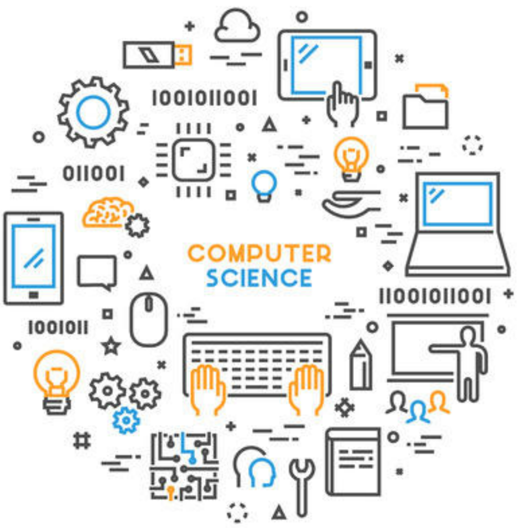
💻 Essentiel d'informatique

2023/2024

Victor Sarrazin



Bienvenue dans l'essentiel d'informatique de mes cours de prépa. Ce document a pour objectif de contenir l'intégralité des cours d'informatique afin de les condenser et de les adapter.

Bonne lecture…

Sommaire

Introduction aux langages :

Structures de données :

Informatique théorique :

Introduction

I Introduction au C

I.1 Variables

Pour définir une variable en C on a la syntaxe suivante : `type nom`

```c
int mango = 0;
```

Il est possible de définir plusieurs variables en même temps :

```c
int banana = apple = 12;
int pear, orange = 14; // pear est non initialisée et orange vaut 14
int potato = 12, tomato = 14; // potato vaut 12 et tomato vaut 14
```

I.2 Opérateurs

On a les opérations arithmétiques suivantes :

Opération	En C
Addition	`a + b`
Soustraction	`a - b`
Multiplication	`a * b`
Division	`a / b`
Modulo	`a % b`

On peut utiliser +=, -=, *=, /= et %= pour faire des opérations arithmétiques et des assignations

De plus on peut utiliser ++ et -- pour incrémenter/décrémenter

Les comparaisons se font avec >, >=, <=, < et ==.

On a des opérateurs binaires && (et logique), || (ou logique) et ! (négation de l'expression suivante)

> **Attention :**
> Le && est prioritaire sur le ||

I.3 Structures de contrôle

Pour exécuter de manière conditionnelle, on utilise `if (cond) {...} else if (...) {} ... {} else {}`

Ainsi le code suivant est valide :

```c
if (x == 1) {
  // Do code
} else if (x > 12) {
  // Do code bis
} else {
  // Do code ter
}
```

> **Attention :**
> En C un 0 est considéré comme `false` et toute autre valeur numérique `true`

Pour faire une boucle on peut utiliser un `while (cond) {}` qui exécute le code tant que la condition est valide

On peut utiliser `do {} while (cond)` qui exécute une fois puis tant que la condition est vérifiée

Il est aussi possible d'utiliser `for (...) {}`, de la manière suivante :

```c
// De 0 à n - 1
for (int i = 0; i < n; i++) {

}

// De 0 à n - 1 tant que cond
for (int i = 0; i < n && cond; i++) {

}
```

A noter qu'en C il est possible de modifier la valeur de `i` et donc de sortir plus tôt de la boucle

Il est possible de sortir d'une boucle avec `break`, ou de passer à l'itération suivante avec `continue`

I.4 Fonctions

Pour définir une fonction on écrit :

```c
int my_func(int a, int b) {
  // Do code
  return 1;
}
```

Si on ne prend pas d'arguments on écrit `int my_func(void) {}` et si on ne veut rien renvoyer on utilise `void my_func(...) {}`

Ainsi pour appeller une fonction on fait :

```c
int resp = my_func(12, 14);
```

> **Attention :**
> Les variables sont copiées lors de l'appel de fonction

On peut déclarer une fonction avant de donner son code mais juste sa signature avec :

```c
int my_func(int);
```

I.5 Tableaux en C

Le type d'un tableau en C est `type[]` ou `* type`

Pour initialiser un tableau on a les manières suivantes :

```c
int[4] test = {0, 1, 2, 3}; // Initialise un tableau de taille 4 avec 0,1,2,3
int[] test = {0, 1, 2, 3}; // Initialise un tableau avec 0,1,2,3 (avec 4 éléments)
int[4] test = {0, 1}; // Initialise un tableau de taille 4 avec 0,1,0,0 (les autres valeurs sont à 0)
```

Il n'est pas obligé de donner la taille d'un tableau elle sera déterminée au moment de l'exécution

> **Attention :**
> Si on dépasse du tableau C ne prévient pas mais s'autorise à faire n'importe quoi

Pour affecter dans une case de tableau on fait :

```c
test[1] = test[2] // On met dans la case 1 la valeur de la case 2
```

Pour faire des tableaux de tableaux on fait :

```c
int[4][4] test = {{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11}, {12, 13, 14, 15}}; // Initialise un tableau de taille 4x4 avec les valeurs
int[4][4] test ={ {0} }; // Initialise un tableau de taille 4x4 avec des 0
int [][4] test = { {0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11} }; // Initialise un tableau de taille 3x4 avec les valeurs
```

