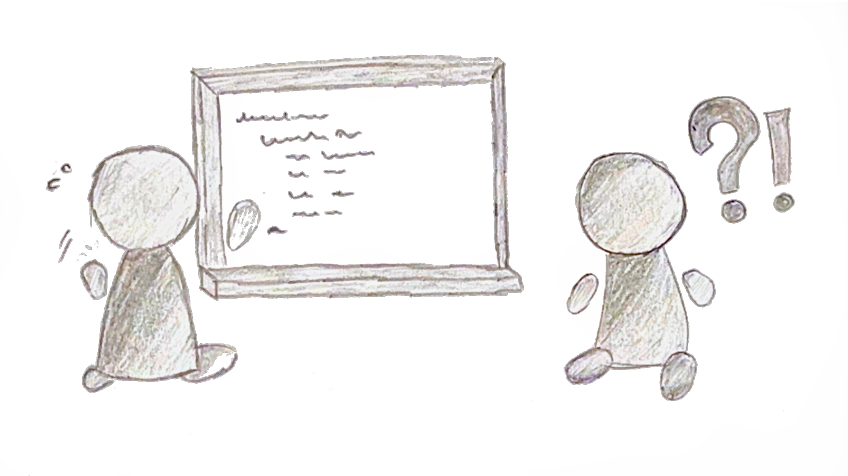
Quelques exercices de colles d'informatique en MP2I

donnés lors de l'année scolaire 2025–2026
dans la classe de Géraldine Olivier au Lycée Montaigne

Rémi Morvan
www.morvan.xyz



Organisation

Ce polycopié d'exercices est intégralement placé sous licence CC BY 4.0. Les exercices sont classés par difficulté (exercices « de cours », d'application et d'approfondissement) ainsi que par thème. La difficulté n'est donnée qu'à titre purement indicative : d'une part celle-ci dépend grandement du moment dans l'année où l'exercice est donné, et d'autre part la difficulté exercices les plus complexes est souvent progressive; ils contiennent donc généralement des questions « de cours ».

Les pages suivantes contiennent d'une part un **sommaire par langage (sous-catégorifié selon la difficulté), ainsi qu'un sommaire par thème abordé**. Par ailleurs, ces exercices ont été donnés entre le mois de décembre et le mois de février : ce polycopié **ne couvre donc pas l'ensemble du programme de MP2I**.

Quand une source est mentionnée, tout ou partie de l'exercice s'inspire directement de cette source, la formulation même de l'exercice m'étant cependant propre. Quand aucune source n'est mentionnée, cela signifie que j'ai écrit cet exercice sans source *directe* : je ne saurais cependant prétendre à aucune forme d'originalité. J'ai par ailleurs essayé, d'autant que faire se peut, de créer des exercices *motivés et motivants*. Par exemple, j'ai essayé de ne jamais demander d'implémenter une structure sans que la suite de l'exercice n'en propose une véritable application. En conséquence, certains exercices sont particulièrement longs.

Ce document a été mis en page avec Typst. Les sources de ce document sont disponibles sur github.com/remimorvan/colles-mp2i. Des corrections imparfaites sont disponibles dans le répertoire `exercices` de ce projet.

Consignes

Chaque sujet était précédé des consignes suivantes.

Apportez du soin à la qualité de vos réponses plus qu'à la quantité, et pensez à écrire des tests. Lisez les exercices dans leur intégralité avant de vous lancer et ayez toujours de quoi écrire devant vous. Tout fichier rendu doit pouvoir être compilé (en C, avec l'option `-Wall`), ou interprété (OCaml) sans erreur ni warning, et contenir des tests codés en dur avec des assert.

Sommaire par langage et difficulté

Exercices en OCaml

Exercices de cours

Exercices d'application

Exercices d'approfondissement

Exercices en C

Exercices de cours

Exercices d'application

Exercices d'approfondissement

Exercices théoriques

Exercices de cours

Exercices d'application

Exercices d'approfondissement

Sommaire par thème

I/O

Arbres

Arithmétique

Complexité

Correction

Dictionnaires

Float

Listes chaînées

Mémoire

Piles/files

Récursivité

Tableaux

Terminaison

Tri

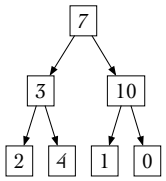
Types

I. Arbres binaires

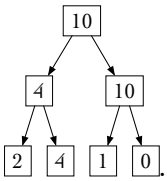
`exercice d'application` `OCaml` `récursivité` `arbres` `complexité`

On s'intéresse à des arbres binaires étiquettés par des entiers. Ce sont des structures, définies récursivement : un tel arbre est soit une simple feuille, à qui on a associé un entier, soit un noeud interne, à qui on associe un entier (son étiquette), ainsi que deux arbres, appelés *fils gauche* et *fils droit*.

1. Définir un type `tree` permettant de représenter cette structure.
2. Définir une fonction
   `max_of_tree: tree -> int`
   qui calcule la plus grande étiquette apparaissant sur un arbre.
3. Définir une fonction
   `max_of_subtrees_quad: tree -> tree`
   qui prend un arbre, et retourne un nouvel arbre obtenu à partir du premier en remplaçant l'étiquette d'un nœud par la plus grande valeur apparaissant sous ce nœud.[1] Par exemple, sur l'entrée



la fonction `max_of_subtrees_quad` retournera



4. Déterminez les complexités spatiales et temporelles de votre fonction `max_of_subtrees_quad`. Votre fonction est-elle récursive terminale ?
5. Si votre fonction n'a pas une complexité temporelle linéaire, donnez-en une nouvelle version pour qui c'est le cas.

II. Types sommes et produits

`exercice de cours` `OCaml` `types`

On souhaite définir un type `locality` qui représente un échelon du maillage territorial français : pour faire simple, on veut seulement représenter soit des communes (représentée par un entier à cinq chiffres, qui est leur code INSEE), soit des départements (représentés par un entier à deux chiffres). Par exemple, le code INSEE de Bordeaux est le 33063, et le numéro de département de la Gironde est le 33.

1. Écrire un type somme `locality` correspondant à la description précédente.
2. Écrire une fonction `is_part_of` de type `locality -> locality -> bool` qui détermine si 1. la première localité est une commune, 2. la seconde est un département, et 3. la commune appartient au département. On admettra que les deux premiers chiffres du code INSEE d'une commune correspondent à son numéro de département.

III. Un problème de mémoire

`exercice de cours` `C` `mémoire` `tableaux`

1. Le code suivant a un comportement indéterminé. Pourquoi ?

[1] Par « sous ce nœud » on veut dire formellement « dans le sous-arbre enraciné en ce nœud ».

```c
#include <stdio.h>

int *foo(int step) {
    int p[9] = {0};
    for (int i = 0; i < 9; i += step) {
        p[i] = 1;
    }
    return p;
}

int main(int argc, char *argv[]) {
  int *p = foo(2);
  printf("%d\n", p[4]);
  free(p);
  return 0;
}
```

2. Corrigez le code de la fonction `foo`, puis dessinez l'état de la mémoire juste avant l'exécution de l'instruction `return p;`.

IV. Min-max d'un tableau

exercice de cours C tableaux

Source : Cours de C de Floréal Morandat à l'ENSEIRB.

On souhaite, étant donné un tableau, calculer son minimum et son maximum. Écrire une fonction `void min_max(int l, int t[], int *min, int *max)` réalisant ce calcul. L'entrée `l` représente la taille du tableau `t`, et `min` et `max` sont des pointeurs vers les cases mémoires où l'on souhaite stocker le résultat. Vous être libres de choisir le comportement de cette fonction si le tableau est vide.

V. Retournement de chaîne

exercice d'application C tableaux mémoire

Source : Cours de C de Floréal Morandat à l'ENSEIRB.

1. Écrire une fonction
   `void reverse(char* str, char* str_rev)`
   qui prend deux chaînes de caractères `str` et `str_rev`, et qui écrit sur `str_rev` le miroir de `str`. Le miroir est obtenu en lisant la chaîne de droite à gauche. Par exemple, le renversé de « Hello world! » est « !dlrow olleH ». On supposera que la chaîne `str_rev` est de taille suffisante pour stocker le résultat.

   Vous pourrez tester votre code avec la fonction `main` suivante.

```c
int main(int argc, char *argv[]) {
    char str[100] = "Hello world!";
    char str_rev[100] = "";
    reverse(str, str_rev);
    printf("%s\n", str_rev);
    return 0;
}
```

2. Dessinez l'état de la mémoire avant l'appel de `reverse`, puis juste avant le retour de la fonction `reverse`.

VI. Types enregistrement

`exercice de cours` `OCaml` `types`

On souhaite créer un type enregistrement `neighbourhood` qui permette de représenter une zone géographique, correspondant à un code postal et à un nom de commune.

1. Définir un type `neighbourhood` de sorte que l'instruction suivante soit valide.

```
let bdx = {
  postal_code = 33000;
  city = "Bordeaux";
}
```

2. Écrire une fonction `string_of_neighbourhood` de type `neighbourhood -> string` qui associe à chaque zone une chaîne de caractères la représentant. Par exemple, `string_of_neighbourhood bdx` retournera « 33000 Bordeaux ».

VII. Uno à une joueuse

`exercice d'approfondissement` `OCaml` `récursivité` `types` `complexité`

On s'intéresse à une variante du jeu de cartes Uno, à une joueuse. Le jeu dispose de deux types de cartes :

- des cartes de valeurs, qui ont toute une couleur (rouge, jaune, vert ou bleu) et un chiffre,
- des cartes de changement de couleur.

Dans cette variante du jeu, il y a une carte initialement posée sur la table, et la joueuse a en main un ensemble de cartes. Son but est de déposer l'ensemble de ses cartes sur la pile. Pour poser une carte sur la pile, il faut respecter les règles suivantes :

- on peut toujours poser une carte de changement de couleur,
- si la carte au sommet de la pile est un changement de couleur, on peut poser la carte qu'on souhaite,
- sinon, si une carte de valeur est au sommet de la pile et qu'on souhaite poser une carte de valeur, il faut soit que leur couleur coïncide, soit que leur chiffre coïncide.

