

Gilles Kahn (1946 – 2006)



Informaticien français. Spécialiste des langages de programmation et des preuves sur ordinateur, il s'est intéressé à la sémantique des programmes, c'est-à-dire au sens de ce qui est calculé. Ses travaux permettent de manipuler les programmes eux-mêmes pour étudier leurs propriétés. Il a largement participé à faire reconnaître en France l'informatique comme une discipline à part entière.

Bref, avec Gilles Kahn, les programmes prennent du sens !

Source : [1]

UE09 – Algorithmique Comparaison d'algorithmes polynomiaux

Nicolas HENDRIKX & Simon LIÉNARDY

Sommaire du thème 1

1. Un premier exercice
2. Définitions
3. Exemples d'algorithme
4. Caractéristiques (domaine, exactitude, efficacité)
5. La notion de complexité
6. Comparaison d'algorithmes polynomiaux
7. Approches de résolution

Complexité - Rappels

- On s'intéresse au comportement des algorithmes pour des tailles croissantes des données (comportement **asymptotique** = pour une taille de données qui tend vers l'infini)
- Pour exprimer les complexités théoriques, on utilise la notation \mathcal{O} (« grand O »). Le but est de pouvoir évaluer son **ordre de croissance** en trouvant une fonction f (proche) qui le borne.
- Analyse dans le **pire des cas** + éventuellement le **cas moyen** quand ce sera possible.
- Règles de calculs, la plupart du temps « mécaniques ». Calculer le **nombre de tours** d'une boucle est parfois un peu + difficile

Algorithmes polynomiaux ?

On distingue :

- **Les problèmes « faciles »** : on connaît des algorithmes de complexité polynomiale pour les résoudre.
 - On les qualifie souvent ces algorithmes d' « **efficaces** ».
 - Même si $x^{17} + x^9 + x^3$ est aussi un polynôme...
- **Les problèmes « difficiles »** : il n'existe pas d'algorithmes polynomiaux pour les résoudre dans le pire des cas.
 - Les algorithmes qui les résolvent sont qualifiés d' « **inefficaces** ».

Exemple de problème difficile

- Soit un voyageur de commerce, qui doit visiter V villes. Trouver le circuit de villes qui minimise la longueur de son voyage.
- Algorithme : tester toutes les combinaisons possibles et retenir la distance minimum
 - Exemple avec $V = 4$: Ath, Belœil, Cambron, Dour.
 - Nombres de circuits possibles ?  3 min

Sommaire de la séance

1. Recherche séquentielle
2. Recherche dichotomique
3. Compléments sur la récursivité

Exercice

Soit le problème suivant :

Précondition : `int[]` entiers, un tableau d'entiers, `v` une valeur entière

Postcondition : Soit `r` le résultat.

`r` est dans `[-1 .. entiers.length[`

ET `r != -1` \Rightarrow `entiers[r] = v`

Code + Complexité (spatiale / temporelle) ? ⏳ 10 min

Recherche séquentielle – sans spoiler



Code : à trouver vous-même !

Complexité spatiale ?

- Quelle est la **taille** du problème ?
- Que se passe-t-il si elle augmente ?

Complexité temporelle : on s'intéresse aux opérations élémentaires.

Quel est le pire des cas ?

Quel est le meilleur des cas ?

Quel est le cas moyen ?

Recherche : tenir compte du contexte

Souvent, on peut **améliorer l'efficacité** d'un algorithme lorsqu'on dispose **d'hypothèses** sur les données.

Dans notre exemple : on vous dis maintenant que le tableau est **trié par ordre croissant**. On peut donc en tenir compte !

Nouvelle condition d'arrêt : on est arrivé à la fin du tableau ou bien sur un élément qui

Séquentielle dans un tableau trié

```
int rechercheSequentielle(Object[] tab, Object o) {  
    int i = 0  
    while(i < tab.length && tab[i].compareTo(o) <= 0){  
        if (tab[i].equals(o))  
            return i;  
        i++;  
    }  
    return -1;  
}
```

Complexité temporelle :
Quel est le pire des cas ?
Quel est le meilleur des cas ?
Quel est le cas moyen ?

Recherche dans un tableau trié

```
/**  
 * Même Postcondition que tout à l'heure !  
 * Que devient la précondition ?  
 *  
 */  
int rechDichotomique(int[] tab, int x)  
    return dichotomie(tab, x, 0,tab.length - 1);  
}
```

Exercice :

À l'aide du slide suivant, faire un schéma des entrées (tab, début, fin). Y inclure la variable locale « milieu » et expliquer le fonctionnement de l'algorithme. ⏳ 5 min

Recherche dans un tableau trié

```
int dichotomie(int[] tab, int x, int debut, int fin) {  
    if (deb > fin )  
        return -1;  
  
    int milieu = debut + (fin - debut) / 2;  
    int e = tab[milieu];  
    if (x==e)  
        return milieu;  
    else{  
        if (x < e)  
            return dichotomie(tab, x, debut, milieu - 1);  
        else  
            return dichotomie(tab, x, milieu + 1, fin);  
    }  
}
```

Wait what! Ce truc est récursif !

