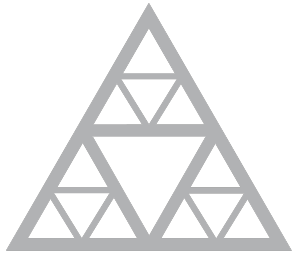
Algorithmique et structures de données

Complexité et structures de données

Nicolas Audebert

Mercredi 30 janvier 2019



École des Ponts
ParisTech

Ressources, polycopié, TD, email

Ressources du cours

- Site du cours : `http://imagine.enpc.fr/~monasse/Algo`
- Slides : `https://nicolas.audebert.at/teaching/ENPC/`
- Email : `nicolas.audebert@onera.fr`

Plan de la séance

Pourquoi ce cours ?

Introduction à la programmation C++

Apprendre à manipuler le C++ comme outil.

- Savoir programmer.
- Concevoir un logiciel.
- Tester et compiler.

Algorithmique

Apprendre l'informatique comme **discipline scientifique**.

- Qu'est-ce qu'un **algorithme** ? Un **bon** algorithme ?
- Comment **évaluer** et comparer différents algorithmes ?
- Quels sont les algorithmes classiques pour mon problème ?

Pourquoi ce cours ?

Introduction à la programmation C++

Apprendre à manipuler le C++ comme outil.

- Savoir programmer.
- Concevoir un logiciel.
- Tester et compiler.

Algorithmique

Apprendre l'informatique comme **discipline scientifique**.

- Qu'est-ce qu'un **algorithme** ? Un **bon** algorithme ?
- Comment **évaluer** et comparer différents algorithmes ?
- Quels sont les algorithmes classiques pour mon problème ?

Algorithme

Qu'est-ce qu'un algorithme ?

Un algorithme est une procédure comportant une **suite finie d'opérations** permettant d'obtenir un **résultat** à partir d'**entrées** connues.

Propriétés d'un algorithme

Un algorithme est une procédure **répétable** (par un humain) :

- finie (réalisable en temps borné),
- non-ambigüe (bien définie),
- travaillant sur des entrées spécifiées,
- éventuellement produisant des sorties.

Thèse de Church

Ces règles suffisent à formaliser correctement la calculabilité.

Algorithme

Qu'est-ce qu'un algorithme ?

Un algorithme est une procédure comportant une **suite finie d'opérations** permettant d'obtenir un **résultat** à partir d'**entrées** connues.

Propriétés d'un algorithme

Un algorithme est une procédure **répétable** (par un humain) :

- finie (réalisable en temps borné),
- non-ambigüe (bien définie),
- travaillant sur des entrées spécifiées,
- éventuellement produisant des sorties.

Thèse de Church

Ces règles suffisent à formaliser correctement la calculabilité.

Algorithme

Qu'est-ce qu'un algorithme ?

Un algorithme est une procédure comportant une **suite finie d'opérations** permettant d'obtenir un **résultat** à partir d'**entrées** connues.

Propriétés d'un algorithme

Un algorithme est une procédure **répétable** (par un humain) :

- finie (réalisable en temps borné),
- non-ambigüe (bien définie),
- travaillant sur des entrées spécifiées,
- éventuellement produisant des sorties.

Thèse de Church

Ces règles suffisent à formaliser correctement la calculabilité.

Algorithme

Qu'est-ce qu'un algorithme ?

Un algorithme est une procédure comportant une **suite finie d'opérations** permettant d'obtenir un **résultat** à partir d'**entrées** connues.

Propriétés d'un algorithme

Un algorithme est une procédure **répétable** (par un humain) :

- finie (réalisable en temps borné),
- non-ambigüe (bien définie),
- travaillant sur des entrées spécifiées,
- éventuellement produisant des sorties.

Thèse de Church

Ces règles suffisent à formaliser correctement la calculabilité.

Algorithme

Qu'est-ce qu'un algorithme ?

Un algorithme est une procédure comportant une **suite finie d'opérations** permettant d'obtenir un **résultat** à partir d'**entrées** connues.

Propriétés d'un algorithme

Un algorithme est une procédure **répétable** (par un humain) :

- finie (réalisable en temps borné),
- non-ambigüe (bien définie),
- travaillant sur des entrées spécifiées,
- éventuellement produisant des sorties.

