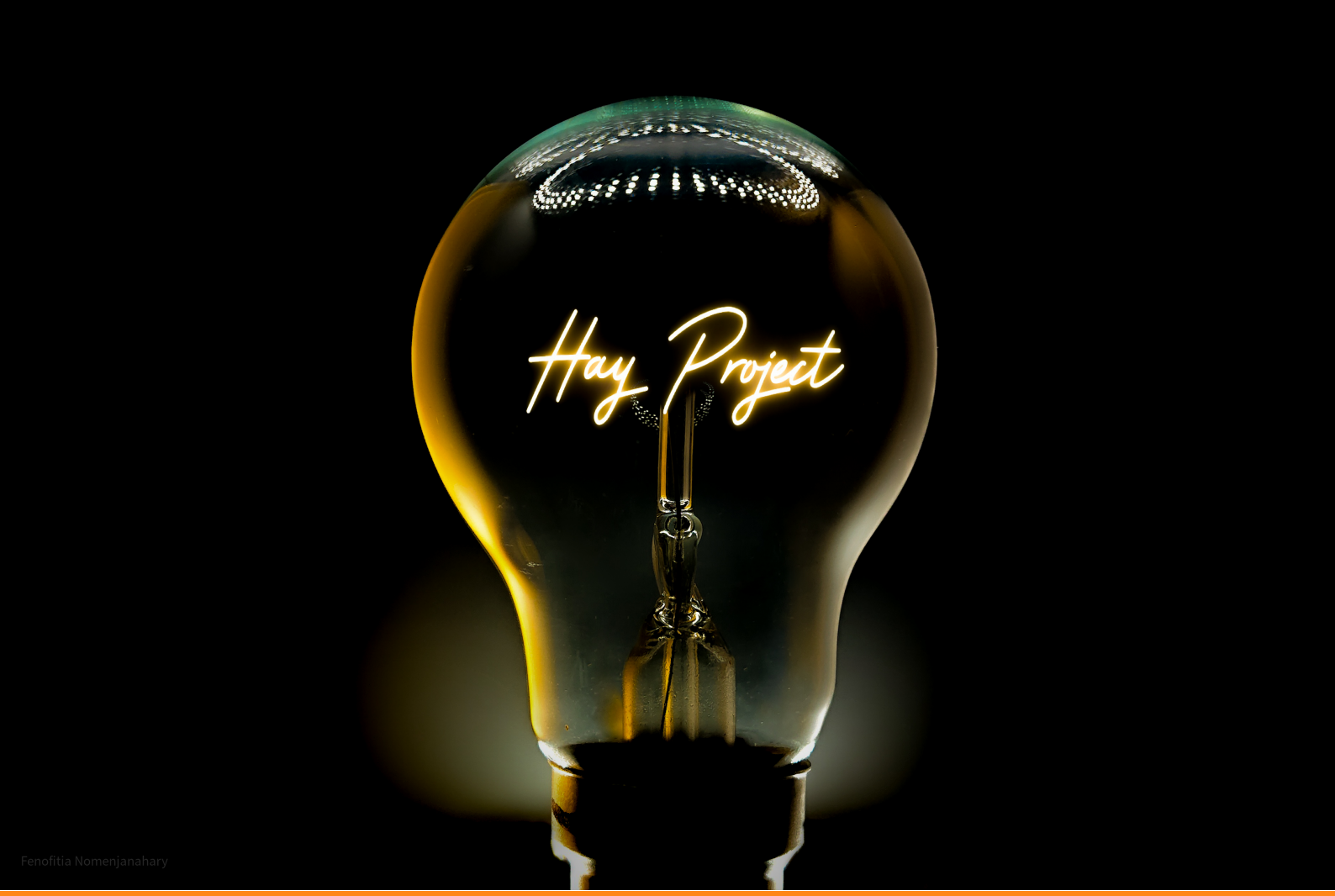

Kajy University : Informatique

Algorithmes et structures de données

Author : Dimby Rabearivony

Date : 7 décembre 2024

Version : 1.1

"Ny fahalalana no valamparihiko."

Table des matières

Introduction

“En fait, je dirai que la différence entre un mauvais programmeur et un bon réside dans le fait qu’il considère son code ou ses structures de données comme plus importants. Les mauvais programmeurs se soucient du code. Les bons programmeurs se soucient des structures de données et de leurs relations.”

- Linus Torvalds

————————-o0o————————-

Bienvenue dans ce cours sur les structures de données et les algorithmes à Kajy University, Antananarivo, Madagascar. Ce cours est conçu pour les étudiants de première année qui souhaitent acquérir une solide compréhension des concepts fondamentaux en informatique. Les structures de données et les algorithmes constituent le fondement de la programmation et du développement logiciel, essentiels pour toute carrière en informatique ou en ingénierie logicielle.

L’objectif principal de ce cours est de vous familiariser avec les différentes structures de données, telles que les tableaux, les listes chaînées, les piles, les files, les arbres binaires, et les graphes. Vous apprendrez également les algorithmes associés à ces structures, tels que les algorithmes de tri, de recherche, et de parcours. Le cours abordera également la complexité algorithmique, un élément crucial pour évaluer l’efficacité des algorithmes.

Les structures de données et les algorithmes sont au cœur de nombreux aspects de l’informatique. Ils permettent de stocker, organiser, et manipuler des données de manière efficace. Comprendre ces concepts vous aidera à résoudre des problèmes complexes, à développer des programmes performants, et à améliorer vos compétences en programmation.

Voici ce que vous apprendrez au cours de ce semestre :

- **Structures de données fondamentales** : Vous apprendrez les concepts de base comme les variables, les pointeurs, les types de données, les tableaux, et les chaînes de caractères. Cela servira de fondation pour comprendre les structures de données avancées et les algorithmes.
- **Algorithmes de recherche et de tri** : Cette partie couvre les algorithmes courants de tri et de recherche, y compris le tri à bulles, le tri par insertion, le tri rapide, la recherche linéaire, et la recherche binaire. Ces algorithmes sont essentiels pour manipuler et organiser des données.
- **Complexité algorithmique** : Vous découvrirez les concepts de complexité algorithmique, y compris la notation O, Ω et Θ. Vous étudierez également la récursion et comprendrez comment elle affecte la complexité algorithmique.
- **Structures de données avancées** : Cette section traite des structures de données avancées comme les listes chaînées, les piles, les files, les arbres binaires, et les graphes. Vous apprendrez également des algorithmes de parcours et leurs applications.

- **Applications d'algorithmes en IA** : Cette dernière partie explore les applications pratiques des structures de données et des algorithmes dans le domaine de l'intelligence artificielle. Vous étudierez des algorithmes comme la régression linéaire, la régression logistique, et le clustering K-means. Ces algorithmes constituent une base fondamentale pour comprendre comment l'intelligence artificielle utilise les algorithmes pour résoudre des problèmes complexes.

Ce cours adopte une approche pratique. Vous aurez des sessions théoriques pour comprendre les concepts de base, suivies de travaux pratiques pour appliquer ces concepts dans des exercices et des projets. Vous travaillerez avec le langage C pour mettre en œuvre des structures de données et des algorithmes, et développer des compétences en programmation.

