Langages, structures de données et complexité

Informatique scientifique pour le Calcul
Ecoles Doctorales 2012/2013

Vincent Miele

CNRS - Biométrie & Biologie Evolutive

Janvier 2013

Document disponible sur `http://lyoncalcul.univ-lyon1.fr`

Passer du papier à l'ordinateur

- quel(s) langage(s) ? [ce cours]
- quelles algorithmes ? adaptés au calcul parallèle ? [pas ce cours] comment quantifier les ressources qu'ils demandent ? [ce cours]
- quelles structures de données pour mon problème (indépendant du langage) ? la transcription de celles que j'utilise sur la papier ? [ce cours]

Passer du papier à l'ordinateur

Objectifs

- efficacité $\rightarrow$ bien comprendre les coûts de calcul, théoriques [ce cours] et coût machine [cours précédent également]
- robustesse $\rightarrow$ utiliser au maximum les standards, les librairies, s'appuyer sur la communauté d'utilisateurs
- maximum de cohérence avec les algorithmes du papier $\rightarrow$ lisibilité, encapsulation

Du pragmatisme, pas de dogmatisme

- la performance
- la lisibilité/maintenabilité/POO
- la disponibilité de librairies (“package”, “toolbox”, “module”, “library”)
- l'adaptabilité au calcul parallele

Dans ce cours, on se concentrera sur les langages les plus courants en Calcul :

C/C++, (Fortran), Python, (Perl), R, Matlab (Scilab)

Quelles différences entre ces langages
pour quelles avantages/inconvénients ?

On parle de *niveau* d'un langage en fonction de la nécessité imposée au programmeur de connaître le fonctionnement d'un ordinateur.

Le langage de très bas niveau est le langage machine binaire.

Plus le niveau est bas, plus les performances sont importantes.

Fortan et C/C++ sont considérés de niveau intermédiaire (permet la gestion fine de la mémoire par exemple, et donc de la performance).

Python ou Perl sont des langages de haut niveau, de même que Matlab/Scilab ou R (peu généralistes donc parfois appelés "environnements de programmation scientifique").

Un langage est dit *compilé* quand le *code* ou programme *source* sous forme de texte est tout d'abord lu et traité par un autre programme appelé *compilateur* qui le convertit en langage machine.

Le compilateur signale les erreurs syntaxiques présentes dans le code source.

Cette phase de compilation peut parfois être très longue.

La compilation (et *l'édition des liens*, voir cours suivants) produit un exécutable autonome qui ne fonctionne que sur le type de machine (OS, 32/64 bits) où la compilation s'est déroulée.

C++, Fortran sont des langages compilés (avec un choix de compilateurs gratuits ou non).

Un programme en langage *interprété* nécessite pour fonctionner un interprète(eur) qui est un autre programme qui va vérifier la syntaxe et traduire directement, au fur et à mesure de son exécution, le programme source en langage machine (un peu comme un interprète durant une interview).

Un programme interprété sera plus lent qu'un programme compilé du fait de la traduction dynamique. Quand une ligne du programme doit être exécutée un grand nombre de fois, l'interpréteur la traduit autant de fois qu'elle est exécutée.

Néanmoins la correction des erreurs sera plus simple car l'interprète signale à l'execution où se trouve l'erreur. Et le code source venant d'être écrit peut être directement testé.

NB : un programme dans un langage interprété est parfois appelé *script*.

Python, R, Matlab (pas si simple. . . , voir après) sont des langages interprétés.

	Included in Level			
Optimization	-O1	-O2	-Os	-O3
defer-pop	●	●	●	●
thread-jumps	●	●	●	●
branch-probabilities	●	●	●	●
cprop-registers	●	●	●	●
guess-branch-probability	●	●	●	●
omit-frame-pointer	●	●	●	●
align-loops	○	●	○	●
align-jumps	○	●	○	●
align-labels	○	●	○	●
align-functions	○	●	○	●
optimize-sibling-calls	○	●	●	●
cse-follow-jumps	○	●	●	●
cse-skip-blocks	○	●	●	●
gcse	○	●	●	●
expensive-optimizations	○	●	●	●
strength-reduce	○	●	●	●
rerun-cse-after-loop	○	●	●	●
rerun-loop-opt	○	●	●	●
caller-saves	○	●	●	●
force-mem	○	●	●	●
peephole2	○	●	●	●
regmove	○	●	●	●
strict-aliasing	○	●	●	●
delete-null-pointer-checks	○	●	●	●
reorder-blocks	○	●	●	●
schedule-insns	○	●	●	●
schedule-insns2	○	●	●	●
inline-functions	○	○	○	●
rename-registers	○	○	○	●

Le compilateur réalise (à la demande) des *optimisations* qui permettent de générer un code plus efficace. Toutefois, le compilateur “prend des risques” (il essaie de se mettre dans la tête du programmeur avec +/- de succès) et du temps.