Thèse de Church

Ces règles suffisent à formaliser correctement la calculabilité.

Algorithme

Qu'est-ce qu'un algorithme ?

Un algorithme est une procédure comportant une **suite finie d'opérations** permettant d'obtenir un **résultat** à partir d'**entrées** connues.

Propriétés d'un algorithme

Un algorithme est une procédure **répétable** (par un humain) :

- finie (réalisable en temps borné),
- non-ambigüe (bien définie),
- travaillant sur des entrées spécifiées,
- éventuellement produisant des sorties.

Thèse de Church

Ces règles suffisent à formaliser correctement la calculabilité.

Algorithme

Qu'est-ce qu'un algorithme ?

Un algorithme est une procédure comportant une **suite finie d'opérations** permettant d'obtenir un **résultat** à partir d'**entrées** connues.

Propriétés d'un algorithme

Un algorithme est une procédure **répétable** (par un humain) :

- finie (réalisable en temps borné),
- non-ambigüe (bien définie),
- travaillant sur des entrées spécifiées,
- éventuellement produisant des sorties.

Thèse de Church

Ces règles suffisent à formaliser correctement la calculabilité.

Plan de la séance

Notion de complexité

Définition

La **complexité** d'un algorithme est une estimation du **nombre d'éléments atomiques** nécessaires à son exécution en fonction des **paramètres caractéristiques** du problème.

Remarque

La complexité s'intéresse le plus souvent au **comportement asymptotique** de l'algorithme (lorsque les dimensions du problème deviennent très grandes).

Notion de complexité

Les types de complexité

- La **complexité en temps** : le nombre d'opérations élémentaires constituant l'algorithme.
- La **complexité en espace** : le nombre de cases mémoires élémentaires occupées lors du déroulement de l'algorithme.

Remarque

On s'intéresse généralement à la complexité dans le pire des cas et, plus rarement, à la complexité en moyenne.

Notion de complexité

Remarques

- La complexité en espace et en temps sont complémentaires :
  - Stocker tous les résultats (évite le recalcul, nécessite beaucoup de mémoire)
  - Calculer à chaque besoin (plus de calculs redondants, peu de mémoire nécessaire)
- En général, la complexité en espace n'est pas un problème et c'est le temps qui importe (par exemple, pour des applications temps réel).

Histogramme d'une image $W \times H$: version rapide

```
// On stocke les 256 valeurs de l'histogramme
int histo[256];
// Initialisation à 0
for(int i=0; i<256; i++){
  histo[i] = 0;
}
// Parcours de l'image par colonne
for(int x=0; x<image.width(); x++){
  for(int y=0; y<image.height(); y++){
    histo[image(x,y)]++;
  }
}
// Affichage de l'histogramme
for(int i=0; i<256; i++){
  drawRect(i,0,1,histo[i])
}
```

Histogramme d'une image $W \times H$: version rapide

```
// On stocke les 256 valeurs de l'histogramme
int histo[256];
// Initialisation à 0
for(int i=0; i<256; i++){
  histo[i] = 0;
}
// Parcours de l'image par colonne
for(int x=0; x<image.width(); x++){
  for(int y=0; y<image.height(); y++){
    histo[image(x,y)]++;
  }
}
// Affichage de l'histogramme
for(int i=0; i<256; i++){
  drawRect(i,0,1,histo[i])
}
```

Analyse

Mémoire :

1 tableau de 256 cases

Temps :

$W \times H$ pixels visités

Histogramme d'une image $W \times H$: version lente

```
for(int i=0; i<256; i++){
  // Valeur i de l'histogramme
  int h = 0;
  // Parcours de l'image
  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);
}
```