À la fin du cours, vous aurez une compréhension solide des structures de données et des algorithmes, et vous serez en mesure de les appliquer dans des contextes réels. Ces compétences vous seront utiles tout au long de vos études et de votre carrière professionnelle.

Nous sommes impatients de vous accompagner dans ce voyage d'apprentissage et de vous voir progresser dans le domaine passionnant des structures de données et des algorithmes. Bon cours !

Chapitre 6

Corrigées des exercices

6.1 Structures de données fondamentales

Variables et types de données

1. Quelle est la différence entre les types de données suivants :
   (a). **int**
   (b). **float**
   (c). **char**
2. Écrivez un programme C qui déclare des variables pour chaque type de données mentionné et affiche leurs valeurs.

   Réponse :

```c
#include <stdio.h>

int main() {
        int entier = 42;
        float flottant = 3.14;
        char caractere = 'A';

        printf("Entier: %d\n", entier);
        printf("Flottant: %.2f\n", flottant);
        printf("Caractere: %c\n", caractere);

        return 0;
    }
```

3. Écrivez un programme C qui utilise des variables de différents types de données pour calculer la somme d'un entier et d'un flottant, puis affichez le résultat.

   Réponse :

```c
#include <stdio.h>

int main() {
        int entier = 10;
```

```
5           float flottant = 2.5;
6
7           float somme = entier + flottant;
8
9           printf("La somme est: %.2f\n", somme);
10
11          return 0;
12      }
```

4. Quelles sont les limites de valeurs pour les types de données suivants :
   (a). **int**
   (b). **float**
   (c). **double**

   Indication : Considérer le nombre de bits et octets. Par exemple, un caractère ASCII est représenté par un octet, c'est-à-dire 8 bits. Le caractère "A" est représenté par 01000001. Avec huit chiffres de 0 et 1, on peut representer $2^8 = 256$ valeurs. Pour les nombres signés (int, float, double, ...), l'opposé est representé différemment.

   Réponse :
   (a). 'int' : généralement de '-2.147.483.648' à '2.147.483.647'.
   (b). 'float' : de '1,2E-38' à '3,4E+38'.
   (c). 'double' : de '2,3E-308' à '1,7E+308'.

Pointeurs

1. Quelle est la différence entre une variable et un pointeur ?
2. Écrivez un programme C qui utilise des pointeurs pour modifier la valeur d'une variable.

   Réponse :

```
1   #include <stdio.h>
2
3   int main() {
4           int x = 10;
5           int *p = &x;
6
7           printf("Valeur de x avant modification: %d\n", x);
8
9           *p = 20;
10
11          printf("Valeur de x apres modification: %d\n", x);
12
```

```
13          return 0;
14      }
```

3. Écrivez un programme C qui utilise des pointeurs pour échanger les valeurs de deux variables.

   Réponse :

```
1   #include <stdio.h>
2
3   void echanger(int *a, int *b) {
4           int temp = *a;
5           *a = *b;
6           *b = temp;
7       }
8
9   int main() {
10          int x = 10;
11          int y = 20;
12
13          printf("Avant l'echange: x = %d, y = %d\n", x, y);
14
15          echanger(&x, &y);
16
17          printf("Apres l'echange: x = %d, y = %d\n", x, y);
18
19          return 0;
20      }
```

4. Expliquez ce qu'est un pointeur nul (null pointer) et ce qui se passe si vous tentez d'utiliser un pointeur nul.

   Réponse : Un pointeur nul ('NULL') ne pointe pas vers une adresse mémoire valide. Tenter d'utiliser ou de déréférencer un pointeur nul peut provoquer des erreurs de segmentation ou des plantages du programme.

Tableaux

1. Déclarez un tableau de 5 entiers et initialisez-le avec des valeurs. Écrivez un programme C pour afficher les valeurs du tableau.
2. Modifiez le programme pour inverser les éléments du tableau et afficher le tableau inversé.

   Réponse :

```c
#include <stdio.h>

int main() {
    int tableau[5] = {1, 2, 3, 4, 5};

    printf("Tableau initial : ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", tableau[i]);
    }
    printf("\n");

    // Inverser le tableau
    int temp;
    for (int i = 0; i < 2; i++) {
        temp = tableau[i];
        tableau[i] = tableau[4 - i];
        tableau[4 - i] = temp;
    }

    printf("Tableau inversE: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", tableau[i]);
    }
    printf("\n");

    return 0;
}
```

3. Écrivez un programme C qui calcule la somme de tous les éléments d'un tableau d'entiers de longueur 'n'.

Réponse :

```c
#include <stdio.h>

int somme(int tableau[], int taille ) {
    int total = 0;

    for (int i = 0; i < taille ; i++) {
        total += tableau[i];
```

```
8               }
9
10              return total;
11          }
12
13      int main() {
14              int tableau[5] = {1, 2, 3, 4, 5};
15
16              int total = somme(tableau, 5);
17
18              printf("La somme totale est: %d\n", total);
19
20              return 0;
21          }
```

4. Que se passe-t-il si vous essayez d'accéder à un index hors limites (out of bounds) dans un tableau C ? Écrivez un programme pour le démontrer.

   Réponse : Accéder à un index hors limites (out of bounds) peut provoquer des erreurs de segmentation ou des plantages du programme.

```
1       #include <stdio.h>
2
3       int main() {
4               int tableau[5] = {1, 2, 3, 4, 5};
5
6               // Essayer d'acceder a un index hors limites
7               printf("AccEs a un index hors limites: %d\n", tableau
                    [10]);
8               // Ceci peut provoquer des erreurs de segmentation
9
10              return 0;
11          }
```

Chaînes de caractères

1. Quelle est la particularité des chaînes de caractères en C par rapport aux autres types de données ?
2. Écrivez un programme C qui déclare une chaîne de caractères et utilise la bibliothèque 'string.h' pour copier du texte dans la chaîne.

   Réponse :

```c
#include <stdio.h>
#include <string.h>

int main() {
    char chaine[50];

    strcpy(chaine, "Bonjour, monde!");

    printf("Chaîne: %s\n", chaine);

    return 0;
}
```

3. Écrivez un programme C qui concatène deux chaînes de caractères en utilisant ‘strcat‘.

   Réponse :

```c
#include <stdio.h>
#include <string.h>

int main() {
    char chaine1[50] = "Bonjour";
    char chaine2[50] = ", monde!";

    strcat(chaine1, chaine2); // Concaténer les deux chaînes

    printf("Résultat de la concaténation: %s\n", chaine1);

    return 0;
}
```

4. Comment compare-t-on deux chaînes de caractères en C ? Écrivez un programme qui compare deux chaînes et indique si elles sont identiques ou non.

