Cours d'informatique MPSI

Nicolas Nisse

Inria & Univ. Nice Sophia Antipolis, I3S, CNRS

Informatique. "Science du traitement rationnel, notamment par machines automatiques, de l'information considérée comme le support des connaissances humaines et des communications dans les domaines technique, économique et social" [définition glanée sur Internet]

"Domaine d'activité scientifique, technique et industriel concernant le traitement automatique de l'information par des machines : des calculateurs, des systèmes embarqués, des ordinateurs, des robots, des automates, etc. Les champs d'applications peuvent être séparés en deux branches, l'une pratique pour l'implémentation technique et l'autre théorique, avec la mise en place de concepts et modèles." [Wikipedia, août 2012]

Contenu du cours, en bref: Comprendre/écrire un algorithme. Correction d'algorithmes. Complexité temporelle. Algorithmes récursifs, Diviser pour Régner, Programmation dynamique, arbres binaires.

Table of Contents

1 Bases d'algorithmique

1.1 Les bases

Un **algorithme** est une suite finie et non-ambiguë d'opérations ou d'instructions permettant de résoudre un problème.
Exemples : recette de cuisine, manuel d'instructions de montage (LEGO, meuble IKEA...), indications pour aller d'un endroit à un autre (itinéraire dans Mappy),

Un algorithme décrit un procédé, susceptible d'une réalisation mécanique, pour résoudre un probème donné. Il consiste en une spécification (ce qu'il doit faire) et une méthode (comment il le fait) :

- La **spécification** précise les données d'entrée avec les préconditions que l'on exige d'elles (entre autre le type de donnée en entrée), ainsi que les données de sortie avec les post-conditions que l'algorithme doit assurer (entre autre, le type de sortie). Autrement dit, les préconditions définissent les données auxquelles l'algorithme s'applique, alors que les post-conditions résument le résultat auquel il doit aboutir.
- La **méthode** consiste en une suite finie d'instructions, dont chacune est soit une instruction primitive (directement exécutable sans explications plus détaillées) soit une instruction complexe (qui se réalise en faisant appel à un algorithme déjà défini). En particulier chaque instruction doit être exécutable de manière univoque, et ne doit pas laisser place à l'interprétation ou à l'intuition.

Exemples d'instructions

- opérations arithmétiques élémentaires
- opérations booléenes élémentaires
- affectations de variables
- boucles conditionnelles ou non

Types. Un type de donné, ou simplement type, définit les valeurs que peut prendre une donnée, ainsi que les opérateurs qui peuvent lui être appliqués.
int, boolean, ...
liste et tableau et opérateurs associés
Notions de Paramètres d'entrée, de Variables

1.2 Premiers exemples

- Echange de 2 éléments dans un tableau (introduit variable auxiliaire)

On rappelle que pour un tableau T de taille n, les éléments sont placés entre les positions 0 et $n-1$. On note $T[i]$ le i^{me} élément de T

Algorithm 1 (Echange)

Entrée Un tableau T de longueur n et deux entiers $0 \leq i, j < n$
Sortie Le tableau T dans lequel on a échangé les éléments $T[i]$ et $T[j]$

1. Soit $aux = T[i]$ une variable auxiliaire initialisée avec le i^{me} élément de T
2. faire $T[i] \leftarrow T[j]$
3. faire $T[j] \leftarrow aux$
4. Renvoyer T.

– il faudrait un exemple simple juste avec IF ELSE THEN
– Somme des n premiers entiers (avec FOR, avec WHILE)

Algorithm 2 (Somme)

Entrée Un entier n
Sortie La somme des n premiers entiers $\frac{n(n+1)}{2}$

1. Soit $s = 0$ la somme courante
2. Pour i allant de 0 à n faire
   $s \leftarrow s + i$.
3. Renvoyer s.

– factorielle
– calcul de 2^n
– recherche d'un maximum dans un tableau
– recherche plus petit p tel que $n < 2^p$ ($p \approx logn$)

Algorithm 3 (Logarithme)

Entrée Un entier $n > 1$
Sortie $\lfloor \log_2 n \rfloor$.

1. Soit $puiss = 1$ et $courant = 2$
2. Tant que $courant \leq n$ faire
   $puiss \leftarrow puiss + 1$
   $courant \leftarrow 2 * courant$
3. Renvoyer $puiss - 1$

– écriture décimale d'un entier en base 2
– écriture binaire entière d'un entier

2 Correction d'algorithmes

On dit qu'un algorithme est correct si la méthode fait ce qu'exige la spécification. Plus précisément : un algorithme est **correct** si pour toute donnée d'entrée vérifiant la précondition, la méthode se termine et renvoie une donnée de sortie vérifiant la postcondition.

Attention. La preuve qu'un algorithme est correct comporte toujours deux parties. D'abord il faut montrer que l'algorithme s'arrête pour toute donnée d'entrée valable (terminaison) ; la méthode aboutit alors à un résultat. Ensuite il faut montrer que ce résultat vérifie la postcondition.

Notion de Convergent. Comme nous avons souligné plus haut, pour tout algorithme la preuve de correction commence par une preuve de terminaison. Suivant la méthode en question ceci peut être plus ou moins difficile. Evidemment, si la méthode n'utilise ni de boucles ni d'appels récursifs, alors l'algorithme se termine toujours. (Le justifier.) Ce cas simpliste est rare ; en général il faut trouver un argument plus profond, typiquement une preuve par une récurrence bien choisie.

Mise en évidence d'un convergent, i.e. une quantité qui diminue à chaque passage, vivant dans un ensemble bien fondé (où il n'existe pas de suites infinies strictement décroissantes).
Notion d'Invariant de boucle. Tentative d'intuitions:

- ensemble de propriétés qui relient les variables du programme
- invariant avant, pendant et après la boucle
- vision statique de la boucle
- instantané générique décrivant la situation
- l'itération change la valeur des variables mais pas les relations qui les lient

2.1 Exemples

On reprends les exercices précédents en identifiant les invariants / convergents.

- puissance. Invariant de boucle $sol = x^i$.
- factorielle. Invariant de boucle $sol = \Pi_{j \leq i} j = i!$
- dérivée n^{me}. Invariant de boucle $sol = f^{(i)}(x)$

Nouveaux exemples:

- evaluation d'un polynome

Algorithm 4 (Evaluation lente de Polynôme)

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

1. Soit $v = 0$ la valeur courante
2. Pour i allant de 0 à n faire
   - Soit $y = 1$
   - Soit un compteur $j = 0$
   - Tant que $j < i$ faire
     - faire $y \leftarrow y * x$
     - faire $j \leftarrow j + 1$
   - faire $y \leftarrow y * a_i$
   - faire $v \leftarrow v + y$
3. Renvoyer v.

Algorithm 5 (Méthode de Hörner)

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

1. Soit $v = a_n$
2. Pour i allant de $n - 1$ à 0 faire
   $v \leftarrow v * x + a_i$
3. Renvoyer v.

- tri par selection

Algorithm 6 (Tri par selection)

Entrée Un tableau T de n entiers.
Sortie Le tableau T dans lequel on a ordonné les éléments dans l'ordre corissant

1. Soit $i = 0$ un compteur
2. Tant que $i < n$ faire
   - Soit $minCourant = T[i]$ une variable auxiliaire
   - Soit $posMin = i$ une autre variable auxiliaire
   - Pour j allant de i à $n - 1$ faire
     Si $minCourant > T[j]$ faire
     * $minCourant \leftarrow T[j]$
     * $posMin \leftarrow j$
   - Echange(T, i, j)
   - faire $i \leftarrow i + 1$
3. Renvoyer T

2.2 Exercices

On admet qu'il existe les instructions $sin(x)$ et $cos(x)$ qui renvoient respectivement le sinus et le cosinus de $x \in \mathbb{R}$.

Question 1. Ecrire un algorithme qui prend un entier $n \in \mathbb{N}$ en entrée et retourne $\sum_{i \leq n} sin^2(i)$.

Algorithm 7 (sinus carré)

Entrée $n \in \mathbb{N}$.

1. $s \leftarrow 0$
2. Pour i allant de 1 à n faire
   $s \leftarrow s + sin^2(i)$
3. Retourne s

Question 2. Prouver que votre algorithme est correct.

Correction. Par induction sur $1 \leq i \leq n$, après l'exécution de la i^{me} itération de la boucle "pour", $s = \sum_{j \leq i} sin^2(j)$. Comme il y a n itérations, l'algorithme termine est renvoie $s = \sum_{i \leq n} sin^2(i)$. □

Question 3. Ecrire un algorithme qui prend un entier $n \in \mathbb{N}$ en entrée et retourne $\sum_{i<n}(sin^2(i) + cos^2(i))$.

Algorithm 8 (Incrément)

Entrée $n \in \mathbb{N}$.

1. Retourne $n + 1$

Question 4. L'algorithme suivant est-il correct ? Que fait il ?

Algorithm 9 (Algo)

Entrée $n \in \mathbb{N}$.

1. $p \leftarrow 1$
2. $i \leftarrow 0$
3. Tant que $i \leq n$ faire
   $p \leftarrow p * i$
4. Retourne p

Correction. La boucle "tant que" ne se termine que lorsque $i > n$. Or i est initialisé à 0 et ne change jamais. Donc, l'algorithme ne termine pas. □

Soit x et y deux entiers avec $y > 0$.

Le reste de la division euclidienne de x par y est l'unique entier $0 \leq r < y$ tel que $x = qy + r$ ($q \in \mathbb{N}$). On note r par $x \mod y$ ("x modulo y"). Par exemple, $23 \mod 11 = 1$ puisque $23 = 2 * 11 + 1$. On dit que y est un diviseur de x si $x \mod y = 0$, i.e., si x est un multiple de y.

