Structure de données

https://sleek-think.ovh/enseignement

Jehan-Antoine Vayssade

Compétence visées

- Compréhension des différents types de structures de données
- Conception et mise en oeuvre en C
- Analyser et comprendre la complexité temporelle et spatial
- Comprendre les optimisations mémoire (cache miss)
- Savoir utiliser la bonne structure de donnée pour optimiser vos algorithms
- Utilisation avancée des pointeurs et de la gestion de la mémoire

Programme

- Niveau -1 : Présentation et notion de complexité
- Niveau 0 : Tableaux et listes (dynamique vs statique)
- Niveau 1 : Listes chainées simple, double, circulaires (push/pop/find)
- Niveau 2 : Piles et files (LIFO / FIFO) et cas d'utilisation
- Niveau 3 : Structures de données hiérarchique (arbre binnaire, arbre avl)
- Niveau 4 : Algorithme de trie et de recherche (dichotomique, quicksort, boustrophedon, …)

Notes

- Moyenne des TPs coef 1
  - ~1 TP par section
- Exam coef 1

Convention d’ecriture

- Style consistant !
  - Vous pouvez utiliser `clang-format`
  - Nommer correctement les fonction et variables (semantique)
  - Indentation (bloque)
  - Aérer les lignes du programme
  - Facilite la lecture !!!
- Pas de variables globales
- Commenter vos programmes
  - Pas de commentaire initile
- Attention sanction sur les notes de TP
- Example https://github.com/MaJerle/c-code-style

-2

```
#include<stdio.h>
#define MAX 1000
int a,b,c;float d;
char e[MAX];void F(int x){
if(x>0){printf("Positive");
}else{if(x<0){
printf("Negative");}else{printf("Zero");}}}
int main()
{
a=10;b=20;
c=a+b;
d=3.14159;
// This is a useless comment
printf("Hello World!");F(c);
for(int i=0;i<MAX;i++){e[i]='A';}
    if(d>3)
{
        printf("d is greater than 3");
    }
    else
        {
        printf("d is not greater than 3");
        }
return 0;
}
```

Niveau 0

Comprendre la complexité

Introduction

Pourquoi est-elle essentielle dans le développement logiciel ?

Objectifs de cette présentation :

- Définir la complexité des algorithmes
- Expliquer les notations asymptotiques (O, Ω, Θ)
- Discuter de l'importance des constantes dans ces notations
- Présenter des exemples pratiques en pseudo-C

Pourquoi est-ce important ?

- Optimisation des performances (CPU vs RAM)
- Provisionné vos ressources (contrainte matériel)
- Amélioration de la qualité du code

Définition de la complexité des algorithmes

Mesurer l'Efficacité des Programmes

La complexité d'un algorithme est une mesure de la quantifir les ressources nécessite pour traiter une donnée. Elle nous permet de prédire comment l'algorithme se comportera lorsque la taille des données augmente.

Types de Complexité :

- **Complexité Temporelle** : Temps nécessaire pour exécuter l'algorithme (CPU).
- **Complexité Spatiale** : Espace mémoire nécessaire pour exécuter l'algorithme (RAM).

Pourquoi est-elle importante ?

- **Optimisation des performances** : Améliorer la rapidité et l'efficacité.
- **Évolutivité** : Prévoir comment l'algorithme se comportera avec des données plus volumineuses.
- **Qualité du code** : Écrire des programmes plus efficaces et maintenables.

Similariter

- Pour faire simple compter le nombre d'opération quand : $\lim_{n \to +\infty} f(n)$

Exemple simple :

Supposons un algorithme de recherche dans une liste de n éléments.

Voici un pseudo-algo qui vérifie un par un les éléments de la liste pour le trouver

```
int recherche(int *array, size_t size, int value) {
    int index = -1;
    for (size_t i=0; i<size; ++i)
        if (array[i] == value)
            index = i;
    return index;
}
```

Cette algorithm a une complexité temporel lineaire, dans tout les cas le temps d'éxécution d'épendra de la taille de `@array`.

Notations asymptotiques

Les notations asymptotiques sont utilisées pour décrire la complexité d'un algorithme en fonction de la taille de l'entrée. Elles nous aident à prédire comment l'algorithme se comportera lorsque la taille des données augmente.