   Réponse : On peut comparer deux chaînes de caractères en utilisant ‘strcmp‘ de la bibliothèque ‘string.h‘.

```c
#include <stdio.h>
#include <string.h>

int main() {
    char chaine1[50] = "Bonjour";
```

```
6           char chaine2[50] = "Bonjour";
7
8           int comparaison = strcmp(chaine1, chaine2);
9
10          if (comparaison == 0) {
11              printf("Les deux chaînes sont identiques.\n");
12          } else if (comparaison < 0) {
13              printf("La première chaîne est inférieure.\n");
14          } else {
15              printf("La première chaîne est supérieure.\n");
16          }
17
18          return 0;
19      }
```

Structures

1. Déclarez une structure en C pour représenter une personne avec un nom, un âge, et une taille.
2. Écrivez un programme C qui utilise cette structure pour créer une personne, assigner des valeurs, et afficher les informations de la personne.

   Réponse :

```
1   #include <stdio.h>
2   #include <string.h>
3
4   typedef struct {
5       char nom[50];
6       int age;
7       float taille ;
8   } Personne;
9
10  int main() {
11      Personne p;
12      strcpy(p.nom, "John Doe");
13      p.age = 30;
14      p. taille = 1.75;
15
16      printf("Nom: %s\n", p.nom);
```

```
17          printf("Âge: %d\n", p.age);
18          printf("Taille: %.2f m\n", p.taille);
19
20          return 0;
21      }
```

3. Écrivez un programme C qui utilise des structures pour représenter un point 3D avec des coordonnées x, y, z. Ensuite, écrivez une fonction qui calcule la distance entre deux points 3D.

Réponse :

```
1   #include <stdio.h>
2   #include <math.h>
3
4   typedef struct {
5       float x;
6       float y;
7       float z;
8   } Point3D;
9
10  float distance(Point3D p1, Point3D p2) {
11      return sqrt(pow(p2.x - p1.x, 2) + pow(p2.y - p1.y, 2) +
            pow(p2.z - p1.z, 2));
12  }
13
14  int main() {
15      Point3D p1 = {0, 0, 0};
16      Point3D p2 = {3, 4, 5};
17
18      float dist = distance(p1, p2);
19
20      printf("Distance entre les deux points: %.2f\n", dist);
21
22      return 0;
23  }
```

4. Créez une structure pour représenter une voiture avec un nom, une année, et un prix. Écrivez un programme C qui utilise cette structure pour créer une voiture et afficher ses propriétés.

Réponse :

```c
#include <stdio.h>
#include <string.h>

typedef struct {
        char nom[50];
        int annee;
        float prix;
  } Voiture;

int main() {
        Voiture v;
        strcpy(v.nom, "Toyota");
        v.annee = 2020;
        v.prix = 25000.50;

        printf("Voiture: %s\n", v.nom);
        printf("Année: %d\n", v.annee);
        printf("Prix: %.2f\n", v.prix);

        return 0;
  }
```

6.2 Algorithmes de tri et de recherche

Algorithmes de tri

1. Pour chaque algorithme de tri, implémentez-le en C et testez-le avec les cas suivants :
   (a). Un tableau vide.
   (b). Un tableau contenant un seul élément.
   (c). Un tableau déjà trié.
   (d). Le tableau $[9, 1, 9, 0, 2, 6, 6, 5, 3, 4, 7]$.

Tri à bulle (Bubble Sort)

Réponse :

```c
#include <stdio.h>

void bubbleSort(int arr [], int n) {
```

```
    for (int i = 0; i < n-1; i++) {
      for (int j = 0; j < n-i-1; j++) {
        if (arr[j] > arr[j+1]) {
          // Échanger les éléments
          int temp = arr[j];
          arr[j] = arr[j+1];
          arr[j+1] = temp;
        }
      }
    }
}

void printArray(int arr [], int size) {
    for (int i = 0; i < size; i++) {
      printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr [] = {9, 1, 9, 0, 2, 6, 6, 5, 3, 4, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    bubbleSort(arr, n);
    printf("Array trié : ");
    printArray(arr, n);

    return 0;
}
```

Tri par insertion (Insertion Sort)

Réponse :

```
#include <stdio.h>

void insertionSort(int arr [], int n) {
    for (int i = 1; i < n; i++) {
      int key = arr[i];
```

```
6         int j = i - 1;
7         while (j >= 0 && arr[j] > key) {
8           arr[j + 1] = arr[j];
9           j = j - 1;
10        }
11        arr[j + 1] = key;
12      }
13    }
14
15    void printArray(int arr[], int size) {
16      for (int i = 0; i < size; i++) {
17        printf("%d ", arr[i]);
18      }
19      printf("\n");
20    }
21
22    int main() {
23      int arr[] = {9, 1, 9, 0, 2, 6, 6, 5, 3, 4, 7};
24      int n = sizeof(arr)/sizeof(arr[0]);
25
26      insertionSort(arr, n);
27      printf("Array trié : ");
28      printArray(arr, n);
29
30      return 0;
31    }
```

Tri rapide (Quick Sort)

Réponse :

```
1   #include <stdio.h>
2
3   int partition(int arr[], int low, int high) {
4     int pivot = arr[high];
5     int i = (low - 1);
6
7     for (int j = low; j < high; j++) {
8       if (arr[j] < pivot) {
```

```
9             i++;
10            int temp = arr[i];
11            arr[i] = arr[j];
12            arr[j] = temp;
13          }
14        }
15        int temp = arr[i + 1];
16        arr[i + 1] = arr[high];
17        arr[high] = temp;
18        return i + 1;
19      }
20
21      void quickSort(int arr[], int low, int high) {
22        if (low < high) {
23          int pi = partition(arr, low, high);
24          quickSort(arr, low, pi - 1);
25          quickSort(arr, pi + 1, high);
26        }
27      }
28
29      void printArray(int arr[], int size) {
30        for (int i = 0; i < size; i++) {
31          printf("%d ", arr[i]);
32        }
33        printf("\n");
34      }
35
36      int main() {
37        int arr[] = {9, 1, 9, 0, 2, 6, 6, 5, 3, 4, 7};
38        int n = sizeof(arr)/sizeof(arr[0]);
39
40        quickSort(arr, 0, n - 1);
41        printf("Array trié : ");
42        printArray(arr, n);
43
44        return 0;
45      }
```

Algorithmes de recherche

1. Pour chaque algorithme de recherche, implémentez-le en C et testez-le en cherchant les éléments 8, 1 et 0 dans le tableau $[9, 1, 9, 0, 2, 6, 6, 5, 3, 4, 7]$.

Recherche linéaire (Linear Search)

Réponse :

```c
#include <stdio.h>

int linearSearch(int arr[], int n, int key) {
  for (int i = 0; i < n; i++) {
    if (arr[i] == key) {
      return i; // Retourne l'indice où l'élément est trouvé
    }
  }
  return -1; // L'élément n'est pas trouvé
}

int main() {
  int arr[] = {9, 1, 9, 0, 2, 6, 6, 5, 3, 4, 7};
  int n = sizeof(arr)/sizeof(arr[0]);

  printf("Recherche de l'élément 8: %d\n", linearSearch(arr, n, 8));
  printf("Recherche de l'élément 1: %d\n", linearSearch(arr, n, 1));
  printf("Recherche de l'élément 0: %d\n", linearSearch(arr, n, 0));

  return 0;
}
```

Recherche binaire (Binary Search)

La recherche binaire n'est applicable que si le tableau est trié. Voici l'implémentation pour une recherche binaire dans un tableau trié.

