DS INFO N°2
Corrigé

Exercice 1 (Questions de cours).

1. (Question de cours.) En n'utilisant que le stylo noir ou le crayon à papier, représentez sur un grand schéma clair et propre la mémoire allouée à un processus en détaillant les différents segments. Ce schéma sert pour la question ci-dessous, lisez donc la suite de l'exercice avant de commencer le schéma.

2. Voici un code C très simple.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int une_fonction(int n, float **ptab)
5  {
6    int p = 2*n + 1;  // FIN DE LA LIGNE 6
7
8    *ptab = calloc(p, sizeof(float)); // FIN DE LA LIGNE 8
9
10   return p;
11 }
12
13 int main(void)
14 {
15   float *tab    = NULL;
16   int taille  = une_fonction(3, &tab);
17
18   tab[2] = 2.0*taille; // FIN DE LA LIGNE 18
19
20   if (tab != NULL)
21     free(tab);
22   tab = NULL;
23
24   return 0;
25 }
```

Donner le diagramme entrée/sortie de l'algorithme codé par la fonction `une_fonction`. Comment appelle-t-on la technique utilisée ? A quoi sert-elle ?

3. On suppose que l'on a suspendu l'exécution de ce programme juste à la fin de la ligne 6. Complétez toujours au crayon ou en noir le schéma de la question précédente en y dessinant les variables allouées par le programme dans les segments **dynamiques** et leur contenu dans leur état **à ce moment de l'exécution**.

4. On exécute l'instruction de la ligne 8. Mettez à jour **en vert** le schéma de la question précédente **à ce moment de l'exécution**. Indiquez bien les relations entre pointeurs et valeurs pointées par des flèches, quand il y en a. Vous pouvez rayer proprement en vert. Pour les valeurs d'adresse vous en "inventerez".

5. On suppose enfin que l'exécution en est maintenant à la fin de la ligne 18. Mettez à jour le schéma avec le stylo **rouge**.

Corrigé de l'exercice 1.

[Retour à l'énoncé]

1. La mémoire virtuelle attribuée par le système d'exploitation à un processus est segmentée, c'est-à-dire découpée en plusieurs zones de mémoire destinées à différents usages. Il y a :

   les segments statiques (taille fixe) : le segment de code, le segment de données statiques (rodata, data et bss)

   les segments dynamiques : le segment de pile et le tas

2. La fonction `une_fonction` prend une valeur entière en entrée (`n`) et renvoie deux valeurs en sortie : un entier et un pointeur vers un flottant (en fait un tableau de flottants), qui a été passé par adresse.

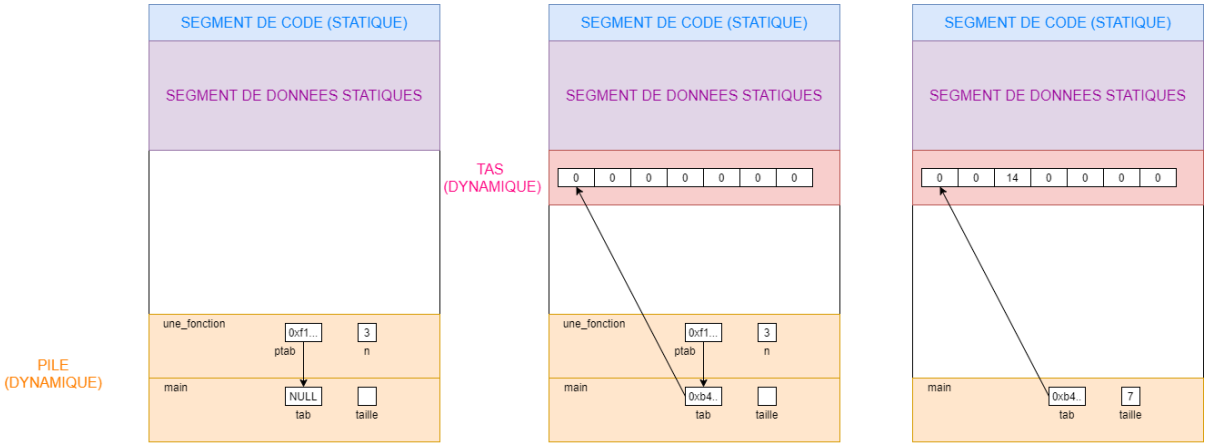