- Rappel de mathématiques 😊 :
 - Un algorithme récursif s'appelle **lui-même**.
 - Comment trouver un algorithme récursif ?
 - Tout se passe lors de la phase préparatoire !
 - Il faut identifier un motif dans le problème à résoudre.
- Définir récursivement un **concept** c'est dire :
Ce **concept**, c'est la *combinaison* de quelque chose et une instance **plus simple** de ce **concept**.
- Il suffit de définir ce qu'on entend par « *combinaison* de quelque chose » et « **plus simple** »

Récursivité

Exemple :

$$\text{factorielle}(n) = \text{factorielle}(n-1) \times n$$

Concept

Instance
plus simple

Combinaison

Il ne faut pas oublier le ou les cas de base :

$$\text{Ici, } \text{factorielle}(0) = 1$$

Récursivité et complexité : les contextes importent !

Petit détour par la factorielle :

```
public static int factorielle(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return n * factorielle(n-1);  
}
```

Complexité spatiale ? Voir [ici](#)

Complexité temporelle ? Nécessite d'analyser le nombre d'appels récursifs.

Récursivité et complexité

```
public static int fibonacci(int n) {  
    if (n < 2)  
        return n;  
    else  
        return fibonacci(n-1) + fibonacci(n-2);  
}
```

Exercice :

Dessiner les appels récursifs pour quelques n (*Conseil : n petits*).

En conclure que cette implémentation, c'est franchement de la  !

Quid de celle-ci ?

```
public static int fibonacci(int n) {  
    return fibonacci(n, 0, 1);  
}
```

```
public static int fibonacci(int n, int prev2, int prev1) {  
    if (n == 1) return prev1;  
    if (n == 0) return prev2;  
    return fibonacci(n-1, prev1, prev2 + prev1);  
}
```

Exercice :

Dessiner les appels à `fibonacci(6, 0, 1)`
Vérifier que vous obtenez bien 8

Java et la récursivité

Dans l'exemple précédent, l'appel récursif est la dernière chose effectuée avant le **return**.

On appelle cela la **récursivité terminale** (*tail recursion*)

Comme vous l'avez vu au cours de mathématiques, tout algorithme récursif peut être transformé en boucle et vice-versa.

Certains compilateurs peuvent optimiser pour vous un code récursif terminal mais ce n'est malheureusement pas le cas du compilateur Java...

En Java, les algorithmes qui tirent partie de la récursivité ont été développés en s'assurant que les appels récursifs sont **limités**.

Retour sur la recherche dichotomique

Complexité temporelle ?

$O(\log(\text{fin} - \text{debut})) \rightarrow \text{Pourquoi ?}$

Complexité spatiale ?

$O(\text{tab.length})$

Sommaire du thème 1

1. Un premier exercice
2. Définitions
3. Exemples d'algorithme
4. Caractéristiques (domaine, exactitude, efficacité)
5. La notion de complexité
6. Comparaison d'algorithmes polynomiaux
7. Approches de résolution

Approches de résolution

Quelles sont les principales approches de résolution d'un problème ?

→ Voir syllabus, pages 11 et suivantes.

- Approche **incrémentale** : à chaque étape de l'algorithme, on ajoute un élément à la solution
 - Exemples ?
- Approche **exhaustive** : on teste toutes les solutions possibles !
(potentiellement un nombre infini...)
 - Exemple ?

Approches de résolution

- **Diviser pour régner :**
 1. **divise** le problème en un certain nombre de sous-problèmes,
 2. **règne** sur les sous-problèmes en les résolvant directement, ou de manière récursive,
 3. **recombine** les solutions obtenues des sous-problèmes pour en déduire la solution du problème de départ.

Exemple ? Voir la séance suivante !
- **Algorithmes gloutons** : choix optimum local, en espérant obtenir un optimum global.
 - Exemple ? Problème du rendu de monnaie (rendre la monnaie avec le moins de pièces possibles) → On commence par rendre la pièce la plus grande, plus petite que ce qu'il faut rendre.

Approches de résolution

- **Programmation dynamique** : on divise aussi le problème en sous-problèmes (comme dans la stratégie diviser pour régner) mais cette méthode permet de gérer le cas des sous-problèmes qui se recoupent en mémorisant la solution des sous-problèmes déjà rencontrés dans la résolution. *NB : le mot programmation fait référence au stockage des résultats intermédiaires dans une table.*
 - Exemple : On espère avoir assez de temps à la fin du cours pour vous en montrer un ;-)

Approches de résolution

- Approche **(h)euristique** (ou approchée): fournir une solution approchée (pas exacte) ou pas nécessairement optimale plutôt qu'une solution exacte/optimale nécessitant un algorithme de complexité temporelle théorique exponentielle ou factorielle
 - Exemple : de nombreux problèmes d'optimisation fonctionnent ainsi. C'était le cas du Bin-Packing.

Sources

1. Maxime Amblard & Christine Leininger. (2019) *Famille « Algorithmes & programmation »*. <https://interstices.info/famille-algorithmes-programmation/> (Consulté le 27/01/24)