Réponse :

```c
#include <stdio.h>
```

```
2
3   int binarySearch(int arr [], int low, int high, int key) {
4     while (low <= high) {
5       int mid = low + (high - low) / 2;
6       if (arr[mid] == key) {
7         return mid;
8       }
9       if (arr[mid] < key) {
10        low = mid + 1;
11      } else {
12        high = mid - 1;
13      }
14    }
15    return -1; // L'élément n'est pas trouvé
16  }
17
18  int main() {
19    int arr [] = {0, 1, 4, 4, 4, 5, 6, 7, 9, 9};
20    int n = sizeof(arr)/sizeof(arr[0]) ;
21
22    printf("Recherche de l'élément 0: %d\n", binarySearch(arr, 0,
          n - 1, 0));
23    printf("Recherche de l'élément 2: %d\n", binarySearch(arr, 0,
          n - 1, 2));
24    printf("Recherche de l'élément 7: %d\n", binarySearch(arr, 0,
          n - 1, 7));
25    printf("Recherche de l'élément 9: %d\n", binarySearch(arr, 0,
          n - 1, 9));
26
27    return 0;
28  }
```

2. Lequel de ces algorithmes n'est pas applicable au tableau précédent ? Pourquoi ?

   La recherche binaire n'est pas applicable au tableau $[9, 1, 9, 0, 2, 6, 6, 5, 3, 4, 7]$ car il n'est pas trié. La recherche binaire fonctionne uniquement sur des tableaux triés, car elle divise le tableau en deux parties et suppose que les éléments à gauche sont inférieurs à ceux à droite. Dans ce cas, un tri préalable est nécessaire.

6.3 Complexité algorithmique

1. Quelle est la complexité temporelle des opérations suivantes ?

   (a). Parcourir un tableau d'entiers de longueur n.

   Réponse : La complexité temporelle de l'opération "parcourir un tableau d'entiers" est $O(n)$. En effet, dans cette opération, chaque élément du tableau est visité une fois, ce qui signifie qu'il y a n opérations à effectuer, où n est la taille du tableau. La notation $O(n)$ exprime que la durée de l'opération est linéaire par rapport à la taille du tableau.

   (b). Multiplier deux matrices de taille $n \times n$.

   Réponse : La multiplication de deux matrices carrées de taille $n \times n$ nécessite d'effectuer n^2 multiplications. En effet, chaque élément de la matrice résultat est le produit de n éléments (lignes de la première matrice et colonnes de la deuxième matrice). Donc, pour chaque élément de la matrice résultat (qui est n^2 éléments au total), nous devons effectuer n multiplications, ce qui donne une complexité de $O(n^3)$. Cette complexité est dans le cas général pour une multiplication de matrices à l'aide de l'algorithme classique. Des algorithmes plus avancés, comme l'algorithme de Strassen, peuvent réduire cette complexité à $O(n^{2.81})$, mais l'algorithme classique reste de complexité $O(n^3)$.

   (c). Effectuer une recherche binaire sur un tableau trié.

   Réponse : La recherche binaire dans un tableau trié a une complexité temporelle de $O(\log n)$. L'idée principale de la recherche binaire est de diviser à chaque étape le tableau en deux sous-tableaux, réduisant ainsi de moitié l'espace de recherche à chaque itération. Cette opération est répétée jusqu'à ce que l'élément recherché soit trouvé ou que l'espace de recherche soit vide. La complexité $\log n$ provient du fait que l'on réduit par moitié la taille de la partie restante du tableau à chaque itération. La recherche binaire est donc très efficace pour les tableaux triés, car elle évite de parcourir l'intégralité du tableau. Contrairement à une recherche linéaire qui aurait une complexité $O(n)$, la recherche binaire est logarithmique.

2. Quelle est la complexité temporelle du code suivant ?

```
1    for (int i = 0; i < n; i++) {
2      for (int j = 0; j < n; j++) {
3        printf("%d ", i * j);
4      }
5    }
```

Réponse : La complexité temporelle de ce code est $O(n^2)$. Voici pourquoi :

- La première boucle for $(\text{int i} = 0; \text{ i} < \text{n}; \text{ i++})$ s'exécute n fois, car i varie de 0 à $n-1$.

- La deuxième boucle imbriquée for (int j = 0; j < n; j++) s'exécute également n fois pour chaque itération de la première boucle.

Par conséquent, pour chaque itération de la première boucle (qui se répète n fois), la deuxième boucle s'exécute également n fois, ce qui donne un total de $n \times n = n^2$ itérations au total. C'est ce que l'on appelle une complexité quadratique.

- Le printf à l'intérieur des boucles effectue une opération constante à chaque itération (une simple impression), ce qui ne change pas la complexité globale de l'algorithme. L'opération printf elle-même, bien que coûteuse en termes de temps d'exécution réel, reste considérée comme une opération de complexité constante dans ce contexte.

3. Comment pouvez-vous améliorer cet algorithme pour réduire sa complexité ?

Réponse : Pour réduire la complexité de cet algorithme, il faudrait d'abord identifier s'il y a une optimisation possible dans la logique de l'algorithme. Ici, le problème semble être celui d'afficher tous les produits de $i \times j$ pour i, j allant de 0 à $n - 1$. Une possible amélioration serait de revoir la structure du code pour éviter des calculs redondants. En effet, si on cherche à afficher les produits $i \times j$ et que l'on parcourt les deux indices i et j indépendamment, cela entraîne une complexité $O(n^2)$. Cependant, il est possible de réorganiser le calcul en utilisant des propriétés symétriques des produits, comme dans le cas d'un tableau triangulaire ou d'autres structures spécifiques.

En fonction de l'objectif spécifique du programme (comme stocker les résultats ou effectuer un calcul particulier), il pourrait être possible de réduire la quantité de travail en évitant de calculer des produits pour des paires (i, j) identiques (par exemple, en exploitant la symétrie si $i \times j = j \times i$), mais cela dépend du contexte de l'application.

6.4 Structures de données avancées

Listes chaînées

1. **Création d'une liste chaînée** :

Réponse : Pour créer une liste chaînée, nous devons définir une structure pour les nœuds de la liste, où chaque nœud contiendra une donnée et un pointeur vers le nœud suivant. La tête de la liste est initialisée à NULL pour indiquer que la liste est vide.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct Node {
5    int data;
6    struct Node* next;
7  };
8
```

```
9   struct Node* head = NULL; // Initialisation de la tête de la
        liste à NULL
10
11  // Fonction pour créer une nouvelle liste
12  void createList(int value) {
13    struct Node* newNode = malloc(sizeof(struct Node));
14    newNode->data = value;
15    newNode->next = NULL;
16    head = newNode;
17  }
```

Explication : La fonction createList alloue de la mémoire pour un nouveau nœud, définit sa donnée, et initialise son pointeur suivant à NULL. Elle affecte ensuite ce nœud à la tête de la liste.

2. **Insertion d'un nœud au début de la liste** :

Réponse : Pour insérer un nœud au début de la liste chaînée, nous créons un nouveau nœud et nous mettons à jour son pointeur next pour qu'il pointe vers l'ancienne tête de la liste. Ensuite, nous mettons la tête de la liste pour qu'elle pointe vers ce nouveau nœud.

