Algorithmie

Complexité et structures de données

BOULCH Alexandre

Introduction

Alexandre Boulch

Ingénieur-chercheur à l'ONERA.

Contact

Site web : `aboulch.github.io`

email : `boulc-ha@imagine.enpc.fr`

Algorithmie

Introduction à la programmation C++

Apprendre les bases du langage.

Algorithmie

- utiliser le langage
- programmer plus efficacement
- connaître quelques algorithmes classiques

Plan de la séance

Rappel sur les tableaux

Complexité

Structures de données

Rappel sur les tableaux

En C++ les tableaux sont :

- de **taille fixe**
- soit **statiques** et doivent avoir une taille connue par le compilateur
- soit **dynamiques** et il faut s'occuper de la mémoire
- difficiles à copier

Pour simplifier

Il est plus facile d'utiliser les vecteurs (`vector`) de la STL.

Plan de la séance

Notion de complexité

Définition

La **complexité** d'un algorithme est une grandeur définie pour un algorithme qui sert à estimer son efficacité en fonction des **paramètres** du problème.

Précision

La complexité représente le comportement asymptotique de l'algorithme (lorsque les paramètres deviennent très grands).

Notion de complexité 2

Les type de complexité

- La **complexité en temps** : le nombre d'opérations élémentaires nécessaires pour effectuer l'algorithme
- La **complexité en espace** : la taille de la mémoire nécessaire.

Notion de complexité 2

Les type de complexité

- La **complexité en temps** : le nombre d'opérations élémentaires nécessaires pour effectuer l'algorithme
- La **complexité en espace** : la taille de la mémoire nécessaire.

Remarques

- La complexité en espace et en temps sont complémentaires :
  - Stocker tous les résultats (grand espace mémoire)
  - Calculer à chaque besoin (faible espace mémoire, peut être lent)
- En général, la complexité en espace n'est pas un problème (PC), c'est le temps qui importe (jeu vidéo : *temps réel*).

L'histogramme d'une image

```
int histo[256];
for(int i=0; i<256; i++){
    histo[i] = 0;
}
for(int x=0; x<image.width(); x++){
    for(int y=0; y<image.height(); y++){
        histo[image(x,y)]++;
    }
}
for(int i=0; i<256; i++){
    drawRect(i,0,1,histo[i])
}
```

Analyse

- Mémoire : 1 tableau de 256 cases
- Temps : chaque pixel est visité une fois, $W \times H = N$

L'histogramme d'une image

```
for(int i=0; i<256; i++){
    int h=0;
    for(int x=0; x<image.width(); x++){
        for(int y=0; y<image.height(); y++){
            if(image(x,y)==c){
                h++;
            }
        }
    }
    drawRect(c,0,1,h);
}
```

Analyse

- Mémoire : pas de tableau
- Temps : 256 passages sur chaque pixels, $256 \times N$

Mesure de complexité

Il est difficile de connaître exactement le nombre d'opération élémentaires qu'effectue un algorithme. Pour plus de commodité :

Notation

On utilise la notation $O(f(N_1, N_2, N_3...))$ où $N_1, N_2, N_3 \dots$ sont les paramètres qu'on veut observer.

Algo précédents

- Cas 1 : Espace $O(nombre\_couleurs)$, temps $O(nombre\_pixels)$
- Cas 2 : Espace $O(1)$, temps $O(nombre\_pixels * nombre\_couleurs)$

Notion de complexité : exemples simples

- Parcours des éléments d'un tableau

```
for(int i=0; i<tab.size(); i++){ // utilisation d'un tableau de type vector
  cout << tab[i] << endl;
}
```

Complexité $O(N)$ (N est la taille du tableau), on accède une fois à chaque case.

- Recherche (naïve) de l'unicité des éléments dans un tableau

```
// unicite dans tab
vector<bool> unique(tab.size(),false);
for(int i=0; i<tab.size(); i++)
  for(int j=i+1; j<tab.size(); j++)
    if(tab[i]==tab[j])
      unique[i] = unique[j] = true;
```

Complexité $O(N^2)$ (N est la taille du tableau).

