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

Exercice 1 : Ordres de grandeur

Question 1.1 Qu'est-ce qui est le plus grand : 10^{100} ou 100^{10} ?

Question 1.2 Combien de chiffres comporte 2^n en écriture décimale ?

Exercice 2 : Échelle de fontion

Question 2.1 Classer ces fonctions par ordre croissant 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 2.2 Les assertions suivantes sont elles vraies ou fausses, pourquoi ?

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

Question 2.3 Calculer $O(\log(n!))$. On pourra utiliser le résultat suivant : si $f \sim g$ alors $O(\log f) = O(\log g)$.

Exercice 3 : 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 3.1 Recopier le code source suivant dans un fichier appelé `hello.c`.

```
#include <stdio.h>

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

Question 3.2 Compiler le programme avec la ligne de commande suivante :

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

Question 3.3 Exécuter le programme :

```
./hello
```

Exercice 4 : 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 4.1 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 4.2 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 4.3 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 4.4 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 5 : Comptage d'opérations

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

Question 5.1 On considère la fonction suivante :

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

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

  return res;
}
```

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

Question 5.2 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 $\mathcal{N}$ en fonction de n ?

Question 5.3 Dans la fonction précédente, on remplace :

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

par :

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

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

Question 5.4 Dans ce dernier cas, proposez une modification pour améliorer $\mathcal{N}$. Que vaut alors $\mathcal{N}$?

Exercice 6 : 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 6.1 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 7 : 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 7.1 Tester cette fonction pour $n = 5$. Que calcule f ?

Question 7.2 On prend comme opération fondamentale l'addition. Calculer la complexité de cette fonction ?

Question 7.3 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 7.4 Peut-on encore améliorer la complexité ?

Exercice 8 : 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 8.1 Quelle est l'opération fondamentale dans ces fonctions ?

Question 8.2 Quelle est la complexité de `f` ? Justifier précisément.

Question 8.3 Donner en fonction de `n` un ordre de grandeur du résultat de ce qui est calculé par `f` ?

Question 8.4 Quelle est la complexité de `g` ? Justifier précisément.

Question 8.5 Donner en fonction de `n` un ordre de grandeur du résultat de ce qui est calculé par `g` ?

Question 8.6 Réécrire la fonction `g` pour diminuer sa complexité ? Quelle est alors sa complexité ?

Exercice 9 : 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 9.1 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 9.2 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 9.3 Quelle est la complexité de `power_v3` ? Quelle conclusion peut-on en tirer ?

Exercice 10 : 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 10.1 Quelle est l'opération fondamentale de cet algorithme ?

Question 10.2 Quel est la complexité $\mathcal{C}_1(n)$ de cet algorithme ? On pourra considérer $\mathcal{C}'_1(n) = \mathcal{C}_1(n) + 1$.

Question 10.3 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 10.4 Proposer un algorithme récursif en C qui permet de calculer $F(n)$ grâce à ces propriétés.

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

Question 10.5 Donner la formule pour calculer la complexité $\mathcal{C}_3(n)$ en fonction de $\mathcal{C}_3\left(\frac{n}{2}\right)$.

Question 10.6 Résoudre cette formule grâce au théorème Diviser Pour Régner en justifiant clairement.

—

On constante 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 10.7 Quel algorithme permet de calculer $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$? Quel est sa complexité ?

Question 10.8 Quel est alors la complexité $\mathcal{C}_4(n)$ pour calculer $F(n)$?

Exercice 11 : 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 11.1 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 11.2 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 11.3 Quelle est alors la formule de complexité ? Résoudre la formule avec le théorème Diviser Pour Régner en justifiant clairement.

Exercice 12 : 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 :

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

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

Question 12.2 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 12.3 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 12.4 Vérifier que ce calcul est bien exact

Question 12.5 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 12.6 Résoudre cette formule grâce au Théorème Diviser pour Régner.