Big O

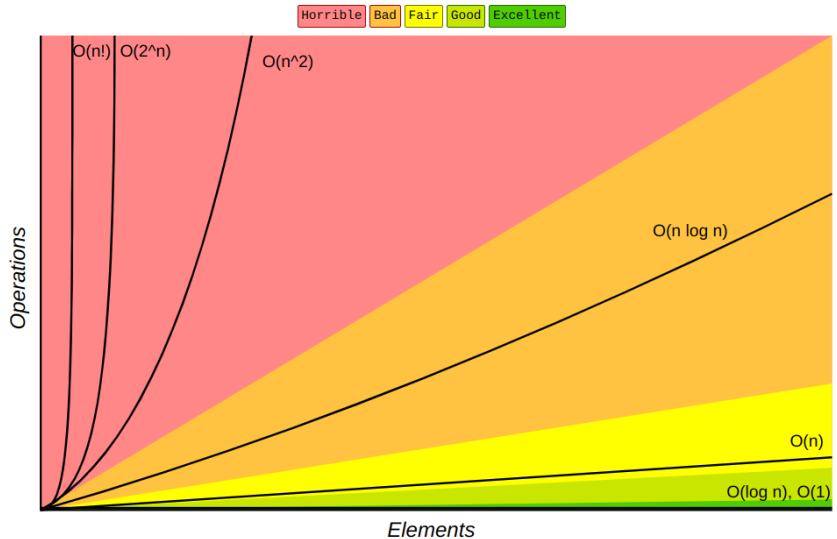
- **Définition :** Limite supérieure. C'est la pire des situations.
- **Interprétation :** L'algorithme ne prendra jamais plus de temps que O(n) pour une liste de n éléments.

Big Ω (parfois minus o)

- **Définition :** Limite inférieure. C'est la meilleure des situations.
- **Interprétation :** L'algorithme prendra au minimum Ω(n) temps pour une liste de n éléments.

Big Θ

- **Définition :** Limite exacte à la complexité d'un algorithme. Quand **O(f)=Ω(f)**
- **Interprétation :** L'algorithme prend exactement Θ(n) temps pour une liste de n éléments.



- 🚀 Complexité constante **O(1)**
- 🚂 Complexité logarithmique **O(log(n))**
- 🚗 Complexité linéaire **O(n)**
- 📈 Complexité quasi-linéaire **O(n · log(n))**
- ⚠️ Complexité quadratique **O(n²)**
- 🚧 Complexité exponentielle **O(2n)**
- 🚨 Complexité factorielle **O(n!)**

Notation asymptotiques et approximation

Dans la notation asymptotique, on oublie les termes constants

- $O(2n) = O(n)$: Les constantes sont ignorées, donc multiplier par 2 ne change "pas" la complexité.
- $O(3n^2) = O(n^2)$: De même, les constantes devant les puissances sont ignorées.
- $O(n + 10) = O(n)$: Les termes constants sont négligés par rapport aux termes variables.
- $O(n^2 + n) = O(n^2)$: Dans les polynômes, on ne garde que le terme dominant.

Pourquoi ignorer les constantes ?

- **Échelle de grandeur :** Lorsque la taille des données augmente, les constantes deviennent négligeables par rapport à la croissance globale.
- **Modélisation simplifiée :** Ignorer les constantes simplifie l'analyse et permet de se concentrer sur la tendance générale.

Attention

> Ses notions sont a savoir et utilisées dans les tests de recrutement, mais c'est fondamentalement incorrect !

Pourquoi les constantes sont-elles importantes ?

- **Performances réelles :** Dans la pratique, les constantes peuvent avoir un impact significatif sur les performances, surtout si elles représentent des opérations complexes ou des accès mémoire coûteux.
- **Optimisation :** Ignorer les constantes peut conduire à négliger des opportunités d'optimisation importantes. Par exemple, un algorithme avec une constante plus petite mais une complexité similaire est plus rapide en pratique.
- **Cas réels :** Dans de nombreux cas réels, les tailles des données ne sont pas toujours très grandes, et les constantes peuvent donc jouer un rôle crucial dans la performance globale.
- **Taille des données :** Si la notation Big O ce concentre sur de large échelles de donnée, en pratique vous connaitrez presque toujours les tailles. Elles seront presque toujours petites dans ce cas un algorithme en O(n) peut être meilleur qu'un algorithm en O(log(n))'


What Big-O notation ACTUALLY tells you, and how I almost failed my Google Inte...
Partager
Big-O
Regarder sur YouTube

Exemples de complexité en Θ(1)

_

Toute opération qui accede directement a une donnée:

```
int tableau[1000];
int index = 5;
int valeur = tableau[index];
```

ou effectue des calcules

```
bool estPair(int nombre) {
  return nombre % 2 == 0;
}
```

- Ceci n’est pas vrais pour certaines structures de données complexe (hash map, bin tree, etc) "caché"
- Quelle complexité est caché pour la fonction estPair ?