Le plus grand commun diviseur (pgcd) de deux entiers x et y est le plus grand entier z qui divise à la fois x et y, i.e., $x \mod z = 0$ et $y \mod z = 0$ et z est le plus grand possible avec cette propriété. Par exemple, on peut vérifier que $pgcd(126, 60) = 6$.

Question 5. Appliquer l'algorithme suivant à 126 et 70. On détaillera les valeurs des paramètres et des variables initialement et après chaque itération de la boucle "tant que".

Algorithm 10 (Euclide)

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 $y \neq 0$ faire
   – Soit la variable *temp* initialisée à y
   – faire $y \leftarrow x \mod y$.
   – faire $x \leftarrow temp$.
3. Renvoyer x.

Pour la dernière question, on rappelle que $pgcd(x; y) = pgcd(y; x \mod y)$ où $pgcd(x; y)$ est le plus grand diviseur commun de x et y, et $x \mod y$ est le reste de la division euclidienne de x par y.

Question 6. L'algorithme suivant est-il correct ? Que fait il ?

Correction. On prouve par induction sur le nomber d'itération de la boucle "tant que" que $pgcd(a, b) = pgcd(x, y)$. De plus (x, y) décroit strictement dans l'ordre lexicographique. En

particulier, y est un entier qui décroit strictement. Comme $\mathbb{N}$ est bien fondé, y atteint 0 et lorsque cela arrive, l'algorithme renvoie $x = pgcd(x, 0) = pgcd(x, y) = pgcd(a, b)$. □

Exercice fourbe. Que fait :

- conjecture de Collatz/Syracuse : On part d'un nombre entier plus grand que zéro ; s'il est pair, on le divise par 2 ; s'il est impair, on le multiplie par 3 et on ajoute 1

3 Récursivité

Récursion vient de *recursare* en latin = courir en arrière, revernir. Le récursivité est une démarche qui fait référence à l'objet de la démarche. Définir un concept en invoquant le même concept. Un algorithme est dit *récursif* s'il s'appelle lui même.

Décrire un processus dépendant de données en faisant appel à de même processus sur des données plus simple. Méthode où la solution d'un problème dépend des solutions d'instances plus petites du problème

Cela suppose une relation d'ordre bien fondé dans l'ensemble des paramètres. Si l'ordre n'est pas total, il peut y avoir plusieurs éléments minimaux.

3.1 Encore des exemples

- somme des n premiers entiers / puissance / factorielle / dérivée
- Exponentiation rapide $x^n = x^{n/2} * x^{n/2}$
- Fibonnacci
  $f_0 = 1$, $f_1 = 1$
  for any $n > 1$, $f_n = f_{n-1} + f_{n-2}$.
- Récursivité sur $\mathbb{N}^2$: Ackermann:
  $a_{0,0} = 1$
  $a_{0,p} = a_{0,p-1} + 1 = p + 1$
  $a_{n,0} = a_{n-1,1}$
  $a_{n,p} = a_{n-1,a_{n,p-1}}$.
- recherche dichotomique dans un tableau
- decider palindrome
- le nombre de partitions d'un entier naturel en au plus q parties
- Tours de Hanoi
- Tri-fusion

4 Complexité temporelle

Notions de complexité temporelle/spatiale. Taille de l'entrée (nombre d'entiers, nombre de bits...?), notion d'opération élémentaire (opérations arithmétiques, comparaisons, assignation de variables...?).
Notion de pire cas.
(rappel ?) de Notations. O, o, Ω, Θ

Pour mesurer la complexité temporelle d'un algorithme, on *évalue* la quantité "d'opérations élémentaires" réalisées par l'algorithme en fonction de la taille de l'entrée. Evaluer signifie ici compter exactement le nombre d'opérations élémentaires, ou en donner une borne supérieure. *Comparer* deux algorithmes revient donc à déterminer le plus efficace, c'est-à-dire celui qui effectue le moins d'opérations élémentaires pour réaliser la même tâche.

On reprends les exercices précédents en explicitant leur complexité.

En particulier, comparer evaluation lente de polynôme et méthode de Hörner.

Question 7. Comparer les deux algorithmes 1 et 2. En particulier, détailler le nombre d'opérations élémentaires de chacun des algorithmes en fonction de leur entrée n, ainsi que le résultat calculé.

Algorithm 11

Entrée $n \in \mathbb{N}$.

1. Soit $res = 0$
2. Soit $i = 0$.
3. Tant que $i \leq n$ faire
   $res \leftarrow res + i$.
   $i \leftarrow i + 1$.
4. Renvoyer res.

Algorithm 12

Entrée $n \in \mathbb{N}$.

1. Renvoyer $\frac{n(n+1)}{2}$.

Correction. Alg. 1 effectue 3 opérations à chaque itération de la boucle (un test, deux additions), soit $3(n+1)$ opérations pour calculer $\sum_{0 \leq i \leq n} i = n(n+1)/2$. Alg. 2 effectue 3 opérations pour le même résultat. □

4.1 Exemple amusant

On considère une ligne de longueur n (ou un chemin (graphe) à n sommets). On part du milieu, noté c. Le but est de trouver un objet caché sur la ligne. On note d la distance de c à l'objet. d est inconnu.

Algorithme 1: on part d'un côté, si on trouve l'objet, on gagne, sinon on va jusqu'au bout du chemin et on fait demi tour jusqu'à trouver l'objet. Complexité en $n + d$.

Algorithme 2: Tant qu'on ne trouve pas l'objet, on va à distance 1 de c à droite, puis à distance 2 de c à gauche, puis à distance 4 de c à droite,..., puis à distance 2^{2i} de c à droite, puis à distance 2^{2i+1} de c à gauche... Complexité $O(9d)$.

4.2 Exponentiation rapide

Soit $x \in \mathbb{R}_+^*$. Nous définissons la suite $(u_n)_{n \in \mathbb{N}}$ telle que, pour tout $n \in \mathbb{N}$, $u_n = x^n$.

Question 8. Ecrire un algorithme **non récursif** qui prend en entrée un entier n et renvoie x^n.

Algorithm 13 (Puissance)

Entrée $n \in \mathbb{N}$.

1. Soit $res = 1$
2. Pour i allant de 1 à n faire
   $res \leftarrow x * res$.
3. Renvoyer res.

Question 9. Définir u_n en fonction de u_{n-1}.

Correction. $u_0 = 1$ et, pour tout $n \in \mathbb{N}$, $u_{n+1} = x * u_n$. □

Question 10. Quel est le résultat de l'algorithme $PuissanceRec_1$? Prouver qu'il est correct.

Algorithm 14 ($PuissanceRec_1$)

Entrée $n \in \mathbb{N}$.

1. Si $n = 0$, renvoyer 1.
2. Sinon, renvoyer $x * PuissanceRec_1(n-1)$.

Correction. Par récurrence sur n, l'algorithme termine et renvoie u_n. Si $n = 0$, c'est trivialement vrai. Supposons que c'est vrai pour $0 \leq n$. Alors $PuissanceRec_1(n+1)$ teste si $n+1 = 0$ ce qui est faux. Puis, $PuissanceRec_1(n)$ est exécuté et par hypothèse de récurrence, termine et renvoie u_n. Finallement, $x * u_n = u_{n+1}$ est renvoyé. □

Question 11. Soit $c(n)$ le nombre d'opérations élémentaires réalisées lors de l'exécution de $PuissanceRec_1(n)$. Exprimer $c(n+1)$ en fonction de $c(n)$. En déduire l'expression de $c(n)$ pour tout $n \in \mathbb{N}$.

Correction. $PuissanceRec_1(0)$ effectue un test, donc $c(0) = 1$. Lors de l'exécution de $PuissanceRec_1(n+1)$, un test est effectué, puis l'exécution de $PuissanceRec_1(n)$ dont le résultat est multiplié par x. Donc $c(n+1) = c(n) + 2$.

Par récurrence sur n, $c(n) = 1 + 2n$. □

On veut proposer et analyser un autre algorithme pour le calcul de x^n.

Question 12. Soit un entier pair $n = 2k$. Exprimer u_n en fonction de u_k.

Supposons que $n = 2k+1$ est impair. Exprimer u_n en fonction de u_k et de x.

Correction. $u_0 = 1$ et, pour tout $n \in \mathbb{N}^*$, si $n = 2k$, alors $u_n = u_k * u_k$ et si $n = 2k+1$ alors $u_n = x * u_k * u_k$. □

Question 13. Ecrire un algorithme récursif $PuissanceRec_2$ qui prend en entrée un entier n et calcule x^n avec un **unique** appel récursif à $PuissanceRec_2(\lfloor \frac{n}{2} \rfloor)$

Algorithm 15 ($PuissanceRec_2$)

Entrée $n \in \mathbb{N}$.

1. Si $n = 0$, renvoyer 1.
2. Sinon, soit $temp = PuissanceRec_2(\lfloor \frac{n}{2} \rfloor)$
   - Si n est pair renvoyer $temp * temp$
   - Sinon renvoyer $x * temp * temp$

Question 14. Soit $d(n)$ le nombre d'opérations élémentaires effectuées par l'exécution de $PuissanceRec_2(n)$.
Que vaut $d(0)$?
Prouver qu'il existe une constante $\gamma \in \mathbb{N}$ telle que, pour tout $n \in \mathbb{N}$, $d(n) \leq d(\lfloor \frac{n}{2} \rfloor) + \gamma$.

Correction. Pour l'algorithme proposé, $\gamma = 4$ (deux tests et une ou deux multiplications). □

Question 15. Soit $n = 2^p$, $p \in \mathbb{N}$. Prouver que $d(n) \leq \gamma p + 1$.

Correction. Par récurrence sur p. Si $p = 0$, $n = 1$ et $d(1) \leq d(0) + \gamma = 1 + \gamma$. Supposons $d(2^p) \leq \gamma p + 1$. D'après la question précédente, $d(2^{p+1}) \leq d(\lfloor \frac{n}{2} \rfloor) + \gamma = d(2^p) + \gamma \leq \gamma p + 1 + \gamma = \gamma(p+1) + 1$. □

Question 16. Montrer que la suite $(d(n))_{n \in \mathbb{N}}$ est croissante.