3. A la fin de la ligne 6, la pile d'appels est formée de deux blocs d'activation empilés : un pour la fonction `main` et un pour la fonction fonction `une_fonction`. Ces blocs et les variables locales qu'ils contiennent sont stockés dans le segment de pile.

   — Le bloc d'activation de la fonction `main` contient les variables locales à cette fonction : un pointeur vers un flottant, `tab`, qui contient toujours à ce stade la valeur d'adresse `NULL` et un entier `taille` à laquelle aucune valeur n'a encore été affectée.

   — Le bloc d'activation de la fonction `une_fonction` contient une copie des valeurs données en entrée de la fonction, à savoir un variable entière `n`, qui contient la valeur 3 et l'adresse d'un pointeur `ptab` qui contient l'adresse du pointeur `tab` stocké dans le bloc d'activation du `main`. Le bloc d'activation de la fonction `une_fonction` contient aussi une variable locale à la fonction, `p`, de type entier et qui contient à ce stade de l'exécution la valeur 7.

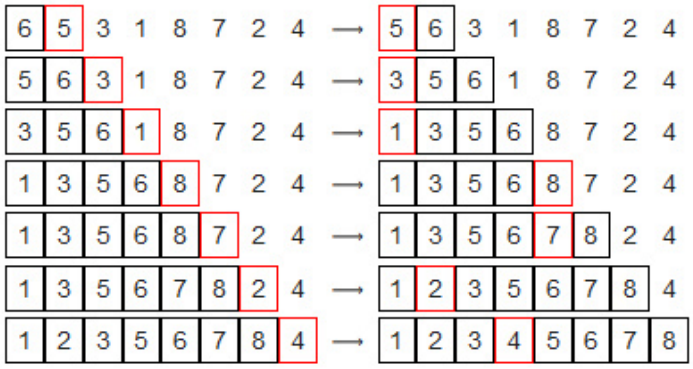
4. Après l'exécution de la ligne 8, un tableau de flottants de taille 7 (7 cases de 32 bits) est alloué dans le tas. L'adresse de la première case de ce tableau a été affectée au pointeur `tab`, lui-même stocké dans le bloc d'activation du `main` dans le segment de pile

5. A la ligne 18, l'appel de la fonction `une_fonction` est terminé : l'espace mémoire occupé par le bloc d'activation correspondant à l'appel de cette fonction à la ligne 16 a été libéré. La variable `p` a disparu, de même que le pointeur de point `ptab`. Mais `tab`, qui est situé dans le bloc d'activation du `main` existe toujours et pointe toujours vers la tableau alloué dans le tas. La valeur 14 a été affectée à la case 2 du tableau `tab` dans le tas.



Exercice 2 (Algorithme de tri).

Le tri par insertion est le tri « du joueur de carte ». On parcourt les éléments du tableau. Pour chaque valeur, on va la replacer (l'insérer) au bon endroit parmi le sous tableau de gauche.
Pour cela, on décale les élément à gauche d'un cran vers la droite jusqu'à ce que la case correspondant à la bonne place de la valeur soit libérée.
Puis on passe à la valeur suivante et on la replace au bon endroit dans le sous tableau de gauche...etc et ceci jusqu'à avoir réinséré tous les éléments.
Voici ci-dessous les étapes de l'algorithme du tri par insertion sur le tableau constitué des valeurs 6, 5, 3, 1, 8, 7, 2, et 4, dans cet ordre.



On montre uniquement le résultat de l'insertion de chaque élément sans montrer tous les décalages réalisés.
