Algorithmique et structures de données 3

Irena.Rusu@univ-nantes.fr

LINA, bureau 123, 02.51.12.58.16

Déroulement

- 12 heures de CM (Irena Rusu, Irena.Rusu@univ-nantes.fr)
- 24 heures de TD (2 groupes)
- 12 heures de TP (3 groupes)

- Contrôle continu : date à préciser
- Note de TP : examen sur machine lors d’une séance à préciser
- Examen : semaine 51 (à confirmer)

- Et ce n’est pas tout …

Une note à chaque CM

- QCM: environ 10 minutes, en début de chaque cours
- Types de questions :
  - sur le cours précédent
  - de pure logique, de base, ou pièges
- Pourquoi : pour être prêt !
- Et aussi : parce qu'il y a une note par QCM

Absences :

- 1 seule absence ponctuelle justifiée (cf. scolarité) est acceptée (note annulée)
- Toutes les autres absences, justifiées ou non, produisent une note de 0 (sauf cas particulier des absences justifiées de longue durée)

Compléments

ni autres tablettes etc.

Réponses à une question que l’on me pose souvent :

- Note : 0.5*(NoteCC*0.4+NoteQCM*0.2+NoteTP*0.4)+0.5*NoteExam1 en 1ère session
- Note : 0.4*(NoteCC*0.4+NoteQCM*0.2+noteTP*0.4)+0.6*NoteExam2 en 2ème session

Dans ce cours

- Encore de l'algorithmique ?!
- Rappels sur l'efficacité des algorithmes
- Efficacité des algorithmes et implémentation
- Quand n requêtes $\neq$ n • (une requête)

Références

- A. Aho, J. Hopcroft, J. Ullman,
  Structures de données et algorithmes
  InterEditions, 1987.
- Th. Cormen, Ch. Leiserson, R. Rivest (© multiples figures du cours)
  Introduction à l'algorithmique
  Dunod, 1994.
- C. Froidevaux, M.C. Gaudel, M. Soria,
  Types de données et algorithmes
  Edisciences, 1994.
- R. Sedgewick

  Algorithmes en Java

  Pearson Education, 2004.

- Transparents : © I. Rusu, © M. Crochemore + bien d'autres (indiqués à chaque cours)

Dans ce cours

- Encore de l'algorithmique ?!
- Rappels sur l'efficacité des algorithmes
- Efficacité des algorithmes et implémentation
- Quand n requêtes $\neq$ n • (une requête)

Pourquoi un autre cours d'algorithmique ?

- L'algorithmique est le « permis de conduire » catégorie ordinateur.
- Le code (le CM) et la conduite (les TD, TP) se font en parallèle …
- … Mais sur la durée.

- Dans ce module, au programme
  - Représenter de manière structurée les ensembles (de … tout).
  - Se donner les moyens d'insérer, chercher, supprimer des éléments.
  - Utiliser ces représentations et les opérations associées pour résoudre des problèmes somme tout pas très compliqués.
  - Jeter un œil au-delà.
- Tout cela avec un mot d'ordre : efficacité (en temps de calcul).

Dans ce cours

- Encore de l'algorithmique ?!
- Rappels sur l'efficacité des algorithmes
- Efficacité des algorithmes et implémentation
- Quand n requêtes $\neq$ n • (une requête)

Efficacité des algorithmes (rappels)

- **Deux** points de vue:
  - Mémoire utilisée
  - Temps d'exécution (~ nombre d'opérations)
- Un **seul** paramètre qui fournit l'unité de mesure:
  - La taille des données en entrée

Exemple. Trouver le trajet SNCF le plus court (en temps) dans une liste fournie :

T1 : 3h46min

T2 : 2h59min

T3 : 3h05min

T4 : 3h01min etc.

Trajets SNCF : un algorithme simple

- T : $t_1, t_2, t_3, t_4, t_5, \ldots, t_n$

  la séquence non-ordonnée des temps de parcours
- Implémentation : tableau ? liste ? Autre ?

Algorithme TrajetsSNCF

min $\leftarrow t_1$	1 opération
pour i de 2 à n faire {	2(n-1) opérations ($i \leftarrow 2$, $i \leftarrow i+1$, $i \leq n$)
si (min > t_i) alors min $\leftarrow t_i$}	1 ou 2 opérations, pour tout i
afficher (min)	1 opération

Total : au plus 1 +2(n-1)+2(n-1)+1=4(n-1)+2 opérations.