Histogramme d'une image $W \times H$: version lente

```
for(int i=0; i<256; i++){
  // Valeur i de l'histogramme
  int h = 0;
  // Parcours de l'image
  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 pixel

$\rightarrow 256 \times W \times H$

Bornes de complexité

Le nombre d'opérations élémentaires constituant un algorithme est souvent complexe à déterminer. Pour plus de commodité, on cherche un encadrement de celui-ci, i.e. f telle que :

$$\exists k_1, k_2,\ k_1 \times f(N) < \text{complexité} < k_2 \times f(N) \tag{1}$$

Notation

On utilise la notation de Landau : $O(f(n_1, n_2, n_3, \ldots))$
où $N = \{n_1, n_2, n_3, \ldots\}$ sont les paramètres caractéristiques du problème.

Exemple de mesure de complexité

Histogramme rapide

- Espace : $O(nombre\_couleurs)$
- Temps : $O(nombre\_pixels)$

Histogramme lent

- Espace : $O(1)$
- Temps : $O(nombre\_pixels * nombre\_couleurs)$

Notion de complexité : exemples simples

Parcours des éléments d'un tableau

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

on accède une fois à chaque case

Notion de complexité : exemples simples

Parcours des éléments d'un tableau

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

on accède une fois à chaque case $\rightarrow$ **complexité** $O(n)$

Notion de complexité : exemples simples

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

```
// Unicité 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;
    }
  }
}
```

deux parcours imbriqués

Notion de complexité : exemples simples

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

```
// Unicité 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;
    }
  }
}
```

deux parcours imbriqués $\rightarrow$ complexité $O(n^2)$

Notion de complexité : exemples simples

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

```
// Unicité 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;
    }
  }
}
```

deux parcours imbriqués $\rightarrow$ complexité $O(n^2)$

On commence par estimer la complexité des instructions dans la boucle la plus profonde, puis on remonte.

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

```
// Calcul à la volée
int s = 0;
for(int i=0; i<n; i++){
    s+= i;
}
```

Peu de temps

```
// Pré-calcul
vector<int> pre(1000000000,0);
for(int i=1; i<pre.size(); i++){
  pre[i] = i + pre[i-1];
}
// Utilisation
int s = pre[n];
```

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

```
// Calcul à la volée
int s = 0;
for(int i=0; i<n; i++){
    s+= i;
}
```

Complexité $O(n)$.

Peu de temps

```
// Pré-calcul
vector<int> pre(1000000000,0);
for(int i=1; i<pre.size(); i++){
  pre[i] = i + pre[i-1];
}
// Utilisation
int s = pre[n];
```

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

```
// Calcul à la volée
int s = 0;
for(int i=0; i<n; i++){
    s+= i;
}
```

Complexité $O(n)$.

Peu de temps

```
// Pré-calcul
vector<int> pre(1000000000,0);
for(int i=1; i<pre.size(); i++){
  pre[i] = i + pre[i-1];
}
// Utilisation
int s = pre[n];
```

Complexité $O(1)$ (en utilisation).

Ordres de grandeur de complexité (1/2)

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

Ordres de grandeur de complexité (2/2)

- $O(n \log(n))$: **linéarithmique**, de nombreux algorithmes “rapides”.
  Exemple : Tri optimal, Fast Fourier Transform
- $O(n^k)$: **polynomiale**, acceptable pour des données petites (faible n) et des puissances faibles (petit k).
  Exemple : $O(n^2)$ tri naïf
- $O(2^n)$: **exponentielle**, utilisable en pratique seulement pour des petites dimensions.
- $O(n!)$: **factorielle**, inutilisable dès que n dépasse la dizaine.

Temps de calcul relatifs

En supposant qu'une opération élémentaire prend 10 ns, pour $n = 50$:

- $O(1)$: 10 ns
- $O(\log(n))$: 20 ns
- $O(n)$: 500 ns
- $O(n \log(n))$: 850 ns
- $O(n^2)$: 25 µs
- $O(2^n)$: 130 jours ($\simeq$ 4 mois)
- $O(n!)$: 10^{48} ans

Temps de calcul relatifs

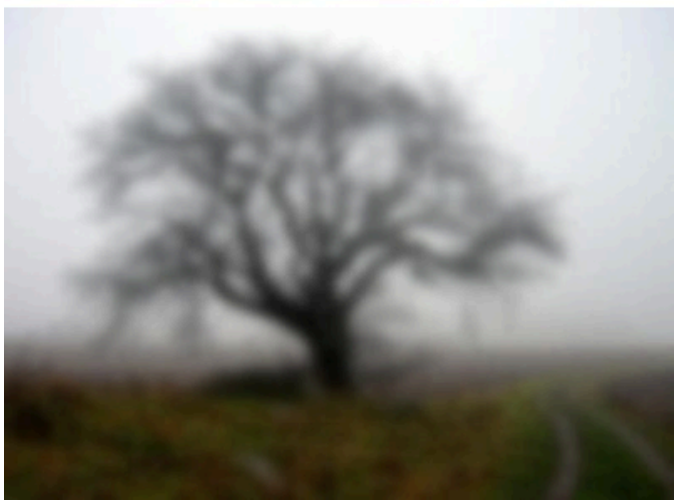
En supposant qu'une opération élémentaire prend 10 ns, pour $n = 50$:

- $O(1)$: 10 ns
- $O(\log(n))$: 20 ns
- $O(n)$: 500 ns
- $O(n \log(n))$: 850 ns
- $O(n^2)$: 25 µs
- $O(2^n)$: 130 jours ($\simeq$ 4 mois)
- $O(n!)$: 10^{48} ans $>>>$ $14e^9$ années (âge de l'univers)

Exemple : flouter une image

Floutage