Correction. Direct par définition de l'algorithme $PuissanceRec_2$. □

Question 17. En déduire que, pour tout $n \in \mathbb{N}$, $d(n) \leq \gamma \lceil \log_2 n \rceil + 1$.
($\log_2$ représente le logarithme en base 2)

Correction. Soit $p \in \mathbb{N}$ tel que $2^{p-1} < n \leq 2^p$. Alors $p - 1 < \log n \leq p$ et $p = \lceil \log n \rceil$. Puisque $d(n)$ est croissante (question précédente) et d'après la question d'avant, $d(n) \leq d(2^p) \leq \gamma p + 1 = \gamma \lceil \log n \rceil + 1$. □

Question 18. Comparer les algorithmes $PuissanceRec_1$ et $PuissanceRec_2$.

Correction. $c(n) \geq 2^{d(n)}$ donc $PuissanceRec_2$ est plus efficace. □

4.3 Interlude : Suite Récurrence affine

Theorem 1. *Soit $(u_n)_{n \in \mathbb{N}}$ une suite définie par $u_0 = O(1)$ et $u_n = a * u_{n-1} + O(b^n)$.*

- *si $a = b$ alors $u_n = O(na^n)$*
- *si $a > b$ alors $u_n = O(a^n)$*
- *si $b > a$ alors $u_n = O(b^n)$.*

4.4 Recherche d'extrema dans un tableau

Dans ce problème, nous considérons des tableaux dont les éléments sont des entiers naturels. C'est-à-dire, pour tout tableau T de longueur $n \in \mathbb{N}^*$, $T.(i) \in \mathbb{N}$ pour tout $0 \leq i < n$.

Le but de l'exercice est de proposer un algorithme efficace qui, étant donné un tableau T, renvoie le maximum et le miminum contenus dans le tableau. Plus précisément, étant donné un tableau T de longueur n, l'algorithme doit retourner la paire d'entiers $(min_T; Max_T) = (\min_{0\leq i<n} T.(i); \max_{0\leq i<n} T.(i))$.

La complexité des algorithmes de ce problème est exprimée en terme de nombre de comparaisons d'entiers.

Question 19. Ecrire en CAML un algorithme itératif (avec une boucle "for") $extr(T)$ qui prend une entrée un tableau T et renvoie la paire $(min_T; Max_T)$.

Question 20. Prouver que le nombre de comparaisons réalisées par l'algorithme $extr(T)$ proposé à la question précédente, est équivalent à $2n$, avec n la longueur de T.

Soient T un tableau de longueur $n \in \mathbb{N}^*$, $0 \leq a \leq b < n$. Considérons l'algorithme suivant:

Algorithm 16

```
Let rec extr2 T a b =
    if a == b then (T.(a), T.(a))
    else
        if a == b − 1 then
            if T.(a) < T.(b) then (T.(a), T.(b)) else (T.(b), T.(a))
        else
            let m = (a + b)/2 in
            let (x, y) = extr2 T a m in
            let (u, v) = extr2 T m+1 b in
            (min x u , max b v);;
```

Question 21. Que calcule l'algorithme précédent ? Prouvez qu'il termine et qu'il est correct (on supposera que $0 \leq a \leq b < n$).

Question 22. Soit $x = b - a$ et posons u_x le nombre de comparaisons réalisées lors de l'appel de extr2 T a b.

Exprimer la relation de récurrence satisfaite par $(u_x)_{x\geq 0}$.

Question 23. Donner explicitement l'expression de $(u_x)_{x\geq 0}$ lorsque $x = 2^p$.

On posera $v_p = u_{2^p}$ pour tout $p \geq 0$ et on étudiera la suite $(v_p + 2)_{p\geq 0}$.

Question 24. Donner un algorithme, plus efficace de celui de la question 1, qui calcule (min_T, max_T).

5 Multiplication de pôlynomes

Soit $\mathbb{C}[X]$ l'ensemble des polynômes à coefficients dans $\mathbb{C}$. Un polynôme $P(X) = \sum_{0\leq i<n} a_i X^i \in \mathbb{C}[X]$ est représenté par le vecteur $[a_0, \cdots, a_{n-1}]$ de ses coefficients ($\forall i \in \{0, \cdots, n-1\}, a_i \in \mathbb{C}$).

On rappelle le théorème suivant :

Theorem 2. *Soit $P(X) \in \mathbb{C}[X]$ de degré au plus $n \in \mathbb{N}$ et soient $(x_0, \cdots, x_n) \in \mathbb{C}^{n+1}$ des complexes deux-à-deux distincts. Alors $P(X) = 0$ (le polynôme nul) si et seulement si $P(x_i) = 0$ pour tout $i \leq n$.*

On étudie ici une méthode pour multiplier rapidement deux polynômes. On transforme les données du problème (deux polynômes) en objets faciles à manipuler (des racines de l'unité) sur lesquels on effectue l'opération considérée puis on effectue la transformation inverse sur le résultat obtenu.

Par compléxité (temporelle), on désigne une estimation ("grand O") du nombre d'opérations élémentaires : multiplication, division, addition et soustraction de nombres complexes. Le parcours d'un vecteur de longueur $n \in \mathbb{N}$ est $O(n)$.

5.1 Préliminaires

Soient $p \in \mathbb{N}^*$ et $n = 2^p$. Soit $P \in \mathbb{C}[X]$ représenté par le vecteur $[a_0, \cdots, a_n]$.

Question 25. Prouver qu'il existe une unique paire $(A(X), B(X)) \in \mathbb{C}[X]^2$ de polynômes de degré $\leq n/2$ tels que $P(X) = X^{n/2}A(X) + B(X)$.

Proof. **Existence.** $P(X) = \sum_{0 \leq i < n} a_i X^i = (\sum_{0 \leq i < n/2} a_i X^i) + X^{n/2} \sum_{0 \leq i < n/2} a_{i+(n/2)} X^i$.
Donc, en posant, $A(X) = \sum_{0 \leq i < n/2} a_i X^i$ et $B(X) = \sum_{0 \leq i < n/2} a_{i+(n/2)} X^i$, on obtient $P(X) = A(X) + X^{n/2}B(X)$.
Unicité. Supposons qu'il existe deux paires de polynômes $(A(X), B(X))$ et $(A'(X), B'(X))$ satisfaisant la propriété désirée. Posons $A(X) = \sum_{0 \leq i < n/2} \alpha_i X^i$, $B(X) = \sum_{0 \leq i < n/2} \beta_i X^i$, $A'(X) = \sum_{0 \leq i < n/2} \alpha'_i X^i$ et $B'(X) = \sum_{0 \leq i < n/2} \beta'_i X^i$.
Soit $Q(X) = A(X) + X^{n/2}B(X) - A'(X) - X^{n/2}B'(X) = (\sum_{0 \leq i < n/2} (\alpha_i - \alpha'_i) X^i) + \sum_{0 \leq i < n/2} (\beta_i - \beta'_i) X^{i+(n/2)}$. Par définition de A, A', B et B', $Q(x) = 0$ pour tout $x \in \mathbb{C}$. Donc, d'après le Théorème 2, $Q(X)$ est le polynôme nul. On en déduit que, pour tout $0 \leq i < n/2$, $\alpha_i = \alpha'_i$ et $\beta_i = \beta'_i$. Donc, $A(X) = A'(X)$ et $B(X) = B'(X)$.

Question 26. Donner les représentations des polynômes A et B tels que $P(X) = X^{n/2}A(X) + B(X)$.

Quelle est la complexité du calcul des représentations des polynômes A et B en fonction de la représentation de P ?

Proof. D'après la preuve précédente, $A(X)$ est représenté par $[a_0, \cdots, a_{(n/2)-1}]$ et $B(X)$ est représenté par $[a_{n/2}, \cdots, a_{n-1}]$. Calculer les représentations de $A(X)$ et $B(X)$ demande seulement un parcours de la représentation de P. Donc, la complexité est $O(n)$.

Algorithm 1 Méthode de Horner

Require: polynôme $P(X) = \sum_{0 \leq i \leq n} a_i X^i \in \mathbb{C}[X]$ et $x \in \mathbb{C}$
1: $z \leftarrow a_n$
2: **for** $i = n - 1$ à 0 **do**
3: $\quad z \leftarrow a_i + x * z$
4: **end for**
5: **return** z

Question 27. On considère l'algorithme 1 (méthode de Horner).

– Expliquer ce qu'il calcule.
– Prouver sa correction.
– Donner sa compléxité en fonction du degré $n \geq 0$ du polynôme.

Proof. L'algorithme de Horner évalue $P(x)$, c'est-à-dire la **valeur** de $P(X)$ en $x \in \mathbb{C}$ donné.

Posons z_0 la valeur de la variable z après la ligne 1 (avant la boucle) et z_i la valeur de la variable z après la i^{me} itération. Alors, on prouve par récurrence que, pour tout $0 \leq i \leq n$, $z_i = \sum_{n-i \leq j \leq n} a_j x^{j+i-n}$. En particulier, pour $i = n$, on obtient $z_n = \sum_{0 \leq j \leq n} a_j x^j = P(x)$ ce qui est la valeur renvoyée par l'algorithme.

Il s'agit d'une boucle **for** dont chaque itération fait une somme, une multiplication et une affectation. Il y a n itérations, donc l'algorithme termine arpès $O(n)$ itérations.

