Algorithmique

Structures de données

Hugues Talbot

Département A2SI
ESIEE

22/03/2005 / ISBS

university-logo

Plan

Transparents très fortement inspirés du cours de Jean-Jacques Levy.
http://www.enseignement.polytechnique.fr/profs/informatique/Jean-Jacques.Levy/
Sujet présenté de façon exhaustive par Donald E. Knuth
“The Art of Computer Programming” (1973).

Concepts de structure de données

Acceder et manipuler des données efficacement

- On apprend toujours aux étudiants à bien structurer leurs programmes – leur code.
- Grâce à la programmation orientée objet, on a plus récemment tendance à associer code et données dans un même cadre.
- En fait les questions de la manipulation, du stockage et de l'accès aux données d'un problème doivent être traités au même niveau que celui de l'algorithme lui-même.

Changement de perspective

Programmation orientée données

- Réflechir aux contraintes du problème.
- Proposer un algorithme
- Gerer l'accès et la manipulation de données
- Réflechir à une structure qui rende la solution
  - Plus simple ;
  - Plus rapide ;
  - Moins gourmande en mémoire.

Exemple de problème

Indexer un fichier

- Entrée: un fichier de longueur inconnue
- Sortie: une liste de tous les mots et de leur position(s) dans le ficher.
- Le nombre de mots est inconnu
- Il n'y a pas de limite sur la fréquence des mots.
- Le fichier est de longueur finie.

TD: tenter une approche sur ce problème

- Peut-on exhiber un algorithme pour résoudre ce problème (même inefficace au début)
- Qu'avons nous besoin de stocker ?
- Qu'avons nous besoin de retrouver ?

Exemple d'algorithme

En pseudo-Java

```
class IndexFile {
  public static void main(String args[]) {
    int  j = 0;
    StringList   wordlist;
    IntList      positionlist;
    String       word[];

    while (word != EOF) {
      word= readNextWord(file);
      if (word != EOF) { // end-of-file
        j++;
        if ((positionlist = wordlist.lookup(word)) == nil)
          positionlist = wordlist.insert(word);
        }
        positionlist.append(j);
      }
    }
    printIndex();
  }
```

Exemple d'algorithme d'indexage - 2

```
class IndexFile {
  void printIndex() {
    foreach word in wordlist {
      System.out.println(word);
      positionlist = wordlist.lookup(word)
      foreach i in positionlist {
        System.out.println(i);
      }
    }
  }
}
```

university-logo

De quoi avons nous eu besoin ?

Pour manipuler nos données

- D'accumuler des chaînes de caractères indéfiniment (append) ;
- D'insérer des chaînes de caractères dans une liste (insert) ;
- De faire des recherches de chaînes (lookup) ;

Ameliorer les tableaux

Les vecteurs

Attention: terminologie C++ : vector

- Croissent et décroissent à volonté ;
- Insertion, recherche lente ;
- Opération de pile à l'arrière (LIFO);
- Accès aléatoire optimal.

Une autre option

Les queues

C++: deque (buffer circulaire)

- Même opérations que les vecteurs ;
- En plus accès pile à l'avant (FIFO) ;
- Accès aléatoire rapide.

Les listes chaînées

Insertion et suppression

- Même opérations que les vecteurs sauf accès aléatoire ;
- Insertion et suprression rapides.

Implementation en Java

a voir

Complexité des opérations

opération	Accès aléatoire	Insertion/suppression	Pile avant	Pile arrière	Iteration
tableau	$O(1)$				Aleat.
vecteur	$O(1)$	$O(n)+$		$O(1)+$	Aleat.
queue	$O(1)$	$O(n)$	$O(1)$	$O(1)$	Aleat.
liste		$O(1)$	$O(1)$	$O(1)$	Bidir.

Types de données abstraits (ADT)

- Pile (LIFO/PEPS)
- File d'attente (FIFO/PEDS)
- File à priorité (Priority Queue, tas)
- Dictionnaires (map)
- Ensembles

Types de données spécialisées

- Arbres binaires
- Tableau hashé (Hashtable)
- Skiplist
- Etc...

university-logo

Pile – Stack

Aussi appelée PADS ou LIFO.

- Restriction des opérations d'une liste à sa *tête*
- On empile des données et on les dépile seulement.
  - `Push()` == insertion à une extrémité
  - `Pop()` == extraction/suppression à la *même* extrémité.
- Mise en oeuvre très simplifiée, soit à partir d'une liste soit à partir d'un vecteur (ou même un tableau).