- *loop unrolling* :

```
for (int i=0; i<n; i++)
{
  sum += data[i];
}
```

```
for (int i=0; i<n; i+=4)
{
  sum1 += data[i+0];
  sum2 += data[i+1];
  sum3 += data[i+2];
  sum4 += data[i+3];
}
sum = sum1 + sum2 + sum3 + sum4;
```

- *inlining* : Effective STL, Scott Meyers, item 46 C++ `sort` est plus + rapide que C `qsort` si on utilise un functor
- voir ci-contre. . .

Typologie des langages

Compilé vs interprété

```
#include <vector>
#include <iostream>
using namespace std;

class Matrix
{
private:
  int _s;
  vector<double> _vinternal;
public:
  Matrix(int s)
    : _vinternal(s*s, 0), _s(s) {}
  ~Matrix() {}
  inline double& element(int i, int j){
    return _vinternal[i*_s+j];
  }
};

int main(int argc, char ** argv )
{
  int s = 20000;

  Matrix m(s);
  // 10.30 secondes avec -O0
  // 1.69 secondes avec -O3
  for (int i=0; i<s; i++)
    for (int j=0; j<s; j++)
      m.element(i,j) = m.element(i,j)+m.element(i,j);
}
```

Les langages de typage fort dit *statique* imposent la déclaration précise de toutes les variables (type, signe, taille) et les éventuelles conversions doivent être explicites.

Rigidité mais sécurité.

Fortran, C++ (NB : mot clé `auto` en C++11).

Les langages non typés ou de typage *dynamique* sont très souples avec les variables : pas de déclaration et possibilité de changement de type à la volée.

Le temps de développement est réduit (moins *verbose*) mais des erreurs non detectables sont possibles (faute de frappe dans le nom de la variable).

La grande flexibilité que permet le typage dynamique se paye en général par une surconsommation de mémoire correspondant à l'encodage du type dans la valeur.

Python, R, Matlab

Langage *procédural* : enchainement de procédures/fonctions sur des données globales

- plusieurs fichiers, mais comment découper ?
- variables globales en début de programme, mais comment ne pas s'y perdre ? si même traitement sur 2 variables ?
- lisibilité de la liste de paramètres ?
  Exple : `fonction correction(nbanimaux, age, poids, taille, seuil, print, nbiterations, .....)`
- Difficile de modifier car tout est imbriqué, idem pour un tester une partie.Tout est accessible par tous (Couplage)
- Pbm de recopie de code (synchronisation !)
- Programme fragile, difficilement extensible

C, Fortran, Matlab ou R sont des langages procéduraux (malgré des évolutions)

Langage *orienté objet* : regrouper les données et les fonctionnalités associés en enitités logicielles autonomes

- Identifier et grouper les ≪choses qui vont bien ensemble≫ en modules
- Les modules sont testés indépendemment pour assurer la robustesse du code (tests unitaires)
- Séparer les modules les uns des autres (modularité)
- Réduire au strict minimum la visibilité inter-modules (encapsulation)
- Si nécessaire : structuration hiérarchique des modules (héritage)
- Programme principal en Lego : chef d'orchestre entre modules

C++ et Python sont nativement des langages orientés objet

La jungle des benchmarks du web est aussi la jungle des biais (sauf à réunir les hyper-spécialistes de chaque langage), mais elle apporte des tendances avérées...

▸ B1 ▸ B2 ▸ B3 ▸ B4

‘C++ rocks”, Python Numpy... lenteur du “pure” Python ou R. Mais est-ce aussi simple ?