Le but de l'exercice est, étant donnée une main de départ, de déterminer si une joueuse a une stratégie lui permettant de vider entièrement sa main.

1. Définir un type `card` permettant de représenter les cartes de ce jeu.
2. Montrer que le problème est non-trivial : donner un (tout petit) exemple où la joueuse peut vider sa main, et un (tout petit) exemple où elle ne peut pas le faire.
3. Définir une fonction
   `is_playable: card -> card -> bool`,
   qui prend la dernière carte jouée et une carte que l'on souhaite jouer, et qui retourne si on peut effectivement jouer cette carte.
4. On veut implémenter ici une stratégie *gloutonne* : on va regarder la liste des cartes qu'on a en main, et on va jouer la première carte que l'on peut jouer. On réitère ce processus jusqu'à ce que l'on ne puisse plus jouer, c'est-à-dire soit jusqu'à ce qu'on n'ait plus de carte en main (victoire !), soit jusqu'à ce qu'aucune de nos cartes ne soit jouable. Définir une fonction récursive terminale
   `greedy_play : card -> card list -> card list -> int`
   qui prend en entrée la dernière carte jouée, et la main de la joueuse (séparée en deux listes), et qui

retourne le nombre de cartes restant dans la main de la joueuse après avoir appliqué cette stratégie gloutonne jusqu'à ne plus pouvoir jouer. Les deux listes de cartes représentent respectivement les cartes que l'on a déjà essayé, sans succès, de jouer sur la pile actuelle, et les cartes que l'on a pas encore essayé de jouer.

5. Quelles sont les complexités temporelles et spatiales, dans le pire cas, de cette fonction ?
6. Montrez que cette stratégie n'est pas optimale, c'est-à-dire qu'il existe une main initiale telle qu'en utilisant la stratégie gloutonne, on se retrouve à ne plus pouvoir jouer en ayant encore des cartes en main, alors qu'il existe une autre stratégie permettant de vider sa main.
7. Implémenter une fonction récursive
   `optimal_play : card -> card list -> int` qui prend en entrée la dernière carte jouée, et la main de la joueuse, et qui retourne le nombre de cartes restant dans la main de la joueuse après avoir appliqué une stratégie optimale.

VIII. Nombre de voyelles

`exercice d'application` `C` `tableaux`

Source : Cours de C de Floréal Morandat à l'ENSEIRB.

On souhaite déterminer le nombre de voyelles dans une chaîne de caractères en C.

1. Écrire une fonction
   `int in_array(char x, int n, char arr[])`
   qui détermine si un caractère `x` est présent dans un tableau de caractères `arr` de taille `n`.
2. **En déduire** une fonction
   `int is_vowel(char c)`
   qui détermine si un caractère est une voyelle.
   (Votre code doit tenir en deux lignes, sinon c'est que vous vous compliquez la vie.)
3. En déduire une fonction
   `int nb_vowels(char *str)`
   qui retourne le nombre total de voyelles contenues dans une chaîne de caractères.

IX. Fusion de tableaux triés

`exercice d'application` `C` `tableaux` `tri`

Écrire une fonction
`int* merge_sorted(int arr1[], int n1, int arr2[], int n2)`
prenant deux tableaux **triés** et leur taille, et retournant un tableau résultant de la fusion de ces deux tableaux. Plus précisément, on souhaite que l'ensemble des nouveaux éléments de ce tableau soit l'union des deux tableaux passés en entrée, et que le nouveau soit trié. Votre algorithme devra être en temps linéaire en la taille de l'entrée.

X. Un brin de génétique

`exercice d'approfondissement` `OCaml` `récursivité` `dictionnaires`

L'acide ribonucléique messager (ARNm) est une molécule qui intervient dans la synthèse des protéines à partir de l'ADN. On peut voir un brin d'ARNm comme un mot sur l'alphabet $\{A, U, G, C\}$,

chaque lettre étant appelée **nucléotide**.[2] L'algorithme pour synthétiser une protéine à partir d'un brin ARNm est le suivant :

- on regroupe les nucléotides par groupe de trois ; chaque groupe de trois nucléotides (par exemple « AUG » et « CGA ») est appelé « codon » ;
- chaque codon produit un acide aminé (par exemple `Phe` (*phénylalanine*), ou `Lys` (*lysine*)), ou indique le début de la synthèse (`init`), ou l'arrêt de la synthèse (`stop`). Un codon qui indique le début de la synthèse est aussi toujours associé à un acide aminé. Par exemple, « AUG » indique le début de la synthèse et est associé à l'acide aminé « Met ». La première fois qu'on le rencontrera, celà initiera la synthèse de la protéine, mais la seconde fois, il produira l'acide aminé « Met ».[3]

Considérons par exemple le brin d'ARN messager

CUCAUGCAGAGUAUGUGAAGCCCCUUC.

Ses codons sont

$$\underbrace{\text{CUC}}_{\texttt{Leu}}\ \underbrace{\text{AUG}}_{\texttt{init/Met}}\ \underbrace{\text{CAG}}_{\texttt{Gln}}\ \underbrace{\text{AGU}}_{\texttt{Ser}}\ \underbrace{\text{AUG}}_{\texttt{init/Met}}\ \underbrace{\text{UGA}}_{\texttt{stop}}\ \underbrace{\text{AGC}}_{\texttt{Ser}}\ \underbrace{\text{CCC}}_{\texttt{Pro}}\ \underbrace{\text{UUC}}_{\texttt{Phe}}.$$

Lors de la synthèse, on obtiendra donc la protéine définie par la suite d'acide aminé `Gln Ser Met`. Le but de cet exercice est de simuler cette synthèse en OCaml.

1. Définir une fonction
   `rna_to_codon_list: string -> string list` qui associe à un brin d'ARNm la liste de ses codons.
2. Écrire des fonctions

   ```
   is_codon_init: string -> bool
   is_codon_stop: string -> bool
   ```

   qui détermine si un codon est initiant ou stopant. Les codons initiants sont UUG, CUG et AUG; les codons stopant sont UAA, UAG et UGA.
3. On souhaite définir une structure qui décrit l'information de quel acide aminé est associé à un codon donné : on va utiliser une **liste d'association**. Une liste d'association est une liste de type `('a * 'b) list`, de sorte que si (c, v) et (c', v') sont tous deux éléments de la liste, alors $c \neq c'$. Autrement dit, une liste d'association encode une fonction qui associe des éléments de type `'a` (appelés **clés**) à des éléments de type `'b` (appelés **valeurs**).
   a. Définir une fonction
      `is_defined: ('a * 'b) list -> 'a -> bool`
      qui prend une liste d'association, une clé, et détermine si une valeur lui est associée.
   b. Définir une fonction
      `value_of: ('a * 'b) list -> 'a -> 'b`
      qui prend une liste d'association, une clé, et retourne la valeur qui lui est associée (si elle existe, sinon elle produit une erreur).
   c. À partir de la liste d'association de type `(string * string) list`

      ```
      let codon_to_amino_acid_data = [ ("UUU", "Phe") ; ("UUC", "Phe") ; ("UUA",
      "Leu") ; ("UUG", "Leu") ; ("CUU", "Leu") ; ("CUC", "Leu") ; ("CUA", "Leu") ;
      ("CUG", "Leu") ; ("AUU", "Ile") ; ("AUC", "Ile") ; ("AUA", "Ile") ; ("AUG",
      "Met") ; ("GUU", "Val") ; ("GUC", "Val") ; ("GUA", "Val") ; ("GUG", "Val") ;
      ("UCU", "Ser") ; ("UCC", "Ser") ; ("UCA", "Ser") ; ("UCG", "Ser") ; ("CCU",
      "Pro") ; ("CCC", "Pro") ; ("CCA", "Pro") ; ("CCG", "Pro") ; ("ACU", "Thr") ;
      ("ACC", "Thr") ; ("ACA", "Thr") ; ("ACG", "Thr") ; ("GCU", "Ala") ; ("GCC",
      ```

[2] Les lettres A, U, G et C font respectivement référence aux nucléotides adénine, uracile, guanine et cytosine.
[3] Bien évidemment, tout ceci est une simplification de la réalité.

```
"Ala") ; ("GCA", "Ala") ; ("GCG", "Ala") ; ("UAU", "Tyr") ; ("UAC", "Tyr") ;
("CAU", "His") ; ("CAC", "His") ; ("CAA", "Gln") ; ("CAG", "Gln") ; ("AAU",
"Asn") ; ("AAC", "Asn") ; ("AAA", "Lys") ; ("AAG", "Lys") ; ("GAU", "Asp") ;
("GAC", "Asp") ; ("GAA", "Glu") ; ("GAG", "Glu") ; ("UGU", "Cys") ; ("UGC",
"Cys") ; ("UGG", "Trp") ; ("CGU", "Arg") ; ("CGC", "Arg") ; ("CGA", "Arg") ;
("CGG", "Arg") ; ("AGU", "Ser") ; ("AGC", "Ser") ; ("AGA", "Arg") ; ("AGG",
"Arg") ; ("GGU", "Gly") ; ("GGC", "Gly") ; ("GGA", "Gly") ; ("GGG", "Gly") ];;
```

définir une fonction `codon_to_amino_acid: string -> string` qui retourne l'acide aminé associé à un codon passé en argument.

4. Le but de cette question est de produire une fonction `synthesis: string -> string list` qui prend une chaîne de caractère représentant un brin d'ARN messager, et retourne la chaîne d'acide aminés de la protéine produite par ce brin.
   a. Donnez une description haut-niveau d'un tel algorithme. *On s'attend à une réponse de quelques lignes, en français (pas de pseudo-code), expliquant le fonctionnement de cet algorithme. Il est inutile de détailler commenter calculer les fonctions implémentées aux questions précédentes.*
   b. Implémentez la fonction `synthesis`.