Exercice 13 : 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. $T(n) = 64T(n/8) - n^2 logn$
20. $T(n) = 7T\left(\frac{n}{3}\right) + n^2$
21. $T(n) = 4T\left(\frac{n}{2}\right) + \log n$
22. $T(n) = T\left(\frac{n}{2}\right) + n(2 - \cos n)$

Exercice 14 : Manipulation de pointeurs

On suppose que les pointeurs p, q et r se trouvent respectivement aux cases mémoire d'adresses 101, 102 et 103. On suppose également que le contenu de la mémoire est donné par la figure 1.

101	104	p
102	104	q
103	105	r
104	23	
105	23	

FIGURE 1 – État de la mémoire

Question 14.1 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 14.2 Quel est l'état de la mémoire après cette exécution ?

Exercice 15 : Exécution d'un programme (1)

Question 15.1 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 16 : Exécution d'un programme (2)

Question 16.1 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 17 : Exécution d'un programme (3)

Question 17.1 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;
}

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 18 : 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 18.1 Que se passe-t-il lors de la compilation du programme suivant ? Expliquer.

Question 18.2 Que se passe-t-il lors de l'exécution une fois les erreurs de compilation supprimées ? Expliquer.

Exercice 19 : 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 19.1 Qu'est-ce qui s'affiche à l'écran lors de l'exécution du programme ? Expliquer.

Exercice 20 : Tableaux

Dans les questions suivantes, coder les fonctions demandées en prenant bien soin de réfléchir au nom et aux paramètres de la fonction. Pour chaque algorithme, on donnera l'opération considérée et sa complexité. On utilisera chaque fonction dans `main` sur un exemple quelconque.

Question 20.1 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.

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

Question 20.2 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 20.3 Écrire une fonction qui calcule la somme des éléments du tableau.

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

Question 20.4 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 20.5 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 20.6 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 21 : Fonctions sur les chaînes de caractères

Question 21.1 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 21.2 Écrire une fonction qui calcule la longueur de la chaîne de caractère. Comparer avec ce que renvoie `strlen(3)`.

Question 21.3 Écrire une fonction qui compte le nombre d'espaces dans la chaînes. Indice : `isspace(3)`.

Question 21.4 Écrire une fonction qui affiche la chaîne en enlevant les voyelles (non-accentuées) de la chaîne de caractères.

Question 21.5 Écrire une fonction qui détermine si la chaîne est bien parenthésée.

Question 21.6 Écrire une fonction qui calcule la valeur numérique d'un entier représenté en binaire par une chaîne de caractère.

Exercice 22 : 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 sorted_list_node {
  int data;
  struct sorted_list_node *next;
};

struct sorted_list {
  struct sorted_list_node *first;
}
```

Question 22.1 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 sorted_list_add(struct sorted_list *self, int data);
```

Question 22.2 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 sorted_list_import(struct sorted_list *self,
    const int *other, size_t k);
```

Question 22.3 Donner le code d'une fonction qui supprime les doublons d'une liste ordonnée. Quelle est sa complexité ?

```
void sorted_list_uniq(struct sorted_list *self);
```

Exercice 23 : 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 23.1 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 23.2 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 23.3 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 24 : 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 24.1 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 24.2 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 24.3 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 24.4 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);
```

Exercice 25 : 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 25.1 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 25.2 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 25.3 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 25.4 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 25.5 Résoudre cette formule grâce au théorème Diviser pour régner.

Exercice 26 : 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 26.1 Donner le code d'une fonction qui initialise la file.

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

Question 26.2 Donner le code d'une fonction qui teste si la file est vide.

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

Question 26.3 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 26.4 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 26.5 Donner le code d'une fonction qui supprime l'élément de tête d'une file.

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

Question 26.6 Donner le code d'une fonction qui supprime tous les éléments de la file.

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

Exercice 27 : 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 node {
  int data;
  struct stack_node *next;
};

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

Question 27.1 Donner le code d'une fonction qui initialise une pile vide :

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

Question 27.2 Donner le code d'une fonction qui examine si la pile est vide :

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

Question 27.3 Donner le code d'une fonction qui empile un élément :

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

Question 27.4 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 27.5 Donner le code d'une fonction qui dépile un élément :

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

Question 27.6 Donner le code d'une fonction qui détruit une pile :

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

Exercice 28 : 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 28.1 Donner le code d'une fonction qui initialise la file.

