Algorithmique et structures de données

Complexité et structures de données

Julien Hauret

Lundi 31 janvier 2022

Accès aux ressources du cours

Liens utiles

- Site du cours : *http://imagine.enpc.fr/~monasse/Algo*
- Slides : *https://jhauret.github.io/teaching/*
- Email : *julien.hauret@lecnam.net*

Plan de la séance

Qu'est-ce que l'algorithmique ?

Complexité

- Notion de complexité

Rappels sur les tableaux

Structures de données

- Vecteurs
- Piles, files et listes
- Les itérateurs
- Récapitulaif
- Autres structures

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 ?

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 (1935)

Ces règles suffisent à formaliser correctement la calculabilité.

Algorithme

Qu'est-ce qu'un algorithme ?

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 (1935)

Ces règles suffisent à formaliser correctement la calculabilité.

Algorithme

Qu'est-ce qu'un algorithme ?

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 (1935)

Ces règles suffisent à formaliser correctement la calculabilité.

Algorithme

Qu'est-ce qu'un algorithme ?

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 (1935)

Ces règles suffisent à formaliser correctement la calculabilité.

Algorithme

Qu'est-ce qu'un algorithme ?

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 (1935)

Ces règles suffisent à formaliser correctement la calculabilité.

Algorithme

Qu'est-ce qu'un algorithme ?

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 (1935)

Ces règles suffisent à formaliser correctement la calculabilité.

Algorithme

Qu'est-ce qu'un algorithme ?

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 (1935)

Ces règles suffisent à formaliser correctement la calculabilité.

Plan de la séance

Qu’est-ce que l’algorithmique ?

Complexité

- Notion de complexité

Rappels sur les tableaux

Structures de données

- Vecteurs
- Piles, files et listes
- Les itérateurs
- Récapitulaif
- Autres structures

Notion de complexité

Définition

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

Opération atomique

Opération de base qui prend toujours le même temps à s'exécuter : multiplication, addition, accès à une case d'un tableau, etc.

Remarque

La complexité représente le **comportement asymptotique** de l'algorithme (lorsque les dimensions des entrées 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 $\Longrightarrow$ évite le recalcul, nécessite beaucoup de mémoire.
  - calculer à la demande $\Longrightarrow$ 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])
}
```

Opération de base : accès à une case d'un tableau.

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])
}
```

Complexités

Mémoire :

1 tableau de 256 cases

Temps :

$W \times H$ pixels visités

Opération de base : accès à une case d'un tableau.

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);
}
```

Complexités

Mémoire :

Pas de tableau

Temps :

256 passages sur chaque pixel

$\rightarrow 256 \times W \times H$

Mesure de complexité

Bornes de complexité

Le nombre exact d'opérations élémentaires constituant un algorithme peut-être complexe à déterminer. Pour plus de commodité, on cherche une fonction f qui encadre celui-ci, c'est-à-dire :

$$\exists\ \alpha, \beta \in \mathbb{R}\ /\quad \alpha \cdot f(N) < \text{complexité} < \beta \cdot f(N) \quad \text{pour} \quad N \to \infty \tag{1}$$

Notation

Par convention, on utilise la notation de Landau. On dit alors que l'algorithme est en $O(f(N))$ avec $N = \{n_1, n_2, n_3, \ldots\}$ sont les paramètres caractéristiques du problème.

Exemple de mesure de complexité

Calcul de l'histogramme d'une image :

- nombre de pixels $N = W \times H$;
- nombre de couleurs c.

Histogramme rapide

- Espace : $O(c)$
- Temps : $O(W \times H)$

Histogramme lent

- Espace : $O(1)$
- Temps : $O(W \times H \times c)$

Notion de complexité : exemples simples

Opération de base : accès à une case d'un tableau.

Paramètre caractéristique : n la longueur du tableau.

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;
}
```

Chaque case est accédée une et une seule fois .

Notion de complexité : exemples simples

Opération de base : accès à une case d'un tableau.