Typologie des langages

Benchmarks, à prendre et à laisser ?

```python
k = 0
for i in range(10**7): # 4.5 seconds
  k = k+i
```

```c
#include <stdlib.h>
#include <math.h>
int main()
{
  int k = 0;
  for (int i=0; i<int(pow(10,7)); i++){ // poussieres de secondes
        k += k+i;
  }
}
```

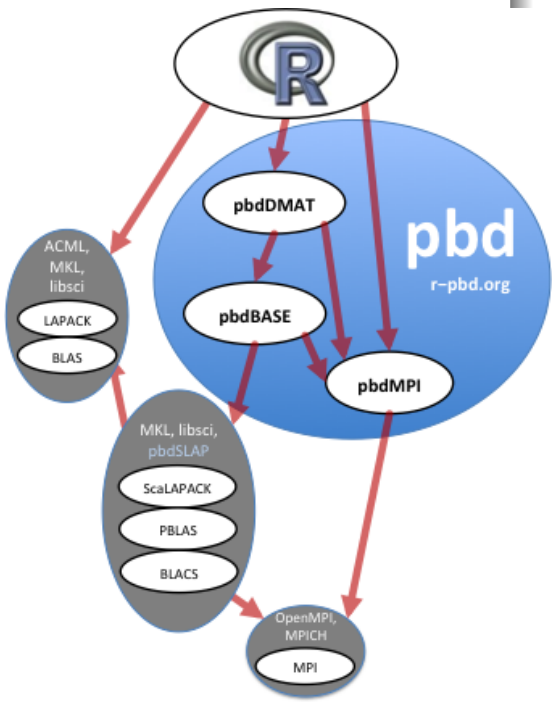
```r
k <- 0
for(i in 1:10^7) # 15 seconds
  k <- k+i
```

LA tendance : interfacer les langages pour prendre le meilleur de chacun, i.e. permettre la transmission des données entre plusieurs parties implementée dans différents langages.

Les modules Python (packages R) les + efficaces sont des interfaces Python (R resp.) vers C/C++ ou Fortran.

L'exemple de NumPy "To build the module, you'll need a C compiler. Various NumPy modules use Fortran 77 libraries, so you'll also need a Fortran 77 compiler"

Nombreux codes C++ interfacés avec les routines Fortran 77 de référence (BLAS, LAPACK, ARPACK) mais attention à l'ordre en mémoire :
$t[0][0], t[0][1] \ldots t[n][m-1], t[n][m]$ en C++
mais
$t(1,1), t(2,1), \ldots, t(m-1,n), t(m,n)$ en Fortran !



LA tendance : interfacer des langages en Calcul.

Schéma classique d'un code de calcul (HNI : langage de haut niveau interprété, NIC : niveau intermédiaire compilé)

- lecture des données + pré-traitement HNI
- étapes de calcul MNC
- post-traitement, visualisation et/ou écriture des données HNI

See "Extending Python with C or C++" or "Writing R extensions with foreign language interfaces"

Au delà des querelles...

Perl is useful but painful !

Utilizing Pythons built-in methods or external modules can produce near or better than C++ performance.

Apprendre Python, c'est apprendre le meilleur du C++, du Java, du Fortran, etc...

Python ran an average of xxx slower than C++

The best thing about R is that it was developed by statisticians. The worst thing about R is that...it was developed by statisticians !

C++ is the only way to go for low level systems programming C++ is life ! It's beauty ! It's elegance and performance !

C/C++

+++	Langage très performant (si on fait du “vrai” C++, pas du C)
–	Apprentissage difficile et long (pour tirer le meilleur du langage), apprentissage de la compilation&co (voir cours suivants)
+++/-	Verbeux mais très sûr
–	Pas adapté pour le prototyping
+	Interface “tricky” avec Fortran
+++	POO nativement
+	Boost librairies, C++11
+++	Adapté à tous les niveaux de parallelisme : classique (openMP, MPI) ou émergent (TBB, Silk)
-	E/S et graphiques

Python

Python remplace avantageusement Matlab (payant, code propriétaire)

`I used Matlab. Now I use Python.`

+/-	Performant ssi utilisation des modules ad-hoc (NumPy)
+++	Apprentissage facile. Adapté à la pédagogie de programmation
++	Rapidité de développement
+++	Prototyping
+++	Interface avec le C++ avec swig, cython, Boost.Python ou weave
+++	POO nativement
++/-	Jungle des modules
+	Modules pour le parallélisme classique (multiprocessing, MPI4py)
++	E/S et graphiques

R

+/-	Langage peu performant (pas de passage by reference)
	sauf si on chaîne les packages plus qu'on ne programme avec la syntaxe R
+/-	Apprentissage ambigüe (facile pour les non-programmeurs)
++	Rapidité de développements de petits programmes
+++	Prototyping
+++	Interface avec C++ et Fortran (R-extensions)
-	POO non native et peu efficace (S4)
+++/-	Jungle des packages
+	Emergence du parallélisme avec R
+++	E/S et graphiques