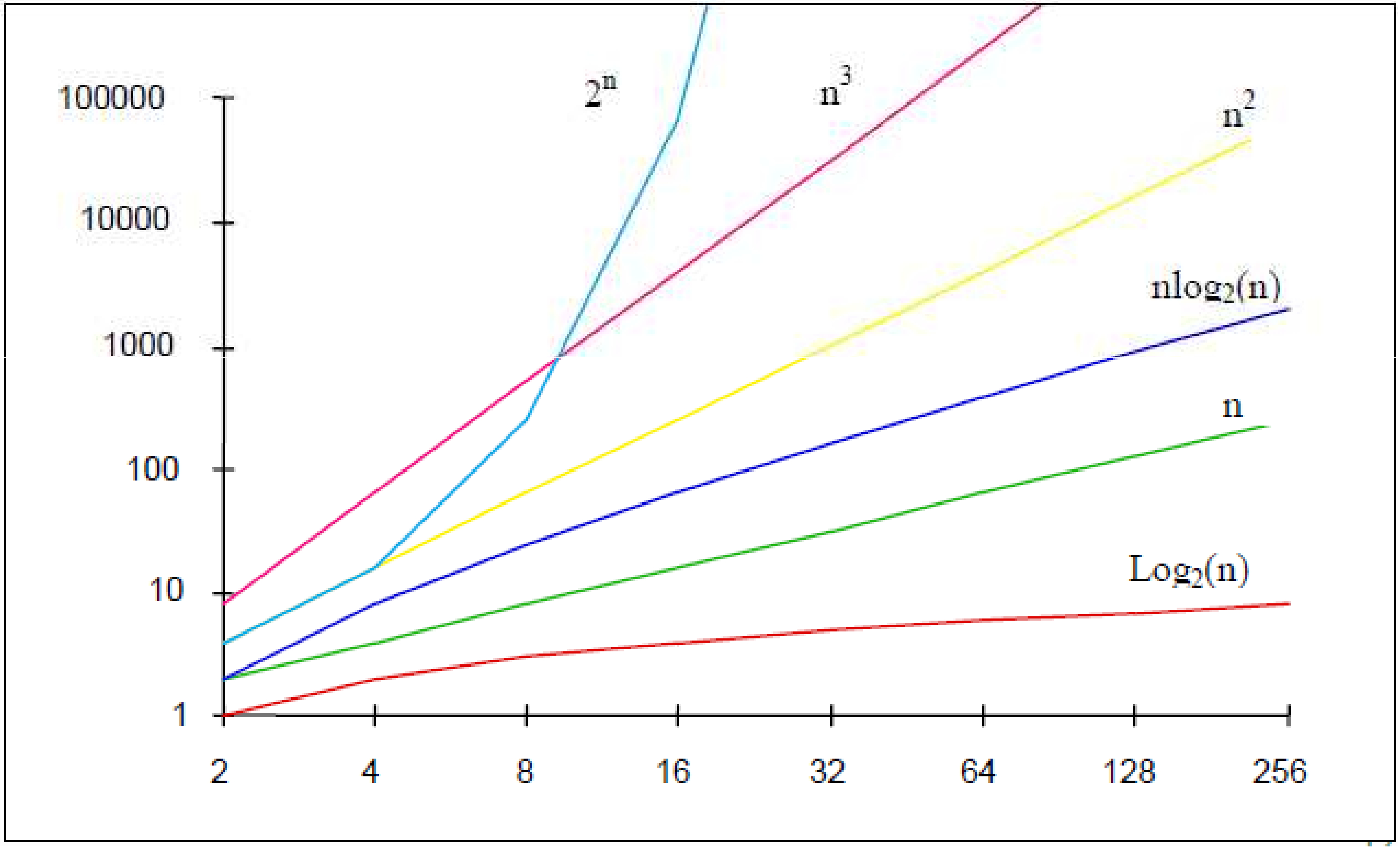
Trajets SNCF, algorithme simple : efficacité ?

- Bilan du problème (hors tout algorithme)
  - n temps de parcours donnés
  - Il faut tous les regarder pour savoir le plus petit
  - Il faut tous les comparer, au moins une fois chacun
  - Donc l'algorithme devra faire au moins 2n opérations
- Notre algorithme fait 4(n-1) +2 opérations, au pire des cas. Est-ce trop ?

