :
        j = 0
        while (i − c[j] ≥ 0) and (j < n) :
            res[i] = min(res[i − c[j]] + 1, res[i])
            j += 1
    return res[x]
```

Terminaison : une boucle *for* contenant une boucle *while*. La boucle *while* termine puisque j est strictement croissant et la boucle while termine quand $j \geq n$.

Validité : Montrez que le résultat produit est le même que celui de l'algorithme précédent, et donc, par la preuve de validité précédente, le résultat est correct.

Algorithm 62 (Rendu de Monnaie par programmation dynamique)

Entrée un tableau c d'entiers tel que $c[0] = 1 < c[2] < \cdots < c[len(c) - 1]$ et un entier $x \in \mathbb{N}$

Sortie $OPT(x)$ et $[k_0, \cdots, k_{len(c)-1}]$ tels que $\sum_{i=0}^{len(c)-1} k[i]c[i] = x$ et $\sum_{i=0}^{len(c)-1} k[i] = OPT(x)$.

```
def renduMonnaieProgDyn(x, c) :
    n = len(c)
    res = [[x, [0 for j in range(n)]] for i in range(x + 1)]
    res[0][0] = 0
    for i in range(1, x + 1) :
        j = 0
        while (i − c[j] ≥ 0) and (j < n) :
            if res[i − c[j]][0] + 1 < res[i][0] :
                res[i][0] = res[i − c[j]][0] + 1
                res[i][1] = copy(res[i − c[j]][1]
                res[i][1][j] += 1
            j += 1
    return res[x]
```

□

Remarque générale sur la complexité temporelle : Cette remarque est loin du programme mais intéressante à connaître. Nous avons abordé informellement la notion de complexité temporelle d'un algorithme. Il s'agit du nombre d'**opérations élémentaires** [2] effectuées par l'algorithme dans un **pire cas** (une instance en entrée qui demande le plus de calculs). Cette complexité est exprimée en fonction de la **"taille de l'entrée"**. Par exemple, nous avons vu des algorithmes de tri avec des complexité en $O(n^2)$ ou en $O(n \log n)$ avec n la longueur du tableau en entrée.

Un problème est dit **polynomial** si il existe un algorithme pour le résoudre dont la complexité peut s'exprimer comme un polynôme de la taille de l'entrée (comme c'est le cas pour des problèmes de tri). L'ensemble des problèmes polynomiaux est noté P et est considéré comme un ensemble de problèmes "faciles" (ils peuvent être résolus relativement efficacement).

Une autre classe de problèmes importante est celle des problèmes dont on peut tester en temps polynomial si une solution est valide (par exemple, je vous donne un tableau et vous devez non pas le trier mais me dire si il est déjà trié ou non). Cette classe s'appelle NP pour Non déterministe Polynomial. Le problème de savoir si $P = NP$ est un des problèmes du millénaire de l'institut de mathématiques Clay. Informellement, il s'agit de savoir si vérifier efficacement qu'une solution à un problème est bien une solution valide revient à pouvoir trouver efficacement une solution valide du problème.

Revenons maintenant au problème de rendu de monnaie. Sa complexité est $O(xn)$. Est-il polynomial ? En d'autres termes, quelle est la taille de l'entrée ? L'entrée consiste en un entier x et un tableau de n entiers $\leq c_n$. Ici, ça devient subtil : si on dit que la taille de l'entrée est l'espace mémoire utilisé pour la stoquer, alors il faut remarquer qu'un entier q est stocké en binaire et donc occupe une place de $O(\log q)$ bits. La taille de l'entrée de notre problème est alors $\log x + n \log c_n$. Mais donc xn est exponentiel en la taille de l'entrée ! !

Le problème de rendu de monnaie (et ceux de sac à dos ci dessous) sont en fait ce qu'on appelle des problèmes **pseudo-polynomiaux** : très grossièrement (voire carrrément faux, mais c'est pour avoir une intuition), ils admettent un algorithme dont la complexité est polynomial en le nombre d'entiers, mais exponentielle si on considère le codage de ces entiers.

5.2 Problèmes du sac à dos

Un problème très similaire est celui du **sac-à-dos avec répétitions**. Supposons que vous braquiez une banque, il y a une infinité de billets de valeur $v_1, v_2, \cdots, v_n$ mais chaque billet de valeur v_i a un poids de p_i. Comment remplir votre sac-à-dos avec le plus d'argent sachant que votre sac ne supporte qu'un poids W.

Problème de sac-à-dos avec répétitions

Entrée : $W \in \mathbb{N}$ et $\mathcal{S} = ((v_1, p_1), (v_2, p_2), \cdots, (v_n, p_n)) \in \mathbb{N}^n$

Sortie : $(k_1, \cdots, k_n) \in \mathbb{N}^n$ tels que $\sum_{i \leq n} k_i p_i \leq W$ et $\sum_{i \leq n} k_i v_i$ est maximum.

Ce problème se résout comme celui de rendu de monnaie en remplaçant la minimisation par de la maximisation.

2. Les opérations élémentaires sont typiquement les opérations arithmétiques, les comparaisons, les affectations... Celles que l'on compte doivent être définies selon le contexte. Par exemple, pour un algorithme de trie, compter les comparaisons semble naturel.

Algorithm 63 (Sac à dos avec remise)

Entrée un tableau s de couples d'entiers $(s[i][0], s[i][1]) = (v_i, p_i)$ tel que $s[0][1] \leq s[2][1] \leq \cdots \leq s[len(s)-1][1]$et un entier $W \in \mathbb{N}$

Sortie $OPT(W)$ et $k = [k_0, \cdots, k_{len(s)-1}]$ tels que $\sum_{i=0}^{len(s)-1} k[i]s[i][1] \leq W$ et $\sum_{i=0}^{len(s)-1} k[i]s[i][0] = OPT(W)$.

```
def sacADosRemise(W, s) :
    n = len(s)
    res = [[0, [0 for j in range(n)]] for i in range(W + 1)]
    for i in range(1, W + 1) :
        j = 0
        while (i − s[j][1] ≥ 0) and (j < n) :
            if res[i − s[j][1]][0] + s[j]][0] > res[i][0] :
                res[i][0] = res[i − s[j][1]][0] + s[j]][0]
                res[i][1] = copy(res[i − s[j][1]][1]
                res[i][1][j] + = 1
            j + = 1
    return res[W]
```

Nous finissons cette section par un problème qui ressemble aux précédent mais qui demande un peu plus de travail.

Dans le problème du **sac-à-dos SANS répétitions**, vous voulez encore une fois remplir votre sac avec des trésors, mais chaque objet n'apparaît qu'une fois.

Problème de sac-à-dos SANS répétitions

Entrée : $W \in \mathbb{N}$ et $\mathcal{S} = (x_1, \cdots, x_n) = ((v_1, p_1), (v_2, p_2), \cdots, (v_n, p_n)) \in \mathbb{N}^n$

Sortie : $(k_1, \cdots, k_n) \in \{\mathbf{0}, \mathbf{1}\}^n$ tels que $\sum_{i \leq n} k_i p_i \leq W$ et $\sum_{i \leq n} v_i$ est maximum.

