Carnet de Travaux Libres
Algorithmique et structures de données
Licence 2 Informatique

Julien BERNARD

Introduction

Le carnet de travaux libres est une compilation de sujets de travaux dirigés et de travaux pratiques. Il peut vous servir à réviser, à vous auto-évaluer, à mieux comprendre certains concepts, etc. Il contient aussi les anciens sujets d'examen, sous forme d'exercices séparés.

Table des matières

1 Initiation aux C et aux pointeurs **5**
Exercice 1 : Mon premier programme en C 5
Exercice 2 : Découverte du langage C 6
Exercice 3 : Manipulation de pointeurs 8
Exercice 4 : Exécution d'un programme (1) 9
Exercice 5 : Exécution d'un programme (2) 10
Exercice 6 : Exécution d'un programme (3) 11
Exercice 7 : Échanges 12
Exercice 8 : Pointeur et récursivité 14

2 Calcul de complexité **15**
Exercice 9 : Ordres de grandeur 15
Exercice 10 : Échelle de fonction 16
Exercice 11 : Comptage d'opérations 17
Exercice 12 : Taille de problème 18
Exercice 13 : Complexité et boucles 19
Exercice 14 : Calcul de complexité 20
Exercice 15 : Application du théorème Diviser pour régner 21

3 Complexité et problèmes **22**
Exercice 16 : La fonction puissance 22
Exercice 17 : Suite de Fibonacci 24
Exercice 18 : Recherche ternaire 26
Exercice 19 : Algorithme de Karatsuba 27
Exercice 20 : Problème des points les plus proches 28
Exercice 21 : Simplification de polyligne 30
Exercice 22 : Multiplication de matrices 31
Exercice 23 : Le grand saut 33
Exercice 24 : Recherche d'un élément majoritaire 34
Exercice 25 : Problème du sous-tableau maximum 38
Exercice 26 : Recherche de pic dans un tableau 40
Exercice 27 : Leaders dans un tableau 41
Exercice 28 : Problème du drapeau hollandais 42

4 Tableaux et chaînes de caractères **43**
Exercice 29 : Tableaux 43
Exercice 30 : Tableaux avancés 44
Exercice 31 : Fonctions sur les chaînes de caractères 45
Exercice 32 : Recherche de sous-chaîne 46
Exercice 33 : Structure `argz` 47
Exercice 34 : Crible d'Ératosthène 49
Exercice 35 : Méthode des k plus proches voisins 50

5 Listes chaînées **51**
Exercice 36 : Liste ordonnée 51
Exercice 37 : Manipulation de liste chaînée 52
Exercice 38 : Listes doublement chaînées 53
Exercice 39 : Liste chaînée cyclique 54

Exercice 40 : Liste chaînée déroulée 56
Exercice 41 : Liste chaînée avec cache des nœuds 58
Exercice 42 : Liste chaînée et tri fusion 59
Exercice 43 : Tours de Hanoï 60

6 Piles et files **62**
Exercice 44 : Implémentation d'une file avec une liste chaînée 62
Exercice 45 : Implémentation d'une file avec deux listes chaînées . . . 63
Exercice 46 : Implémentation d'une file avec une liste doublement chaînée 65
Exercice 47 : Implémentation d'une pile avec une liste chaînée 66
Exercice 48 : Implémentation d'une file à double entrée 67

7 Structures linéaires et tri **68**
Exercice 49 : Tri par base 68
Exercice 50 : Tri d'un tableau binaire 69
Exercice 51 : Algorithmes de sélection 70
Exercice 52 : Différences entre les éléments distincts d'un tableau . . . 72
Exercice 53 : Union de segments 73
Exercice 54 : Vecteur creux 74
Exercice 55 : Répertoire téléphonique 76
Exercice 56 : Résultat d'examen 78
Exercice 57 : Table de hachage 80

8 Matrices **82**
Exercice 58 : Jeu de la vie 82
Exercice 59 : Problème des n reines 83
Exercice 60 : Carré magique 84
Exercice 61 : Traitement d'image 85
Exercice 62 : Recherche d'un élément dans une matrice ordonnée . . . 87
Exercice 63 : Vérification d'un sudoku 89
Exercice 64 : Compter les zones connexes dans une grille (4,5 points) . 90

9 Arbres **91**
Exercice 65 : Arbre d'expression arithmétique 91
Exercice 66 : Expression arithmétique avec une variable 92
Exercice 67 : Arbre de formule logique booléenne 94
Exercice 68 : Arbre préfixe 95
Exercice 69 : Transformation d'un arbre général en arbre binaire . . . 96
Exercice 70 : Codage de Huffman 98
Exercice 71 : Corde 100
Exercice 72 : Arbre de décision des algorithmes de tri 102

10 Arbres binaires de recherche **104**
Exercice 73 : Propriétés des arbres binaires de recherche 104
Exercice 74 : Arbre binaire de recherche 105
Exercice 75 : Arbre binaire de recherche de chaînes de caractères . . . 106
Exercice 76 : i-ième élément d'un arbre binaire de recherche 107
Exercice 77 : Insertion à la racine d'un arbre binaire de recherche . . . 109
Exercice 78 : Vérification d'un arbre binaire de recherche 111
Exercice 79 : Reconstruction d'un arbre binaire de recherche 112

Exercice 80 : Ensemble plat 113
Exercice 81 : Arbretas 114
Exercice 82 : Parcours itératif d'un arbre binaire de recherche 116

11 Tas **118**
Exercice 83 : Propriétés des tas 118
Exercice 84 : Implémentation d'un tas pour l'algorithme de Dijkstra . 119
Exercice 85 : File de priorité et algorithme de tri (7,5 points) 120

12 Graphes **121**
Exercice 86 : La plante, la chèvre et le loup 121
Exercice 87 : Rayon et diamètre d'un graphe 122

13 Algorithmes gloutons **123**
Exercice 88 : Problème de rendu de monnaie 123
Exercice 89 : Problème du sac à dos 123

1 Initiation aux C et aux pointeurs

Exercice 1 : Mon premier programme en C

Le langage C est le langage que nous allons utiliser et apprendre pendant le cours d'Algorithmique. C'est un langage compilé, c'est-à-dire qu'il est nécessaire d'appeler un compilateur pour produire un exécutable qui sera fonctionnel uniquement sur la machine sur laquelle il a été compilé.

Question 1.1 Recopier le code source suivant dans un fichier appelé `hello.c`.

```
#include <stdio.h>

int main() {
  printf("Hello World\n");
  return 0;
}
```

Question 1.2 Compiler le programme avec la ligne de commande suivante :

```
gcc -Wall -std=c99 -O2 -g -o hello hello.c
```

Question 1.3 Exécuter le programme :

```
./hello
```

Exercice 2 : Découverte du langage C

Le but de cet exercice est de découvrir le langage C. Ce langage fait partie de la même famille que Java au niveau syntaxique, beaucoup de constructions sont similaires, de même que certains types.

Question 2.1 Recopier le code source suivant dans un fichier appelé `arg.c`.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  if (argc != 2) {
    printf("Un argument est nécessaire !\n");
    return 1;
  }

  int arg = atoi(argv[1]);
  printf("L'argument est : %d\n", arg);

  return 0;
}
```

Dans ce programme, on utilise la ligne de commande pour fournir un nombre au programme. S'il n'y a pas assez d'argument, un message d'erreur est renvoyé et le programme s'arrête. Sinon, l'argument est transformé en nombre grâce à la fonction `atoi(3)`. Puis, on affiche le nombre grâce à `printf(3)`. Cette fonction prend comme argument une chaîne de caractères entre guillemets, puis éventuellement des noms d'identifiants de variables. Lors de l'exécution, la chaîne de caractères est affichée ainsi que la valeur de chaque variable à l'endroit indiqué. Les variables doivent apparaître dans l'ordre de leur apparitions dans la chaîne de caractères. Un entier est marqué grâce à `%d`, un flottant grâce à `%f`, un caractère grâce à `%c`.

```
$ ./arg 32
L'argument est : 32
```

On utilisera ce code comme base pour les questions suivantes.

Question 2.2 Faire un programme qui affiche la suite de Collatz. L'argument sera l'élément initial de la suite. Pour rappel, la suite de Collatz est définie par un élément initial u_0 strictement positif et :

$$u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est divisible par 2} \\ 3u_n + 1 & \text{sinon} \end{cases}$$

On utilisera une boucle `while` et on s'arrêtera quand u_n vaudra 1.

```
$ ./collatz 46
46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1
```

Question 2.3 Faire un programme qui affiche les nombres de 1 à n où n est passé en argument. Dans cette suite, on remplace les multiple de 3 par «Fizz», les multiple de 5 par «Buzz» (et donc les multiples de 3 et 5 par «FizzBuzz»). On utilisera une boucle `for`.

```
$ ./fizzbuzz 16
1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 FizzBuzz 16
```

Question 2.4 Faire un programme qui permet d'afficher un triangle d'une longueur qu'on précisera en argument. On utilisera une double boucle `for`. Par exemple, pour un argument de 7, le programme affichera :

```
$ ./triangle 7
#
# #
# # #
# # # #
# # # # #
# # # # # #
# # # # # # #
```

Exercice 3 : Manipulation de pointeurs

On suppose que les pointeurs p, q et r se trouvent respectivement aux cases mémoire d'adresses 0x0101, 0x0102 et 0x103 et ont été déclarés de la manière suivante :

```
int *p;
int *q;
int *r;
```

On suppose également que le contenu de la mémoire est donné par la figure 1.

0x0101	0x0104	p
0x0102	0x0104	q
0x0103	0x0105	r
0x0104	23	
0x0105	23	

FIGURE 1 – État de la mémoire

Question 3.1 Les assertions suivantes sont-elles vraies ?

1. `p == q`
2. `p == r`
3. `*p == *q`
4. `*p == *r`

On exécute les instructions suivantes :

```
*p = 24;
*r = 25;
q = r;
*q = 26;
```

Question 3.2 Quel est l'état de la mémoire après cette exécution ?

Exercice 4 : Exécution d'un programme (1)

Question 4.1 Qu'affiche le programme ? Justifier.

```
#include <stdio.h>

int main() {
  int a = 1;
  int b[2] = { 3, 4 };
  int *p;
  int *q;

  p = &a;
  q = b;
  printf("%d %d %d\n", a, *p, *q);
  *p = *q + 1;
  printf("%d %d %d\n", a, *p, *q);
  p = q;
  printf("%d %d %d\n", a, *p, *q);
  *p = *p - *q;
  printf("%d %d %d\n", a, *p, *q);
  *q = *(q + 1);
  printf("%d %d %d\n", a, *p, *q);
  a = *q * *p;
  printf("%d %d %d\n", a, *p, *q);
  p = &a;
  printf("%d %d %d\n", a, *p, *q);
  *q = *p;
  printf("%d %d %d\n", a, *p, *q);

  return 0;
}
```

Exercice 5 : Exécution d'un programme (2)

Question 5.1 Qu'affiche le programme ? Justifier.

```
#include <stdio.h>
#include <stdlib.h>

void f(int *p1, int *p2, int *p3) {
  p1 = p2;
  *p2 = *p3;
  *p3 = 8;
}

void g(int **pp1, int **pp2, int **pp3) {
  *pp1 = *pp2;
  **pp2 = **pp3;
  **pp3 = 12;
}

int main() {
  int *a = malloc(sizeof(int));
  int *b = malloc(sizeof(int));
  int *c = malloc(sizeof(int));
  *a = 1;
  *b = 2;
  *c = 3;
  f(a,b,c);
  printf("%d %d %d\n", *a, *b, *c);
  g(&a,&b,&c);
  printf("%d %d %d\n", *a, *b, *c);
  free(a);
  free(b);
  free(c);
  return 0;
}
```

Exercice 6 : Exécution d'un programme (3)

Question 6.1 Qu'affiche le programme ? Justifier.

```
#include <stdio.h>
#include <stdlib.h>

void f(int *tab, int *p, int q) {
  tab[0] = *p;
  p = tab;
  *(p + 1) = 8;
  q = *p + 1;
}

void g(int **pp, int **qq) {
  *qq = *pp;
  *(*pp + 1) = 2;
  **qq = 13;
}

int main(){
  int a = 1;
  int b[3] = { 3, 4, 5 };
  int *c = malloc(sizeof(int));
  int *d;
  *c = 6;
  f(b,c,a);
  printf("%d %d %d %d %d\n", a, b[0], b[1], b[2], *c);
  d = b;
  g(&d, &c);
  printf("%d %d %d %d %d\n", b[0], b[1], b[2], *c, *d);
  return 0;
}
```

Exercice 7 : Échanges

On considère le programme suivant :

```
#include<stdio.h>
#include<stdlib.h>

void echange1(int x, int y) {
  int tmp = x;
  x = y;
  y = tmp;
}

void echange2(int *x, int *y) {
  int *tmp = x;
  x = y;
  y = tmp;
}

void echange3(int *x, int *y){
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

void echange4(int *x, int *y) {
  int *tmp = *x;
  *x = *y;
  *y = *tmp;
}

int main() {
  int a, b;
  a=2; b=3;
  echange1(a,b);
  printf("%d %d\n", a, b);
  a=2; b=3;
  echange2(&a,&b);
  printf("%d %d\n", a, b);
  a=2; b=3;
  echange2(a,b);
  printf("%d %d\n", a, b);
  a=2; b=3;
  echange3(&a,&b);
  printf("%d %d\n", a, b);
  a=2; b=3;
  echange3(a,b);
  printf("%d %d\n", a, b);
  a=2; b=3;
  echange4(&a,&b);
```

```
  printf("%d %d\n", a, b);
  return 0;
}
```

Question 7.1 Que se passe-t-il lors de la compilation du programme suivant ? Expliquer.

Question 7.2 Que se passe-t-il lors de l'exécution une fois les erreurs de compilation supprimées ? Expliquer.

Exercice 8 : Pointeur et récursivité

On considère le programme suivant :

```
#include <stdio.h>

void modif (int *px) {
  printf("%d ", *px);

  if ((*px) > 0) {
    (*px)--;
    modif(px);
  }

  printf("%d ", *px);
}

int main() {
  int x = 3;
  modif(&x);
  return 0;
}
```

Question 8.1 Qu'est-ce qui s'affiche à l'écran lors de l'exécution du programme ? Expliquer.

2 Calcul de complexité

Exercice 9 : Ordres de grandeur

Question 9.1 Qu'est-ce qui est le plus grand : 10^{100} ou 100^{10} ? Quelle est la moitié de 4^{20} ?

Question 9.2 Combien de chiffres comporte 2^n en écriture décimale ?

Exercice 10 : Échelle de fonction

Question 10.1 Classer ces fonctions par ordre décroissant asymptotique :
$n \log n$, $2^n \log^3 n$, $n^2 \log n$, $3^n \log n$, $n \log \log n$, $n \log^3 n$, n, $2^n n^2$

Question 10.2 Les assertions suivantes sont elles vraies ou fausses, pourquoi ?

1. $3^n = O(2^n)$
2. $\log 3^n = O(\log 2^n)$

Question 10.3 Calculer un ordre de grandeur de $\log(n!)$. On pourra utiliser le résultat suivant : si $f \sim g$ alors $\log f = O(\log g)$.

Exercice 11 : Comptage d'opérations

Dans cet exercice, on note $\mathcal{N}$ le nombre d'opérations `+` effectuées par chaque fonction.

Question 11.1 On considère la fonction suivante :

```
int f(int n) {
  int res = 0;

  for (int i = 0; i < n; ++i) {
    res += i;
  }

  return res;
}
```

Calculer $\mathcal{N}_f$ en fonction de n ?

Question 11.2 On considère la fonction suivante :

```
int g(int n) {
  int res = 0;

  for (int i = 0; i < n; ++i) {
    res += f(i);
  }

  return res;
}
```

Calculer $\mathcal{N}_g$ en fonction de n ?

Question 11.3 Dans la fonction précédente, on remplace :

```
res += f(i);
```

par :

```
res += f(n)
```

Que vaut alors $\mathcal{N}_g$ en fonction de n ?

Question 11.4 Dans ce dernier cas, proposez une modification pour améliorer $\mathcal{N}_g$. Que vaut alors $\mathcal{N}_g$?

Exercice 12 : Taille de problème

On considère la fonction `h` dont la signature est la suivante :

```
int h(int n);
```

Sur votre ordinateur, en 1 minute, la fonction h peut renvoyer un résultat jusqu'à $n = M$. Une entreprise vous propose son nouveau microprocesseur qui est 100 fois plus rapide que le vôtre ! Si vous achetez ce microprocesseur, en 1 minute votre fonction pourra sans doute atteindre des n plus grands.

Question 12.1 Donner une fourchette du nouveau n maximum atteint si le nombre d'opérations $\mathcal{N}$ effectuées par la fonction h est : n, $2n$, $3n$, n^2, n^4, 2^n

Exercice 13 : Complexité et boucles

On considère la fonction suivante :

```
int f(int n) {
  int sum = 0;

  for (int i = 0; i <= n; ++i) {
    for (int j = 0; j <= (n-i)/2; ++j) {
      for (int k = 0; k <= (n-i-2*j)/5; ++k) {
        if (i + 2*j + 5*k == n) {
          sum++;
        }
      }
    }
  }

  return sum;
}
```

Question 13.1 Tester cette fonction pour $n = 5$. Que calcule f ?

Question 13.2 On prend comme opération fondamentale l'addition. Calculer la complexité de cette fonction ?

Question 13.3 Réécrire cette fonction en inversant l'ordre des boucles. Éliminer, en le justifiant, la boucle la plus interne. Quelle est la complexité de cette nouvelle fonction ?

Question 13.4 Peut-on encore améliorer la complexité ?

Exercice 14 : Calcul de complexité

On considère les fonctions suivantes :

```
int f(int n) {
  int x = 0;

  for (int i = 1; i < n; i++) {
    for (int j = i; j < i + 5; j++) {
      x += j;
    }
  }

  return x;
}

int g(int n) {
  int y = 0;

  for (int i = 0; i < f(n); i++) {
    y += i;
  }

  return y;
}
```

Question 14.1 Quelle est l'opération fondamentale dans ces fonctions ?

Question 14.2 Quelle est la complexité de `f` ? Justifier précisément.

Question 14.3 Donner en fonction de `n` un ordre de grandeur du résultat de ce qui est calculé par `f` ?

Question 14.4 Quelle est la complexité de `g` ? Justifier précisément.

Question 14.5 Donner en fonction de `n` un ordre de grandeur du résultat de ce qui est calculé par `g` ?

Question 14.6 Réécrire la fonction `g` pour diminuer sa complexité ? Quelle est alors sa complexité ?

Exercice 15 : Application du théorème Diviser pour régner

Pour chacune des relations de récurrence suivante, dire si le théorème peut s'appliquer et donner le résultat le cas échéant en précisant lequel des trois cas est utilisé.

1. $T(n) = 3T\left(\frac{n}{2}\right) + n^2$
2. $T(n) = 4T\left(\frac{n}{2}\right) + n^2$
3. $T(n) = T\left(\frac{n}{2}\right) + 2^n$
4. $T(n) = 2^n T\left(\frac{n}{2}\right) + n^n$
5. $T(n) = 16T\left(\frac{n}{4}\right) + n$
6. $T(n) = 2T\left(\frac{n}{2}\right) + n\log n$
7. $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\log n}$
8. $T(n) = 2T\left(\frac{n}{4}\right) + n^{0.51}$
9. $T(n) = 0.5T\left(\frac{n}{2}\right) + \frac{1}{n}$
10. $T(n) = 16T\left(\frac{n}{4}\right) + n!$
11. $T(n) = \sqrt{2}T\left(\frac{n}{2}\right) + \log n$
12. $T(n) = 3T\left(\frac{n}{2}\right) + n$
13. $T(n) = 3T\left(\frac{n}{3}\right) + \sqrt{n}$
14. $T(n) = 4T\left(\frac{n}{2}\right) + cn$
15. $T(n) = 3T\left(\frac{n}{4}\right) + n\log n$
16. $T(n) = 3T\left(\frac{n}{3}\right) + \frac{n}{2}$
17. $T(n) = 6T\left(\frac{n}{3}\right) + n^2\log n$
18. $T(n) = 4T\left(\frac{n}{2}\right) + \frac{n}{\log n}$
19. $T(n) = 64T(n/8) - n^2\log n$
20. $T(n) = 7T\left(\frac{n}{3}\right) + n^2$
21. $T(n) = 4T\left(\frac{n}{2}\right) + \log n$
22. $T(n) = T\left(\frac{n}{2}\right) + n(2 - \cos n)$

3 Complexité et problèmes

Exercice 16 : La fonction puissance

Dans cet exercice, on considère le problème du calcul de la puissance $n^{\text{ième}}$ d'un nombre flottant x. L'opération fondamentale est évidemment la multiplication.

On considère tout d'abord la fonction suivante :

```
double power_v1(double x, unsigned n) {
  if (n == 0) {
    return 1;
  }

  return x * power_v1(x, n - 1);
}
```

Question 16.1 Quelle est la complexité de `power_v1` ?

On considère maintenant la fonction suivante :

```
double power_v2(double x, unsigned n) {
  if (n == 0) {
    return 1;
  }

  double p = power_v2(x, n / 2);

  if (n % 2 == 0) {
    return p * p;
  }

  return x * p * p;
}
```

Question 16.2 Quelle est la complexité de `power_v2` ? Est-elle meilleure que celle de `power_v1` ?

On considère enfin la fonction suivante :

```
double power_v3(double x, unsigned n) {
  if (n == 0) {
    return 1;
  }

  if (n % 2 == 0) {
    return power_v3(x, n/2) * power_v3(x, n/2);
  }

  return x * power_v3(x, n/2) * power_v3(x, n/2);
}
```

Question 16.3 Quelle est la complexité de `power_v3` ? Quelle conclusion peut-on en tirer ?

Exercice 17 : Suite de Fibonacci

La suite de Fibonacci est définie par :

$$\begin{cases} F(0) = 1 \\ F(1) = 1 \\ F(n) = F(n-1) + F(n-2),\ \forall n \geq 2 \end{cases}$$

On propose l'algorithme en C suivant :

```
unsigned fibo1(unsigned n) {
  if (n <= 1) {
    return 1;
  }
  return fibo1(n - 1) + fibo1(n - 2);
}
```

Question 17.1 Quelle est l'opération fondamentale de cet algorithme ?

Question 17.2 Quel est la complexité $\mathcal{C}_1(n)$ de cet algorithme ? On pourra considérer $\mathcal{C}'_1(n) = \mathcal{C}_1(n) + 1$.

Question 17.3 Proposer un algorithme itératif en C qui permet de calculer $F(n)$ avec une complexité $\mathcal{C}_2(n) = O(n)$.

