# Algorithme

## introduction

Les caractéristiques d'un logiciel la fiabilité, la robustesse, l'extensibilité, la réutilisabilité, la compatibilité et l'efficacité. Algorithme : ensemble finit d'étape dont le but est de résoudre un problème ou d'accomplir une tâche. Ces étapes sont formées d'un ensemble fini d'opérations. Une recherche dichotomique a une complexité en O(log(n))

## Complexité

Cela définit, en termes de ressources, le besoin d'une algorithme pour être exécutée. L'efficacité se mesur en fonction d'un paramètre n qui caractérise la quantité d'éléments à traiter. La complexité caractérise son coût en temps processeur en fonction de la taille de la données. Cependant on ne cherche pas une mesure précise mais une estimation du nombre d'opérations élémentaires nécessaires.

- Complexité de O(1)
- Complexité de O(N)
- complexité quadratique
- sont de complexité logarithmique
- La complexité dans le meilleur des cas, dans le pire des cas et en moyenne

Le tri d'un vecteur, par exemple, peut faire appel au probabilités. Dans le meilleur des cas il est déjà trié, dans le pire des cas il est trié à l'envers. Dans ce genre de cas la complexité moyenne est la même que la complexité dans le pire des cas. La complexité de la recherche séquentielle est dans le meilleure des cas de O(1) si on trouve directement. Dans le pire des cas on parcout tout l'objet et la complexité est de O(n). Le cas moyen donne $(n+1)/2$ ce qui vaut à O(n).

## Notation de Landau

Une fonction f est un grand O de la fonction g. Ce qui signifie que f croît au plus vite que g. De cela découle deux règles simple:

## Notations complémentaires

La notation o() est notée quand f(n) croît strictement plus lentement que g(n). La notation () est utilisée quand f(n) croît au moins aussi vite que g(n). La notation () est utilisée quand f(n) est du même ordre de grandeur que g(n).O(log_(\phi)(n))

si une fonction est la somme de plusieurs termes, on garde que celui qui croît le plus vite. Si une fonction est le produit de plusieurs termes, on peut ignorer le facteur constant.

f = O(g) f croît au plus vite que g f = o(g) f croît strictement plus lentement que g. f = Ω(g) f croît au moins aussi vite que g . f Θ(g) f et g ont le même ordre de grandeur.

# Recursivité

L'algorithme de Fibonacci ne doit surtout pas être écrite de manière récursive. Quand un appel est tout à la fin de la fonction, l'optimisation du compilateur fait que la fonction a la même exécution qu'une fonction itérative.

La complexité d'une fonction récursive est en général constante si on omet les appels récursif. Ce qui veut dire que la complexité final dépend uniquement du nombre d'appel. Dans le cas d'une fonction calculant la factorielle de N, la complexité est de O(N)

Dans le cas de la fonction de Fibonacci, la complexité est de $x^{\phi}$, ou est le nombre d'or. Pour l'algorithme d'Euclide le pire cas est quand une des opérandes est un nombre de Fibonacci et que l'autre est un nombre de Fibonacci de valeur k-1. La complexité dans ce cas est de $O(log_{(\phi)}(n))$.Dans le cas des tours de Hanoï, la complexité dépend du nombre et donc du nombre de déplacement. Pour n nombre de disque, on aura N-1 déplacement pour déplacer tous les petits disques du piquet de base au piquet intermédiaire + 1 déplacement pour le grand disque sur le piquet final.

Une fonction factorielle a une complexité de O(N) , de O(log(n)) si on divise n dans l'appel de la fonction. L'algorithme de Fibonacci est de O(log(n)) avec comme base ( 1 ± √5 ) / 2. le nombre de déplacement pour les tours de hanoï est de 2^n -1 et sa complexité est exponentielle O(2^n).

## Complexité exo prog

La tour de Hanoï a une complexité qui a une complexité de =(2^n). Le triangle récursif, la complexité est de O(a^2 - b^2). Le PGDC, la complexité est de O(log_x(a)) avec phi comme x. Les fractales, la complexité est de O(4^n + i*2^n) car le 4^n calcul un carré et le i * 2^n le triangle. Et pour les conversions de base, la complexité est de O(log_B(n)).