XI. Tri d'un tableau 0/1 en place

exercice d'application · C · tableaux · tri · complexité

Source : Exercices 25, 27 et 28 du livre « Informatique - MP2I/MPI - CPGE 1re et 2e années - Cours et exercices corrigés », de Balabonski Thibaut, Conchon Sylvain, Filliâtre Jean-Christophe, Nguyen Kim, Sartre Laurent.

1. Écrire une fonction
   `void swap(int arr[], int i, int j)`
   qui échange les éléments n°`i` et `j` du tableau `arr`.
2. Écrire une fonction
   `twoway_sort(int arr[], int n)`
   qui prend en entrée un tableau et sa taille, et qui le **trie en place**. **On supposera que le tableau ne contient que les valeurs 0 et 1**, et la seule opération qui vous est permise est la fonction `swap` de la question précédente. La complexité temporelle de votre algorithme doit être au pire linéaire.
3. **Question bonus, à ne faire que si vous avez terminé tout le reste de la feuille.**
   Écrire une fonction
   `dutch_flag(int arr[], int n)`
   qui prend en entrée un tableau et sa taille, et qui le **trie en place**. On supposera que le tableau ne contient que les valeurs 0, 1 ou 2, et la seule opération qui vous est permise est la fonction `swap` de la question précédente. La complexité temporelle de votre algorithme doit être au pire linéaire.
   Cette question nécessite une réflexion algorithmique non-triviale. Plus encore que pour les autres questions, ***faites des dessins sur une feuille****.*

XII. Un peu de géométrie du plan

exercice d'application · OCaml · types · float

On souhaite représenter des points, des cercles, et des disques en OCaml. Un point sera représenté par une paire d'abscisse et d'ordonnée (qui seront des `float`), un cercle par un point (son centre) et un rayon (un `float`), et un disque par les mêmes informations.

1. Définir un type `point` représentant un point.

2. Définir un type `shape` représentant un objet géométrique (soit un cercle, soit un disque).
3. Écrire une fonction
   `belongs_to: point -> shape -> bool`
   qui détermine si un point appartient à un objet.[4]

XIII. Le cycle des quintes

`exercice d'application` `C` `mémoire` `listes chaînées`

Source : L'interface de la structure est adaptée du TP de Géraldine Olivier sur les listes chaînées en C.

On veut ici implémenter une structure de **liste circulaire doublement chaînée**. Concrètement, contrairement à une liste chaînée, cette structure est circulaire (un élément a toujours un successeur), et on veut aussi pouvoir accéder au prédécesseur d'un élément de la liste. Un exemple est donné en Fig. 1. Notons qu'une telle liste a toujours un élément spécial, appelé « élément de tête ».

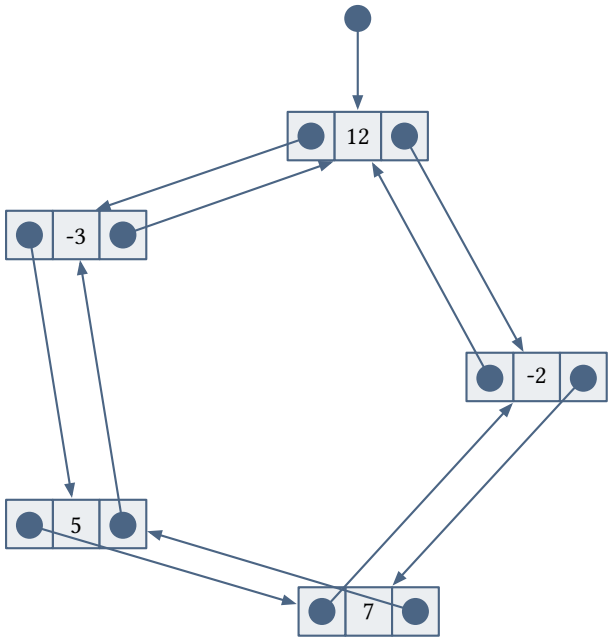


Fig. 1. – Une liste circulaire doublement chaînée.

On propose d'implémenter cette structure avec des **maillons**, chaque **maillon** contenant une valeur (des entiers sur la Fig. 1), un pointeur vers l'élément **suivant**, et un pointeur vers l'élément **précédent**. **Vous avez le droit, et êtes même très vivement encouragé·e·s, de travailler à partir de la correction du TP sur les listes chaînées.**

Voici l'interface abstraite de la structure.

```
// Constructeur
list    *list_create();

// Accesseurs
int      list_length(list *l);
bool     list_is_empty(list *l);
void     list_print(list *l);
elt_type list_get_ith(list *l, int i);
```

[4] On rappelle qu'un disque est plein, contrairement à un cercle. Par exemple, le point de coordonnées $\left(1, \frac{1}{2}\right)$ appartient au disque de centre $(0, 0)$ et de rayon 2.

```
// Transformateurs
void     list_set_ith(list *l, int i, elt_type v);
void     list_insert_ith(list *l, int i, elt_type v);
elt_type list_rotate(list *l, int i);
elt_type list_delete_ith(list *l, int i);
void     list_delete_all(list *l);

// Destructeur
extern void list_free(list **addr_l);
```

Notons par ailleurs que **l'entier `i` peut désormais être négatif** : par exemple, le (-3)-ème élément d'une liste circulaire doublement chaînée, c'est le prédécesseur du prédécesseur du prédécesseur de l'élément de tête. Il n'y a pas ici de fonction de concaténation.[5] On a cependant rajouté une nouvelle opération `list_rotate` : celle-ci fait rotationner la liste, c'est-à-dire qu'elle déplace l'élément de tête de la liste. Elle retourne par ailleurs le nouvel élément de tête. Par exemple, sur la Fig. 1, une rotation de –2 (ou de 3) ferait que 5 serait le nouvel élément de tête.

1. Implémentez les fonctions de l'interface. Après chaque fonction, **compilez et testez**.
2. Application.
   a. Créez une liste circulaire doublement chaînée `chromatic_scale` dont les éléments sont, dans l'ordre, les chaînes de caractères « do », « do# », « ré », « ré# », « mi », « fa », « fa# », « sol », « sol# », « la », « la# » et « si ».
   b. Écrire une fonction
   `list* list_walk(list *l, int step)`,
   qui, à partir d'une liste circulaire doublement chaînée, retourne une structure du même type, obtenue en lisant les éléments à partir de l'élément de tête par pas de `step`, jusqu'à retomber sur l'élément de tête. Ce pas pourra être positif, négatif ou nul.
   c. En musique, le **cycle des quintes** est une liste circulaire obtenue à partir de l'échelle chromatique en la parcourant par pas de 7.[6] Calculez et affichez ce cycle.

XIV. Recherche dichotomique

`exercice de cours` `OCaml` `récursivité` `tableaux`

Écrire une fonction

```
dichotomy_search: 'a -> 'a array -> bool
```

déterminant si un élément appartient à un tableau **trié**. Votre algorithme devra utiliser une fonction auxiliaire qui sera **récursive terminale**, et qui sera basée sur le principe de la dichotomie.

XV. Exponentiation rapide

`exercice de cours` `OCaml` `récursivité` `complexité` `terminaison` `arithmétique`

1. Implémentez un algorithme d'exponentiation rapide.
2. Justifiez sa terminaison.
3. Quelle est sa complexité temporelle dans le pire cas ?

[5] On pourrait toutefois donner un sens à une telle opération si on le souhaitait.

[6] Un intervalle de 7 demi-tons (l'intervalle élémentaire séparant ‹ do › de ‹ do# ›, ou encore ‹ mi › de ‹ fa ›) est appelé **quinte**, d'où le nom « cycle des quintes ».

XVI. Tri d'un tableau borné

`exercice d'application` `C` `tri` `tableaux` `complexité`

Le but de cet exercice est de trier un tableau d'entiers **positifs**, lorsque l'on connaît une **borne (strictement) supérieure b sur les entrées du tableau**. L'idée de l'algorithme est la suivante : on va créer un nouveau tableau `count` provisoire, de taille b, qui va compter le nombre d'occurrences de chaque élément : plus précisément `count[i]` (pour $i \in [\![0, B[\![$) sera le nombre d'éléments du tableau d'entrée égaux à i. À partir de ce tableau, on pourra alors trier le tableau d'origine simplement en réécrivant les éléments dans le bon ordre.

1. Écrire une fonction
   `void bounded_sort(int arr[], int n, int b)`
   qui prend en entrée un tableau `arr` de taille `n`, dont les entrées sont toutes comprises entre 0 (inclus) et b (exclus), et qui trie le tableau `arr` avec la méthode décrite précédemment.
2. L'hypothèse que l'on connaît une borne supérieure b est-elle contraignante ? Comment s'en débarrasser ? Implémentez votre solution dans une fonction
   `void bounded_sort_bis(int arr[], int n)`.
3. Quelle est la complexité en temps et en espace de cet algorithme, dans le pire cas ? Dans le meilleur cas ?
4. Pour quels types de tableaux cet algorithme est-il bien plus intéressant à utiliser plutôt qu'un algorithme de tri par comparaison s'exécutant au pire cas en temps $O(n \log(n))$? *La réponse attendue n'est pas mathématique, mais une description en quelques mots de la « forme » des données.*

XVII. Couplage de Cantor

`exercice de cours` `OCaml` `récursivité`

Le but de cet exercice est de calculer la bijection f de $\mathbb{N}^2$ dans $\mathbb{N}$ représentée en Fig. 2, appelée « couplage de Cantor ». L'idée derrière cette bijection est simplement d'énumérer les paires d'entiers par diagonale. Au sein d'une diagonale, on énumère les paires selon les x croissants. Ainsi, on a par exemple $f(0,0) = 0$, $f(0,1) = 1$ et $f(1,0) = 2$, comme représenté en Fig. 2.