NON, c'est parfait

- Les constantes (4, -1, 2) ne comptent que très peu lorsque n augmente.

Croissance des fonctions utilisées



Le plus souvent on écrit log n au lieu de $\log_2 n$

Conclusions sur le nombre d'opérations (≈ temps d'exécution)

- La position relative des courbes reste globalement la même lorsque n augmente, même si on multiplie par (ou on ajoute) des constantes
- Donc une fonction du genre

$$f(n)= 2n \text{ ou } g(n)=4n+2$$

suivra le même genre de courbe que $h(n)=n$, et restera assez proche de la courbe de $h(n)$.

- Et donc en général nous ne ferons aucune différence entre

  f, g ou h. Les ordres de grandeur de f, g, n sont similaires.
- On écrit : $f(n)=\theta(n)$, $g(n)=\theta(n)$, $h(n)=\theta(n)$ (lire « Theta »)

Conclusions sur le nombre d'opérations (≈ temps d'exécution)

- La position relative des courbes reste globalement la même lorsque n augmente, même si on multiplie par (ou on ajoute) des constantes
- Donc une fonction du genre

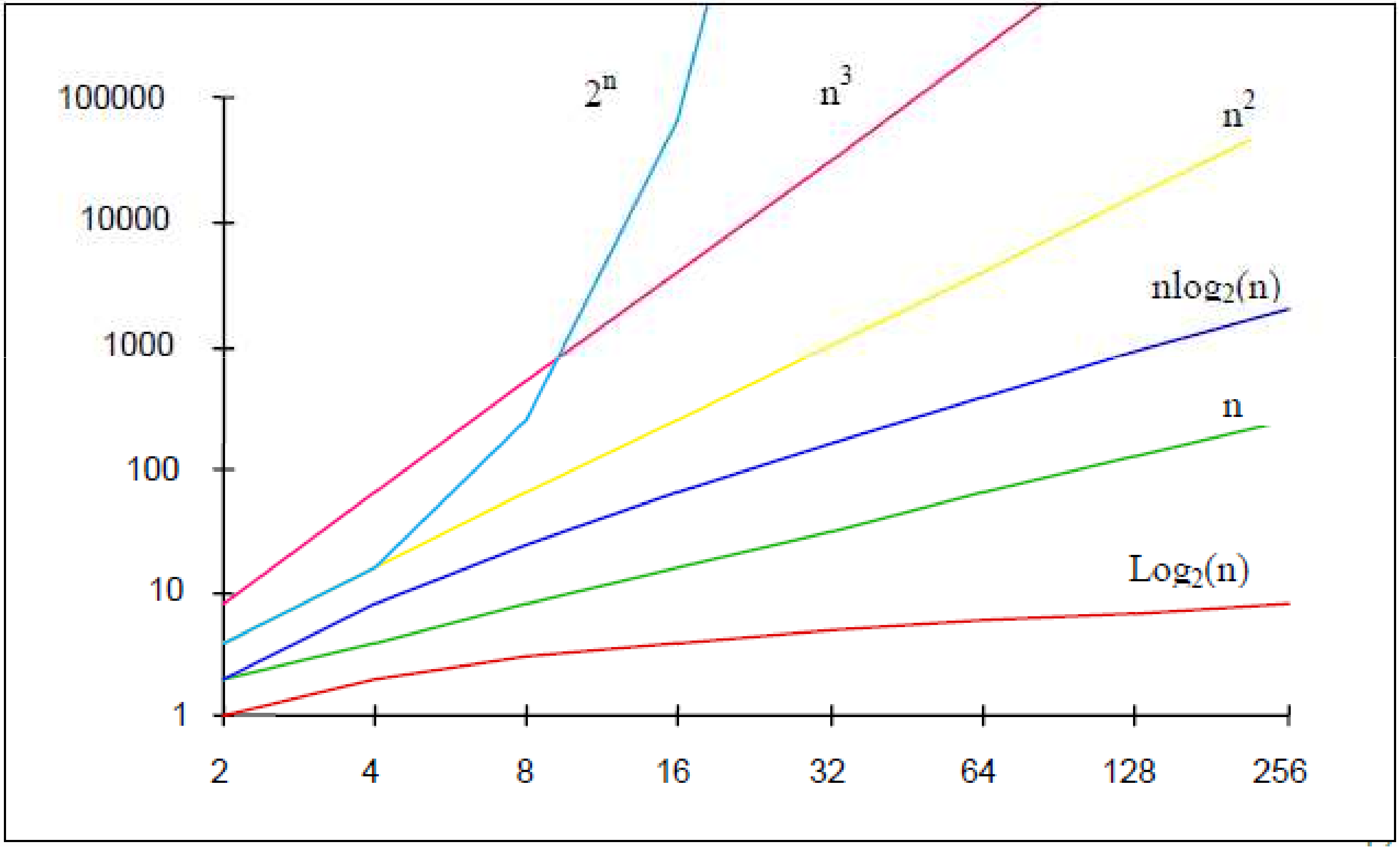
$$f(n)= 2n \text{ ou } g(n)=4n+2$$

aura une courbe globalement comme $h(n)=n$, qui elle-même est globalement en dessous de la courbe de (par exemple) $r(n)=n \log n$, à partir d'une certaine valeur seuil (qui n'est pas importante).

- On écrit : $f(n)=O(n \log n)$, $g(n)=O(n \log n)$, $r(n)=O(n \log n)$ (lire grand « O ») pour dire que f, g, n sont « globalement » bornées supérieurement par n log n.
- Evidemment $f(n)\neq\theta(n \log n)$, $g(n)\neq\theta(n \log n)$ mais $r(n)=\theta(n \log n)$

Croissance des fonctions utilisées

Exemples : $2n^3+250=O(2^n)$, $2n^2+5=\theta(n^2)$, $4n^5+3n^3+12=\theta(n^5)$ (eh, oui …)



Le plus souvent on écrit log n au lieu de $\log_2 n$

