Chapitre 1 Début du C

Exercice 1. Écrire un programme qui lit en entrée deux entiers a et b, et qui affiche les entiers de a à b inclus dans l'ordre croissant si $a \leq b$, et de b à a inclus sinon.

Exercice 2. Reprendre l'exercice précédent, sans utiliser de boucles.

Exercice 3. Écrire une fonction prenant en entrée deux bornes $a \leq b$ et tirant un entier aléatoire uniforme dans l'intervalle $[\![a, b]\!] = \{a, a+1, \ldots, b\}$.

Exercice 4. Écrire un programme lisant en entrée un entier n, puis n entiers quelconques $x_0, \ldots x_{n-1}$ et affichant leur somme.

Exercice 5. Corriger le code suivant :

```
1 /* Affiche les entiers de 1 à x (x >= 0) */
2 f(int x){
3   assert(x < 0);
4   for(i = 1; x; i+1){
5     printf(x);
6   }
7 }
```

Exercice 6. Écrire une fonction déterminant si un nombre est premier. En l'utilisant, écrire un programme qui prend en entrée un entier n et affiche les n premiers nombres premiers.

Exercice 7. Corriger le code suivant :

```
1 int main(){
2   // Lire N dans le terminal
3   int N;
4   scanf("%d\n", N);
5   // générer et afficher un nombre aléatoire entre 1 et N inclus
6   int x = rand()%N;
7   printf("%d\n", x);
8 }
```

Exercice 8. En s'inspirant de l'exponentiation rapide, écrire une fonction calculant le produit de deux nombres sans jamais utiliser l'opérateur de multiplication, **sauf** pour faire des multiplications par 2.

Exercice 9. La conjecture de Goldbach affirme que tout entier pair supérieur à 3 peut s'écrire comme la somme de deux nombres premiers. Vérifier que cette conjecture tient pour tous les entiers inférieurs à 10^6.

Exercice 10. Écrire une fonction calculant et affichant la décomposition en facteurs premiers d'un entier n, puis écrire un programme permettant de l'utiliser via le terminal.

Exercice 11. Proposer un ou plusieurs invariants de boucle pour l'algorithme suivant :

Algorithme 1 : `indice_min`$(n, X_0, \ldots, X_{n-1})$

Entrée(s) : $n \in \mathbb{N}^*, X_0, \ldots, X_{n-1} \in \mathbb{R}$
Sortie(s) : $i_m \in [\![0, n-1]\!]$ indice d'un élément minimum de X

```
1 i_m ← 0;
2 i ← 1;
  // Invariant(s): ...
3 tant que i < n faire
4     si X_i > X_{i_m} alors
5         i_m ← i;
6     i ← i + 1;
7 retourner i_m
```

Exercice 12. Corriger et simplifier le code suivant :

```c
/* Renvoie le premier entier k tel que B^k (B puissance k) > n */
int f(int n, int B){
  int pB = 1;
  int i  = 0;
  // Invariant: pB = B^i
  while (pB <= n){
    pB = pB * pB;
    i = i + 1;
  }
  if (pB < n){
    return i + 1;
  } else {
    return i;
  }
}
```

Exercice 13. La méthode de Monte-Carlo d'approximation de π consiste à tirer des points aléatoires dans le carré unité $[0,1] \times [0,1]$, et à compter la proportion p de points qui sont dans le quart de cercle de centre 0 et de rayon 1. L'aire du carré étant 1, et l'aire du quart de cercle étant $\frac{\pi}{4}$, on s'attend à avoir $4p \approx \pi$. Implémenter cette méthode en C, et approximer π en tirant 10^8 points. L'approximation est-elle précise ?

Exercice 14. On suppose disposer d'une fonction `float f(int n)`, dont le temps d'exécution est très long. Expliquer pourquoi le programme suivant, qui calcule la valeur maximale atteinte par `f` entre 1 et 1000, n'est pas efficace, et le corriger :

```c
int main(){
  float m = f(1); // maximum courant
  // invariant: m = max {f(1), ... f(i-1)}
  for (int i = 2; i <= 1000; i++){
    if (f(i) > m){
      m = f(i);
    }
  }
  printf("%f\n", m);
}
```

Exercice 15. Déterminer à la main ce qu'affiche le programme suivant lorsqu'on le compile et qu'on l'exécute. On pourra chercher des invariants liant a, b et s.

```c
#include <stdio.h>

int main(){
  int a = 0;
  int b = 1;
  int s = 0;
  while (a < 200){
    s += b;
    a += 1;
    b += 2;
  }
  printf("%d\n", s);
}
```

Exercice 16. Écrire une fonction qui affiche l'heure actuelle **aussi précisément que possible**. L'utiliser pour écrire un programme horloge qui affiche l'heure chaque seconde.