5.2 Méthode de Karatsuba

Soient $P, Q \in \mathbb{C}[X]$ de degré au plus $n = 2^p$. Soient A, B, C et D les quatre polynômes de degré au plus $n/2$ tels que $P(X) = X^{n/2}A(X) + B(X)$ et $Q(X) = X^{n/2}C(X) + D(X)$. On rappelle ici que la méthode de Karatsuba pour multiplier deux polynômes P et Q utilise le fait que

$$P(X)Q(X) = X^n(A(X)C(X)) + X^{n/2}((A(X)+B(X))(C(X)+D(X)) - A(X)C(X) - B(X)D(X)) + B(X)D(X).$$

Soit $c(p)$ le nombre d'opérations élémentaires pour calculer le produit de deux polynômes de degré $n = 2^p$ par la méthode de Karatsuba.

Question 28. Exprimer $c(p)$ en fonction de $c(p-1)$. En déduire $c(p)$ (sans preuve).

Proof. Soient $U(X) = A(X)C(X)$, $V(X) = B(X)D(X)$ et $W(X) = (A(X) + B(X))(C(X) + D(X))$. $P(X)Q(X) = X^n U(X) + X^{n/2}(W(X) - U(X) - V(X)) + V(X)$. Chacun des polynômes U, V et W est le résultat de la multiplication de deux polynômes de degré au plus 2^{p-1}, c'est-à-dire, que chacun de ces polynômes est obtenu en temps $c(p-1)$. Le produit PQ résulte d'un nombre constant de sommes et différences de polynômes de degré au plus 2^p et de produits par $X^{n/2}$ ou X^n et donc peut être réalisé en temps $O(2^p)$. Donc, $c(p) = 3c(p-1) + O(2^p)$ donc $c(p) = O(3^p) = O(n^{\log_2 3}) \approx O(n^{1,585})$.

5.3 Alternative pour la représentation de polynômes

Soit $n \in \mathbb{N}$. Soient $n+1$ nombres complexes $Y = (y_0, \cdots, y_n) \in \mathbb{C}^{n+1}$ deux-à-deux distincts. Soient P et $Q \in \mathbb{C}[X]$ de degré au plus n.

Question 29. Prouver que $P = Q$ si et seulement si P et Q ont même degré et $P(y_i) = Q(y_i)$ pour tout $0 \leq i \leq n$.

Proof. $P = Q$ si et seulement si $P - Q = 0$. Le résultat découle donc directement du Théorème 2.

D'après la question précédente, un polynôme P de degré $n \geq 0$ est uniquement défini par ses valeurs en $n+1$ points distincts, c'est-à-dire que, pour tout $Y = (y_0, \cdots, y_n) \in \mathbb{C}^{n+1}$ et, pour tout $0 \leq i, j \leq n$, $y_i = y_j \Leftrightarrow i = j$, P est uniquement défini par $P\{Y\} = (P(y_0), \cdots, P(y_n))$.

Question 30. Proposer un algorithme qui calcule $P.Q\{Y\}$ en fonction de $P\{Y\}$ et $Q\{Y\}$ en $O(n)$ opérations élémentaires.

Proof. Soient $P\{X\} = (a_0, \cdots, a_n)$ et $Q\{X\} = (b_0, \cdots, b_n)$. Alors, $P.Q\{X\} = (P.Q(x_0), \cdots, P.Q(x_n)) = (P(x_0)Q(x_0), \cdots, P(x_n)Q(x_n)) = (a_0 b_0, \cdots, a_n b_n)$.

Soit $Y \in \mathbb{C}^{n+1}$. Soient $f : \mathbb{N} \to \mathbb{N}$ et $g : \mathbb{N} \to \mathbb{N}$.

Supposons qu'il existe un algorithme $\mathcal{A}_1$ qui permet de calculer $P\{Y\}$ en fonction de a en temps $f(n)$ et qu'il existe un algorithme $\mathcal{A}_2$ qui permet de calculer a en fonction de $P\{Y\}$ en temps $g(n)$, pour tout polynôme $P(X) = \sum_{0 \leq i \leq m} a_i X^i \in \mathbb{C}[X]$, représenté par le vecteur $a = [a_0, \cdots, a_m]$ avec $m \leq n$.

Question 31. Soient $P(X) = \sum_{0 \leq i \leq n/2} a_i X^i \in \mathbb{C}[X]$ et $Q(X) = \sum_{0 \leq i \leq n/2} b_i X^i \in \mathbb{C}[X]$. Proposer un algorithme qui calcule $P.Q(X) = \sum_{0 \leq i \leq n} c_i X^i$ en temps $O(f(n) + g(n) + n)$.

Proof. Soit $Y \in \mathbb{C}^{n+1}$ tels que les fonctions f et g ci-dessus sont définies.

En utilisant l'algorithme $\mathcal{A}_1$, on calcule $P\{Y\}$ en fonction de $a = [a_0, \cdots, a_{n/2}]$ en temps $f(n)$.

En utilisant l'algorithme $\mathcal{A}_1$, on calcule $Q\{Y\}$ en fonction de $b = [b_0, \cdots, b_{n/2}]$ en temps $f(n)$.

En utilisant l'algorithme proposé à la Question 6, $P.Q\{Y\}$ est calculé en temps $O(n)$.

Enfin, en utilisant l'algorithme $\mathcal{A}_2$, on calcule les coefficients de $P.Q(X)$ en fonction de $P.Q\{Y\}$ en temps $g(n)$.

Question 32. Quels doivent être les ordres de grandeur de f et g pour que l'algorithme précédent soit plus efficace que l'algorithme de Karatsuba ?

Proof. L'algorithme a une complexité de $h(n) = O(f(n)+g(n)+n)$. Il est plus efficace que celui de Karatsuba si $h(n) = O(n^{\log_2 3})$. Donc si il existe $c < \log_2 3 \approx 1,585$ telle que $f(n) = O(n^c)$ et $g(n) = O(n^c)$.

Dans la suite, on explique comment concevoir les algorithmes $\mathcal{A}_1$ et $\mathcal{A}_2$ pour "battre" l'algorithme de Karatsuba. Plus précisemment, en choisissant astucieusement $Y \in \mathbb{C}^{n+1}$, pour tout polynôme $P(X) = \sum_{0 \leq i < \leq n} a_i X^i \in \mathbb{C}[X]$, il est possible de "passer" efficacement des coefficients $[a_0, \cdots, a_n]$ à $P\{Y\}$ et réciproquement.

5.4 Transformée de Fourrier

Soit $p \in \mathbb{N}$ et $n = 2^p$.

Soit $Y = (y_0, \cdots, y_{n-1})$ la suite des racines n-ièmes de l'unité, c'est-à-dire $y_k = e^{2ik\pi/n}$, $k < n$. Soit $Z = (z_0, \cdots, z_{n-1}) \in \mathbb{C}^n$ et soit $P \in \mathbb{C}[X]$ le polynôme de degré $< n$ tel que $P\{Y\} = Z$. Soit $Z_0 = (z_0, z_2, z_4, \cdots, z_{n-2})$ et $Z_1 = (z_1, z_3, \cdots, z_{n-1})$. Soit $Y_0 = (y_0, y_2, y_4, \cdots, y_{n-2})$ et $Y_1 = (y_1, y_3, \cdots, y_{n-1})$. Soient P_0 et $P_1 \in \mathbb{C}[X]$ de degré $< 2^{p-1} = n/2$ tels que $P_0\{Y_0\} = Z_0$ et $P_1\{Y_0\} = Z_1$.

Pour la question suivante, on rappelle que $y_k^{n/2} = 1$ si k est pair et $y_k^{n/2} = -1$ sinon.

Question 33. Prouver que, pour tout $x \in \mathbb{C}$, $P(x) = \frac{1+x^{n/2}}{2} P_0(x) + \frac{1-x^{n/2}}{2} P_1(e^{-2i\pi/n} x)$.

Proof. On montre que pour tout $k < n$, $P(y_k) = \frac{1+y_k^{n/2}}{2} P_0(y_k) + \frac{1-y_k^{n/2}}{2} P_1(e^{-2i\pi/n} y_k)$. D'après le Théorème 2, on en déduit que les polynômes P et $Q(X) = \frac{1+X^{n/2}}{2} P_0(X) + \frac{1-X^{n/2}}{2} P_1(e^{-2i\pi/n} X)$ (de degré $< n$) sont égaux.

Soit $k < n$. Si k est pair, $y_k^{n/2} = 1$ et $P_0(y_k) = z_k$ et si k est impair, $x_k^{n/2} = -1$ et $P_1(e^{-2i\pi/n} y_k) = P_1(y_{k-1}) = z_k$.

Question 34. Donner un algorithme qui calcule les coefficients de P en fonction des coefficients de P_0 et P_1.

Proof. Soient $[\alpha_0, \cdots, \alpha_{n/2}]$ les coefficients de P_0 et $[\beta_0, \cdots, \beta_{n/2}]$ les coefficients de P_1.

D'après la question précédente et le Théorème 2, $P(X) = \frac{1+X^{n/2}}{2} P_0(X) + \frac{1-X^{n/2}}{2} P_1(e^{-2i\pi/n} X)$.

Donc $P(X) = (\sum_{0 \leq j < n/2} \frac{\alpha_j + \beta_j * e^{-2i\pi * j/n}}{2} X^j) + X^{n/2} \sum_{0 \leq j < n/2} \frac{\alpha_i - \beta_i * e^{-2i\pi * j/n}}{2} X^j$.

Donc $P(X) = \sum_{0 \leq j < n} a_j X^j$ avec $a_j = \frac{\alpha_j + \beta_j * e^{-2i\pi * j/n}}{2}$ et $a_{n/2+j} = \frac{\alpha_i - \beta_i * e^{-2i\pi * j/n}}{2}$ pour tout $0 \leq j < n/2$.

On en déduit que $[a_0, \cdots, a_{n-1}]$ peut être calculé en temps $O(n)$.

Question 35. En déduire un algorithme **récursif** qui calcule les coefficients de P en fonction de $P\{Y\} = Z$. Donner la complexité $c(p)$ de cet algorithme en fonction de p.

