



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 :

 I Introduction au C	3
 II Introduction au OCaml	7

Structures de données :

 I Structures de données	8
 II Piles, files, dictionnaires	8
 III Arbres	8
 IV Graphes	8

Informatique théorique :

 I Bases	9
 II Récursion	13
 III Stratégies algorithmiques	13
 IV SQL	14

Introduction

I Introduction au C

I.1 Variables

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

```
1 int mango = 0;
```



Il est possible de définir plusieurs variables en même temps :

```
1 int banana = apple = 12;
```



I.2 Opérateurs

On a les opérations arithmétiques suivantes :

Opération	En C
Addition	<code>a + b</code>
Soustraction	<code>a - b</code>
Multiplication	<code>a * b</code>
Division	<code>a / b</code>
Modulo	<code>a % b</code>

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 :

```

1  if (x == 1) {
2      // Do code
3  } else if (x > 12) {
4      // Do code bis
5  } else {
6      // Do code ter
7  }

```



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 :

```

1  // De 0 à n - 1
2  for (int i = 0; i < n; i++) {
3
4  }
5
6  // De 0 à n - 1 tant que cond
7  for (int i = 0; i < n && cond; i++) {
8
9  }

```

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 :

```

1  int my_func(int a, int b) {
2      // Do code
3      return 1;
4  }

```

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 appeler une fonction on fait :

```

1  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 :

```
1 int my_func(int);
```

1.5 Tableaux en C

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

Pour initialiser un tableau on a les manières suivantes :

```
1 int[4] test = {0, 1, 2, 3}; // Initialise un tableau de taille 4 avec 0,1,2,3
2 int[] test = {0, 1, 2, 3}; // Initialise un tableau avec 0,1,2,3 (avec 4
3 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 :

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

Pour faire des tableaux de tableaux on fait :

```
1 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
2 int[4][4] test = { {0} }; // Initialise un tableau de taille 4x4 avec des 0
3 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

1.6 Pointeurs

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

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

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

```
1 int a = 12;
2 int * b = &a;
3 // c est la valeur de a
4 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` :

```
1  int * a = malloc(sizeof(int));
2
3  int * tab = malloc(4 * sizeof(int)); // Alloue un tableau de 4 éléments
4
5  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 :

```
1  int * a = malloc(sizeof(int));
2
3  // Do code
4
5  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 :

```
1  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 & :

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



II Introduction au OCaml

Structures de données

I Structures de données

II Piles, files, dictionnaires

III Arbres

IV Graphes

IV.1 Recherche de plus courts chemin

IV.1.a Graphes avec poids négatifs

Dans un graphe on dit que l'arc $u \triangleright v$ 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 tri 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 :

```

1 void selection_sort(int arr[], int n) {
2     for (int i = 0; i < n; i++) {
3         // Les i premiers éléments sont bien triés
4         int min_i = i;
5
6         for (int j = i+1; j < n; j++) {
7             if (arr[j] < arr(min_i)) {
8                 min_i = j;
9             }
10        }
11
12        // On échange les éléments en i et min_i
13        int tmp = arr[i];
14        arr[i] = arr[min_i];
15        arr[min_i] = tmp;
16    }
17 }

```

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 inconvénient d'avoir une complexité quadratique et de ne pas être stable

1.3.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 :

```

1  let bubble_sort(int arr[], int n) {
2      for (int i = 0; i < n; i++) {
3          // Les i premiers éléments sont bien passés
4          int k_last_perm = n-1;
5          int smallest = arr[n-1];
6
7          for (int j = n-1; j > i; j--) {
8              if (arr[j-1] <= smallest) {
9                  // On change de bulle
10                 arr[j] = smallest;
11                 smallest = arr[j-1];
12             } else {
13                 // On fait descendre la bulle
14                 arr[j] = arr[j-1];
15                 k_last_perm = j - 1;
16             }
17         }
18
19         arr[i] = smallest;
20         // On n'a pas besoin de regarder les éléments entre i+1 et k_last_perm car on
21         // n'a fait aucune modification
22         i = k_last_perm + 1;
23     }
24 }

```

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 :

```

1  int insertion_sort(int arr[], int n) {
2      for (int i = 0; i < n; i++) {
3          // Les i premiers éléments sont bien triés
4          int j = i;
5          int elem = arr[i];
6
7          for (; j>0 && elem < arr[j-1]; j--) {
8              arr[j] = arr[j-1];
9          }
10
11         arr[j] = elem;
12     }
13 }

```

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 :

```
1 void quick_sort(int * arr, int n) {  
2     if (n <= 1) { // Déjà trié  
3         return;  
4     }  
5  
6     int pivot = partition(arr, n);  
7     quick_sort(arr, pivot);  
8     quick_sort(&arr[pivot+1], n-pivot-1);  
9 }
```

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

 A faire (Écrire la fonction partition)

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 :

```
1 let index(int * arr, int n, int elem) {  
2     if (n == 0) {  
3         return -1; // On ne peut pas trouver  
4     }  
5  
6     int m = n/2;  
7  
8     if (arr[m] == elem) { // On a trouvé!  
9         return m;  
10    } else if (arr[m] > elem) { // L'élément se situe peut être dans la partie gauche  
11        return index(arr, m, elem);  
12    } else { // L'élément se situe peut être dans la partie droite  
13        int idx = index(&arr[m+1], n-m-1, elem);  
14  
15        if (idx != -1) {  
16            idx += m+1;  
17        }  
18  
19        return idx;  
20    }  
21 }
```

On peut aussi faire de manière itérative :