```
unsigned fibo2(unsigned n);
```

—

On a les propriétés suivantes :

$$\begin{cases} F(2k) = (2F(k-1) + F(k))F(k) \\ F(2k+1) = F(k+1)^2 + F(k)^2 \end{cases}$$

Question 17.4 Proposer un algorithme récursif en C qui permet de calculer $F(n)$ grâce à ces propriétés.

```
unsigned fibo3(unsigned n);
```

Question 17.5 Donner la formule pour calculer la complexité $\mathcal{C}_3(n)$ en fonction de $\mathcal{C}_3\left(\frac{n}{2}\right)$.

Question 17.6 Résoudre cette formule grâce au théorème Diviser Pour Régner en justifiant clairement.

—

On constate que :

$$\begin{pmatrix} F(n+1) \\ F(n) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} F(1) \\ F(0) \end{pmatrix}$$

Question 17.7 Quel algorithme permet de calculer $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$? Quel est sa complexité ?

Question 17.8 Quel est alors la complexité $\mathcal{C}_4(n)$ pour calculer $F(n)$?

Exercice 18 : Recherche ternaire

On suppose qu'on dispose d'un tableau `data` de n entiers tels que les valeurs du tableau sont d'abord décroissante jusqu'à un indice m puis croissante jusqu'à la fin. Une recherche ternaire consiste à trouver m étant donné le tableau. Par exemple, étant donné le tableau $[35, 31, 23, 17, 13, 11, 24, 35]$, m vaut 5 (l'indice de 11).

Question 18.1 Soit a, b, m_1 et m_2 tels que $a < m_1 < m_2 < b$ (par exemple avec $a = 0$ et $b = n - 1$). On suppose que `data`$[m_1] <$ `data`$[m_2]$. Que peut-on dire de l'intervalle dans lequel se situe m ? Justifier. On pourra éventuellement utiliser des schémas. Même question si `data`$[m_1] >$ `data`$[m_2]$.

Question 18.2 En utilisant les résultats de la question précédente, écrire un algorithme qui effectue une recherche ternaire.

```
size_t ternary_search(int *data, size_t n);
```

Question 18.3 Quelle est alors la formule de complexité ? Résoudre la formule avec le théorème Diviser Pour Régner en justifiant clairement.

Exercice 19 : Algorithme de Karatsuba

Pour multiplier deux polynômes $A(X)$ et $B(X)$ de degré n, une technique consiste à couper les polynômes en deux de la manière suivante :

$$\begin{cases} A(X) = A_1(X) \times X^{\frac{n}{2}} + A_2(X) \\ B(X) = B_1(X) \times X^{\frac{n}{2}} + B_2(X) \end{cases}$$

avec $A_1(X)$, $A_2(X)$, $B_1(X)$ et $B_2(X)$ de degré inférieur à $\frac{n}{2}$. Ensuite, on effectue la multiplication de la manière suivante :

$$A(X)B(X) = C_1(X) \times X^n + C_2(X) \times X^{\frac{n}{2}} + C_3(X)$$

avec :

$$\begin{cases} C_1(X) = A_1(X) \times B_1(X) \\ C_2(X) = A_1(X) \times B_2(X) + A_2(X) \times B_1(X) \\ C_3(X) = A_2(X) \times B_2(X) \end{cases}$$

On peut appliquer récursivement l'algorithme de multiplication pour le calcul de $C_1(X)$, $C_2(X)$, $C_3(X)$. Quand les polynômes sont de degré 0, il s'agit d'une multiplication sur les réels. La multiplication par X^k consiste simplement à décaler les coefficients du polynômes et est donc une opération constante.

Question 19.1 Quelle est la complexité de l'addition de deux polynômes de degré n en nombre d'additions et de multiplications sur les réels.

Question 19.2 Donner la formule pour calculer la complexité $C(n)$ de la multiplication de deux polynômes de degré n en nombre d'additions et de multiplications sur les réels, en fonction de $C(\frac{n}{2})$. Justifier précisément.

Question 19.3 Résoudre cette formule grâce au Théorème Diviser pour Régner.

On calcule désormais $C_2(X)$ de la manière suivante :

$$C_2(X) = (A_1(X) + A_2(X)) \times (B_1(X) + B_2(X)) - C_1(X) - C_3(X)$$

Question 19.4 Vérifier que ce calcul est bien exact

Question 19.5 Donner la formule pour calculer la complexité $C(n)$ de la multiplication de deux polynômes de degré n en nombre d'additions et de multiplications sur les réels, en fonction de $C(\frac{n}{2})$. Justifier précisément.

Question 19.6 Résoudre cette formule grâce au Théorème Diviser pour Régner.

Exercice 20 : Problème des points les plus proches

Le problème des points les plus proches consiste à trouver, parmi un ensemble de points du plan, les deux points les plus proches.

On considère la structure suivante pour représenter un point :

```
struct point {
  double x;
  double y;
};
```

On suppose qu'on dispose d'une fonction `distance` qui calcule la distance euclidienne entre deux points :

```
double distance(const struct point *p1, const struct point *p2);
```

L'opération fondamentale considérée pour ce problème est l'appel à la fonction `distance`.

Question 20.1 Écrire un algorithme simple qui permet de résoudre le problème des points les plus proches. La fonction prend en paramètre un tableau `data` de `n` points. Les indices des deux points les plus proches seront stockés dans `*i1` et `*i2`. La fonction renvoie la plus petite distance obtenue.

```
double closest_points(const struct point *data, size_t n,
    size_t *i1, size_t *i2);
```

Question 20.2 Quelle est la complexité de cet algorithme simple ?

—

Nous allons maintenant définir un algorithme récursif qui résout ce problème. L'idée de cet algorithme est le suivant :

1. on trie l'ensemble de points par ordre des x croissants ;
2. on résout le problème :
   (a) on coupe l'ensemble des points en deux sous-ensembles de taille égale avec une ligne $x = x_M$;
   (b) on résout le problème récursivement sur ces deux sous-ensembles, ce qui nous donne une distance minimale d_L pour le sous-ensemble gauche et d_R pour le sous-ensemble droit.
   (c) on cherche la distance minimale d_M entre les couples de points dont le premier est dans le sous-ensemble gauche et le second est dans le sous-ensemble droit ;
   (d) on conclue en fonction de d_L, d_R et d_M.

Question 20.3 Donner le code d'une fonction qui permet de comparer deux points pour le tri effectué à l'étape 1.

```
int point_comp(const struct point *p1, const struct point *p2);
```

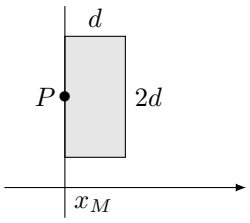


FIGURE 2 – Cas limite de recherche pour l'étape 2c

Question 20.4 Donner le nom d'un algorithme pour effectuer l'étape 1. Expliquer son principe brièvement. Quelle est sa complexité ?

Question 20.5 Quels sont les cas d'arrêt dans les appels récursifs des étapes 2a et 2b ?

Question 20.6 À l'étape 2c, expliquer avec des phrases un algorithme simple qui permet d'obtenir le résultat. Quel est sa complexité ? Quelle est alors la formule de complexité de l'algorithme de l'étape 2. La résoudre avec le théorème Diviser Pour Régner.

—

Nous allons maintenant tenter d'améliorer la complexité de l'étape 2c. On note $d = \min(d_L, d_R)$. On considère un point P qui se situe à gauche. Dans le pire des cas, il est proche de la ligne de séparation $x = x_M$. On cherche donc les points à droite qui sont à une distance inférieure à d, ce qui réduit le champ de recherche à un rectangle de taille $(d, 2d)$ comme le montre la figure 2.

Question 20.7 Combien de points y a-t-il au maximum dans la partie grise de la figure 2 ? Justifier.

Question 20.8 Quelle est alors la complexité de l'étape 2c ? Justifier très précisément.

Question 20.9 Quelle est alors la formule de complexité de l'algorithme de l'étape 2. La résoudre avec le théorème Diviser Pour Régner.

Question 20.10 Au total, quelle est la complexité de l'algorithme complet ?

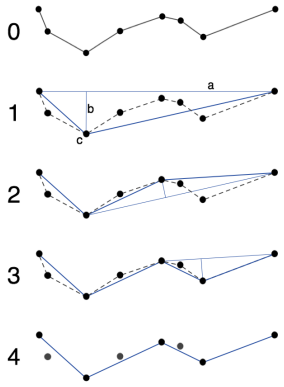


FIGURE 3 – Simplification d'une polyligne

Exercice 21 : Simplification de polyligne

Une polyligne est une ligne brisée composée de n points. Si on n'a pas besoin d'une grande précision, on peut simplifier la polyligne en supprimant des points. Plus précisément, on va supprimer des points qui sont suffisamment proches de la ligne formée par deux autres points de la polyligne. L'algorithme utilisé est le suivant :

1. À l'initialisation, on sélectione le premier et le dernier point de la polyligne qui forment les bornes.
2. À chaque étape, on parcourt tous les points entre les bornes et on sélectionne le point le plus éloigné du segment formé par les bornes puis :
   — s'il n'y a aucun point, l'algorithme se termine ;
   — si cette distance est inférieure à un certain seuil ε, on supprime tous les points entre les bornes ;
   — sinon, la polyligne n'est pas directement simplifiable, on appelle récursivement l'algorithme sur deux sous-parties de la polyligne : de la première borne au point le plus distant, et du point le plus distant à la deuxième borne.

La figure 3 montre une exécution de cet algorithme.

Question 21.1 Que se passe-t-il quand $\varepsilon = 0$?

Question 21.2 On suppose que le point le plus distant est le p^{e} de la polyligne. Donner la formule de complexité pour cet algorithme en fonction de n et p. Expliquer.

Question 21.3 Résoudre cette formule pour donner la complexité en moyenne de l'algorithme.

Question 21.4 Quelle est la complexité en pire cas ? Dans quel cas arrive-t-elle ?

Exercice 22 : Multiplication de matrices

Soit $A = (a_{ij})_{1 \leq i,j \leq n}$ et $B = (b_{ij})_{1 \leq i,j \leq n}$ où i représente le numéro de colonne et j le numéro de ligne. Le produit $C = (c_{ij})_{1 \leq i,j \leq n}$ des deux matrices A et B est donné par :

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Question 22.1 Quelle est la complexité de l'algorithme naïf qui implémente la formule ci-dessus pour des matrices de taille n ? Justifier.

—

On décide de diviser les matrices en sous-matrices de taille $\frac{n}{2}$:

$$A = \begin{pmatrix} A_{11} & A_{21} \\ A_{12} & A_{22} \end{pmatrix} \; B = \begin{pmatrix} B_{11} & B_{21} \\ B_{12} & B_{22} \end{pmatrix} \; C = \begin{pmatrix} C_{11} & C_{21} \\ C_{12} & C_{22} \end{pmatrix}$$

On a alors :

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

Question 22.2 Quelle est la complexité d'une addition de deux matrices de taille n ?

Question 22.3 Donner la formule pour calculer la complexité de cette multiplication.

Question 22.4 Résoudre cette formule grâce au théorème Diviser Pour Régner en justifiant clairement.

—

On décide de calculer les matrices suivantes :

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22})B_{11} \\ M_3 &= A_{11}(B_{12} - B_{22}) \\ M_4 &= A_{22}(B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12})B_{22} \\ M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

On a alors :

$$\begin{aligned} C_{11} &= M_1 + M_4 - M_5 + M_7 \\ C_{12} &= M_3 + M_5 \\ C_{21} &= M_2 + M_4 \\ C_{22} &= M_1 - M_2 + M_3 + M_6 \end{aligned}$$

Question 22.5 Combien y a-t-il d'additions (ou de soustractions) dans le calcul des C_{ij} ? Combien y a-t-il de multiplications dans le calcul des C_{ij} ?

Question 22.6 Donner la formule pour calculer la complexité de cette multiplication.

Question 22.7 Résoudre cette formule grâce au théorème Diviser Pour Régner en justifiant clairement.

Exercice 23 : Le grand saut

Le problème est de déterminer à partir de quel étage d'un immeuble sauter par une fenêtre est fatal. Vous êtes dans un immeuble à n étages (numérotés de 1 à n) et vous disposez de k étudiants. Il n'y a qu'une opération possible pour tester si la hauteur d'un étage est fatale : faire sauter un étudiant par la fenêtre. S'il survit, vous pouvez le réutiliser ensuite, sinon vous ne pouvez plus.

Dans chacun des cas suivants, vous devez proposer un algorithme pour trouver la hauteur à partir de laquelle un saut est fatal (c'est-à-dire renvoyer $n+1$ si on survit encore en sautant du n^{e} étage) en faisant le minimum de sauts. Vous pouvez décrire les algorithmes en français.

Question 23.1 On suppose que $k = 1$. Proposer un algorithme naïf. Quel est sa complexité ?

Question 23.2 On suppose que $k \geq \lceil \log_2(n) \rceil$. Proposer un algorithme en $O(\log_2 n)$ sauts. Quel algorithme connu fonctionne sur ce principe.

Question 23.3 On suppose que $k < \lceil \log_2(n) \rceil$. En vous inspirant de l'algorithme précédent, proposer un algorithme en $O(k + \frac{n}{2^{k-1}})$.

Question 23.4 On suppose que $k = 2$, proposer un algorithme en $O(\sqrt{n})$.

Exercice 24 : Recherche d'un élément majoritaire

On dispose d'un tableau avec n éléments. La seule opération disponible sur les éléments est le test d'égalité (avec $=$ ou $\neq$). Étant donné un élément x, on définit $c(x)$ comme le nombre d'éléments du tableau égaux à x. On dit qu'un élément x est majoritaire dans le tableau si $c(x) > \frac{n}{2}$. On définit la structure suivante permettant de stocker la valeur de l'élément majoritaire ainsi que le nombre d'éléments égaux à l'élément majoritaire.

```
struct majority {
  int value;    // x
  size_t count; // c(x)
};
```

Question 24.1 Y a-t-il toujours un élément majoritaire dans un tableau ? Justifier.

Question 24.2 Combien peut-on avoir d'élément majoritaire au maximum dans un tableau ? Justifier.

Question 24.3 Écrire un algorithme qui calcule $c(x)$.

```
size_t count(int *data, size_t n, int x);
```

Question 24.4 Écrire un algorithme simple utilisant l'algorithme précédent qui dit s'il existe un élément majoritaire et qui le stocke le cas échéant dans `res`. Quel est sa complexité ?

```
bool search_majority(int *data, size_t n, struct majority *res);
```

—

On veut construire un algorithme récursif permettant de chercher un élément majoritaire. Pour cela, on coupe le tableau en deux et on va chercher un élément majoritaire dans chacune des deux moitiés.

Question 24.5 Justifier que si un élément est majoritaire dans le tableau, il est majoritaire dans au moins une des deux moitiés du tableau.

Question 24.6 On définit le début de l'algorithme qui va effectuer la recherche entre les indices i (inclus) et j (exclus) de la manière suivante :

```
bool search_majority_recursive(int *data, size_t i, size_t j,
                               struct majority *res) {
  size_t size = j - i;

  if (size == 1) {
    res->value = XXX;
    res->count = YYY;
    return ZZZ;
```

```
}

size_t mid = (i + j) / 2;

struct majority x;
bool bx = search_majority_recursive(data, i, mid, &x);

struct majority y;
bool by = search_majority_recursive(data, mid, j, &y);

...
```

Donner les valeurs `XXX`, `YYY` et `ZZZ`.

Question 24.7 Si `bx` et `by` valent tous les deux `false`, que renvoie-t-on ?

Question 24.8 Si `bx` vaut `true`, expliquer comment on vérifie s'il est bien majoritaire sur l'ensemble du tableau.

Question 24.9 Compléter l'algorithme précédent (sans réécrire le début).

Question 24.10 Donner la formule pour calculer $\mathcal{C}(n)$, la complexité pour un tableau de taille n, en fonction de $\mathcal{C}(n/2)$.

Question 24.11 Résoudre cette formule grâce au théorème Diviser Pour Régner en justifiant clairement.

—

On veut essayer de faire mieux. Pour cela, on va chercher un élément pseudo-majoritaire. C'est-à-dire que l'algorithme va renvoyer :

- `false` si on a la garantie qu'il n'y a aucun élément majoritaire ;
- `true` si on a un élément pseudo-majoritaire, c'est-à-dire qu'on a un élément x qui apparaît **au plus** p fois dans le tableau, avec $p > \frac{n}{2}$, et que tout autre élément n'apparaît jamais plus de $n - p$ fois.

Question 24.12 Si on a un élément x pseudo-majoritaire, justifier qu'aucun autre élément y ne peut être l'élément majoritaire.

Question 24.13 Si un élément x est majoritaire, justifier qu'il est pseudo-majoritaire.

Question 24.14 Si on trouve un élément pseudo-majoritaire, décrire comment on peut déterminer s'il est majoritaire ou non. Quelle est la complexité de cette opération ?

Question 24.15 En réutilisant notre structure `majority` pour stocker x et p, on définit le début de l'algorithme qui va effectuer la recherche entre les indices i (inclus) et j (exclus) de la manière suivante :

```
bool search_pseudo_majority(int *data, size_t i, size_t j,
                                          struct majority *res) {
  size_t size = j - i;

  if (size == 1) {
    res->value = XXX;
    res->count = YYY;
    return ZZZ;
  }

  size_t mid = (i + j) / 2;

  struct majority x;
  bool bx = search_pseudo_majority(data, i, mid, &x);

  struct majority y;
  bool by = search_pseudo_majority(data, mid, j, &y);

  ...
```

Donner les valeurs `XXX`, `YYY` et `ZZZ`.

Question 24.16 Si `bx` et `by` valent tous les deux `false`, que renvoie-t-on ? Justifier.

Question 24.17 On suppose que `bx` vaut `true` et `by` vaut `false`. Justifier que `x.value` apparaît au plus `x.count + size / 4` fois entre i et j. Justifier qu'aucun autre élément n'apparaît `size - (x.count + size / 4)` fois.

Question 24.18 On suppose désormais que `bx` et `by` valent tous les deux `true`. Dans le cas où `x.value == y.value`, que renvoie-t-on ? Justifier.

Question 24.19 On suppose désormais que `x.value != y.value`. Dans le cas où `x.count == y.count`, que renvoie-t-on ? Justifier.

Question 24.20 On suppose que `x.count > y.count`. Justifier que, dans ce cas, `x.value` est pseudo-majoritaire avec p qui vaut `size / 2 + x.count - y.count`.

Question 24.21 Dans tous ces cas, quel est la complexité des opérations en jeu ?

Question 24.22 Donner la formule pour calculer $\mathcal{C}(n)$, la complexité de la recherche d'un pseudo-majoritaire pour un tableau de taille n, en fonction de $\mathcal{C}(n/2)$.

Question 24.23 Résoudre cette formule grâce au théorème Diviser Pour Régner en justifiant clairement.

Question 24.24 Au final, quelle est la complexité de la recherche d'un majoritaire en utilisant la recherche d'un pseudo-majoritaire.

Exercice 25 : Problème du sous-tableau maximum

Le problème du sous-tableau maximum *(Maximum subarray problem)* consiste, à partir d'un tableau A de taille n, à trouver le sous-tableau contigu pour lequel la somme des éléments est maximale. En fait, on cherche les indice i et j tel que la somme suivante soit maximale :

$$S = \sum_{i}^{j-1} A[i]$$

Par exemple, pour le tableau suivant :

$$A = [-2, 1, -3, 4, -1, 2, 1, -5, 4]$$

le sous-tableau maximum est $[4, -1, 2, 1]$ avec une somme de 6.

Question 25.1 Si le tableau ne contient que des nombres positifs, quelle est la solution ?

Question 25.2 Si le tableau ne contient que des nombres négatifs, quelle est la solution ?

—

On suppose désormais que le tableau contient des positifs et des négatifs. On suppose aussi qu'on dispose de la structure suivante pour stocker les indices du sous-tableau maximum (la somme de ce sous-tableau sera la valeur de retour des fonctions).