Typologie des langages

Au delà des querelles...

```
f = open('lang4.dat')
dico = {}

for line in f.xreadlines():
    elts = line.strip().split(' ')
    dico[elts[0]] = elts[1]

for key in dico.keys():
    print dico[key],' pour la cle ',key
```

```
PuyDeDome 63
Cantal 15
Hauteloire 43
Allier 03
```

```
vmiele@tulipe$ python lang4.py
43  pour la cle  Hauteloire
15  pour la cle  Cantal
03  pour la cle  Allier
63  pour la cle  PuyDeDome
```

Typologie des langages

Au delà des querelles...

```
#include<fstream>
#include<iostream>
#include<map>
#include<string>
#include<sstream>
using namespace std;
int main()
{
  ifstream f;
  f.open("lang4.dat");
  map<string, int> dico;

  string line;
  while (!f.eof()){
    getline(f, line);
    istringstream linestream(line);
    string a;
    int b;
    linestream>>a;
    linestream>>b;
    dico[a] = b;
  }

  for (map<string, int>::iterator iter=dico.begin();
       iter!=dico.end(); iter++){
    cout<<iter->first<<" pour la cle "<<iter->second<<endl;
  }
}
```

Definition

La *complexité* d'un problème caractérise les ressources nécessaires pour les résoudre. Les ressources essentielles sont le temps (d'éxecution) et l'espace (mémoire).

Jusqu'aux années 70, seule la mesure expérimentale de la complexité d'un algorithme était (parfois) effectuée et dépendait des machines. . . une mesure théorique devint souhaitable !

Definition

La *complexité algorithmique* est un ordre de grandeur théorique du temps de calcul (*complexité en temps*) et/ou de l'espace mémoire (*complexité en mémoire*) utilisé en fonction d'une mesure des données.

Souvent, si un algorithme permet de gagner du temps de calcul, il occupe davantage de place en mémoire. . . vigilance. . .

La complexité n'est pas la même selon les cas

- complexité au pire : complexité maximum, dans le cas le plus défavorable
- complexité en moyenne : il s'agit de la moyenne des complexités obtenues selon les cas
- complexité au mieux : complexité minimum, dans le cas le plus favorable. En pratique, cette complexité n'est pas très utile

Exemple (complexité en temps) : chercher un élement dans une liste de taille n
pire : n opérations
moyenne : (n+1)/2 opérations
au mieux : 1 opération

Si on veut comparer des algorithmes sans considérer l'implémentation, on peut comparer les complexités au regard de la taille attendue des données

- complexité au pire
- forme générale de la complexité, qui indique la façon dont elle évolue en fonction d'un paramètre (ou plusieurs).
  On utilisera la notation $O(\ldots)$ qui veut dire *de l'ordre de...*

Definition

Notation de Landau : g est dite en $O(f)$ s'il existe des constantes $c > 0$ et $x_0 >> 0$ tq $g(x) < c * f(x)$ pour tout $x > x_0$. On note $g = O(f)$ et on dit que g est dominée asymptotiquement par f.

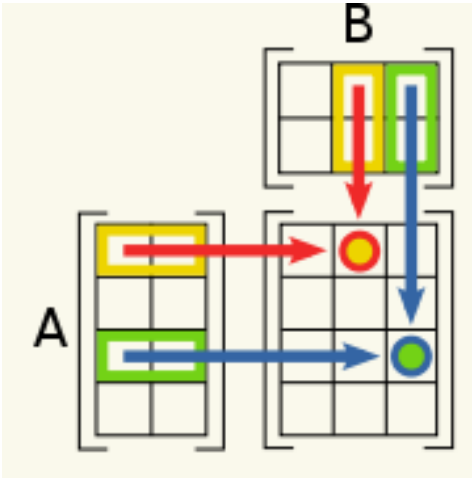
Si $g = O(f)$ et $f = O(h)$ alors $g = O(h)$
si $g = O(f)$ et k un nombre, alors $k * g = O(f)$
si $f1 = O(g1)$ et $f2 = O(g2)$ alors $f1 + f2 = O(g2 + g2)$
si $f1 = O(g1)$ et $f2 = O(g2)$ alors $f1 * f2 = O(g2 * g2)$

Si n est la taille des données, complexité en temps :
$O(1)$ *constante* : extraction d'un élément dans un tableau
$O(log(n))$ *logarithmique* : recherche dichotomique dans une liste triée (voir après)
$O(n)$ *linéaire* : parcours d'une liste
$O(nlog(n))$ *quasi-linéaire* : tri d'un tableau
$O(n^2)$ *quadratique* : calcul du maximum d'une matrice carré
$O(n^3)$ *cubique* : multiplication matricielle (voir après), classification ascendante hiérarchique (voir après)
$O(2^n)$ *exponantielle* : problème du sac à dos par force brute (énumération)...