Comme pour les tableaux on peut initialiser partiellement un tableau de tableaux

I.6 Pointeurs

Toute variable en C est une adresse mémoire, on peut donc récupérer cette adresse avec `&` :

```c
int a = 12;
// b est l'adresse mémoire de a
int * b = &a;
```

Il est possible de récupérer la valeur d'une adresse mémoire avec `*` (**déréférencement**) :

```c
int a = 12;
int * b = &a;
// c est la valeur de a
int c = *b;
```

On remarque donc que un pointeur a un type `type * var`

Il est aussi possible de prendre l'adresse d'un pointeur, ainsi on aura un `type ** var` (généralisable…)

Si on ne connaît pas l'adresse d'un pointeur on peut le déclarer avec `type * var = NULL`

> **Attention :**
> Il ne faut SURTOUT PAS déréférencer un pointeur `NULL`, ou on aura une erreur de segmentation (segmentation fault)

Il est possible d'allouer de la mémoire avec `malloc` :

```c
int * a = malloc(sizeof(int));

int * tab = malloc(4 * sizeof(int)); // Alloue un tableau de 4 éléments

int * tab2 = malloc(4 * sizeof(*tab2)); // Alloue un tableau de 4 éléments
```

L'appel à `malloc` renvoie un pointeur, et un pointeur `NULL` si il n'y a pas assez de mémoire (il peut donc être judicieux de vérifier si le pointeur est `NULL` sur des grosses allocations)

L'appel à `sizeof` renvoie la taille en octets de l'élément passé en argument, on peut passer un type ou une variable.

Après utilisation de `malloc` il est important de libérer la mémoire avec `free` quand on a fini d'utiliser la mémoire :

```c
int * a = malloc(sizeof(int));

// Do code

free(a);
```

> **Attention :**
> Il est important de libérer la mémoire après utilisation pour éviter les fuites mémoires (memory leaks) ou on finit avec un bluescreen

Il est ainsi possible de créer un tableau avec un malloc en modifiant la taille du tableau :

```c
int * tab = malloc(4 * sizeof(int)); // Alloue un tableau de 4 éléments
```

On pourra donc utiliser `tab[0]`, `tab[1]`, `tab[2]` et `tab[3]`…

Quand on passe un tableau à une fonction on passe un pointeur, ainsi on peut modifier le tableau dans la fonction

> **Attention :**
> Ainsi une fonction NE PEUT renvoyer un tableau créé normalement, il faut ABSOLUMENT renvoyer un tableau qui a été alloué avec `malloc`

Les tableaux, et notamment les cases des tableaux étant des pointeurs, on peut récupérer l'adresse d'une case de tableau avec `&` :

```c
int tab[4] = {0, 1, 2, 3};

int * a = &tab[2]; // a est l'adresse de la case 2
```

Il est intéressant de noter que `tab[2]` est équivalent à `*(tab + 2)` mais que l'arithmétique des pointeurs n'est pas au programme et qu'elle permet d'avoir des erreurs plus facilement

Il est aussi possible de faire des tableaux de tableaux avec des pointeurs :

```c
int ** tab = malloc(4 * sizeof(int *)); // Alloue un tableau de 4 pointeurs
```

Enfin on peut aussi passer une fonction en argument d'une autre fonction :

```c
int my_func(int (*func)(int, int), int a, int b) {
  return func(a, b);
} // my_func prend une fonction en argument qui prend deux entiers et renvoie un entier
```

I.7 Types construits

A faire (Structures et unions)

II Introduction au OCaml

II.1 Expressions

En OCaml on retrouve les types `int`, `float` (qui correspond au `double` du C) et `bool`.

Pour les opérations arithmétiques **sur les entiers** on a :

Opération	En OCaml
Addition	`a + b`
Soustraction	`a - b`
Multiplication	`a * b`
Division	`a / b`
Modulo	`a mod b`

Pour les opérations arithmétiques **sur les flottants** on a :

Opération	En OCaml
Addition	`a +. b`
Soustraction	`a -. b`
Multiplication	`a *. b`
Division	`a /. b`

Exponentiation	`a ** b`

On notera que le modulo n'est pas défini pour les flottants et que l'exponentiation est définie pour les flottants

On dispose aussi des fonctions mathématiques classiques, comme `sin`, `cos`, `tan`, `exp`, `log`, `sqrt`, `abs`, `acos`, `asin`, `atan`... avec `log` la fonction logarithme népérien

Comme en C on a les opérateurs binaires `&&` (et logique), `||` (ou logique) et `not` (négation de l'expression suivante)

Pour faire des comparaisons **sur des valeurs** on a = pour l'égalité, <> pour la différence, et >, >=, <=, < pour les comparaisons

> **Attention :**
> En OCaml un == est une comparaison de référence (d'étiquette), il ne faut pas l'utiliser pour comparer des valeurs, et de même pour `!=`

II.2 Typage fort

On a pu remarquer notamment sur les entiers et float que le typage est fort : aucune conversion implicite n'est faite c'est à l'utilisateur de le faire

Il n'est donc pas possible de faire `1 +. 2` mais il faut faire `1. +. 2.`

Pour passer d'un type à un autre on utilise les fonctions `int_of_float`, `float_of_int`, `int_of_string`, `float_of_string`...