Proof. Si $n = 1$, alors $Y = (y_0)$ et P est constant. Alors $P\{Y\} = (P(y_0))$ peut être calculer en temps constant.

Sinon, on calcule Z_0 etZ_1 en temps $O(n)$ en fonction de Z. Puis, on calcule les coefficients de P_0 en temps $c(p-1)$ en fonction de $P_0\{Y_0\} = Z_0$. Puis les coefficients de P_1 en temps $c(p-1)$ en fonction de $P_1\{Y_0\} = Z_1$. Enfin, en utilisant la question précédente, on calcule les coefficients de P en fonction de ceux de P_0 et P_1, en temps $O(n)$.

$c(p) = 2c(p-1) + O(2^p)$ et $c(0) = O(1)$ donc $c(p) = O(p2^p)$.

Ainsi, il est possible de passer efficacement de la représentation $P\{Y\} = Z$ d'un polynôme $P(X) = \sum_{0 \leq i < n} a_i X^i \in \mathbb{C}[X]$ à celle de ses coefficients.

Soit $P(X) = \sum_{0 \leq i < n} a_i X^i \in \mathbb{C}[X]$. On pose $Q_0(X) = \sum_{0 \leq i < n/2} a_{2i} X^i$ et $Q_1(X) = \sum_{0 \leq i < n/2} a_{2i+1} X^i$.

Question 36. Prouver que, pour tout $x \in \mathbb{C}$, $P(x) = Q_0(x^2) + x \cdot Q_1(x^2)$.

Proof. Trivial.

Question 37. Soit $Y \in \mathbb{C}^n$ la suite des racines n-ièmes de l'unité. Proposer un algorithme qui calcule $P\{Y\}$ en fonction des coefficients de P de degré $< n = 2^p$, en temps $O(n \log n)$.

Proof. Si $n = 1$, le résultat est immédiat.

Soit $Y_{n/2}$ la suite des racines $(n/2)$-ièmes de l'unité. Supposons par récurrence sur $n = 2^p$ que l'on dispose d'un algorithme qui calcule $P\{Y_{n/2}\}$ pour tout polynôme de degré $< n/2$ en fonction de ses coefficients et en temps $c(n/2)$.

Soit P de degré $< n$. Alors, on calcule les coefficient s de Q_0 et Q_1 (de la question précédente) en temps $O(n)$. Puis, en utilisant l'agorithme récursif, on calcule $Q_0\{Y_{n/2}\}$ et $Q_1\{Y_{n/2}\}$ en temps $2 \cdot c(n/2)$.

Enfin, pour tout $y_k \in Y$, $y_k^2 \in Y_{n/2}$. Donc, on peut déduire $P(y_k)$ en fonction de $Q_0(y_k^2) \in Q_0\{Y_{n/2}\}$ et de $Q_1(y_k^2) \in Q_1\{Y_{n/2}\}$, en temps constant. Donc on peut calculer $P\{Y\}$ en temps $O(n)$.

En tout, l'algorithme a une complexité $c(n) = 2 \cdot c(n/2) + O(n) = O(n \log n)$.

Question 38. Déduire des questions précédentes un algorithme pour multiplier deux polynômes de degré $< n$. Donner sa complexité et comparer avec la méthode de Karatsuba.

Proof. Soit $X = (x_0, \cdots, x_{2n-1})$ la suite des racines $2n$-ièmes de l'unité, c'est-à-dire $x_k = e^{2ik\pi/(2n)}$, $k < 2n$. On calcule $P\{X\}$ en fonction des coefficients de P et $Q\{X\}$ en fonction des coefficients de Q en temps $O(n \log n)$ puis $P.Q\{X\}$ en temps $O(n)$ et enfin les coefficients de $P.Q$ en fonction de $P.Q\{X\}$ en temps $O(n \log n)$.

Pour tout $\epsilon > 0$, $n \log n = o(n^{1+\epsilon})$. Donc, la méthode de transformée de Fourrier rapide est plus efficace que celle de Kuratsuba.

6 Multiplication de Matrices, algorithme de Strassen

6.1 Opérations sur les matrices

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.

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].

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

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

Question 39. Soit $C = \begin{bmatrix} 1 & 3 & 1 \\ 0 & 0 & 1 \\ 2 & 1 & 0 \end{bmatrix}$ et $D = \begin{bmatrix} 6 & 1 & 0 \\ 0 & 2 & 3 \\ -2 & 0 & 0 \end{bmatrix}$. Calculer $D * C$ et $C * D$. Conclusion ?

Proof. La multiplication de matrice n'est pas commutative.

Question 40. Donner un algorithme qui calcule le produit d'une matrice $n \times m$ avec une matrice $m \times p$ en $O(npm)$ opérations. Prouver la correction et la complexité de l'algorithme.

En CAML, une matrice peut être représentée par un vecteur de vecteurs (lignes). Donner le code CAML correspondant.

Question 41. Soient A et B deux matrices $n \times m$ et C une matrice $m \times n$. Prouver que $C * (A + B) = C * A + C * B = C * B + C * A$. Conclusions ?

Sans détailler la preuve, comparer $(A + B) * C$ et $A * C + B * C$.

[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.

Proof. Soient $D = C * (A + B) = [d_{i,j}]_{i\leq m,j\leq m}$, $A' = C * A = [a'_{i,j}]_{i\leq m,j\leq m}$, $B' = C * B = [b'_{i,j}]_{i\leq m,j\leq m}$, $E = C * A + C * B = [e_{i,j}]_{i\leq m,j\leq m}$ et $E' = C * B + C * A = [e'_{i,j}]_{i\leq m,j\leq m}$. Notons que ces 5 matrices ont mêmes dimensions : ce sont des matrices $m \times m$.

Pout tout $i \leq m$ et $j \leq m$, $e_{i,j} = a'_{i,j}+b'_{i,j} = b'_{i,j}+a'_{i,j} = e'_{i,j}$. Donc $C*A+C*B = C*B+C*A$.

Pout tout $i \leq m$ et $j \leq m$, $d_{i,j} = \sum_{1\leq k\leq n} c_{i,k}(a_{k,j}+b_{k,j}) = \sum_{1\leq k\leq n} c_{i,k}a_{k,j}+\sum_{1\leq k\leq n} c_{i,k}b_{k,j} = a'_{i,j} + b'_{i,j} = e_{i,j}$. Donc, $C * (A + B) = C * A + C * B$.

L'addition de matrices est commutative et la multiplication est associative à droite.

De la même façon, on prouve que la multiplication est associative à gauche et $(A+B)*C = A * C + B * C$.

6.2 Matrices par blocs

Pour simplifier les notations, 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.

Question 42. 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 $X * Y = \begin{bmatrix} A*U+B*W & A*V+B*R \\ C*U+D*W & C*V+D*R \end{bmatrix}$.

6.3 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 $P_7 = (A - C) * (E + F)$.

Question 43. 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) pour calculer $M * N$.

Proof. L'addition (respectivement la soustraction) de deux matrices $n \times n$ requiert $O(n^2)$ opérations élémentaires.

Donc le produit $M * N$ requiert $O(2^{2(p-1)}) = O((2^p)^2)$ opérations élémentaires.

Question 44. 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 $c(p-1)$.

Proof. $c(p) = 7 * c(p-1) + O((2^p)^2)$.

Question 45. Calculer $c(p)$ et comparer à l'algorithme naïf proposé plus haut.

Proof. $c(p) = O((2^p)^{\ln 7})$. L'algorithme naïf réalise $O((2^p)^3)$ opérations élémentaires. $\ln 7 \approx 2,81 < 3$.

7 Algorithmes de Tri

Dans ce sujet, la complexité d'une fonction désigne l'ordre de grandeur du nombre d'opérations réalisées par la fonction.

7.1 Le 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 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.

Question 46. Écrire la fonction

```
indiceMaxi : int vect -> int
```

telle que `indiceMaxi t` recherche dans le tableau t le plus grand élément et retourne son indice. S'il y a plusieurs maxima égaux, elle retourne l'indice du premier de ceux-ci.

Question 47. Écrire la fonction ITÉRATIVE

```
triSelecIter : int vect -> unit
```

qui trie par sélection le tableau passé en entrée.

Question 48. Écrire la fonction RÉCURSIVE

```
triSelecRec : int vect -> unit
```

qui trie par sélection le tableau passé en entrée.

Question 49. Donner les complexités des fonctions triSelecIter et triSelecRec en fonction de la longueur du tableau passé en entrée.

7.2 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 triés. Une fois cette place déterminée, on procède par échanges successifs pour que le tableau des éléments d'indice 0 à i soient triés.

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 2 à n.

La recherche de la place d'insertion peut s'effectuer

– soit *séquentiellement* (vu en cours) : les éléments d'indice 0 à $i-1$ sont considérés séquentiellement
– soit *par dichotomie* : pour insérer l'élément d'indice i à sa place, on teste d'abord si il est inférieur ou égal à l'élément d'indice $\lfloor i/2 \rfloor$, et on continue ainsi récursivement.

Question 50. Expliquer brièvement les deux principes de recherche (séquentielle et dichotomique) et donner leurs complexités (en pire cas).

Question 51. Ecrire la fonction

```
posInser1 : int vect -> int -> int
```

où `posInser1 t x` recherche séquentiellement dans un tableau t trié la place d'insertion de x.

Question 52. Ecrire la fonction

```
posInser2 : int vect -> int -> int
```

où `posInser1 t x` recherche par dichotomie dans un tableau t trié la place d'insertion de x.

Question 53. Ecrire la fonction

```
decale : int vect -> int (i) -> int (j) -> unit
```

qui 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 $j-1$ en entrée et met l'élément d'indice j en position i.

Question 54. Écrire la fonction ITÉRATIVE

```
triInsertion : int vect -> unit
```

qui trie par insertion le tableau passé en entrée.