```
1   // Fonction pour insérer un œnud au début de la liste
2   void insertAtBeginning(int value) {
3     struct Node* newNode = malloc(sizeof(struct Node));
4     newNode->data = value;
5     newNode->next = head; // Le nouveau œnud pointe vers l'
          ancienne tête
6     head = newNode; // La tête pointe maintenant vers le
          nouveau œnud
7   }
```

Explication : La fonction insertAtBeginning crée un nouveau nœud avec la valeur donnée, fait pointer ce nœud vers l'ancienne tête de la liste, puis met la tête de la liste pour qu'elle pointe vers le nouveau nœud. Cela insère efficacement le nœud au début de la liste.

3. **Insertion d'un nœud à une position spécifique** :

Réponse : Pour insérer un nœud à une position spécifique dans la liste, nous devons parcourir la liste jusqu'à la position souhaitée, puis insérer le nœud à cet endroit en ajustant les pointeurs. Si la position est zéro, on insère au début de la liste.

```
1   // Fonction pour insérer un œnud à une position spécifique
2   void insertAtPosition(int value, int position) {
3     if (position == 0) {
4       insertAtBeginning(value);
```

```
5         return;
6       }
7
8       struct Node* newNode = malloc(sizeof(struct Node));
9       newNode->data = value;
10      struct Node* current = head;
11
12      for (int i = 0; i < position - 1 && current != NULL; i++)
            {
13        current = current->next;
14      }
15
16      if (current == NULL) {
17        printf("Position invalide\n");
18        return;
19      }
20
21      newNode->next = current->next;
22      current->next = newNode;
23    }
```

Explication : La fonction insertAtPosition parcourt la liste jusqu'à la position spécifiée. Si la position est zéro, elle appelle la fonction insertAtBeginning. Sinon, elle insère un nouveau nœud à la position donnée en ajustant les pointeurs. Si la position est invalide, un message d'erreur est affiché.

4. **Suppression d'un nœud de la liste chaînée** :

Réponse : Pour supprimer un nœud de la liste chaînée, nous devons parcourir la liste jusqu'à ce que nous trouvions le nœud à supprimer. Ensuite, nous ajustons le pointeur du nœud précédent pour qu'il pointe vers le nœud suivant du nœud à supprimer.

```
1   // Fonction pour supprimer un œnud d'une liste chaînée
2   void deleteNode(int value) {
3     struct Node* current = head;
4     struct Node* previous = NULL;
5
6     // Si la tête contient la valeur à supprimer
7     if (current != NULL && current->data == value) {
8       head = current->next;
9       free(current);
```

```
10          return;
11        }
12
13        // Parcours de la liste à la recherche du œnud
14        while (current != NULL && current->data != value) {
15          previous = current;
16          current = current->next;
17        }
18
19        // Si la valeur n'est pas trouvée
20        if (current == NULL) {
21          printf("Valeur non trouvée dans la liste\n");
22          return;
23        }
24
25        // Supprimer le œnud
26        previous->next = current->next;
27        free(current);
28      }
```

Explication : La fonction deleteNode commence par vérifier si le nœud à supprimer est le premier nœud de la liste (la tête). Si ce n'est pas le cas, elle parcourt la liste pour trouver le nœud correspondant à la valeur à supprimer. Une fois le nœud trouvé, elle ajustera le pointeur du nœud précédent pour qu'il saute le nœud à supprimer et libère la mémoire allouée pour ce nœud.

Piles

1. **Construction d'une pile** :

   Réponse : Pour construire une pile, vous devez définir une structure qui contient les éléments de la pile et un pointeur ou un indice qui indique le sommet de la pile. La structure suivante utilise un tableau pour représenter la pile et un indice pour suivre le sommet de la pile.

```
1     #include <stdio.h>
2     #include <stdlib.h>
3
4     #define MAX 10
5
6     struct Stack {
```

```
7       int items[MAX];
8       int top;
9     };
10
11    // Fonction pour initialiser la pile
12    void initStack(struct Stack* stack) {
13      stack->top = -1; // La pile est vide au départ
14    }
15
16    // Fonction pour vérifier si la pile est vide
17    int isEmpty(struct Stack* stack) {
18      return stack->top == -1;
19    }
```

Explication : La fonction initStack initialise la pile en définissant le sommet à -1, ce qui indique que la pile est vide. La fonction isEmpty permet de vérifier si la pile est vide en vérifiant si l'indice du sommet est égal à -1.

2. **Insertion d'un élément dans une pile (empilage)** :

Réponse : L'empilage consiste à ajouter un élément au sommet de la pile. Voici la fonction pour empiler un élément dans la pile, en vérifiant d'abord si la pile n'est pas pleine.

```
1     // Fonction pour empiler un élément
2     void push(struct Stack* stack, int value) {
3       if (stack->top == MAX - 1) {
4         printf("Erreur : pile pleine\n");
5         return;
6       }
7       stack->items[++stack->top] = value; // Incrémente le
            sommet puis ajoute l'élément
8     }
```

Explication : La fonction push ajoute un élément au sommet de la pile. Si le sommet est égal à MAX - 1, cela signifie que la pile est pleine et une erreur est renvoyée. Sinon, l'élément est ajouté au sommet, et l'indice du sommet est incrémenté.

3. **Suppression d'un élément dans une pile (dépilage)** :

Réponse : Le dépilage consiste à retirer l'élément du sommet de la pile. Si la pile est vide, une erreur est renvoyée. Voici la fonction pour dépiler un élément de la pile :

```
1     // Fonction pour dépiler un élément
2     int pop(struct Stack* stack) {
3       if (isEmpty(stack)) {
```

```
4         printf("Erreur : pile vide\n");
5         return -1;  // Valeur spéciale pour indiquer une erreur
6       }
7       return stack->items[stack->top--];  // Retourne l'élément au
            sommet puis décrémente le sommet
8     }
```

Explication : La fonction pop supprime l'élément au sommet de la pile. Si la pile est vide, elle renvoie une erreur. Sinon, elle retourne l'élément au sommet et décrémente l'indice du sommet pour retirer l'élément de la pile.

4. **Test de la pile** :

Réponse : Voici un exemple de code qui utilise les fonctions d'empilage et de dépilage sur une pile.