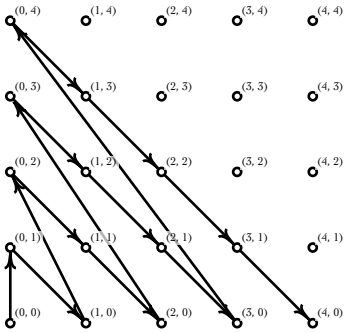


Fig. 2. – Une bijection de $\mathbb{N}^2$ dans $\mathbb{N}$.

1. Définir une fonction **récursive** `bij : int * int -> int` (en OCaml) calculant f.
2. Vérifiez empiriquement que

$$f(x,y) = \frac{(x+y)(x+y+1)}{2} + x \quad \text{pour} \quad x, y \in \mathbb{N}.$$

Pour la culture : Le couplage de Cantor a été introduit en 1873 par Georg Cantor. Vous remarquerez que la fonction f est un polynôme quadratique : en 1923, Rudolf Fueter et Georg Pólya publient le théorème de Fueter–Pólya, qui affirme que f, et son symétrique $(x,y) \mapsto f(y,x)$, sont les **seules**

fonctions quadratiques qui réalisent une bijection de $\mathbb{N}^2$ dans $\mathbb{N}$. La conjecture Fueter–Pólya affirme elle que ce sont les **seules fonctions polynomiales** satisfaisant cette propriété. À ce jour, la conjecture est encore ouverte !

XVIII. Arbres de décision

`exercice d'approfondissement` `OCaml` `récursivité` `arbres`

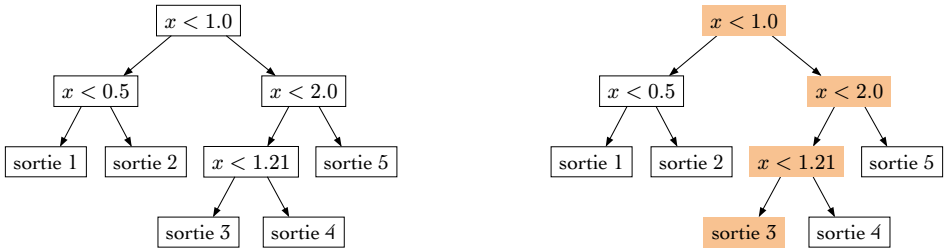


Fig. 3. – Un arbre de décision (à gauche) et nœuds visités par l'algorithme d'évaluation de l'arbre sur l'entrée $x = 1.13$ (à droite).

Un arbre de décision est un arbre tel que celui représenté en Fig. 3 : il représente un algorithme prenant en entrée une variable x de type `float`, et retournant une sortie de type `'a`—sur l'exemple de Fig. 3, le type de sortie est `string` puisque ces sorties sont « sortie 1 », …, « sortie 5 ». Les feuilles de cet arbre (les nœuds qui n'ont pas de fils) correspondent tous à des sorties. Les nœuds internes (ceux avec des fils) sont eux étiquetés par des tests de la forme $x <$ cst où cst est un flottant. Chaque nœud interne a exactement deux fils, appelés *fils gauche* et *fils droit*.

Un arbre représente une fonction de type `float -> 'a` (appelée **sémantique** de l'arbre), définie de la façon récursive. Étant donné un flottant x, la sortie de cette fonction est obtenue en commençant à la *racine* de l'arbre. Si cette racine est une sortie, c'est notre valeur de retour. Sinon, c'est un nœud interne, qui est donc étiqueté par un test de la forme $x <$ cst : c'est par exemple le cas sur la Fig. 3, et le test à la racine est $x < 1.0$. Si le test est satisfait, on poursuit l'exécution de notre algorithme en allant dans le *fils gauche* du nœud, et sinon dans le fils droit. Par exemple, si $x = 1.13$, sur l'arbre de Fig. 3, la sortie sera « sortie 3 » : l'exécution de l'algorithme est représentée sur la moitié droite de Fig. 3.

1. On définit ces arbres en OCaml de la façon suivante :

```
type 'a decision_tree_univariate =
  | TestUni of float * 'a decision_tree_univariate * 'a decision_tree_univariate
  | OutputUni of 'a;;
```

   Les trois arguments du constructeur `TestUni` correspondent respectivement à la constante avec laquelle on compare x, le fils gauche du nœud, et son fils droit.
   a. Définir un arbre de décision `some_tree` de type `string decision_tree_univariate` correspondant à l'arbre de la Fig. 3.
   b. Écrire une fonction récursive
   `eval_univariate: 'a decision_tree_univariate -> float -> 'a`
   qui prend un arbre de décision, un flottant, et évalue la fonction définie par cet arbre sur ce flottant.
2. Les vrais arbres de décision ne manipulent en réalité pas qu'une seule variable mais plusieurs. Par convention et souci de simplicité, on nommera ces variables $x_0, x_1, \ldots, x_{n-1}$ ($n \in \mathbb{N}$ étant le nombre total de variables). Les tests sont désormais de la forme $x_i <$ cst ($i \in [\![0, n[\![$) : on ne peut

pas comparer des variables entre elles, mais on peut comparer n'importe quelle variable avec une constante.

a. Définir un type `'a decision_tree` permettant de représenter ces arbres de décision à plusieurs variables. *Indice : On pourra représenter la variable x_i par l'entier i.*

b. Dans le cas $n = 2$ (les variables sont donc x_0 et x_1), définir un arbre `quarter_planes` de type `string decision_tree` qui retourne « NE » (north-east), « NW » (north-west), « SW » (south-west), « SE » (south-east) selon la position du point (x_0, x_1) dans le plan : par exemple le quart de plan « NE » correspond aux points $[0, +\infty[\times [0, +\infty[$, alors que le quart de plan « SE » correspond aux points $[0, +\infty[\times]-\infty, 0[$.

c. On souhaite maintenant écrire un algorithme pour évaluer ces arbres. Dans le cas univarié, l'entrée était représentée par un flottant. Dans notre cas, pour représenter une entrée $(x_0, x_1, ..., x_{n-1})$, on va utiliser un `float array`. Écrire une fonction récursive
`eval: 'a decision_tree -> float array -> 'a`
qui prend un arbre de décision, un tableau de flottants, et évalue la fonction définie par cet arbre sur ce tableau. Pour rappel, la taille d'un tableau `arr` peut être obtenue avec `Array.length arr`, et la i-ème entrée de ce tableau avec `arr.(i)`. On peut définir un tableau à l'aide de la syntaxe `[| x0 ; x1 ; ... |]`.

d. Justifiez brièvement la terminaison de votre fonction `eval`. Donnez des bornes supérieures (raisonnables) sur la complexité temporelle et spatiale de cette même fonction.

Pour la culture : Arbres de décision & apprentissage

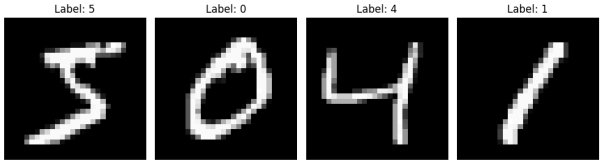


Fig. 4. – Images, représentant des chiffres, issues du jeu de données MNIST.

Les arbres de décision sont au cœur de plusieurs techniques de *machine learning*. Nous illustrons cela sur un exemple de classification d'images, de 28×28 pixels, en noir et blanc, représentant des chiffres, voir la Fig. 4. Une telle image est représentée par un tableau de flottants de taille 784 ($= 28 \cdot 28$). La i-ème entrée représente la couleur du i-ème pixel : 0.0 est un pixel parfaitement blanc, et 1.0 représente un pixel parfaitement noir.

Le but d'un algorithme d'*apprentissage supervisé*, est, à partir d'un grand jeu d'exemples (c'est-à-dire d'images, munies de la sortie attendue, c'est-à-dire ici du chiffre représenté sur l'image), d'apprendre une *fonction* qui prend en entrée une telle image, et retourne le chiffre indiqué dessus. Bien sûr, la difficulté ne réside pas tant dans le fait de retourner la bonne réponse sur les données sur lesquelles on a appris, mais de retourner la bonne réponse sur d'autres données…

Un arbre de décision est une façon naturelle et simple de représenter une telle fonction, dont les sorties sont de type `int` (plus précisément, elles sont dans l'intervalle $[\![0, 9]\!]$.) Souvent, apprendre un arbre de décision se révèle être relativement peu efficace (l'arbre est généralement très grand, et la fonction apprise n'est pas toujours très satisfaisante). Une technique un peu plus raffinée, appelée **forêts aléatoires** a été développée à la fin des années 1990s, pour pallier certains désavantages des arbres de décision. L'idée est de, plutôt que d'apprendre **un seul grand arbre** de décision, de plutôt apprendre **plusieurs petits arbres**—ce qui justifie le nom de « forêt ». Pour que ces arbres soient distincts les uns des autres, un facteur aléatoire est introduit, en limitant artificiellement (et aléatoirement) quels pixels peuvent être utilisés dans un test. Les forêts aléatoires sont, bien que relativement

simples, terriblement efficaces sur certains problèmes d'apprentissage : c'est notamment le cas pour le problème de classification des chiffres sur le jeu de données MNIST (Fig. 4).

XIX. Triangle de Sierpiński

exercice d'approfondissement | C | tableaux | récursivité | complexité | terminaison | I/O

Le **triangle de Sierpiński** est une figure fractale, représentée sur la Fig. 5, et découverte en 1915 par le mathématicien Wacław Sierpiński. C'est probablement l'une des figures fractales les plus simples à construire, grâce à son étonnant rapport avec les coefficients binomiaux. Le but de cet exercice est d'afficher cette fractale.

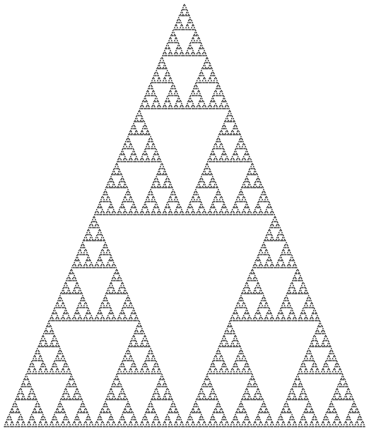


Fig. 5. – Le triangle de Sierpiński.