```
// Flouter une image W × H sur un rayon r
for(int i=r/2; i<W-r/2; i++){
  for(int j=r/2; j<H-r/2; j++){
    newIm[i,j] = 0;
    for(int k=i-r/2; k<i+r/2; k++){
      for(int m=j-r/2; m<j+r/2; m++){
        newIm[i,j] += im(k,m);
      }
    }
    newIm[i,j]/= r*r;
  }
}
```

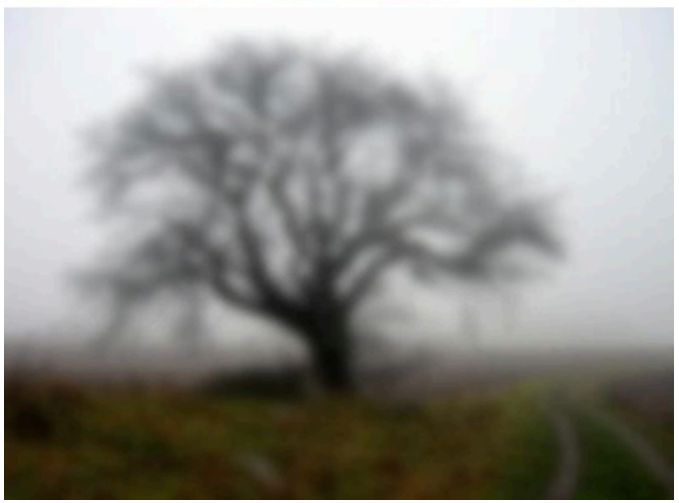


Exemple : flouter une image

Floutage