7.3 Tri fusion (merge sort)

7.4 Tri à Bulles

7.5 Tri rapide (quick sort)

Le principe du quick sort est de partitionner le tableau à trier (s'il a au moins deux éléments) en deux sous-tableaux, le premier comprenant tous les éléments inférieurs ou égaux à un élément (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 on réapplique **récursivement** le quick sort sur les premier et troisième sous-tableau (l'élément pivot, lui, est à sa place définitive).

Soient les fonctions

```
partition : int vect -> int -> int -> int
```

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 de sous-tableau, positionne le pivot à sa place définitive et retourne l'indice de cette place

et

```
quicksort : int vect -> int -> int -> unit
```

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

On considère de plus le tableau suivant :

indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13
t	5	1	12	3	24	8	10	2	14	7	2	9	4	17

Question 55. Donner un contenu possible de t après l'appel à `partition t 0 13` et sa valeur de retour.

Question 56. Avec quels paramètres doit-on appeler `quicksort` pour continuer le tri après cette partition ?

Question 57. Écrire la fonction `quicksort` .

Question 58. Écrire la fonction `partition` .

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.

Question 59. Donner et expliquer la complexité de ce tri.

7.6 Borne Inférieure, battre la borne inf?

Prouver que on ne peut pas faire mieux que $\Omega(n \log n)$ comparaisons.

Montrer qu'on peut "tricher" si le plus grand entier est petit: on prend un tableau T de longueur max des entiers et on positionne chaque entier du tableau en entrée à sa place dans T. On supprime les cases vides.

8 Complexité en moyenne

8.1 Tri par insertion et complexité en moyenne

Theorem 3. $O(n^2/4)$

Proof. Nombre d'inversions d'un tableau $T = [|x_1, \cdots, x_n|]$: $inv(T) = |\{i, j\} \mid i < j \text{ et } x_j \leq x_i\}|$. Nombre de comparaisons pour trier T par insertion $comp(T)$. Par récurrence sur n, $inv(T) \leq comp(T) \leq inv(T) + n - 1$.

Le nombre d'inversion pour un tableau T et pour son "miroir" sont égaux. Donc $inv_{moy} = O(n^2/4)$. □

8.2 Recherche d'un élément dans un tableau

Theorem 4. *La complexité en moyenne de la recherche d'un élément dans un tableau de n entiers dans $\{1, \cdots, k\}$ tend vers k quand n tend vers ∞.*

Proof. La complexité est

$$C = \frac{1}{k^n}(\sum_{j=1}^{n}(j(k-1)^{j-1}k^{n-j}) + n(k-1)^{n-1})$$

(le terme $(n(k-1)^{n-1})$ correspond au cas où l'élément n'est pas dans le tableau)
Donc

$$C = n(\frac{k-1}{k})^n + \frac{1}{k}\sum_{j=1}^{n} j(\frac{k-1}{k})^{j-1} = n(\frac{k-1}{k})^n + \frac{1}{k}f'(\frac{k-1}{k})$$

avec $f'(x) = \sum_{j=0}^{n-1}(j+1)x^j$, donc $f(x) = \sum_{j=0}^{n-1} x^{j+1} = \frac{1-x^n}{1-x}$ (pour $x \neq 1$)
Donc, $f'(x) = \frac{-nx^{n-1}}{1-x} + \frac{1-x^n}{(1-x)^2}$ et le résultat suit avec un peu de calcul. □

8.3 Juste Prix

Note. cet exercice n'a finalement pas (encore) été donné.

Vous participez à un jeu où vous faîtes face à $k \geq 2$ objets $A_1, \cdots, A_k$. On note le prix de l'objet A_i par $p(A_i)$, pour tout $i \leq k$. . Vous disposez d'une opération élémentaire : lorsque vous désignez deux objets, A_i et A_j, $i, j \leq k$, l'animateur vous indique si $p(A_i) = p(A_j)$ ou $p(A_i) < p(A_j)$ ou $p(A_i) > p(A_j)$.

Le but est d'ordonner les objets par prix croissants en utilisant le moins d'opérations élémentaires possibles.

Rappel sur la complexité en moyenne Soit $\mathcal{P}$ un problème quelconque dont l'ensemble des instances (ensemble des entrées possibles) est $\mathcal{I}$ et soit $\mathcal{A}$ un algorithme pour le résoudre. Pour tout $i \in \mathcal{I}$, on note $c_{\mathcal{A}}(i)$ le nombre d'opérations élémentaires qu'exécute $\mathcal{A}$ lorsqu'il est appliqué à i.

On rappelle que la complexité en pire cas de l'algorithme $\mathcal{A}$ est $c_{pire}(\mathcal{A}) = \max_{i \in \mathcal{I}} c_{\mathcal{A}}(i)$. Pour tout $k \in \mathbb{N}$, soit $I_k = \{i \in \mathcal{I} \ : \ c_{\mathcal{A}}(i) = k\}$. La complexité en moyenne de l'algorithme $\mathcal{A}$ est $c_{moyenne}(\mathcal{A}) = \frac{1}{|\mathcal{I}|} \sum_{k \in \mathbb{N}} |I_k| \cdot k$.

Question 60. Montrer que pour tout problème $\mathcal{P}$ et pour tout algorithme $\mathcal{A}$ pour le résoudre :

$$c_{pire}(\mathcal{A}) \geq c_{moyenne}(\mathcal{A}).$$

Proof. $c_{moyenne}(\mathcal{A}) = \frac{1}{|\mathcal{I}|} \sum_{k \in \mathbb{N}} |I_k| \cdot k \leq \frac{1}{|\mathcal{I}|} \sum_{k \in \mathbb{N}} |I_k| \cdot c_{pire}(\mathcal{A}) = c_{pire}(\mathcal{A})$.

Cette question est un peu pour voir si ils ont compris les concepts, ou sont capables de comprendre des définitions données.

Prix deux-à-deux distincts Dans cette partie, on suppose que les objets ont des prix distincts deux-à-deux ($\forall i, j \leq k$, $p(A_i) = p(A_j) \Leftrightarrow i = j$).

De plus, supposons que toutes les instances sont *équiprobables*. C'est-à-dire, pour toutes permutations $(i_1, i_2, \cdots, i_k)$ et $(i'_1, i'_2, \cdots, i'_k)$ de $\{1, \cdots, k\}$, il y a autant d'instances telles que $p(A_{i_1}) < p(A_{i_2}) < \cdots < p(A_{i_k})$ que d'instances telles que $p(A_{i'_1}) < p(A_{i'_2}) < \cdots < p(A_{i'_k})$.

Algorithm 2

Require: trois objets A_1, A_2 et A_3.
1: Désigner A_1 et A_2
2: Soit a l'objet de plus grand prix parmi A_1 et A_2, et soit b l'autre objet
3: Désigner a et A_3
4: **if** il est imposible de conclure **then**
5: Désigner b et A_3
6: **end if**
7: Donner la séquence des objets par ordre de prix croissants.

Question 61. Prouver que l'algorithme 2 résout le problème pour $k = 3$. Donner sa complexité en pire cas et un exemple d'exécution qui atteint ce pire cas.

Proof. Après la seconde comparaison (ligne 3), on a soit $p(b) < p(a) < p(A_3)$ auquel cas c'est fini, ou $p(b) < p(a)$ et $p(A_3) < p(a)$. Dans le second cas, la troisième comparaison (ligne 4) permet de conclure si $p(b) < p(A_3) < p(a)$ ou si $p(A_3) < p(b) < p(a)$. Il y au pire trois comparaisons et c'est le cas si $p(A_3) < p(a)$.

Question 62. Prouver que la complexité en moyenne de l'algorithme 2 est $\frac{2}{3} \cdot 3 + \frac{1}{3} \cdot 2 = \frac{8}{3}$.

Proof. Dans un cas, il faut 2 comparaisons et dans deux cas, il en faut 3 (cf. preuve plus haut). Comme les cas sont équiprobables, le premier cas apparaît dans un tiers des cas, et les deux autres cas englobent les deux tiers des cas.

Algorithm 3

Require: quatre objets A_1, A_2, A_3 et A_4.
1: Désigner A_1 et A_2. Soit a l'objet de plus petit prix parmi A_1 et A_2, et soit b l'autre objet.
2: Désigner A_3 et A_4. Soit c l'objet de plus petit prix parmi A_3 et A_4, et soit d l'autre objet.
3: Désigner a et c
4: **if** $p(a) < p(c)$ **then**
5: Soient $x = a$, $y = b$, $v = c$ et $w = d$
6: **else**
7: Soient $x = c$, $y = d$, $v = a$ et $w = b$
8: **end if**
9: **if** il est imposible de conclure **then**
10: Désigner y et v
11: **end if**
12: **if** il est imposible de conclure **then**
13: Désigner y et w
14: **end if**
15: Donner la séquence des objets par ordre de prix croissants.

// JE PREFERERAIS DONNER L'ALGO SUIVANT, MAIS LE COMPRENDRONT ILS ?

Algorithm 4

Require: quatre objets a, b, c et d.
1: Désigner a et b. Sans perte de généralité, $p(a) < p(b)$
2: Désigner c et d. Sans perte de généralité, $p(c) < p(d)$
3: Désigner a et c. Sans perte de généralité, $p(a) < p(c)$
4: **if** il est imposible de conclure **then**
5: Désigner b et c
6: **end if**
7: **if** il est imposible de conclure **then**
8: Désigner b et d
9: **end if**
10: Donner la séquence des objets par ordre de prix croissants.

Question 63. Prouver que l'algorithme 4 résout le problème pour $k = 4$. Donner sa complexité en pire cas et un exemple d'exécution qui atteint ce pire cas. Prouver que sa complexité en moyenne est $\frac{2}{3} \cdot 5 + \frac{1}{3} \cdot 4 = \frac{14}{3}$.