II.3 Définitions

En OCaml le principe de variable n'existe pas réellement, on a des constantes, on ne peut pas modifier une variable à proprement parler

Pour faire une **définition** on utilise `let` :

```ocaml
let a = 12;; (* Définit a comme étant 12 *)

let b = 12 + a;; (* Définit b comme étant 24 *)
```

Il est possible de redéfinir une variable, mais on ne modifie pas la variable mais on en crée une nouvelle :

```ocaml
let a = 12;;

let a = a + 1;; (* a est maintenant 13 *)
```

La mémoire est gérée différement qu'en C, par exemple avec le code suivant en C on a `b` qui est une copie de `a` :

```c
int a = 12;
int b = a;
```

Alors qu'en OCaml `b` est une référence à `a`, si on modifie `a` on modifie `b` et inversement :

```ocaml
let a = 12;;
let b = a;;
```

Il est possible de définir plusieurs variables en même temps :

```ocaml
let a = 12 and b = 14;; (* a est 12 et b est 14 *)
```

Il est aussi possible de faire des variables locales en utilisant `in` :

```ocaml
let a = 12 and b = 14 in
    a + b;; (* a + b vaut 26, et a et b ne sont pas accessibles en dehors du bloc *)
```

Il est bien sûr possible d'imbriquer les `in` :

```ocaml
let a = 12 in
    let b = 14 in
        a + b;; (* a + b vaut 26, et a et b ne sont pas accessibles en dehors du bloc *)
```

II.4 Fonctions

Le OCaml est un langage fonctionnel, il est donc possible de définir des fonctions, de plusieurs manières différentes.

La première manière est de définir une fonction d'une manière semblable à une variable :

```ocaml
let sum a b = a + b;; (* Définit une fonction sum qui prend deux arguments a et b et renvoie a + b *)
```

Il existe un mot clé `function` (qui ne peut prendre qu'un argument) pour définir une fonction anonyme, ainsi on peut faire :

```ocaml
let sum = function a -> function b -> a + b;; (* Définit une fonction sum qui prend deux arguments a et b et renvoie a + b *)
```

On remarque que pour passer plusieurs arguments avec `function` on utilise plusieurs `function`, ce qui peut être fastidieux

Ainsi il existe le mot clé `fun` qui permet de définir une fonction de manière plus simple :

```ocaml
let sum = fun a b -> a + b;; (* Définit une fonction sum qui prend deux arguments a et b et renvoie a + b *)
```

Pour appeller une fonction on fait :

```ocaml
sum 12 14;; (* Renvoie 26 *)
```

Il faut faire attention au fait que chaque bloc est considéré comme un argument, ainsi on a les cas de figure suivants :

```ocaml
sum -12 12;; (* Erreur, on a -, 12 et 12 comme arguments *)
sum (-12) 12;; (* Renvoie 0 *)
```

> **Attention :**
> Il est important de bien mettre des parenthèses pour les arguments négatifs, ou pour des appels intermédiaires

OCaml va déterminer tout seul la signature de la fonction, ainsi on peut faire :

```ocaml
let sum a b = a + b;; (* Définit une fonction sum qui prend deux arguments a et b et renvoie a + b *)
(* sum : int -> int -> int *)
```

En analysant la signature de la fonction on peut voir que `sum` prend deux entiers et renvoie un entier

Mais on peut aussi faire du polymorphisme, ainsi on peut faire :

```ocaml
let min a b = if a < b then a else b;; (* Définit une fonction min qui prend deux arguments a et b et renvoie le minimum *)
(* min : 'a -> 'a -> 'a *)
```

Ainsi ici `min` prend deux arguments de même type et renvoie un argument du même type (il est aussi possible de faire du polymorphisme sur plusieurs types et d'avoir des `'b`, `'c`...)

Il peut arriver qu'une fonction ait des effets de bord, ainsi elle peut renvoyer le type `unit` :

```ocaml
let nothing a = ();; (* Définit une fonction nothing qui prend un argument a et ne renvoie rien *)
(* nothing : 'a -> unit *)
```

Il est aussi possible de ne pas prendre d'arguments :

```ocaml
let nothing = ();; (* Définit une fonction nothing qui ne prend pas d'arguments et ne renvoie rien *)
(* nothing : unit *)
```

> **Attention :**
> Pour appeller une fonction qui ne prend pas d'arguments il faut mettre des parenthèses, sinon on aura une erreur (ici `nothing ()`)

Mais le mot clé `fonction` a un avantage : il permet de faire des **match** qui vont être des conditions sur les arguments :

```ocaml
let my_func a = function
  | 0 -> 1 * a (* Si a est 0 on renvoie a *)
  | 1 -> 2 * a (* Si a est 1 on renvoie 2a *)
  | _ -> 3 * a;; (* Sinon on renvoie 3a *)
(* my_func : int -> int -> int *)
```