Par exemple, lors de l'insertion du 2 à l'avant dernière étape, pour placer le 2 au bon endroit, on a décalé vers la droite le 8, puis le 7, puis le 6, puis le 5, puis le 3.

1. On considère le tableau contenant les valeurs suivantes, dans cet ordre : `-11, 8, 4, -3, 1`. En suivant le modèle ci-dessus, détaillez les différents états du tableau au cours de l'algorithme
2. Écrivez l'algorithme du tri par insertion en pseudo-code **sans utiliser la récursivité** : détaillez bien les entrées, les sorties éventuelles et l'algorithme. Vous pouvez utiliser les notations abrégées usuelles pour éviter de longues phrases.
3. Traduisez cet algorithme sous la forme d'une fonction OCaml nommée `tri_insertion`
4. En fonction de quel paramètre mesure t-on la complexité de cet algorithme ? Évaluez la complexité temporelle **au pire** de cet algorithme. Vous pourrez annoter proprement votre pseudo-code pour expliquer votre évaluation.
5. Réécrivez l'algorithme du tri par insertion en pseudo-code en utilisant cette fois une formulation récursive
6. Quelle structure de données permet de faciliter l'implémentation de cet algorithme en OCaml ? (Réponse en 1 phrase)
7. Traduisez cet algorithme récursif en OCaml.

Corrigé de l'exercice 2.

[Retour à l'énoncé]

1. On marque en rouge l'élément à insérer à chaque étape :

Début	-11, 8, 4, -3, 1
Après insertion du 1er élément	-11, 8, 4, -3, 1
Après insertion du 2eme élément	-11, 8, 4, -3, 1
Après insertion du 3eme élément	-11, 4, 8, -3, 1
Après insertion du 4eme élément	-11, -3, 4, 8, 1
Après insertion du 5eme élément	-11, -3, 1, 4, 8
Fin	

2. L'algorithme de tri par insertion en pseudo-code s'écrit :

Donnée : `tab`, `tableau` type non défini (polymorphisme)

`n`, `entier`, taille du tableau
`i, j`, `entiers`, indices de boucles
`v`, `non défini`, variable tampon pour sauvegarder une valeur

Pour *`i` allant de 0 à `n-1`* **faire**
    `v` ← `tab[i]` (sauvegarder la valeur d'indice i du tableau que l'on va réinsérer)
    `j` ← `i` (on part de l'indice i)
    Tant que *(`j` ≥ 1 ET `v` < `tab[j-1]`)* **faire**
        `tab[j]` ← `tab[j-1]` (décaler la valeur vers la droite)
        `j` ← `j - 1` (passer à la valeur suivante vers la gauche)
    A la fin de cette boucle, on a décalé toutes les valeurs plus grandes que `v` vers la droite et l'indice `j` correspond à la case que l'on a libéré pour insérer la valeur `v`
    `tab[j]` ← `v`

---

3. Voici le code OCaml du tri par insertion (version itérative = sans récursivité)