```
struct sub {
  size_t i; // included
  size_t j; // excluded
};
```

Question 25.3 Écrire une fonction qui permet de calculer la somme entre les indices i (inclus) et j (exclus). Quelle est sa complexité ?

```
int sub_sum(const int *data, size_t i, size_t j);
```

Question 25.4 Écrire une fonction triviale qui permet de calculer le sous-tableau maximum. Quelle est sa complexité ?

```
int max_sub_1(const int *data, size_t n, struct sub *res);
```

—

On veut désormais utiliser un algorithme de type «Diviser pour régner». L'idée est de couper le tableau en deux, puis de résoudre récursivement le problème pour les deux moitiés, et de considérer le meilleur résultat parmi celui de droite, celui de gauche et celui qui prend en compte le milieu.

Question 25.5 Écrire une fonction qui permet de chercher le sous-tableau maximal entre les indices `lo` (inclus) et `hi` (exclus) avec comme condition que le sous-tableau doit contenir l'indice `mid` donné. On pourra d'abord parcourir le tableau à gauche de `mid` depuis `mid` pour trouver le sous-tableau maximum, puis recommencer l'opération à droite et conclure. Quelle est la complexité de cette fonction ?

```
int max_middle_sum(const int *data,
    size_t lo, size_t mid, size_t hi, struct sub *res);
```

Question 25.6 Écrire une fonction qui permet de calculer le sous-tableau maximum avec la méthode «Diviser pour régner».

```
int max_sub_2(const int *data, size_t n, struct sub *res);
```

Question 25.7 Donner la formule pour calculer $\mathcal{C}(n)$, la complexité pour un tableau de taille n, en fonction de $\mathcal{C}(n/2)$.

Question 25.8 Résoudre cette formule grâce au théorème Diviser Pour Régner en justifiant clairement.

—

On va tenter d'obtenir une meilleure complexité. On appelle S_k le sous-tableau qui termine à l'indice k (exclus) maximum. En particulier, puisqu'on a supposé que le tableau contenait des positifs et des négatifs, on peut poser $S_0 = 0$.

Question 25.9 Si $S_k \leq 0$, pourquoi doit-on commencer une nouvelle séquence de sous-tableau à l'indice k ? Dans le cas contraire, que vaut S_{k+1} en fonction de S_k et A_k ?

Question 25.10 Écrire une fonction qui permet de calculer le sous-tableau maximum avec la méthode décrite précédemment (c'est-à-dire qui permet de calculer $\max_{0 \leq k \leq n} S_k$). Quelle est sa complexité ?

```
int max_sub_3(const int *data, size_t n, struct sub *res);
```

Exercice 26 : Recherche de pic dans un tableau

Un pic dans un tableau est un élément du tableau tel que ses deux éléments voisins sont plus petits ou égaux. Si l'élément est au bord du tableau, on ne considère qu'un seul voisin. On s'intéresse au problème de trouver un pic dans un tableau.

Question 26.1 On considère un tableau trié par ordre strictement croissant, déterminer le ou les pics.

Question 26.2 On considère un tableau avec tous les éléments égaux, déterminer le ou les pics.

Question 26.3 Écrire une fonction simple qui recherche un pic dans un tableau et renvoie l'indice du pic.

```
size_t array_find_peak(const int *data, size_t n);
```

Question 26.4 Quelle est la complexité de cette fonction ?

—

On veut désormais utiliser un algorithme de type «Diviser pour régner». L'idée est de regarder si l'élément au milieu du tableau est un pic. Si ce n'est pas le cas, alors, on regarde si l'élément à gauche est plus grand et alors, il existe un pic à gauche et on continue sur la partie gauche. Sinon, on procède de même à droite.

Question 26.5 Justifier que si un élément n'est pas un pic, au moins un de ses deux éléments voisins est strictement plus grand.

Question 26.6 Écrire une fonction qui recherche un pic suivant la méthode proposée.

```
size_t array_find_peak(const int *data, size_t n);
```

Question 26.7 Donner la formule pour calculer $\mathcal{C}(n)$, la complexité pour un tableau de taille n, en fonction de $\mathcal{C}(n/2)$.

Question 26.8 Résoudre cette formule grâce au théorème Diviser Pour Régner en justifiant clairement.

Exercice 27 : Leaders dans un tableau

Soit un tableau d'entiers tous distincts. On appelle *leader* un élément du tableau qui est plus grand que tous les éléments qui le suivent dans le tableau. L'élément le plus à droite est toujours *leader*. On considère le problème de compter le nombre de *leaders* dans un tableau.

Question 27.1 Écrire une fonction qui détermine si l'élément à l'indice i est *leader* dans le tableau. On supposera $i < n$. Quelle est sa complexité ?

```
bool is_leader(const int *data, size_t n, size_t i);
```

Question 27.2 À l'aide de la fonction précédente, écrire une fonction qui détermine le nombre de *leaders* dans un tableau. Quelle est sa complexité ?

```
size_t count_leaders(const int *data, size_t n);
```

Question 27.3 Désormais, on va partir de la droite du tableau et parcourir le tableau de droite à gauche. À quelle condition un entier rencontré est-il *leader* ? Soyez précis dans votre réponse.

Question 27.4 Écrire une fonction qui utilise le principe précédent. Quelle est sa complexité ?

```
size_t count_leaders(const int *data, size_t n);
```

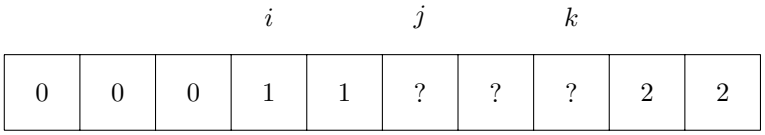


FIGURE 4 – Une étape intermédiaire pour résoudre le problème du drapeau hollandais

Exercice 28 : Problème du drapeau hollandais

Le problème du drapeau hollandais consiste à réorganiser un tableau d'éléments identifiés par leur couleur sachant que seules trois couleurs sont présentes : bleu, blanc et rouge. Le nombre d'éléments de chaque couleur n'est pas connu, et leur disposition initiale n'est pas connue non plus. À la fin, tous les éléments de même couleur doivent être rangés ensemble dans le tableau et l'ordre entre les couleurs doit être respecté.

Dans cet exercice, on considère que bleu = 0, blanc = 1 et rouge = 2.

On considère qu'on a résolu en partie le problème du drapeau hollandais en ayant placé tous les 0 rencontrés au début, suivis de tous les 1 rencontrés, et tous les 2 rencontrés sont disposés à la fin du tableau. Les éléments entre les zones 1 et 2 sont à traiter. On dispose de trois indice : i désigne l'indice de la première case après la zone 0, j désigne l'indice de la première case après la zone 1, k désigne l'indice de la case avant la zone 2. La figure 4 montre la situation.

Question 28.1 À quelles valeurs sont initialisées i, j, et k au début de l'algorithme ?

Question 28.2 On traite désormais l'élément à l'indice j. Si cet élément vaut 1, que doit-on faire ? Justifier.

Question 28.3 Si cet élément vaut 0, que doit-on faire ? Justifier.

Question 28.4 Si cet élément vaut 2, que doit-on faire ? Justifier.

Question 28.5 Écrire l'algorithme obtenu. Quel est sa complexité ?

```
void dutch_problem(int *data, size_t n);
```

Question 28.6 À votre avis, quel algorithme vu en cours pourrait bénéficier de cet algorithme ? Expliquer dans ce cas à quoi correspondent les trois «couleurs».

4 Tableaux et chaînes de caractères

Exercice 29 : Tableaux

Pour chaque algorithme, on donnera l'opération considérée et sa complexité. On utilisera chaque fonction dans `main` sur un exemple quelconque.

Question 29.1 Écrire une fonction qui alloue un tableau d'entiers à l'aide de `calloc(3)`. Utiliser la fonction `rand(3)` pour remplir le tableau à l'aide de valeur entre 0 et 99. Dans `main`, on appellera cette fonction avec 10000 comme argument.

```
int *array_new(size_t size);
```

Question 29.2 Écrire une fonction qui renvoie l'indice de l'élément le plus grand du tableau.

```
size_t array_index_max(const int *data, size_t size);
```

Question 29.3 Écrire une fonction qui calcule la somme des éléments du tableau.

```
int array_sum(const int *data, size_t size);
```

Question 29.4 Écrire une fonction qui renvoie le nombre d'occurrences dans le tableau d'une valeur passées en paramètre.

```
size_t array_count(const int *data, size_t size, int value);
```

Question 29.5 Écrire un algorithme qui effectue un décalage du tableau d'une case vers la gauche, la première valeur étant placée à la fin du tableau.

```
void array_shift_left(int *data, size_t size);
```

Question 29.6 Écrire une fonction qui renvoie l'indice de la plus grande suite de nombres pairs dans le tableau.

```
size_t array_longest_even_seq(const int *data, size_t size);
```

Exercice 30 : Tableaux avancés

Pour chaque algorithme, on donnera l'opération considérée et sa complexité. On utilisera chaque fonction dans `main` sur un exemple bien choisi.

Question 30.1 Écrire une fonction qui renverse l'ordre des éléments d'un tableau.

```
void array_swap(int *data, size_t size);
```

Question 30.2 Écrire une fonction qui supprime les éléments consécutifs égaux pour n'en garder qu'un seul. La fonction renverra la nouvelle taille du tableau.

```
size_t array_uniq(int *data, size_t size);
```

Question 30.3 Écrire une fonction qui détermine si le tableau suivant représente une permutation, c'est-à-dire qu'il contient une et une seule fois tous les nombres compris entre 0 et (`size` - 1).

```
bool array_is_permutation(const int *data, size_t size);
```

Exercice 31 : Fonctions sur les chaînes de caractères

Question 31.1 Afficher la chaîne de caractère passée en argument du programme. On mettra des guillemets autour de la chaîne sur la ligne de commande pour qu'elle soit considéré comme un argument unique pour le programme :

```
./compute_string "c'est pas faux !"
```

Question 31.2 Écrire une fonction qui calcule la longueur de la chaîne de caractère. Comparer avec ce que renvoie `strlen(3)`.

Question 31.3 Écrire une fonction qui compte le nombre d'espaces dans la chaînes. Indice : `isspace(3)`.

Question 31.4 Écrire une fonction qui affiche la chaîne en enlevant les voyelles (non-accentuées) de la chaîne de caractères.

Question 31.5 Écrire une fonction qui détermine si la chaîne est bien parenthésée.

Question 31.6 Écrire une fonction qui calcule la valeur numérique d'un entier représenté en binaire par une chaîne de caractère.

Exercice 32 : Recherche de sous-chaîne

La recherche de sous-chaîne consiste à recherche une chaîne `needle` de taille k dans une chaîne `haystack` de taille n. Le résultat envoyé est l'indice du début de la sous-chaîne si elle est présente ou n si elle est absente.

Question 32.1 Donner un algorithme simple pour rechercher une sous-chaîne.

```
size_t find_simple(const char *needle, const char *haystack);
```

Question 32.2 Quelle est sa complexité en pire cas ?

—

On souhaite maintenant améliorer la complexité en pire cas. Pour cela, on calcule une empreinte de la sous-chaîne à chercher en faisant une addition de tous ses caractères. Puis, on réalise une addition de k caractères consécutifs de la chaîne dans laquelle on cherche et, si les empreintes correspondent, on réalise une comparaison des chaînes.

Question 32.3 Si x_i est l'empreinte des caractères entre i et $i+k-1$, comment calculer x_{i+1} l'empreinte des caractères entre $i+1$ et $i+k$ en fonction de x_i ? Quelle est la complexité de cette opération ?

Question 32.4 Donner l'algorithme qui utilise les empreintes.

```
size_t find_rk(const char *needle, const char *haystack);
```

Question 32.5 Quelle est sa complexité en pire cas ? Justifier très précisément. En particulier, on pourra discuter sur la fonction de calcul d'empreinte utilisée.

—

On souhaite adapter l'algorithme précédent à la recherche d'un ensemble de m sous-chaînes de même taille k.

Question 32.6 Quelle structure de données utiliser pour stocker l'ensemble des empreintes ? Justifier très précisément.

Question 32.7 Quelle est alors la complexité ?

Exercice 33 : Structure `argz`

La structure `argz` est un tableau de chaînes de caractères stockées dans un bloc de mémoire contigu, c'est-à-dire que toutes les chaînes sont mises bout à bout en mémoire, y compris le zéro final. La structure retient également la taille totale de la zone mémoire utilisée, qu'on appelera n dans la suite. La figure 5 montre un exemple avec trois éléments et $n = 12$.

```
struct argz {
  char *data;   // start of the memory block
  size_t size;  // size of the memory block
};
```

Dans cet exercice, en plus des fonctions de gestion de la mémoire, vous aurez le droit d'utiliser les fonctions suivantes (ce qui implique que vous devrez réimplémenter toute autre fonction manipulant des chaînes de caractères) :

```
size_t strlen(const char *s);
void *memcpy(void *dest, const void *src, size_t n);
```

Question 33.1 Si la structure `argz` contient k chaînes de caractères de tailles respectives $\ell_1, \ldots, \ell_k$, que vaut n ?

Question 33.2 Écrire une fonction qui initialise une structure `argz` vide.

```
void argz_create(struct argz *self);
```

Question 33.3 Écrire une fonction qui initialise une structure `argz` à l'aide d'un tableau de chaînes de caractères.

```
void argz_create_from_args(struct argz *self,
                           int argc, const char *argv[]);
```

Question 33.4 Écrire une fonction qui compte le nombre d'éléments présents. Quelle est la complexité de cette fonction ?

```
size_t argz_count(const struct argz *self);
```

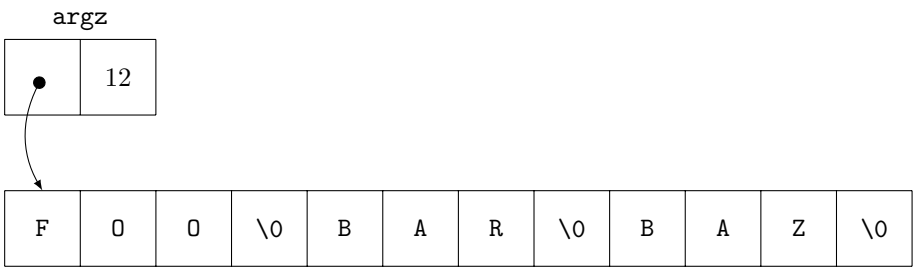


FIGURE 5 – La structure `argz` contenant trois éléments : `"FOO"`, `"BAR"` et `"BAZ"`

Question 33.5 Écrire une fonction qui renvoie le i[e] élément. Quelle est la complexité de cette fonction ?

```
char *argz_ith(struct argz *self, size_t i);
```

Question 33.6 Écrire une fonction qui insère un élément `entry` avant un élément déjà présent `before`. Si `before` est nul, alors on insère l'élément à la fin. Quelle est la complexité de cette fonction ?

```
void argz_insert(struct argz *self,
                                 const char *entry, char *before);
```

Question 33.7 Écrire une fonction qui supprime un élément `entry`. Quelle est la complexité de cette fonction ?

```
void argz_remove(struct argz *self, char *entry);
```

Question 33.8 Écrire une fonction qui, à partir d'un élément `entry`, calcule l'élément suivant. Si `entry` est nul, on renvoie le premier élément. S'il n'y a pas d'élément suivant, on renvoie `NULL`.

```
char *argz_next(struct argz *self, char *entry);
```

L'idée de cette fonction est de pouvoir écrire une itération de la manière suivante :

```
for (char *entry = argz_next(argz, NULL); entry != NULL;
                                 entry = argz_next(argz, entry)) {
  // ...
}
```

Exercice 34 : Crible d'Ératosthène

Un nombre est dit premier s'il admet exactement 2 diviseurs distincts (1 et lui-même). 1 n'est donc pas premier. On désigne sous le nom de crible d'Ératosthène une méthode de recherche des nombres premiers plus petits qu'un entier naturel n donné. La méthode est la suivante :

1. On supprime tous les multiples de 2 inférieurs à n.
2. L'entier 3 n'ayant pas été supprimé, il ne peut être multiple des entiers qui le précèdent, il est donc premier. On supprime alors tous les multiples de 3 inférieurs à n.
3. L'entier 5 n'ayant pas été supprimé, il ne peut être multiple des entiers qui le précèdent, il est donc premier. On supprime alors tous les multiples de 5 inférieurs à n.
4. Et ainsi de suite jusque n. Les valeurs n'ayant pas été supprimées sont les nombres entiers plus petits que n.

Pour programmer cette méthode, on va utiliser un tableau `is_prime` de n entiers qui contiendra des 1 et des 0. On donne à ces 1 et 0 le sens suivant : si `is_prime[i]` vaut 0 alors `i` n'est *pas premier*, et si `is_prime[i]` vaut 1 alors `i` est *premier*. Lors de la méthode, on élimine les nombres non premiers au fur et à mesure qu'on les rencontre. Donc au départ, on suppose que tous les entiers sont premiers.

Question 34.1 Récupérer la taille n sur la ligne de commande.

Question 34.2 Allouer un tableau `is_prime` de taille n à l'aide de `calloc(3)`.

Question 34.3 Initialiser ce tableau avec des 1.

Question 34.4 Implémenter la méthode du crible d'Ératosthène.

Question 34.5 Afficher les nombres premiers inférieurs à n.

```
$ ./eratosthene 100
Nombres premiers inférieurs à 100 :
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

Exercice 35 : Méthode des k plus proches voisins

On considère un tableau de n données. Chaque données est caractérisées par une position dans un espace à deux dimensions $(x, y) \in \mathbb{R}^2$, une classification qui peut par exemple être un entier $c \in \mathbb{N}$. Dans notre cas, on suppose que c ne peut prendre que les valeurs 1 ou 2.

La méthode des k plus proches voisins consiste à attribuer à une nouvelle donnée la classification majoritaire (relative) parmi ses k plus proches voisins. Ici, «proche» sera entendu au sens de la distance euclidienne entre les positions des données.

On utilise la structure de données suivante pour représenter une données :

```
struct data {
  double x;
  double y;
  int c;
};
```

On suppose qu'on dispose d'une fonction qui donne la distance euclidienne entre deux données :

```
double distance(const struct data *d1, const struct data *d2);
```

Question 35.1 Montrer que suivant la valeur de k, la classification d'une nouvelle donnée peut changer.

Question 35.2 Proposer un algorithme naïf pour calculer la classification d'une nouvelle donnée. Quel est sa complexité ?

```
int classify1(struct data *reference, size_t n, size_t k,
                                    const struct data *candidate);
```

Question 35.3 Est-il nécessaire de trier tous les éléments ? En vous inspirant d'un algorithme de sélection, proposer un nouvel algorithme pour calculer la classification d'une nouvelle donnée. Quel est sa complexité ?

```
int classify2(struct data *reference, size_t n, size_t k,
                                    const struct data *candidate);
```

Question 35.4 On suppose que k est une petite constante. On peut alors parcourir k fois le tableau pour trouver les k plus proches voisins. Décrire l'algorithme. Quel est sa complexité ?

```
int classify3(struct data *reference, size_t n, size_t k,
                                    const struct data *candidate);
```

Question 35.5 Si k est de l'ordre de $\log(n)$, la réponse à la question précédente est-elle toujours pertinente ?