Ainsi le mot clé fonction avec un match permet de prendre un argument mais sans le nommer

Il est aussi possible de faire des motifs **gardés**, pour imposer une condition sur un motif avec `when` :

```ocaml
let my_func a = function
  | 0 when a > 0 -> 1 * a (* Si a est 0 et a > 0 on renvoie a *)
  | 0 -> -1 * a (* Si a est 0 et a <= 0 on renvoie -a *)
  | 1 -> 2 * a (* Si a est 1 on renvoie 2a *)
  | _ -> 3 * a;; (* Sinon on renvoie 3a *)
(* my_func : int -> int -> int *)
```

Il faut noter que les motifs sont examinés dans l'ordre, ainsi si on a plusieurs motifs qui correspondent on prend le premier qui correspond

Enfin il est possible de faire des fonctions récursives, pour cela on utilise le mot clé `rec` :

```ocaml
let rec fact = function
  | 0 -> 1
  | n -> n * fact (n - 1);;
(* fact : int -> int *)
```

II.5 Expressions plus complexes

Si on veut faire des expressions plus complexes on peut utiliser `if ... else if ... else ...` :

```ocaml
let my_func a =
  if a = 0 then
    1 * a (* Si a est 0 on renvoie a *)
  else if a = 1 then
    2 * a (* Si a est 1 on renvoie 2a *)
  else
    3 * a;; (* Sinon on renvoie 3a *)
(* my_func : int -> int *)
```

Si on veut faire des opérations plus complexes entre les `if ... else` on peut utiliser `begin ... end` ou `(...)` :

```ocaml
let my_func a =
  if a = 0 then
    begin
      let b = 1 in
      b * a (* Si a est 0 on renvoie a *)
    end
  else if a = 1 then
    (let b = 2 in b * a) (* Si a est 1 on renvoie 2a *)
  else
    3 * a;; (* Sinon on renvoie 3a *)
(* my_func : int -> int *)
```

Il est aussi possible de réaliser des filtrages sans le mot clé `function` :

```ocaml
let my_func a b =
  match b with
  | 0 -> 1 * a (* Si b est 0 on renvoie a *)
  | 1 -> 2 * a (* Si b est 1 on renvoie 2a *)
  | _ -> 3 * a;; (* Sinon on renvoie 3a *)
(* my_func : int -> int *)
```

On peut construire des n-uplets avec `(..., ...)`, et en OCaml on peut les déconstruire avec `let (..., ...) = ...`

A noter que les couples possèdent les fonctions `fst` et `snd` pour récupérer le premier et le second élément

```ocaml
let a = (12, 14);; (* a est un couple de 12 et 14 *)

print_int (fst a);; (* Affiche 12 *)
```

Il est donc possible de donner un couple à `fonction` ou `match` :

```ocaml
let my_func a =
  match a with
  | (0, 0) -> 1 (* Si a est (0, 0) on renvoie 1 *)
  | (1, 0) -> 2 (* Si a est (1, 0) on renvoie 2 *)
  | _ -> 3;; (* Sinon on renvoie 3 *)
(* my_func : int * int -> int *)
```

On remarque sur la signature qu'un n-uplet est défini par `type1 * type2 * ...` et qu'on peut donc définir des n-uplets de n'importe quel type (même avec des types différents)

II.6 Listes

Il peut être utile de définir des listes, pour cela on utilise le module `List`

II.7 Tableaux

II.8 Types construits

II.9 Programmation impérative

Structures de données

I Structures de données

II Piles, files, dictionnaires

III Arbres

IV Graphes

IV.1 Recherche de plus cours chemin

IV.1.a Graphes avec poids négatifs

Dans un graphe on dit que l'arc $u \rhd v$ est en **tension** si $\delta(v) > \delta(u) + w(u, v)$

L'approche de Ford est donc d'éliminer les arcs en tension

Tant qu'il existe des arcs en tension, on traite tous les arcs de E et on traite ceux en tension, on a donc une complexité $O(n \times p)$

Informatique théorique

I Bases

I.1 Fonctions

On dit qu'une fonction a des **effets de bord** si son exécution a des conséquences sur d'autres choses que ses variables locales

Une fonction est **déterministe** si le résultat est toujours le même avec les mêmes arguments

Une fonction est dite **pure** lorsqu'elle est déterministe et sans effets de bord

I.2 Complexité

On dit qu'un algorithme est en $O(f(n))$ **pire cas** si il existe une constante k telle que pour tout n assez grand, le nombre d'opérations est inférieur à $kf(n)$

On dit qu'un algorithme est en $\Omega(f(n))$ **meilleur cas** si il existe une constante k telle que pour tout n assez grand, le nombre d'opérations est supérieur à $kf(n)$

On dit qu'un algorithme est en $\Theta(f(n))$ **cas moyen** si il est en $O(f(n))$ et en $\Omega(f(n))$

On parle alors :

- $O(1)$ pour une complexité **constante**
- $O(\log(n))$ pour une complexité **logarithmique**
- $O(n)$ pour une complexité **linéaire**
- $O(n \log(n))$ pour une complexité **quasi–linéaire**
- $O(n^2)$ pour une complexité **quadratique**
- $O(k^n)$ pour une complexité **exponentielle**

I.3 Algorithmes de tri

En informatique on a souvent besoin de trier des listes, on a plusieurs algorithmes pour cela

> **Tri stable :**
> Un tri est dit **stable** si l'ordre des éléments égaux est conservé

I.3.a Tri par sélection

Le par sélection est l'algorithme le plus simple de tri, on prend le minimum et on le met en tête de liste.