```
# let tri_insertion tab =
  let n = Array.length tab in
  for i = 1 to (n-1) do    n-1 tours de boucle
    let v = tab.(i) in
    let j = ref i in
    while ( !j >= 1 && v < tab.((!j-1)) ) do  i tours de boucle au pire
      tab.(!j) <- tab.(!j-1); (* on est pas obligé de faire des swaps
 ici car on décale tout vers la droite. Il suffit de conserver la pre
mière valeur comme nous l'avons fait dans v *)
      j := !j - 1
    done;
    tab.(!j) <- v
  done;
  tab;;
val tri_insertion : 'a array -> 'a array = <fun>
# let tab_trie = tri_insertion_rec [ 6; -5; 7; 1];;
val tab_trie : int list = [-5; 1; 6; 7]
# let tab_trie2 = tri_insertion_rec [ 1; 7; 3; 2; -5; 7; 1];;
val tab_trie2 : int list = [-5; 1; 1; 2; 3; 7; 7]
# let tab_trie3 = tri_insertion_rec [ 1; ];;
val tab_trie3 : int list = [1]
# let tab_trie4 = tri_insertion_rec [ 'f'; 'z'; 'a'; 'c'; 'w'];;
val tab_trie4 : char list = ['a'; 'c'; 'f'; 'w'; 'z']
# let tab_trie5 = tri_insertion_rec [ 1.3; -5.1; 8.1; -3.0; -4.1];;
val tab_trie5 : float list = [-5.1; -4.1; -3.; 1.3; 8.1]
```

4. La complexité temporelle est évaluée en fonction de la taille du tableau, que l'on note n. Étudier la complexité de cet algorithme, c'est chercher à comprendre de quelle manière

(logarithmique, linéaire, quadratique, exponentielle) évolue le nombre d'opérations nécessaires lorsque n croit. On parle de complexité asymptotique car ce sont les cas des grands tableaux $n \to \infty$ qui nous intéressent.

Il y a deux boucles imbriquées : une boucle for et une boucle while. On s'attend à une complexité quadratique.

De façon un peu plus rigoureuse, on note C_{int} le nombre d'opérations élémentaires effectuées par tour de la boucle interne :

$$C_{\text{int}} = 2 \text{ tests de boucle} + 1 \text{ ET logique} + 3 \text{ soustractions} + 2 \text{ affectations} = 7$$

et C_{ext} le nombre d'opérations élémentaires effectuées dans le for, hors du while :

$$C_{\text{ext}} = 1 \text{ test de boucle} + 1 \text{ incrémentation} + 3 \text{ affectations} = 5$$

$$\sum_{i=0}^{n-1} \sum_{j=1}^{i} C_{\text{int}} + C_{\text{ext}} = C \times \sum_{i=0}^{n-1} i = C_{\text{int}} \times \frac{(n-1)(n-2)}{2} + (n-1) \times C_{\text{ext}} \in O(n^2)$$

C'est le terme en n^2 qui dominera tout quand n va devenir très grand (évolution asymptotique).

En fait, on remarque que le décompte précis de C_{ext} et C_{int} n'a en fait aucun intérêt et est même complètement arbitraire car :

- il dépend de ce que l'on appelle "opérations élémentaire",
- il dépend aussi de chaque exécution, par exemple s'il y a des instructions conditionnelles (en général on considère le pire cas)
- et surtout, il y a certainement bien plus d'opérations élémentaires effectuées au niveau machine, mais cela est caché par le langage de haut niveau utilisé.

Il n'y a donc aucun intérêt à faire ce genre de décompte.

Finalement, la seule chose importante est de bien comprendre que ces valeurs sont des constantes indépendantes de n : elles ne changeront pas en fonction de la taille du tableau. Dans la suite, vous n'aurez pas à donner de détails sur la valeur de ces constantes. Je l'ai juste fait pour

5. Pour l'algorithme récursif, il faut arriver à changer de point de vue. L'algorithme de tri par insertion consiste à réinsérer successivement les valeurs du tableau dans le tableau lui même.

- si le tableau est vide, rien à faire
- sinon, on lance le tri par insertion sur les (n-1) premiers éléments (récursivité) et on a juste à insérer le dernier

Pour avoir une méthode totalement récursive, il faut donc maintenant trouver un algorithme récursif qui insère une valeur v dans un tableau trié $[t_0; ...t_{p-1}]$. Deux cas se présentent :

- si le tableau est vide, insérer v revient à retourner un tableau contenant une seule valeur valant v
- si $v > t_p$ comme le tableau t est supposé trié, v est à la bonne place et le tableau résultat trié est simplement la concaténation $[t_0; ...t_{p-1}; v]$
- sinon, on insère v dans le sous tableau $[t_0; ...t_{p-2}]$ (récursivité), on obtient un tableau de taille p dans lequel v a été inséré et on ajoute t_{p-1} à la suite de ce tableau

6. En OCaml, les listes sont plus adéquates que les tableaux pour écrire des algorithmes récursifs, notamment grâce à l'opération `::` qui peut servir de motif permettant de déconstruire une liste en récupérant sa valeur de tête et sa queue. Mais dans ce cas, il faut donc penser à inverser la vision : on n'insère plus en parcourant un tableau de gauche à droite (occidentaux que nous sommes) mais de droite à gauche car le dernier élément correspond à la tête de liste.

7. Voici le code OCaml récursif obtenu :