1. En utilisant la formule de Pascal, qu'on rappelle être
$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$
pour $n \in \mathbb{N}$ et $k \in [\![1, n-1]\!]$, écrire une fonction récursive, la plus simple possible,
`int binom_naive(int k, int n)`
qui calcule $\binom{n}{k}$, sous réserve que $n \geq 0$ et $k \in [\![0, n]\!]$. On ne cherchera **pas** à optimiser cette fonction.
2. Justifiez que cette fonction termine, puis donnez une borne supérieure (raisonnable) sur la complexité temporelle de `binom_naive(int k, int n)`, en fonction de n.
3. Écrire une fonction
`void print_sierpinski_naive(int n)`
qui prend en entrée un entier n, et affiche $n+1$ lignes du triangle de Pascal, c'est-à-dire que sur la m-ième ligne ($m \in [\![0, n]\!]$), on affichera les coefficients binomiaux $\binom{m}{0}, \binom{m}{1}, ..., \binom{m}{m}$. Vous devriez remarquer que votre fonction est relativement lente à s'exécuter dès que n s'approche de ~20.
4. **Modifiez** la fonction précédente pour qu'au lieu d'afficher l'entier $\binom{m}{k}$, elle affiche le caractère « ▉ » si $\binom{m}{k}$ est impair, et le caractère espace sinon. Tada !
5. Tout ceci est bien joli, mais fort peu efficace. Pour améliorer notre algorithme, on va partir de l'observation que si l'on connaît tous les coefficients binomiaux de la forme $\binom{n-1}{\_}$, alors on peut calculer $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ en temps constant ($n \in \mathbb{N} \setminus \{0\}, k \in [\![0, n]\!]$). Écrire une fonction
`int *binom_list(int n, int *coefs_prec)`
qui prend un entier $n \in \mathbb{N}$, et le tableau des coefficients binomiaux d'ordre $n-1$, càd le tableau $\left\{\binom{n-1}{0}, \binom{n-1}{1}, ..., \binom{n-1}{n-1}\right\}$, et qui retourne le tableau des coefficients binomiaux d'ordre n. Lorsque $n = 0$, on pourra supposer que le pointeur passé en entrée est `NULL`.

6. Donnez une borne supérieure sur la complexité temporelle de votre fonction `binom_list`, en fonction de n.
7. En déduire une fonction
   `void print_sierpinski(int n)`
   qui affiche les lignes 0 à n du triangle de Sierpiński, calculée avec la fonction `binom_list`. Vous ferez particulièrement attention à ne pas avoir de fuite mémoire.
8. **Bonus (à faire seulement si tous les autres exercices sont finis, ou chez vous) :**
   Améliorer votre fonction `print_sierpinski` pour que votre figure ressemble à celle de la Fig. 5 (càd pour que la pointe de votre triangle soit centrée et non alignée à gauche). Faire en sorte que pouvoir passer en argument à l'exécutable le nombre de lignes du triangle que l'on souhaite afficher. Pour rappel, dans le prototype
   `int main(int argc, char *argv[])`
   `argc` désigne le nombre d'arguments passés à l'exécutable, et `argv` est un tableau de chaînes de caractères contenant ces arguments.

XX. Tri fusion

`exercice de cours` `OCaml` `tri` `récursivité` `complexité`

On rappelle que le tri fusion est un algorithme de tri récursif dont le principe est le suivant : on découpe la liste en deux, on trie chaque moitié (récursivement), puis on fusionne les deux résultats.

Voici une implémentation partielle de cet algorithme.

```
let rec split lst = match lst with
  (* Takes a list and splits its content into
    two lists of equal length (±1 if the initial list has odd length). *)
  | [] -> ([], [])
  | h::tail ->
    let (lst1, lst2) = split tail in (h::lst2, lst1);;

let rec merge lst1 lst2 =
  (* Merges two sorted lists into a sorted list. *)
  failwith "todo";;

let rec merge_sort lst = match lst with
  (* Sorts a list using the merge sort algorithm. *)
  | [] -> []
  | [x] -> [x]
  | _ ->
    let lst1, lst2 = split lst in
    merge (merge_sort lst1) (merge_sort lst2);;
```

1. Écrire une fonction `print_list: int list -> unit` qui permet d'afficher une liste d'entiers. Vérifiez que la fonction `split` a bien le comportement attendu sur un ou deux exemples.
2. Implémentez la fonction `merge`, et testez les fonctions `merge` et `merge_sort`.
3. Déterminez (et **justifiez**) les complexités temporelles des fonctions `split`, `merge` et `merge_sort`.

XXI. Télégraphe de Chappe en milieu montagneux

`exercice d'approfondissement` `C` `I/O` `récursivité` `tableaux` `complexité`

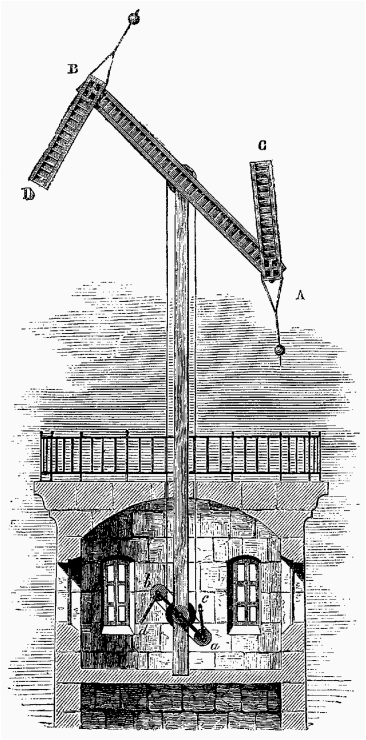


Fig. 6. – *Télégraphe de Chappe*, Louis Figuier.

Le **télégraphe de Chappe** est un système de télégraphe datant de la fin du XVIIIème siècle, qui permet la transmission d'un message de façon **visuelle**. Des tours comme celles de la Fig. 6 sont placées à intervalle régulier. Au sommet de ses tours se trouvent deux bras articulés, qui selon leurs positions, encodent une information. Une première tour transmet cette suite d'information, qui est réceptionné par une deuxième tour, qui la transmet à son tour, etc.

Ce système était redoutablement efficace : les tours étaient séparées d'une quinzaine de kilomètres, et il suffisait d'une dizaine de minutes pour transmettre un message entre Paris et Lille.

Le but de cet exercice est de déterminer **où construire des tours Chappe** pour transmettre un message entre une ville **A** et une ville **Z**, séparées par des montagnes.

Pour modéliser ce problème, on se donne en entrée un tableau d'entiers. Ce tableau relève l'altitude (assimilée à un entier) sur la ligne droite reliant la ville **A** à la ville **Z**. Par exemple, le tableau `{0,3,2,7,4,2,3,6,1}` signifie que le point **A** est à altitude 0 et que le point **Z** est à altitude 1. On représente cette information visuellement de la façon suivante :

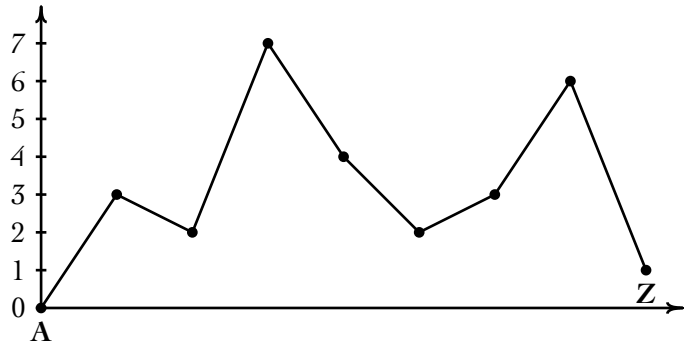


Fig. 7. – Visualisation du profil topographique décrit par le tableau `{0,3,2,7,4,2,3,6,1}`.

On suppose qu'une tour se trouve au point **A**, et une autre au point **Z**. Le but de l'exercice est de **déterminer à quels points il faut placer une tour**, sous les contraintes suivantes :

- deux tours consécutives doivent pouvoir se voir mutuellement,
- on souhaite minimiser le nombre de tours (ça coûte cher : en plus de la construction, il y a un employé dans chaque tour).

1. Écrire une fonction
   `int are_mutually_visible(int elevation[], int x1, int x2)`
   qui prend en entrée le tableau `elevation` décrivant le profil topographique, et deux indices `x1` et `x2` de ce tableau, et qui détermine si, si des tours étaient placées au point d'abscisse `x1` et d'abscisse `x2`, alors ces tours pourraient se voir mutuellement.

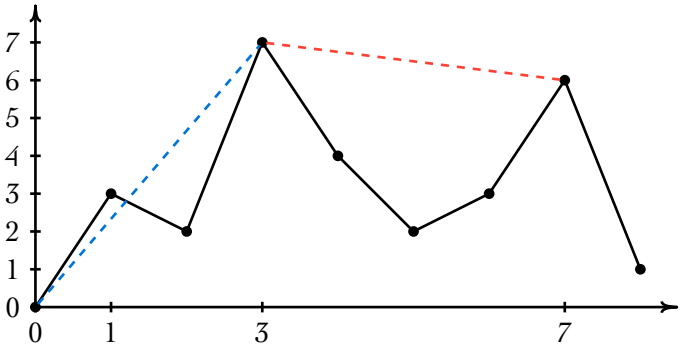


Fig. 8. – Lignes droites entre les points d'abscisse `x1` $= 0$ et `x2` $= 3$ (bleu), et entre `x1` $= 3$ et `x2` $= 7$ (rouge) pour le profil `{0,3,2,7,4,2,3,6,1}`.