Proof. Après la quatrième comparaison (ligne 6), on a soit $p(a) < p(b) < p(c) < p(d)$ auquel cas c'est fini, ou $p(a) < p(b)$ et $p(c) < p(d)$ et $p(c) < p(a)$. Dans le second cas, la cinquième comparaison (ligne 7) permet de conclure si $p(a) < p(c) < p(b) < p(d)$ ou si $p(a) < p(c) < p(d) < p(b)$. Il y au pire cinq comparaisons et c'est le cas si $p(a) < p(c) < p(b) < p(d)$ ou si $p(a) < p(c) < p(d) < p(b)$ alors qu'il n'y en a que quatre dans le cas $p(a) < p(b) < p(c) < p(d)$. Comme ces trois cas sont équiprobables, on obtient la complexité en moyenne annoncée.

Question 64. Prouver que tout algorithme pour résoudre le problème d'ordonnancement requiert au moins $k-1$ opérations élémentaires.

Proof. Avec $< k-1$ comparaisons, il existe un objet qui n'a été comparé avec aucun autre. Il est impossible de le classer.

Question 65. Donner un algorithme pour résoudre le problème avec une complexité $O(k^2)$.

Proof. Pour tout $i \leq k$ et pour tout $i < j \leq k$, Désigner A_i et A_j. Une fois cette boucle effectuée, tous les objets ont été comparés deux-à-deux et il est possible de les classer. Cela requiert $\sum_{k=1}^{n-1} k = n(n-1)/2$ comparaisons.

Prix bornés Dans cette partie, on suppose de nouveau que $k = 3$.

Pour tout $i \leq k$, $p(A_i)$ est un entier entre 1 et q. Plusieurs objets peuvent avoir un prix identique. En d'autres termes, l'ensemble des instances possibles est maintenant l'ensemble de tous les triplets $(x, y, z) \in \{1, \cdots, q\}^3$ où x représente $p(A_1)$, y représente $p(A_2)$ et z représente $p(A_3)$.

Question 66. Pour $j \in \{2, 3\}$, soit I_j le nombre d'instances telles que exactement j objets ont un prix identique. Montrer que $|I_3| = q$ et $|I_2| = 3q(q-1)$.

Question 67. Donner un algorithme de comparaison de complexité en moyenne $\frac{8}{3} - \frac{1}{q} + \frac{1}{3q^2}$.

8.4 Tri rapide et complexité en moyenne

Question 68. Rappeler brievement (pas plus de 5 lignes) le principe de l'algorithme de tri-fusion et donner sa complexité (sans preuve).

Dans ce problème, nous nous intéressons à un autre algorithme de tri, appelé *Tri rapide* (*quick sort* en anglais). On rappelle que, étant donné un tableau d'entiers, le but est de renvoyer un tableau avec les mêmes éléments, ordonnés dans l'ordre croissant.

Le principe du quick sort est de partitionner le tableau trier (s'il a au moins deux lments) en trois sous-tableaux, le premier comprenant tous les lments infrieurs ou gaux un lment (l'lment pivot), le deuxième ne contenant qu'un seul lment (l'lment pivot) et le troisième contenant tous les lments suprieurs ou gaux l'lment pivot. Puis on rapplique **rcursivement** le quick sort sur les premier et troisième sous-tableau (l'lment pivot, lui, est sa place dfinitive).

On considère tout d'abord la fonction *pivot*, décrite ci-dessous, qui prend en entrée un tableau T de longueur n et deux entiers $0 \leq a \leq b < n$.

Algorithm 17

```
let rec pivot T a b=
    if a == b then a
    else
        let aux = T.(a + 1) in
        if T.(a) > aux then T.(a + 1) ← T.(a); T.(a) ← aux; pivot T (a + 1) b
        else T.(a + 1) ← T.(b); T.(b) ← aux; pivot T a (b − 1)
```

Question 69. Soit $T = [4, 3, 7, 9, 2]$. Que retourne *pivot* T 0 4 ?

Quelle est la valeur de T après l'exécution de cette fonction ?

Question 70. Quel est le nombre de comparaisons d'entiers réalisées par *pivot* T a b ?

Algorithm 18

```
let rec quickSort T =
    let n = vect_length T in
        let rec quickAux T a b=
            if a == b then T
            else let i = pivot T a b in quickAux (quickAux T a i) i+1 b;;
        in quickAux T 0 (n − 1);;
```

Question 71. Soit $T = [1, 2, 3, \cdots, n]$.

Quel est le nombre de comparaisons d'entiers réalisées par *quickSort* T ? Justifiez.

D'après la question précédente, il existe des "mauvais" tableaux pour lesquels l'algorithme quickSort se comporte "mal": sa complexité est quadratique en la longueur du tableau.

Dans le pire cas, l'algorithme de tri-fusion est donc meilleur que quickSort. Cependant, nous allons voir que quickSort est efficace "en moyenne".

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.

Question 72. 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* T.

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.

Question 73. 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.

Question 74. Prouver que $C_n = \frac{1}{n} \sum_{1\leq k\leq n} C_n(k)$.

Question 75. Prouver que $C_n(k) = n + 1 + C_{k-1} + C_{n-k}$.

Question 76. En déduire que $C_n = n + 1 + \frac{2}{n} \sum_{1\leq k<n} C_k$

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

Question 77. 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* $\frac{C_{n+1}}{n+2} = \frac{C_n}{n+1} + \frac{4}{n+2} - \frac{2}{n+1}$

Question 78. Conclusion ?

9 Programmation dynamique

Algorithme qui calcule une solution en combinant les solutions de sous-problèmes.

9.1 Problèmes de rendu de monnaie et sac à dos

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 $(c_1 = 1, c_2, \cdots, c_n)$).

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 $\mathcal{S} = (c_1 = 1, c_2, \cdots, c_n) \in \mathbb{N}^n$
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.

On supposera toujours que $c_1 = 1 \leq c_2 \leq \cdots \leq c_n$. 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.

Un premier algorithme est l'algorithme **glouton** qui consiste à toujours rendre la plus grande pièce possible.

Algorithm 19 (Algorithme glouton pour le rendu de monnaie)

Entrée x et $\mathcal{S} = (c_1 = 1, c_2, \cdots, c_n)$
Sortie une solution réalisable $(k_1, \cdots, k_n) \in \mathbb{N}^n$.

1. Soit $1 \leq i \leq n$ l'entier maximum tel que $c_i \leq x$.
2. Soit $(k'_1, \cdots, k'_n) = Glouton(x - c_i, \mathcal{S})$
3. Renvoyer $(k'_1, \cdots, k'_{i-1}, k'_i + 1, k'_{i+1}, \cdots, k'_n)$

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

Malheureusement, l'algorithme glouton précédent ne trouve pas toujours une solution optimale.

Par exemple, pour $(x = 6, \mathcal{S} = (1, 3, 4))$. En effet, l'algorithme glouton renvoie $4 + 1 + 1$ alors que la solution optimale est $3 + 3$.

Question 79. Montrer que l'algorithme glouton est optimal pour $n \leq 2$.

Remarque. On ne sait pas s'il existe une algorithme polynomial pour décider si un système $\mathcal{S}$ est *canonique*, i.e., si l'algorithme glouton est optimal pour $\mathcal{S}$ quelque soit x.

Lemma 1. *Soit $(k_1, \cdots, k_n)$ une solution optimale pour $(x > 0, \mathcal{S})$, et soit $i \leq n$ tel que $k_i > 0$. Alors $(k_1, \cdots, k_{i-1}, k_i - 1, k_{i+1}, \cdots, k_n)$ est une solution optimale pour $(x - c_i, \mathcal{S})$*

Proof. Par l'absurde. □

Theorem 5. $OPT(x) = 1 + \min_{1 \leq i \leq n, x - c_i \geq 0} OPT(x - c_i)$

Proof. $\leq$ trivial.$\geq$ lemme □

Donc on va calculer $OPT(x - c_i)$ pour tout i tel que $x - c_i \geq 0$ (les sous-problèmes) et on va en déduire $OPT(x)$.

Algorithm 20 (Algorithme par programmation dynamique)

Entrée x et $\mathcal{S} = (c_1 = 1, c_2, \cdots, c_n)$
Sortie $\sum_{i \leq n} k_i$ avec $(k_1, \cdots, k_n) \in \mathbb{N}^n$ une solution réalisable optimale .

1. Soit SOL un tableau de longeur $x + 1$.
2. $SOL(0) = 0$ et $SOL(i) = \infty$ pour tout $0 < i \leq x$.
3. Pour $k = 1$ à x faire
   - Soit i le plus grand entier tel que $c_i \leq k$
   - Pour $j = 1$ à i faire $SOL(k) \leftarrow \min\{SOL(k); 1 + SOL(k - c_i)\}$.
4. Renvoie $SOL(x)$.

Preuve de correction: Theoreme + par récurrence à l'étape k, pour tout $j < k$, $SOL(j) = OPT(j)$.

Complexité $O(xn)$. Notons que c'est en fait "pseudo polynomial" car la taille de l'entrée est $O(n \log x)$. C'est polynomial uniquement si $x = O(poly(n))$.

Remarquons également qu'on peut modifier l'algo pour renvoyer une solution optimale (pas seulement sa valeur)

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 c_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 $\sum_{i \leq n} v_i$ est maximum.

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

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 note $OPT(W, j) = OPT(W, x_1, \cdots, x_j)$ et on prouve que $OPT(W, j) = \max\{OPT(W - p_j, j-1) + v_j, OPT(W, j-1)\}$. On peut alors calculer la solution du problème: $OPT(W, n)$.

9.2 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 indépendament.

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

Algorithm 21

```
Let SomMax1 T =
    let n = vect_length T in
    let sMax = ref T.(0) in
    for a = 0 to n − 1 do
        for b = a to n − 1 do
            let m = ref T.(a) in
            for k = a + 1 to b do m := !m + T.(k) done;
            sMax := max m !sMax
        done;
    done;
    !sumMax;;
```

Question 80. 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.