Q: quelle extrémité doit on utiliser si on se base sur un vecteur?

Exemple d'appliation : la calculatrice

- Soit l'expression `5 * (((9+8) * (4 * 6)) + 7)`
- On empile les nombres
- On dépile lorsqu'une parenthèse se ferme et on execute l'opération
- On empile le résulat.
- Lorsque la pile ne contient plus qu'un seul élément c'est la solution.

Solution possible

```
class ManualCalc {
        void compute() {
        push(5);
        push(9);
        push(8);
        push(pop()+pop());
        push(4);
        push(6);
        push(pop() * pop());
        push(pop() * pop());
        push(7);
        push(pop() + pop());
        push(pop() * pop());
        System.out.print(pop());
    }
}
```

university-logo

Notation polonaise inversée

- La façon standard d'écrire les expressions est dite "infixée".
- On peut réecrire toute expression infixée en une notation telle que les opérandes apparaissent d'abord et les operateurs aprés: exemple:
  `5 9 8 + 4 6 * * 7 + *.`
- Notation dite "postfixée" ou *Polonaise inverse*, popularisée par les calculateurs HP.
- Conversion infixée / postfixée facile, avec une pile

Conversion de notation

```
class InfixtoPostfix {
    void convert(File input) {
        char c;
        for (stackinit() ; read(input, c) != EOF ; ) {
            if (c == ')') System.out.print(pop());
            if (c == '+') push(c);
            if (c == '*') push(c);
            while (c >= '0' && c <= '9') {
                System.out.print(c);
                read(input, c);
            }
            if (c != '(') System.out.print("␣");
        }
        System.out.println("");
    }
}
```

Intérêt de la conversion en postfixé

- La notation postfixée n'a pas besoin de parenthèse et n'est pourtant pas ambiguüe: exemples:
- `((5*9)+8)*((4*6)+7)`
- `(5*((9+8)*(4*6))+7)`
- Evaluation de l'expression postfixée est très facile.

Evaluation de la notation postfixée

Encore à l'aide d'une pile !

```
class EvaluatePostfix {
    void evaluate(File input) {
        char c ; int x;
        for (stackinit() ; read(input, c) != EOF ; ) {
            x = 0;
            if (c == '+') x = pop() + pop();
            if (c == '*') x = pop() * pop();
            while (c >= '0' && c <= '9') {
                x = 10 * x + (c-'0')
                read(input, c);
            }
            push(x);
        }
        System.out.println(x);
    }
}
```

Utilité des piles

- Très simple à mettre en oeuvre et à utiliser
- Peuvent simplifier beaucoup de problèmes.

File d'attente – Queue

Aussi appelée PAPS ou FIFO.

- Très semblable à la pile, mais on doit utiliser et la tête et la queue de la structure.
- On empile des données et on les dépile seulement.
  - `Put()` == insertion à une extrémité
  - `Get()` == extraction/suppression à *l'autre* extrémité.
- Mise en oeuvre très simplifiée, soit à partir d'une liste soit à partir d'un vecteur (ou même un tableau).

Q: comment faire si on se base sur un vecteur?

File à deux extrémités – deque

- Combinaison d'une pile et d'une file d'attente.
- On peut insérer et supprimer/extraire aux deux extrémit'es:
  - `push_front();`
  - `pop_front();`
  - `push_back();`
  - `pop_back();`
- Pas beaucoup plus cher à mettre en oeuvre qu'une file d'attente
- Plus flexible.

university-logo

Type de données abstraits – ADT

- On ne s'occupe pas de détails de la mise en oeuvre
- Séparation du "concept" et de la fonctionnalité
- Moyen d'organiser la programmation à grande échelle
- Tous les langages récents proposent une mise en oeuvre des ADT prête à l'emploi : Java, C#, C++, Perl, Python, etc. Ce n'est pas le cas de C !
- Extrèmement important de savoir utiliser ces structures pour programmer efficacement.

Définition

- Sous ensemble des graphes
- Terminologie des graphes: sommets, arêtes.
- Terminologie des arbres: noeuds, racine, chemins, (grand-)parent, enfants, feuilles / sommets terminaux, branches/sommets internes, sous-arbre, niveaux, hauteur, longueur de chemin.
- Un arbre peut-être ordonné ou non.
- Dans un arbre chaque sommet peut avoir soit un nombre quelconque de descendants, soit un nombre pré-determiné, exemple: arbres binaires.
- Un arbre binaire peut être vide, rempli ou complet.

university-logo

Propriétés

- Il n'y a qu'un seul chemin entre deux sommets quelconques d'un arbre
- Un arbre avec N sommets a $N - 1$ arêtes
- Un arbre binaire avec N sommets internes possède $N + 1$ feuilles
- Le chemin externe d'un arbre binaire avec N sommets internes est $2N + L$, I étant la longueur interne de l'arbre.
- La hauteur d'un arbre binaire plein est approximativement $\log_2 N$.