NB : la notation de Landau peut utiliser plusieurs paramètres décrivant la taille du problème (exple : $O(m * n + q)$)

Exple : Produit matrice x matrice

Pour la produit de deux matrice de taille mxn et nxp, on réalise n produits $+n$ sommes pour chaque élement de la matrice finale de taille mxp.
On a donc une complexité $O(m * p)$ en espace et $O(n * m * p)$en temps



Pour le cas où $m = p = n$, $O(n^2)$ en espace et $O(n^3)$en temps.

"L'algorithme de Coppersmith-Winograd [...] est en $O(n^{2,376})$ [...] Mais aucune implémentation de l'algorithme n'est utilisée car la constante dans le grand O est prohibitive." (Wikipedia)

Exple : Recherche dichotomique dans un tableau trié

On coupe le tableau en deux et on cherche l'élément dans une des deux parties en répétant le même traitement récursivement.

```
fonction rechercheElement(tab, x)
   i <- 0; j <- tab.longueur-1;
   tantque (i <= j) faire
      si (tab[(j+i)/2] = x) alors retourne VRAI;
      sinon
         si (tab[(j+i)/2] > x) alors j <- (j+i)/2 - 1;
         sinon i <- (j+i)/2 + 1;
```

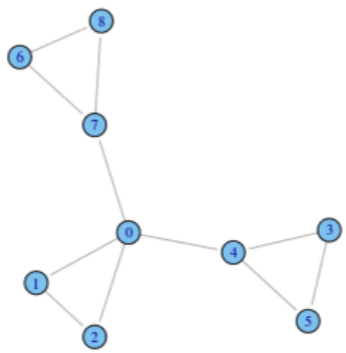
Au pire, la longueur de la partie du segment de tableau $[i, j]$ à traiter est d'abord n, puis $n/2$, puis $n/4$, ..., jusqu'à ce que $n/2^t = 1$.
Le nombre de tours de boucles est $t = log2(n)$. La complexité en temps est donc $O(log(n))$. $O(n)$ en espace.

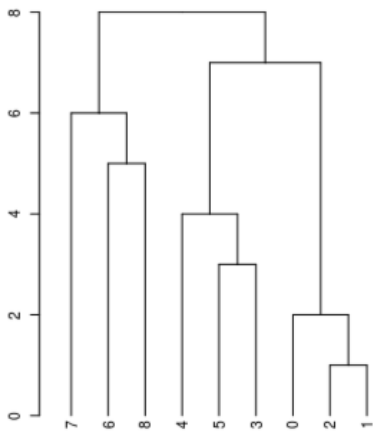
Exple : Classification ascendante hiérarchique

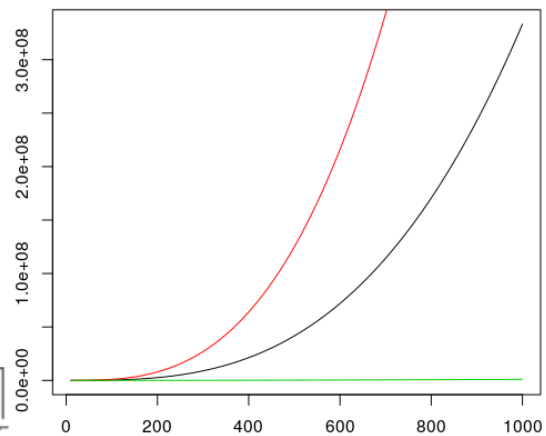
n entités, et on sait calculer la distance entre 2 groupes d'entités (ou singletons)

```
n(n-1)/2 calculs de distance
puis
k <- 0
tantque k<n-2
      (n-k)(n-k-1)/2 recherche de min -> (i,j,min)
      (n-k-1) recalculs de distance après fusion(i,j)
```

En espace $n(n-1)/2$ si on est malin $= O(n^2)$
En temps $n(n-1)/2 + \sum_{k=0,n-2}(n-k)(n-k-1)/2 + (n-k-1) = O(n^3)$







Le nombre de processeurs disponibles est une données à prendre en compte dans la complexité. Deux alternatives pour exprimer la complexité :

- $O(f(n))$ avec $g(n)$ processeurs : exprime la complexité obtenue avec un nombre idéal de processeurs
- $O(f(n,p))$ avec p le nombre de processeurs : plus pragmatique
  Permet d'intégrer la loi d'Amdhal : $O_{seq}(f(n))) + O_{par}(g(n)/p)$
  Si $O_{seq}(f(n))$ est dominé, alors on garde $O_{par}(g(n)/p)$.