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 81. En suivant la suggestion ci-dessus, écrire en CAML 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 $\Theta(n^2)$.

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

Question 83. Écrire en CAML 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 $SomMax_3$.

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 84. 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 85. Écrire en CAML 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 86. Écrire en CAML 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 87. En déduire en CAML une fonction récursive $SomMax_4$ qui prend en entrée un tableau T et calcule $S(T)$.

Question 88. 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 ?

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 89. 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 90. En déduire en CAML une fonction $SomMax_5$ qui prend en entrée un tableau T et calcule $S(T)$ en temps linéaire en la taille de T.

10 Arbres Binaires

10.1 Définitions et propriétés simples

Graphe $G = (V, E)$, sommet (nœuds), arêtes, sommets adjacents, voisins, degré, chemin, cycle, graphe connexe, distance

Arbre= graphe acyclique connexe

Question 91. Montrer que pour tout graphe, $\sum_{v \in V} degree(v) = 2|E|$.

Question 92. Montrer qu'un graphe est un arbre ssi $|V| = |E| + 1$.

Arbre binaire T: soit arbre vide $T = \emptyset$, soit $T = (T_g, r, T_d)$ avec r le sommet racine, T_g le sous-arbre gauche et T_d le sous-arbre droit.

Notion de feuille, nœud interne, hauteur, arbre localement complet, arbre complet

Proposition 1. *Soit un arbre binaire T de n sommets, avec f feuilles et de hauteur h*

- $f \leq (n+1)/2$ *avec égalité ssi* $T \neq \emptyset$ *et* T *complet*
- $n + 1 \leq 2^{h+1}$ *avec égalité ssi* T *complet*
- $h \leq n - 1$ *avec égalité ssi tout nœud a au moins un fils vide.*

Question 93. Ecrire des algorithmes qui prend un arbre binaire T en entrée et calcule:

1. le nombre de sommets
2. le nombre de feuilles
3. la hauteur
4. si T est localement complet
5. si T est complet

10.2 Indépendant Maximum dans les arbres (prog dyn)

Un *ensemble indépendant* $S \subseteq V$ (ou *stable*) est un ensemble de sommets tel que $\forall u, v \in S$, $\{u, v\} \notin E$. Le problème de trouver un stable de taille maximum dans un graphe quelconque est NP-complet (le meilleur algorithme connu a complexité $O(1.2^{|V|} poly(|V|)$. Une recherche exhaustive donne un algorithme en $O(2^{|V|} poly(|V|))$.

Ici, on va donner un algorithme qui calcule, par programmation dynamique et en temps polynomial, un stable maximum dans le cas des arbres. Soit $T = (V, E)$ un arbre enraciné en $r \in V(T)$. Pour tout $v \in V(T)$, on note T_v le sous-arbre de T constitué de v et de ses descendants.

Soit T enraciné en r. Soit $\alpha(T)$ la taille d'un stable maximum de T. Soit $\alpha_0(T)$ la taille d'un stable maximum de T ne contenant pas r et $\alpha_1(T)$ la taille d'un stable maximum de T contenant r. Par définition:

Lemma 2. $\alpha(T) = \max\{\alpha_0(T); \alpha_1(T)\}$.

Lemma 3. *Soient $T_1, \cdots, T_d$ les sous-arbres de T enracinés en $v_1, \cdots, v_d$ les fils de r.*

1. $\alpha_0(T) = \sum_{1 \leq i \leq d} \alpha(T_i)$.
2. $\alpha_1(T) = 1 + \sum_{1 \leq i \leq d} \alpha_0(T_i)$.

Proof. Soit X un stable de T de contenant pas r et tel que $|X| = \alpha_0(T)$. Soit $X_i = X \cap V(T_i)$. Clairement, X_i est un stable de T_i. Par l'absurde, si $X_i < \alpha(T_i)$, alors soit Y_i un stable maximum de T_i. Alors $Z = X \cup Y_i \setminus X_i$ est un stable de T ne contenant pas r, et $|Z| > |X|$, une contradiction. Donc, pour tout $i \leq d$, X_i est un stable maximum de T_i et, par conséquent, $\alpha_0(T) = \sum_{1 \leq i \leq d} \alpha(T_i)$.

La preuve de la seconde affirmation est similaire. □

L'algorithme commence par calculer α_0 et α_1 pour chaque feuille f de T ($\alpha_0(f) = 0$ et $\alpha_1(f) = 1$) et les stables correspondants ($\emptyset$ et $\{f\}$). Puis, récursivement, étant donné un sous-arbre enraciné en v, on calcule, pour chaque fils v_i de v: un stable maximum de T_i contenant v_i (i.e., de taille $\alpha_1(T_i)$) et un stable maximum de T_i ne contenant pas v_i (i.e., de taille $\alpha_0(T_i)$). On les assemble alors comme dans le lemme précédent pour calculer un stable maximum de T_v contenant v (i.e., de taille $\alpha_1(T_v)$) et un stable maximum de T_v ne contenant pas v (i.e., de taille $\alpha_0(T_v)$).

10.3 Le tri par tas (heap sort)

Définitions préliminaires

Definition 1. *Un* arbre binaire *est*

- *soit un arbre vide appelé nil et noté ici* $\emptyset$
- *soit un nud (père)* (u, k, v) *où* u *et* v *sont des arbres (appelés ses fils gauche et droit) et* $k \in N$ *est sa valeur.*

Un nud de la forme $(\emptyset, k, \emptyset)$ *est appelé une feuille, on la note dans la suite* k*.*

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

Definition 2. *Soit* (g, r, d) *un arbre binaire.* g *est le sous arbre gauche,* d *est le sous arbre droit et* r *est la* racine *de l'arbre, c'est-à-dire, le nœud situé en haut sur le dessin.*

Dans l'exemple précédent, g est le sous-arbre réduit au nœud 1, la racine est le nœud 2 et $(3, 4, 5)$ est le sous arbre droit. Les feuilles sont les nœuds 1, 3 et 5.

Definition 3. *Si* n *est un nud, le nombre de nuds (*n *exclu) pour aller de* n *à la racine est appelé sa profondeur. La racine est le seul nœud de niveau* 0*. L'ensemble des nœuds de mme profondeur* p *dans un arbre est appelé le niveau de profondeur* p *de l'arbre.*
Le nombre de nuds d'un arbre est appelé sa taille.

Dans l'arbre précédente la racine est le nud de valeur 2, la feuille de valeur 1 est de profondeur 1 et les autres feuilles sont de profondeur 2. Le niveau de profondeur 1 est constitué des nuds de valeur $1, 3$ et 5. Cet arbre est de taille 5.

Definition 4. *Un arbre binaire presque complet (ABPC) est un arbre binaire dans lequel toutes les feuilles sont au mme niveau de profondeur et où toutes les feuilles sont à gauche dans la représentation.*

Par exemple l'arbre suivant est un ABPC :

Il dispose de quatre niveaux de profondeurs :

- 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}.

Question 94. On considère ici un ABPC dont les feuilles sont au niveau de profondeur n.
Déterminer le nombre de nuds au niveau de profondeur $k < n$.
En déduire une majoration de la taille de l'arbre en considérant le cas où le niveau n serait entièrement rempli.

Il est possible de représenter un ABPC par un tableau en stockant successivement les nuds du niveau 0, les nuds du niveau 1, les nuds du niveau n jusqu'à finir par les feuilles.
Dans le cas de l'ABPC précédent on obtient le tableau

```
let t = [| n1; n2; n3; n4; n5; n6; n7; n8; n9; n10 |]
```

Question 95. Dessiner l'arbre représenté par le tableau suivant :

```
let t = [| 9; 5; 8; 9; 7; 6; 7; 3; 6; 4 |]
```

Definition 5. *Un nud (u, k, v) est dit dominant si k est supérieure ou égale à la valeur de u et à la valeur de v (on considère que $\emptyset$ est de valeur $-\infty$).*
Un tas est un ABPC dont tous les nuds sont dominants.

Question 96. Indiquer les nuds dominants dans l'arbre obtenu à la question précédente.

Construction d'un tas

Question 97. Écrire les fonctions

```
indiceFilsG : int -> int
indiceFilsD : int -> int
indicePere : int -> int
```

telle que `indiceFilsG i` (resp. `indiceFilsD i`) renvoie l'indice du fils gauche (resp. droit) du nud d'indice i dans un ABPC, et `indicePere i` renvoie l'indice du père du nud d'indice i.

Question 98. Écrire les fonctions

```
estFeuille : int -> int -> bool
estPere1 : int -> int -> bool
estPere2 : int -> int -> bool
```

où `estFeuille n i` renvoie `vrai` si et seulement si i est l'indice d'une feuille d'un ABPC de taille n, et `estPere1 n i` (resp. `estPere2 n i`) renvoie `vrai` si et seulement si i est l'indice d'un nœud ayant un fils (resp. deux fils) dans un ABPC de taille n.

Question 99. Écrire la fonction

```
estDominant : int vect -> int -> bool
```

où `estDominant t i` retourne `vrai` si et seulement si i est l'indice d'un nud 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 ?

Question 100. Écrire la fonction

```
retablirTas : int vect -> int -> unit
```

où `retablirTas t i` 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 fils de i, s'ils existent, sont des tas**.

Par exemple, si `t = [| 9; 5; 8; 9; 7; 6; 7; 3; 6; 4 |]` l'appel à `retablirTas t 2` transformera l'arbre en `[|9; 9; 8; 6; 7; 6; 7; 3; 5; 4|]`.

Décrire les opérations effectuées par la fonction.

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

Question 101. Ecrire la fonction

```
construireTas : int vect -> unit
```

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 |]`.

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

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.

Question 102. Écrire la fonction

```
heapsort : int vect -> unit
```

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 ?