```
# let rec tri_insertion_rec l =
  let rec insert v l_trie =
    match l_trie with
    | [] -> [v]
    | h::t when v < h -> v::l_trie
    | h::t -> h :: (insert v t)
  in
  match l with
  |[] -> []
  |h::t -> insert h (tri_insertion_rec t);;
val tri_insertion_rec : 'a list -> 'a list = <fun>
# let liste_triee = tri_insertion_rec [ 6; -5; 7; 1];;
val liste_triee : int list = [-5; 1; 6; 7]
# let liste_triee2 = tri_insertion_rec [ 1; 7; 3; 2; -5; 7; 1];;
val liste_triee2 : int list = [-5; 1; 1; 2; 3; 7; 7]
# let liste_triee3 = tri_insertion_rec [ 1; ];;
val liste_triee3 : int list = [1]
# let liste_triee4 = tri_insertion_rec [ 'f'; 'z'; 'a'; 'c'; 'w'];;
val liste_triee4 : char list = ['a'; 'c'; 'f'; 'w'; 'z']
# let liste_triee5 = tri_insertion_rec [ 1.3; -5.1; 8.1; -3.0; -4.1];;
val liste_triee5 : float list = [-5.1; -4.1; -3.; 1.3; 8.1]
# 
```

On note que notre fonction d'insertion récursive `insert_rec` est déclarée en interne de la fonction de tri par insertion pour éviter toute utilisation à mauvais escient hors de la fonction `tri_insertion_rec`.

Exercice 3 (Inférence de type).

Voici plusieurs fonctions OCaml. Indiquez **sur l'énoncé** le prototype de chaque fonction tel qu'inféré par l'interpréteur. Pour chaque fonction, vous expliquerez brièvement comment vous avez inféré le type. Vous pouvez expliquer en annotant directement le code sur l'énoncé. Proposez des annotations claires, concises et efficaces sans y passer trop de temps.

Corrigé de l'exercice 3.

[Retour à l'énoncé]