   Par exemple (voir la Fig. 8), les tours entre `x1` $= 3$ et `x2` $= 7$ peuvent se voir, mais celles en `x1` $=$ 0 et `x2` $= 3$ ne le peuvent pas puisque le point d'abscisse 1 bloque leur champ de vision.
2. Écrire une fonction
   `int is_solution_valid(int elevation[], int towers[], int n)`
   qui détermine si une solution `towers` au problème `elevation` est **valide**. `towers` et `elevations` sont des tableaux de taille n, et `towers` sera tel que `towers[i]` vaut 1 si on y a construit une tour, et 0 sinon. Par **valide**, on entend qu'un message peut être transmis de **A** à **Z** en utilisant ces tours : on ne cherche pas à déterminer si cette solution minimise le nombre de tours.
3. Nous allons implémenter des solutions au problème des tours de Chappe avec des algorithmes récursifs. On place initialement des tours aux points d'abscisses 0 et $n-1$ (n étant le nombre d'entrées), correspondant aux points **A** et **Z**. On va ensuite déterminer s'il y a besoin de construire une nouvelle tour. Si ce n'est pas le cas, l'algorithme termine. Sinon, on détermine un point **M**, situé entre **A** et **Z**, où construire une tour. On appelle alors récursivement cette procédure entre les points **A** et **M**, et entre les points **M** et **Z**. Différents algorithmes pour déterminer ce point **M** où construire une tour intermédiaire donneront différents algorithmes pour résoudre le problème.
   a. Exécutez cet algorithme à la main (*sur papier*) sur l'exemple de Fig. 6 lorsque la stratégie pour choisir **M** est de prendre un point d'altitude maximal. (*Pas besoin de rendre cette question : vous pouvez m'appeler pour me montrer vos exemples.*)
   b. Écrivez une fonction
      `int *build_towers_highest(int elevation[], int n)`
      qui implémente cet algorithme. En entrée, on prendra le profil topographique, et on retournera un tableau de 0/1 de même taille, dont la i-ème entrée vaut 1 ssi on a construit une tour au point d'abscisse i.
   c. Quelle est la complexité temporelle de cette fonction ?
   d. Montrez que cette stratégie ne minimise pas le nombre de tours construites.
4. On change désormais de stratégie pour choisir **M** : pour tout point d'abscisse x, on considère son altitude $y(x)$ et l'ordonnée $y'(x)$ du point d'abscisse x sur la droite (**AZ**). On choisit **M** comme étant le point pour lequel $y(x) - y'(x)$ est maximal. Vous remarquerez que $y(x) - y'(x) > 0$ ssi

le point d'abscisse x entrave la vision entre **A** et **Z**. En un sens, le point **M** représente donc le point qui entrave le plus la vision entre les tours **A** et **Z**.

a. Exécutez cet algorithme à la main sur l'exemple que vous avez trouvé à la question 3d.

b. Écrivez une fonction

```
int *build_towers_biggest_obstruction(int elevation[], int n)
```

qui implémente cet algorithme.

5. On admet qu'une solution est **optimale** si, et seulement si, pour tout point **M** où l'on a placé une tour, pour toute tour **G** strictement à gauche de **M**, pour toute tour **R** (*right*) située à droite de **M**, alors **M** entrave la vision entre **G** et **R**.

   a. Implémentez une fonction

   ```
   int is_solution_optimal(int elevation[], int towers[], int n)
   ```

   qui détermine si une solution est optimale.

   b. Vérifiez empiriquement que les solutions calculées par `build_towers_biggest_obstruction` sont optimales.

6. Faire en sorte que, à l'exécution de votre programme, celui-ci demande à l'utilisateur de saisir le profil d'élévation (on rentrera un entier par ligne). Le programme affichera la solution en listant les abscisses des points où il y a des tours.

XXII. Notation polonaise inversée

`exercice d'application` `OCaml` `récursivité` `complexité` `arbres` `piles/files` `arithmétique`

La **notation polonaise inversée** (*reverse polish notation* ou *RPN* en anglais) permet de décrire des expressions arithmétiques sans utiliser de parenthèses. L'idée est simple : plutôt que d'écrire les opérateurs entre ses arguments, comme on le fait en notation infixe (la notation « classique »), on écrit plutôt **l'opérateur après les arguments.** Par exemple, $1 + 2$ devient $1\ 2\ +$. De même, $(1 + 2) + 3$ devient $12 + 3 +$. Au contraire, $1 + (2 + 3)$ devient $1\ 2\ 3\ +\ +$.

1. Écrire $(1 + 2) * (3 + 4)$ et $1 + 2 * 3 + 4$ en notation polonaise inversée.
2. On se dote d'un type récursif en OCaml

```
type expr =
| Const of int
| Add of expr * expr
| Mult of expr * expr;;
```

qui permet de représenter des expressions arithmétiques.

   a. Écrire une fonction

   ```
   eval_expr: expr -> int
   ```

   qui évalue une telle expression.

   b. Quelle est la complexité temporelle de votre fonction ?

3. Écrire une fonction

   ```
   expr_to_rpn: expr -> int_or_op list
   ```

   qui transforme une expression arithmétique en une liste qui représente cette expression en notation polonaise inversée, où

   ```
   type int_or_op = Int of int | Plus | Times;;
   ```

   Par exemple, sur l'entrée `Add(Mult(Const(1),Const(2)), Const(3))` votre algorithme retournera `[Int(1);Int(2);Times;Int(3);Plus]`.

4. Le but de cette question est d'écrire une fonction
   `eval_rpn: int_or_op list -> int`
   qui évalue une liste de caractères représentant une expression arithmétique en notation polonaise inversée. Un algorithme très efficace pour ce faire consiste à utiliser une pile : on part d'une pile vide, et on traite la liste de caractères de la façon suivante :
   - si c'est un entier, on l'empile
   - si c'est un opérateur, on l'applique aux deux entiers présents au sommet de la pile, et on empile le résultat.

   Un exemple, pour le calcul $3 * (10 + 5)$, qui donne 3 10 5 + * en notation polonaise inversée, est donné en Fig. 9.

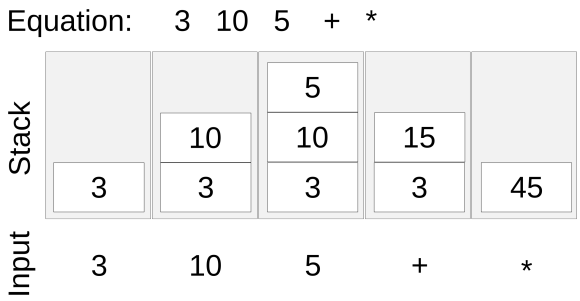


Fig. 9. – « Représentation de la structure de lecture d'une expression RPN par stacks. De gauche à droite et de haut en bas, case par case (étapes). » Figure par Stonemountain420, issue de Wikimedia, sous licence CC BY SA 3.0.

   a. Écrire la fonction `eval_rpn: int_or_op list -> int`.
   b. Vérifiez sur plusieurs expressions de type `expr` que les évaluer directement avec `eval_expr` donne le même résultat que de les transformer en notation polonaise inversée avec `expr_to_rpn` puis de les évaluer avec `eval_rpn`.

XXIII. Crible d'Ératosthène

`exercice d'application` `C` `I/O` `tableaux` `complexité` `arithmétique`

Le crible d'Ératosthène est un algorithme permettant de déterminer tous les nombres premiers plus petits qu'un entier m passé en entrée. L'algorithme repose sur une observation élémentaire : pour tout nombre naturel $k \geq 2$, tous ses multiples de la forme $k * i$ avec $i > 1$ sont forcément composés. En fait, la réciproque est aussi vraie, par définition d'un nombre premier. Le crible d'Ératosthène fonctionne en identifiant tous les entiers composés : ceux qui restent sont les nombres premiers ! L'algorithme maintient un tableau, qui contient l'information de si un nombre est composé ou premier (jusqu'à preuve du contraire). Initialement, tous les nombres sont supposés premiers, sauf 0 et 1. On commence par éliminer tous les multiples de 2 (c'est-à-dire qu'on déclare tous les nombres de la forme $2 * i$ avec $i > 1$ comme étant composés), puis tous les multiples de 3, puis de 4, etc, jusqu'à $m - 1$.

1. Exécutez cet algorithme à la main pour $m = 20$.
2. Quand est venue l'étape d'éliminer les multiples de 4, avez-vous éliminé des nombres qui étaient encore supposés être premiers ? Expliquez pourquoi, puis proposez une amélioration de l'algorithme.
3. Écrire une fonction
   `void remove_multiples(int arr[], int n, int k)`
   qui prend en entrée un tableau `arr`, rempli de 0 et de 1, de taille n, ainsi qu'un entier k, et qui définit la valeur de `arr[k*i]` à 0 pour tous les $i > 1$.

4. En déduire une fonction
   `int* sieve_eratosthenes(int m)`
   qui prend en entrée un entier m, et retourne un tableau de taille m, dont la i-ème entrée vaut 1 si i est premier, et 0 sinon.
5. Donner une borne supérieure sur la complexité spatiale et temporelle de la fonction `remove_multiples` puis de la fonction `sieve_eratosthenes`.
6. On peut remarquer que si un nombre n est composé, alors un de ses facteurs est forcément plus petit ou égal à $\sqrt{n}$ (preuve, par l'absurde : si d'aventure n pouvait s'écrire $k_1 \cdot k_2$ avec $k_1 > \sqrt{n}$ et $k_2 > \sqrt{n}$ alors on aurait $n = k_1 \cdot k_2 > \sqrt{n} \cdot \sqrt{n} = n$: que nenni !). En déduire une amélioration de la fonction `sieve_eratosthenes`. Que deviennent ses complexités spatiales et temporelles ?
7. Faire en sorte qu'à l'exécution de votre programme, l'utilisateur doive saisir un entier m ; votre programme écrira alors ensuite l'ensemble des entiers strictement plus petits que m dans un fichier `./primes.txt`.