Et formellement

- $f(n)=O(g(n))$ s'il existe $k>0$ et n_0 tels que

  pour tout $n>n0$, $f(n) \leq k \bullet g(n)$

  (f est asymptotiquement bornée supérieurement par g)

- $f(n)=\theta(g(n))$ s'il existe $k_1>0$, $k_2>0$, n0 tels que

  pour tout $n>n0$, $k_1 \bullet g(n) \leq f(n) \leq k_2 \bullet g(n)$

  (f est asymptotiquement bornée inférieurement et supérieurement par g).

Les temps d'exécution confirment la légitimité de ces approximations.

Exemples de temps d'exécution

Not.	O(1)	O(log n)	$O(n^{1/2})$	O(n)	O (nlogn)	$O(n^2)$	$O(n^3)$	$O(n^{\log n})$	$O(e^n)$	O(n!)
N=5	10ns	10ns	22ns	50ns	40ns	250ns	1.25µs	30ns	320ns	1.2µs
N=50	10ns	20ns	71ns	500ns	850ns	25µs	1.25ms	7µs	130j	10^{48}ans
N=250	10ns	30ns	158ns	2.5µs	6µs	625µs	156ms	5ms	10^{59} ans	
$N=10^3$	10ns	30 ns	316ns	10µs	30µs	10ms	10s	10s		
$N=10^5$	10ns	60ns	10µs	10ms	60ms	2.8h	316ans	10^{20} ans		

Efficacité des algorithmes

- Très grossièrement (et théoriquement) :

« efficace » ~ polynomial ($1, \log n, n\log n$, n^2, n^3, $n \log n$, n^{17}), où n = taille des données

«inefficace » ~ au-delà (2^n, e^n, $n!$, 2^{2^n}, $n^{n^{n^n}}$)

- Pratiquement :
  - Des algorithmes polynomiaux avec un gros exposant ne seront pas efficaces
  - Certains algorithmes exponentiels seront efficaces sur une très grande majorité d'instances (et inefficaces sur très peu d'instances). Un calcul de complexité « moyenne » permet de les identifier.

But suivi : trouver des algorithmes aussi efficaces que possible.

Dans ce cours

- Encore de l'algorithmique ?!
- Rappels sur l'efficacité des algorithmes
- Efficacité des algorithmes et implémentation
- Quand n requêtes $\neq$ n • (une requête)

Trajets SNCF : et l'implémentation ?