```
// Flouter une image W × H sur un rayon r
for(int i=r/2; i<W-r/2; i++){
  for(int j=r/2; j<H-r/2; j++){
    newIm[i,j] = 0;
    for(int k=i-r/2; k<i+r/2; k++){
      for(int m=j-r/2; m<j+r/2; m++){
        newIm[i,j] += im(k,m);
      }
    }
    newIm[i,j]/= r*r;
  }
}
```

$\rightarrow$ complexité $O(W \times H \times r^2)$

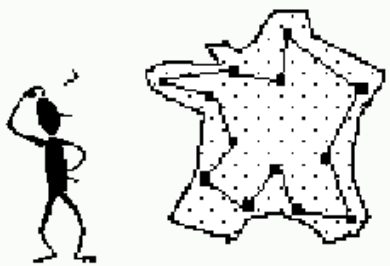


Classes de complexité

P versus NP

Les problèmes de classe P sont les problèmes que l'on peut résoudre en **temps polynomial** ($O(n^p)$ avec p fixe).
Les problèmes de classe NP sont ceux pour lesquels on peut seulement **vérifier** la solution en temps polynomial.

Example de problème NP



Un commercial doit parcourir $N = \{n_1, \ldots, n_k\}$ villes séparées par les distances $d_{i,j}$. Quelle est le chemin qui minimise la distance totale parcourue ?

Question à 1 000 000$

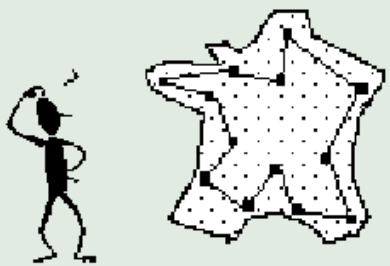
$P = NP$ ou $P \neq NP$? (problème du millénaire de l'Institut Clay)

Classes de complexité

P versus *NP*

Les problèmes de classe *P* sont les problèmes que l'on peut résoudre en **temps polynomial** ($O(n^p)$ avec p fixe).
Les problèmes de classe *NP* sont ceux pour lesquels on peut seulement **vérifier** la solution en temps polynomial.

Example de problème *NP*



Un commercial doit parcourir $N = \{n_1, \ldots, n_k\}$ villes séparées par les distances $d_{i,j}$. Quelle est le chemin qui minimise la distance totale parcourue ?

Question à 1 000 000$

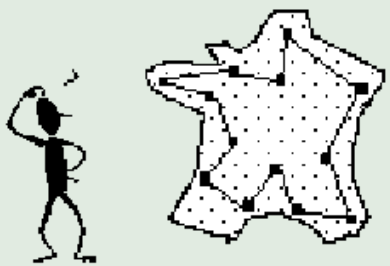
$P = NP$ ou $P \neq NP$? (problème du millénaire de l'Institut Clay)

Classes de complexité

P versus NP

Les problèmes de classe P sont les problèmes que l'on peut résoudre en **temps polynomial** ($O(n^p)$ avec p fixe).
Les problèmes de classe NP sont ceux pour lesquels on peut seulement **vérifier** la solution en temps polynomial.

Example de problème NP



Un commercial doit parcourir $N = \{n_1, \ldots, n_k\}$ villes séparées par les distances $d_{i,j}$. Quelle est le chemin qui minimise la distance totale parcourue ?

Approche naïve : $O(n!)$, *meilleur algorithme exact connu :* $O(n^2 2^n)$.

Question à 1 000 000$

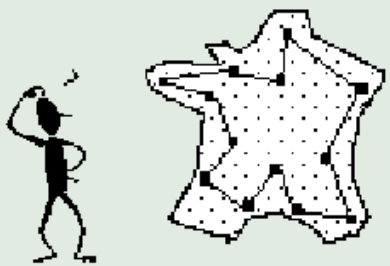
$P = NP$ ou $P \neq NP$? (problème du millénaire de l'Institut Clay)

Classes de complexité

P versus *NP*

Les problèmes de classe *P* sont les problèmes que l'on peut résoudre en **temps polynomial** ($O(n^p)$ avec p fixe).
Les problèmes de classe *NP* sont ceux pour lesquels on peut seulement **vérifier** la solution en temps polynomial.

Example de problème *NP*