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

Question 28.2 Donner le code d'une fonction qui teste si la file est vide.

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

Question 28.3 Donner le code d'une fonction qui ajoute une valeur au début de la file.

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

Question 28.4 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 28.5 Donner le code d'une fonction qui supprime une valeur au début de la file.

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

Question 28.6 Donner le code d'une fonction qui supprime une valeur à la fin de la file.

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

Question 28.7 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);
```

Exercice 29 : 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 2 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 2 – Exemple du tri d'un tableau d'entiers avec le tri par base

Question 29.1 Implémenter un tri par base sur des entiers avec 10 comme base.

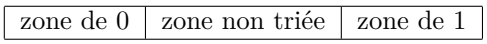
```
void radix_sort(int *data, size_t n);
```

Question 29.2 Quelle est la complexité du tri par base ?

Question 29.3 Ce tri est-il stable ?

Exercice 30 : 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 30.1 Créer un tableau de 100000 éléments de 0 et de 1 tirés au hasard.

Question 30.2 Compter le nombre de 1 du tableau.

Question 30.3 Trier le tableau selon la méthode indiquée.

Question 30.4 Vérifier que le nombre de 1 est identique au nombre de 1 précédemment calculé.

Question 30.5 Quel est la complexité de cet algorithme ?

Exercice 31 : 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 des entiers.

Question 31.1 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 31.2 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 31.3 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 31.4 Écrire une fonction qui échange deux éléments d'un tableau.

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

Question 31.5 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 31.6 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 31.7 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 31.8 On considère le pire cas. Que vaut p ? Quelle est alors la formule de complexité ? Résoudre la formule en justifiant clairement.

Question 31.9 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 32 : Différences entre les éléments distinct 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 32.1 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 $u_0 = 1$ et $u_4 = 8$. On justifiera que l'algorithme est efficace en indiquant sa complexité en nombre de comparaisons.

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

Question 32.2 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(const int *data, size_t n);
```

Question 32.3 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 autre). On justifiera que l'algorithme est efficace en indiquant sa complexité en nombre de comparaisons.

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

Question 32.4 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(const int *data, size_t n);
```

Exercice 33 : 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;
}
```

Question 33.1 On commence par trier les segments. 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. Quelle est la complexité de cette étape de tri ?

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

Question 33.2 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);
```

Question 33.3 Quelle est la complexité globale de cet algorithme ?

Exercice 34 : 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 {
  int 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 34.1 Donner une implémentation en C de `vector_access`. Quelle est sa complexité ?

Question 34.2 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 34.3 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 34.4 Quelle taille peut avoir le tableau `result->data` au maximum en fonction de `v1->size` et `v2->size` ? Justifier.

Question 34.5 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 34.6 Quel algorithme utilise-t-on pour `vector_access` ? Quelle est sa complexité ?

Question 34.7 Donner une idée de l'algorithme pour `vector_set` ? Quelle est sa complexité ?

Question 34.8 Donner une implémentation complète en C de `vector_add`. On prendra garde à la gestion de la mémoire.

Question 34.9 Quelle est la complexité de la fonction que vous venez d'implémenter ?

Question 34.10 Quelle amélioration peut-on proposer pour l'algorithme de la question 5 ? Quelle est alors la complexité de `vector_add` avec cette amélioration ?

Exercice 35 : 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 3 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.

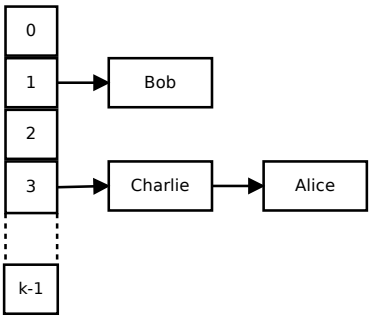


FIGURE 3 – Une table de hachage

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

Question 35.2 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 35.3 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 35.4 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 35.5 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 35.6 Comparer ces complexités avec les complexités moyennes pour la structure de données de la question 1.

Question 35.7 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 35.8 Comparer ces complexités avec les complexités en pire cas pour la structure de données de la question 1.

Question 35.9 Pour une fonction de hachage fixée, que proposeriez-vous de changer pour améliorer la complexité du pire cas ?

Exercice 36 : 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 36.1 Proposer une structure de données pour représenter une grille de 1000×1000.

Question 36.2 Proposer une fonction qui prend en paramètre deux grilles et qui calculent une itération du jeu de la vie.

Exercice 37 : 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 37.1 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 37.2 Quelle structure peut-on utiliser pour représenter un plateau ?

Question 37.3 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 37.4 Donner une formule pour calculer la complexité de l'algorithme précédent. Quel est alors la complexité ?

Question 37.5 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 37.6 Combien y a-t-il de diagonales (ou d'anti-diagonales) sur un plateau de $n \times n$ cases ?

Question 37.7 Grâce aux observations précédentes, écrire un nouvel algorithme qui résout le problème des n reines. Quel est sa complexité ?

Exercice 38 : 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 38.1 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 38.2 Proposer un algorithme pour vérifier que la structure d'arbre est cohérente.

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

Question 38.3 Proposer un algorithme pour évaluer une expression. Quel type de parcours est utilisé dans cet algorithme ?

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

Question 38.4 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 38.5 Proposer un algorithme qui vérifie si deux expressions sont égales. On prendra en compte la commutativité de $+$ et $\times$.