5 Listes chaînées

Exercice 36 : Liste ordonnée

Le but de cet exercice est de gérer une liste chaînée dont les éléments sont ordonnés par ordre croissant.

On définit les types suivants :

```
struct sl_node {
  int data;
  struct sl_node *next;
};

// sorted list
struct sl {
  struct sl_node *first;
};
```

Question 36.1 Donner le code d'une fonction qui ajoute à une liste triée un entier passé en paramètre, de façon à ce que la liste reste triée. Quelle est sa complexité ?

```
void sl_add(struct sl *self, int data);
```

Question 36.2 Donner le code d'une fonction qui prend en paramètre un tableau de k entiers non-triés et qui les insère dans une liste ordonnée. Quelle est sa complexité ?

```
void sl_import(struct sl *self,
    const int *other, size_t k);
```

Question 36.3 Donner le code d'une fonction qui supprime les doublons d'une liste ordonnée. Quelle est sa complexité ?

```
void sl_uniq(struct sl *self);
```

Exercice 37 : Manipulation de liste chaînée

On utilise la structure de liste suivante :

```
struct list_node {
  int data;
  struct list_node *next;
};

struct list {
  struct list_node *first;
};
```

Question 37.1 Donner le code d'une fonction qui prend en paramètre une liste et retourne une liste miroir, c'est-à-dire avec les mêmes éléments en ordre inverse. Quelle est sa complexité ?

```
void list_mirror(const struct list *self, struct list *res);
```

Question 37.2 Donner le code d'une fonction qui prend en paramètre une liste et retourne une copie de la liste. Quelle est sa complexité ?

```
void list_copy(const struct list *self, struct list *res);
```

Question 37.3 Donner le code d'une fonction qui prend en paramètre deux listes et qui renvoie la concaténation des deux listes. Quelle est sa complexité ?

```
void list_concat(const struct list *l1, const struct list *l2,
    struct list *res);
```

Exercice 38 : Listes doublement chaînées

On définit la structure suivante qui représente une liste doublement chaînée :

```
struct node {
  int data;
  struct node *next;
  struct node *prev;
}

struct list {
  struct node *first;
  struct node *last;
}
```

Dans la suite, on prendra garde à traiter les cas où la liste est vide, ainsi qu'à maintenir le double chaînage cohérent.

Question 38.1 Écrire une fonction `list_size` qui renvoie la taille de la liste. Quelle est sa complexité ?

```
size_t list_size(const struct list *self);
```

Question 38.2 Écrire une fonction `list_add_front` qui ajoute un élément en début de liste. Quelle est la complexité de cette fonction ?

```
void list_add_front(struct list *self, int data);
```

Question 38.3 Écrire une fonction `list_remove_node` qui supprime un nœud donné de la liste.

```
void list_remove_node(struct list *self, struct node *node);
```

Question 38.4 Écrire une fonction `list_remove_value` qui supprime tous les nœuds ayant une certaine valeur. On pourra utiliser la fonction de la question précédente. Quelle est sa complexité ?

```
void list_remove_value(struct list *self, int value);
```

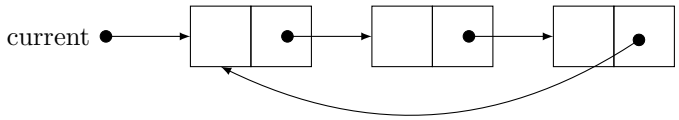


FIGURE 6 – Liste chaînée cyclique

Exercice 39 : Liste chaînée cyclique

Une liste chaînée cyclique est une liste chaînée dans laquelle le dernier nœud est relié au premier, formant ainsi un cycle. La figure 6 montre une liste simplement chaînée cyclique avec trois éléments. Le but de cet exercice est d'implémenter des fonctions relatives aux listes chaînées cyclique. Pour cela, on définit la structure de données suivante :

```
struct node {
  int data;
  struct node *next;
};

struct cycle {
  struct node *current;
};
```

Question 39.1 Écrire une fonction qui initialise une liste chaînée cyclique à la liste vide.

```
void cycle_create(struct cycle *self);
```

Question 39.2 Écrire une fonction qui calcule la taille d'une liste chaînée cyclique. Donner sa complexité.

```
size_t cycle_size(const struct cycle *self);
```

Question 39.3 Écrire une fonction qui ajoute un élément à une liste chaînée cyclique. On prendra garde au cas où la liste est vide. Donner sa complexité.

```
void cycle_add(struct cycle *self, int data);
```

Question 39.4 Écrire une fonction qui retire l'élément courant d'une liste chaînée cyclique. On prendra garde au cas où la liste ne contient qu'un seul élément. Donner sa complexité.

```
void cycle_remove(struct cycle *self);
```

Question 39.5 Écrire une fonction qui recherche un élément et le place en élément courant s'il est présent. La fonction renvoie un booléen qui indique si l'élément est présent. Donner sa complexité.

```
bool cycle_search(struct cycle *self, int data);
```

Question 39.6 Écrire une fonction qui détruit une liste chaînée cyclique.

```
void cycle_destroy(struct cycle *self);
```

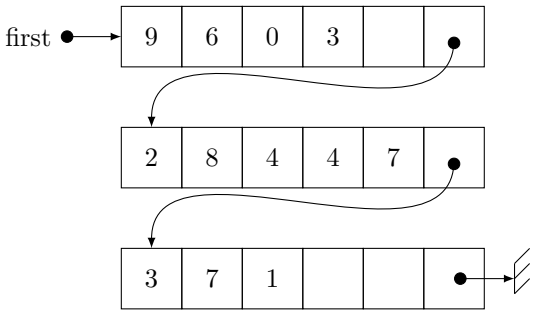


FIGURE 7 – Liste chaînée déroulée

Exercice 40 : Liste chaînée déroulée

Une liste chaînée déroulée *(unrolled linked list)* est une liste chaînée dans laquelle chaque maillon contient un tableau d'élément plutôt qu'un seul élément. De plus, on s'assure que chaque tableau soit rempli au minimum à moitié en permanence (sauf quand la liste contient peu d'éléments). La figure 7 montre une liste chaînée déroulée où chaque maillon contient cinq éléments au maximum. La liste contient 12 éléments en tout.

On définit la structure de données suivante pour implémenter une liste chaînée déroulée :

```
#define CAPACITY 5
#define HALF_CAPACITY (CAPACITY / 2)

struct unode {
  int data[CAPACITY];
  size_t size;
  struct unode *next;
};

struct ulist {
  struct unode *first;
};
```

Question 40.1 Écrire une une fonction qui crée une liste vide. Quelle est sa complexité ?

```
void ulist_create(struct ulist *self);
```

Question 40.2 Écrire une une fonction qui donne le nombre d'éléments d'une liste chaînée déroulée. Quelle est sa complexité ?

```
size_t ulist_size(const struct ulist *self);
```

Question 40.3 Écrire une une fonction qui renvoie l'élément à un indice donné. Quelle est sa complexité ?

```
int ulist_at(const struct ulist *self, size_t index);
```

Question 40.4 Écrire une une fonction qui insère un élément à un indice donné. Si le tableau dans lequel on doit insérer est déjà plein, alors on créera un nouveau maillon et on séparera les éléments en deux de manière à ce que chaque maillon soit rempli au moins à moitié. Quelle est sa complexité ? Justifier.

```
void ulist_insert(struct ulist *self, int data, size_t index);
```

Question 40.5 Écrire une une fonction qui supprime un élément à un indice donné. Si après avoir supprimé l'élément, le maillon est rempli à moins de la moitié, on complètera avec un élément du maillon suivant. Si cette opération ne permet pas que le maillon suivant soit rempli à plus de la moitié, c'est que tous les éléments (ceux du maillon courant et ceux du maillon suivant) peuvent tenir sur le maillon courant et donc, on peut supprimer le maillon suivant. Quelle est sa complexité ? Justifier.

```
void ulist_remove(struct ulist *self, size_t index);
```

Question 40.6 Écrire une fonction qui détruit une liste chaînée déroulée.

```
void ulist_destroy(struct ulist *self);
```

Exercice 41 : Liste chaînée avec cache des nœuds

On considère une liste chaînée dans laquelle on ne va pas libérer les nœuds immédiatement. Les nœuds non-utilisés vont plutôt être conservés dans un cache pour pouvoir les réutiliser en cas de besoin par la suite. On alloue donc un nœud uniquement si le cache est vide. On utilise les structures de données suivantes :

```
struct node {
  int data;
  struct node *next;
};

struct clist { // cached list
  struct node *first;
  struct node *cache;
};
```

Question 41.1 Écrire une fonction qui initalise la liste.

```
void clist_create(struct clist *self);
```

Question 41.2 Écrire une fonction qui teste si la liste est vide.

```
bool clist_is_empty(const struct clist *self);
```

Question 41.3 Écrire une fonction qui ajoute un élément en tête de liste.

```
void clist_add_front(struct clist *self, int data);
```

Question 41.4 Écrire une fonction qui supprime un élément en tête de liste.

```
void clist_del_front(struct clist *self);
```

Question 41.5 Écrire une fonction qui affiche les éléments d'une liste.

```
void clist_display(const struct clist *self);
```

Question 41.6 Écrire une fonction qui détruit la liste, y compris le cache.

```
void clist_destroy(struct clist *self);
```

Exercice 42 : Liste chaînée et tri fusion

On dispose de la structure de données suivante :

```
struct list_node {
  int data;
  struct list_node *next;
}

struct list {
  struct list_node *first;
}
```

Question 42.1 Donner le code d'une fonction en C qui prend en paramètre une liste et qui sépare la liste en deux sous-listes de taille égale (à un élément près). Quelle est sa complexité ?

```
void list_split(struct list *list,
    struct list *res1, struct list *res2);
```

Question 42.2 Donner le code d'une fonction en C qui prend en paramètre deux listes triées (par ordre croissant) et qui renvoie une liste triée en fusionnant les deux listes. Quelle est sa complexité ?

```
void list_merge(struct list *list1, struct list *list2,
    struct list *res);
```

Question 42.3 Donner le code d'une fonction en C qui prend en paramètre une liste et qui renvoie la liste triée. On utilisera le tri fusion et on pourra utiliser les fonctions définies dans les questions précédentes.

```
void list_sort(struct list *list);
```

Question 42.4 Donner la formule pour calculer la complexité $C(n)$ du tri fusion d'une liste de n éléments en nombre de comparaisons en fonction de $C(\frac{n}{2})$.

Question 42.5 Résoudre cette formule grâce au théorème Diviser pour régner.

Exercice 43 : Tours de Hanoï

Le problème des tours de Hanoï est un grand classique illustrant l'usage de la récursivité. On a trois tours A, B et C et n disques numérotés 1 , 2, ..., n. En position initiale, tous les disques sont sur la tour A, chaque disque reposant sur un disque de taille plus grande. À chaque étape, on a le droit de déplacer le disque du haut d'une tour pour le mettre sur une autre tour, à condition qu'un disque ne soit jamais posé sur un disque plus petit.

Le but du jeu est de déplacer les n disques de la tour A vers la tour B comme illustré par la figure 8.

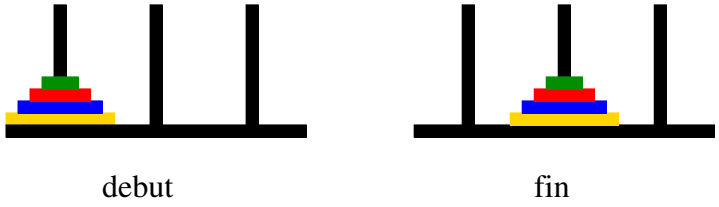


FIGURE 8 – Position de début et de fin pour les tours de Hanoï

Question 43.1 Proposez une solution dans les cas $n = 1, 2, 3, 4$.

Question 43.2 On voit se dessiner une approche récursive. Proposez un algorithme pour ce problème qui va décrire à l'utilisateur les mouvements à effectuer.

Question 43.3 Analysez le nombre de mouvements de disques.

Question 43.4 On considère la structure suivante pour représenter une tour. La structure `struct disc` représente un disque, et en particulier, sa taille (`width`). La structure `struct tower` est alors une pile de disques.

```
struct disc {
  int width;
  struct disc *next;
};
struct tower {
  struct disc *first;
};
```

Coder les fonctions de bases suivantes qui permettent de manipuler la structure.

```
// create an empty tower
void tower_create(struct tower *self);
// push a disc of size k on top of the tower
void tower_push_disc(struct tower *self, int width);
// pop the top disc and return its size
int tower_pop_disc(struct tower *self);
// print the content of the tower
void tower_print(const struct tower *self);
```

Question 43.5 Le jeu comprend trois tours. Proposer un algorithme pour initialiser le jeu avec n disques.

```
struct hanoi {
  struct tower towers[3];
};
void hanoi_create(struct hanoi *self, int n);
```

Question 43.6 Proposer un algorithme qui affiche le jeu.

```
void hanoi_print(const struct hanoi *self);
```

Question 43.7 Proposer un algorithme qui déplace le disque de la tour i vers la tour j.

```
void hanoi_move_one_disc(struct hanoi *self, int i, int j);
```

Question 43.8 Proposer un algorithme qui déplace les n disques de la tour i vers la tour j. La tour k qui n'est ni i, ni j pourra être calculé par la formule $k = 3 - i - j$.

```
void hanoi_move(struct hanoi *self, int n, int i, int j);
```

Question 43.9 Proposer un programme qui prend en paramètres le nombre n de disque et qui retourne une description de la solution.

```
$ hanoi 2

A: 2 1
B:
C:

A: 2
B:
C: 1

A:
B: 2
C: 1

A:
B: 2 1
C:
```

6 Piles et files

Exercice 44 : Implémentation d'une file avec une liste chaînée

Le but de cet exercice est de proposer une implémentation de file à l'aide d'une liste chaînée. On définit le type file de la manière suivante :

```
struct queue_node {
  int value;
  struct queue_node *next;
}

struct queue {
  struct queue_node *first;
  struct queue_node *last;
}
```

Question 44.1 Donner le code d'une fonction qui initialise la file.

```
void queue_create(struct queue *self);
```

Question 44.2 Donner le code d'une fonction qui teste si la file est vide.

```
bool queue_is_empty(const struct queue *self);
```

Question 44.3 Donner le code d'une fonction qui ajoute une valeur à la fin de la file.

```
void queue_enqueue(struct queue *self, int value);
```

Question 44.4 Donner le code d'une fonction qui donne la valeur de la tête d'une file non-vide.

```
int queue_peek(const struct queue *self);
```

Question 44.5 Donner le code d'une fonction qui supprime l'élément de tête d'une file.

```
void queue_dequeue(struct queue *self);
```

Question 44.6 Donner le code d'une fonction qui supprime tous les éléments de la file.

```
void queue_destroy(struct queue *self);
```

Exercice 45 : Implémentation d'une file avec deux listes chaînées

On propose d'implémenter une file en utilisant deux listes simplement chaînées. La première liste `input` contient les éléments qu'on introduit dans la file. La seconde liste `output` contient les éléments qu'on retire de la file. Si la liste `output` est vide, alors, on dépile tous les éléments de `input` un par un et on les empile dans la liste `output`.

On définit la structure de données suivante pour implémenter cette file.

```
struct node {
  int data;
  struct node *next;
};

struct queue {
  struct node *input;
  struct node *output;
};
```

Question 45.1 Qu'est-ce qu'une file ?

Question 45.2 Écrire une fonction qui crée une file vide.

```
void queue_create(struct queue *self);
```

Question 45.3 Écrire une fonction qui teste si une file est vide.

```
bool queue_is_empty(const struct queue *self);
```

Question 45.4 Écrire une fonction qui introduit un élément dans la file.

```
void queue_enqueue(struct queue *self, int data);
```

Question 45.5 Écrire une fonction qui transfert les éléments de `input` dans `output` tel qu'indiqué précédemment. Quelle est la complexité de cette fonction ?

```
void queue_transfer(struct queue *self);
```

Question 45.6 Écrire une fonction qui supprime l'élément de tête d'une file non-vide.

```
void queue_dequeue(struct queue *self);
```

Question 45.7 On suppose que la liste `input` contient n éléments, et que la liste `output` n'en contient aucun. Quel est la complexité du premier appel à la fonction précédente ? Quel est la complexité des appels suivants ? Combien peut-on faire d'appel avec cette complexité ? Au final, quelle est la complexité de la fonction précédente ?

Question 45.8 Écrire une une fonction qui donne la valeur de tête d'une file non-vide.

```
int queue_peek(const struct queue *self);
```

Question 45.9 Écrire une fonction qui détruit une file.

```
void queue_destroy(struct queue *self);
```

FIGURE 9 – Liste doublement chaînée

Exercice 46 : Implémentation d'une file avec une liste doublement chaînée

On propose d'implémenter une file à l'aide d'une liste doublement chaînée. On définit la structure de données suivante pour implémenter cette file.

```
struct node {
  int data;
  struct node *next;
  struct node *prev;
};

struct queue {
  struct node *first;
  struct node *last;
};
```

La figure 9 présente une liste doublement chaînée avec trois éléments.

Question 46.1 Qu'est-ce qu'une file ?

Question 46.2 Écrire une fonction qui crée une file vide.

```
void queue_create(struct queue *self);
```

Question 46.3 Écrire une fonction qui teste si une file est vide.

```
bool queue_is_empty(const struct queue *self);
```

Question 46.4 Écrire une fonction qui introduit un élément dans la file.

```
void queue_enqueue(struct queue *self, int data);
```

Question 46.5 Écrire une fonction qui retire un élément de la file.

```
int queue_dequeue(struct queue *self);
```

Question 46.6 Écrire une fonction qui détruit une file.

```
void queue_destroy(struct queue *self);
```

Exercice 47 : Implémentation d'une pile avec une liste chaînée

Le but de cet exercice est de proposer une implémentation d'une pile grâce à une liste chaînée.

On dispose de la structure de donnée suivante :

```
struct stack_node {
  int data;
  struct stack_node *next;
};

struct stack {
  struct stack_node *first;
};
```

Question 47.1 Donner le code d'une fonction qui initialise une pile vide :

```
void stack_create(struct stack *self);
```

Question 47.2 Donner le code d'une fonction qui examine si la pile est vide :

```
bool stack_is_empty(const struct stack *self);
```

Question 47.3 Donner le code d'une fonction qui empile un élément :

```
void stack_push(struct stack *self, int data);
```

Question 47.4 Donner le code d'une fonction qui donne la valeur du premier élément d'une pile non-vide :

```
int stack_top(const struct stack *self);
```

Question 47.5 Donner le code d'une fonction qui dépile un élément :

```
void stack_pop(struct stack *self);
```

Question 47.6 Donner le code d'une fonction qui détruit une pile :

```
void stack_destroy(struct stack *self);
```

Exercice 48 : Implémentation d'une file à double entrée

Le but de cet exercice est de proposer une implémentation de file à double entrée à l'aide d'un buffer circulaire. On définit le type file de la manière suivante :

```
struct deque {
  size_t capacity;
  size_t start;
  size_t end;
  int *data;
};
```

Les indices `start` et `end` indique le début et la fin de la file. S'ils sont égaux, c'est que la file est vide. On prendra garde pour chacune des fonctions suivantes à traiter les deux cas où `start` est plus petit que `end` et vice-versa. On traitera également les cas particuliers où `start` et `end` sont à la limite du tableau. Enfin, on fera grossir le tableau dynamiquement au besoin, en faisant attention lors de la copie de l'ancien vers le nouveau tableau.

Question 48.1 Donner le code d'une fonction qui initialise la file.

```
void deque_create(struct deque *self);
```

Question 48.2 Donner le code d'une fonction qui teste si la file est vide.

```
bool deque_is_empty(const struct deque *self);
```

Question 48.3 Donner le code d'une fonction qui ajoute une valeur au début de la file.

```
void deque_push(struct deque *self, int value);
```

Question 48.4 Donner le code d'une fonction qui ajoute une valeur à la fin de la file.

```
void deque_inject(struct deque *self, int value);
```

Question 48.5 Donner le code d'une fonction qui supprime une valeur au début de la file.

```
void deque_pop(struct deque *self);
```

Question 48.6 Donner le code d'une fonction qui supprime une valeur à la fin de la file.

```
void deque_eject(struct deque *self);
```

Question 48.7 Donner le code de fonctions qui renvoient la valeur en début et en fin de file.

```
int deque_front(const struct deque *self);
int deque_back(const struct deque *self);
```

7 Structures linéaires et tri

Exercice 49 : Tri par base

Le tri par base *(radix sort)* est un tri fondé sur l'utilisation d'une base. Chaque élément est décomposé suivant cette base et on tri alors le tableau selon un ordre lexicographique relatif à cette base. Par exemple, pour des entiers, on peut prendre comme base l'entier 10, et donc, les différent éléments d'un entier sont ses chiffres en base 10. La figure 10 montre un exemple de tri par base sur des entiers.