Ainsi on a un invariant de boucle : la liste est triée jusqu'à l'indice i

Pour l'implémenter en C on fait :

```c
void selection_sort(int arr[], int n) {
  for (int i = 0; i < n; i++) {
    // Les i premiers éléments sont bien triés
    int min_i = i;

    for (int j = i+1; j<n; j++) {
      if (arr[j] < arr(min_i)) {
        min_i = j;
      }
    }

    // On échange les éléments en i et min_i
    int tmp = arr[i];
    arr[i] = arr[min_i];
    arr[min_i] = tmp;
  }
}
```

Le tri par sélection est en $O(n^2)$, on a n comparaisons pour le premier élément, $n-1$ pour le second, etc.

Le tri par sélection a donc comme inconvéniant d'avoir une complexité quadratique et de ne pas être stable

I.3.b Tri bulle

Le tri bulle est un algorithme de tri simple, on compare les éléments deux à deux et on les échange si ils ne sont pas dans le bon ordre, comme des bulles qui remontent à la surface

On peut réaliser un tri pierre en descendant les éléments au lieu de les monter

Pour l'implémenter en C on fait :

```c
let bubble_sort(int arr[], int n) {
  for (int i = 0; i < n; i++) {
    // Les i premiers éléments sont bien placés
    int k_last_perm = n-1;
    int smallest = arr[n-1];

    for (int j = n-1; j > i; j--) {
      if (arr[j-1] <= smallest) {
        // On change de bulle
        arr[j] = smallest;
        smallest = arr[j-1];
      } else {
        // On fait descendre la bulle
        arr[j] = arr[j-1];
        k_last_perm = j - 1;
      }
    }

    arr[i] = smallest;
    // On n'a pas besoin de regarder les éléments entre i+1 et k_last_perm car on n'a fait aucune modification
    i = k_last_perm + 1;
  }
}
```

Le tri bulle a une complexité en $O(n^2)$, on a n comparaisons pour le premier élément, $n-1$ pour le second, etc, mais cette complexité est rarement atteinte. De plus le tri bulle est stable

I.3.c Tri par insertion

Le tri par insertion est un algorithme de tri qui consiste à insérer un élément à sa place dans une liste triée (les éléments précédents sont déjà triés mais pas forcément à leur place définitive)

Pour l'implémenter en C on fait :

```c
int insertion_sort(int arr[], int n) {
  for (int i = 0; i < n; i++) {
    // Les i premiers éléments sont bien triés
    int j = i;
    int elem = arr[i];

    for (; j>0 && elem < arr[j-1]; j--) {
      arr[j] = arr[j-1];
    }

    arr[j] = elem;
  }
}
```

Le tri par insertion a une complexité en $O(n^2)$, on a n comparaisons pour le premier élément, $n-1$ pour le second, etc, mais cette complexité est rarement atteinte. De plus le tri par insertion est stable

I.3.d Tri rapide

Le tri rapide est un algorithme de tri qui consiste à choisir un pivot et à partitionner la liste en deux parties, les éléments plus petits que le pivot et les éléments plus grands que le pivot, on réitère sur les deux listes

Pour l'implémenter en C on fait :

```c
void quick_sort(int * arr, int n) {
  if (n <= 1) { // Déjà trié
    return;
  }

  int pivot = partition(arr, n);
  quick_sort(arr, pivot);
  quick_sort(&arr[pivot+1], n-pivot-1);
}
```

Tout l'intérêt du tri rapide est dans la fonction `partition` qui permet de partitionner la liste en deux parties

On utilise la partition de Lomuto, qui consiste à garder le pivot en première position, puis les éléments plus petits que le pivot, puis les éléments plus grands que le pivot et enfin ceux qui ne sont pas encore triés

```c
int partition(int arr[], int n) {
  int pivot = arr[0];
  int p = 1;

  for (int i = 1; i<n; i++) {
    if (arr[i] < pivot) {
      arr_swap(arr, i, p); // On échange les éléments i et p
      p++;
    }
  }

  arr_swap(arr, 0, p-1); // On échange le pivot et le dernier élément plus petit que le pivot
  return p-1;
}
```

I.4 Algorithmes classiques

I.4.a Dichotomie

La dichotomie est un algorithme de recherche efficace : on prend le milieu de la liste et on regarde si l'élément est plus grand ou plus petit, on réitère sur la moitié de la liste etc…