Paramètre caractéristique : n la longueur du tableau.

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;
}
```

Chaque case est accédée une et une seule fois → **complexité** $O(n)$.

Notion de complexité : exemples simples

Opération de base : accès à une case d'un tableau.

Paramètre caractéristique : n la longueur du tableau.

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 du tableau imbriqués .

Notion de complexité : exemples simples

Opération de base : accès à une case d'un tableau.

Paramètre caractéristique : n la longueur du tableau.

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 du tableau imbriqués → **complexité** $O(n^2)$.

Exemple : le n^e terme de la suite de Fibonacci

Différentes implémentations d'un même calcul peuvent avoir des complexités différentes !

Peu de mémoire

```
// Calcul à la volée
int fibonacci(int n){
    int prec = 1, int p_prec = 0;
    int resultat;
    for(int i=0; i<n; i++){
        resultat = prec + p_prec;
        p_prec = prec;
        prec = resultat;
    }
    return resultat;
}
```

Peu de temps

```
// Pré-calcul
vector<int> fibo(10000000,0);
fibo[1] = 1;
for(int i=2; i<fibo.size(); i++){
    fibo[i] = fibo[i-1] +
                          fibo[i-2];
}
// Utilisation
int fibonacci(int n){
    return fibo[n];
}
```

Exemple : le n^e terme de la suite de Fibonacci

Différentes implémentations d'un même calcul peuvent avoir des complexités différentes !

Peu de mémoire

```
// Calcul à la volée
int fibonacci(int n){
    int prec = 1, int p_prec = 0;
    int resultat;
    for(int i=0; i<n; i++){
        resultat = prec + p_prec;
        p_prec = prec;
        prec = resultat;
    }
    return resultat;
}
```

Complexité $O(n)$.

Peu de temps

```
// Pré-calcul
vector<int> fibo(10000000,0);
fibo[1] = 1;
for(int i=2; i<fibo.size(); i++){
    fibo[i] = fibo[i-1] +
                          fibo[i-2];
}
// Utilisation
int fibonacci(int n){
    return fibo[n];
}
```

Exemple : le n^e terme de la suite de Fibonacci

Différentes implémentations d'un même calcul peuvent avoir des complexités différentes !

Peu de mémoire

```
// Calcul à la volée
int fibonacci(int n){
    int prec = 1, int p_prec = 0;
    int resultat;
    for(int i=0; i<n; i++){
        resultat = prec + p_prec;
        p_prec = prec;
        prec = resultat;
    }
    return resultat;
}
```

Complexité $O(n)$.

Peu de temps