```
# let f1 y =
  let tab = [|"m"; "p"; "i"|] in
  let r = ref false in
  for j = 0 to (Array.length tab) -1 do
    if y = tab.(j) then
      r := true
  done;
  !r;;
val f1 : string -> bool = <fun>
# let f2 q r s =
  let h = ref 0 in
  while ( !q < (Array.length s) )
  do
    if (s.(!q) > 2. *. r ) then
      h := !h + 1
  done;
  !h;;
val f2 : int ref -> float -> float array -> int = <fun>
# let f3 k l f =
  let t = k.(f) in
  if (t > l.(f+1) ) then
    (l.(f) <- l.(f+1);l.(f+1) <- t);;
val f3 : 'a array -> 'a array -> int -> unit = <fun>
# let f4 r s t =
  let (_, x) = s in t.(r) <- 2.0 *. x;;
val f4 : int -> 'a * float -> float array -> unit = <fun>
# let f5 k l f =
  let t = k f in
  if (t > l.(f+1) ) then
    (l.(f) <- l.(f+1);l.(f+1) <- t);;
val f5 : (int -> 'a) -> 'a array -> int -> unit = <fun>
```

r est une ref sur un booléen

on compare y à un string donc y est un string

string

on renvoie la valeur référence par r donc un booléen

h est une ref vers un entier

q est une ref puisqu'on le déréférence et on compare la valeur référencée à un int donc q est une ref vers un int

s est un tableau de flottants

r est un flottant car on utilise l'opérateur *.

on renvoie la valeur référencée par h donc un entier

f est un indice donc c'est un entier, k est un tableau

l est un tableau

On ne peut déterminer les types de éléments de l, ni le type de t, mais ils doivent être identiques car on fait une comparaison

t est un tableau de flottants

r est un indice donc un entier

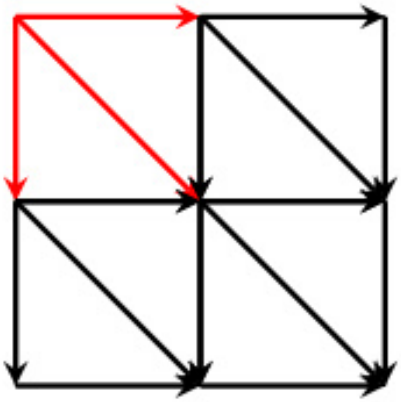
s est une paire dont le deuxième élément est un flottant

Les trois dernières fonctions terminent par une opération d'affectation donc ne renvoient rien (unit)

l est un tableau, f un indice donc un entier

Exercice 4.

1. Une grille comporte 2×2 cases. Partant du coin supérieur gauche, déterminer le nombre de chemins menant au coin inférieur droit si les seules directions de déplacements autorisées sont celles indiquées par les flèches.



2. Une grille comporte $m \in \mathbb{N}^*$ lignes et $n \in \mathbb{N}^*$ colonnes. On note $c_{i,j}$ le nombre de chemins issus d'un nœud $(i,j) \in [\![0,m]\!] \times [\![0,n]\!]$
Déterminer $c_{0,n}$ et $c_{m,0}$ puis, pour $i \geq 1$ et $j \geq 1$, établir une relation de récurrence donnant $c_{i,j}$.

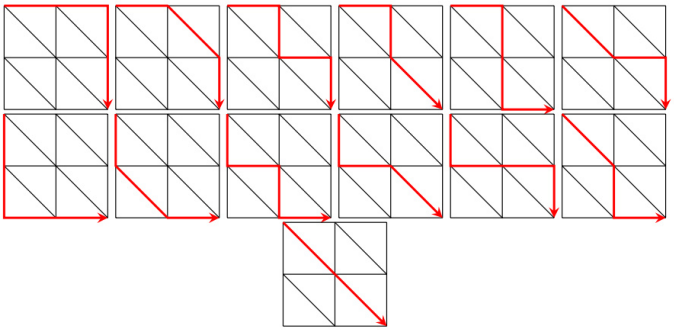
3. Écrire une fonction OCaml de prototype `c : int → int → int` qui calcule $c_{m,n}$

4. Discutez son efficacité en 3 phrases maximum.

Corrigé de l'exercice 4.

[Retour à l'énoncé]

1. On dénombre 13 chemins :



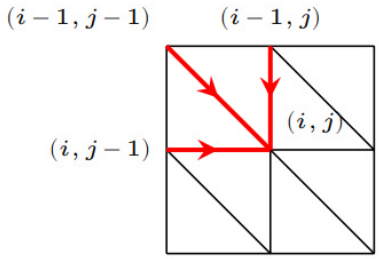
2. **Si** $m = 0$ la grille se réduit à un segment horizontal. Il n'existe qu'un seul chemin joignant le nœud $(0,0)$ au nœud $(n,0)$:

$$c_{0,n} = 1$$

Si $n = 0$ la grille se réduit à un segment vertical. Il n'existe qu'un seul chemin joignant le nœud $(0,0)$ au nœud $(0,m)$:

$$c_{m,0} = 1$$

Cas général : on se place en un nœud (i,j). Les nœuds voisins qui y aboutissent sont les nœuds $(i-1,j)$, $(i-1,j-1)$ et $(i,j-1)$.



On en déduit la relation de récurrence suivante :

$$c_{i,j} = c_{i-1,j} + c_{i-1,j-1} + c_{i,j-1}$$

3. Voici le code OCaml récursif correspondant. On rassemble les deux entrées i et j dans une paire et on utilise des motifs de filtrage pour différencier les cas. On teste également sur le cas de la question 1, qui nous donne bien 13 chemins possibles.