XXIV. Anagrammes

`exercice d'application` `OCaml` `dictionnaires` `récursivité` `complexité`

Le but de cet exercice est de déterminer si deux chaînes de caractères sont des **anagrammes**, c'est-à-dire si l'une peut être obtenue en permutant les caractères de l'autre. Rien de plus simple pour déterminer si deux chaînes de caractères sont des anagrammes : il suffit de compter le nombre d'occurrences de chaque caractère dans la chaîne, et ces valeurs sont égales si et seulement si les deux chaînes sont des anagrammes. Par exemple, « niche » est une anagramme de « chien », « la crise économique » et « le scénario comique » sont des anagrammes, mais en revanche « être ou ne pas être, voilà la question » n'est pas une anagramme de « oui et la poser n'est que vanité orale » (la première chaîne contient deux ‹ ê › alors que la seconde non).[7]

Pour résoudre notre problème, je vous propose d'utiliser une **liste d'association**, qui est une liste de type `('a * 'b) list`, de sorte que si (c, v) et (c', v') sont tous deux éléments de la liste, alors $c \neq c'$. Autrement dit, une liste d'association encode une fonction qui associe des éléments de type `'a` (appelés **clés**) à des éléments de type `'b` (appelés **valeurs**). Par exemple, dans la liste
`[('a', 1); ('b', 5); ('c', 0)]`
de type `(char * int) list`, on a associé la valeur 1 à ‹ a ›, 5 à ‹ b ›, et 0 à ‹ c ›.

1. a. Définir une fonction
      `is_defined: ('a * 'b) list -> 'a -> bool`
      qui prend une liste d'association, une clé, et détermine si une valeur lui est associée.
   b. Définir une fonction
      `get_value: ('a * 'b) list -> 'a -> 'b`
      qui prend une liste d'association, une clé, et retourne la valeur qui lui est associée (si elle existe, sinon elle produit une erreur).
   c. Définir une fonction
      `update_value: ('a * 'b) list -> 'a -> 'b -> ('a * 'b) list`
      qui prend une liste d'association, une clé c , une valeur v et qui retourne une nouvelle liste d'association où la nouvelle valeur de la clé c est v. *Si la clé n'est pas présente, on se contentera d'ajouter la paire (c,v) à la liste. Sinon, on modifiera la valeur présente dans la liste.*
   d. Écrivez une fonction
      `count_chars_of_str: string -> (char * int) list` qui étant donné une chaîne de caractères, retourne une liste d'association comptant le nombre d'occurrences de chaque caractère.

[7] Ce sont en revanche des anagrammes si on ignore les espaces, les accents et la ponctuation. (Source : topito.com.)

2. a. De quelle(s) fonction(s) sur les listes d'associations auriez-vous besoin pour déterminer, à l'aide de la fonction `count_chars_of_str`, si deux chaînes sont des anagrammes ? Implémentez ces fonctions annexes.
   b. En déduire une fonction `are_anagrams: string -> string -> bool` qui détermine si deux chaînes sont des anagrammes.
   c. Donnez une borne supérieure (raisonnable) sur la complexité temporelle de votre fonction `count_chars_of_str`, puis de votre fonction `are_anagrams`.

XXV. Correction et terminaison d'une somme sur un tableaux

exercice de cours · théorique · correction · terminaison · complexité

```c
int array_sum(int *arr, int n) {
  assert(arr != NULL);
  assert(n >= 0);
  int sum = 0;
  int i = 0;
  while (i < n) {
    sum = sum + arr[i];
    i++;
  }
  return sum;
}
```

1. Démontrez que la fonction `array_sum` termine sur toute entrée.
2. Déterminez la complexité temporelle de cette fonction.
3. Démontrez sa correction, c'est-à-dire que `array_sum(int *arr, int n)` retourne la somme des `n` premiers éléments du tableau `arr` pour tout entier positif n.

XXVI. Correction d'une boucle while

exercice d'application · théorique · correction

On considère la fonction suivante en C :

```c
int get_first(int *arr, int n) {
  int i = 0;
  while (i < n && arr[i] < 42) {
    i++;
  }
  return i;
}
```

Démontrez que la fonction `get_first`, sur un tableau `arr`, de taille `n`, retourne

- le plus petit indice `i` tel que `arr[i] >= 42`, si un tel indice existe ;
- `n`, sinon.

Remarque : Ne pas sous-estimer la difficulté de cet exercice pour les étudiants les moins rigoureux. D'habitude on peut oublier de parler dans l'invariant de la condition de la boucle, et s'en sortir par une petite arnaque (du style « on voit bien que »). Si on fait cette erreur ici, il ne reste plus grand chose dans l'invariant… C'est donc un très bon exercice pour apprendre la rigueur, mais il est beaucoup plus difficile que le précédent.

XXVII. Correction d'un calcul récursif de la factorielle

exercice d'application | théorique | correction | terminaison | complexité

On considère la fonction suivante en OCaml, qui implémente le calcul de la factorielle avec une fonction récursive terminale.

```
let fact n =
  assert(n >= 0);
  let rec fact_aux n acc =
    if n = 0 then
      acc
    else
      fact_aux (n - 1) (n * acc)
  in fact_aux n 1
```

1. Démontrez que la fonction `fact` termine sur toute entrée.
2. Déterminez la complexité temporelle de cette fonction.
3. Démontrez que la fonction `fact` est correcte, c'est-à-dire que `fact n` vaut $n!$ pour tout $n \in \mathbb{N}$.

XXVIII. Correction d'un calcul de Fibonacci

exercice d'approfondissement | théorique | correction

On considère la fonction suivante en OCaml.

```
let fibo n =
  assert(n>=0);
  let rec fibo_aux k a b =
    if k = n then
      b
    else
      fibo_aux (k+1) a (a + b)
  in fibo_aux 0 1 1;;
```

La personne qui a écrit cette fonction souhaitait que `n` retourne le n-ème terme f_n de la suite de Fibonacci, où $f_0 = f_1 = 1$ et $f_{n+2} = f_n + f_{n+1}$ pour tout $n \in \mathbb{N}$. Ce n'est malheureusement pas tout à fait le cas. Que calcule `n` ? Prouvez-le.

XXIX. Correction d'un calcul d'une somme d'entiers

exercice d'application | théorique | correction

On considère la fonction suivante en OCaml.

```
let sum_integers n =
  let rec aux k acc =
    assert(k >= 0);
    if k = 0 then
      acc
    else
      aux (k-1) (k+acc)
  in aux n 0;;
```

Démontrez que `sum_integers n` retourne $\frac{n(n+1)}{2}$ pour tout $n \in \mathbb{N}$.

XXX. Complexité d'une fonction sur des listes

exercice de cours | théorique | complexité | terminaison

On considère les fonctions suivantes en OCaml.

```
let rec sum_list lst = match lst with
  | [] -> 0
  | h::t -> h + sum_list t;;

let rec is_bigger_than_sum lst = match lst with
  | [] -> true
  | h::t -> (h > sum_list t) && (is_bigger_than_sum t);;
```

Démontrez brièvement que la fonction `is_bigger_than_sum` termine sur toute entrée. Quelle est sa complexité temporelle ?

XXXI. Complexité d'une drôle de fonction récursive

exercice d'approfondissement | théorique | complexité | terminaison

On considère la fonction suivante en OCaml, de type `int -> int -> int`.

```
let rec foo a b =
  if a <= 0 || b <= 0 then
    max a b
  else (
    let x = foo (a-1) b
    and y = foo a (b-1) in
    if x <= y then
      2 * x
    else
      a * a + x
  );;
```

1. Démontrez que cette fonction termine sur toute entrée à l'aide d'un variant dans $\mathbb{N}$.
2. Quelle est sa complexité temporelle ?

Remarque : Attention à ne pas sous-estimer la difficulté de cet exercice, qui paraît *a priori* trivial à toute personne expérimentée, mais qui contient en fait un difficulté conceptuelle : définir un variant dans $\mathbb{N}$ pour une fonction ayant plusieurs paramètres. Il me semble important d'insister sur le fait que ce variant soit dans $\mathbb{N}$: on pourrait certes démontrer la terminaison à l'aide dans $\mathbb{N}^2$ muni de l'ordre lexicographique, mais la première question sert aussi à mettre sur la bonne voie pour la question de la complexité.

XXXII. Nombres de Hamming

exercice d'application | C | piles/files | complexité | I/O | arithmétique

Source : Cours de Jean-Pierre Becirspahic au lycée Louis-le-Grand

Le but de cet exercice est d'implémenter une structure de files en C. Je vous laisse le choix de l'implémentation (maillons chaînés, tableau circulaire, etc.).

