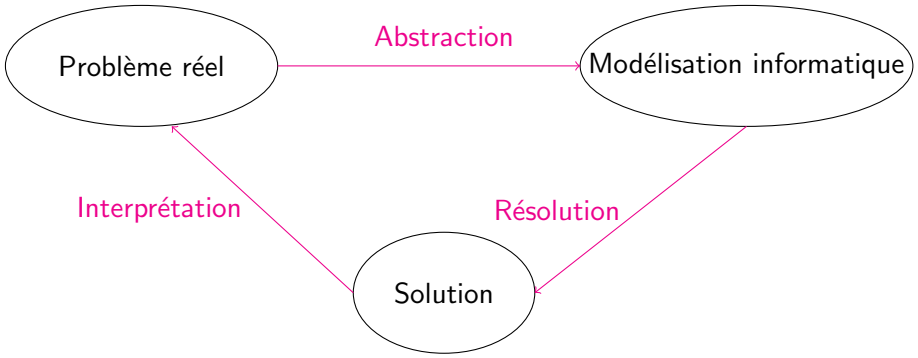
Algorithmique 1
Conception d'algorithmes combinatoires (suite)

Hajer Akid & Sandrine Julia

Semestre 4

Algorithmique



Comment concevoir des algorithmes efficaces ?

De nouveaux conseils

- abstraire, bien **modéliser** un problème réel
- ne pas tenir compte des détails non indispensables
- utiliser des objets algorithmiques
  - nombres, points, listes, ensembles, arbres, graphes *etc*
- ré-utiliser à bon escient des algoritmes connus
  - utiliser une solution connue comme sous-routine
  - adapter une solution connue à son problème
- soigner la représentation des données
  - choisir la **structure de données**
  - connaître toutes les SDD et leurs performances

Comment concevoir des algorithmes efficaces ? (suite)

- ajouter des informations
  - ajouter des références, des mots-clefs (*tags*)
  - calculer des empreintes (*hash*)
- mémoriser des solutions
  - ne pas faire plusieurs fois le même calcul
- être exigeant
  - principe d'économie
    - complexité en temps
    - complexité en espace
  - simplicité de la solution
  - lisibilité de l'algorithme

"Perhaps the most important principle for a good algorithm designer is to refuse to be content".

Savoir incontournable

Problèmes classiques

Il faut connaître les algorithmes qui les résolvent, leur complexité, les structures de données possibles, leurs variantes :

- parcours
- tris
- recherche d'un élément
  - recherche dichotomique dans un tableau trié (*binary search*)
  - recherche du k$^{\text{ème}}$ plus petit élément (*k-selection*)
- création, insertion, suppression, mise-à-jour dans toutes les SDD
- opérations arithmétiques (multiplication, division, pgcd *etc*)
- recherche de motifs (*pattern-matching*)
- algorithmique des graphes
- ...

Structures de données

Elémentaires

- liste
  - liste chaînée
  - liste doublement chaînée
  - liste chaînée circulaire
- tableau
  - trié ou pas
  - 1 ou + dimensions (matrices)
  - dynamiques ou pas
- arbre
  - arbre binaire
  - ABR
  - arbre binaire équilibré
    - AVL
    - *B*-arbres

Types de données abstraits

- pile
- file
- dèque *double-ended queue*
- queue de priorité
- tas binaire
  - tas binaire-min
  - tas binaire-max
- dictionnaire
- trie (arbre-préfixe)
- table de hachage
- *union-find*

Problème de la *k*-sélection

Données

- un tableau de n entiers
- un entier k inférieur ou égal à n

Problème

- trouver le k$^{\text{ème}}$ plus petit élément du tableau

Exemple

$k = 7$ et $T =$

10	0	4	6	0	2	3	5	6	8

Le 7$^{\text{ème}}$ élément est : **6**

Vérification :

T trié =

0	0	2	3	4	5	**6**	6	8	10

Problème de la *k*-sélection : un premier algorithme

Une première idée directement inspirée du tri par sélection

```
entier k_selection (entier T[], entier k){
   n ← longueur(T)
   pour (i ← 1; i ≤ k; i++) {
     // invariant : T trié des indices 1 à i−1
     imin ← i
     min ← T[i]
     pour (j ← i+1; j ≤ n; j++) {
       si (T[j] < min) {
         imin ← j
         min ← T[j]
       }
       echanger (T[i],T[imin])
     }
   }
   retourner T[k]
}
```