435	656	578	428	432	671	443	568

Après le tri des unités :

671	432	443	435	656	578	428	568

Après le tri des dizaines :

428	432	435	443	656	568	671	578

Après le tri des centaines :

428	432	435	443	568	578	656	671

FIGURE 10 – Exemple du tri d'un tableau d'entiers avec le tri par base

Question 49.1 Implémenter un tri par base sur des entiers avec 10 comme base.

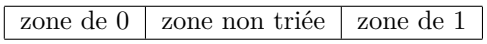
```
void radix_sort(int *data, size_t n);
```

Question 49.2 Quelle est la complexité du tri par base ?

Question 49.3 Ce tri est-il stable ?

Exercice 50 : Tri d'un tableau binaire

On considère un tableau dont les éléments appartiennent à l'ensemble $\{0, 1\}$. On se propose de trier ce tableau. À chaque étape du tri, le tableau est constitué de trois zones consécutives, la première ne contenant que des 0, la seconde n'étant pas triée et la dernière ne contenant que des 1.



On range le premier élément de la zone non triée si celle-ci n'est pas encore vide : si l'élément vaut 0, il ne bouge pas ; si l'élément vaut 1, il est échangé avec le dernier élément de la zone non triée. Dans tous les cas, la longueur de la zone non triée diminue de 1.

Question 50.1 Créer un tableau de 100000 éléments de 0 et de 1 tirés au hasard.

Question 50.2 Compter le nombre de 1 du tableau.

Question 50.3 Trier le tableau selon la méthode indiquée.

Question 50.4 Vérifier que le nombre de 1 est identique au nombre de 1 précédemment calculé.

Question 50.5 Quel est la complexité de cet algorithme ?

Exercice 51 : Algorithmes de sélection

Un algorithme de sélection est une méthode ayant pour but de trouver le k-ième plus petit élément d'un tableau de n objets. On considère ici que le tableau contient des entiers tous distincts.

Question 51.1 Écrire une fonction pour le cas particulier $k = 1$. Quel est sa complexité ? Comment aurait pu s'appeler cette fonction ?

```
int select_1(const int *data, size_t n);
```

Question 51.2 On suppose que le tableau est trié. Écrire une fonction pour le cas général. Quel est sa complexité ?

```
int select_sorted(const int *data, size_t n, size_t k);
```

Question 51.3 Si le tableau n'est pas trié, on peut le trier auparavant. Écrire une fonction pour le cas général (on peut utiliser la fonction précédente). On supposera qu'on dispose d'un algorithme de tri. Quel est la complexité ? Justifier.

```
int select_unsorted(int *data, size_t n, size_t k);
```

—

Nous allons maintenant réaliser un algorithme de sélection avec une complexité moyenne optimale. L'idée de cet algorithme est de choisir un pivot, de partitionner le tableau en fonction du pivot et de regarder si le pivot est en k-ième position. Si c'est le cas, on arrête, sinon, on recommence récursivement avec une des deux parties en fonction de la position du pivot.

Question 51.4 Écrire une fonction qui échange deux éléments d'un tableau.

```
void swap(int *data, size_t i, size_t j);
```

Question 51.5 Écrire une fonction qui partitionne un tableau entre les indices i et j inclus.

```
size_t partition(int *data, size_t i, size_t j);
```

Question 51.6 Écrire une fonction qui réalise l'algorithme décrit précédemment.

```
int select(int *data, size_t n, size_t k);
```

On pourra écrire une fonction intermédiaire qui fera les appels récursifs.

```
int select_partial(int *data, size_t i, size_t j, size_t k);
```

Question 51.7 Si on appelle p le nombre d'éléments plus petits que le pivot, donner la formule de complexité $\mathcal{C}(n)$ en fonction de n et p.

Question 51.8 On considère le pire cas. Que vaut p ? Quelle est alors la formule de complexité ? Résoudre la formule en justifiant clairement.

Question 51.9 On considère le cas en moyenne. Que vaut p ? Quelle est alors la formule de complexité ? Résoudre la formule avec le théorème Diviser Pour Régner en justifiant clairement.

Exercice 52 : Différences entre les éléments distincts d'un tableau

On note $u_1, \ldots, u_n$, les éléments d'un tableau, $n \geq 2$. On considère l'ensemble des différences absolues entre des éléments distincts du tableau :

$$\mathcal{D} = \{|u_i - u_j|, i \neq j\}$$

Question 52.1 Proposer un algorithme efficace (en nombre de comparaisons) prenant un *tableau trié* et déterminant le maximum de l'ensemble $\mathcal{D}$. Par exemple, si le tableau est `[1, 3, 5 ,8]`, le maximum est 7. Il est obtenu pour $u_1 = 1$ et $u_4 = 8$. On justifiera que l'algorithme est efficace en indiquant sa complexité en nombre de comparaisons.

```
int sorted_array_max_diff(int *data, size_t n);
```

Question 52.2 Proposer un algorithme efficace (en nombre de comparaisons) prenant un *tableau non-trié* et déterminant le maximum de l'ensemble $\mathcal{D}$. On justifiera que l'algorithme est efficace en indiquant sa complexité en nombre de comparaisons.

```
int array_max_diff(int *data, size_t n);
```

Question 52.3 Proposer un algorithme efficace (en nombre de comparaisons) prenant un *tableau trié* et déterminant le minimum de l'ensemble $\mathcal{D}$. Par exemple, si le tableau est `[1, 3, 5 ,8]`, le minimum est 2. Il est obtenu pour $u_2 = 3$ et $u_3 = 5$ (entre autres). On justifiera que l'algorithme est efficace en indiquant sa complexité en nombre de comparaisons.

```
int sorted_array_min_diff(int *data, size_t n);
```

Question 52.4 Proposer un algorithme efficace (en nombre de comparaisons) prenant un *tableau non-trié* et déterminant le minimum de l'ensemble $\mathcal{D}$. On justifiera que l'algorithme est efficace en indiquant sa complexité en nombre de comparaisons.

```
int array_min_diff(int *data, size_t n);
```

Exercice 53 : Union de segments

Dans cet exercice, on propose un algorithme efficace pour calculer l'union de segments de la forme $[a, b[$. Par exemple :

$$[1, 3[\cup[2, 5[\cup[7, 8[= [1, 5[\cup[7, 8[$$

On utilise les structures de données suivantes :

```
struct segment {
  int a;
  int b;
}

struct segment_set {
  size_t size;
  size_t capacity;
  struct segment *data;
}
```

On procède en deux étapes :

1. on trie les segments ;
2. on calcul l'union sur les segments triés.

Question 53.1 Donner le code d'une fonction qui compare deux segments $S_1 = [a_1, b_1[$ et $S_2 = [a_2, b_2[$ selon un ordre lexicographique (c'est-à-dire l'ordre du dictionnaire).

```
int segment_compare(const segment *s1, const segment *s2);
```

Question 53.2 Donner le nom d'un algorithme de tri optimal. Donner en quelques lignes son fonctionnement. Quel est sa complexité ?

Question 53.3 À partir des segments triés, proposer un algorithme pour calculer l'union des segments. Quelle est la complexité de cette étape ?

```
void sorted_segment_set_union(const segment_set *self,
    segment_set *res);
```

On pourra supposer qu'on dispose de la fonction suivante qui ajoute un segment dans un ensemble de segment :

```
void segment_set_add(const segment_set *self, int a, int b);
```

Question 53.4 Quelle est la complexité globale de cet algorithme ?

Exercice 54 : Vecteur creux

Un vecteur est un élément de $\mathbb{R}^k$, c'est-à-dire un k-uplet de coordonnées. Chaque coordonnées est représentée à l'aide d'un `double`. Il y a deux manières de représenter un vecteur :

— si le vecteur a toutes ses coordonnées définies, on l'appelle un *vecteur dense*, et on le représente à l'aide d'un tableau de taille k ;

— si le vecteur a seulement quelques coordonnées non-nulles, on l'appelle un *vecteur creux*, et on ne représente alors que les coordonnées du vecteur qui sont non-nulles.

Par exemple, pour $k = 10$, le vecteur dense $[0, 0, 3.2, 0, 0, 4.5, 0, 0, 7.9, 6.1]$ peut être représenté par :

$$[(6, 4.5), (10, 6.1), (3, 3.2), (9, 7.9)]$$

Dans cet exercice, nous allons évaluer plusieurs manières de représenter un vecteur creux de $\mathbb{R}^k$ (où k est une constante très grande) avec un tableau dynamique.

Nous utiliserons les structures de données suivantes :

```
// coordonnée d'un vecteur creux
struct coord {
  size_t index;    // indice dans le k-uplet dans [1, k]
  double value; // valeur de la coordonnée
}

// tableau dynamique de coordonnées
struct vector {
  size_t capacity;    // nombre d'éléments maximum du tableau
  size_t size;        // nombre d'éléments du tableau
  struct coord *data; // tableau
}
```

Pour que le tableau dynamique représente un vecteur creux, on a deux contraintes :

1. chaque indice n'est présent qu'en un seul exemplaire dans le tableau ;
2. toutes les valeurs de coordonnées sont non-nulles.

Dans tout cet exercice, on va s'intéresser à trois opérations :

— `vector_access`, l'accès à une valeur suivant son indice ;

— `vector_set`, la définition d'une nouvelle valeur à un indice donné ;

— `vector_add`, l'addition de deux vecteurs creux.

Les prototypes de ces fonctions sont :

```
double vector_access(const struct vector *self, size_t index);
void vector_set(struct vector *self, size_t index, double value);
void vector_add(const struct vector *v1, const struct vector *v2,
    struct vector *result);
```

On suppose tout d'abord que le tableau dynamique n'est pas trié par ordre d'indice.

Question 54.1 Donner une implémentation en C de `vector_access`. Quelle est sa complexité ?

Question 54.2 Quelle est la complexité de `vector_set` si on sait qu'`index` n'existe pas dans le tableau ? On donnera une réponse très précise et justifiée.

Question 54.3 Quelle est la complexité de `vector_set` si on ne sait pas qu'`index` existe dans le tableau ? On donnera une réponse très précise et justifiée.

Question 54.4 Dans la fonction `vector_add`, quelle taille peut avoir le tableau `result->data` au maximum en fonction de `v1->size` et `v2->size` ? Justifier.

Question 54.5 Donner une idée de l'algorithme pour `vector_add`. Quelle est sa complexité ?

On suppose désormais que le tableau dynamique est trié par ordre croissant d'indice.

Question 54.6 Quel algorithme utilise-t-on pour `vector_access` ? Quelle est sa complexité ?

Question 54.7 Donner une idée de l'algorithme pour `vector_set` ? Quelle est sa complexité ?

Question 54.8 Donner une implémentation complète en C de `vector_add`. On prendra garde à la gestion de la mémoire.

Question 54.9 Quelle est la complexité de la fonction que vous venez d'implémenter ?

Question 54.10 Quelle amélioration peut-on proposer pour l'algorithme de la question 5 ? Quelle est alors la complexité de `vector_add` avec cette amélioration ?

Exercice 55 : Répertoire téléphonique

On veut stocker un répertoire téléphonique dans un tableau dynamique. On se pose la question de savoir s'il vaut mieux trier le tableau ou pas. Pour cela, on va examiner deux implémentations : la première où les entrées du répertoire ne sont pas triées ; la seconde où les entrées de répertoire sont triées. Dans toute la suite, on suppose qu'il peut y avoir plusieurs numéros pour le même nom et dans ce cas, il y a plusieurs entrées.

On imagine qu'on dispose des structures de données suivantes :

```
struct entry {
  const char *name;
  const char *phone;
};

struct phonebook {
  struct entry *data;
  size_t size;
  size_t capacity;
};
```

On examine trois fonctions :
- Insertion d'une nouvelle entrée :
  `phonebook_insert`
- Recherche d'une entrée :
  `phonebook_search`
- Créaton d'un répertoire à partir d'un fichier :
  `phonebook_create_from_file`

Pour la fonction de création, on suppose que toutes les entrées peuvent être stockées dans un fichier et qu'on peut lire le fichier pour récupérer les différentes entrées.

On suppose tout d'abord que les entrées ne sont pas triées dans le tableau.

Question 55.1 Quelle est la complexité de `phonebook_insert` ? Justifier.

Question 55.2 Pour la fonction `phonebook_search`, décrire brièvement l'algorithme utilisé. Quel est sa complexité ?

Question 55.3 Pour la fonction `phonebook_create_from_file`, décrire brièvement l'algorithme utilisé. Quel est sa complexité ?

—

On suppose désormais que les entrées sont triées.

Question 55.4 Pour la fonction `phonebook_search`, quel algorithme est utilisé ? Quel est sa complexité ?

Question 55.5 Quelle est la complexité de `phonebook_insert` ? Justifier précisément.

Question 55.6 Pour la fonction `phonebook_create_from_file`, si on utilise la fonction `phonebook_insert`, décrire brièvement l'algorithme. Quelle est alors la complexité ?

Question 55.7 Pour la fonction `phonebook_create_from_file`, on décide tout d'abord d'insérer les entrées sans les trier (étape 1), puis d'appliquer un algorithme de tri (étape 2). Quelle est la complexité de l'étape 1 ? Donner le nom d'un algorithme de tri optimal, décrire son fonctionnement. Quelle est la complexité de l'étape 2 ? Quelle est alors la complexité de la fonction ?

—

Question 55.8 Argumenter pour savoir quelle est la meilleure implémentation.

Exercice 56 : Résultat d'examen

On considère qu'on dispose d'un tableau contenant des résultats d'examen. Chaque case du tableau contient une entrée constituée d'un nom, sous forme d'une chaine de caractère, et d'une note, sous forme d'un réel entre 0 et 20. On utilise la structure de données suivante pour représenter nos résultats d'examen.

```
struct entry {
  char *name;
  double mark;
};

struct results {
  struct entry *entries;
  size_t size;
  size_t capacity;
};
```

On s'intéresse à deux fonctions :
- la fonction `results_get_mark()` qui donne la note d'un étudiant à partir de son nom ;
- la fonction `results_get_rank()` qui renvoie le rang de l'étudiant parmi tous les résultats à partir de son nom.

On s'intéresse tout d'abord à `results_get_mark()`.

Question 56.1 On suppose que le tableau n'est pas trié. Donner une idée de l'algorithme à utiliser. On précisera en particulier la fonction utilisée pour effectuer les comparaisons. Quel est sa complexité ?

Question 56.2 Dans le cas où le tableau n'est pas trié, on veut ajouter un étudiant et sa note. Donner une idée de l'algorithme à utiliser. Quel est sa complexité ?

Question 56.3 On suppose à partir de maintenant que le tableau est trié. Quel algorithme est utilisé pour obtenir la note de l'étudiant ? Donner une idée de son fonctionnement. Quel est sa complexité ?

Question 56.4 On veut ajouter un étudiant et sa note. Donner une idée de l'algorithme à utiliser. Quel est sa complexité ?

Question 56.5 On veut ajouter n étudiants et leur note à un tableau initialement vide. Si on utilise la fonction précédente, quelle est la complexité de la fonction ? On peut faire mieux. Comment et quelle est alors la complexité ?

—

On s'intéresse désormais à `results_get_rank()`.

Question 56.6 On trie le tableau par note croissante. Donner le nom d'un algorithme de tri par comparaison optimal. Donner une idée de son fonctionnement. Quel est sa complexité ?

Question 56.7 On suppose que le tableau est trié par note croissante. Donner une idée du fonctionnement de la fonction `results_get_rank()`. Quel est sa complexité ?

Question 56.8 On suppose que le tableau n'est pas trié par note croissante. Décrire la fonction `results_get_rank()`. Quel est sa complexité ? Améliore-t-on l'algorithme si le tableau est trié par nom ? Justifier.

Exercice 57 : Table de hachage

Une table de hachage est une structure de données qui permet de représenter un ensemble. On supposera ici que les éléments de l'ensemble sont des chaînes de caractères. Une table de hachage est un tableau de k listes chaînées. Pour insérer un élément dans l'ensemble, on utilise une fonction de hachage `hash` qui prend un élément et renvoie un entier h, l'élément est alors inséré dans la liste chaînée qui se trouve à l'indice $h\%k$.

La figure 11 montre un exemple de table de hachage avec trois éléments. Deux de ces éléments ont le même indice et sont donc dans la même liste chaînée, on parle alors de *collision*. On essaie de choisir une fonction de hachage qui permet d'éviter les collisions.

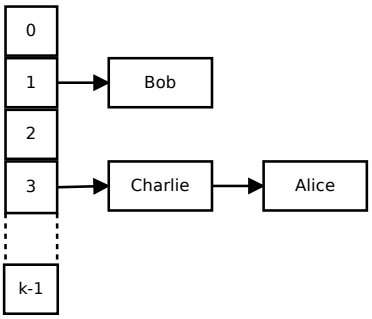


FIGURE 11 – Une table de hachage

On appelle *facteur de compression* le rapport $\frac{n}{k}$, c'est-à-dire le nombre d'éléments de la table divisé par le nombre de cases du tableau.

Question 57.1 Quelle est l'autre structure de données qui permet de représenter un ensemble ?

Question 57.2 On suppose que la fonction de hachage est suffisamment bonne et que toutes les listes chaînées ont la même taille. Quelle est la taille moyenne d'une liste chaînée ?

Question 57.3 Donner l'algorithme en français en deux lignes pour rechercher un élément e dans la table. Avec les mêmes hypothèses que la question précédente, quelle est alors la complexité d'une recherche d'un élément dans la table ? On précisera la ou les opérations élémentaires.

Question 57.4 Donner l'algorithme en français en trois lignes pour insérer un élément e dans la table (en veillant à ce qu'il n'y soit pas déjà). Avec les mêmes hypothèses que la question précédente, quelle est alors la complexité d'une insertion d'un élément dans la table ? On précisera la ou les opérations élémentaires.

Question 57.5 On suppose que le facteur de compression est borné par une constante. Quelles sont alors les complexités de la recherche et de l'insertion ?

Question 57.6 Comparer ces complexités avec les complexités moyennes pour la structure de données de la question 1.

Question 57.7 On suppose désormais que la fonction de hachage est plutôt mauvaise, c'est-à-dire qu'elle donne toujours le même indice. Quelle est alors la complexité au pire cas de la recherche et de l'insertion ?

Question 57.8 Comparer ces complexités avec les complexités en pire cas pour la structure de données de la question 1.

Question 57.9 Pour une fonction de hachage fixée, que proposeriez-vous de changer pour améliorer la complexité du pire cas ?

8 Matrices

Exercice 58 : Jeu de la vie

Le jeu de la vie a été imaginé par John Horton Conway en 1970. Le jeu se déroule sur une grille à deux dimensions, théoriquement infinie (mais de longueur et de largeur finies et plus ou moins grandes dans la pratique), dont les cases peuvent prendre deux états distincts : «vivantes» ou «mortes». À chaque étape, l'évolution d'une case est entièrement déterminée par l'état de ses huit voisines de la façon suivante :

— Une case morte possédant exactement trois voisines vivantes devient vivante (elle naît).
— Une case vivante possédant deux ou trois voisines vivantes le reste, sinon elle meurt.

Question 58.1 Proposer une structure de données pour représenter une grille de 1000×1000.

Question 58.2 Proposer une fonction qui prend en paramètre deux grilles et qui calcule une itération du jeu de la vie.

Exercice 59 : Problème des n reines

Le but du problème des n reines est de placer n reines d'un jeu d'échecs sur un plateau de $n \times n$ cases sans que les reines ne puissent se menacer mutuellement, conformément aux règles du jeu d'échecs (la couleur des pièces étant ignorée). Par conséquent, deux reines ne devraient jamais partager la même rangée, colonne, ou diagonale. On suppose que n est une constante.

```
#define N 8
```

Question 59.1 On fait une recherche exhaustive de tous les placements possibles des n reines où deux reines peuvent éventuellement tomber sur la même case. Quelle est la complexité ?

Question 59.2 Quelle structure peut-on utiliser pour représenter un plateau ?

Question 59.3 Écrire un algorithme qui résout le problème des n reines avec la structure définie précédemment. Indication : on pourra placer une reine dans la colonne 0, puis, placer une reine dans la colonne 1 qui ne soit pas en conflit avec la reine précédemment placée, etc. Il est conseillé de faire un algorithme récursif.

Question 59.4 Donner une formule pour calculer la complexité de l'algorithme précédent. Quel est alors la complexité ?