NB : attention à $O_{comm}(h(n))$!

La complexité nous aide à choisir le bon algorithme et les bonnes structures de données. C'est un indicateur.

Mais attention aux constantes devant le $O(\ldots)$!

Et attention aux coûts réels engendrés par les accès mémoire, la bande passante, les architectures des processeurs.

Un bon algorithme en machine est d'abord celui qui privilégie la localité spatiale et temporelle !

Exple : Deux algorithmes de même complexité, mais l'un respecte la localité spatiale et temporelle, l'autre non.

```
#include <vector>
using namespace std;
int main(int argc, char ** argv )
{
  int s = 18000;
  vector<vector<double> > v(s,vector<double>(s, 0.));

  // real 0m8.970s
  for (int i=0; i<s; i++)
    for (int j=0; j<s; j++)
      v[i][j] = v[i][j] + v[i][j];

  // real 0m45.077s
  for (int i=0; i<s; i++)
    for (int j=0; j<s; j++)
      v[j][i] = v[j][i] + v[j][i];
}
```

Du cerveau à l'odinateur : copier/coller ?

Prélude : dans ce qui suit, on s'appuit sur les *containeurs* de la STL C++, mais des équivalents existent dans les autres langages.

Sur la papier, on a des structures de données de type :

- vecteur
- matrice pleine
- matrice creuse (ou graphe) avec m élements tq $m \sim O(n)$ ou $m << n^2$
- des listes (triés), des piles
- des couples "identifiants-observations"
- ...

Doit-on chercher à utiliser formellement les même structures sur le papier et en machine ? ou seulement moralement ?
On veut des performances en temps ET de la parcimonie en espace.

La clé de voûte de l'efficacité, car cohérence avec la structure en lignes de caches (cf. cours précédent)

- simple tableau de taile n (mémoire contigüe !) avec information de longueur
- access at i $O(1)$
- find $O(n)$
- insert at i $O(n - i)$ copy (double sa capacité si nécessaire)
- append $O(1) + O(n)$ copy au pire (idem)
- delete (after access) $O(n)$

NB : on peut aussi créer des tableaux comme en C

Exple : Encapsulation d'un vecteur interne pour gérer les matrices pleines

```
class Matrix
{
private:
  int _s;
  vector<double> _vinternal;
public:
  Matrix(int s)
    : _vinternal(s*s, 0), _s(s) {}
  Matrix(const Matrix& m)
    : _vinternal(m._vinternal), _s(v._s) {}
  ~Matrix() {}
  inline double& element(int i, int j){
    return _vinternal[i*_s+j];
  }
};
```

Structure favorisant l'ajout/retrait des élements. La notion de longueur n'est pas fondamentale car variable.

- liste doublement-chainée
- access at i $O(i)$
- find $O(n)$
- insert at i $O(i)$
- append $O(1)$
- delete (after acess) $O(1)$

Exple : Un liste d' élements évolue, chacun pouvant survire ou non (delete de liste) et donner naissance à un nouvel élément (ajout en fin)

- une `list<.>` devrait convenir
- une approche basée sur les `vector<.>` mérite d'être testée, avec ajout en fin de vecteur (naissance) ou copie du dernier élement à la place du delete (non survie) $\rightarrow$ danger de la copie

A tester après profiling !

Permet de gérer un ensemble (au sens ensembliste - unicité) d'élements qui seront placés dans la structure selon un critère de tri.

Premier containeur associatif, i.e. on accède à l'élément par une clé.

- red-black tree (variante des arbres binaires [1]) Démo
- access at i dangerous ($O(i)$ mais c'est mouvant)
- find/delete/insert a specific key in essentially $O(logn)$ (sauf cas de dégénéresence)

Structure triée par construction, mais as-t-on besoin qu'elle le soit à chaque étape de sa construction ?

1. plus petites (grandes) valeurs dans le sous arbre de gauche (droite resp.)