Exemple : le n^e terme de la suite somme

Le choix de l'implémentation dépend de l'application et influe beaucoup sur le temps de calcul.

Peu de mémoire

```
int s = 0;
for(int i=0; i<n; i++)
  s+= i;
```

Complexité $O(n)$.

Peu de temps

```
// precalcul
vector<int> tab(1000000000,0);
for(int i=1; i<tab.size(); i++)
  tab[i] = i + tab[i-1];

// utilisation
int s = tab[n];
```

Complexité $O(1)$ (en utilisation).

Les grandes classes de complexité

- $O(1)$: **constant**, pas d'influence des grandeurs du problème.
  Exemple : accès à une case d'un tableau
- $O(\log(N))$: **logarithmique**, algorithmes rapides, pas besoin de lire toutes les données.
  Exemple : recherche dans un tableau trié
- $O(N)$: **linéaire**, proportionnel au nombre d'éléments.
  Exemple : faire la somme des éléments d'un tableau

Les grandes classes de complexité

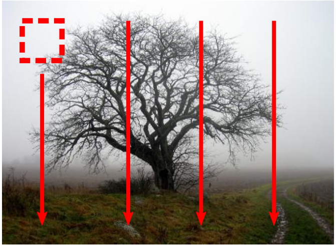
- $O(N \log(N))$: **algorithmes dits rapides**, fréquents.
  Exemple : Tri optimal, Fast Fourier Transform
- $O(N^k)$: **complexité polynomiale**, acceptable pour des tailles de données faibles (petit N) et des puissances faibles (petit k).
  Exemple : $O(N^2)$ tri naïf
- $O(2^N)$: **complexité exponentielle**, utilisable en pratique que pour les problèmes de petite dimension.
- ...

Exemple

Flou sur une image.

```
for(int i=l/2; i<W-l/2; i++)
  for(int j=l/2; j<H-l/2; j++){
        newIm(i,j) = 0;
        for(int k=i-l/2; k<i+l/2; k++)
          for(int m=j-l/2; m<j+l/2; m++)
            new(i,j) += im(k,m);
        newIm(i,j)/= l*l;
  }
```

Complexité $O(N \; l^2)$ (N est le nombre de pixels, l la taille de la fenêtre de flou)



P et NP

Les algorithmes P sont les algorithmes que l'on peut résoudre en temps polynomial. NP sont ceux pour lesquels ont ne connait pas de solution polynomiale.

Question

$P = NP$? ou $P \neq NP$? Question à 1M \$.

Relativisons

Attention

Les complexités sont asymptotiques : elles sont valables pour les grandes tailles de données. Les constantes multiplicatives sont oubliées dans la notation.

Exemple

Algo A, $O(N)$: $10^6 N$
Algo B, $O(N^2)$: N^2
L'algo A est plus rapide si $N > 10^6$

Complexité en moyenne et dans le pire des cas

La complexité d'un algorithme peut varier (présence d'aléatoire).
On distingue alors deux types de complexité :

- **Complexité moyenne** : caractérise le comportement attendu pour des répétitions.
- **Complexité dans le pire des cas** : caractérise le comportement dans la pire configuration des données.

Importance

L'application détermine le comportement important.

- **Complexité moyenne** : requêtes dans un moteur de recherche
- **Complexité dans le pire des cas** : aéronautique, temps à chaque utilisation.

Plan de la séance

Le vecteur

La classe

La classe est `std::vector` (**vector** si on a utilisé **using namespace std**). C'est un tableau encapsulé. Elle est « templatée », elle peut être utilisée pour contenir n'importe quoi.

Avantages

- Plus de mémoire à gérer
- Il connait sa taille
- On n'est plus obligé de déclarer sa taille au départ

Complexité du `push_back`

Implémentation naïve

Quand on ajoute un élément, on redimensionne le tableau avec taille plus grande d'une case.

Ceci implique une recopie du tableau à chaque `push_back`.

Complexité du `push_back`

Implémentation naïve

Quand on ajoute un élément, on redimensionne le tableau avec taille plus grande d'une case.

Ceci implique une recopie du tableau à chaque `push_back`.