Représentation des arbres binaries

- Comme une liste chaînée, mais avec 2 liens par structure.
-

Exemple d'élément de liste chaînée

```
public class listNode{
    protected Object data;
    protected listNode next;

    public listNode(){
        next = null;
        data = null;
    }
    public listNode(Object d, listNode n){
        data = d;
        next = n;
    }
    public void setNext(listNode n){
        next = n;
    }
    public void setData(Object d){
        data = d;
    }
    public listNode getNext(){
        return next;
    }
    public Object getData(){
        return data;
    }
 }
```

Exemple d'élément d'arbre binaire

```
public class btreeNode{
    protected Object data, key;
    protected btreeNode left,right;

    public btreeNode(){
        data = null;
        left = right = null;
    }
    public btreeNode(Object d){
        data = d;
        left = right = null;
    }
    public void setLeft(btreeNode l){
        left = l;
    }
    public void setRight(btreeNode r){
        right = r;
    }
    public void setData(Object d){
        data = d;
    }
    public void setKey(Object k) {
        key = k;
    }
    public btreeNode getLeft(){
        return left;
    }
    public btreeNode getRight(){
        return right;
    }
    public Object getData(){
```

Exemple d'application: arbre d'analyse – parsing tree

- Soit l'expression `A * (((B+C)*(D*E))+F)`
- On peut représenter cette expression de la façon suivante:
  - On transforme l'expression en infixée : `A B C + D E * * F + *`
  - on scanne l'expression infixée de gauche à droite
  - les opérandes forment un élément d'arbre sans enfants qui sont simplement empilées
  - les opérateurs forment un élément d'arbre. L'enfant de gauche et de droite sont dépilés.
  - On continue jusqu'à la fin, on obtient un arbre qui represente l'expression, dont les operandes forment les feuilles et les opérateurs les branches.
- Cette façon de procéder est la base des "scanners", indispensables pour la technologie des compilateurs, interpréteurs, etc.

Code pour l'analyse syntaxique

[fragile]

```
public class Scanner{
    void scannexp(File input) {
        for (stackinit() ; read(input, c) ; ) {
            x = new btreeNode();
            x.setData(c);
            if (x == '+' || c == '*') {
                x.setRight(pop());
                x.setLeft(pop());
            }
            push(x);
        }
    }
}
```

Traversée d'un arbre binaire

- Une fois qu'un arbre binaire a été construit, en général on veut en faire quelque-chose, en particulier le *traverser*.
- Pour une liste c'est facile, pour un arbre on a plusieurs choix.
  - En *pré-ordre* : méthode récursive: on visite la racine, puis le sous-arbre de gauche, puis le sous-arbre de droite (permet d'exprimer l'arbre précédent en infixée).
  - Dans l'ordre: on visite d'abord le sous-arbre de gauche, puis la racine, puis le sous-arbre de droite (permet le tri).
  - En *post-ordre* : visite d'abord le sous-arbre de gauche, puis le sous-arbre de droite, puis la racine (permet d'exprimer l'arbre précédent en post-fixée).
  - Par niveau: de haut en bas et de gauche à droite.

Mise en oeuvre des la traversée ordonnées

```
public class Traverse{
    void preorder(Tree T) {
        btreeNote root=T.Root(), t;
        push(root);
        while (!stackempty()) {
            t = pop(); visit(t);
            if (t.getRight()) push(t.getRight());
            if (t.getLeft()) push(t.getLeft());
        }
    }
}
```

Modifications triviales pour les autres traversées ordonnées.