# Tris

## Tri à bulles

Il compare deux élément successif d'un tableau. S'ils sont en désordre ils sont échangé. au niveau mémoire juste besoin d'une variable tampon.

## Tri par sélection

Recherche de l'élément le plus petit et l'échange avec le début du tableau.

## Tri par insertion

On prend le premier élément non trié et tant qu'il est plus petit que le précédent, on le permute avec celui-ci.

## Tri de shell

On divise le tableau en sous tableaux d'abord très grand puis de plus en plus petit jusqu'à des tableaux d'un élément. la complexité dépend de la séquence choisi pour h (taille du tableau). La performance est assez proche du tri par fusion. les petits sous-tableaux sont triés par insertion h(x)=3h(x-1)+1.

## Tri fusion

On divise en sous-tableaux, on compare à chaque fois les premiers. Pour les petits tableaux, on fait un tri pas insertion.

## Tri rapide

Pour optimiser le tri rapide, on fait un appel récursif seulement sur la partition la plus petite. le pire cas est en théorie impossible si on choisit bien le pivot. le quick select et nth_element à la même complexité que le tri rapide.

le C++ sort a besoin de l'opérateur < pour fonctionner, il garanti une complexité de O(n log(n)) mais pas la stabilité contrairement à stable sort.

- tri stable : tri à bulles, tri par insertion, tri par fusion.
- tri par échange : tri à bulle, tri par sélection, tri rapide.
- tri qui ne dépendent pas de la taille des données :tri à bulle, tri par sélection, tri par fusion. le tri par insertion dépend de l'état de l'entrée.

# Structures linéaire

## Tableau à taille fixe

Un tableau de taille fixe est constitué d'un pointeur sur le premier élément et de sa taille. Les éléments sont accéder par l'indice physique. L'indice pour accéder à chaque élément est valide de 0 à N compris. On peut accéder à son emplacement mémoire via la formule: adresse <-- tableau + i *taille(type).

La notion de capacité fixe permet de pouvoir insérer et supprimer en fin en changeant la variable taille. Si ces actions ne sont pas possible, il faut alerter l'utilisateur. Même chose s'il on veut insérer et supprimer en position quelconque.

Le buffer circulaire permet de se libérer des contraintes de commencer à l'indice 0. Ce dernier possède un indice physique pour l'emplacement mémoire et un indice logique pour les indices utilisateur.

## Tableau de taille variable et de capacité fixe

Cela permet d'allouer dynamiquement de la mémoire si la capacité n'est plus suffisante. S'il n'y a plus de capacité suffisante, il faut allouer un espace mémoire plus grand, déplacer les éléments contenu de l'ancien espace data puis détruire l'ancienne espace de data.

Complexité de l'insertion en fin: si assez de mémoire O(1) dans le pire des cas, il est de O(N). le cas moyen se situe dans l'intervalle 2 et 3 on dit alors que la complexité et constante en temps amorti. Une bonne méthode pour gérer une capacité plus gande que la taille est d'attendre que taille sois 4 fois plus petite que capacité.

vector sort O(N*log(N)). si erase au début = pop_back = O(1).

# Listes

## simplement chaînées

Les éléments ne sont pas forcémt consécutif. Mis en oeuvre avec une structure générique. pour le parcours il est préférable de faire de manière itérative. L'encapsulation permet de ne pas avoir de problème lors de la manipulation des éléments. Le mieux serait d'avoir des itérateurs ainsi que des opérateurs pour accéder et passer au suivant (approche de la STL).

## doublement chaînées

L'utilisation d'un attribut vide pour stocker l'adresse du premier et l'adresse du dernier. Pour une liste vide, les deux pointeurs pointe sur lui même. la méthode dernier retourne l'adresse de apad.

la méthode emplace permet d'insérer en place avec une complexité constante

la méthode splice insert à l'endroit voulu en une complexité constante à linéaire

la méthode insert est de complexité linéaire

## Tri

On ne peut pas appliquer les algorithmes de tri vu précédemment car il n'y pas d'accès indexé. Les tris par sélection, par insertion et à bulles peuvent être adapté pour les listes. le tri se fait en réordonnant les éléments. Il ne faut pas changer la valeur de l'élément. Splice after permet de déplacer des maillons pour le tri par insertion. La fonction splice nous permet de mettre en œuvre le tri fusion. Cette dernière a une complexité temporelle et spatiale constante.