```
bool expr_equals(struct expr *self, struct expr *other);
```

Exercice 39 : 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 4 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.

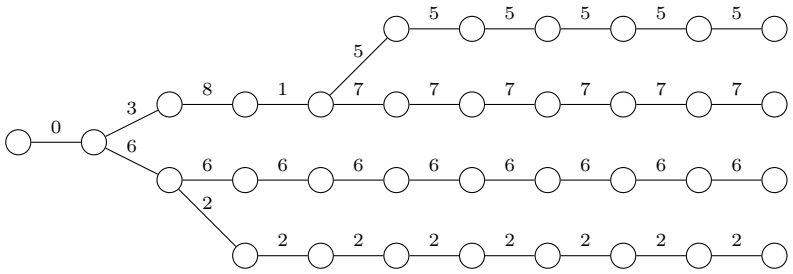


FIGURE 4 – Un arbre préfixe avec les numéros 0381555555, 0381777777, 0666666666, 0622222222

On propose la structure de donnée suivante :

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

Question 39.1 Donner un algorithme qui compte le nombre de nœuds d'un arbre préfixe.

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

Question 39.2 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 39.3 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 39.4 Quel est l'inconvénient de la structure de données précédente si l'alphabet utilisé contient plus de caractères que les 10 chiffres considérés ?

Question 39.5 Proposer une structure de donnée pour les arbres préfixes qui résout le problème précédent. Justifier précisément.

Exercice 40 : Propriétés des arbres binaires de recherche

Question 40.1 Combien y a-t-il d'arbres binaires de recherche dont les éléments sont $\{3, 5, 8, 12\}$?

Question 40.2 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 40.3 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 40.4 Proposer un algorithme qui calcule le maximum d'un arbre binaire de recherche.

Exercice 41 : Arbre binaire de recherche

Question 41.1 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 41.2 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 41.3 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 41.4 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 42 : 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 42.1 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 42.2 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 43 : 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 43.1 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 ?

```
int bst_in_array(const struct bst *self, struct array *a);
int bst_ith_element(const struct bst *self, size_t i);
```

Question 43.2 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 *self, size_t i,
                                            size_t *size);
```

Question 43.3 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 *self, size_t i);
```

Question 43.4 Sans changer la structure, quelle est la complexité de la fonction `bst_size` ?

Question 43.5 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 43.6 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 *self, int data);
```

Exercice 44 : Insertion à la racine d'un arbre binaire de recherche

Question 44.1 On considère l'arbre binaire de recherche de la figure 5. Quel est l'abre binaire de recherche obtenu après avoir inséré 14 avec l'algorithme vu en cours ?

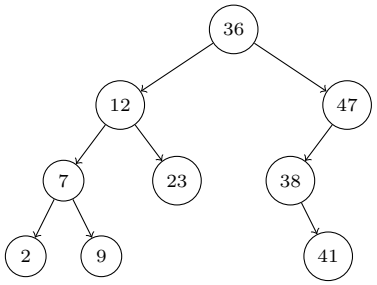


FIGURE 5 – Un arbre binaire de recherche

Question 44.2 La figure 6 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 5.