Question 59.5 Soit une reine placée sur la ligne l_1 et la colonne c_1. Soit une autre reine placée sur la ligne l_2 et la colonne c_2. Quelle relation lie l_1, c_1, l_2, c_2 si la seconde reine est placée sur la même diagonale ($\backslash$) que la première reine ? Même question si la seconde reine est placée sur la même anti-diagonale ($/$) que la première reine.

Question 59.6 Combien y a-t-il de diagonales (ou d'anti-diagonales) sur un plateau de $n \times n$ cases ?

Question 59.7 Grâce aux observations précédentes, écrire un nouvel algorithme qui résout le problème des n reines. Quel est sa complexité ?

Exercice 60 : Carré magique

Un carré magique d'ordre n est composé de n^2 entiers strictement positifs, écrits sous la forme d'un tableau carré. Ces nombres sont disposés de sorte que leurs sommes sur chaque rangée, sur chaque colonne et sur chaque diagonale principale soient égales. On nomme alors *constante magique* la valeur de ces sommes. Un carré magique normal est un cas particulier de carré magique, constitué de tous les nombres entiers de 1 à n^2, où n est l'ordre du carré.

Dans cet exercice, on souhaite représenter un carré magique et vérifier qu'il est bien magique. Il est fortement recommandé d'utiliser des fonctions annexes dont vous fournirez l'implémentation.

Question 60.1 Dans le cas d'un carré magique normal, que vaut la constante magique C en fonction de n ? Justifier.

Question 60.2 Proposer une structure de données `struct square` pour représenter un tableau carré d'ordre n.

Question 60.3 Écrire une fonction qui récupère l'élément situé à la ligne `row` et à la colonne `col`, les deux étants des nombres compris entre 0 et $n - 1$.

```
int square_get(const struct square *self, size_t row, size_t col);
```

Question 60.4 Écrire une fonction qui vérifie qu'un tableau carré d'ordre n est un carré magique normal. Quelle est la complexité de cette fonction en fonction de n ?

```
bool is_normal_magic(const struct square *self);
```

Exercice 61 : Traitement d'image

En traitement d'image, une technique consiste à appliquer un *noyau* à chaque pixel de l'image. Un noyau est une matrice $k \times k$ où k est une petite constante. Dans la suite, on supposera $k = 3$.

Soit K le noyau suivant :

$$K = \begin{pmatrix} k_{0,0} & k_{0,1} & k_{0,2} \\ k_{1,0} & k_{1,1} & k_{1,2} \\ k_{2,0} & k_{2,1} & k_{2,2} \end{pmatrix}$$

L'application du noyau K sur le pixel $p_{i,j}$ (avec i la ligne du pixel et j la colonne du pixel) est :

$$q_{i,j} = \sum_{u=-1}^{1} \sum_{v=-1}^{1} p_{i+u,j+v} * k_{1+u,1+v}$$

On définit les structures de données suivantes pour représenter une image et un noyau. On suppose que le noyau et l'image sont stockés en *row-major*, c'est-à-dire ligne par ligne. On suppose que l'image est indicée à partir de 0 pour les lignes et les colonnes.

```
struct image { // grayscale image
  uint8_t *data;
  int width;
  int height;
};

struct kernel {
  double coeff[3][3];
};
```

Question 61.1 Écrire une fonction qui donne la valeur du pixel à la ligne i et la colonne j. Si (i, j) n'est pas dans l'image, on renverra la valeur du pixel le plus proche.

```
uint8_t image_get(const struct image *self, int i, int j);
```

Question 61.2 Écrire une fonction qui calcule l'application d'un noyau à un pixel. Quelle est sa complexité ?

```
uint8_t image_apply_kernel(const struct image *self,
    int i, int j, const struct kernel *ker);
```

Question 61.3 Écrire une fonction qui calcule un traitement d'image en utilisant un noyau. Quelle est sa complexité ?

```
void image_compute(struct image *out, const struct image *in,
    const struct kernel *ker);
```

Question 61.4 On suppose qu'on travaille avec le noyau $K = \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$ sur l'image $I = \begin{pmatrix} 20 & 10 \\ 25 & 15 \end{pmatrix}$. Calculer l'image J résultante de l'application du noyau K à l'image I en détaillant vos calculs.

$$\begin{pmatrix} 10 & 20 & 30 & 40 \\ 15 & 25 & 35 & 45 \\ 27 & 29 & 37 & 48 \\ 32 & 33 & 39 & 50 \end{pmatrix}$$

FIGURE 12 – Un example de matrice ordonnée (4×4)

Exercice 62 : Recherche d'un élément dans une matrice ordonnée

Une matrice ordonnée ($n \times n$) est une matrice dans laquelle chaque ligne $i \in [0, n-1]$ et chaque colonne $j \in [0, n-1]$ est triée par ordre croisant. La figure 12 montre un exemple de matrice ordonnée. Le but de cet exercice est de chercher un élément x dans une matrice ordonnée et de renvoyer ses coordonnées (i, j) si l'élément est dans la matrice.

On dispose de la structure suivante :

```
struct coord {
  int i;
  int j;
};
```

Question 62.1 Proposer une structure de données `struct mat` pour représenter une matrice ordonnée d'entiers (`int`). On appellera `n` le nombre de lignes et de colonnes. Justifier votre choix.

Question 62.2 Écrire une fonction qui permet de récupérer l'élément aux coordonnées (i, j).

```
int mat_get(const struct mat *self, int i, int j);
```

—

Dans un premier temps, on ne va pas prendre en compte le fait que la matrice est ordonnée.

Question 62.3 Écrire une fonction qui recherche naïvement dans la matrice un élément x. La fonction renverra `true` si l'élément a été trouvé et dans ce cas, `res` contiendra les coordonnées de l'élément trouvé ; et `false` sinon.

```
bool mat_search_1(const struct mat *self, int x, struct coord *res);
```

Question 62.4 Quelle est la complexité $\mathcal{C}_1(n)$ de cette fonction ?

—

On prend désormais en compte le fait que la matrice est ordonnée. En particulier, chaque ligne est ordonnée.

Question 62.5 Quel algorithme utiliser pour trouver si l'élément recherché se trouve sur une ligne i donnée ? Expliquer son principe. Quel est sa complexité ?

Question 62.6 Si on utilise cet algorithme sur toutes les lignes jusqu'à ce qu'on trouve l'élement, quelle est la complexité globale $\mathcal{C}_2(n)$ de la recherche ?

—

On propose maintenant d'utiliser un algorithme de type Diviser Pour Régner. Pour cela, on examine l'élément au milieu de la matrice.

Question 62.7 Si l'élément au milieu de la matrice de coordonnées (i, j) est strictement plus petit que l'élément recherché x, que peut-on en déduire sur les coordonnées (i_x, j_x) de l'élément recherché ? Même question si l'élément au milieu de la matrice est strictement plus grand que x. On pourra faire un schéma pour justifier la réponse.

Question 62.8 Écrire une fonction qui utilise le principe précédent.

```
bool mat_search_3(const struct mat *self, int x, struct coord *res);
```

On pourra d'abord écrire une fonction qui fait une recherche sur une sous-matrice avec $i \in [i_{\min}, i_{\max}]$ et $j \in [j_{\min}, j_{\max}]$.

```
bool mat_search_partial(const struct mat *self, int x,
      int imin, int imax, int jmin, int jmax, struct coord *res);
```

Question 62.9 Donner la formule pour calculer $\mathcal{C}_3(n)$, la complexité pour une matrice de taille $(n \times n)$, en fonction de $\mathcal{C}_3(n/2)$.

Question 62.10 Résoudre cette formule grâce au théorème Diviser Pour Régner en justifiant clairement.

—

On propose ici d'utiliser un algorithme astucieux pour rechercher un élément dans une matrice ordonnée. L'idée est d'éliminer à chaque fois une ligne ou une colonne de la recherche. Pour cela, on commence en haut à droite, c'est-à-dire aux coordonnées $(0, n-1)$.

Question 62.11 Si l'élément considéré est strictement plus petit que x, que peut-on éliminer ? De même, si l'élément est strictement plus grand que x, que peut-on éliminer ? Justifier.

Question 62.12 Écrire une fonction qui utilise le principe précédent.

```
bool mat_search_4(const struct mat *self, int x, struct coord *res);
```

Question 62.13 Quelle est sa complexité $\mathcal{C}_4(n)$? Justifier.

—

Question 62.14 Ordonner asymptotiquement $\mathcal{C}_1(n)$, $\mathcal{C}_2(n)$, $\mathcal{C}_3(n)$ et $\mathcal{C}_4(n)$.

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

FIGURE 13 – Un sudoku complété

Exercice 63 : Vérification d'un sudoku

Le sudoku est un jeu qui consiste à remplir une grille de 9×9 avec une série de chiffres tous différents, qui ne se trouvent jamais plus d'une fois sur une même ligne, dans une même colonne ou dans une même région de 3×3. Le but de cet exercice est de vérifier qu'une grille de sudoku a été correctement remplie et qu'elle satisfait toutes les conditions décrites précédemment. La figure 13 montre un sudoku complété correctement.

On indiquera une ligne par un indice $i \in [0, 8]$ (de gauche à droite) et une colone par un indice $j \in [0, 8]$ (de haut en bas).

Question 63.1 Proposer une structure de données `struct sudoku` pour représenter une grille de sudoku.

Question 63.2 Écrire une fonction qui permet de récupérer l'élément aux coordonnées (i, j).

```
int sudoku_get(const struct sudoku *self, int i, int j);
```

Question 63.3 Écrire une fonction qui vérifie qu'une ligne est correcte. De même pour vérifier qu'une colonne est correcte.

```
bool sudoku_verify_row(const struct sudoku *self, int i);
bool sudoku_verify_col(const struct sudoku *self, int j);
```

Question 63.4 Écrire une fonction qui vérifie qu'une région dont le carré en haut à gauche est donné par les coordonnées (i, j) est correcte.

```
bool sudoku_verify_region(const struct sudoku *self,
                                                int i, int j);
```

Question 63.5 Écrire une fonction qui vérifie qu'un sudoku est correct. Quelle est sa complexité ? Justifier.

```
bool sudoku_verify(const struct sudoku *self);
```

1	1	1	3	3
1	2	2	3	3
1	1	2	3	3
1	1	2	3	3
1	1	2	2	1

FIGURE 14 – Une grille de Kingdomino avec quatre domaines

Exercice 64 : Compter les zones connexes dans une grille (4,5 points)

Le jeu Kingdomino se joue sur une grille 5×5. Dans ce jeu, il est nécessaire de compter les zones connexes, appelés «domaines» et définis de la manière suivante : «groupes de cases connectées horizontalement ou verticalement du même type». Les types sont représentés par des entiers strictement positifs. La figure 14 présente une grille avec quatre domaines.

Question 64.1 Proposer un type `struct kd` pour implémenter la grille du jeu. Justifier le choix.

Question 64.2 Écrire une fonction qui permet de récupérer le type aux coordonnées (i, j) et une fonction qui permet de mettre à jour le type aux coordonnées (i, j).

```
int kd_get(const struct kd *self, int i, int j);
void kd_set(struct kd *self, int i, int j, int value);
```

Question 64.3 Écrire une fonction qui, à partir d'une position (i, j) va explorer récursivement les voisins horizontaux et verticaux du même type et marquer dans un second tableau les voisins visités (avec la convention «déjà visité» = 1 et «pas encore visité» = 0).

```
void kd_visit(const struct kd *self, int i, int j,
                                              struct kd *visited);
```

Question 64.4 Écrire une fonction pour compter le nombre de zones connexes, ou domaines, dans la grille. On pourra considérer qu'on dispose d'une fonction `kd_create` qui initialise une grille à zéro et d'une fonction `kd_destroy` qui nettoie la grille à la fin.

```
int kd_count_domains(const struct kd *self);
```

9 Arbres

Exercice 65 : Arbre d'expression arithmétique

On propose de représenter une expression arithmétique à l'aide de la structure d'arbre suivante :

```
struct expr {
  enum { OP, VAL } type;
  union {
    char op; // '+', '-', '*', '/'
    int val;
  } data;

  struct expr *left;
  struct expr *right;
}
```

Un nœud est soit un opérateur (`type == OP`) et alors, il a deux fils, soit une valeur (`type == VAL`) et alors, il n'a aucun fils.

Question 65.1 Dessiner les arbres correspondant aux expressions suivantes :

1. $(3 + 1) \times 2$
2. $3 + 1 \times 2$
3. $3 + 1 + 2$
4. $(3 \times 4) + (2 \times 4 - 3)/2$

Question 65.2 Proposer un algorithme pour vérifier que la structure d'arbre est cohérente.

```
bool expr_is_well_formed(const struct expr *e);
```

Question 65.3 Proposer un algorithme pour évaluer une expression. Quel type de parcours est utilisé dans cet algorithme ?

```
int expr_eval(const struct expr *e);
```

Question 65.4 Proposer un algorithme qui prend une expression e en entrée et renvoie l'expression $2 \times e$.

```
struct expr *expr_double(struct expr *e);
```

Question 65.5 Proposer un algorithme qui vérifie si deux expressions sont égales. On prendra en compte la commutativité de $+$ et $\times$.

```
bool expr_equals(struct expr *e1, struct expr *e2);
```

Exercice 66 : Expression arithmétique avec une variable

On propose de représenter une expression arithmétique avec une variable x à l'aide de la structure d'arbre suivante :

```
struct expr {
  enum { OP, VAL, VAR } type;
  union {
    char op; // '+', '-', '*', '/'
    double val;
  } data;

  struct expr *lhs;
  struct expr *rhs;
}
```

Un nœud est soit un opérateur (`type == OP`) et alors, il a deux fils, soit une valeur (`type == VAL`) et alors, il n'a aucun fils, soit la variable x (`type == VAR`) et alors, il n'a aucun fils ni aucune donnée. La figure 15 montre un exemple d'arbre avec une variable.

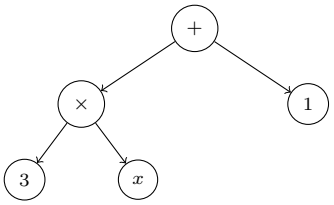


FIGURE 15 – Arbre de l'expression $3x + 1$

Question 66.1 À quelle catégorie d'arbre appartiennent les arbres d'expression ? Expliquer.

Question 66.2 Soit un arbre binaire de taille n et de hauteur h, quelles relations y a-t-il entre n et h ? Dans quels cas y a-t-il égalité dans les relations précédentes ?

Question 66.3 Écrire une fonction qui crée un nœud de type `OP`.

```
struct expr *expr_make_binop(char op,
                             struct expr *lhs, struct expr *rhs);
```

Question 66.4 Écrire une fonction qui duplique un arbre d'expression.

```
struct expr *expr_dup(const struct expr *e);
```

Question 66.5 Écrire une fonction d'évaluation d'un arbre d'expression en assignant une valeur à x.

```
double expr_eval(const struct expr *e, double x);
```

—

Nous allons maintenant essayer de calculer une dérivée formelle d'une expression. La table 1 donne les dérivées usuelles.

$f(x)$	$f'(x)$
λ	0
x	1
$g(x) + h(x)$	$g'(x) + h'(x)$
$g(x) \times h(x)$	$g'(x)h(x) + g(x)h'(x)$
$\frac{g(x)}{h(x)}$	$\frac{g'(x)h(x) - g(x)h'(x)}{h(x)^2}$

TABLE 1 – Rappel sur les dérivées usuelles

On supposera qu'on dispose des fonctions suivantes :

```
struct expr *expr_make_var();
struct expr *expr_make_const(double value);
```

Question 66.6 Écrire une fonction qui calcule l'expression dérivée de l'expression donnée en paramètre.

```
struct expr *expr_derivative(const struct expr *e);
```

Question 66.7 La question précédente peut faire apparaître des expressions triviales. Par exemple, l'expression $3 \times x$ dérivée à l'aide de la multiplication donnera $0 \times x + 3 \times 1$. Écrire une fonction qui simplifie une expression. On prendra en compte les simplifications suivantes : $0 + e = e \pm 0 = e$; $0 \times e = e \times 0 = 0$; $1 \times e = e \times 1 = e$.

```
struct expr *expr_simplify(struct expr *e);
```

Exercice 67 : Arbre de formule logique booléenne

On propose de représenter une formule booléenne avec n variables à l'aide de la structure d'arbre suivante :

```
struct logic {
  enum { OP, VAL, VAR_P, VAR_N } type;
  union {
    char op; // '&', '|'
    bool val;
    size_t var;
  } data;

  struct logic *lhs;
  struct logic *rhs;
}
```

Un nœud est soit un opérateur (`type == OP`) et alors, il a deux fils, soit une valeur (`type == VAL`) et alors, il n'a aucun fils, soit une variable b_i (`type == VAR_P`) ou sa négation $\neg b_i$ (`type == VAR_N`) et alors, il n'a aucun fils et son indice i se situe dans `var`. La figure 16 montre un exemple d'arbre avec trois variables.

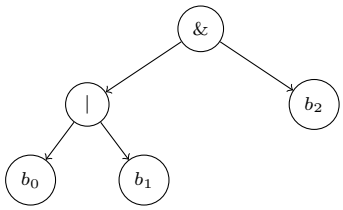


FIGURE 16 – Arbre de la formule $(b_0 \mid b_1)$ & b_2

Question 67.1 Écrire une fonction qui crée un nœud de type `OP`.

```
struct logic *logic_make_binop(char op,
    struct logic *lhs, struct logic *rhs);
```

Question 67.2 Écrire une fonction d'évaluation d'une formule booléenne non-nulle en utilisant un tableau qui indique la valeur de chacune des variables.

```
bool logic_eval(const struct logic *self, bool *values);
```

Question 67.3 Écrire une fonction qui calcule la négation d'une formule booléenne non-nulle.

```
struct logic *logic_negate(const struct logic *self);
```

Exercice 68 : Arbre préfixe

Un arbre préfixe *(trie)* est un arbre qui permet de stocker un grand ensemble de chaînes de caractères de manière compressée. Deux chaînes ayant le même préfixe partagent la même branche pour ce préfixe. La figure 17 montre un arbre préfixe pour des numéros de téléphone à 10 chiffres. Dans la suite, on considère les arbres préfixes qui stockent des numéros de téléphone à 10 chiffres.

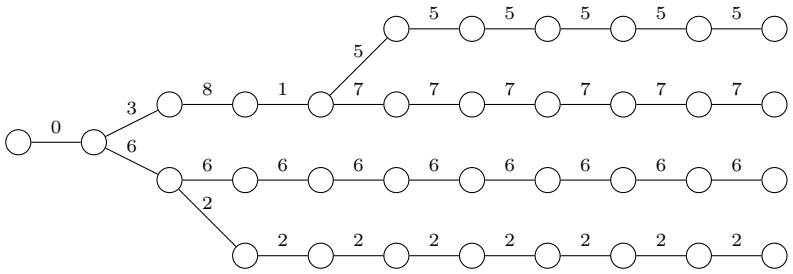


FIGURE 17 – Un arbre préfixe avec les numéros 0381555555, 0381777777, 0666666666, 0622222222

On propose la structure de donnée suivante :

```
struct prefix {
  struct prefix *children[10];
};
```

Question 68.1 Donner un algorithme qui compte le nombre de nœuds d'un arbre préfixe.

```
size_t prefix_count_nodes(const struct prefix *self);
```

Question 68.2 Donner un algorithme qui compte le nombre de numéros de téléphone stockés dans un arbre préfixe.

```
size_t prefix_count_numbers(const struct prefix *self);
```

Question 68.3 Donner un algorithme qui permet de stocker un nouveau numéro dans un arbre préfixe, éventuellement vide.

```
struct prefix *prefix_add(struct prefix *self,
                                          const char *phone);
```

Question 68.4 Quel est l'inconvénient de la structure de données précédente si l'alphabet utilisé contient plus de caractères que les 10 chiffres considérés ?

Question 68.5 Proposer une structure de donnée pour les arbres préfixes qui résout le problème précédent. Justifier précisément.