Quelle est la complexité de ce code ?

- Complexité spatial ?
- Complexité temporel ?

```
int x = 3;
int n = 10;
int result = pow(x, n) * sqrt(x, n);
```

- **A** : $\Theta(1)$
- **B** : $\Theta(2)$
- **C** : On ne sait pas ?
- **D** : Expliquer

Exemples de complexité en Θ(n)

```
double pow(double base, int exponent) {
  double result = 1.0;
  for (int i = 0; i < exponent; i++) {
    result *= base;
  }
  return result;
}
```

```
int recherche(int *array, size_t size, int value) {
    int index = -1;
    for (size_t i=0; i<size; ++i)
        if (array[i] == value)
            index = i;
    return index;
}
```

Quelle est la complexité de ce code ?

```c
int recherche(int *array, size_t size, int value) {
  for (size_t i=0; i<size; ++i)
    if (array[i] == value)
      return i;
  return -1;
}
```

- Complexité spatial ?
  - **A**: O(1)
  - **B**: Θ(1)
  - **C**: Ω(1)
- Complexité temporel ?
  - **A**: O(n)
  - **B**: Θ(n)
  - **C**: Ω(1) et O(n)

Exemples de complexité en $\Theta(\log(n))$

```
double pow(double base, int exponent) {
    double result = 1.0;

    while (exponent > 0) {
        if (exponent % 2 == 1)
            result *= base;
        exponent /= 2;
        base *= base;
    }

    return result;
}
```

Exemples de complexité en $\Theta(\log(n))$

```
double sqrt(double x) {
    if (x < 0)
        return NAN;
    else if (x == 0 || x == 1)
        return x;

    double guess = x / 2.0;
    double precision = 0.000001;

    while (1) {
        double betterGuess = (guess + x / guess) / 2.0;
        if (fabs(guess - betterGuess) < precision)
            return betterGuess;
        guess = betterGuess;
    }
}
```

Peut-on faire mieux ?

John Carmack et Quake III

```
float fast_sqrt(float number) {
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    y = number;
    i = *(long*)&y;
    i = 0x5f3759df - (i >> 1);
    y = *(float*)&i;

    y = y * (threehalfs - (x2 = number * 0.5F) * y * y);
    y = y * (threehalfs - (x2 = number * 0.5F) * y * y);

    return 1/y;
}
```

https://en.wikipedia.org/wiki/Fast_inverse_square_root

Exemples de complexité en Θ(n^2)

```
double pow(double base, int exponent) {
    double result = 1.0;

    for (int i = 0; i < exponent; i++)
        for (int j = 0; j < exponent; j++)
            if (i == j)
                result *= base;

    return result;
}
```

> Attention, le compilateur passe par là et fait aussi des optimisations ! (-Ofast -O3 -Og) ;)

Niveau 1

Tableaux et listes

Tableau statique

Les tableaux statiques sont des tableaux dont la taille est fixée au moment de la compilation.

- Ont peu calculer la taille des tableaux statique à l'aide de `sizeof(localArray)/sizeof(*localArray)`
- Attention l'espace de stockage diffère et peu dans certain cas être `read only`

```
// stockage dans .text / might be read only
static const char DATA[] = "SOME DATA";
// stockage dans .data
int globalArray[10] = {1, 2, 3, 4, 5, 0, 0, 0, 0, 0};
// stockage dans .bss
int globalUninitializedArray[10];
// stockage dans .rodata ! / read only
const int globalConstArray[] = {1, 2, 3, 4, 5};
int data[] __attribute__((section("rodata"))) = {1, 2, 3, 4, 5};

int main() {
    // stockage dans .data
    static int localStaticArray[10] = {1, 2, 3, 4, 5, 0, 0, 0, 0, 0};
    // stockage dans .bss
    static int localStaticUninitializedArray[10];
    // dans une fonction, stockage dans la stack
    int localArray[10];
}
```

Tableau statique

Sur une architecture 64bits : Que fait le code suivant ?

```c
#include <stdio.h>

void afficheTaille(int *array) {
  printf("%ld\n", sizeof(array));
  printf("%ld\n", sizeof(array)/sizeof(*array));
}

int main() {
  int localArray[10];
  printf("%ld\n", sizeof(localArray));
  printf("%ld\n", sizeof(localArray)/sizeof(*localArray));
  afficheTaille(localArray);
}
```

Tableau dynamique

Les tableaux dynamiques sont des tableaux dont la taille est déterminée au moment de l'exécution.