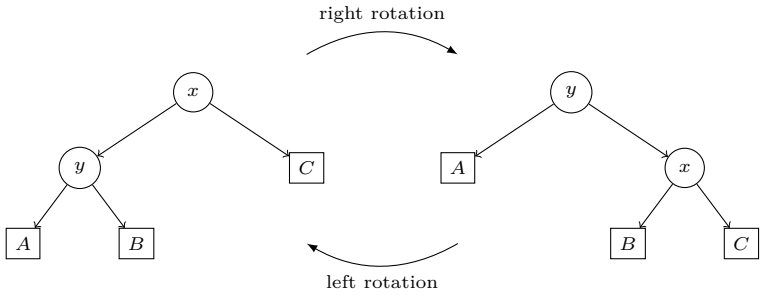


FIGURE 6 – Rotations droite et gauche. x et y sont des nœuds, A, B et C sont des arbres, éventuellement vides

Question 44.3 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 *self);
struct tree *tree_right_rotate(struct tree *self);
```

Question 44.4 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 44.5 Écrire l'algorithme qui permet d'insérer à la racine.

```
struct tree *tree_insert_root(struct tree *self, int data);
```

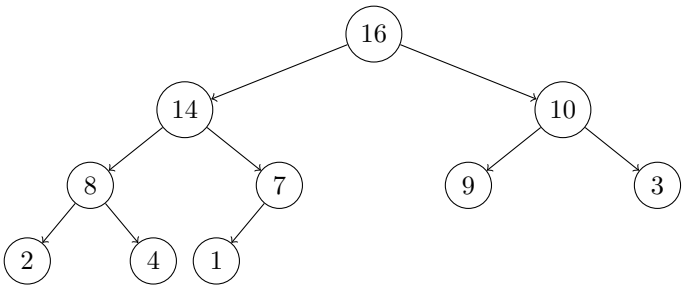
Question 44.6 Quelle est la complexité de cet algorithme ? Justifier.

Question 44.7 Quel peut être l'intérêt d'insérer à la racine ?

Exercice 45 : Propriétés des tas

Question 45.1 Dessiner tous les tas possibles avec les éléments suivants : 1, 4, 7, 9.

Question 45.2 Entasser l'élément 11 dans le tas suivant :



Question 45.3 Supprimer l'élément 14 du tas de la question précédente.

Question 45.4 Donner la représentation en tableau du tas de la question précédente.

Question 45.5 Dans un tas où tous les éléments sont distincts, quelles sont les positions possibles pour le plus petit élément ?

Exercice 46 : 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 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 46.1 Modéliser ce problème par un graphe et proposer une solution.

Exercice 47 : 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 47.1 Donner en le justifiant un exemple de graphe avec quatre nœuds tel que son rayon est égal à son diamètre.

Question 47.2 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 47.3 Justifier que le calcul de l'excentricité pour un nœud x a une complexité en $\mathcal{C}_0 + |V|$.

Question 47.4 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 47.5 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 47.6 Comparer les complexités des deux questions précédentes en fonction de la densité du graphe.

Question 47.7 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 ?

Exercice 48 : 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 7.

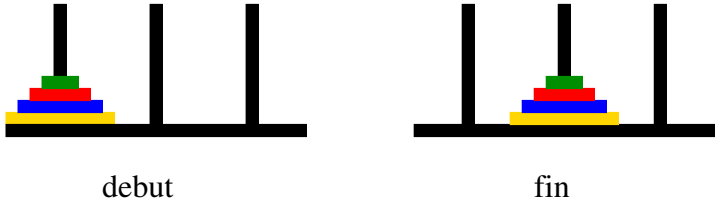


FIGURE 7 – Position de début et de fin pour les tours de Hanoï

Question 48.1 Proposez une solution dans les cas $n = 1, 2, 3, 4$.

Question 48.2 On voit se dessiner une approche récursive. Proposez un algorithme pour ce problème qui va décrire à l'utilisateur les mouvements à effectuer.

Question 48.3 Analysez le nombre de mouvements de disques.

Question 48.4 On considère la structure suivante pour représenter une tour. La structure `struct disc` représente un disque, et en particulier, sa taille (`k`). La structure `struct tower` est alors une pile de disques.