```
// Pré-calcul
vector<int> fibo(10000000,0);
fibo[1] = 1;
for(int i=2; i<fibo.size(); i++){
    fibo[i] = fibo[i-1] +
                          fibo[i-2];
}
// Utilisation
int fibonacci(int n){
    return fibo[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, transformée de Fourier rapide.
- $O(n^k)$: **polynomiale**, acceptable pour des données petites (faible n) et des puissances faibles (petit k).
  Exemple : tri naïf ($O(n^2)$), multiplication matricielle ($O(n^3)$).
- $O(2^n)$: **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 $>>>$ $14e^9$ (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;
  }
}
```

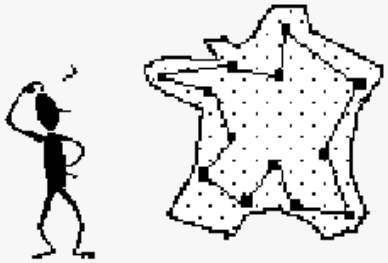
→ complexité $O(W \times H \times r^2)$

Classes de complexité

P versus *NP*

Les problèmes de classe *P* sont les problèmes pour lesquels un algorithme de résolution en **temps polynomial** est connu.
Les autres problèmes sont dits de classe *NP* si l'on connaît seulement un algorithme polynomial permettant de *vérifier* une solution et *NP*-difficile sinon.

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$

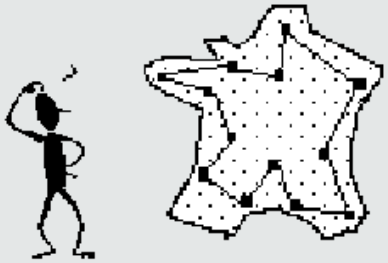
$P = NP$ ou $P \neq NP$? (problème du millénaire de l'Institut Clay)

Classes de complexité

P versus *NP*

Les problèmes de classe *P* sont les problèmes pour lesquels un algorithme de résolution en **temps polynomial** est connu.
Les autres problèmes sont dits de classe *NP* si l'on connaît seulement un algorithme polynomial permettant de *vérifier* une solution et *NP*-difficile sinon.

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$

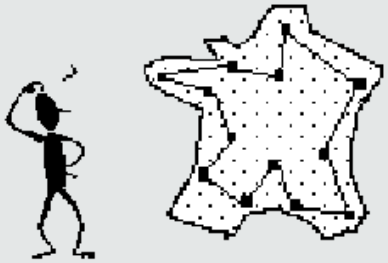
$P = NP$ ou $P \neq NP$? (problème du millénaire de l'Institut Clay)

Classes de complexité

P versus *NP*

Les problèmes de classe *P* sont les problèmes pour lesquels un algorithme de résolution en **temps polynomial** est connu.