# Deque

Les données sont stockées dans plusieurs petits tableaux de capacité fixe. Ils sont appelés chunk. Les adresse de ces chunk sont stocké dans un buffer circulaire, appelé map, qui est a capacité variable. Pour accéder aux éléments, il faut pratiquer un double déréférencement. La map et le chunk possède chacun un méthode pour le calcul de l'indic physique.

# Tas

Le tas est un tableau de taille variable que l'on représente comme un arbre bianaire. Chaque élément a un parent sauf la racine, aucun élément n'a plus de deux enfants. La complexité pour trouver l'élément le plus grand est constante car c'est le premier. L'insertion et la suppression se font avec une complexité logarithmique.

push_heap : L'insertion se fait en fin de tableau. Ensuite il faut faire remonter l'élément jusqu'à que la condition de tas soit respectée.

pop_heap : La suppression s'effectue en général sur l'élément le plus grand. Pour ce faire on échange la racine et le plus petit élément et ensuite on supprime le dernier. Pour rétablir la condition de tas, on fait redescendre l'élément par le plus grand de ses enfants jusqu'à respecté la condition de tas.

make_Heap sert à faire le tas, il fait descendre les parents, s'ils sont plus petits, vers le plus grand enfant. La complexité est de O(N*log(N)) car chaque insertion est logarithmique.

sort_heap: il faut créer le tas avec make heap. Ensuite on prend à chaque fois la racine et on la met en dernière position et on fait remonter le plus grand des enfants. ce tri est instable et s'effectue en place.

les conditions de tas sont :

- tout élément est plus grand (ou égal) que ses enfants
- réciproquement, tout élément est plus petit (ou égal) que son parent

# Type de donnée abstrait

Ces types restreignent la manière d'insérer, d'accéder et de supprimer les éléments.

## Pile

Principe LIFO. opérations de base push, pop et top. Les évaluations d'expressions arithmétiques se fait lorsqu'on rencontre des ')'. Pour stocker les opérandes et les opérateurs, on utilise deux piles et on évalue dès qu'une ')' est là. la notation polonaise inverse permet d'éviter davoir deux piles.

## Queue

Principe FIFO, opérations de bases push, pop, front et back

## priority_queue

Même chose que queue mais avec notion de priorité. Si on rentre une paire d'élément dans la file, on ressort en regardant d'abord le premier mais si ils sont égales on regarde le deuxième.

emplace sort les éléments dans l'ordre inverse de pop.

# Structures linéaire en C++

- Sequence container : Array, Vector, Deque, List, Forward_list.
- Container adaptors: Stack, Queue, priority_queue.
- Les fonctions de tas sont aux nombre de 6. push_heap, pop_heap, make_heap, sort_heap, is_heap et is_heap_until.

# Allocation dynamique

```
T* ident = new T; // ident est un pointeur sur la variable de type T
int* i = new int(42); // ou {42}
delete ident;

T* ident = new T[N]; // ident est un pointeur sur le tableau de type T
// on doit initialiser chaque élément
delete[] ident;
// Il est autoriser un tableau de taille nulle
```

Il faut appeler delete dans le flot d'exécution pour éviter une fuite de mémoire. Il est de bonne pratique de mettre le pointeur supprimé à nullptr. L'opérateur new appelle le constructeur spécifié, malloc et calloc allouent juste de la mémoire sans appeler le constructeur. Même différence entre delete et free.

```
T* Ptr = (T*) ::operator new(N*sizeof(T)); // ne construit rien mais alloue la mémoire
::operator delete(Ptr);// ne détruit pas mais libère la mémoire
```

Les destructeurs ne lèvent pas d'exceptions. Les constructeurs lèvent soient des std::bad_alloc ou des exceptions du constructeurs.