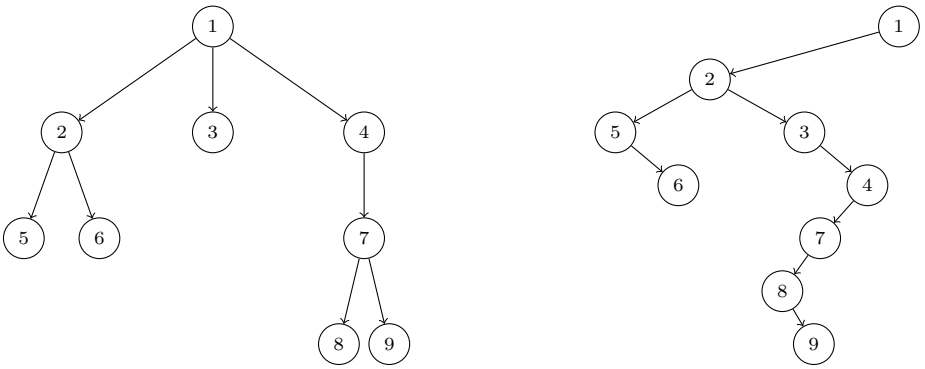


FIGURE 18 – L'arbre général à gauche a pour équivalent l'arbre binaire à droite

Exercice 69 : Transformation d'un arbre général en arbre binaire

Dans cet exercice, le but est de transformer un arbre général en un arbre binaire. Ici, on appelle abre général un arbre n-aire avec n assez grand (dans cet exercice, on prendre $n = 64$). La procédure pour cette transformation est appelée LCRS *(Left-Child Right-Sibling)*, elle consiste pour chaque nœud à prendre pour fils gauche son premier fils dans l'arbre général et pour fils droit son frère à droite dans l'arbre général. La figure 18 montre une telle transformation sur un exemple.

On considère les structures de données suivantes :

```
#define MAX_CHILDREN 64

struct gtree { // general tree
  int data;
  size_t count; // number of children
  struct gtree *children[MAX_CHILDREN];
}

struct btree { // binary tree
  int data;
  struct btree *left;
  struct btree *right;
}
```

Question 69.1 Écrire une fonction qui transforme un arbre général en arbre binaire.

```
struct btree *gtree_to_btree(const struct gtree *g);
```

Question 69.2 Écrire une fonction qui transforme un arbre binaire en arbre général.

```
struct gtree *btree_to_gtree(const struct btree *b);
```

Question 69.3 Si h_G est la hauteur d'un arbre général, quelle est la hauteur maximum h_B de l'arbre binaire résultat ?

Question 69.4 Donner le parcours préfixe pour l'arbre général de la figure 18 ? Et pour l'arbre binaire de la figure 18 ? Que remarque-t-on ? Quel est l'équivalent dans l'arbre binaire du parcours postfixe dans l'arbre général ?

Question 69.5 Que penser de la représentation en arbre binaire par rapport à la représentation en arbre général en terme de mémoire ? Quel inconvénient peut-il y avoir avec la représentation en arbre binaire ?

Exercice 70 : Codage de Huffman

Le codage de Huffman est un algorithme de compression de données sans perte. Son implémentation repose sur la création d'un arbre binaire à partir d'un texte. Plus précisément, le principe de l'algorithme pour le codage de Huffman est le suivant :

1. À partir du texte, on fait une analyse statistique en comptant les occurrences de chaque caractère. On supposera que les caractères du texte sont uniquement des minuscules, sans accent, et l'espace.
2. Pour chaque caractère, on crée un nœud avec comme poids son nombre d'occurences. On trie l'ensemble des nœuds par ordre croissant de poids.
3. Tant qu'on a plus d'un nœud, on prend les deux nœuds de poids le plus faible et on crée un nouveau nœud avec comme fils les deux nœuds et comme poids la somme des poids des deux nœuds. On insère ce nouveau nœud dans la liste triée des nœuds.
4. Quand il n'y a plus qu'un seul nœud, on a un arbre binaire. Chaque caractère est alors codé par son chemin depuis la racine : si on va à gauche, on code avec un 0 et si on va à droite, on code avec un 1.

Ainsi, les caractères apparaissant le plus souvent auront un code binaire plus court que ceux apparaissant moins souvent.

Dans la suite, on donne des structures de données que vous pouvez utiliser. Vous devrez réfléchir aux fonctions à réaliser, en particulier leurs paramètres.

Question 70.1 Saisir un texte grâce à `fgets(3)` en le nettoyant du caractère de passage à la ligne.

Question 70.2 Compter le nombre d'occurrence de chaque caractères du texte.

```
#define CHARS_NUMBER 27

struct statistics {
  int count[CHARS_NUMBER];
};
```

Question 70.3 Construire les nœuds pour chaque caractère et insérer les nœuds dans une liste chaînée triée en fonction des occurrences, et en ne considérerant que les nœuds qui ont plus d'une occurrence. On réalise ainsi un tri par insertion sur une liste chaînée.

```
struct node {
  char c; // '#' for internal nodes
  int count;
  struct node *left;
  struct node *right;
};

struct list {
  struct node *data;
  struct list *next;
};
```

Question 70.4 Appliquer l'algorithme de Huffman décrit précédemment pour contruire l'arbre de Huffman.

Question 70.5 Pour chaque lettre, déterminer son codage en binaire. On parcourera l'arbre en profondeur en construisant le codage au fur et à mesure. Le codage sera stocké sous forme de chaîne de caractère avec des `'0'` et des `'1'`.

```
#define CODING_MAX 27

struct coding {
  char code[CHARS_NUMBER][CODING_MAX];
};
```

Question 70.6 Afficher le codage en binaire du texte initial. Combien de bits sont utilisés ? Combien d'octets fait le texte initial ? Quel est le taux de compression ?

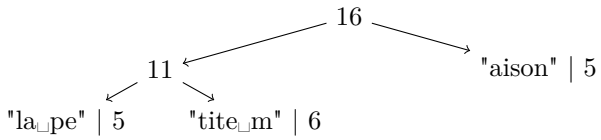


FIGURE 19 – Un exemple de corde

Exercice 71 : Corde

Une corde *(rope)* est un arbre binaire entier qui sert à représenter une très grande chaîne de caractères. Plus précisément, une cordre peut être :

— une feuille composée d'une chaîne de caractères non-vide ;
— un nœud interne qui ne contient pas de chaîne de caractères et dont les deux fils sont des cordes.

En plus, chaque nœud contient un poids qui est, pour une feuille la taille de la chaîne de caractères qu'elle contient, pour les nœuds internes la somme des poids de ses deux fils. La figure 19 représente la corde pour la chaine "la petite maison".

On utilise la structure de données suivante pour représenter une corde :

```
struct rope {
  size_t weight;
  char *data;
  struct rope *left;
  struct rope *right;
};
```

Dans toute la suite, on supposera que l'arbre est équilibré.

Question 71.1 Qu'est-ce qu'un arbre binaire entier ?

Question 71.2 Écrire une fonction qui construit une corde à partir d'une chaîne de caractères. On dupliquera la chaîne de caractères passée en paramètre.

```
struct rope *make_rope_string(const char *data);
```

Question 71.3 Écrire une fonction qui concatène deux cordes. Quelle est sa complexité ?

```
struct rope *rope_concat(struct rope *lhs, struct rope *rhs);
```

Question 71.4 Écrire une fonction qui donne le i^{e} caractère d'une corde. Quelle est sa complexité ?

```
char rope_ith_char(const struct rope *self, size_t i);
```

Question 71.5 Écrire une fonction qui calcule la longueur de la corde, c'est-à-dire la longueur de la chaîne de caractères représentée. Quelle est sa complexité ?

```
size_t rope_length(const struct rope *self);
```

Question 71.6 Écrire une fonction qui insère une chaîne de caractère au rang i d'une corde en créant une nouvelle feuille pour cette chaîne. On dupliquera la chaîne de caractères passée en paramètre. Quelle est sa complexité ?

```
struct rope *rope_insert(struct rope *self,
                                    const char *data, size_t i);
```

Question 71.7 Écrire une fonction qui affiche une corde.

```
void rope_print(const struct rope *self);
```

Question 71.8 Pour les fonctions `concat`, `ith_char`, `length` et `insert`, faire une comparaison de complexité avec les chaînes de caractères.

Exercice 72 : Arbre de décision des algorithmes de tri

On considère un tableau à n éléments dont les éléments sont initialement $[x_1, x_2, \ldots, x_n]$. Un arbre de décision d'un algorithme de tri est un arbre binaire avec les caractéristiques suivantes :

— les nœuds internes sont étiquetés par une condition de la forme $x_i < x_j$, avec le fils gauche qui représente les décisions si la condition est vraie et le fils droit les décisions si la condition est fausse.

— les feuilles sont étiquetées par une permutation des éléments initiaux de telle sorte qu'ils soient classés par ordre croissant

La figure 20 montre l'arbre de décision pour l'algorithme de tri par insertion. Pour une instance donnée, c'est-à-dire des valeurs concrètes affectées aux x_i, l'exécution de l'algorithme de tri correspond à un chemin jusqu'à une feuille. Par exemple, si $x_1 = 27, x_2 = 42, x_3 = 7$, alors l'exécution va arriver à la feuille indiquée en pointillés (qui correspond bien au tableau trié).

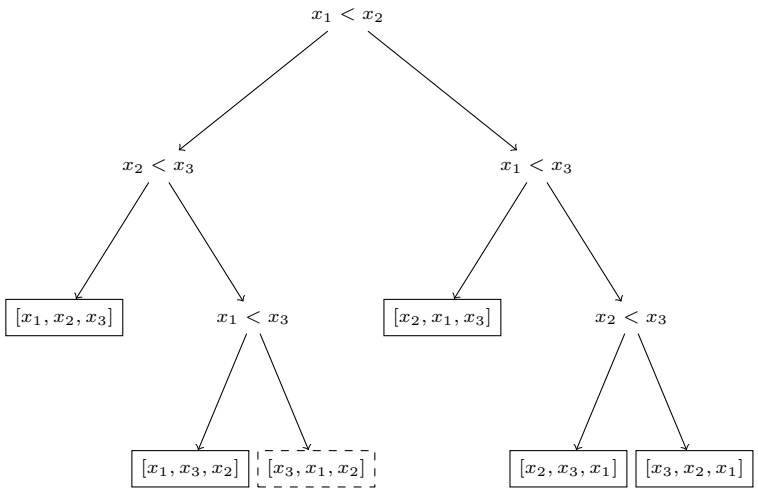


FIGURE 20 – Arbre de décision pour l'algorithme de tri par insertion

On représente un arbre de décision avec la structure suivante :

```
struct decision {
  size_t *permutation;
  size_t indices[2];
  struct decision *left;
  struct decision *right;
};
```

Si `permutation` est non-nul, alors le nœud est une feuille et ce champ représente une permutation ($[x_3, x_1, x_2]$ est représentée par $[3, 1, 2]$). Sinon, le tableau `indices` donne les deux indices des éléments qui sont comparés et les deux fils sont non-nuls.

Question 72.1 On suppose qu'on dispose d'un arbre de décision. Écrire une fonction pour exécuter cet arbre pour trier un tableau donné en paramètre.

```
void execute(const struct decision *self,
          const int *input, int *output, size_t n);
```

Question 72.2 Combien de feuilles f doit avoir au minimum l'arbre de décision en fonction de n la taille du tableau ? Justifier.

Question 72.3 Soit un arbre binaire de taille k et de hauteur h, quelles relations y a-t-il entre k et h ? Dans quels cas a-t-on égalité ?

Question 72.4 En déduire qu'on a alors $f \leq 2^h$. Justifier.

Question 72.5 Quelle est alors la hauteur minimale d'un arbre de décision ? On pourra utiliser $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

Question 72.6 En déduire que tout algorithme de tri par comparaison effectue au moins $\Omega(n \log n)$ comparaisons. Justifier.

10 Arbres binaires de recherche

Exercice 73 : Propriétés des arbres binaires de recherche

Question 73.1 Combien y a-t-il d'arbres binaires de recherche dont les éléments sont $\{3, 5, 8, 12\}$?

Question 73.2 On suppose que les entiers compris entre 1 et 1000 sont disposés dans un arbre binaire de recherche, et on souhaite retrouver le nombre 363. Parmi les séquences suivantes, lesquelles ne pourraient pas être la séquence de nœuds parcourus ?

1. 2, 252, 401, 398, 330, 344, 397, 363 ;
2. 924, 220, 911, 244, 898, 258, 362, 363 ;
3. 925, 202, 911, 240, 912, 245, 363 ;
4. 2, 399, 387, 219, 266, 382, 381, 278, 363 ;
5. 935, 278, 347, 621, 299, 392, 358, 363.

Est-il nécessaire de faire des schémas ? Écrire sous une forme minimale la propriété à vérifier.

Question 73.3 On considère tous les nombres compris entre 1 et 1000. Donnez deux ordres d'insertion de ces nombres dans un ABR :

— l'un qui va donner un arbre complètement déséquilibré, c'est-à-dire de hauteur maximale ;
— l'autre qui va donner un arbre équilibré, c'est-à-dire le moins haut possible.

Question 73.4 Proposer un algorithme qui calcule le maximum d'un arbre binaire de recherche.

Exercice 74 : Arbre binaire de recherche

Question 74.1 Indiquer l'arbre binaire de recherche obtenu en partant de l'arbre vide et en ajoutant successivement les éléments suivants : 8, 5, 3, 10, 4, 7, 2, 9, 11

Question 74.2 Indiquer l'arbre binaire de recherche obtenu en partant de l'arbre précédent et en supprimant l'élément 5.

On dispose des structures de données suivantes :

```
struct node {
  int value;
  struct node *left;
  struct node *right;
}

struct tree {
  struct node *root;
}
```

Question 74.3 Donner le code d'une fonction en C qui ajoute une valeur à un arbre binaire de recherche avec le prototype suivant. Quelle est la complexité de cette fonction ?

```
void tree_add(struct tree *self, int value);
```

Question 74.4 Donner le code d'une fonction en C qui calcule la somme des éléments des nœuds d'un arbre avec le prototype suivant. La fonction fait-elle un parcours préfixe, infixe, postfixe ? Quelle est la complexité de cette fonction ?

```
int tree_sum(const struct tree *self);
```

Exercice 75 : Arbre binaire de recherche de chaînes de caractères

On considère la structure de données suivante :

```
struct node {
  char *data;
  struct node *left;
  struct node *right;
};

struct bst {
  struct node *root;
};
```

Question 75.1 Donner le code d'une fonction qui examine si un élément est présent dans un arbre binaire de recherche :

```
bool bst_search(const struct bst *self, char *data) {
```

Question 75.2 Donner le code d'une fonction qui ajoute un élément à un arbre binaire de recherche. On prendra garde à dupliquer la chaîne passée en paramètre.

```
void bst_insert(struct bst *self, char *data);
```

Exercice 76 : i-ième élément d'un arbre binaire de recherche

Dans cet exercice, on veut trouver le i-ième élément d'un arbre binaire de recherche. On supposera que la taille de l'arbre est supérieure à i. On dispose de la structure suivante :

```
struct bst {
  int data;
  struct bst *left;
  struct bst *right;
};
```

Question 76.1 On suppose qu'on dispose d'un tableau dynamique `struct array` et d'une fonction pour ajouter un élément à la fin du tableau dynamique `array_add` (on ne demande pas son implémentation).

```
struct array {
  int *data;
  size_t capacity;
  size_t size;
};

void array_add(struct array *self, int e);
```

L'idée du premier algorithme est de parcourir l'arbre et ajouter ses éléments au tableau dynamique puis de renvoyer le i-ième élément du tableau. Écrire cette fonction. Quelle est sa complexité ?

```
void bst_in_array(const struct bst *node, struct array *a);
int bst_ith_element(const struct bst *node, size_t i);
```

Question 76.2 Dans ce deuxième algorithme, on ajoute un paramètre qui va permettre de calculer la taille de l'arbre. Si i est plus grand que la taille de l'arbre, alors on indique la taille de l'arbre dans `*size` et on renvoie 0. Sinon, on renvoie la i-ème valeur et on indique i dans `*size`. Écrire cette fonction. Quelle est sa complexité ?

```
int bst_ith_element(const struct bst *node, size_t i,
                                            size_t *size);
```

Question 76.3 On suppose qu'on dispose d'une fonction `bst_size` qui renvoie la taille de l'arbre avec une complexité $O(1)$. Écrire une fonction qui répond au problème en utilisant `bst_size`.

```
int bst_ith_element(const struct bst *node, size_t i);
```

Question 76.4 Sans changer la structure, quelle est la complexité de la fonction `bst_size` ?

Question 76.5 On décide de changer la structure. Comment la modifier ? Donner le code de la fonction `bst_size` ? Est-elle bien de complexité $O(1)$?

Question 76.6 On veut vérifier que le changement de la structure ne modifie pas la complexité des opérations usuelles sur les arbres binaires de recherche. Écrire la fonction d'insertion d'un élément en tenant compte du changement de la structure. Quelle est sa complexité ?

```
struct bst *bst_insert(struct bst *node, int data);
```

Exercice 77 : Insertion à la racine d'un arbre binaire de recherche

Question 77.1 On considère l'arbre binaire de recherche de la figure 21. Quel est l'abre binaire de recherche obtenu après avoir inséré 14 avec l'algorithme vu en cours ?

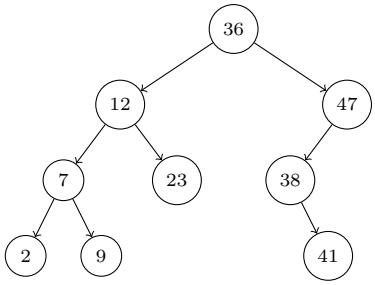


FIGURE 21 – Un arbre binaire de recherche

Question 77.2 La figure 22 montre le principe des rotation droite et gauche. Donner l'arbre binaire de recherche obtenu en appliquant une rotation droite à l'arbre de la figure 21.

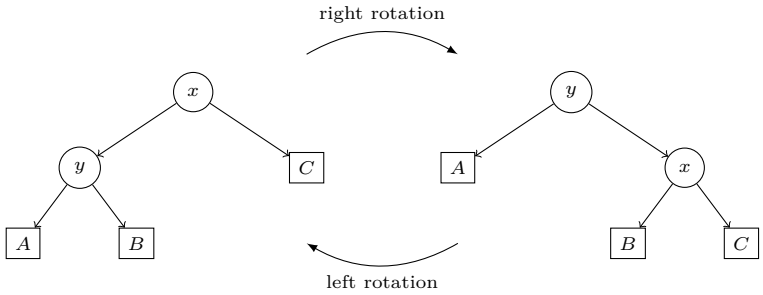


FIGURE 22 – Rotations droite et gauche. x et y sont des nœuds, A, B et C sont des arbres, éventuellement vides

Question 77.3 On considère la structure de données suivante :

```
struct tree {
  int data;
  struct tree *left;
  struct tree *right;
};
```

Écrire une fonction qui permet de faire une rotation gauche, et une fonction qui permet de faire une rotation droite.

```
struct tree *tree_left_rotate(struct tree *node);
struct tree *tree_right_rotate(struct tree *node);
```

Question 77.4 Le principe de l'algorithme d'insertion à la racine est d'insérer la valeur à la racine du sous-arbre droit ou du sous-arbre gauche, puis de réaliser une rotation pour remettre la valeur à la racine. Comment détermine-t-on si on insère à la racine du sous-arbre droit ou du sous-arbre gauche ? Quel est l'abre binaire de recherche obtenu après avoir inséré 14 à la racine ?

Question 77.5 Écrire l'algorithme qui permet d'insérer à la racine.

```
struct tree *tree_insert_root(struct tree *node, int data);
```

Question 77.6 Quelle est la complexité de cet algorithme ? Justifier.

Question 77.7 Quel peut être l'intérêt d'insérer à la racine ?

Exercice 78 : Vérification d'un arbre binaire de recherche

Dans cet exercice, on dispose d'un arbre binaire et on veut vérifier que cet arbre est bien un arbre binaire de recherche. On supposera que les arbres sont équilibrés.

Question 78.1 Montrer avec un contre-exemple que vérifier la propriété «un élément est plus grand que son fils gauche et plus petit que son fils droit» ne garantie pas d'avoir un arbre binaire de recherche.

Question 78.2 On considère la structure de données suivante :

```
struct tree {
  int data;
  struct tree *left;
  struct tree *right;
};
```

Si on suppose que cet arbre est un arbre binaire de recherche, où se situe le plus petit élement α et le plus grand élément ω ? Écrire des fonctions pour renvoyer les éléments situés à ces endroits. Quelle est leur complexité ?

```
int tree_get_maybe_min(const struct tree *node);
int tree_get_maybe_max(const struct tree *node);
```

Question 78.3 Si on rencontre un élément qui ne se situe pas dans l'intervalle $[\alpha, \omega]$, quelle conclusion peut-on en tirer ? Justifier.

Question 78.4 Écrire une fonction qui vérifie qu'un arbre est un arbre binaire de recherche en prenant en compte cet intervalle et la remarque de la question précédente. Quelle est sa complexité ?

```
bool is_bst_range(const struct tree *node, int alpha, int omega);
```

Question 78.5 Écrire la fonction qui vérifie si un arbre est un arbre binaire de recherche en utilisant les fonctions précédentes.

```
bool is_bst(const struct tree *node);
```

Question 78.6 Quel type de parcours permet de rencontrer les éléments d'un arbre binaire de recherche dans l'ordre ? Donner une définition de ce parcours.