Mise en oeuvre de la traversée par niveaux

Même chose, mais cette fois-ci avec une file d'attente:

```
public class Traverse{
    void level(Tree T) {
        btreeNote root=T.Root(), t;
        put(root);
        while (!queueempty()) {
            t = get(); visit(t);
            if (t.getRight()) put(t.getRight());
            if (t.getLeft()) put(t.getLeft());
        }
    }
}
```

university-log

Traversée recursive

```
public class Traverse{
    void rectrav(btreeNode t) {
        if (t) {
            rectrav(t.getLeft());
            visit(t);
            rectrav(t.getright());
        }
    }
}
```

Dictionnaires

- On cherche à résoudre le problème suivant:

```
public class Exemple{
    void level(Tree T) {
    SomeStruct a;
    Object     cle, o, p;

    // Entr\'ee dans un dictionnaire
    a.set(cle, o); // equiv: a[cle] = o;
    // recherche dans un dictionnaire
    p = a.get(cle); // equiv: p = a[cle];
}
```

- Note: on recherche les données par clé.
- Q: comment résoudre ce problème ?
- Solution plus simple : un simple tableau, ou une liste.
- Autre façon plus efficace : utiliser un arbre.

Arbre de recherche

- Dans un arbre binaire, on met les clés plus petites à gauche et les clés plus grandes à droite.
- L'idée est assez simple : on utilise un arbre binaire (structure connue) pour maintenir une structure triée. La recherche s'effecture naturellement.
- La structure d'arbre binaire permet de mettre en oeuvre la stratégie "diviser pour regner" clairement et simplement.

Code pour la recherche dans un arbre binaire

```
public class BinTree {
    private Tree T;

    Object search(Object key) {
        btreeNote root=T.Root(), t;

        t = root;
        // attention aux recherches infructueuses
        while (key != t.getKey()) {
            if (key < t.getKey())
                t = t.getLeft();
            else
                t = t.getRight();
        }

        return(t.getData());
```

Pb1: recherches infructueuses

- Dans le cas présent une recherche infructueuse va générer une exception.
- Pour éviter cela on peut avoir un noeud special `z` vers lequels pointe tous les enfants non attribués.
- Dans ce cas une recherche infructueuse retournera simplement `null`.
- Pour rester concis, dans la suite on garde la version avec exceptions.

university-logo

Pb2: Maintenir la structure d'arbre

- Bien entendu pour que ça marche, il faut un arbre binaire pour commencer.
- Il faut donc être capable d'insérer des données dans un arbre binaire tout en maintenant la structure.
- Pour cela, on fait une recherche infructueuse forcée, et on insère à cet endroit.
- Une recherche infructueuse forçee signifie que si l'objet est déjà dans l'arbre, alors on l'insère une deuxième fois.

Insertion dans un arbre binaire

```
public class BinTree {
    private Tree T;

    void insert(Object key, Object data) {
        btreeNote root=T.Root(), t, x;

        t = root;
        x = t.getRight();
        // recherche d\'elib\'erement infructueuse
        while (x != null) {
            t = x; // remember last non-null
            if (key < x.getKey())
                x = x.getLeft();
            else
                x = x.getRight();
        }
        x = new TreeNode;
        x.setKey(key); x.setData(data) ;
```

Tri d'un arbre binaire

Note: le tri du contenu d'un arbre binaire est preque gratuit:

```
public class BinTree {
    private Tree T;

    void treesort (void) {
        treesortr(T.Root());
    }

    void treesortr(TreeNode n) {
        if (n != null) {
            treesortr(n.getLeft());
            System.Console.Println(n.getData());
            treesortr(n.getRight());
        }
    }
}
```

Complexité

- Dans le cas idéal, où l'arbre est *équilibré*, recherche et insertion sont logarithmiques.
- Dans le pire des cas, où l'arbre est tout d'un côté ou de l'autre, recherche et insertion sont linéaires (comme un tableau).

Autres remarques

- La délétion est plus complexe à mettre en oeuvre (Q: pourquoi?). Souvent on ne le fait pas, on utilise une technique dite de “deletion paresseuse”, où on marque simplement un noeud comme effacé, et on reconstruit l’arbre de temps en temps sans ces noeuds.
- Il est impossible de garantir qu’un arbre binaire simple reste équilibré. Pour pallier ce défaut, on utilise des arbres différents:
  - Les arbres 2-3-4 (qui contienne 1, 2, ou 3 clé pour chaque noeud)
  - Les arbres rouges et noirs = Red-Black trees (arbres binaires avec un bit de plus d’information)
  - Les arbres AVL (AVL-trees)
  - Les B-trees, extension des arbres 2-3-4, qui forment la base de toutes les bases de données.
- L’étude de ces arbres prendrait trop de temps.

Tâches à priorité

Dans beaucoup d'applications, des structures avec clé doivent être traîtées dans un certain ordre, mais:

- pas nécessairement toutes d'un coup : on peut vouloir collecter certaines données, puis traîter certaines, puis en collecter encore, etc.
- on n'a pas forcément besoin de trier toutes les données complêtement. On peut par exemple ne vouloir que la donnée avec la clé la plus élevée à chaque itération.