Un commercial doit parcourir $N = \{n_1, \ldots, n_k\}$ villes séparées par les distances $d_{i,j}$. Quelle est le chemin qui minimise la distance totale parcourue ?

Approche naïve : $O(n!)$, *meilleur algorithme exact connu :* $O(n^2 2^n)$.

Question à 1 000 000$

$P = NP$ ou $P \neq NP$? (problème du millénaire de l'Institut Clay)

En pratique...

Relativisons !

- Les complexités sont asymptotiques : elles sont valables pour les grandes tailles de données.
- On omet généralement les constantes multiplicatives dans la notation $O()$.

Exemple

- Algorithme A : $10^6 \times n = O(n)$
- Algorithme B : $n^2 = O(n^2)$

$\rightarrow$ L'algorithme A est plus rapide ssi $n > 10^6$

Complexité en moyenne et dans le pire des cas

- **Complexité moyenne** : caractérise le comportement attendu pour des répétitions sur des données aléatoires.
- **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, traitement d'image.
- **Complexité dans le pire des cas** : applications critiques (aéronautique, applications temps-réel...).

Plan de la séance

Les tableaux en C++

En C++ les tableaux sont de taille fixe. Cette taille peut-être déterminée soit :

- à la compilation (tableau **statique**),
- à l'exécution (tableau **dynamique**)

Les tableaux sont des structures difficiles à manipuler (gestion de la mémoire, mécanisme de copie inexistant, ...).

Pour simplifier

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

Plan de la séance

Le vecteur

La classe

La classe est `std::vector` (simplement `vector` si on a utilisé `using namespace std`).

Elle est « templatée », elle peut être utilisée pour contenir n'importe quel type de variable.

Avantages

- Pas de mémoire à gérer
- Taille du vecteur connue grâce à la méthode `size()`
- La taille n'a pas besoin d'être connue dès le départ

Complexité de l'ajout d'un élément

Implémentation naïve

Quand on ajoute un élément avec la fonction `push_back()`, 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 de l'ajout d'un élément

Implémentation naïve

Quand on ajoute un élément avec la fonction `push_back()`, 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 de l'ajout d'un élément

Implémentation astucieuse

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

Complexité

$O(1)$ (en moyenne, $O(n)$ quand la taille maximale est atteinte)

Complexité de l'ajout d'un élément

Implémentation astucieuse

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

Complexité

$O(1)$ (en moyenne, $O(n)$ quand la taille maximale est atteinte)

Complexité moyenne du `push_back` : preuve

Supposons que l'on effectue n ajouts par `push_back` .