Complexité

$$O(N)$$

Complexité du `push_back`

Implémentation judicieuse

- La taille du vector ne correspond pas la taille du tableau alloué.
- Quand on atteint la taille du tableau, on réalloue en multipliant cette taille par un facteur m.

Complexité

$$O(1)$$

Complexité du push_back : Preuve

On suppose n push_back.

Complexité du `push_back` : Preuve

On suppose n `push_back`.

Il y a aura k redimensionnements. Comme on multiplie la taille par m à chaque fois, on a :

$$n < m^k \Rightarrow k \approx log_m(n)$$

Complexité du `push_back` : Preuve

On suppose n `push_back`.
Il y a aura k redimensionnements. Comme on multiplie la taille par m à chaque fois, on a :

$$n < m^k \Rightarrow k \approx log_m(n)$$

Nombre de recopies :

$$\sum_{i=1}^{log_m(n)} m^i = m\frac{m^{log_m(n)} - 1}{m - 1} = \frac{mn}{m - 1}$$

Complexité du `push_back` : Preuve

On suppose n `push_back`.
Il y a aura k redimensionnements. Comme on multiplie la taille par m à chaque fois, on a :

$$n < m^k \Rightarrow k \approx log_m(n)$$

Nombre de recopies :

$$\sum_{i=1}^{log_m(n)} m^i = m\frac{m^{log_m(n)} - 1}{m-1} = \frac{mn}{m-1}$$

Coût moyen :

$$\frac{m}{m-1} = O(1)$$

Le vecteur : complexité

- Ajout à la fin : $O(1)$
- Supression à la fin : $O(1)$
- Ajout à position donnée (`insert(it,val)`) : $O(N)$
- Surpression à position donnée (`erase(it)`) : $O(N)$
- Lecture / Ecriture : $O(1)$

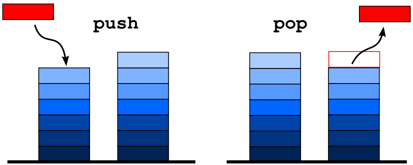
La pile (LIFO)

Principe

On ajoute les éléments sur le dessus et on retire les éléments là aussi par le dessus.

Iplémentations

- `vector` (`#include <vector>`) : `push_back(elt)`, `pop_back()`
- `stack` (`#include <stack>`) : `push(elt)`, `pop()`



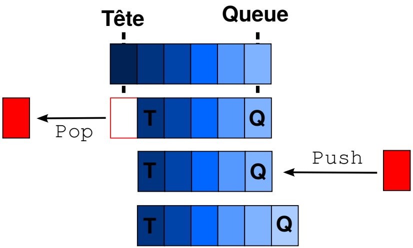
La file (FIFO)

Principe

On ajoute les éléments à l'arrière et on retire les éléments par l'avant (exemple file d'attente).

Méthodes principales

Comme dans le cas de la pile : `push` et `pop`.



File : implémentation

```cpp
class File{
  std::vector<double> v;
  int debut, fin;
  int modulo(int i) const;
public:
  File();
  bool empty();
  double front();
  void push(double d);
  double pop();
};
```

```cpp
File::File() { debut= fin = 0 ; }

bool File::empty() const {
  return ( debut==fin ) ;
}

// Tete de la file , prochain client
double File::front() const{
  return v[debut];
}

//arithmetique modulo v.capacity()
int File::modulo(int i){
  if(v.capacity()==0)
    return 0;
  return i%v.capacity();
}
```

File : implémentation

```cpp
//Ajout d'un element
void File::push(double d){
  int fin2 = modulo(fin+1);
  if(fin2==debut){
    std::vector<double> v2;
    for(int i=debut; i!=fin; i=modulo(i+1))
      v2.push_back(v[i]);
    v= v2;
    v.reserve(v.capacity()*2);
    debut = 0;
    fin2 = v.size()+1;
  }
  if(fin == v.size())
    v.push_back(d);
  else
    v[fin] = d;
  fin = fin2;
}
```

```cpp
//retrait de l'element de tete
double File::pop(){
  double d = front();
  debut = modulo(debut+1);
  return d;
}
```