```
1   int main() {
2     struct Stack stack;
3     initStack(&stack);
4
5     push(&stack, 10);
6     push(&stack, 20);
7     push(&stack, 30);
8
9     printf("Dépiler : %d\n", pop(&stack)); // Devrait afficher
          30
10    printf("Dépiler : %d\n", pop(&stack)); // Devrait afficher
          20
11    printf("Dépiler : %d\n", pop(&stack)); // Devrait afficher
          10
12
13    return 0;
14  }
```

Explication : Dans cet exemple, nous empilons trois éléments (10, 20, 30) dans la pile et ensuite nous les dépilons un par un. Les éléments sont retirés dans l'ordre inverse de leur insertion, conformément au principe LIFO (Last In, First Out).

Files

1. **Construction d'une file** :

Réponse : Pour construire une file, vous devez définir une structure qui contient les éléments de la file, ainsi que des pointeurs pour l'avant et l'arrière. Ces pointeurs permettent

de gérer les opérations d'enfilage (ajout d'un élément à l'arrière) et de défilage (retrait du premier élément). Voici un exemple de structure de file en C :

```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
  int data;
  struct Node* next;
};

struct Queue {
  struct Node* front;
  struct Node* rear;
};

// Fonction pour initialiser la file
void initQueue(struct Queue* q) {
  q->front = NULL;
  q->rear = NULL;
}
```

Explication : La structure Queue contient deux pointeurs, front et rear, qui pointent respectivement vers le premier et le dernier élément de la file. La fonction initQueue initialise ces pointeurs à NULL pour indiquer que la file est vide.

2. **Insertion d'un élément dans une file (enfilage)** :

Réponse : L'enfilage consiste à ajouter un élément à la fin de la file. Si la file est vide, l'élément ajouté devient à la fois l'avant et l'arrière de la file. Si la file contient déjà des éléments, vous devez ajuster le pointeur rear pour indiquer le nouvel élément.

```c
// Fonction pour enfiler un élément
void enqueue(struct Queue* q, int value) {
  struct Node* newNode = malloc(sizeof(struct Node));
  newNode->data = value;
  newNode->next = NULL;

  if (q->rear == NULL) { // Si la file est vide
    q->front = newNode;
    q->rear = newNode;
  } else {
```

```
11          q->rear->next = newNode;
12          q->rear = newNode;
13        }
14    }
```

Explication : La fonction enqueue crée un nouveau nœud et le place à la fin de la file. Si la file est vide, l'élément ajouté devient à la fois le premier et le dernier élément. Sinon, le nouvel élément devient le dernier de la file, et rear est mis à jour pour pointer vers ce nouvel élément.

3. **Suppression d'un élément dans une file (défilage)** :

Réponse : Le défilage consiste à retirer le premier élément de la file. Si la file est vide, la fonction retourne une erreur ou une valeur spéciale. Sinon, elle met à jour le pointeur front pour qu'il pointe vers le prochain élément de la file et libère la mémoire du nœud retiré.

```
1   // Fonction pour défiler un élément
2   int dequeue(struct Queue* q) {
3     if (q->front == NULL) {
4       printf("La file est vide!\n");
5       return -1;  // Valeur d'erreur
6     }
7
8     struct Node* temp = q->front;
9     int value = temp->data;
10    q->front = q->front->next;
11
12    if (q->front == NULL) { // Si la file devient vide après le
          défilage
13      q->rear = NULL;
14    }
15
16    free(temp);
17    return value;
18  }
```

Explication : La fonction dequeue retire l'élément au début de la file en mettant à jour le pointeur front. Si la file devient vide après le défilage, le pointeur rear est également mis à NULL. La mémoire du nœud retiré est libérée, et la fonction retourne la valeur de l'élément retiré.

Arbres binaires

1. **Construction d'un arbre binaire** :

   Réponse : Pour construire un arbre binaire, vous devez définir une structure qui représente chaque nœud de l'arbre. Chaque nœud contient des données et deux pointeurs, un pour l'enfant gauche et un pour l'enfant droit. L'arbre commence avec un seul nœud racine, et les autres nœuds sont ajoutés en fonction de leur relation avec la racine. Voici un exemple de code en C pour définir la structure d'un arbre binaire :

```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
  int data;
  struct Node* left;
  struct Node* right;
};

// Fonction pour créer un nouveau œnud
struct Node* createNode(int data) {
  struct Node* newNode = malloc(sizeof(struct Node));
  newNode->data = data;
  newNode->left = NULL;
  newNode->right = NULL;
  return newNode;
}

// Fonction pour insérer un élément dans l'arbre binaire
void insert(struct Node** root, int data) {
  if (*root == NULL) {
    *root = createNode(data);
    return;
  }
  if (data < (*root)->data)
  insert(&(*root)->left, data);
  else
  insert(&(*root)->right, data);
}
```

   Explication : La structure Node contient un entier data et deux pointeurs left et right

qui pointent respectivement vers l'enfant gauche et l'enfant droit. La fonction createNode crée un nouveau nœud avec la valeur donnée, et la fonction insert insère un élément dans l'arbre en suivant la règle de l'arbre binaire de recherche (les éléments plus petits vont à gauche et les plus grands à droite).

2. **Insertion dans un arbre binaire** :

   Réponse : L'insertion dans un arbre binaire se fait en suivant la règle de comparaison avec la racine. Si le nouvel élément est plus petit que la racine, il va à gauche, sinon il va à droite. Le processus continue récursivement jusqu'à ce qu'un emplacement vide soit trouvé pour le nouvel élément.

   Voici l'exemple d'insertion dans un arbre binaire :

```
1   int main() {
2     struct Node* root = NULL;
3     insert(&root, 50);  // Insertion de 50 à la racine
4     insert(&root, 30);  // Insertion de 30 à gauche de 50
5     insert(&root, 70);  // Insertion de 70 à droite de 50
6     insert(&root, 20);  // Insertion de 20 à gauche de 30
7     insert(&root, 40);  // Insertion de 40 à droite de 30
8     insert(&root, 60);  // Insertion de 60 à gauche de 70
9     insert(&root, 80);  // Insertion de 80 à droite de 70
10    return 0;
11  }
```

   Explication : Dans cet exemple, l'arbre est construit à partir de la racine et chaque élément est inséré à la bonne position en fonction de sa valeur par rapport à la racine. Les éléments plus petits que la racine vont à gauche et ceux plus grands vont à droite. L'insertion est effectuée récursivement jusqu'à ce que l'emplacement correct soit trouvé.

3. **Algorithmes de parcours des arbres binaires** :

   Réponse : Les algorithmes de parcours des arbres binaires sont utilisés pour visiter tous les nœuds de l'arbre dans un ordre spécifique. Il existe trois principaux types de parcours d'arbres binaires :

   - **Parcours pré-ordre** : Visite la racine, puis le sous-arbre gauche et enfin le sous-arbre droit.
   - **Parcours en-ordre** : Visite le sous-arbre gauche, puis la racine, et enfin le sous-arbre droit. Ce type de parcours est utilisé dans les arbres binaires de recherche pour parcourir les éléments dans l'ordre croissant.
   - **Parcours post-ordre** : Visite le sous-arbre gauche, puis le sous-arbre droit, et enfin la racine.

   Exemple de code pour un parcours en-ordre en C :

```c
// Fonction de parcours en-ordre
void inorder(struct Node* root) {
  if (root == NULL) {
    return;
  }
  inorder(root->left); // Visiter le sous-arbre gauche
  printf("%d ", root->data); // Visiter la racine
  inorder(root->right); // Visiter le sous-arbre droit
}