- La mémoire des tableaux dynamiques est allouée à l'aide de fonctions d'allocation
- `malloc()`, `calloc()` et `realloc()`.
- Attention aux fuite mémoire, ne pas oublié `free()`
- Il faut stocker la taille du segment associé, ou avoir une méthode de détection

```c
unsigned int size = 10 * sizeof(int);
// stockage dans la heap
int *array = malloc(size);
// attention, malloc may fail (return NULL)
...
unsigned int new_size = 100 * sizeof(int);
int* temp = realloc(array, new_size);
// attention, realloc may fail (return NULL, @array still valid)
array = temp;
...
free(array);
```

Liste chainées

- **Concept :**
  - On associe une donnée avec un ou plusieurs pointeurs
- **Avantages :**
  - Insertion et suppression efficaces : En modifiant les pointeurs des nœuds adjacents.
  - Utilisation "efficace" de la mémoire : Seule la mémoire "nécessaire" est allouée
- **Inconvénients :**
  - Recherche très lente : Doit parcourir la liste séquentiellement (mémoire non contigue).
  - Peu d'optimisation possible

Pensé a utiliser une structure pour géré votre liste :

```
typedef struct LinkedList {
    struct Node* head;
    struct Node* tail;
    // etc
} LinkedList;
```

Liste chainées simple

- **Définition :** Chaque élément (ou nœud) pointe vers le suivant.
- **Head :** Le premier élément de la list (NULL si vide)
- **Tail :** Le dernier élément de la list (NULL si vide, next = NULL)

```c
typedef struct IntNode {
    int data;
    struct IntNode* next;
} IntNode;

typedef struct UserDataNode {
    int size;
    void *data;
    struct UserDataNode* next;
} UserDataNode;
```

Liste chainées double

- **Définition :** Chaque élément (ou nœud) pointe vers le suivant et le précédent.
- **Mémoire :** Consomme plus de mémoire.
- **Head :** Le premier élément de la list (prev = NULL)
- **Tail :** Le dernier élément de la list (next = NULL)

```c
typedef struct IntNode {
    int data;
    struct IntNode* next;
    struct IntNode* prev;
} IntNode;

typedef struct UserDataNode {
    int size;
    void *data;
    struct UserDataNode* next;
    struct UserDataNode* prev;
} UserDataNode;
```

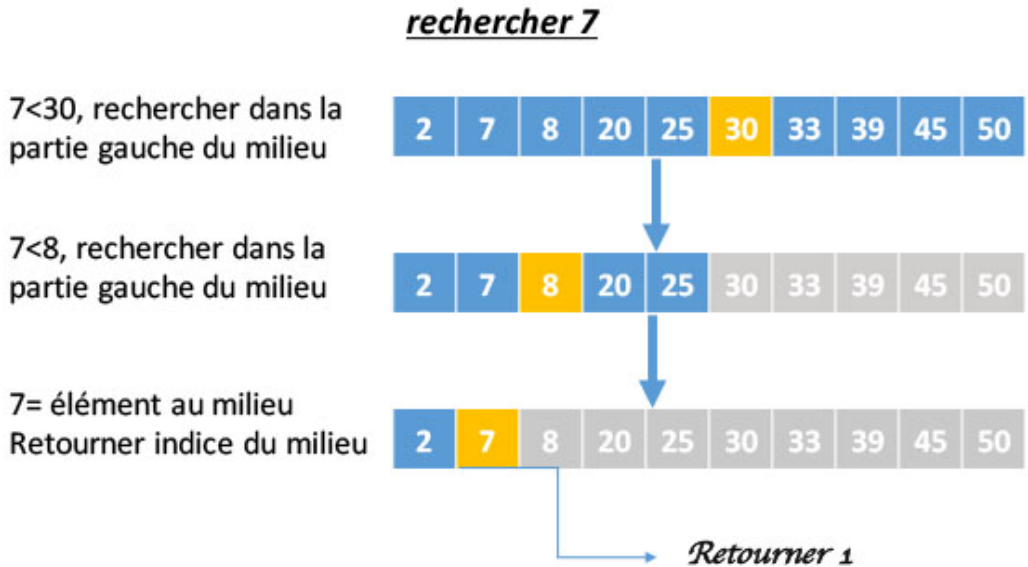
Liste chainées circulaire

- **Définition :** Il n'existe pas de `head` ou de `tail` mais plutot un `cursor`
- **Simple ou double :** A choisir en fonction des besoins
- **Prev :** Prédécésseur n'est jamais null (sauf si list vide)
- **Next :** Sucesseur n'est jamais null (sauf si list vide)