Dans ce cours: JAVA (de base …)

Ici : implémentations très proches de l'algorithme (et très loin des possibilités offertes par Java)

Algorithme TrajetsSNCF

min ← t_1
pour i de 2 à n faire {
si (min > t_i) alors
min ← t_i
}
afficher (min)

```
public void CalculerTrajets (Trajets T)
  {
    float min;
    int n =T.longueur();
    min = T.elem(0);
    for (int i = 1; i<=n-1; i++)
      {if (min>T.elem(i))
          min = T.elem(i);}
     System.out.println(''min :''+min);
  }
```

Attention : ALGORITHME ≠ PROGRAMME

Remarques sur l'implémentation

Java est un langage « objet » …

Alors :

- Trajets est une classe, avec des champs privés
- La structure de données utilisée (tableau, liste …) n'est pas visible
- Mais on sait qu'elle est parcourue de manière linéaire …
- On accède aux champs privés avec des méthodes de la classe T.elem(), T. longueur() …

```
public void CalculerTrajets (Trajets T)
  {
    float min;
    int n=T.longueur();
    min=T.elem(0);
    for (int i=1; i<=n-1; i++)
      {if (min>T.elem(i))
          min=T.elem(i);}
     System.out.println(''min :''+min);
   }
```

Implémentation et nombre d'opérations

- Un appel à une fonction n'est (d'habitude) pas une opération
- Le calcul du nombre d'opérations se fait sur l'algorithme en imaginant l'implémentation, pour que chaque appel soit correctement évalué.
- Sinon … problèmes :
  - T. longueur() peut être en O(1) ou O(n), sans changer la complexité calculée
  - T.elem(i) ?

```
public void CalculerTrajets (Trajets T)
   {
     float min;
     int n=T.longueur();
     min=T.elem(0);
     for (int i=1; i<=n-1; i++)
       {if (min>T.elem(i))
          min=T.elem(i);}
      System.out.println(''min :''+min);
    }
```

Implémentation et nombre d'opérations

- T.elem(i) ? Tableau

```
public class Trajets
    {private float[ ] tab;
     [….]
    public float elem (int i)
   {return tab[i];}
    public int longueur ()
   {return tab.length;}
   }
```

Nombre d'opérations:

T.elem(i) en O(1)

CalculerTrajets en O(n)

```
public void CalculerTrajets (Trajets T)
   {
     float min;
     int n=T.longueur();
     min=T.elem(0);
     for (int i=1; i<=n-1; i++)
       {if (min>T.elem(i))
           min=T.elem(i);}
      System.out.println(''min :''+min);
    }
```

OK !

Implémentation et nombre d’opérations

- T.elem(i) ?   Liste chaînée

```
public class Trajets
    {private LinkedList<Float> li;
     [….]
    public float elem (int i)
  {ListIterator <Float> it =
    li.listIterator();
    int j=0;
    while (j<i)
      { it.next(); j++; }
    return(it.next());
    }}
```

```
public void CalculerTrajets (Trajets T)
   {
     float min;
     int n=T.longueur();
     min=T.elem(0);
     for (int i=1; i<=n-1; i++)
       {if (min>T.elem(i))
           min=T.elem(i);}
      System.out.println(‘’min :’’+min);
    }
```

Nombre d’opérations:

T.elem(i) en O(i)   CalculerTrajets en $O(n^2)$ pas OK !!!

Conclusions sur cet exemple

- Cacher à l'utilisateur la structure des données au niveau de CalculerTrajets : OK (« abstractisation par encapsulation »)

- … mais avec une perte (non-négligeable) lors de l'implémentation par des listes (parcours en O(n) très facile à réaliser par ailleurs)

- Comment y remédier dans ce cas précis ?

  Pas de T.elem(i) mais parcours à la file (par un iterateur, p.ex)

Dans ce cours

- Encore de l'algorithmique ?!
- Rappels sur l'efficacité des algorithmes
- Efficacité des algorithmes et implémentation
- Quand n requêtes $\neq$ n • (une requête)

Trajets SNCF : un pas plus loin

- Et si on n'a pas qu'une liste de trajets, mais plusieurs listes ? (p.ex., chaque ville a plusieurs gares, et une liste est pour aller d'une gare à une autre gare)
  - Trouver à quelle liste appartient un trajet donné
  - Réunir des listes
- Le tout, efficacement, puisque a priori à répéter beaucoup de fois.