Exemple de tels problèmes

Les problèmes qui requièrent une telle structure sont en fait nombreux.

- Compression de fichier (codage de Huffman)
- Allocation mémoire (retourner le plus grand espace vide)
- Certaines recherches sur graphes
- De nombreux algorithmes d'analyse d'image.

Quelle structure utiliser ?

Pour résoudre ce problème on pourrait utiliser un tableau, un vecteur ou une liste ou un arbre binaire, mais aucune de ces structures n'est idéale:

- Les structures "plates" ne permettent pas une recherche efficace
- Les arbres binaires simples ne garantissent pas une recherche efficace non plus.

On introduit la structure de file à priorité.

Opérations nécessaires aux files à priorité

on veux pouvoir effectuer les opérations suivante efficacement:

- Construction d'une file à partir de données initiales ;
- Insertion d'une donnée à clé quelconque ;
- Rechercher ou Retirer l'élément le plus grand (ou le plus petit) ;
- Remplacement de l'élément le plus grand par un autre ;
- Changement de priorité;
- Deletion d'un élément.

Note: type de donnée abstrait.
Certaines de ces opérations peuvent être concues comme des opération composées, par exemple remplacement = insertion puis retirer, mais peuvent peut-être se mettre en oeuvre de façon plus efficace en tant que telles.

Mise en oeuvre élémentaire

En utilisant un simple tableau:

```
public class SimplePQ {
    private Object T[];
    int N

    void construct(int M, Object D[]) {
      T = new Object[M];
      N = M
      int i;
      for (i = 0 ; i < N ; ++i)
          T[i] = D[i];
    }

    void insert (Object v) {
        T[++N] = v;
    }

    Object remove() {
```

Critique de la mise en oeuvre élémentaire

- La recherche est linéaire en $\mathbb{N}$.
- Au lieu d'un vecteur on pourrait utiliser une liste ordonnée, mais la création demanderait un tri, et l'insertion serait linéaire.
- Q: si au lieu d'une liste ordonnée on utilise un vecteur ordonné, quelle est l'op'eration la plus chère ?

Q: toute structure de file à priorité induit une méthode de tri. Comment ? Quel tri correspond à l'utilisation d'un vecteur ? d'une liste ordonnée ?

Structure de tas

- On utilise un arbre binaire
- Condition du tas: la clé dans chacun des noeuds doit être plus grande ou égale (ou plus petite ou égale) que la clé dans chacun de ses enfants. Note: la racine contient donc l'élément le plus grand.
- Un tas est toujours équilibré
- Un tas peut être représenté simplement par un tableau.

1	2	3	4	5	6	7	8	9	10	11	12
X	T	O	G	S	M	N	A	E	R	A	I

  Les enfants de `j` sont en position `2j` et `2j+1`. Le parent de `i` est à `i/2`.
- Il y a juste assez de flexibilité dans cet arrangement pour la file à priorité.
- pour N entrées il y a $\log_2 N$ générations. Trivial de parcourir un tas.

Maintenir la structure de tas

- L'idée dans l'utilisation des tas est de faire une modification de la structure, puis d'apporter des changements simples pour maintenir la structure.
- Certaines opérations manipule la structure par la racine, d'autres par la fin.

Insertion dans un tas

```
public class HeapPQ {
    private Object T[];
    int N;

    void upheap(int k) {
        Object v;
        v = T[k] ; T[1].setKey(INT_MAX);
        while (T[k/2].getKey() <= v.getKey()) {
            T[k] = T[k/2] ; k = k/2;
        }
        T[k] = v;
    }
    void insert(Object v) {
        T[++N] = v;
        upheap(N);
    }
}
```

Retirer l'élément le plus grand

```
public class HeapPQ {
    private Object T[];
    int N;

    void downheap(int k) {
        Object v;
        int j;
        v = T[k] ;
        while (k <= N/2) {
            j = 2*k;
            if (j < N && (T[j].getKey() < T[j+1].getKey(
                j++;
            if (v.getKey() > T[j].getKey())
                break;
            T[k] = T[j] ; k = j;
        }
        T[k] = v;
    }
```

Heapsort

Tas = heap en anglais. Le tri correspondant à la structure de tas est `heapsort`. Aussi efficace que QuickSort.