Circular Buffer

- Même principe mais dans une zone mémoire contigue
- Implémenter les deux versions

Recherche dichotomique



- Implémenter une recherche dichotomique
- Quelle est la complexité temporel de cet algorithm ?
- https://sleek-think.ovh/index.php/cours/11-l3-algorithmie/12-recherche-dichotomique

Exercice :

- Partir du code https://classroom.github.com/a/g5Lddf1f
- Ecrire un code dans un nouveau fichier (compilation séparé)
  - Les tableaux dynamique et statique (`dynamic_array.c` et `static_array.c`)
  - Les listes simple `single_linked_list.c` et double `double_linked_list.c`
  - Les listes circulaire `circular_linked_list.c`
  - **Bonnus :** Écrire un CircularBuffer (Tableaux dynamique + List Circulaire)
- Pour chaque méthode implémenter `push_front` / `push_back` / `insert(index)` / `remove(index)`
  - Ex: `push_front(SingleLinkedList*, SLL_IntNode*)`
- Pour chaque méthode expliqué les différents problèmes
  - Complexité temporel : appoximation
  - Complexité spatial
  - **Bonnus :** Expliqué les problemes de cache miss associé, avec schéma
- Chercher des exemples de cas d'utilisation de ces structures de données
- Petit rapport

Problème de cache miss

- **Accès imprévisible à la mémoire :** Les nœuds sont dispersés dans la mémoire.
- **Manque de localité :** Pas d'accès aux emplacements de mémoire direct [index].
- **Augmentation du temps d'accès à la mémoire :** Les échecs de mise en cache entraînent un ralentissement des performances en raison de l'attente des données de la mémoire principale.
- **Chargement inefficace des blocs de mémoire :** Un seul nœud par bloc de cache est utilisé
- **Impact :** Ralentissement de la traversée, recherche très lente.

Niveau 2

FIFO et LIFO

Introduction à FIFO

FIFO est un principe de gestion des données où le premier élément ajouté est le premier à être traité. Cela ressemble à une file d'attente dans la vie réelle, où la personne qui arrive en premier est servie en premier.

Opération	Description
Enqueue	Ajouter un élément à la fin de la file.
Dequeue	Supprimer l'élément du début de la file.
Peek	Voir l'élément du début sans le supprimer.

On peut implémenter un FIFO de déffirentes façon

- Avec un buffer (gestion des reallocation potentiel)
- Avec une list chainé

Écrire un code de test de vos structure de donnée permettant de faire une FIFO

Introduction à LIFO

LIFO est un principe de gestion des données où le dernier élément ajouté est le premier à être traité. Cela ressemble à une pile de livres, où le livre posé en dernier est le premier à être retiré.

Opération	Description
Push	Ajouter un élément au sommet de la pile.
Pop	Supprimer l'élément du sommet de la pile.
Peek	Voir l'élément du sommet sans le supprimer.

On peut implémenter un LIFO de différentes façons :

- Avec un tableau (gestion des réallocations potentielles)
- Avec une liste chainée

Écrire un code de test de votre structure de données permettant de faire une LIFO

Quelles sont les différences ?

Caractéristique	FIFO	LIFO
Ordre de Traitement	Premier entré, premier sorti	Dernier entré, premier sorti
Utilisations	Planification des tâches, messagerie	Appels de fonctions, annulation
Implémentation	Files	Piles

Niveau 2

FIFO et LIFO

Arbres binnaire

Un arbre binaire est une structure de données non linéaire où chaque nœud peut avoir au maximum deux enfants. Les arbres binaires sont utilisés pour représenter des hiérarchies et sont efficaces pour certaines opérations comme la recherche et l'insertion.

- **Maximum de deux enfants** : Chaque nœud a au plus deux enfants.
- **Structure hiérarchique** : Les nœuds sont organisés de manière hiérarchique.
- **Flexibilité** : Peuvent être utilisés pour diverses applications sans nécessiter un ordre spécifique.
- **Problème de balancement :** Largeur et profondeur variable -> change la complexité temporel

Comment peut-on écrire une structures de données en C ?

Comparaison avec la Recherche Dichotomique

Caractéristique	Arbre Binaire	Recherche Dichotomique
Structure	Hiérarchique avec nœuds	Tableau triée
Recherche	O(log n) en moyenne pour les arbres équilibrés	O(log n) pour les listes triées
Insertion	Peut être complexe si l’arbre devient déséquilibré	Simple mais nécessite de réorganiser la liste
Utilisation	Utile pour les données hiérarchiques et les requêtes complexes	Idéal pour les recherches rapides dans des listes triées