Question 78.7 Écrire une fonction qui réalise ce parcours et qui vérifie que les éléments rencontrés sont bien dans l'ordre.

```
bool is_bst_walking(const struct tree *node);
```

Exercice 79 : Reconstruction d'un arbre binaire de recherche

En utilisant les fonctions vues en cours, un arbre binaire de recherche peut devenir déséquilibré. On propose dans cet exercice de reconstruire un arbre binaire de recherche, c'est-à-dire de produire un arbre binaire de recherche avec les mêmes éléments mais dont on saura qu'il est bien équilibré. Pour cela, on stockera tous les éléments de l'arbre dans un tableau, puis à partir du tableau, on générera un nouvel arbre binaire de recherche.

On dispose de la structure suivante :

```
struct bst {
  int data;
  struct bst *left;
  struct bst *right;
};
```

Question 79.1 Qu'est-ce qu'un arbre binaire de recherche ?

Question 79.2 Écrire une fonction qui calcule la taille d'un arbre binaire.

```
size_t bst_size(const struct bst *self);
```

Question 79.3 Définir précisément ce qu'est le parcours infixe d'un arbre binaire. Dans le cas des arbres binaires de recherche, quelle propriété possède le parcours infixe ?

Question 79.4 Écrire une fonction qui réalise un parcours infixe pour stocker les nœuds d'un arbre binaire de recherche dans un tableau. On utilisera la structure `struct array` qui contient un tableau avec suffisamment de place pour stocker tous les nœuds de l'arbre binaire de recherche et un compteur qui indique le nombre de cases occupées à partir du début du tableau.

```
struct array {
  struct bst **data;
  size_t count;
};

void bst_to_array(const struct bst *self, struct array *out);
```

Question 79.5 Écrire une fonction qui génère un arbre binaire de recherche à partir d'un tableau. On s'assurera que le résultat est un arbre binaire bien équilibré.

```
struct bst *bst_from_array(const struct array *in);
```

Question 79.6 Finalement, écrire une fonction qui reconstruit un arbre binaire de recherche. En détaillant les opérations en jeu, quelle est sa complexité ? Quelle est sa complexité en mémoire ?

```
struct bst *bst_reconstruct(const struct bst *self);
```

Exercice 80 : Ensemble plat

Un ensemble plat est un ensemble qui utilise un tableau trié à la place d'un arbre binaire de recherche. Nous allons examiner la pertinence de ce choix. Dans toute la suite, on supposera que les arbres binaires de recherche sont équilibrés. On notera h la hauteur d'un arbre et n le nombre d'éléments dans l'ensemble. Soyez le plus précis possible dans vos réponses.

Question 80.1 Quelle est la complexité d'une recherche dans un arbre binaire de recherche ?

Question 80.2 Quel algorithme permet de chercher dans un tableau trié ? Expliquer son principe en quelques mots. Quel est sa complexité ?

Question 80.3 Quelle est la complexité d'une insertion dans un arbre binaire de recherche ?

Question 80.4 Quelle est la complexité d'une insertion dans un tableau trié ? Justifier.

Question 80.5 On suppose qu'on part d'un ensemble vide et qu'on doit insérer n éléments en une seule fois. Quelle est alors la complexité de l'insertion de n éléments dans un arbre binaire de recherche vide ?

Question 80.6 Quelle est la complexité de l'insertion de n éléments en une seule fois dans un tableau trié vide ? Expliquer l'algorithme en quelques mots.

Question 80.7 Quelle est la complexité de la suppression dans un arbre binaire de recherche ?

Question 80.8 Quelle est la complexité de la suppression dans un tableau trié ?

Question 80.9 Au final, dans quels cas recommanderiez-vous d'utiliser les ensembles plats ?

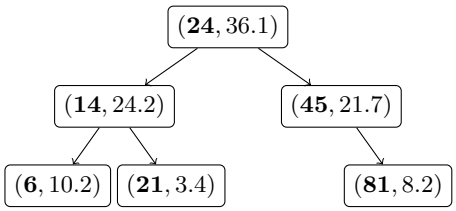


FIGURE 23 – Un example d'arbretas

Exercice 81 : Arbretas

Un arbretas *(treap)* est une fusion entre un arbre binaire de recherche et un tas. Plus exactement, les éléments d'un arbretas sont des couples (x, y) de sorte que les x forment un arbre binaire de recherche tandis que les y sont partiellement ordonnés (comme un tas). Les x sont donc les données de l'arbre, les y sont choisis au hasard (on les appelle *priorités* des nœuds et ils sont supposés tous différents). On peut montrer que les arbretas ont une hauteur logarithmique avec une forte probabilité. La figure 23 montre un exemple d'arbretas avec les x en gras.

On dispose de la structure suivante :

```
struct treap {
  int data;
  float priority;
  struct treap *left;
  struct treap *right;
};
```

Question 81.1 Écrire une fonction qui sépare un arbretas en deux en fonction d'un élément non-présent dans l'arbretas. L'arbretas `*left` contiendra les éléments plus petits que `data` et l'arbretas `*right` contiendra les éléments plus grand que `data`. Quelle est la complexité de cette fonction ? Justifer.

```
void treap_split(struct treap *node, int data,
                      struct treap **left, struct treap **right);
```

Question 81.2 Écrire une fonction qui insère un élément dans un arbretas. On supposera que l'élément n'est pas présent. Pour cela, on descend dans l'arbretas comme dans un arbre binaire de recherche jusqu'à ce que la priorité soit supérieure à la priorité du nœud. On a donc trouvé le bon emplacement, et il suffit alors de séparer le nœud trouvé pour créer les sous-arbres gauche et droit du nouveau nœud. Quelle est la complexité de cette fonction ? Justifer.

```
struct treap *treap_insert(struct treap *node,
                                          int data, float prio);
```

Question 81.3 Nous avons implémenté l'insertion d'un nœud à l'aide de l'opération de séparation. Expliquer en français comment on pourrait implémenter l'opération de séparation à l'aide de l'insertion.

Question 81.4 Écrire une fonction qui fusionne deux arbretas. On supposera que tous les éléments de `left` sont plus petits que tous les éléments de `right`. On fera attention à conserver l'ordre partiel des nœuds. Quelle est la complexité de cette fonction ? Justifer.

```
struct treap *treap_merge(struct treap *left,
                                              struct treap *right);
```

Question 81.5 Écrire une fonction qui supprime un élément. Pour cela, une fois l'élément trouvé, on fusionnera les deux sous-arbres gauche et droite. Quelle est la complexité de cette fonction ? Justifer.

```
struct treap *treap_remove(struct treap *node, int data);
```

Question 81.6 Nous avons implémenté la suppression d'un élément à l'aide de l'opération de fusion. Expliquer en français comment on pourrait implémenter l'opération de fusion à l'aide de la suppression.

Exercice 82 : Parcours itératif d'un arbre binaire de recherche

Le but de cet exercice est de faire un parcours d'un arbre binaire de recherche avec une fonction itérative. Plus précisément, le but est de calculer, étant donné un nœud, quel est le nœud suivant dans l'ordre infixe.

On considère la structure de données suivante :

```
struct bst {
  int data;
  struct bst *parent;
  struct bst *left;
  struct bst *right;
};
```

Par rapport à la structure classique vu en cours, on ajoute un pointeur vers le nœud parent, qui est `NULL` pour la racine de l'arbre. Dans tout cet exercice, il est recommandé d'illustrer vos explications par des dessins.

Question 82.1 Qu'est-ce que le parcours infixe d'un arbre binaire ?

Question 82.2 Écrire une fonction qui insère un élément dans un arbre binaire de recherche. On prendra garde à bien mettre à jour le pointeur `parent`.

```
struct bst *bst_insert(struct bst *node, int data);
```

Question 82.3 Écrire une fonction qui détermine si un nœud est la racine.

```
bool bst_is_root(struct bst *node);
```

Question 82.4 Écrire une fonction qui détermine si un nœud est l'enfant gauche de son parent, et une fonction qui détermine si un nœud est l'enfant droit de son parent. La racine n'est l'enfant d'aucun nœud.

```
bool bst_is_left_child(struct bst *node);
bool bst_is_right_child(struct bst *node);
```

Question 82.5 Écrire une fonction qui renvoie le premier nœud dans l'ordre infixe à partir de la racine de l'arbre.

```
struct bst *bst_first(struct bst *root);
```

Question 82.6 On suppose que le nœud courant a un enfant droit non-vide. Expliquer, en français, comment trouver le nœud qui suit le nœud courant.

Question 82.7 On suppose que le nœud courant a un enfant droit vide. Si le nœud courant est un enfant gauche, expliquer, en français, comment trouver le nœud qui suit le nœud courant. Si le nœud courant est un enfant droit, expliquer, en français, comment trouver le nœud qui suit le nœud courant.

Question 82.8 À l'aide des réponses précédentes, écrire une fonction qui renvoie le nœud qui suit le nœud courant. S'il n'y a pas de nœud suivant, la fonction renvoie `NULL`. Quelle est sa complexité ?

```
struct bst *bst_next(struct bst *node);
```

11 Tas

Exercice 83 : Propriétés des tas

Question 83.1 Dessiner tous les tas possibles avec les éléments suivants : 1, 4, 7, 9, 12.

Question 83.2 Entasser l'élément 11 dans le tas de la figure 24.

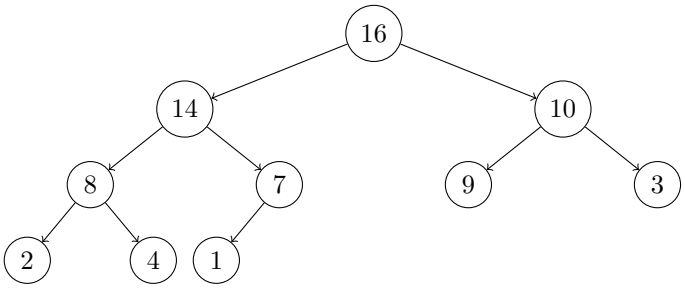


FIGURE 24 – Un tas

Question 83.3 Supprimer le sommet du tas de la figure 24.

Question 83.4 Donner la représentation en tableau du tas de la figure 24.

Question 83.5 Dans un tas où tous les éléments sont distincts, quelles sont les positions possibles pour le plus petit élément ?

Exercice 84 : Implémentation d'un tas pour l'algorithme de Dijkstra

L'algorithme de Dijkstra utilise un tas pour stocker les nœuds du graphe à traiter. Cependant, l'opération `decrease_key` ne fait pas partie des opérations de base vues en cours pour un tas. Le but de cet exercice est de voir comment implémenter cette opération dans le cadre de l'algorithme de Dijkstra.

Dans l'algorithme de Dijkstra, les éléments du tas sont des nœuds du graphe. On décide de représenter ces n nœuds avec un identifiant entier entre 0 et $n-1$. La difficulté consiste à retrouver l'indice d'un élément dans le tableau du tas à partir de cet élément. Pour cela, on propose d'utilise un tableau annexe qui donnera cette information : à partir d'un identifiant de nœud, on aura son indice dans le tableau du tas.

On utilise les structures suivantes : le type `node_id_t` sert pour l'identifiant des nœuds, la structure `heap_element` contient un identifiant de nœud et la priorité de ce nœud dans le tas, la structure `heap` est le tas muni du tableau annexe `indices`. On supposera que les tableaux ont une taille suffisante.

```
typedef unsigned long node_id_t;

struct heap_element {
  node_id_t node;
  int priority;
};

struct heap {
  size_t size;
  struct heap_element *data;
  size_t *indices;
};
```

Question 84.1 Donner le code d'une fonction qui permet d'échanger deux éléments du tas en prenant en compte le tableau annexe.

```
void heap_swap(struct heap *self, size_t i, size_t j);
```

Question 84.2 Rappeler l'algorithme pour l'insertion d'un élement dans un tas. Donner les modifications à apporter pour prendre en compte le tableau annexe. Quelle est la nouvelle complexité de cette fonction ? Justifier.

Question 84.3 Donner le code de la fonction `decrease_key`. Quelle est sa complexité ? Justifier.

```
void heap_decrase_key(struct heap *self,
                                  node_id_t node, int priority);
```

Question 84.4 Si les identifiants de nœuds sont des entiers quelconques, on ne peut plus utiliser un tableau. On peut alors utiliser un tableau associatif *(map)* implémenté avec un arbre binaire de recherche. Quel est alors la complexité de `heap_swap` ? Quelle la complexité d'une insertion dans le tas ? Justifier.

Exercice 85 : File de priorité et algorithme de tri (7,5 points)

Une file de priorité *(priority queue)* est une structure de données qui est définie par l'interface suivante :

- `pq_insert` : cette fonction insère un élément dans la file de priorité, sa complexité est notée $f(n)$;
- `pq_pop` : cette fonction extrait l'élément ayant la priorité la plus grande, sa complexité est notée $g(n)$.

Question 85.1 Décrire en français un algorithme de tri qui utilise une file de priorité. Quel est la complexité de cet algorithme en fonction de $f(n)$ et $g(n)$?

—

On suppose qu'on implémente la file de priorité avec un tas.

Question 85.2 Qu'est-ce qu'un tas ?

Question 85.3 Rappelez la complexité de chacune des deux opérations. Quelle est alors la complexité du tri utilisant cette implémentation ?

—

On suppose désormais qu'on implémente la file de priorité avec un tableau non-trié.

Question 85.4 Expliquer comment implémenter la fonction `pq_insert`. Quelle est sa complexité ?

Question 85.5 Expliquer comment implémenter la fonction `pq_pop`. Quelle est sa complexité ?

Question 85.6 Comment s'appelle le tri utilisant cette implémentation ? Quel est sa complexité ?

—

On suppose désormais qu'on implémente la file de priorité avec un tableau trié.

Question 85.7 Expliquer comment implémenter la fonction `pq_insert`. Quelle est sa complexité ?

Question 85.8 Expliquer comment implémenter la fonction `pq_pop`. Quelle est sa complexité ?

Question 85.9 Comment s'appelle le tri utilisant cette implémentation ? Quel est sa complexité ?

12 Graphes

Exercice 86 : La plante, la chèvre et le loup

Un homme (H) se trouve au bord d'une rivière avec une plante (P), une chèvre (C) et un loup (L). Il possède une barque et veut traverser la rivière. Cependant, il ne peut mettre en même temps, en plus de lui, dans la barque qu'un des trois. Et, malheureusement aussi, il ne peut laisser sur une rive seul le loup avec la chèvre, ni la chèvre avec la plante.

Question 86.1 Modéliser ce problème par un graphe et proposer une solution.

Exercice 87 : Rayon et diamètre d'un graphe

Soit $G = (V, E)$ un graphe valué, la *distance* entre les nœuds x et y est la longueur du plus court chemin entre x et y. On peut alors définir l'*excentricité* d'un nœud x, noté $e(x)$ comme étant la distance maximale à tous les autres sommets. Le rayon d'un graphe, noté R, est alors l'excentricité minimale de ses nœuds, et le diamètre d'un graphe, noté D, est l'excentricité maximale de ses nœuds.

Question 87.1 Donner en le justifiant un exemple de graphe avec quatre nœuds tel que son rayon est égal à son diamètre.

Question 87.2 Donner le nom de deux algorithmes qui permettent de calculer le plus court chemin. Quelle est leur complexité ? On notera $\mathcal{C}_0$ la plus petite de ces complexités dans la suite.

Question 87.3 Justifier que le calcul de l'excentricité pour un nœud x a une complexité en $\mathcal{C}_0 + |V|$.

Question 87.4 Donner un algorithme pour calculer toutes les excentricités des nœuds d'un graphe. Quelle est sa complexité en fonction de $\mathcal{C}_0$? Quelle est sa complexité ?

Question 87.5 L'algorithme de Floyd-Warshall est un algorithme qui permet de calculer la distance des plus courts chemins entre tous les nœuds d'un graphe. Il renvoie donc une matrice W de taille $|V| \times |V|$ telle que W_{ij} est la distance du plus court chemin entre i et j. Sa complexité est en $O(|V|^3)$. Donner un algorithme pour calculer toutes les excentricités des nœuds d'un graphe à l'aide de l'algorithme de Floyd-Warshall. Quelle est sa complexité ?

Question 87.6 Comparer les complexités des deux questions précédentes en fonction de la densité du graphe.

Question 87.7 Donner un algorithme pour calculer le rayon et le diamètre d'un graphe. Si on appelle $\mathcal{C}_1$ la complexité de l'algorithme qui calcule toutes les excentricités des nœuds, quelle est sa complexité ?

13 Algorithmes gloutons

Exercice 88 : Problème de rendu de monnaie

Un système de pièce S est un n-uplet $(c_1, \ldots, c_n)$ où c_i représente la valeur de la i^{e} pièce. On suppose que ces valeurs sont des entiers strictement croissants et que $c_1 = 1$.

Étant donné un système S et une valeur v, le problème de rendu de monnaie consiste à trouver un n-uplet d'entiers positifs $(x_1, x_2, \ldots, x_n)$ qui minimise $M(v) = \sum_{i=1}^{n} x_i$ sous la contrainte $\sum_{i=1}^{n} x_i c_i = v$. Dit autrement, v représente la somme de monnaie à rendre et x_i le nombre de pièces c_i à rendre.

Question 88.1 Proposer un algorithme glouton qui choisit la plus grosse pièce qu'on peut rendre (sans rendre trop) tant qu'il reste quelque chose à rendre et qui renvoie le nombre de pièces choisies.

```
int money(int *system, size_t size, int value);
```

Question 88.2 Montrer avec un exemple que cet algorithme glouton ne fait pas toujours le choix optimal, c'est-à-dire celui qui minimise le nombre de pièces.

Question 88.3 On remarque que si on sait rendre de façon optimale toutes les valeurs strictement inférieures à v. Alors pour rendre v, il suffit de prendre une pièce parmi les n possibles et une fois la pièce choisie, la somme restante est strictement inférieure à v donc on sait la rendre de manière optimale. On a donc :

$$M^{\text{OPT}}(v) = 1 + \min_{1 \leq i \leq n} M^{\text{OPT}}(v - c_i)$$

Imaginer un algorithme qui permettent de calculer $M^{\text{OPT}}(v)$. L'appliquer sur l'exemple précédent.

Exercice 89 : Problème du sac à dos

On dispose de n objets. L'objet i a un poids w_i et une valeur p_i. On dispose également d'un sac à dos ne pouvant contenir qu'un poids total W. Le problème du sac à dos consiste à choisir parmi les n objets ceux qui auront la valeur maximale et qui rentreront dans le sac à dos. On cherche donc $x_i \in \{0, 1\}$ qui maximise $\sum_{i=1}^{n} x_i p_i$ sous la contrainte $\sum_{i=1}^{n} x_i w_i \leq W$.

Question 89.1 Quel est le nombre de combinaisons d'objets possibles au maximum ?

Question 89.2 Proposer un algorithme glouton qui permet de remplir le sac à dos en partant des objets qui sont le plus «rentable». Quel est sa complexité ?

```
void knapsack(const double w[], const double p[], size_t n,
                                        double weight, int x[]);
```

Question 89.3 L'appliquer sur les objets suivants en considérant que le sac peut contenir jusqu'à $W = 15$.

i	1	2	3	4	5
w_i	2	5	9	7	12
p_i	1	2	10	3	7

Question 89.4 Montrer à l'aide d'un exemple simple que la qualité de la solution fournie par l'algorithme glouton précédent peut être rendue aussi mauvaise que possible, c'est-à-dire que le rapport entre la valeur du sac optimal sur la valeur du sac glouton peut être aussi grande que l'on veut.

Question 89.5 Si on suppose que les poids et les valeurs sont des entiers, on peut alors résoudre le problème de manière optimale. On considère un tableau T à deux dimensions où $T[i, c], 0 \leq i \leq n, 0 \leq c \leq W$ représente une solution optimal du problème avec i objets et un sac de contenance c. Étant donné un nombre d'objet i et une contenance c, on remarque alors que :

— soit on prend l'objet i ($x_i = 1$), et on s'intéresse au problème à $i - 1$ objets et un sac de contenance $c - w_i$;

— soit on ne prend pas l'objet i ($x_i = 0$), et on s'intéresse au problème au problème à $i - 1$ objets et un sac de contenance c.

En regardant la meilleure des deux situations, on détermine la solution optimale pour le problème avec i objets et un sac de contenance c. Contruire le tableau T pour l'exemple de la question 3. Quel est la complexité en temps de cet algorithme ? Quel est la complexité en mémoire ?