Pour l'implémenter en C on fait de manière récursive :

```
let index(int * arr, int n, int elem) {
  if (n == 0) {
    return -1; // On ne peut pas trouver
  }

  int m = n/2;

  if (arr[m] == elem) { // On a trouvé!
    return m;
  } else if (arr[m] > m) { // L'élément se situe peut être dans la partie gauche
    return index(arr, m, elem);
  } else { // L'élément se situe peut être dans la partie droite
    int idx = index(&arr[m+1], n-m-1, elem);

    if (idx != -1) {
      idx += m+1;
    }

    return idx;
  }
}
```

On peut aussi faire de manière itérative :

```
let index(int * arr, int n, int elem) {
  int l = 0, r = n;

  while (l < r) { // On recherche dans le tableau avec deux compteurs
    int m = (l+r)/2;

    if (arr[m] == val) { // On a trouvé!
      return m;
    } else if (arr[m] > m) { // L'élément se situe peut être dans la partie
gauche
      r = m;
    } else { // L'élément se situe peut être dans la partie droite
      l = m + 1;
    }
  }

  return -1; // Pas trouvé!
}
```

L'avantage de la dichotomie est qu'elle a une complexité en $O(\log(n))$: elle permet donc une recherche efficace

II Récursion

III Stratégies algorithmiques

III.1 Algorithmes gloutons

III.2 Diviser pour régner

Le **tri fusion** est un tri en $\Theta(n \log(n))$, on sépare les listes puis on les trie en interne et on fusionne les deux listes triées

Pour l'implémenter en OCaml on fait :

```ocaml
let rec partition = function
  | h1::h2::t -> let l,r = partition t in h1::l, h2::r
  | lst -> lst, [];;

let rec merge l1 l2 = match l1,l2 with
  | (h1::t1), (h2::t2) when h1 <= h2 -> h1::(merge t1 l2)
  | (h1::t1), (h2::t2) -> h2::(merge l1 t2)
  | l1, [] -> l1;;

let rec fusion_sort lst = match split lst with
  | lst, [] -> lst
  | l1, l2 -> merge (fusion_sort l1) (fusion_sort l2)
```

Analysons l'algorithme du tri fusion, en regardant le nombre de comparaisons on retrouve une complexité en $\Theta(n \log(n))$ pour ces étapes

Plus mathématiquement on a pour $n \geq 2$, $u_{\lfloor \frac{n}{2} \rfloor} + u_{\lceil \frac{n}{2} \rceil} + \frac{n}{2} \leq u_n \leq u_{\lfloor \frac{n}{2} \rfloor} + u_{\lceil \frac{n}{2} \rceil} + n$ d'où on a $u_n = u_{\lfloor \frac{n}{2} \rfloor} + u_{\lceil \frac{n}{2} \rceil} + \Theta(n)$

A faire (Suites récurrentes d'ordre 1)

Suites "diviser pour régner" :
Soit a_1, a_2 deux réels positifs vérifiant $a_1 + a_2 \geq 1$ et $(b_n)_{n\in\mathbb{N}}$ une suite positive et croissante et $(u_n)_{n\in\mathbb{N}}$ une suite vérifiant :

$$u_n = a_1 u_{\lfloor \frac{n}{2} \rfloor} + a_2 u_{\lceil \frac{n}{2} \rceil} + b_n$$

Ainsi en posant $\alpha = \log_2(a_1 + a_2)$, on a :

- Si $(b_n) = \Theta(n^\alpha)$, alors $(u_n) = \Theta(n^\alpha \log(n))$
- Si $(b_n) = \Theta(n^\beta)$ avec $\beta < \alpha$, alors $(u_n) = \Theta(n^\alpha)$
- Si $(b_n) = \Theta(n^\beta)$ avec $\beta > \alpha$, alors $(u_n) = \Theta(n^\beta)$

Attention :
A savoir que si on retombe sur une relation de récurrence connue on peut donner directement la complexité

Pour l'implémenter en C on fait de la manière suivante :

A faire (Réecrire)

```c

```

IV SQL

IV.1 Généralités

En SQL on stocke des entités avec des attributs et à chaque attribut on lui associe un type

On peut définir des relations entre les différentes entités

On stocke ces entités dans des tables : dans chaque table on stocke une entité

Il est possible de garder une case vide en plaçant un `NULL` dans la case

IV.2 Requêtes

Pour récupérer des données (projections) dans une table on a :

```sql
# Seulement les colonnes spécifiées
SELECT col1, ..., coln FROM table

# Toutes les colonnes
SELECT * FROM table

# Toutes les colonnes mais sans doublon
SELECT DISTINCT * FROM table
```

Ainsi on récupère toutes les lignes de la table avec ces projections

On peut aussi faire une sélection sur un critère :

```sql
SELECT * FROM table WHERE bool
```

Les opérations booléennes sont les suivantes :

- `col > a`/`col < a`/`col = a` pour faire des comparaisons
- `col IN (a, b, c)` pour savoir si la cellule est dans un ensemble de valeur
- `col IS NULL`/`IS NOT NULL` pour savoir si la cellule est nulle ou non
- `col LIKE '% Text %'` pour regarder si `Text` est dans la chaine de caractère de la cellule