→ Structures de données **de type Classe-Union (angl., Union-Find)**

- Ensembles discrets disjoints (appelés des classes)
- Opérations : classe (trouver l'ensemble dans lequel se trouve un élément) , union (union de deux ensembles)

Temps d'exécution selon implémentation

Complexité (nombre d'opérations) au pire des cas

pour k ensembles $S_1, S_2, \ldots, S_k$ de $n_1, n_2, \ldots, n_k$ éléments respectivement

Représentation pour chaque classe (*implémentation*)	Classe(x)*	Union(S_1,S_2)	Idée pour Classe
Table	*cst*	$O(n_2)$**	chercher table[1]***
Table triée	*cst*	$O(n_1+n_2)$**	chercher table[1]***
liste chaînée	$O(n_{Classe(x)})$****	*cst***	remonter jusqu'au début de la classe
table de hachage	*cst***** (en moyenne)	$O(n_2)$**	remonter
Objectif	$O(\log n_{Classe(x)})$	*cst*	

en moyenne

**en supposant que l'on ait un pointeur sur x*

***assume que S2 est inséré dans S1*

**** l'identité du tableau est stockée dans chaque élément*

***** le numéro de la classe est stocké dans le 1er élément de la classe*

Trajets SNCF : deux pas plus loin

- Et si les listes de trajets sont dynamiques, et doivent être consultées de nombreuses fois ?
  - Ajouter un trajet à l'ensemble de trajets
  - Enlever un trajet de l'ensemble (après l'avoir cherché)
  - Vérifier s'il reste des éléments dans l'ensemble
  - Calculer le minimum des temps de trajet
- Le tout, efficacement, puisque a priori à répéter beaucoup de fois.

→ Structures de données **de type dynamique**

- Un ensemble discret
- Opérations : ajouter, enlever, chercher, calculer min, calculer max …

Temps d'exécution selon implémentation

Complexité (nombre d'opérations) au pire des cas

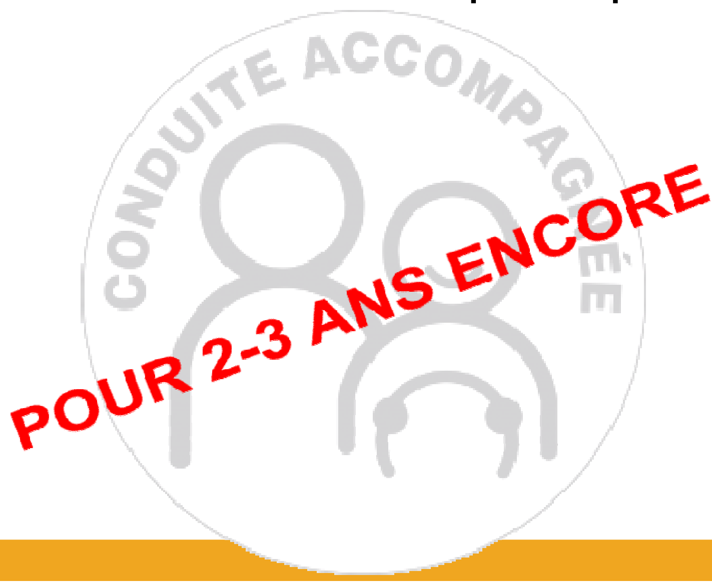
implémentation	Tester Ens. vide	Ajouter/ Enlever	Chercher un élém.	Calcul Min/max
table	*cst*	*cst*/O(n)*	*O(n)*	*O(n)*
table triée	*cst*	*O(n)*	*O(log n)*	*cst*
liste chaînée	*cst*	*cst**	*O(n)*	*O(n)*
table de hachage	*O(B)*	*cst*	*cst*	*O(B)*
Objectif	*cst*	*O(log n)*	*O(log n)*	*O(log n)*

en moyenne

n nombre d 'éléments
B > *n* taille de la table de hachage
**sans le test d 'appartenance*

Conclusions sur l'efficacité d'une solution

- Elle doit être toujours visée.
- Elle dépend le plus souvent de l'implémentation (structures de données utilisées)
- Une information bien structurée (listes chaînées, piles, files, tableaux mais aussi arbres et graphes de plusieurs types) est plus efficace qu'un stockage brut.
- … mais elle nécessite de maîtriser des aspects plus poussés de l'algorithmique.
- D'où :



Plan du cours

- Arbres et arbres binaires (rappels ?)
- Structures Classe-Union
- Arbres binaires de recherche
- Arbres équilibrés
- Arbres rouges et noirs
- Algorithmes gloutons
- Le codage de Huffman
- Arbres recouvrants d’un graphe