```
struct disc {
  int k;
  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 k);
// 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 48.5 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 48.6 Proposer un algorithme qui affiche le jeu.

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

Question 48.7 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 48.8 Proposer un algorithme qui déplace les n disques de la tour i vers la tour j.

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

Question 48.9 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:
```

Exercice 49 : 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 49.1 Récupérer la taille n sur la ligne de commande.

Question 49.2 Allouer et initialiser un tableau `is_prime[i]` avec des 1.

Question 49.3 Implémenter la méthode du crible d'Ératosthène.

Question 49.4 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 50 : 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 50.1 Saisir un texte grâce à `fgets(3)` en le nettoyant du caractère de passage à la ligne.

Question 50.2 Compter le nombre d'occurrence de chaque caractères du texte.

```
#define CHARS_NUMBER 27

struct statistics {
  int count[CHARS_NUMBER];
};
```

Question 50.3 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 50.4 Appliquer l'algorithme de Huffman décrit précédemment pour contruire l'arbre de Huffman.

Question 50.5 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 50.6 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 ?

Solutions aux exercices

Solution de l'exercice 3

Rien de très compliqué pour ce premier exercice. Voici quelques indications pour la correction.

Réponse à la question 3.1 Il faut bien insister sur le fait de *recopier* et pas de faire un copier-coller depuis le PDF.

Réponse à la question 3.2 Les options ont la signification suivante :

- `-Wall` : active tous les avertissements. Option obligatoire !
- `-std=c99` : active la version C99 du langage C. Permet par exemple de déclarer une variable à l'initialisation d'une boucle.
- `-O2` : active les optimisations de niveaux 2.

Réponse à la question 3.3 On peut faire remarquer l'utilité du passage à la ligne.

Solution de l'exercice 4

Réponse à la question 4.1 Il suffit de compiler le programme grâce à la commande habituelle.

Réponse à la question 4.2

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

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

  int u = atoi(argv[1]);
  printf("%d", u);

  while (u != 1) {
    if (u % 2 == 0) {
      u = u / 2;
    } else {
      u = 3 * u + 1;
    }
    printf(", %d", u);
  }

  printf("\n");
  return 0;
}
```

Réponse à la question 4.3

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

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

  int n = atoi(argv[1]);
  int i;

  for (i = 1; i <= n; ++i) {
    if (i % 3 == 0 || i % 5 == 0) {
      if (i % 3 == 0) {
        printf("Fizz");
      }
```

```
      if (i % 5 == 0) {
        printf("Buzz");
      }
    } else {
      printf("%d", i);
    }

    printf(" ");
  }

  printf("\n");
  return 0;
}
```

Réponse à la question 4.4

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

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

  int n = atoi(argv[1]);
  int i;

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

    for (j = 0; j < i; ++j) {
      printf("# ");
    }

    printf("\n");
  }

  return 0;
}
```

Solution de l'exercice 20

Petite difficulté dans la dernière question : il faut choisir quoi renvoyer s'il n'y a aucun nombre pair dans le tableau. Une solution est de renvoyer `size`.

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

static int *array_new(size_t size) {
  int *data = calloc(size, sizeof(int));

  if (data == NULL) {
    return NULL;
  }

  for (size_t i = 0; i < size; ++i) {
    data[i] = rand() % 100;
  }

  return data;
}

static size_t array_index_max(const int *data, size_t size) {
  assert(data);
  assert(size > 0);

  size_t max = 0;

  for (size_t i = 1; i < size; ++i) {
    if (data[i] > data[max]) {
      max = i;
    }
  }

  return max;
}

static int array_sum(const int *data, size_t size) {
  assert(data);
  assert(size > 0);

  int sum = 0;

  for (size_t i = 0; i < size; ++i) {
    sum += data[i];
  }

  return sum;
}

static size_t array_count(const int *data, size_t size, int value) {
  assert(data);
  assert(size > 0);
```