Comme précédement, on a $OPT(W, x_1, \cdots, x_n) = \max_{i \leq n} v_i + OPT(W - p_i, x_1, \cdots, x_{i-1}, x_{i+1}, \cdots, x_n)$. Cependant, si l'on se contente de cette formule, il y a un nombre exponentiel de sous-problème à traiter, ce qui donne un algorithme en $O(W2^n)$.

Pour pallier ce problème, on observe que l'on peut en fait se limiter à un nombre plus restreint de sous-problèmes. On note $OPT(W, j) = OPT(W, x_1, \cdots, x_j)$.

Exercice 74 (*) *Soit* $W > 0$. *Montrer que* $OPT(W, j) = \max\{OPT(W - p_j, j-1) + v_j, OPT(W, j-1)\}$.

Réponse : Preuve similaire à celle de l'exercice 72. □

Exercice 75 (*) *Déduire de la question précédente un algorithme qui prend calcule* $OPT(W, n)$ *ainsi qu'une solution optimale.*

Réponse :

Algorithm 64 (Sac à dos (sans remise)) **Entrée** un tableau s de couples d'entiers $(s[i][0], s[i][1]) = (v_i, p_i)$ tel que $s[0][1] \leq s[2][1] \leq \cdots \leq s[len(s)-1][1]$et un entier $W \in \mathbb{N}$

Sortie $OPT(W)$ et $k = [k_0, \cdots, k_{len(s)-1}]$ (tableau de booléens) tels que $\sum_{i=0}^{len(s)-1} k[i]s[i][1] \leq W$ et $\sum_{i=0}^{len(s)-1} k[i]s[i][0] = OPT(W)$.

```
def sacADos(W, s) :
    n = len(c)
    res = [[[x, [0 for k in range(n)]] for j in range(n+1)] for i in range(W+1)]
    for j in range(1, n+1) :
        for i in range(1, W+1) :
            k = 0
            while (i - s[k][1] ≥ 0) and (k < n) :
                if res[i - s[k][1]][j-1][0] + s[k][0] > res[i][j][0] :
                .   res[i][j][0] = res[i - s[k][1]][j-1][0] + s[k][0]
                .   res[i][j][1] = copy(res[i - s[k][1]][j-1][1]
                .   res[i][j][1][k] += 1
                k += 1
    return res[W][n]
```

Preuve par récurrence sur i et j que $res[i][j][0] = OPT(i, j)$ et $res[i][j][1]$ contient une solution optimale pour un sac à dos de taille i lorsque sont disponibles les objets de type 1 à j. Référez vous à la preuve de l'algorithme 60. □

5.3 Plus grande sous-séquence d'un tableau

Le but de ce problème est d'étudier différents algorithmes pour calculer le maximum des sommes d'éléments consécutifs d'un tableau donné d'entiers relatifs.

Plus précisément, soit T un tableau de n éléments à valeur dans $\mathbb{Z}$.

Pour tout $0 \leq a \leq b < n$, on note $t(a, b) = \sum_{k=a}^{b} T.(k)$. On cherche la plus grande somme de ce type. C'est-à-dire, on cherche à calculer $S(T) = \max_{0 \leq a \leq b < n} t(a, b)$.

Par exemple, pour $T = [-3, 5, -4, 8, -2]$, $S(T) = t(1, 3) = 9$. Pour $T = [-5, -7, -1, -4]$, $S(T) = t(2, 2) = -1$.

Les 5 sous-parties peuvent être traitées indépendamment.

5.3.1 Algorithme Naïf.

On admet que l'algorithme suivant calcule effectivement $S(T)$.

Algorithm 65

```
Def SomMax_1(T) :
    n = len(T)
    sMax = T[0]
    for a in range(n) :
        for b in range(a, n) :
            m = T[a]
            for k in range(a + 1, b + 1) :
                m = m + T[k]
            sMax = max(m,sMax)
    Return sMax
```

Question 1 *Calculer le nombre d'itérations de l'opération "$m = m + T[k]$". En déduire l'ordre de grandeur de la complexité de l'algorithme $SomMax_1$ en fonction de la longueur n de son entrée.*

5.3.2 Première amélioration.

On se propose de supprimer l'une des boucles "for" imbriquées de la fonction $SomMax_1$. Pour cela, on peut mettre la variable $sMax$ à jour au fur et à mesure du calcul des $t(a,b) = \sum_{k=a}^{b} T[k]$ pour a fixé et b variant de a à $n-1$.

Question 2 *En suivant la suggestion ci-dessus, écrire en Python la fonction $SomMax_2$ qui prend en entrée un tableau T et renvoie $S(T)$. Prouver que le nombre d'additions réalisées par $SomMax_2$ est $\Theta(n^2)$.*

5.3.3 Version Récursive.

Soit T un tableau de longueur n et soit $k \leq n$. On note $S(T,k) = \max_{0 \leq a \leq b < k} t(a,b)$.
Remarquez que $S(T,n) = S(T)$.

Question 3 *Exprimer $S(T,n)$ en fonction de $S(T,n-1)$ et de $t(a,n-1)$, pour $0 \leq a < n$.*

Question 4 *Écrire en Python une fonction récursive $SomMax_3$ qui prend en entrée un tableau T et renvoie $S(T)$. Calculer l'ordre de grandeur du nombre d'additions réalisées par $SomMax_3$.*

5.3.4 Diviser pour régner.

Soit T un tableau de longueur n et soit $0 \leq a \leq b < n$.
Posons $t_{aux}(a,b) = \max_{a \leq x \leq y \leq b} t(x,y)$. C'est-à-dire, $t_{aux}(a,b)$ est le maximum des sommes d'éléments consécutifs compris entre les indices a et b de T.

Soit $a \leq m \leq b$. On pose finalement $t_{join}(a, m, b) = \max_{a \leq x \leq m \leq y \leq b} t(x, y)$. C'est-à-dire, $t_{aux}(a, b)$ est le maximum des sommes d'éléments consécutifs compris entre les indices a et b de T et contenant l'élément d'indice m.

Question 5 *Soit $a \leq m < b$. Exprimer $t_{aux}(a, b)$ en fonction de $t_{aux}(a, m)$, $t_{aux}(m + 1, b)$ et $t_{join}(a, m, b)$.*

Question 6 *Écrire en Python une fonction $Junction(T, a, m, b)$ qui prend en entrée un tableau T de longueur n et trois entiers $0 \leq a \leq m \leq b < n$ et calcule $t_{join}(a, m, b)$ avec un nombre d'additions de l'ordre de $b - a$.*

Dans la suite on pourra admettre le résultat de la question précédente.