// Fonction pour afficher l'arbre en-ordre
int main() {
  struct Node* root = NULL;
  insert(&root, 50);
  insert(&root, 30);
  insert(&root, 70);
  insert(&root, 20);
  insert(&root, 40);
  insert(&root, 60);
  insert(&root, 80);

  printf("Parcours en-ordre : ");
  inorder(root); // Afficher les éléments de l'arbre en-ordre
  return 0;
}
```

Explication : La fonction inorder parcourt l'arbre en-ordre en visitant d'abord le sous-arbre gauche, puis la racine, puis le sous-arbre droit. Le résultat du parcours en-ordre d'un arbre binaire de recherche est toujours les éléments dans l'ordre croissant.

Graphes

1. **Construction d'un graphe** :

   Réponse : Pour construire un graphe, il faut définir une structure de données qui contient les sommets (ou nœuds) ainsi que les arêtes qui les relient. On peut utiliser une structure avec des listes d'adjacence ou des matrices d'adjacence pour représenter le graphe. Dans ce cas, nous allons utiliser des listes d'adjacence, où chaque sommet a une liste de connexions vers les autres sommets.

   Voici un exemple de code C pour définir un graphe avec des listes d'adjacence :

```c
#include <stdio.h>
#include <stdlib.h>

// Structure représentant un œnud
struct Node {
  int vertex;
  struct Node* next;
};

// Structure représentant un graphe
struct Graph {
  int numVertices;
  struct Node** adjLists;
};

// Fonction pour créer un œnud
struct Node* createNode(int vertex) {
  struct Node* newNode = malloc(sizeof(struct Node));
  newNode->vertex = vertex;
  newNode->next = NULL;
  return newNode;
}

// Fonction pour créer un graphe avec un nombre de sommets donné
struct Graph* createGraph(int vertices) {
  struct Graph* graph = malloc(sizeof(struct Graph));
  graph->numVertices = vertices;
  graph->adjLists = malloc(vertices * sizeof(struct Node*));

  for (int i = 0; i < vertices; i++) {
    graph->adjLists[i] = NULL; // Initialisation de chaque liste d'adjacence
  }
  return graph;
}
```

Explication : Dans ce code, nous avons une structure Node qui représente un nœud avec

un sommet et un pointeur vers le nœud suivant (pour former une liste chaînée). La structure Graph contient un tableau de pointeurs vers les listes d'adjacence. Chaque élément de ce tableau représente un sommet du graphe, et chaque liste contient les sommets connectés à celui-ci.

2. **Ajout d'arêtes dans un graphe** :

   Réponse : Pour ajouter des arêtes dans un graphe, on insère un nouveau nœud dans la liste d'adjacence du sommet de départ (source) et dans la liste d'adjacence du sommet de destination. Si le graphe est orienté, l'arête va seulement du sommet source vers le sommet destination. Si le graphe est non orienté, l'arête est bidirectionnelle, donc on doit ajouter des connexions dans les deux directions.

   Voici un exemple de fonction en C pour ajouter une arête dans un graphe non orienté :

```
1  // Fonction pour ajouter une arête entre deux sommets dans un
       graphe non orienté
2  void addEdge(struct Graph* graph, int src, int dest) {
3    // Ajouter une arête de src à dest
4    struct Node* newNode = createNode(dest);
5    newNode->next = graph->adjLists[src];
6    graph->adjLists[src] = newNode;
7
8    // Ajouter une arête de dest à src (pour un graphe non orient
         é)
9    newNode = createNode(src);
10   newNode->next = graph->adjLists[dest];
11   graph->adjLists[dest] = newNode;
12 }
```

   Explication : La fonction addEdge crée un nouveau nœud pour chaque arête et l'ajoute à la liste d'adjacence correspondante. Dans le cas d'un graphe non orienté, il y a deux ajouts : un de src à dest et un de dest à src, ce qui rend l'arête bidirectionnelle.

3. **Recherche en largeur (Complétez le code)** :

   Réponse : Pour implémenter la recherche en largeur (BFS), on utilise une file (FIFO) pour explorer chaque niveau du graphe. On commence à partir du sommet source et on explore tous ses voisins, puis les voisins de ces voisins, et ainsi de suite.

   Voici un exemple de code pour implémenter la recherche en largeur (BFS) :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4
```

```
5   // Structure représentant une file
6   struct Queue {
7     int* items;
8     int front, rear, size;
9   };
10
11  // Fonction pour initialiser la file
12  void initQueue(struct Queue* q, int size) {
13    q->items = malloc(size * sizeof(int));
14    q->front = 0;
15    q->rear = 0;
16    q->size = size;
17  }
18
19  // Fonction pour ajouter un élément à la file
20  void enqueue(struct Queue* q, int item) {
21    q->items[q->rear++] = item;
22  }
23
24  // Fonction pour retirer un élément de la file
25  int dequeue(struct Queue* q) {
26    return q->items[q->front++];
27  }
28
29  // Fonction pour vérifier si la file est vide
30  bool isEmpty(struct Queue* q) {
31    return q->front == q->rear;
32  }
33
34  // Fonction pour effectuer une recherche en largeur (BFS)
35  void bfs(struct Graph* graph, int start) {
36    bool* visited = malloc(graph->numVertices * sizeof(bool));
37    for (int i = 0; i < graph->numVertices; i++) {
38      visited[i] = false;
39    }
40
41    struct Queue q;
42    initQueue(&q, graph->numVertices);
```

```
43
44          visited[start] = true;
45          enqueue(&q, start);
46
47          while (!isEmpty(&q)) {
48              int current = dequeue(&q);
49              printf("%d ", current);
50
51              struct Node* temp = graph->adjLists[current];
52              while (temp) {
53                  if (!visited[temp->vertex]) {
54                      visited[temp->vertex] = true;
55                      enqueue(&q, temp->vertex);
56                  }
57                  temp = temp->next;
58              }
59          }
60      }
```

Explication : La fonction bfs commence par marquer le sommet source comme visité et l'ajoute à la file. Ensuite, elle explore tous les voisins du sommet actuel, en ajoutant les non visités à la file. Cela continue jusqu'à ce que tous les sommets accessibles aient été visités.

6.5 Applications d'algorithmes en IA

Régression Linéaire

Données

Les données d'entrée sont les suivantes :

Table 6.1 – Données pour la régression linéaire

Surface (m²)	Prix (MGA)
50	70,000,000
75	95,000,000
100	130,000,000

Nous allons utiliser ces données pour calculer les coefficients a et b de la régression linéaire, puis prédire y pour $x = 80$.