```
# let rec c i j = match (i,j) with
  | (0, j) -> 1
  | (i, 0) -> 1
  | _      -> ( c (i-1) j ) + ( c i (j-1) ) + ( c (i-1) (j-1) );;
val c : int -> int -> int = <fun>
# c 2 2;;
- : int = 13
```

4. Les coûts temporel et spatial sont élevés en raison de nombreux appels récursifs redondants sans possibilité évidente pour transformer cette récursion en une récursion terminale.

Exercice 5 (Crible d'Ératosthène).
Écrire une fonction en langage C nommée `eratosthene` qui prend en entrée un entier naturel n et qui **affiche** tous les nombres premiers jusqu'à n en utilisant l'algorithme du crible d'Ératosthène. Cet algorithme fonctionne de la manière suivante : on construit un tableau contenant tous les entiers de 2 à n, puis on supprime les multiples de 2, puis ceux de 3, etc. Les entiers qui n'ont pas été barrés sont premiers.
On rappelle qu'il est possible d'utiliser des booléens en C (en utilisant la bibliothèque `stdbool.h`)
On pourra également utiliser la racine carrée qui se nomme `sqrt` (et se trouve dans le module `math.h`)
On ne vous demande d'écrire que la fonction `eratosthene` avec les spécifications données.
Quelle est la complexité temporelle de cet algorithme ?

Petit rappel :

Définition 1 (Nombre premier)

Un nombre est premier s'il a **exactement 2 diviseurs : 1 et lui-même.**

Remarque. 1 n'est pas premier car il n'a qu'un seul diviseur : 1

Corrigé de l'exercice 5. *[Retour à l'énoncé]*

On utilise un tableau de booléens `tab` : si `tab[i]` contient la valeur vraie à la fin de l'algorithme, alors l'entier `i` est premier. Si elle contient la valeur faux, `i` n'est pas premier. L'algorithme détaillé fonctionne de la manière suivante :

— On initialise le tableau à `true` pour tous les entiers de 2 à `n`. 0 et 1 sont des cas un peu à part et on sait qu'ils ne sont pas premiers donc on met directement les booléens correspond à ces deux valeurs à `false`

— Ensuite, on va devoir rayer les nombres pour lesquels on trouve des diviseurs autres que 1 et `n`. Tous les nombres entre 2 et `n-1` sont des diviseurs potentiels. On boucle donc sur les diviseurs `d`, avec `d` allant de 2 à `n` (tous les nombres sont divisibles par 1 donc ne surtout pas démarrer la boucle à 1 !)

— Soit `d` un diviseur. Seuls les nombres plus grands que `d` peuvent être divisibles par `d` donc inutile de tester les entiers plus petits que `d` : on sait déjà qu'ils ne seront pas divisibles par `d` et donc on n'aura pas à modifier le booléen qui leur est associé dans `tab`. Pour rayer les nombres divisibles par `d`, on fait donc une boucle de `d+1` à `n`. Et là encore, il ne faut pas commencer la boucle à `d` car tout nombre est divisible par lui-même : ce sont les diviseurs autre que 1 et le nombre lui-même qui nous intéressent pour déterminer la primalité.

— Le test de divisibilité est réalisé en calculant le reste de la division euclidienne de `i` par `d` et en regardant si ce reste est nul.

— Une fois les boucles terminées, tous les indice `i` pour lequel `tab[i]` est resté à `true` sont premiers : en effet, pour ces nombres, le test de divisibilité n'a jamais été positif, ce qui signifie que ce nombre ne possède aucun autre diviseur que 1 ou lui-même.

Il existe de très nombreuses possibilités pour optimiser ce code.
Par exemple, on peut se contenter d'aller seulement seulement jusqu'à $\sqrt{n}$ dans la boucle sur les diviseurs. En effet, si `d` est un diviseur de `i`, alors il existe un entier `d'` tel que $i = d \times d'$. Et ce nombre `d'` est aussi un diviseur. Cela signifie que si on a trouvé un diviseur, on en a trouvé automatiquement un autre. On peut montrer mathématiquement (essayez de le faire !) que si un diviseur est entre 2 et $\sqrt{n}$, son jumeau est entre $n-1$ et $\sqrt{n}+1$. Nous n'avons besoin que d'en trouver un des deux pour savoir s'il faut rayer ou non `i`. Donc il suffit d'une boucle allant de 1 à $\sqrt{n}$.

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h> // penser à compiler avec l'option -lm
#include <assert.h>

void eratosthene(int n)
{
  assert(n > 0);

  if (n == 0 || n == 1)
  {
    printf("Il n'y a pas de nombre premier entre et 0 et  %d\n", n);
    return;
  }

  bool *tab = (bool *)malloc((n+1)*sizeof(bool));
  int i;
  int d;

  tab[0] = false;
  tab[1] = false;

  for (i = 2; i <= n; i++)
    tab[i] = true;

  for (d = 2; d <= abs(sqrt(n)); d++)
  {
    for (i = d+1; i <= n; i++) // on commence à d+1 car les nombres inférieurs à d ne peuvent pas être divisibles par d
    {
      if (i % d == 0)
        tab[i] = false;
    }
  }

  printf("Liste des nombres premiers entre 2 et %d: ", n);
  for (i = 1; i <= n; i++)
  {
    if (tab[i] == true)
      printf("%d ", i);
  }
  printf("\n");

  if (tab != NULL)
    free(tab);
  tab = NULL;

  return;

}


int main (int argc, char **argv)
{
  assert(argc == 2);
  int n = atoi(argv[1]);

  eratosthene(n);

  return 0;
}
```