On peut combiner les critères avec `AND`/`OR`/`NOT`

Il est possible de sélectionner un attribut non projeté

Pour ordonner les résultats on ordonne en utilisant

```sql
# Triés par valeur croissante
SELECT * FROM table ORDER BY col

# Triés par valeur décroissante
SELECT * FROM table ORDER BY col DESC
```

Pour limiter le nombre de valeurs on utilise

```sql
# On prend au maximum 3 éléments
SELECT * FROM table LIMIT 3

# On prend au maximum 3 éléments mais sans les 2 premiers
SELECT * FROM table LIMIT 3 OFFSET 2
```

IV.3 Fonctions

On peut compter le nombre d'entités qui vont être renvoyées

```sql
# Nombre d'éléments dans la table
SELECT COUNT(*) FROM table
```

On peut compter sur une colonne spécifique avec `COUNT(col1, ..., col2)`, les cases ne sont pas comptées si `NULL`,

Il est aussi possible de compter le nombre de valeur distinctes pour une colonne :

```sql
SELECT COUNT(DISTINCT col) FROM table
```

On peut utiliser `MAX`, `MIN`, `SUM` et `AVG` pour avoir du préprocessing, il est aussi possible d'avoir la moyenne en faisant `SUM(col)/COUNT(*)`

> **Attention :**
> On ne peut mélanger une colonne et une fonction dans la projection

Il est possible de grouper les valeurs

```sql
# Renvoie des groupes des valeurs de col
SELECT col FROM table GROUP BY col
```

> **Attention :**
> Il n'est pas possible d'utiliser `GROUP BY` sur des colonnes non groupées

Par contre les fonctions agissent sur chaque groupe, ainsi il est possible d'écrire

```sql
# Renvoie des groupes des valeurs de col avec le nombre d'occurence de cette valeur dans la table
SELECT col, COUNT(*) FROM table GROUP BY col
```

Pour sélectionner des groupes on peut utiliser :

```sql
# Renvoie des groupes des valeurs de col si la valeur minimale du groupe dans la colonne col2 est supérieure à x avec la valeur minimale de col2 de ce groupe dans la table
SELECT col1, MIN(col2) FROM table GROUP BY col1 HAVING MIN(col2) > x
```

Les opérations sont executées dans cet ordre :

- WHERE
- GROUP BY
- HAVING
- ORDER BY
- LIMIT/OFFSET
- SELECT à la fin bien qu'on le mette en tête de la requête

Ainsi une clause valide est

```sql
SELECT * WHERE cond GROUP BY col HAVING cond2 ORDER BY col2 LIMIT 3 OFFSET 2
```

IV.4 Sous requêtes

Il est possible d'écrire une sous requête :

```sql
# Ici on sélectionne seulement les éléments donc la valeur col est supérieure à la valeur moyenne de col
SELECT * FROM table WHERE col > (SELECT AVG(col) FROM table)
```

Il est donc aussi possible d'utiliser cette syntaxe avec des `IN`

```sql
# Ici on va sélectionner seulement les lignes dont la valeur de col correspond à la condition cond
SELECT * FROM table WHERE col IN (SELECT DISTINCT col FROM * WHERE cond)
```

Le `col AS nameBis` permet de renommer une colonne

Si on reçoit un tableau, on peut sélectionner dans les réponses

```sql
# Ainsi on renvoie la moyenne d'une colonne col2 telle que ses éléments vérifient la condition
SELECT AVG(resp.colName) FROM (SELECT col1, col2 AS colName FROM table WHERE cond) AS resp
```

IV.5 Combiner les tables

Il est possible de combiner des tables

```sql
# Sélectionne dans le produit cartésien des deux tables
SELECT * FROM table1, table2
```

Mais en faisant ça on va avoir plein de lignes qui n'ont pas de sens, ainsi si on veut garder seulement les lignes qui nous intéressent

```sql
# Sélectionne dans le produit cartésien des deux tables seulement les éléments donc la col1 de la table 1 est le même que celui de la col 2 de la table 2
SELECT * FROM table1, table2 WHERE table1.col1 = table2.col2
```

Mais pour éviter ça on peut aussi de manière équivalente écrire :

```sql
# On sélectionne les éléments de la table1 en ajoutant la table2 si la condition est vérifiée, le ON est donc un WHERE
SELECT * FROM table1 JOIN table2 ON table1.col1 = table2.col2
```

Le produit cartésien n'est donc qu'une manière de jointure

On peut aussi utiliser le `LEFT JOIN` qui permet de garder un élément de la première table même si il n'a pas d'équivalent dans la seconde table

```sql
# On sélectionne les éléments de la table1 en concaténant les éléments dont la condition est vérifiée, et rien si il n'y a pas d'équivalent
SELECT * FROM table1 LEFT JOIN table2 ON table1.col1 = table2.col2
```

On peut faire l'union de deux requêtes

```sql
# On a les éléments qui vérifient la cond1 ou cond2
SELECT * FROM table WHERE cond1 UNION SELECT * FROM table WHERE cond2
```

> **Attention :**
> Pour utiliser l'union il faut juste que les types sont compatibles mais pas les noms de colonne