Implémentation en C

Voici l'implémentation en C pour résoudre l'exercice :

Listing 6.1 – Implémentation de la Régression Linéaire en C

```c
#include <stdio.h>

// Fonction pour calculer les coefficients a et b
void linearRegression(int x[], int y[], int n, float *a, float *b) {
    int sumX = 0, sumY = 0, sumXY = 0, sumX2 = 0;

    // Calcul des sommes nécessaires
    for (int i = 0; i < n; i++) {
        sumX += x[i];
        sumY += y[i];
        sumXY += x[i] * y[i];
        sumX2 += x[i] * x[i];
    }

    // Calcul des coefficients
    *a = (n * sumXY - sumX * sumY) / (float)(n * sumX2 - sumX *
        sumX);
    *b = (sumY - (*a) * sumX) / (float)n;
}

// Fonction pour prédire y à partir de x
float predictY(float a, float b, int x) {
    return a * x + b;
}

int main() {
    // Données fournies
    int x[] = {50, 75, 100}; // Surface des maisons en m²
    int y[] = {70000000, 95000000, 130000000}; // Prix en MGA
    int n = 3; // Nombre de paires de données

    float a, b;

    // Appel de la fonction de régression linéaire
```

```
34      linearRegression(x, y, n, &a, &b);
35
36      // Affichage des résultats
37      printf("Coefficients de la régression linéaire :\n");
38      printf("a = %.2f\n", a);
39      printf("b = %.2f\n\n", b);
40
41      // Prédire le prix pour x = 80
42      int x_predict = 80;
43      float predicted_y = predictY(a, b, x_predict);
44
45      printf("Le prix prédit pour une surface de %d m² est : %.2f MGA\n"
            , x_predict, predicted_y);
46
47      return 0;
48  }
```

Explication du Code

Le programme calcule les coefficients a et b de la régression linéaire en utilisant les formules suivantes :

$$a = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{n\sum x_i^2 - (\sum x_i)^2}$$

$$b = \frac{\sum y_i - a\sum x_i}{n}$$

Ensuite, il utilise ces coefficients pour prédire y pour $x = 80$.

Résultats

En exécutant ce programme avec les données fournies, le programme affiche les coefficients de régression et la prédiction pour $x = 80$.

Régression Logistique

Implémentation en C

Voici l'implémentation en C pour résoudre l'exercice :

Listing 6.2 – Implémentation de la Régression Logistique en C

```
1   #include <stdio.h>
2   #include <math.h>
```

```
3
4   // Fonction pour calculer z
5   float computeZ(float beta0, float beta1, float x1) {
6     return beta0 + beta1 * x1;
7   }
8
9   // Fonction sigmoïde pour transformer z en probabilité P
10  float sigmoid(float z) {
11    return 1 / (1 + exp(-z));
12  }
13
14  int main() {
15    // Données d'exemple
16    float beta0 = 0.5;  // Coefficient intercept
17    float beta1 = 0.1;  // Coefficient pour x1 (temps)
18    float x1 = 12;      // Temps en minutes
19
20    // Calcul de z
21    float z = computeZ(beta0, beta1, x1);
22
23    // Calcul de la probabilité P
24    float P = sigmoid(z);
25
26    // Affichage des résultats
27    printf("Valeur de z : %.2f\n", z);
28    printf("Probabilité P : %.2f\n", P);
29
30    return 0;
31  }
```

Le programme utilise la fonction sigmoïde pour transformer la valeur z calculée en probabilité P, qui peut ensuite être utilisée pour prendre des décisions dans un modèle de régression logistique.

Explication

Le programme calcule d'abord z en utilisant la formule de la régression linéaire :

$$z = \beta_0 + \beta_1 \cdot x_1$$

Ensuite, il applique la fonction sigmoïde :

$$P = \frac{1}{1 + e^{-z}}$$

Cela donne la probabilité P associée à une certaine valeur x_1 (dans ce cas, 12 minutes).

K-means

Voici un programme en C qui résout l'exercice demandé. Le programme suit les étapes suivantes :

- Accepter une liste de points (x, y).
- Initialiser 2 centroïdes aléatoires.
- Assigner chaque point au centroïde le plus proche.
- Recalculer les centroïdes et afficher les groupes finaux.

Données

Les données d'entrée sont les suivantes :

Table 6.2 – Données pour K-means

Point	Coordonnées (x, y)
A	(1, 2)
B	(3, 4)
C	(5, 6)
D	(8, 9)

Implémentation en C

Voici l'implémentation en C pour résoudre l'exercice :

Listing 6.3 – Implémentation de l'algorithme K-means en C

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

// Structure pour un point (x, y)
typedef struct {
    float x;
    float y;
} Point;

// Fonction pour calculer la distance entre deux points
```

```
12  float distance(Point a, Point b) {
13    return sqrt(pow(a.x - b.x, 2) + pow(a.y - b.y, 2));
14  }
15
16  // Fonction pour initialiser les centroïdes aléatoires
17  void initializeCentroids(Point points[], int k, Point centroids[]) {
18    centroids[0] = points[0]; // Initialisation manuelle pour le
          premier centroïde
19    centroids[1] = points[3]; // Initialisation manuelle pour le
          deuxième centroïde
20  }
21
22  // Fonction pour assigner chaque point au centroïde le plus proche
23  void assignPointsToCentroids(Point points[], int n, Point centroids[],
        int clusters[]) {
24    for (int i = 0; i < n; i++) {
25      float minDistance = distance(points[i], centroids[0]);
26      clusters[i] = 0;
27
28      for (int j = 1; j < 2; j++) {
29        float dist = distance(points[i], centroids[j]);
30        if (dist < minDistance) {
31          minDistance = dist;
32          clusters[i] = j;
33        }
34      }
35    }
36  }
37
38  // Fonction pour recalculer les centroïdes
39  void recalculateCentroids(Point points[], int n, Point centroids[], int
        clusters[]) {
40    float sumX[2] = {0, 0}, sumY[2] = {0, 0};
41    int count[2] = {0, 0};
42
43    for (int i = 0; i < n; i++) {
44      sumX[clusters[i]] += points[i].x;
45      sumY[clusters[i]] += points[i].y;
```

```
46        count[clusters[i]]++;
47      }
48
49      for (int i = 0; i < 2; i++) {
50        centroids[i].x = sumX[i] / count[i];
51        centroids[i].y = sumY[i] / count[i];
52      }
53    }
54
55    int main() {
56      // Données des points
57      Point points[] = {{1, 2}, {3, 4}, {5, 6}, {8, 9}};
58      int n = 4;  // Nombre de points
59
60      // Initialisation des centroïdes
61      Point centroids[2];
62       initializeCentroids(points, 2, centroids);
63
64      // Assignation des points aux centroïdes
65      int clusters[4];
66      assignPointsToCentroids(points, n, centroids, clusters);
67
68      // Affichage des groupes après la première assignation
69      printf("Assignation des points aux clusters :\n");
70      for (int i = 0; i < n; i++) {
71        printf("Point (%f, %f) -> Cluster %d\n", points[i].x, points[i].y,
              clusters[i]);
72      }
73
74      // Recalcul des centroïdes
75      recalculateCentroids(points, n, centroids, clusters);
76
77      // Affichage des nouveaux centroïdes
78      printf("\nNouveaux centroïdes :\n");
79      for (int i = 0; i < 2; i++) {
80        printf("Centroïde %d : (%f, %f)\n", i + 1, centroids[i].x,
              centroids[i].y);
81      }
```

```
82
83      return 0;
84  }
```

Le programme fonctionne comme suit : 1. Il initialise deux centroïdes à partir des points donnés (ici, manuellement sélectionnés pour simplifier). 2. Chaque point est assigné au centroïde le plus proche. 3. Les centroïdes sont recalculés en prenant la moyenne des points de chaque groupe. 4. Enfin, les résultats sont affichés, montrant les groupes de points ainsi que les nouveaux centroïdes.

Explication des résultats

Après avoir exécuté le programme, vous obtiendrez l'assignation des points à leurs clusters respectifs et les nouveaux centroïdes recalculés. L'algorithme peut être itéré plusieurs fois pour améliorer les résultats.