Les autres problèmes sont dits de classe *NP* si l'on connaît seulement un algorithme polynomial permettant de *vérifier* une solution et *NP*-difficile sinon.

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$

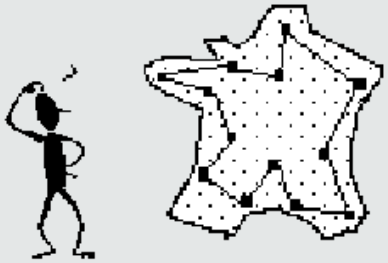
$P = NP$ ou $P \neq NP$? (problème du millénaire de l'Institut Clay)

Classes de complexité

P versus *NP*

Les problèmes de classe *P* sont les problèmes pour lesquels un algorithme de résolution en **temps polynomial** est connu.
Les autres problèmes sont dits de classe *NP* si l'on connaît seulement un algorithme polynomial permettant de *vérifier* une solution et *NP*-difficile sinon.

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 : $n^2 = O(n^2)$

→ 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

Qu’est-ce que l’algorithmique ?

Complexité

- Notion de complexité

Rappels sur les tableaux

Structures de données

- Vecteurs
- Piles, files et listes
- Les itérateurs
- Récapitulaif
- Autres structures

Les tableaux en C++

Les tableaux en C++ sont de taille fixe. Elle 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, pas de mécanisme de copie inexistant, etc.).

Pour simplifier

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

Plan de la séance

Qu'est-ce que l'algorithmique ?

Complexité

- Notion de complexité

Rappels sur les tableaux

Structures de données

- Vecteurs
- Piles, files et listes
- Les itérateurs
- Récapitulaif
- Autres structures

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 (redimensionnable !).

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 :

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

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 :

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

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 :

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

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 :

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

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)`) : $O(N)$