Exple : Insertion dans un `set<.>` ou tri/unicité à la fin avec un `vector<.>` ?

```
1  #include<set>
   #include<vector>
3  #include<algorithm>
   #include<stdlib.h>
5  using namespace std;
   int tirage(int n){
7    return (int)((double)rand()/((double)RAND_MAX + 1) * n);
   }
9  int main(int argc, char ** argv)
   {
11   vector<int> vec;
     set<int> s;
13   int n=1e7;

15   // 2 sec
     for (int i=0; i<n; i++)
17     vec.push_back(tirage(n));
     sort(vec.begin(), vec.end());
19   vec.erase( unique( vec.begin(), vec.end() ), vec.end() );

21   // 16 secondes
     for (int i=0; i<n; i++)
23     s.insert(tirage(n));
   }
```

B

Exple : Une matrice creuse en 0/1 (cas d'école ici) est une liste d'élements non nuls avec leur index. Comment l'interroger ? A-t-on besoin de la modifier ?

- `list<pair<unsigned int, unsigned int> >` : liste des paires d'indices → difficile à interroger
- `list<list<unsigned int> >` : une liste par ligne de la matrice, chaque liste contient les indices des colonnes non nulles → dynamique mais pas de contiguïté
- `vecteur<list<unsigned int> >` : si on connait le nombre de lignes, comme précédemment mais plus efficace ?
- `vecteur<set<unsigned int> >` : comme précédemment mais on a le tri et l'unicité des indices des colonnes → le tri est-il nécessaire à chaque étape ?

Exple : Une matrice creuse en 0/1 (cas d'école ici) est une liste d'élements non nuls avec leur index. Comment l'interroger ? A-t-on besoin de la modifier ?

- `vector<list<unsigned int> >` puis `sort/uniq` puis `vector<vector<unsigned int> >` : on passe par une structure temporaire pour aller vers la structure la plus efficace $\rightarrow$ poids de la recopie versus poids de l'utilisation de la structure finale

Quid des performances en machine : il faut tester/profiler.

Les conteneurs STL sont hyper-optimisés, les différences sont donc sensibles au delà d'une certaine taille de problèmes.
C++ coding standards, Herb Sutter, item 8 "Don't optimize prematurely"

Dictionnaire et table de hachage - `map<.,.>` et `unordered_map<.,.>`

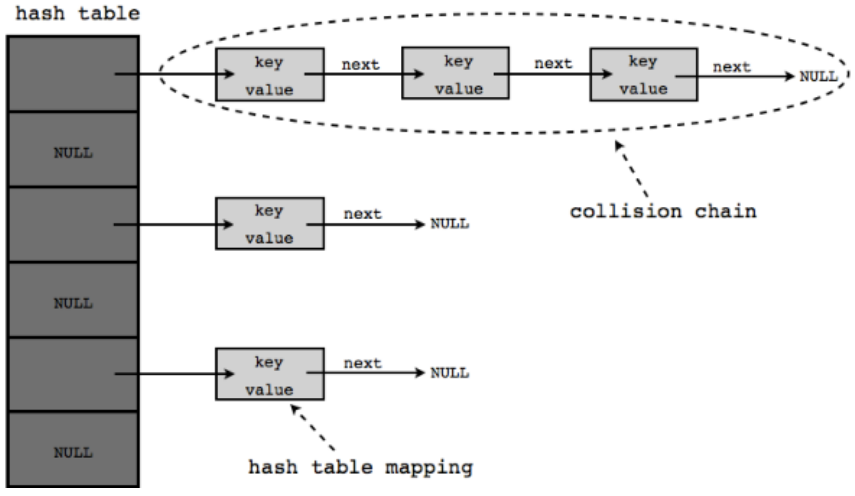
Associer des valeurs à des clés qui ne sont pas des entiers consécutifs (containeur associatif)

`map`

- `map<.,.>` == `set<pair<.,.> >`
- même comportement que le `set<.>`
- access avec opérateur [ ] mais attention $O(logn)$!

`unordered_map`

- access par fonction de hachage (calcul d'un index depuis une clé) en $O(1)$ au mieux
- comme son nom l'indique, les élements ne sont pas ordonnés mais on peut les parcourir
- très efficace en pratique



Structures de données

Dictionnaire et table de hachage - map<.,.> et unordered_map<.,.>

```
#include<map>
#include<vector>
#include<string>
#include<stdlib.h>
using namespace std;
int tirage(int n){
  return (int)((double)rand()/((double)RAND_MAX + 1) * n);
}
int main()
{
  vector<string> v;
  map<int, string> m;
  int n = 1e7;

  /******* init ***************/
  for (int i=0; i<n; i++)
    v.push_back("PuyDeDome");
  for (int i=0; i<n; i++)
    m[i] = "PuyDeDome";

  /******** acces **************/
  // 4 secondes
  for (int i=0; i<n; i++){
    int j = tirage(n);
    v[j] = "Cantal";
  }

  // 25 secondes
  for (int i=0; i<n; i++){
    int j = tirage(n);
    m[j] = "Cantal";
  }
}
```