Exercice 17. Écrire un programme qui génère en boucle des problèmes de calcul mental aléatoires et demande à l'utilisateur de les résoudre. On pourra générer des additions, des multiplications, mais aussi des questions plus exotiques : des calculs de fractions, de racines carrées à 3 décimales près, des résolutions d'équations, etc...

Exercice 18. (*Difficile*) On plie un ruban de papier en deux, k fois d'affilée, en ramenant systématiquement le bord droit sur le bord gauche, puis on le déplie. On voit apparaître des pliures orientées vers le haut (H), et d'autres vers le bas (B). Par exemple, pour $k = 1$, une seule pliure H apparaît, et pour $k = 2$, trois pliures apparaissent : HHB (voir schéma).

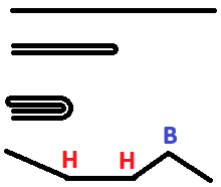

FIGURE 1 – Pliage et dépliage pour $k = 2$

Écrire un programme prenant en entrée k et affichant le sens des pliures de gauche à droite après avoir plié k fois puis déplié la feuille. On pourra chercher une structure récursive au problème, et commencer par déterminer le **nombre** de pliures qui apparaissent.

Chapitre 2 Encodage des entiers

Exercice 19. Écrire une fonction qui, étant donné B une base, $n \in \mathbb{N}$ et $i \in \mathbb{N}$, renvoie le i-ème chiffre de n en base B.

Exercice 20. Écrire une fonction qui compte le nombre de 1 dans l'écriture binaire d'un entier n.

Exercice 21. Écrire un programme qui affiche la décomposition en base B d'un entier n rentré par l'utilisateur.

Exercice 22. Écrire une fonction qui prend en entrée un entier k, et énumère tous les entiers de 0 à $2^k - 1$ en binaire.

Exercice 23. Écrire une fonction `int lecture(int B, int k)` qui demande k chiffres en base B à l'utilisateur, et affiche le nombre n obtenu.

Exercice 24. Écrire une fonction `int longueur(int n, int B)` qui calcule la longueur d'un entier n en base B (par exemple, la longueur de 1563 en base 10 est 4). On considèrera que la longueur de 0 est 0 pour simplifier le code.

Exercice 25. Un nombre premier tronquable est un nombre premier qui reste premier si l'on enlève un nombre arbitraire de chiffres à sa droite. Par exemple, 7393 est un nombre premier tronquable car $7393, 739, 73$ et 7 sont tous premiers. A l'aide d'un programme, déterminer le plus grand nombre premier tronquable inférieur à 10^9.

Chapitre 3 Pointeurs, tableaux, structures, fichiers

Pointeurs et tableaux

Exercice 26. Écrire une fonction `void swap(int* p, int* q)` qui prend en entrée deux adresses mémoires et échange les valeurs qui s'y trouvent.

Exercice 27. Écrire une fonction récursive `int factorielle(int n)`, et afficher l'adresse de `n` à chaque appel récursif. En déduire le sens dans lequel la pile d'appel grandit.

Exercice 28. Écrire une fonction `int* range(int n)` qui renvoie le tableau $[0, 1, \ldots, n-1]$.

Exercice 29. Écrire une fonction `void affiche_entiers(int* t, int n)` qui affiche les n entiers stockés à partir de l'adresse t.

Exercice 30. Corriger le code suivant pour qu'il libère la mémoire correctement :

```
1 int** p = malloc(sizeof(int*));
2 *p = malloc(3*sizeof(int));
3 (*p)[0] = 1
4 (*p)[1] = 2
5 (*p)[2] = 3
6 free(p);
7 free(*p);
```

Exercice 31. L'ordre des variables en mémoire n'est pas forcément l'ordre de déclaration au sein d'une fonction. Déterminer le plus précisément possible la stratégie que semble appliquer GCC pour choisir l'ordre de stockage des variables d'une fonction.