Soit k le nombre de redimensionnements. La taille du vecteur étant multipliée par m à chaque fois, on a :

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

Le nombre total de recopies est donc :

$$\sum_{p=1}^{k} m^p = \sum_{p=1}^{log_m(n)} m^p = m\frac{m^{log_m(n)} - 1}{m-1} = \frac{m \times n}{m-1}$$

Le coût moyen (nombre de recopies / nombre d'ajouts) est alors :

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

Complexité moyenne du `push_back` : preuve

Supposons que l'on effectue n ajouts par `push_back` .

Soit k le nombre de redimensionnements. La taille du vecteur étant multipliée par m à chaque fois, on a :

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

Le nombre total de recopies est donc :

$$\sum_{p=1}^{k} m^p = \sum_{p=1}^{log_m(n)} m^p = m\frac{m^{log_m(n)} - 1}{m-1} = \frac{m \times n}{m-1}$$

Le coût moyen (nombre de recopies / nombre d'ajouts) est alors :

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

Complexité moyenne du `push_back` : preuve

Supposons que l'on effectue n ajouts par `push_back`.

Soit k le nombre de redimensionnements. La taille du vecteur étant multipliée par m à chaque fois, on a :

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

Le nombre total de recopies est donc :

$$\sum_{p=1}^{k} m^p = \sum_{p=1}^{log_m(n)} m^p = m\frac{m^{log_m(n)} - 1}{m-1} = \frac{m \times n}{m-1}$$

Le coût moyen (nombre de recopies / nombre d'ajouts) est alors :

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

Complexité moyenne du `push_back` : preuve

Supposons que l'on effectue n ajouts par `push_back`.

Soit k le nombre de redimensionnements. La taille du vecteur étant multipliée par m à chaque fois, on a :

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

Le nombre total de recopies est donc :

$$\sum_{p=1}^{k} m^p = \sum_{p=1}^{log_m(n)} m^p = m\frac{m^{log_m(n)} - 1}{m-1} = \frac{m \times n}{m-1}$$

Le coût moyen (nombre de recopies / nombre d'ajouts) est alors :

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

Propriétés des vecteurs

Complexités des opérations sur les vecteurs

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

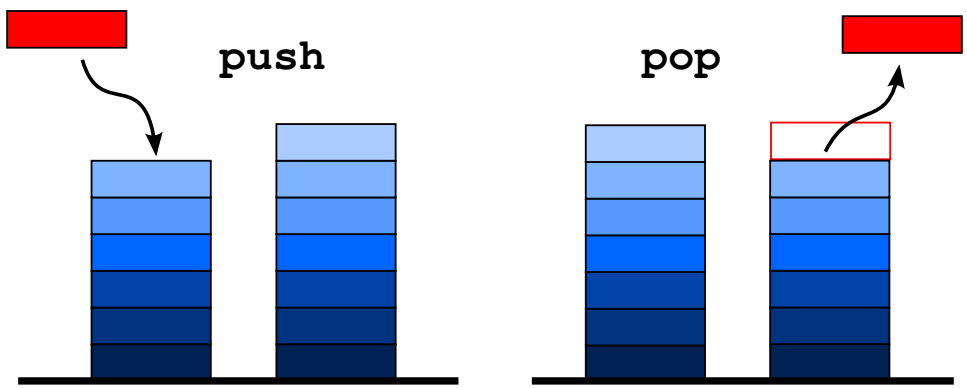
La pile : *Last In First Out* (LIFO)

Principe

On ajoute et on retire les éléments un par un par le dessus.

Implémentations

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



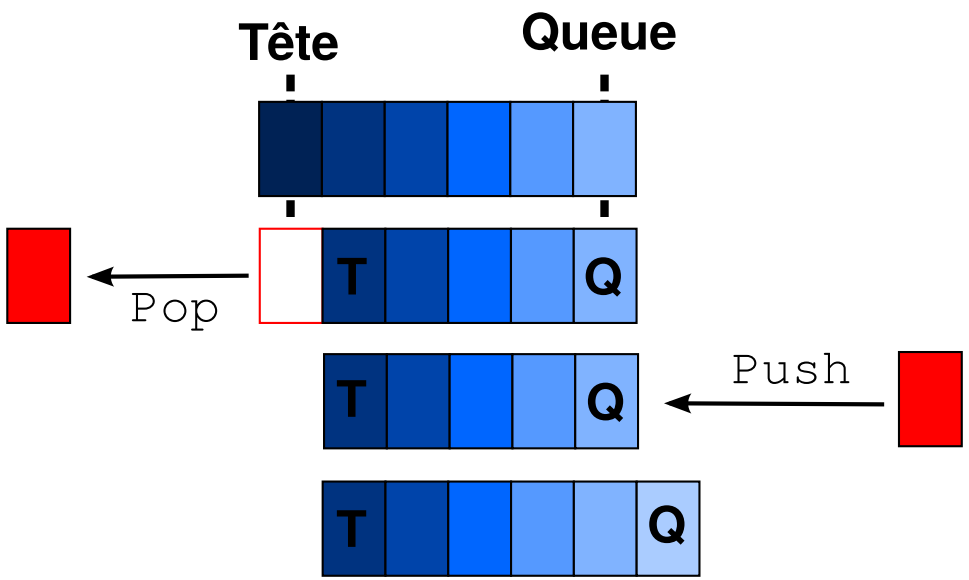
La file : *First In First Out* (FIFO)

Principe

On ajoute les éléments à l'arrière et on retire les éléments par l'avant.

Implémentations

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



File : implémentation