Conclusion provisiore

- Une grosse partie (30-50%) de l'algorithmique consiste en des façons triviales et non triviales de trier et de rechercher.
- Parmi le reste :
  - Traitement de chaînes
  - Algorithmes géométriques (intersections de droites, etc)
  - Graphes
  - Algorithmes mathématiques (nombres aléatoires, arithmétique, algèbre linéaire, FFT, etc)
  - Programmation dynamique et linéaire
  - Problèmes difficiles
- *Presque tous* utilisent des structures de données adéquates.

university-logo

Problèmes plus ou moins faciles

- Certains problèmes sont plus faciles que d'autres, au sens où il existe des algorithmes plus ou moins efficaces pour les résoudre
  - Accès aléatoire dans un vecteur: $O(1)$.
  - Maintien du plus petit (ou plus grand) élément dans un tas: $O(\log N)$
  - Recherche du plus petit élément dans un vecteur : $O(N)$
  - Tri général: $O(N \log N)$
  - Multiplication élémentaire de deux nombres : $O(N^2)$
  - Multiplication élémentaire de deux matrices : $O(N^3)$

Problèmes plus ou moins faciles (suite)

- Il existe des problèmes plus difficiles. Meilleurs algorithmes connus pour les problèmes suivants:
  - Plus grand convexe inscrit dans un polygone : $O(N^7)$
  - `SUBSET-SUM` : $O(2^{(}N/2)$?
  - Meilleur coup aux échecs (ou au go): $O(p^N)$
  - Vérité des expressions dans l'arithmetique de Preburger : $\Omega(2^{2^{cN}})$
  - Problme de l'arrêt : pas de solution !.

Simplification des classes de difficultés

- On considère la classe des problèmes qui ont une solution en $O(kN^p)$, k et p constantes.
- Ces problèmes sont dits polynomiaux, et appartiennent à la classe **P**.
- Tous les problèmes plus difficiles appartiennent à la classe **Non-P**.
- En quelque sorte, les problèmes dans **P** sont “faciles” et les problèmes dans **non-P** sont “difficiles”.

Simplification ignore certaines choses

- Une solution à un problème dans **P** en $O(N^2)$ n'est pas aussi utile qu'en $O(N \log N)$. (exemple : FFT)
- Constantes, détails de mise en oeuvre sont tous ignorés.
- On exige une solution exacte. Parfois une solution approchée est tout à fait suffisante, exemple: Kasparov battu par Deep Blue.
- Cette distinction a surtout une valeur mathématique plutôt que pratique
- Utile néanmoins, ne serais-ce que pour reconnaître les problèmes vraiment difficile

university-logo

D'autres difficultés surgissent

- Il existe des problèmes clairement dans **P**, par exemple le tri général.
- Il existe des problèmes clairement dans **non-P**, par exemple le problème des échecs.
- Mais il existe un vaste nombre de problèmes dont on ne sait pas s'ils appartiennent à l'une ou l'autre classe. Il ne sont que *apparemment difficiles*, c-à-d on ne connait pas d'algorithme efficace pour les résoudre mais on n'a pas non plus réussi à démontrer qu'un tel algorithme n'existe pas.
  - `SUBSET-SUM`
  - Cycle Hamiltonien dans un graphe
  - Allocation de tâche dans un système d'exploitation
  - factorisation d'un entier

Les chercheurs trouvent parfois

Parce qu'on n'a pas réussi à trouver un algorithme efficace pour le moment ne veut pas dire qu'il n'en existe pas.

- Les algorithmes efficaces sur les graphes datent des années 50-60 pour la plupart.
- Certains problèmes n'ont été résolus que très récemment, par exemple le problème du postier Chinois.
- Un algorithme polynomial pour décidé si un entier est premier ou non à été trouvé pour la première fois en 2002.
- Il existe un algorithme quantique pour factoriser un entier en temps polynomial (algorithme de Shor).

Il est facile de rendre un problème facile difficile

- Souvent un problème facile se transforme en problème apparemment difficile avec peu de changements:
  - Cycle Eulerien, facile ; cycle Hamiltonien, difficile.
  - Chemin minimal, facile ; chemin maximal sans cycle, difficile.

Comment progresser ?

Il faut formaliser un peu, préciser une classe de problème géneraux contenant des problèmes apparemment difficiles.

- Problème décisionnels.
- Oracle, qui donne une solution si une existe, verifiable en temps polynomal.
- Classe des problèmes **NP**, non-deterministes polynomiaux. *ne pas confondre avec* **non-P**.