1. Définir un type `Queue` permettant de stocker des files d'entiers.
2. Définir des fonctions

```c
Queue *queue_create(void);
void queue_enqueue(Queue *, int);
void queue_print(Queue *);
int queue_peek(Queue *);
int queue_dequeue(Queue *);
int queue_is_empty(Queue *);
void queue_free(Queue *);
```

Chaque fonction devra être testée, et la complexité temporelle de la fonction donnée en commentaire. **Testez chaque fonction avant d'implémenter la suivante.**

3. **Application.** Une *nombre de Hamming* est un entier naturel non-nul qui n'est divisible que par 2, 3, et 5. Les plus petits nombres de Hamming sont 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, etc. On se propose de générer les nombres de Hamming en utilisant la remarque suivante : tout nombre de Hamming autre que 1 est le produit d'un nombre de Hamming strictement inférieur avec 2, 3 ou 5. Et réciproquement, le produit d'un nombre de Hamming avec 2, 3 ou 5 est toujours un nombre de Hamming. On considère l'algorithme suivant : on maintient trois files d'entiers, `h2`, `h3` et `h5`, toutes initialisées pour ne contenir qu'une valeur : l'entier 1. Tant qu'on veut produire un nombre de Hamming, on chosit le plus petit entier `n` parmi les sommets de `h2`, `h3` et `h5`, on le défile, on affiche `n`, puis on enfile `2 * n` à `h2`, `3 * n` à `h3` et `5 * n` à h5.
   Écrire une fonction `void hamming(int m)` qui affiche tous les entiers de Hamming inférieur à m.
4. **Comparaison avec un algorithme naïf.** On considère ici un algorithme plus naïf, qui se contente d'énumérer les entiers de 1 à $m-1$ et de tester pour chacun s'il est un nombre de Hamming. Pour tester si un entier n est un nombre de Hamming, on peut par exemple le diviser par 2 tant qu'il est divisible par 2, puis par 3, puis par 5. Le résultat final est égal à 1 si et seulement si n est un nombre de Hamming.
   a. Empiriquement, pour $m = 10^9$, mon ordinateur met 10s à exécuter l'algorithme naïf, contre moins de 0.01s pour l'algorithme avec files. Expliquez cette différence.
   b. Implémentez cette fonction naïve (et vérifiez qu'elle donne les même résultats que la fonction précédente). *Si le besoin se fait sentir, vous pouvez utiliser les commandes Unix* `time` *pour mesurer le temps d'exécution d'un programme,* `wc -l` *pour compter le nombre de lignes d'un fichier et l'opérateur* `>` *pour rediriger la sortie vers un fichier.*

XXXIII. Un compteur et sa complexité moyenne

exercice d'application · OCaml · piles/files · terminaison · complexité · récursivité

On considère la fonction (partielle) suivante, qui transforme une pile de 0 et de 1 en une autre pile de 0 et de 1.

- initialement, on part d'une pile remplie d'un nombre arbitraire de 1;
- ensuite, on itère la construction suivante sur la pile :
  - si le sommet de la pile est un 1, on le remplace par un 0.
  - sinon, on dépile tous les 0 jusqu'à tomber sur le premier 1 ; on remplace ce 1 par un 0, puis on réempile un 1 pour chaque 0 dépilé.
  - sinon (càd s'il n'y a que des 0), on s'arrête.

1. Itérez cette fonction sur la pile [1;1;1], jusqu'à ce que l'algorithme termine. Que semble faire cette fonction ?
2. On souhaite maintenant l'implémenter en OCaml. Comme notre fonction est **partielle** (sur la pile remplie de 0, on ne retourne rien), on va utiliser le **type option**. On rappelle que le type `'a option` permet de représenter soit un objet de type `'a`, avec la syntaxe `Some (x)`, soit rien (`None`). Par ailleurs on représentera les piles avec des listes.

a. Écrire une fonction
`next_stack: int list -> int list option` qui prend une pile et retourne soit `Some(p)` où p est la pile obtenue par la fonction décrite dans l'énoncé, soit `None` s'il n'y a rien à retourner.
b. Écrire une fonction récursive
`countdown: int list option -> unit`
qui ne fait rien sur `None`, et sur `Some(p)` applique récursivement la fonction `next_stack` à `p` en affichant l'état de la pile à chaque étape.

3. Donnez un variant permettant de montrer que la fonction `countdown` termine.
4. Quelle est la complexité temporelle **moyenne** de la fonction `next_stack` ?

XXXIV. Numération de Zeckendorf

exercice d'approfondissement · C · arithmétique · tableaux · complexité

Vous connaissez l'écriture en base 10, en base 2, et plus généralement en base $b \in \mathbb{N}_{>0}$, où la séquence $(x_0, x_1, x_2, ..., x_{n-1})$ (avec $n \in \mathbb{N}$ et $x_i \in [\![0, b-1]\!]$ pour tout $i < n$) représente l'entier $\sum_{i=0}^{n-1} x_i b^i$. On s'intéresse ici à système de numération plus excentrique : le **système de numération de Zeckendorf**. Dans ce système, on représente un nombre en le décomposant comme la somme de **nombres** appartenant à la suite de Fibonacci. Pour rappel, la suite de Fibonacci est définie récursivement par $f_0 = 1$, $f_1 = 2$ et $f_{n+2} = f_n + f_{n+1}$ pour $n \in \mathbb{N}$: ses premiers termes sont donc $1, 2, 3, 5, 8, 13$, etc.[8] Par exemple, on peut écrire 6 comme $5 + 1$ et 17 comme $13 + 3 + 1$. On remarque vite que cette écriture n'est pas unique : 5 peut s'écrire comme 5 (c'est un nombre de Fibonacci) mais aussi comme $2 + 3$. Par suite, on peut écrire 6 comme $5 + 1$, mais aussi $2 + 3 + 1$. Le **théorème de Zeckendorf**, énoncé et démontré par le médecin belge Édouard Zeckendorf dans les années 1950, montre que tout entier peut s'écrire de façon unique comme la **somme de nombres de Fibonacci distincts et tels que deux nombres de Fibonacci consécutifs ne puissent apparaître dans cette somme**. C'est-à-dire que, par exemple, on s'interdit d'utiliser à la fois 3 et 5 (on utilisera à la place leur somme, 8, qui par un heureux miracle est aussi un nombre de Fibonacci). On admet ici ce théorème : étant donné un entier $n \in \mathbb{N}$, l'unique suite $(x_i)_{i \in \mathbb{N}} \in \{0, 1\}^{\mathbb{N}}$ telle que $n = \sum_{i=0}^{+\infty} x_i f_i$, et $\forall i \in \mathbb{N}, \neg(x_i = 1 \wedge x_{i+1} = 1)$, est appelée **écriture de Zeckendorf** (ou *représentation de Zeckendorf*) de n.

1. Calculez à la main l'écriture de Zeckendorf des entiers 10, 20 et 30.
2. Proposez une description haut-niveau de quelques lignes, en français (pas de pseudo-code !), d'un algorithme permettant de calculer l'écriture de Zeckendorf d'un nombre.

Dans le reste de cet exercice, on va manipuler ce système de numération en C. Sans surprise, le calcul des nombres de la suite de Fibonacci va donc jouer un rôle crucial. Pour des raisons de performance,[9] nous souhaitons éviter de répéter ces calculs. Pour ce faire, on va utiliser la **mémoïzation**, c'est-à-dire qu'on va stocker dans une structure de données les valeurs de la suite qui ont déjà été calculées. Naturellement, on veut une structure qui permette d'accéder à tout élément en temps constant, mais aussi qui soit redimensionnable. On va donc utiliser des… tableaux redimensionnables !

3. a. Définir un type `resizable_array` permettant de représenter un tableau redimensionnable contenant des **entiers positifs**.
b. Définir des fonctions

```
resizable_array *rar_create();
int rar_get_elem(resizable_array *rar, int i);
void rar_set_elem(resizable_array *rar, int i, int x);
```

[8] Il est plus commun de choisir comme premiers termes 0 et 1, ou même 1 et 1, mais le choix de $f_0 = 1$ et $f_1 = 2$ est important pour cet exercice : nous avons besoin que tous les entiers de la suite soient distincts.

[9] (Et pour coller au programme de colle.)

```
void rar_print(resizable_array *rar);
void rar_free(resizable_array *rar);
```

La fonction `rar_get_elem` devra toujours retourner quelque chose : on retournera une valeur par défaut, par exemple -1, si l'élément n'a pas été initialisé. La fonction `rar_set_elem` changera la valeur d'un élément du tableau : bien sûr, si le tableau n'est pas assez grand, on l'aggrandira auparavant. Vous vous assurerez que votre implémentation a une complexité amortie raisonnable; je ne vous demande cependant pas de le justifier par écrit.

c. Écrire une fonction `int fib(int n)`, qui :
   - vérifie dans une variable globale stockant un pointeur vers un `resizable_array` si on y a stocké le n-ème élément de la suite de Fibonacci, et
   - le retourne si c'est le cas,
   - sinon, le calcule, l'y stocke, puis le retourne.

d. Quelle est la complexité temporelle dans le pire cas de `fib(n);` en fonction de n, sos l'hypothèse qu'on a déjà calculé f_i pour $i < n$?
   Et de `for (int i = 0; i < n; i++) { fib(i); }`, sous l'hypothèse que l'on n'a encore calculé aucune valeur ?

4. On se propose de représenter une écriture de Zeckendorf $(x_i)_{i\in\mathbb{N}}$ comme un array contenant les valeurs $\{x_0, x_1, ..., x_{k-1}, -1\}$ où k un entier tel que $x_n = 0$ pour tout $n \geq k$.
   a. Écrire une fonction `void print_zeck_repr(int repr[])` qui affiche une telle représentation (on n'affichera pas le -1 final, celui-ci ne sert qu'à marquer la fin du tableau).
   b. Écrire une fonction `int int_of_zeck_repr(int repr[])` qui prend un array représentant $(x_i)_{i\in\mathbb{N}}$, et retourne $\sum_{i=0}^{+\infty} x_i \cdot f_i$.
   c. Écrire une fonction `int *zeck_repr_of_int(int n)` qui retourne l'écriture de Zeckendorf de son entrée.

5. **Bonus (difficile).** Déterminez un algorithme pour additionner des nombres écrits sous leur écriture de Zeckendorf. On s'interdira de calculer les entiers qu'ils représentent : on souhaite trouver un algorithme qui travaille directement sur les écritures de Zeckendorf. Implémentez cet algorithme en une fonction `int *add_zeck_repr(int x[], int y[])`. *Indice :* Commencez par additionner naïvement $(x_i)_{i\in\mathbb{N}}$ et $(y_i)_{i\in\mathbb{N}}$, ce qui nous donne la représentation $(z_i)_{i\in\mathbb{N}} = (x_i + y_i)_{i\in\mathbb{N}}$. Elle ne satisfait les règles de Zeckendorf (càd que z_i in {0,1} et qu'il ne peut y avoir de i tel que $z_i = 1$ et $z_{i+1} = 1$) : trouvez des règles de réécriture à appliquer à $(z_i)_{i\in\mathbb{N}}$ pour lui faire respecter les contraintes souhaitées.