```
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();
};

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

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

// Tête de la file
double File::front() const{
  return v[debut];
}

// Arithmétique modulo la
// taille de la file
int File::modulo(int i){
  if(v.capacity()==0)
    return 0;
  return i%v.capacity();
}
```

File : implémentation

```
// Ajout d'un élément
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;
}
```

```
// Retrait de l'élément
// de tête
double File::pop(){
    double d = front();
    debut =
     ↪  modulo(debut+1);
    return d;
}
```

File : propriétés

Complexité

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

Dans la STL

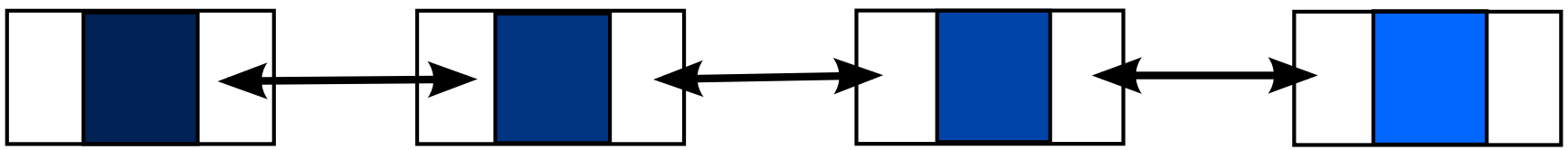
- `queue` (`#include <queue>`) : file
- `deque` (`#include <deque>`) : *double ended queue*

La liste chaînée

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

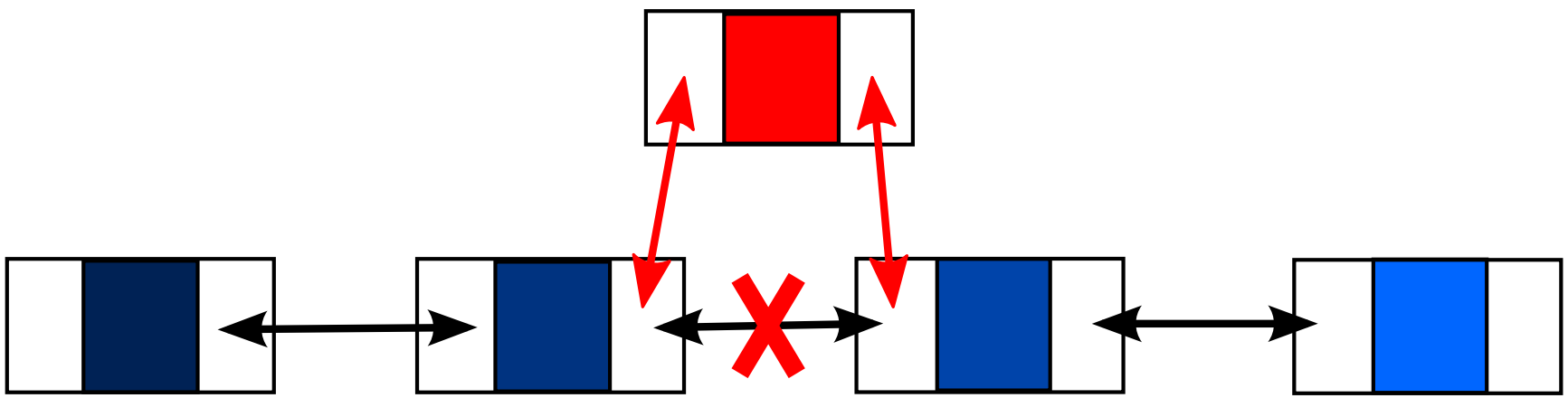
Chaque maillon connait le maillon précédent et le maillon suivant.



La liste : insérer un élément

Idée

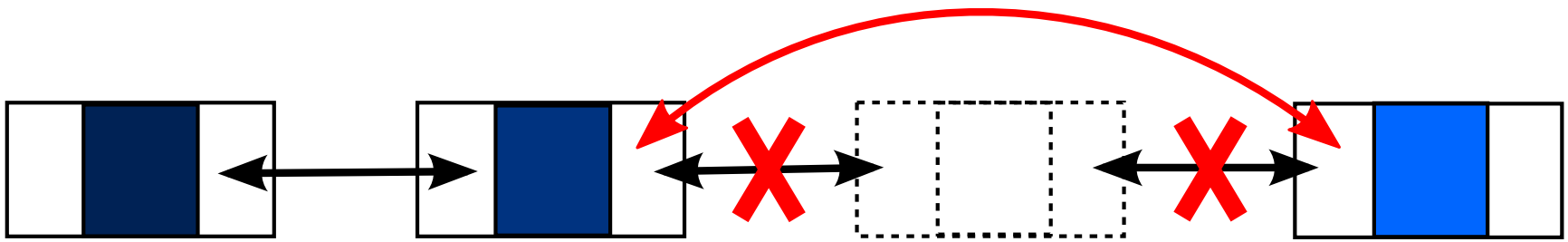
Il suffit de modifier les indices des maillons précédents et suivants.



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 pointent sur les différentes cases du tableau.

```
class Chainon{
  public:
    int prev, next;
    double value;
};
```

La liste : implémentation - insertion

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

```
t[j].val = elem; // assignation de elem dans la liste
t[j].prev = i;
// t[j] est maintenant chaîné au successeur de t[i]
t[j].next = t[i].next;
// le nouveau successeur de t[i] est désormais t[j]
t[i].next = j;
// Par convention, -1 signifie que l'élément suivant n'existe pas
// (fin de la liste)
if(t[j].next != -1){
  t[t[j].next].prev = j;
}
```

La liste : implémentation - suppression

```
// On chaîne le prédécesseur s'il existe
if(t[i].prev != -1){
  t[t[i].prev].next = t[i].next;
}
// On chaîne le successeur s'il existe
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, c'est-à-dire des pointeurs sur les chaînons.

STL

Implémentation standard : 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 ($\simeq$ dictionnaires Python)
- `hashmap` : similaire à une `map`, mais les élément sont indexés avec une fonction de hachage pour accélérer les temps d'accès.
- La file de priorité : files dont les éléments sortent par ordre de priorité.
- Les graphes : généralisation des listes chaînées (réseaux, arbres de dépendances, ...)