File : complexité

- `push` : $O(1)$
- `pop` : $O(1)$

Dans la STL

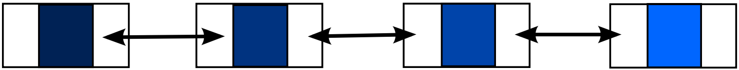
- `queue` (`#include <queue>`) : file
- `deque` (`#include <deque>`) : *double ended queue*

La liste chaînée

On observe que les structure vues précédemment ne sont efficaces que pour les ajouts en début ou en fin de tableau. Si on veut insérer ou supprimer au milieu du tableau, on utilise une **liste chaînée**.

Structure

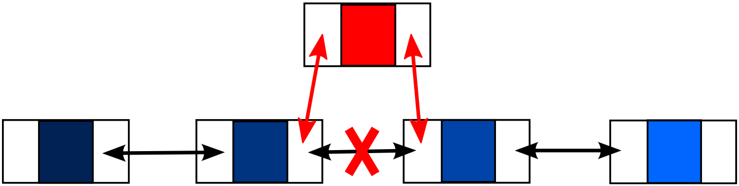
Chaque maillon connait le maillon précedent et le maillon suivant.



La liste : insérer un élément

Idée

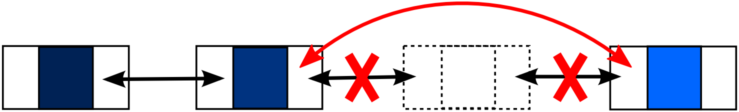
Il suffit de modifier les indices des maillons précédents et suivants.



La liste : supprimer un élément

Idée

On lie simplement les maillons précédents et suivants afin d'éviter le maillon à supprimer.



La liste : implémentation

On crée un tableau de maillons, ces maillons pointes sur les différentes cases du tableau.

```
class chainon{
  public:
  int prev, next;
  double val;
};
```

La liste : implémentation - insertion

On veux insérer l'élément d juste après l'indice i. Il est placé dans le tableau à la case d'indice j.

```
t[j].val = d; //assignation de la valeur
t[j].prev = i;
t[j].next = t[i].next;
t[i].next = j;
if(t[j].next != -1){
  t[t[j].next].prev = j;
}
```

La liste : implémentation - suppression

```
if(t[i].prev != -1){
  t[t[i].prev].next = t[i].next;
}
if(t[i].next !=-1){
  t[t[i].next].prev = t[i].prev;
}
```

La liste : implémentation réelle

Pointeurs

En pratique les champs `next` et `prev`, sont des adresses mémoires (des pointeurs).

STL

C'est la classe `std::list` (`#include <list>`).

Les itérateurs

Les itérateurs sont des éléments de la STL, qui permettent de parcourir les structures comme les listes, les piles, les files, les vecteurs ...
Ainsi si une pile ne donne accès qu'au premier élément, on peut quand même parcourir tout les éléments.

```
vector<double>::iterator it = vect.begin();
vector<double>::const_iterator it2 = vect.begin();
for(; it != vect.end(); it++){
  *it = 10;
}
for(; it2 != vect.end(); it2++){
  cout << *it2 << endl;
}
```

Récapitulatif des complexités

	vecteur	pile	file	liste
push_back	$O(1)$	$O(1)$	$O(1)$	$O(1)$
pop_back	$O(1)$	$O(1)$	-	$O(1)$
push_front	$O(N)$	-	-	$O(1)$
pop_front	$O(N)$	-	$O(1)$	$O(1)$
tab[i]	$O(1)$	-	-	$O(N)$
insert	$O(N)$	-	-	$O(1)$
erase	$O(N)$	-	-	$O(1)$

Autres structures classiques

- `set` : un ensemble dans lequel un élément ne peut-être présent qu'une fois
- `map` : associe à un élément une clé qui permet de le retrouver rapidement
- `hashmap` : une table de hâchage, similaire à une `map`, mais les élément sont indexés avec une fonction de hâchage.
- La file de priorité : les éléments sortent de la file en fonction de leur priorité.
- Les graphes : généralise est listes chaînées : réseau, arbres ...