Question 7 *Écrire en Python une fonction récursive $SumAux(T, a, b)$ qui prend en entrée un tableau T de longueur n et deux entiers $0 \leq a \leq b < n$ et calcule $t_{aux}(a, b)$.*

Question 8 *En déduire en Python une fonction récursive $SomMax_4$ qui prend en entrée un tableau T et calcule $S(T)$.*

Question 9 *Soit u_n le nombre d'additions réalisées par la fonction $SomMax_4$ appliquée à un tableau de longueur n. Exprimer la relation de récurrence satisfaite par $(u_n)_{n \geq 0}$. Conclusion ?*

5.3.5 Programmation dynamique.

Dans cette section, nous étudions un algorithme encore plus efficace.
Soit T un tableau de longueur n et soit $k \leq n$. Rappelons que $S(T, k) = \max_{0 \leq a \leq b < k} t(a, b)$.
De plus, posons $R_k = \max_{0 \leq a < k} t(a, k - 1)$.

Question 10 *Pour $1 < k \leq n$, exprimer R_k en fonction de R_{k-1}, $S(T, k - 1)$ et $T[k - 1]$. En déduire la valeur de $S(T, k)$.*

Question 11 *En déduire en Python une fonction $SomMax_5$ qui prend en entrée un tableau T et calcule $S(T)$ en temps linéaire en la taille de T.*

6 Algorithmes de graphes

Le premier exercice ne traite pas de graphes mais pourrait être adapté en un exercice de graphe. Il permet, je crois, d'avoir une meilleur intuition de ce qu'est la complexité temporelle (en pire cas) d'un algorithme.

Exercice 76 (Cow path) *Vous partez de l'origine dans un repère orthonormé. Un trésor est caché sur l'axe des abscisses (dans un premier temps, on pourra considérer qu'on connaît une borne supérieure n sur la distance entre l'origine et la position du trésor). Donner un algorithme pour trouver "rapidement" le trésor. Borner la longueur du trajet effectivement réalisé en suivant votre algorithme en fonction de la distance (inconnue) d entre l'origine et le trésor.*

Réponse : Un premier algorithme consiste à partir d'un côté, disons vers la droite, et on marche tant qu'on n'a pas trouvé le trésor et, de plus, si on arrive à l'extrémité droite du chemin (ou si on a parcouru plus que la borne n), on fait demi-tour et on marche vers la gauche jusqu'à découvrir le trésor. Cet algorithme a l'inconvénient que si le chemin est infini et qu'on ne connait pas de borne n, vous risquez de partir du mauvais côté et de marcher indéfiniment sans trouver le trésor. Si n est connu, dans le pire cas (si vous partez du mauvais côté au départ), vous aurez marché une distance $d + 2n$.

Un meilleur algorithme est de partir vers la droite, de marcher une distance $2^0 = 1$, si vous n'avez pas trouvé le trésor, de revenir à votre point de départ et marcher une distance $2^2 = 2$ vers la gauche, revenir au départ, marcher une distance $2^2 = 4$ vers la droite... À l'itération $0 \leq i$, si vous n'avez pas déjà trouvé le trésor, vous marchez (vers la droite si i est pair, vers la gauche sinon) jusqu'à une distance 2^i et, si vous n'avez pas trouvé le trésor, revenez au centre.

Clairement, cet algorithme finira par vous mener au trésor. En effet, si on note d_i^g et d_i^d les distances des points extrémaux respectivement à gauche et à droite que vous avez déjà atteints à la fin de l'itération i, les suites (d_i^g) et (d_i^d) sont telles que $d_{i+2}^g > d_i^g$ et $d_{i+2}^d > d_i^d$ pour tout i.

Soit k tel que $2^{k-1} < d \leq 2^k$. Lors de la k^{me} itération vous atteignez un point à distance 2^k du centre. Dans le pire cas, le trésor est de l'autre côté et vous allez donc l'atteindre lors de la prochaine itération $k+1$. Au total, vous aurez parcouru une distance $d+2*\sum_{i=0}^{k} 2^i = d+2(2^{k+1}-1) < d + 2(4d - 1) \leq 9d$. □

Les deux exercices suivants visent à montrer l'intérêt (et la difficulté) de modéliser un problème à l'aide de graphes.

Exercice 77 (Le loup, la chèvre et le chou) *Vous devez faire traverser une rivière à un loup, une chèvre et un chou. Pour cela, vous ne pouvez embarquer que l'un d'entre eux à chaque traverée. Pour corser l'affaire, vous ne pouvez jamais laisser seuls le loup et la chèvre, ni la chèvre et le chou. Donner une stratégie qui utilise le moins de traversées.*

Réponse : Dans cet exercice, la modélisation sous forme de graphe n'est pas nécessaire, mais peut être formatrice. Nous vous laissons construire le graphe équivalent au problème par vous même (si vous ne voyez pas comment faire, inspirez vous de l'exercice, plus difficile, suivant).

Pour résoudre l'exercice, il suffit de voir qu'à chaque instant, un seul mouvement est possible (c'est une forme d'algorithme glouton). D'abord on fait traverser la chèvre (seule possibilité). Ensuite, on revient de l'autre côté et faisons traverser le chou (si on commence par le loup, cela donne le même nombre de traversées). On revient à nouveau de l'autre côté, mais cette fois ci en ramenant la chèvre. On fait alors traverser le loup. Finalement, on refait traverser la chèvre. Cette stratégie résout le problème en 3 aller-retour plus une traversée. Elle est optimale puisque implicitement nous avons envisagé toutes les possibilités. Si vous n'êtes pas convaincus (ce que j'espère), utilisez la méthode de l'exercice suivant. □

Exercice 78 (Les randonneurs) *Quatre personnes (et une lampe de poche) arrivent (de nuit) devant un pont. Le pont ne supporte que le poids de deux personnes à la fois. Pour traverser le pont, il faut être muni de la lampe de poche. Finalement, les randonneurs (appelés poétiquement A, B, C et D) peuvent traverser le pont à des vitesses de $1, 2, 5$ et 10 minutes respectivement. Lorsque 2 personnes traversent, leur vitesse est celle de la plus lente d'entre elles. Donner une stratégie qui minimise le temps total de traversée.*

Réponse : Nous modélisons le problème sous forme d'un graphe de la manière suivante. Une *configuration* est une "situation" à un moment donné, i.e., une description de qui est du "mauvais" côté du pont (personnes qui n'ont pas encore traversées), qui est du "bon" côté (ceux qui ne sont pas du mauvais côté), et de quel côté se trouve la lampe de poche. Nous adoptons la convention suivante pour décrire les configurations : l'ensemble $S \subseteq \{A, B, C, D\}$ représente la configuration où les personnes dans l'ensemble S sont du mauvais côté du pont et n'ont pas la lampe de poche (et donc, les personnes dans $\{A, B, C, D\} \setminus S$ ont déjà traversé et ont la lampe de poche). L'ensemble $S^* \subseteq \{A, B, C, D\}$ représente la configuration où les personnes dans l'ensemble S^* sont du mauvais côté du pont et ont la lampe de poche (et donc, les personnes dans $\{A, B, C, D\} \setminus S^*$ ont déjà traversé et n'ont pas la lampe de poche).