Algorithmique des tableaux

Exercice 32. Écrire une fonction `int* ecriture(int n, int B)` qui renvoie le tableau des chiffres de n en base B.

Exercice 33. Écrire une fonction `int lecture(int* t, int n, int B)` qui lit le tableau t de n chiffres en base B et renvoie le nombre correspondant.

Exercice 34. Écrire une fonction `int plus_grande_somme(int* t, int n)` qui prend en entrée un tableau t de n valeurs, et renvoie la plus grande valeur de $t[i]+t[j]$ possible avec $0 \leq i, j < n$.

Exercice 35. Écrire une fonction `int serie_zeros(int* t, int n)` qui prend en entrée un tableau t de n valeurs, et renvoie le plus grand nombre de zéros consécutifs trouvés dans t. Par exemple, pour $t = [2, 3, 0, 0, 1, 0, 1, 0, 0, 0, 0, 7]$, la fonction renverra 4 car t a une série de 4 zéros consécutifs.

Exercice 36. Corriger la fonction suivante (sans ordinateur) :

```
/* Détermine si t est trié dans l'ordre croissant.
   n est la taille de t */
bool est_trie(int* t, int n){
  for (int i = 0; i < n; i++){
    if (t[i] > t[i+1]){
      return false;
    }
  }
  return true;
}
```

Exercice 37. Écrire une fonction `int* concat(int* t1, int n1, int* t2, int n2)` prenant en entrée deux tableaux t_1, t_2 de tailles respectives n_1, n_2, et renvoyant un nouveau tableau contenant les valeurs de t_1 suivies de celles de t_2.

Exercice 38. On suppose avoir écrit une fonction `bool condition(int x)`. Écrire une fonction `int* filtre(int* t, int n)` prenant en entrée t un tableau de taille n et renvoyant un tableau contenant les valeurs de t vérifiant `condition`. Par exemple, si `condition` est la fonction testant si un entier est pair, pour $t = [5, 12, 4, 7, 9, 3, 6, 4]$ la fonction `filtre` renverra $[12, 4, 6, 4]$.

Structures

Exercice 39. On considère la structure suivante pour représenter des cartes à jouer

```
struct carte {
  int valeur; // 1-10 pour les chiffres, 11-13 pour les têtes
  int couleur; // 1, 2, 3 ou 4 pour coeur, carreau, pique, trèfle
};
```

Renommer le type `struct carte` en `carte_t`, puis écrire une fonction `main` créant un Sept de Trèfle et un Valet de Carreau.

Exercice 40. On propose la structure suivante représentant des groupes d'amis :

```c
#define MAX_MEMBRES 100
struct groupe {
  int nb_membres;
  // prenom[i] est le prénom du i-ème membre
  char* prenom[MAX_MEMBRES];
  // est_ami[i][j] vaut true si i considère
  // que j est son ami
  bool est_ami[MAX_MEMBRES][MAX_MEMBRES];
};
typedef struct groupe groupe_t;
```

Q1. Écrire une fonction `void print_amis(groupe_t* g, int i)` affichant le prénom de la personne i, puis de chacun de ses amis.

Q2. Écrire une fonction `void init_groupe(groupe_t* g)` initialisant g en un nouveau groupe d'amis vide.

Q3. Écrire une fonction `void ajouter_personne(groupe_t* g, char* prenom` qui rajoute à g une nouvelle personne.

Q4. Écrire une fonction `void ajouter_amitie(groupe_t* g, char* p1, char* p2)` qui met à jour g pour que p_1 considère que p_2 est son ami. On supposera que tous les prénoms sont distincts, et que p_1 et p_2 sont bien des membres du groupe g.

Exercice 41. On considère le type suivant :

```c
typedef struct humain {
  char prenom[20];
  char* nom;
  int age;
}  humain_t;
```

Dans le main, créer un humain entièrement stocké dans le **tas**, et un autre entièrement stocké dans la **pile**.

Exercice 42. Créer un type `point_t` permettant de représenter une masse ponctuelle dans l'espace 3D. Implémenter une fonction `void chute_libre(point_t* p, float g)` prenant en entrée une masse p, ainsi que l'intensité du champ de gravité g. Cette fonction simulera la chute de p par pas de 1 milliseconde, et affichera les coordonnées de p après chaque pas.