```

1  let index(int * arr, int n, int elem) {
2      int l = 0, r = n;
3
4      while (l < r) { // On recherche dans le tableau avec deux compteurs
5          int m = (l+r)/2;
6
7          if (arr[m] == val) { // On a trouvé!
8              return m;
9          } else if (arr[m] > m) { // L'élément se situe peut être dans la partie
10             gauche
11                 r = m;
12             } else { // L'élément se situe peut être dans la partie droite
13                 l = m + 1;
14             }
15         }
16
17         return -1; // Pas trouvé!
18     }

```

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 :


```

1  let rec partition = function
2      | h1::h2::t -> let l,r = partition t in h1::l, h2::r
3      | lst -> lst, [];;
4
5  let rec merge l1 l2 = match l1,l2 with
6      | (h1::t1), (h2::t2) when h1 <= h2 -> h1::(merge t1 l2)
7      | (h1::t1), (h2::t2) -> h2::(merge l1 t2)
8      | l1, [] -> l1;;
9
10 let rec fusion_sort lst = match split lst with
11     | lst, [] -> lst
12     | 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 (Réécrire)

1



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 :

```

1  # Seulement les colonnes spécifiées
2  SELECT col1, ..., coln FROM table
3
4  # Toutes les colonnes
5  SELECT * FROM table
6
7  # Toutes les colonnes mais sans doublon
8  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 :

```

1  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 chaîne 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

```

1  # Triés par valeur croissante
2  SELECT * FROM table ORDER BY col
3
4  # Triés par valeur décroissante
5  SELECT * FROM table ORDER BY col DESC

```

Pour limiter le nombre de valeurs on utilise

```

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

```

IV.3 Fonctions

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

```

1  # Nombre d'éléments dans la table
2  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 :

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



On peut utiliser MAX, MIN, SUM et AVG pour avoir du preprocessing, 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

```
1 # Renvoie des groupes des valeurs de col
2 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

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



Pour sélectionner des groupes on peut utiliser :

```
1 # 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
2 SELECT col1, MIN(col2) FROM table GROUP BY col1 HAVING MIN(col2) > x
```



Les opérations sont exécutées 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

```
1 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 :


```

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

```

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

```

1 # Ici on va sélectionner seulement les lignes dont la valeur de col correspond à la condition cond
2 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

```

1 # Ainsi on renvoie la moyenne d'une colonne col2 telle que ses éléments vérifient la condition
2 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

```

1 # Sélectionne dans le produit cartésien des deux tables
2 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

```

1 # 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
2 SELECT * FROM table1, table2 WHERE table1.col1 = table2.col2

```

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

```

1 # On sélectionne les éléments de la table1 en ajoutant la table2 si la condition est vérifiée, le ON est donc un WHERE
2 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

```

1 # 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
2 SELECT * FROM table1 LEFT JOIN table2 ON table1.col1 = table2.col2

```

On peut faire l'union de deux requêtes

```

1 # On a les éléments qui vérifient la cond1 ou cond2
2 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

```

1 # On a les éléments qui vérifient la cond1 et cond2
2 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

```

1 CREATE TABLE IF NOT EXISTS table (
2     col1 TYPE1,
3     col2 TYPE2,
4     col3 TYPE3
5 )

```

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 :

```

1 CREATE TABLE IF NOT EXISTS table (
2     col1 TYPE1 PRIMARY KEY,
3     ...
4 )

```

Les autres attributs 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

```

1 CREATE TABLE IF NOT EXISTS table (
2     ...,
3     FOREIGN KEY (col) REFERENCES table(col)
4 )

```

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

Pour insérer dans une table on utilise :

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



On peut modifier un élément :

1 `UPDATE table SET col1 = value WHERE cond`




On peut aussi supprimer un élément :

1 `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

Liste d'algorithmes













































Tri par sélection 	9
Tri bulle 	10
Tri par insertion 	11
Tri rapide 	12
Dichotomie (Récursive) 	12
Dichotomie (Impérative) 	12
Tri fusion 	13
Tri fusion 	14

Table des matières

 I Introduction au C	3
 I.1 Variables	3
 I.2 Opérateurs	3
 I.3 Structures de contrôle	3
 I.4 Fonctions	4
 I.5 Tableaux en C	5
 I.6 Pointeurs	5
 II Introduction au OCaml	7
 I Structures de données	8
 II Piles, files, dictionnaires	8
 III Arbres	8
 IV Graphes	8
 IV.1 Recherche de plus courts chemin	8
 IV.1.a Graphes avec poids négatifs	8
 I Bases	9
 I.1 Fonctions	9
 I.2 Complexité	9
 I.3 Algorithmes de tri	9
 I.3.a Tri par sélection	9
 I.3.b Tri bulle	10
 I.3.c Tri par insertion	11
 I.3.d Tri rapide	12
 I.4 Algorithmes classiques	12
 I.4.a Dichotomie	12
 II Récursion	13
 III Stratégies algorithmiques	13
 III.1 Algorithmes gloutons	13
 III.2 Diviser pour régner	13
 IV SQL	14
 IV.1 Généralités	14
 IV.2 Requêtes	14
 IV.3 Fonctions	15
 IV.4 Sous requêtes	16
 IV.5 Combiner les tables	17
 IV.6 Créer une BDD	18
 IV.7 Type entités	19
Liste d'algorithmes	20
Table des matières	21