Nous modélisons l'ensemble des configurations et des transitions entre elles par un graphe. Les sommets de ce graphe sont les configurations $S, S^* \subseteq \{A, B, C, D\}$. Il y a une arête de poids p entre deux configurations x et y, si on peut passer par UNE traversée de pont en p minutes de la configuration x à la configuration y (notons que c'est symmétrique). Le problème est donc équivalent à trouver un plus court chemin (dans le graphe pondéré) de la configuration $ABCD^*$ (tous les randonneurs et la lampe de poche sont du mauvais côté du pont) à la configuration $\emptyset$ (tout le monde a traversé).

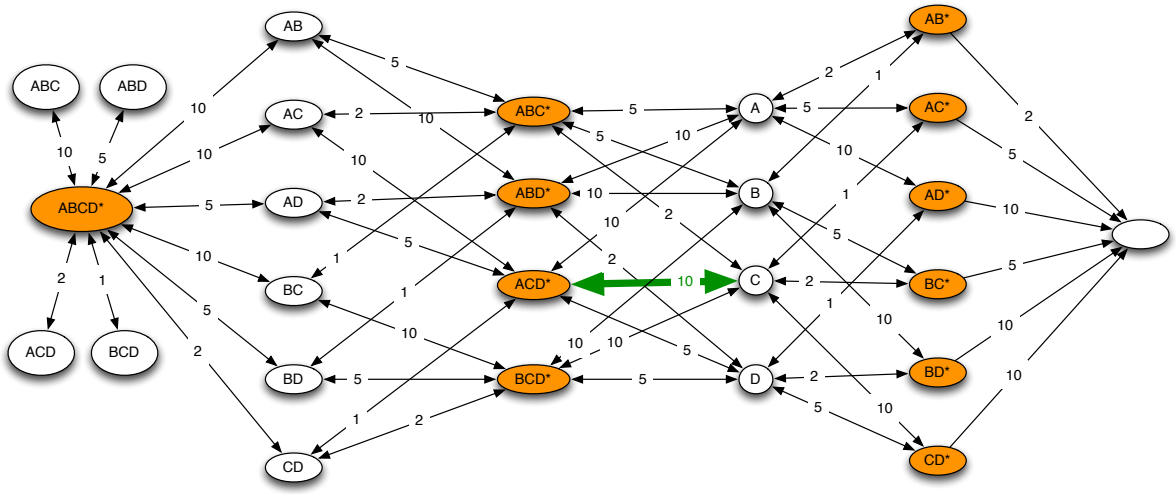


FIGURE 3 – Modélisation du problème sous forme d'un graphe. Les sommets oranges sont ceux où la lampe de poche est du "mauvais" côté du pont. Par exemple, l'arc en gras et vert représente le passage de la configuration ACD^* à celle C, ce qui correspond à la traversée des randonneurs A et D (donc en 10 minutes) d'un côté à l'autre, laissant (ou rejoignant) C du mauvais côté du pont.

Le problème est donc équivalent à celui de trouver un plus court chemin dans le graphe pondéré ci-dessus. L'algorithme de Dijkstra (1959) (voir sur le web et/ou `http://www-sop.inria.fr/members/Nicolas.Nisse/lectures/2weightedGraph.pdf`) permet de calculer un tel plus court chemin en temps $O(m + n \log n)$ (avec n le nombre de sommet et m le nombre d'arêtes du graphe).

Les 26 images suivantes décrivent l'exécution de l'algorithme de Dijkstra sur le graphe du problème de l'exercice 78. Un sommet cerclé de rouge (en gras) avec une étiquette x rouge désigne

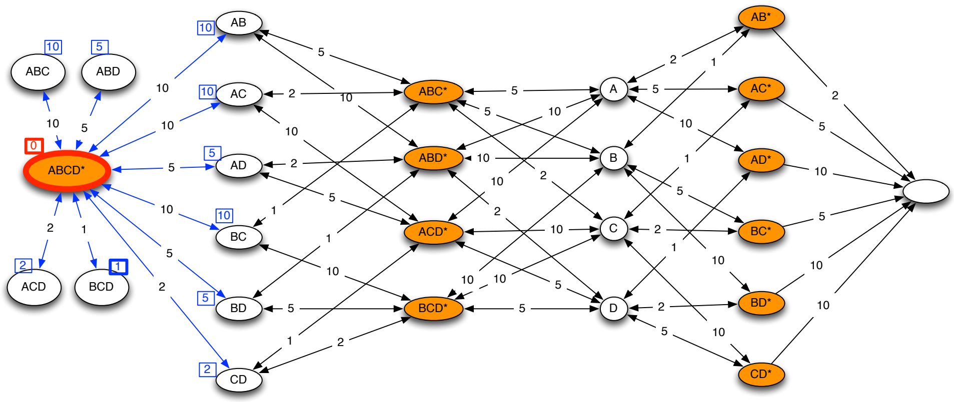
un sommet dont on sait qu'il est à distance x du sommet $ABCD^*$. Un sommet avec une étiquette x bleue désigne un sommet dont on sait qu'il est à distance $\leq x$ du sommet $ABCD^*$. Les arêtes bleues sont celles qu'on a déjà traîtée une fois. Les arêtes en rouge gras sont celles qui peuvent être utilisées pour les plus courts chemins. Finalement, les sommets avec une étiquette x bleue entourée d'un carré gras désignent les prochains sommets potentiellement traîtés (ce sont ceux avec l'étiquette bleue la plus petite). Sur la dernière image, le chemin vert représente la solution optimale (dans cet exercice, elle est unique), de durée 17 minutes. □

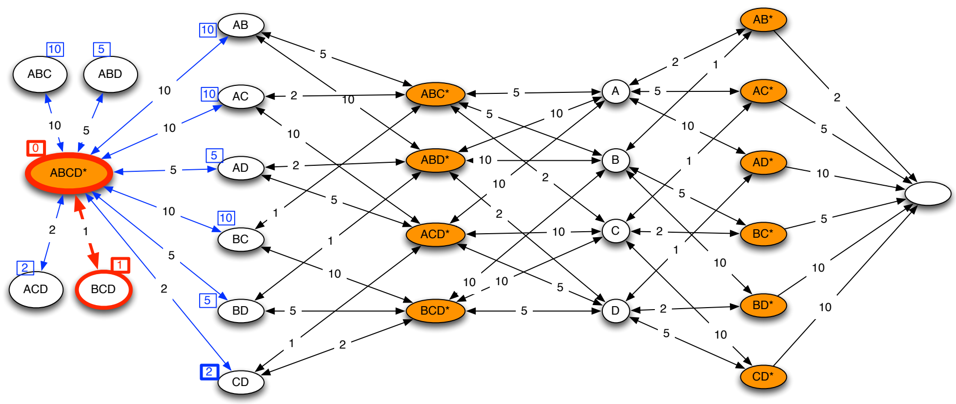
L'exercice suivant est un exemple de problème pour lequel un algorithme glouton est optimal ! En deux mots, il faut trier les arêtes par ordre de poids croissant, puis on parcourt la liste des arêtes par ordre de poids croissant et les ajoutons à la solution si elles ne créent pas de cycle.