```
  size_t count = 0;

  for (size_t i = 0; i < size; ++i) {
    if (data[i] == value) {
      count++;
    }
  }

  return count;
}

static void array_shift_left(int *data, size_t size) {
  assert(data);
  assert(size > 0);

  int tmp = data[0];

  for (size_t i = 1; i < size; ++i) {
    data[i - 1] = data[i];
  }

  data[size - 1] = tmp;
}

static size_t array_longest_event_seq(const int *data, size_t size) {
  assert(data);
  assert(size > 0);

  size_t index = size;
  size_t count = 0;
  size_t current_index = 0;
  size_t current_count = 0;

  for (size_t i = 0; i < size; ++i) {
    if (i % 2 == 0) {
      if (current_count == 0) {
        current_index = i;
      }

      current_count++;
    } else {
      if (current_count != 0) {
        if (current_count > count) {
          count = current_count;
          index = current_index;
          current_count = 0;
        }
      }
    }
  }

  return index;
}
```

```
int main() {
  const size_t size = 10000;

  int *data = array_new(size);
  assert(data);

  size_t max = array_index_max(data, size);
  printf("index of max is: %zu\n", max);

  int sum = array_sum(data, size);
  printf("sum: %i\n", sum);

  size_t count = array_count(data, size, 42);
  printf("count of 42: %zu\n", count);

  array_shift_left(data, size);

  size_t index = array_longest_event_seq(data, size);
  printf("longest pair sequence starts at: %zu\n", index);

  return 0;
}
```

Solution de l'exercice 21

```
#include <assert.h>
#include <ctype.h> // for isspace
#include <stdbool.h>
#include <stdio.h>
#include <string.h>

static size_t string_length(const char *str) {
  assert(str);
  size_t size = 0;

  while (str[size] != '\0') {
    size++;
  }

  return size;
}

static size_t string_count_spaces(const char *str) {
  assert(str);
  size_t count = 0;
  size_t i = 0;

  while (str[i] != '\0') {
    if (isspace(str[i])) {
      count++;
    }

    i++;
  }

  return count;
}

static void string_print_no_vowels(const char *str) {
  assert(str);
  size_t i = 0;

  while (str[i] != '\0') {
    char c = str[i];

    if (c != 'a' && c != 'e' && c != 'i' && c != 'o' && c != 'u') {
      printf("%c", c);
    }

    i++;
  }
}

static bool string_well_parenthesized(const char *str) {
  assert(str);
  int paren = 0;
  size_t i = 0;
```

```
  while (str[i] != '\0') {
    char c = str[i];

    if (c == '(') {
      paren++;
    } else if (c == ')') {
      paren--;

      if (paren < 0) {
        return false;
      }
    }

    i++;
  }

  return paren == 0;
}

static int string_binary_value(const char *str) {
  assert(str);
  int value = 0;
  size_t i = 0;

  while (str[i] != '\0') {
    char c = str[i];

    switch (c) {
      case '0':
        value = value * 2;
        break;
      case '1':
        value = value * 2 + 1;
        break;
      default:
        // error
        return 0;
    }

    i++;
  }

  return value;
}

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

  const char *str = argv[1];
  printf("%s\n", str);
```

```
  size_t size = string_length(str);
  size_t size_from_strlen = strlen(str);
  printf("size: %zu (%zu)\n", size, size_from_strlen);

  size_t count = string_count_spaces(str);
  printf("number of spaces: %zu\n", count);

  printf("string without vowels: '");
  string_print_no_vowels(str);
  printf("'\n");

  if (string_well_parenthesized(str)) {
    printf("the string is well parenthesized.\n");
  } else {
    printf("the string is NOT well parenthesized.\n");
  }

  int value = string_binary_value(str);
  printf("binary value: %i\n", value);

  return 0;
}
```

Solution de l'exercice 37

Réponse à la question 37.1 La complexité est de n^{2n}. En effet, chacune des n reines peut être placée sur une des n^2 cases du plateau.

Réponse à la question 37.2 n étant connu et étant une constante, on peut utiliser une matrice.

```
struct queens {
  int board[N][N];
};
```