Exercice 43. Reprendre l'exercice précédent, en rajoutant un sol dur en $y = 0$ sur lequel la masse peut rebondir.

Fichiers

Exercice 44. Écrire une fonction `void affiche_entiers(char* nom_fichier)` lisant dans un fichier et affichant les entiers trouvés, jusqu'à atteindre la fin du fichier.
Tester la fonction sur des exemples corrects, puis sur un fichier contenant un caractère invalide (une lettre, une parenthèse, etc...). Corriger la fonction pour qu'elle s'arrête soit à la fin du fichier, soit dès qu'un caractère invalide est rencontré et empêche de lire un entier.

Exercice 45. Modifier la fonction précédente pour qu'elle lise les entiers deux par deux et affiche la somme à chaque fois. Par exemple, si le fichier contient :

```
3 15
47 4
2 3
```

alors la fonction affichera 18, 51, et 5.

Exercice 46. Écrire une fonction qui cherche dans un fichier si un entier donné apparaît.

Exercice 47. Écrire un programme qui lit dans un fichier supposé contenir sur chaque ligne un mot (sans espace) puis un entier :

```
carotte 3
cerise 2
...
```

Pour chaque couple (s, n), le programme affiche n fois le mot s dans le terminal :

```
carotte
carotte
carotte
cerise
cerise
...
```

Exercice 48. Écrire un programme qui fait l'inverse du précédent : il lit dans un fichier contenant un mot par ligne, et regroupe les mots identiques adjacents.

Exercice 49. Écrire un programme C qui recopie tous les flottants que l'utilisateur rentre dans un fichier.

Exercice 50. Se renseigner sur la commande cat du terminal qui sert à afficher le contenu d'un fichier, et en implémenter une version basique.

Exercice 51. Se renseigner sur la commande cp du terminal qui sert à copier des fichiers, et en implémenter une version basique.

Exercice 52. On rappelle l'existence de la fonction `strcat` de la librairie `string.h`.
On suppose disposer d'un fichier `syllabes.txt` contenant plusieurs syllabes d'au plus 10 lettres chacune, sur des lignes distinctes. Écrire un programme C qui prend en entrée deux entiers n et k, et génère n mots aléatoires de k syllabes. Par exemple, si le fichier contient :

```
ra
ba
tol
bou
```

le programme pourra générer des mots comme `raboutol`, `barababou`, etc...

Chapitre 4 Algorithmique, complexité

Pour chaque programme, algorithme et fonction de cette section (et des suivantes), vous pouvez vous entraîner à évaluer la complexité pire cas asymptotique. On rappelle que pour exprimer la complexité d'un $\mathcal{O}$, il faut simplement majorer le temps d'exécution dans le cas général, et que pour l'exprimer sous la forme d'un θ, il faut chercher une famille d'entrées qui atteignent la borne trouvée.

Exercice 53.

Implémenter la recherche par dichotomie en C :

```
1 /* Cherche x dans T, tableau de taille n. Renvoie un indice
2    d'une case contenant x le cas échéant, -1 si x n'apparaÃ®t
3    pas dans T */
4 int dichotomie(int* T, int n, int x);
```

Exercice 54. Implémenter le tri par sélection et/ou le tri par insertion en C.

Exercice 55. Implémenter la fonction de partition du tri rapide, puis le tri rapide.

Exercice 56. Qu'arrive-t-il lorsque l'on utilise le tri rapide sur un tableau de booléens ? En déduire un algorithme de tri en place linéaire pour les tableaux de booléens.

Exercice 57.

Q1. Écrire un algorithme qui, étant donné $a \leq b \in \mathbb{N}$ les bornes d'un intervalle et $x \in \mathbb{N}$, détermine si $x \in [\![a, b]\!]$.

Q2. Écrire un algorithme qui, étant donné T un tableau d'intervalles $[(a_0, b_0), \ldots, (a_{n-1}, b_{n-1})]$, renvoie un intervalle représentant l'intersection $\bigcap_i [\![a_i, b_i]\!]$.

Q3. (Difficile) Écrire un algorithme qui étant donné T un tableau d'intervalles calcule le **nombre d'éléments** présent dans l'union des intervalles de T (attention aux chevauchements !).