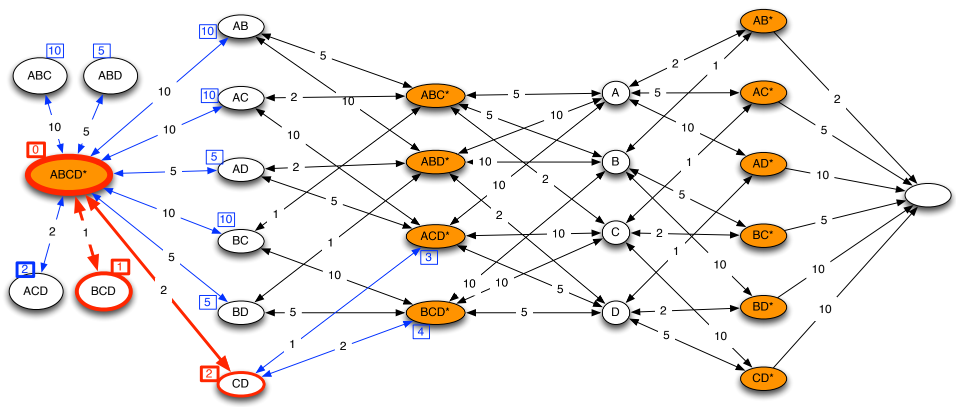
Exercice 79 *Arbre couvrant de poids minimum dans un graphe.*

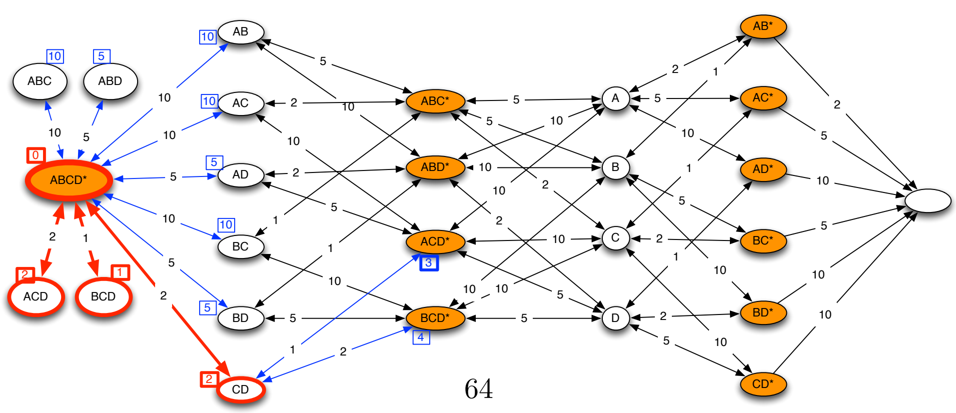
L'algorithme de Dijkstra requis pour l'exercice suivant est à la fois un exemple d'algorithme glouton et d'algorithme par programmation dynamique.

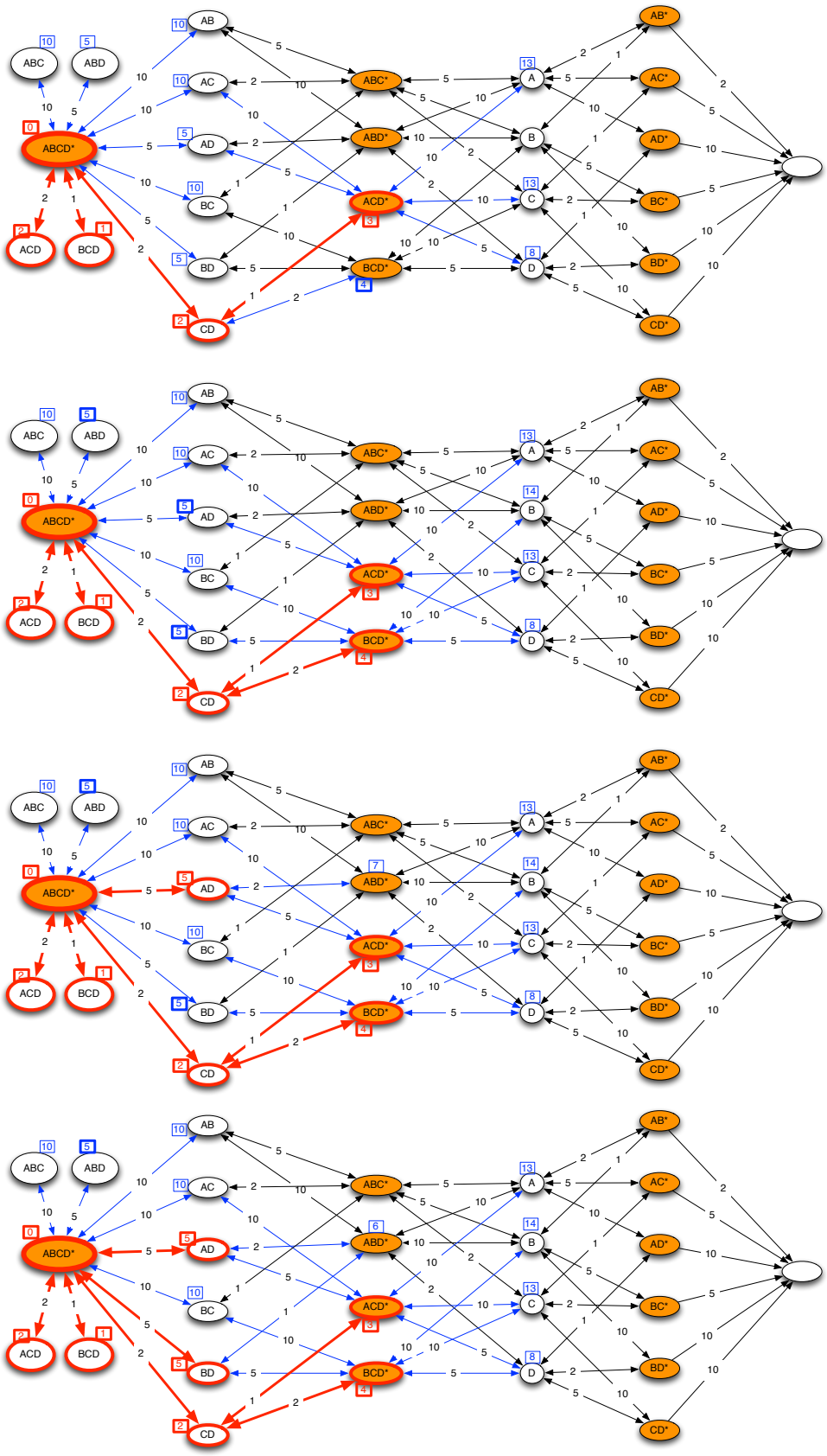
Exercice 80 *Algorithme de plus court chemin dans un graphe.*