Cet algorithme a une complexité temporelle quadratique si on le code naïvement sans faire optimiser les bornes des boucles. En effet, il y a deux boucles imbriquées, et on

réalise $C = 5$ opérations élémentaires au pire (1 test, 1 affectation, 1 calcul de reste + une incrémentation d'indice de boucle + 1 test de boucle) [1] .

$$\sum_{d=2}^{n}\sum_{i=d+1}^{n} C = C\sum_{d=2}^{n}(n-d) = C\sum_{d=0}^{n-2} d = C \times \frac{(n-2)(n-3)}{2} \in O(n^2)$$

Si l'on applique l'optimisation expliquée ci-dessus (ce qui est le cas dans ce code corrigé), alors :

$$\sum_{d=2}^{\lfloor\sqrt{n}\rfloor}\sum_{i=d+1}^{n} C = C\sum_{d=2}^{\lfloor\sqrt{n}\rfloor}(n-d) = C \times n(\sqrt{n}-1) + C\left(\frac{\sqrt{n}(\sqrt{n}-1)}{2} - 1\right) \in O(n\sqrt{n})$$

On a donc gagné un peu en terme de complexité temporelle puisque l'algorithme est en $O(n^{\frac{3}{2}})$, ce qui est moins bien qu'un algorithme linéaire mais un peu mieux qu'un algorithme quadratique.
Attention, il y a avait beaucoup d'autres façons de coder cet algorithme, et cette version est loin d'être la version la plus optimisée de cet algorithme. Je vous invite à aller lire cette page web passionnante et bien vulgarisée si vous avez un peu de temps :

`https://interstices.info/le-crible-deratosthene/`

1. Là encore, peu importe en fait la valeur exacte de C, l'important est qu'il s'agisse d'une constante indépendante de n.