Trois sources de fuite de mémoire:

- écraser la valeur du pointeur
- perdre le pointeur
- le programme ne s'exécute pas de manière attendue et fait perdre le pointeur Si une exception est levée elle doit appelé les destructeurs. Le constructeur de copie par défaut va copier tout les attributs d'une classe. Cela pose problème lors de leurs destructions car ils ont les même pointeurs. Pour pouvoir trier une classe, il faut mettre en oeuvre les opérateurs > et <. Un emplace_back appelle juste le constructeur avec élément, le push_back appelle le constructeur vide puis l'opérateur de copie, et le destructeur. Si on affecte vecteur avec un type complexe, il va construire un élément, affecter ce élément puis le détruire. La complexité de std::swap est de O(N). si on fait une méthode qui une fonction qui swap simplement les éléments, on passe à O(1). Une rvalue reference est une rvalue dont l'existence est temporaire, une expression en cours d'évaluation, une valeur en retour de fonction. Le move est noexecpt contrairement à l'opérateur d'affectation.

# Arbres

Le but est d'insérer et de supprimer en O(log(N)) et qui permet de chercher en O(log(N)).

## arbres quelconques

- degré de noeud : nombre de fils
- Racine : seul noeud sans père
- étiquette: information associée à un noeud
- feuille : noeud sans fils
- chemin : suite de noeuds reliant la racine à un noeud
- niveau: nombre de noeuds sur le chemin du noeud
- noeud interne : noeud avec fils
- hauteur longueur du chemin le plus long

- degré de l'arbre: degrés max de ses noeuds
- sous-arbre : sous ensemble de noeuds ayant une structure d'arbre
- arbre plein: degré(N) = degré(arbre), en tout nœud interne N
- arbre complet : arbre plein, rempli depuis la gauche.
- arbre dégénéré: degré de chaque noeud interne vaut 1 Le parcours en profondeur est un algorithme récursif qui permet de parcourir de haut en bas et de gauche à droite. Le parcour en largeur est un algorithme nécessitant une queue où on fait chaque niveau de gauche à droite.

# Arbres binaires

C'est un arbre de degré 2. Un parcour préordonné est un parcours en profondeur. Un parcours symétrique consiste à prendre l'enfant gauche, la racine et l'enfant droit. Le parcours postordonné prend l'enfant gauche, droit et la racine. Puis parcours en largeur. Les expressions arithmétiques sont équivalentes à des arbres dont les feuilles sont des valeurs et les noeuds internes des opérations. On peut rechercher, insérer, supprimer, un élément dans un ABR. Le rang d'une clé est toujours de 1 pour une feuille.

| | **complexité min** | **complexité moyenne** | **complexité max** |
|---|---|---|---|
| Recherche | O(h) | O(log(N)) | O(N) |
| insertion | O(h) | O(log(N)) | O(N) |
| suppression | O(h) | O(log(N)) | O(N) |
| min et max | O(h) | O(log(N)) | O(N) |
| rang | O(h) | O(log(N)) | O(N) |
| sélection | O(h) | O(log(N)) | O(N) |
| Parcours(pré, post, sym) | - | O(N) | O(N) |
| élément suivant | - | O(1) | O(h) |
| tri | | O(n*log(n)) | |

L'équilibrage permet d'éviter de grande complexité pour avoir une hauteur en O(log(N)). On considère qu'un arbre est équilibré si la différence entre la hauteur de l'enfant gauche et droit est plus grand ou égal à 1.

# Set

Les sets stockent des éléments dans un ordre spécifique et ne peuvent être modifiés dans la structure. utilisé pour les ABR.

# Multiset

si on insère des éléments non trié c'est un complexité linéarithmique sinon linéaire.

# Map

Structure formé par une combianaison de clé et de valeur. Les clés sont utilisé pour triés la map.

## Multimap

complexité comme Multiset.

| | Map | Multimap | Set | Multiset | priority_queue |
|---|---|---|---|---|---|
| insertion | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) |
| suppression | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | |
| recherche | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | |

# Arbres

# Allocation dynamique

# Graphes non orientés

# complexité