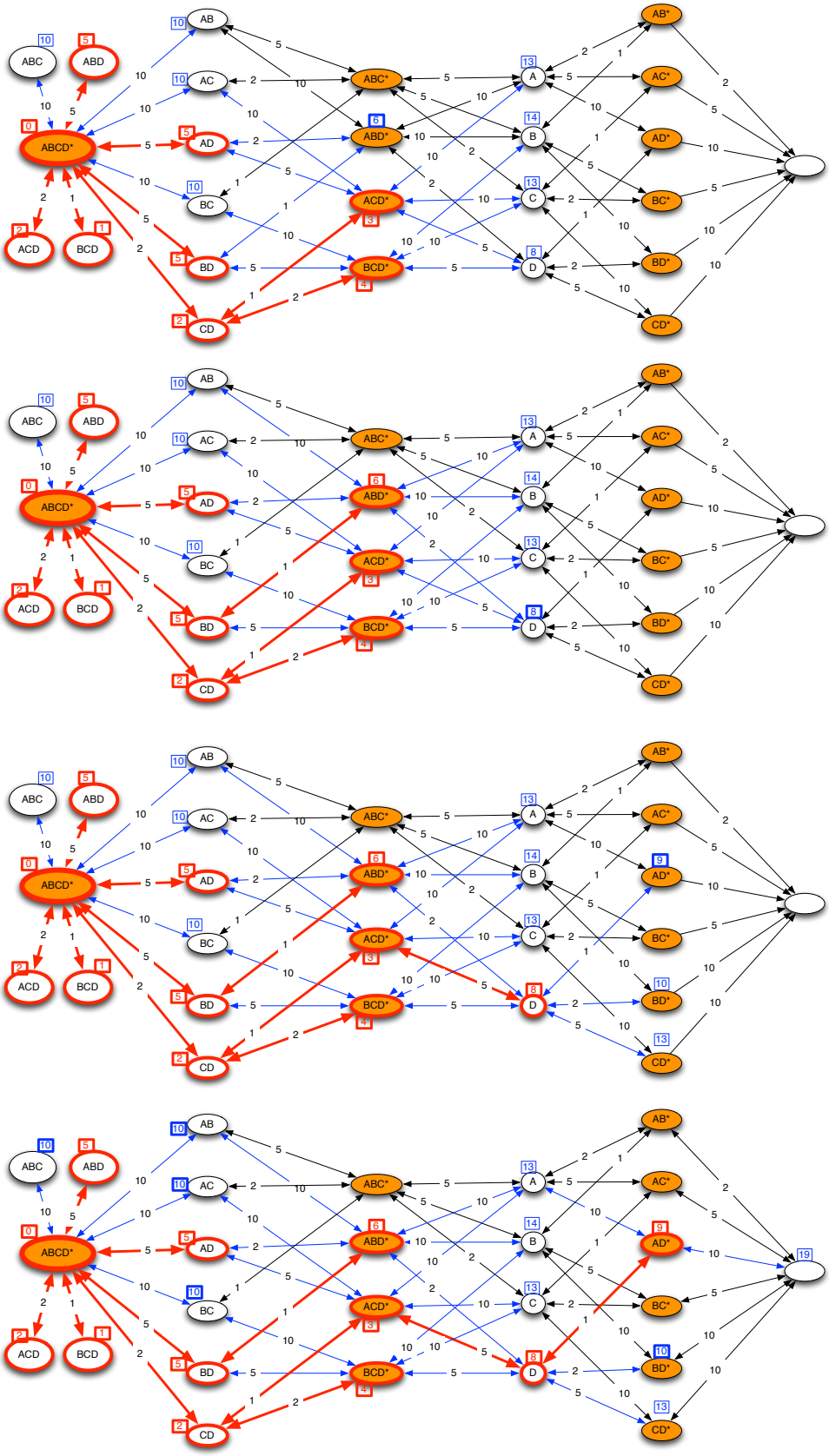

ABC
ABD
ABCD*
ACD
BCD
AB
AC
AD
BC
BD
CD
ABC*
ABD*
ACD*
BCD*
A
B
C
D
AB*
AC*
AD*
BC*
BD*
CD*

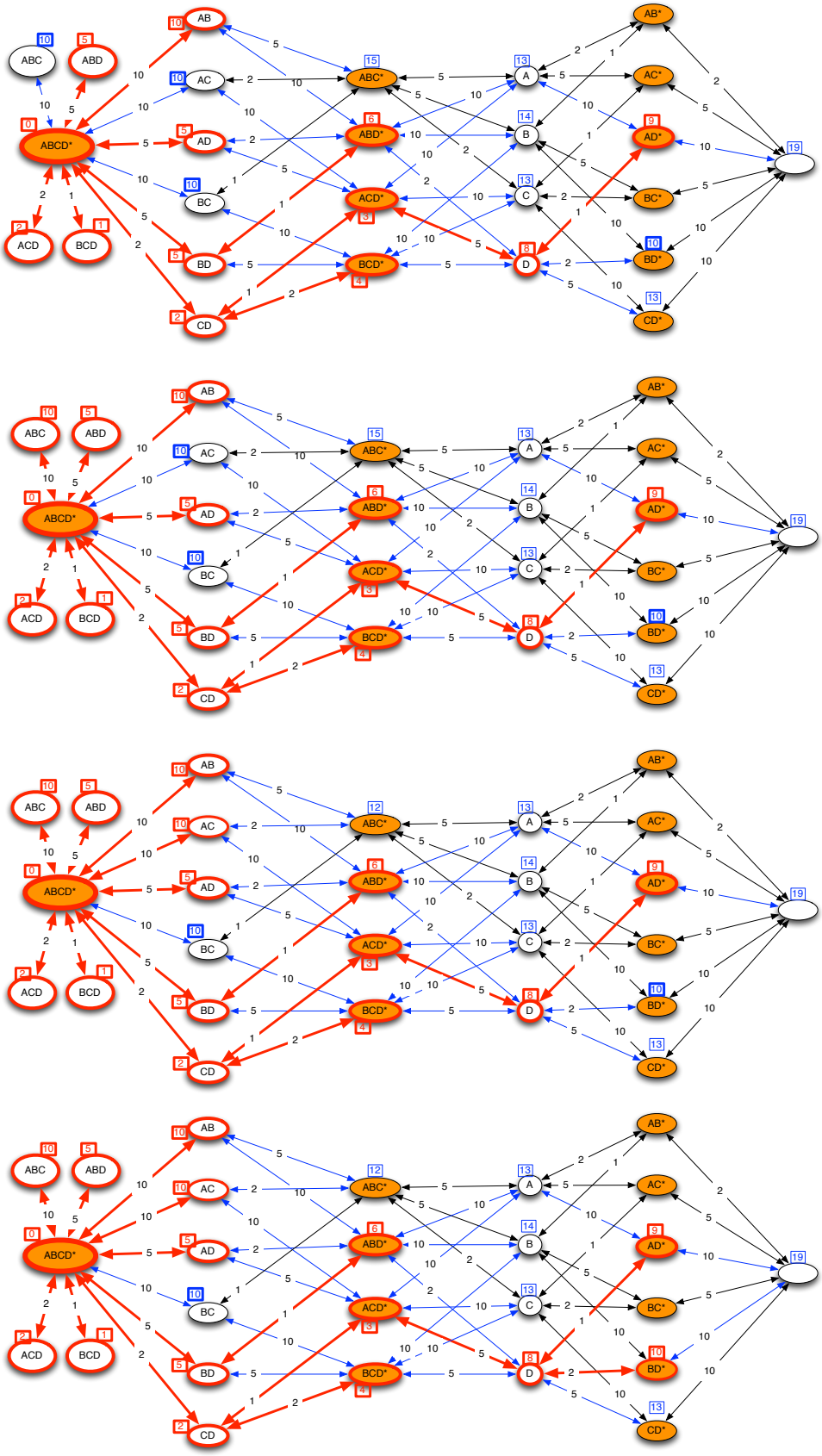

ABC
ABD
ABCD*
ACD
BCD
AB
AC
AD
BC
BD
CD
ABC*
ABD*
ACD*
BCD*
A
B
C
D
AB*
AC*
AD*
BC*
BD*
CD*

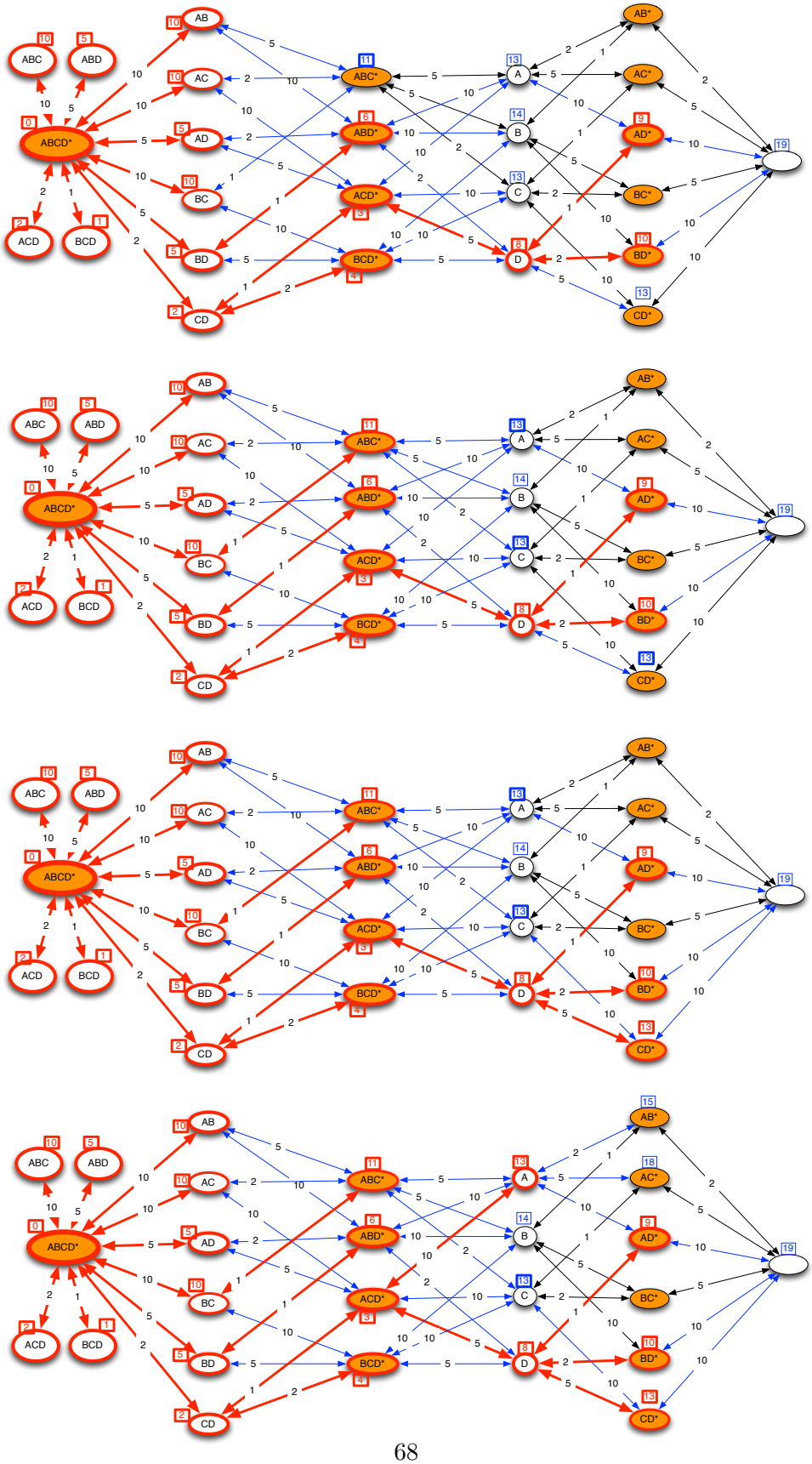

ABC
ABD
ABCD*
ACD
BCD
AB
AC
AD
BC
BD
CD
ABC*
ABD*
ACD*
BCD*
A