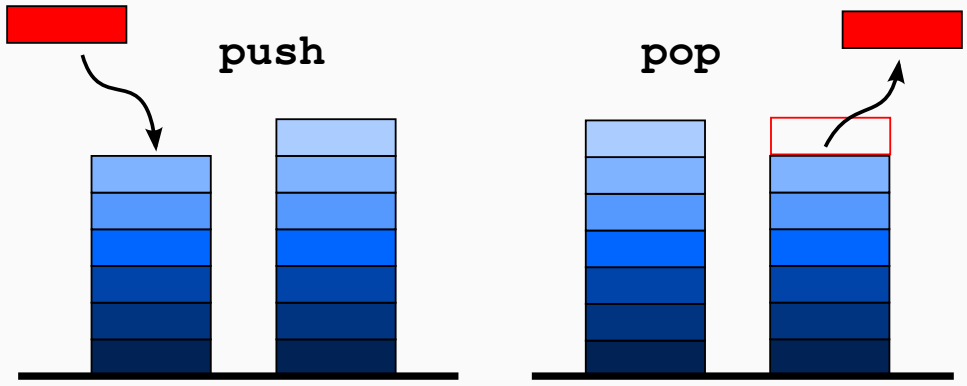
La pile : *Last In First Out* (LIFO)

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()`



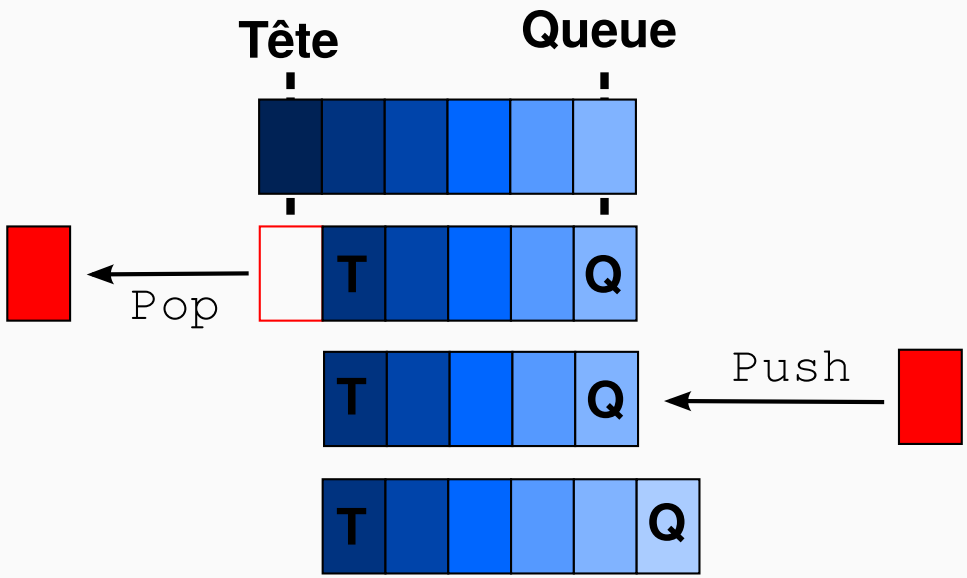
La file : *First In First Out* (FIFO)

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 : propriétés

Complexité

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

Dans la STL

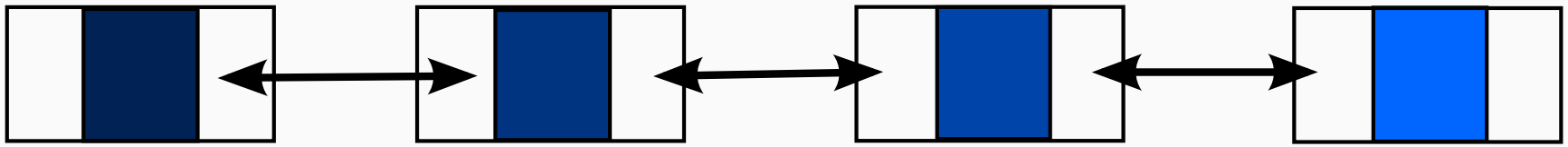
- `queue` (`#include <queue>`) : file
- `deque` (`#include <deque>`) : *double ended queue*

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

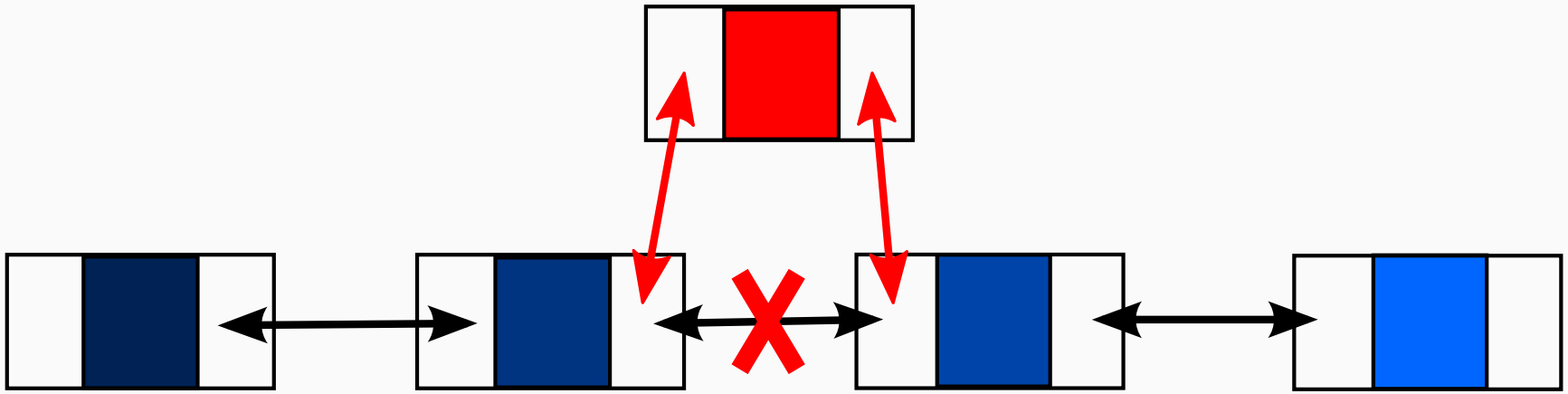
Chaque maillon connait le maillon précédent et le maillon suivant.



La liste : insérer un élément

Idée

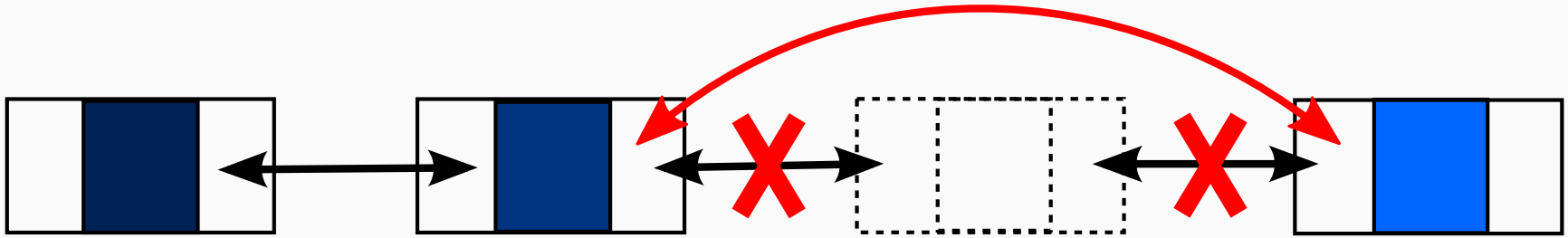
Il suffit de modifier les indices des maillons précédents et suivants.



La liste : supprimer un élément

Idée

Le maillon qui précède celui à supprimer est lié directement à son successeur.



La liste : implémentation

Une liste chaînée est un tableau de maillons (ou chainons). Chaque chainon connaît sa valeur, son précédesseur et son successeur.

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

La liste : implémentation - insertion

Pour insérer l'élément `elem` à l'indice i, on le place dans le tableau à la case d'indice j puis on modifie les chainons à l'indice $i-1$ pour le faire pointer sur son nouveau successeur.

```
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`* : ensemble dans lequel un élément est présent au plus une fois.
- *`map`* : associe à chaque élément une clé permettant de le retrouver rapidement (≃ dictionnaires Python)
- *`hashmap`* : similaire à une *`map`*, mais les élément sont indexés avec une fonction de hachage pour accélérer les temps d'accès.
- File de priorité : file dont les éléments sortent par ordre de priorité.
- Graphes : généralisation des listes chaînées (réseaux, arbres de dépendances…)