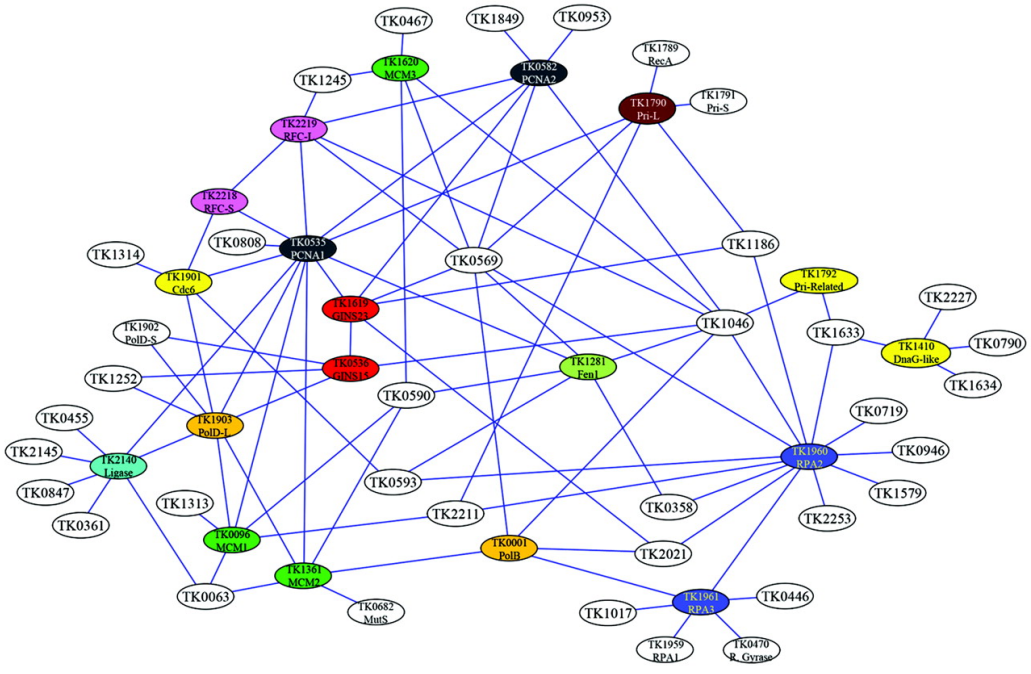
Dictionnaire et table de hachage - `map<.,.>` et `unordered_map<.,.>`

Exple : Une matrice creuse à valeurs réelles. . .

- `map<pair<unsigned int, unsigned int>, double>` : pratique mais pas de contiguité en mémoire, empreinte mémoire conséquente
- triplet de `vector` : les valeurs, le positionnement par ligne et les indices de colonnes $\rightarrow$ encapsulation
- s'appuyer sur une librairie

Dictionnaire et table de hachage - `map<.,.>` et `unordered_map<.,.>`

Exple :



Nécessité de retenir les noms des n sommets d'un graphe pour le pre/post-traitement, mais inutilité dans les étapes de calculs :

- conversion des noms en entiers consecutifs + stockage de arètes en matrice creuse
- utilisation de `map<string,unsigned int>` pour la conversion ou `vector<string>` + `find` en $O(log(n))$
- attention à la consommation de mémoire additionnelle

Use vector by default !

Lecture : Effective STL, Scott Meyers, item 23 “Consider replacing associative containers with sorted vectors”

Lecture : C++ coding standards, Herb Sutter, item 76 “Use vector by default. Otherwise choose an appropriate container”

No container independent code !

Lecture : Effective STL, Scott Meyers, item 44 “Prefer member functions to algorithms with the same names”

“Be curious” = Lire les specs des structures proposées (cppreference, Wikipedia,. . .)

Lecture : C++ coding standards, Herb Sutter, item 8 “Don’t optimize prematurely”

Lecture :The C++ standard library, Nicolai Josuttis, page 10, “Remeber to think big when you consider complexity”

Faire des benchmarks pour se convaincre.

Prendre en considération l’empreinte mémoire des structures choisies.

“Be lazy” = Toujours utiliser les structures de données des bibliothéques au plus haut niveau possible.

Ouvrir des livres et des revues

+ faire des benchmarks et du profiling (voir cours suivants)

+ participer à des réseaux métiers/collègues

+ éviter tout dogmatisme

+ Lecture : C++ coding standards, Herb Sutter, item 6 “Correctness, simplicity and clarity come first”