On peut aussi faire l'intersection de deux requêtes

```sql
# On a les éléments qui vérifient la cond1 et cond2
SELECT * FROM table WHERE cond1 INTERSECT SELECT * FROM table WHERE cond2
```

On peut faire des différences ensemblistes avec `MINUS` ou `EXCEPT`

IV.6 Créer une BDD

Pour créer une base de données on utilisera

```sql
CREATE TABLE IF NOT EXISTS table (
  col1 TYPE1,
  col2 TYPE2,
  col3 TYPE3
)
```

Si on veut limiter le nombre de caractères, on peut le préciser entre parenthèses, par exemple `VARCHAR(6)` pour avoir des chaînes d'au plus 6 caractères

On peut définir une **clé primaire** qui ne peut avoir 2 fois la même valeur, on indiquera `PRIMARY KEY` après le type :

```sql
CREATE TABLE IF NOT EXISTS table (
  col1 TYPE1 PRIMARY KEY,
  ...
)
```

Les autres attibuts seront dépendant de la clé primaire : si on connaît la clé primaire on peut connaître les autres valeurs associées à la liste

Si on a une clé primaire dans un GROUP BY autorise à projeter sur tous les éléments (pas comme précédemment)

Il y a au plus une clé primaire par table, et une valeur `NULL` ne peut être une valeur pour cette case

On peut définir un clé étrangère qui vont être des liens entre les différentes tables

```sql
CREATE TABLE IF NOT EXISTS table (
  ...,
  FOREIGN KEY (col) REFERENCES table(col)
)
```

Il est aussi possible de modifier une table en utilisant `ALTER TABLE`

Pour insérer dans une table on utilise :

```sql
INSERT INTO table (col1, col2, col3) VALUES (value1, value2, value3)
```

On peut modifier un élement :

```sql
UPDATE table SET col1 = value WHERE cond
```

On peut aussi supprimer un élement :

```sql
DELETE FROM table WHERE cond
```

IV.7 Type entités

Les types entités sont liées par des types associations

A faire (Cardinalité)

On précise les cardinalités :

- $1, 1$ en liaison avec une et une seule entité
- $1, n$ en liaison avec au moins une autre entité
- $0, 1$ en liaison avec au plus une autre entité
- $0, n$ en liaison avec un nombre quelconque d'entités

V Algorithmes des textes

V.1 Bases

En C on représente les chaînes de caractère par des `char *` avec un `\0` à la fin de la chaîne (donc un `0` dans la dernière case)

On peut utiliser `strlen` pour connaître la longueur d'une chaîne

En OCaml on a le module `String` qui permet de manipuler les chaînes de caractères et les chaînes de caractères sont immuables

On peut concaténer des chaînes avec `^` et on peut accéder à un caractère avec `.[i]`

On peut aussi utiliser `String.length` pour connaître la longueur d'une chaîne (en $O(1)$)

Pour lire tous les éléments d'une chaine en C on fera :

```c
for (int i = 0; str[i] != '\0'; i++) {
  // Do code
}
```

En C un `char` correspond à un entier entre -128 et 127, ainsi on peut écrire `int a = (int) 'a'` (le cast n'est pas obligatoire) pour avoir 97

A noter que `'` est un caractère et `"` est une chaîne de caractère

> **Attention :**
> On ne fera pas une boucle for avec `strlen` car on va recalculer la longueur de la chaîne à chaque itération

V.2 Algorithmes

Imaginons que l'on veuille trouver si une chaîne de caractères n'est constituée que de mots valides (en supposant que la fonction `is_word` existe) :

```c
void is_sentence(char * s) {
  if (s[0] == '\0') {
    return;
  }

  int n = strlen(s);
  int * arr = malloc((n+1) * sizeof(*arr));
  arr[0] = 0;

  for (int i = 1; i <= n; i++) {
    arr[i] = -1; // On initialise à false car le malloc ne le fait pas
    char tmp = s[i];
    s[i] = '\0';
    for (int j = i-1; arr[i] != -1 && j >= 0; --j) {
      if (arr[j] != -1 && is_word(&s[j])) {
        arr[i] = j;
      }
    }
    s[i] = tmp;
  }
  // Le tableau arr contient l'indice du début du mot précédent (ou -1 si il n'y en a pas)
  free(arr);
}
```

Il est intéressant de mémoïser cette fonction pour éviter de recalculer plusieurs fois la même chose

Pour déterminer si une chaîne de caractères est un mot, on a plusieurs approches, en considérant N mots et p la longueur de la chaîne :

- Approche naïve : On compare pour chaque mot $O(N \times p)$
- Approche dicothomique : On trie les mots et on fait une recherche dichotomique $O(p \times \log(N))$
- On utilise un *TRIE*, c'est à dire un arbre où chaque noeud est une lettre et chaque branche est un mot, on a une complexité en $O(p)$ (selon l'implémentation de chaque noeud et de son stockage), on privilégiera de stocker dans un dictionnaire les mots. Une autre solution est de stocker tous les mots dans un dictonnaire et de regarder si le mot est dedans

V.3 Recherche de motifs

Liste d'algorithmes

Table des matières