B
C
D
AB*
AC*
AD*
BC*
BD*
CD*

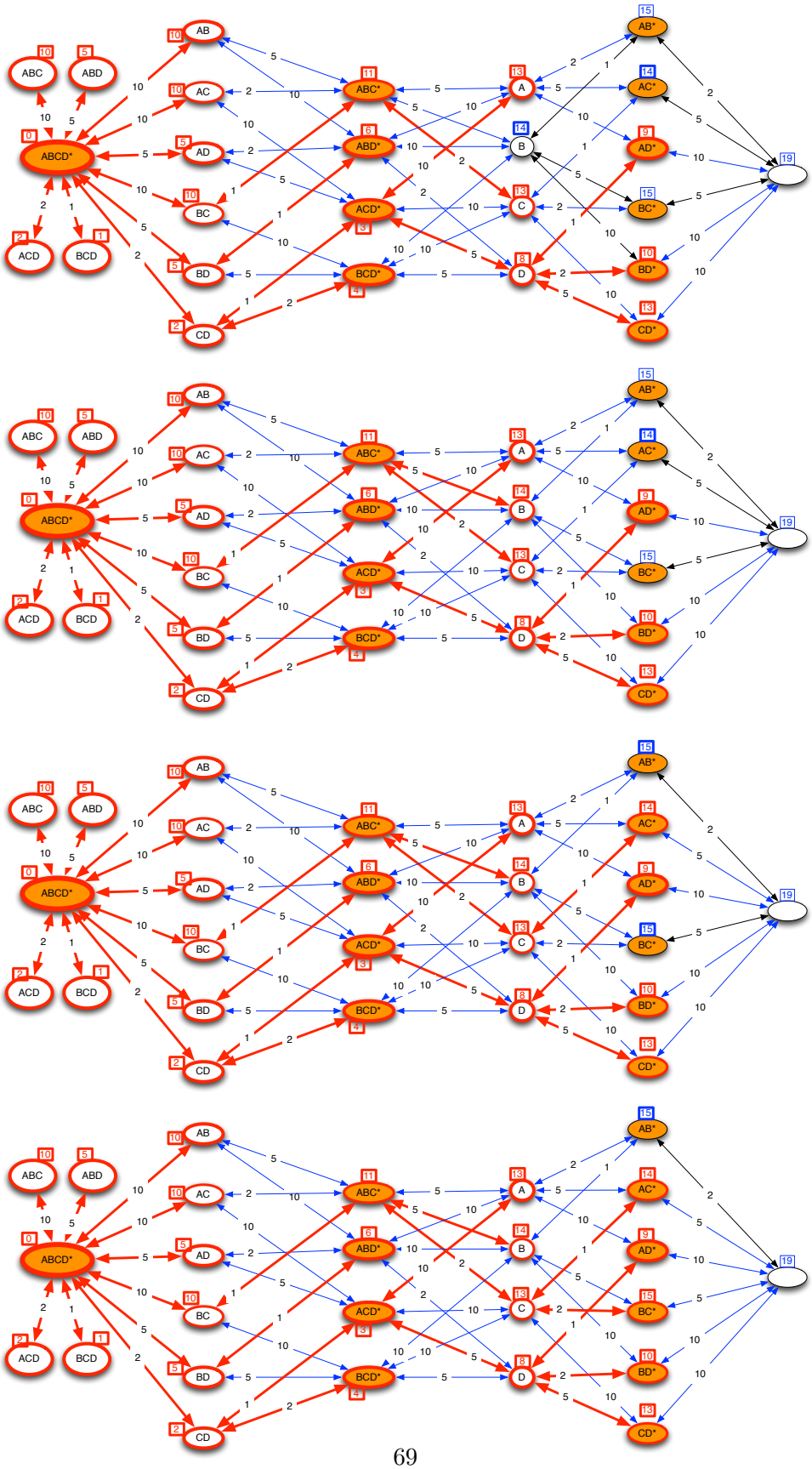

ABC
ABD
ABCD*
ACD
BCD
AB
AC
AD
BC
BD
CD
ABC*
ABD*
ACD*
BCD*
A
B
C
D
AB*
AC*
AD*
BC*
BD*
CD*

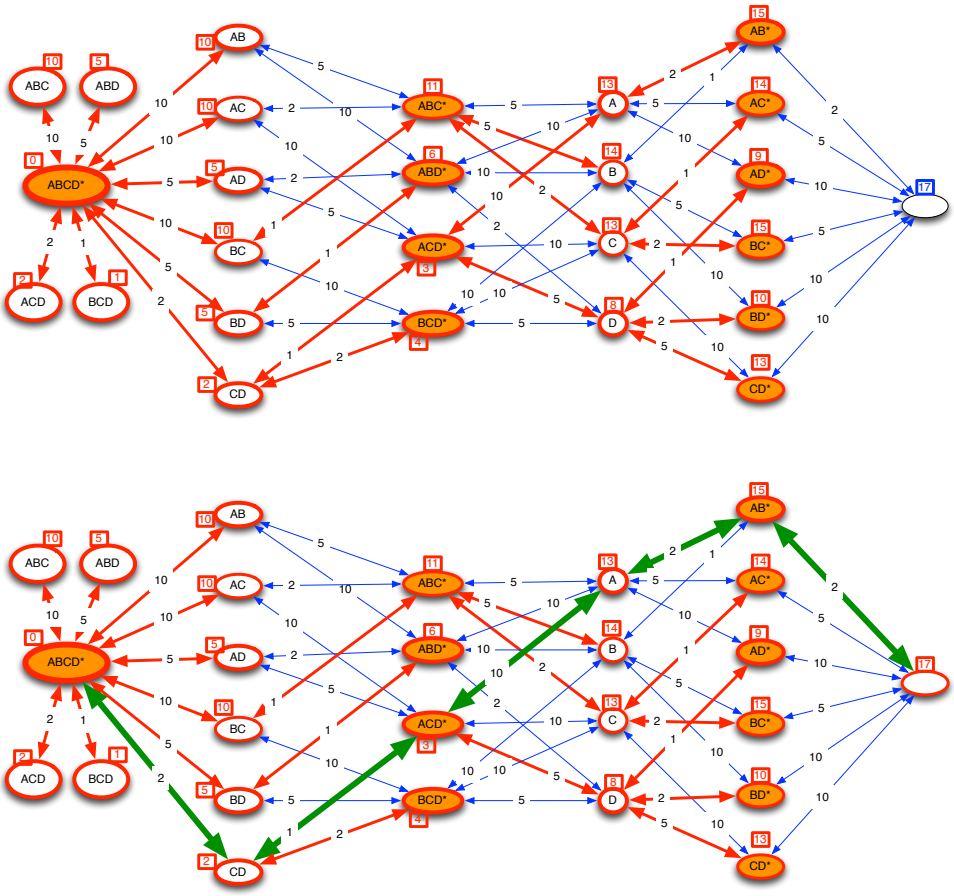

ABC
ABD
ABCD*
ACD
BCD
AB
AC
AD
BC
BD
CD
ABC*
ABD*
ACD*
BCD*
A
B
C
D
AB*
AC*
AD*
BC*
BD*
CD*


ABC
ABD
ABCD*
ACD
BCD
AB
AC
AD
BC
BD
CD
ABC*
ABD*
ACD*
BCD*
A
B
C
D
AB*
AC*
AD*
BC*
BD*
CD*


ABC
ABD
ABCD*
ACD
BCD
AB
AC
AD
BC
BD
CD
ABC*
ABD*
ACD*
BCD*
A
B
C
D
AB*
AC*
AD*
BC*
BD*
CD*


ABC
ABD
ABCD*
ACD
BCD
AB
AC
AD
BC
BD
CD
ABC*
ABD*
ACD*
BCD*
A
B
C
D
AB*
AC*
AD*
BC*
BD*
CD*


ABC
ABD
ABCD*
ACD
BCD
AB
AC
AD
BC
BD
CD
ABC*
ABD*
ACD*
BCD*
A
B
C
D
AB*
AC*
AD*
BC*
BD*
CD*
19


ABC
ABD
ABCD*
ACD
BCD
AB
AC
AD
BC
BD
CD
ABC*
ABD*
ACD*
BCD*
A
B
C
D
AB*
AC*
AD*
BC*
BD*
CD*