Complexité : $O(k.n)$

Problème de la k-sélection : 2ème et 3ème algorithmes

En triant tout le tableau efficacement :

```
entier k_selection (entier T[], entier k){
  T ← tri(T)
  retourner T[k]
}
```

Complexité : $O(n \log(n))$

On pourrait également utiliser un ABR :

- insertion des n éléments du tableau
- parcours infixe de longueur k
- on retourne le dernier élément

Complexité : $O(n \log(n))$

Problème de la k-sélection : 4ème et 5ème algorithmes

En utilisant un tas binaire ?

```
entier k_selection (entier T[], entier k){
  build_min_heap(T)
  pour (i ← 1; i < k; i++){
    min_heap_pop(T)
  }
  retourner T[1]
}
```

Complexité : $O(n + k \log(n))$

Faut-il mettre tous les éléments dans le tas ?

```
entier k_selection (entier T[], entier k){
  build_max_heap(T[1..k])
  pour (i ← k+1; i ≤ n ; i++) {
    si (T[i] < T[1]) {
      max_heap_pop(T)
      max_heap_push(T,T[i])
    }
    retourner T[1]
  }
}
```

Complexité : $O(k + (n-k)\log(k))$

Problème de la k-sélection : un algorithme encore meilleur

Algorithme BFPRT

- Inventé en 1972 par M. Blum, R. Floyd, V. Pratt, R. Rivest et R. Tarjan
- Aussi appelé l'algorithme de la **médiane des médianes**
- Nombre de comparaisons compris entre $1,5n$ et $5,43n$

Scénario

- on partitionne le tableau en sous-tableaux de longueur 5
- on les trie et on retorune leur médiane
- on recherche récursivement la médiane de la médiane en se ramenant à des tableaux de longueur 5
- la médiane des médianes sert de pivot pour séparer les plus petits des plus grands
- on continue récursivement du côté du $k^{\text{ème}}$ élément

Complexité : $O(n)$

Modélisation d'ensembles

On considère un ensemble d'éléments totalement ordonnés et on l'appelle l'univers $\mathcal{U}$.

Entrée : deux ensembles **finis** A et B inclus dans $\mathcal{U}$

Problème : donner des fonctions pour construire :

- l'union $A \cup B$
- l'intersection $A \cap B$
- la différence $A \setminus B$
- ...

ou des tests afin de décider :

- l'inclusion : $A \subseteq B$?
- l'égalité : $A = B$?
- ...

Représentation : on choisit les *listes chainées* ...

Même problème à traiter sur les tableaux en TD !

Union de 2 ensembles

Listes chaînées

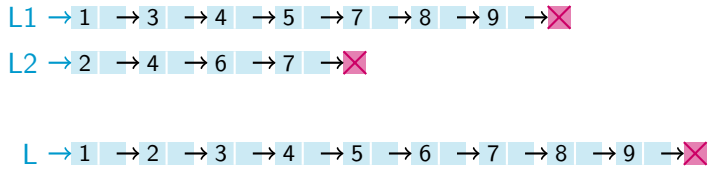
L'idée est de les maintenir **triées** pour faciliter les opérations même si l'ordre des éléments n'est pas significatif dans un ensemble.

On garde à l'idée qu'un ensemble ne contient **pas de doublons**.

Exemple

Prenons $A = \{7, 4, 8, 1, 3, 9, 5\}$ et $B = \{4, 7, 2, 6\}$, on veut construire l'union $A \cup B$.

Les trois ensembles sont respectivement representés par les listes L1, L2 et L :



Union de 2 ensembles en itératif

```
Union(liste_chaînee L1, liste_chaînee L2) {
  // antécédent: L1 et L2 listes triées représentant les ensembles E1 et E2
  AjouteEnQueue(entier d){ ... } // voir page d'après
  prem ← nil
  der ← nil
  tant que (L1 ≠ nil et L2 ≠ nil){
    si (donnee(L1) < donnee(L2)) {
      AjouteEnQueue(donnee(L1))
      L1 ← suivant(L1)
    }
    sinon si (donnee(L1) > donnee(L2)) {
      AjouteEnQueue(donnee(L2))
      L2 ← suivant(L2)
    }
    sinon { // égalité : on évite le doublon
      AjouteEnQueue(donnee(L1))
      L1 ← suivant(L1)
      L2 ← suivant(L2)
    }
  }
  tant que (L1 ≠ nil){
    AjouteEnQueue(donnee(L1))
    L1 ← suivant(L1)
  }
  tant que (L2 ≠ nil){
    AjouteEnQueue(donnee(L2))
    L2 ← suivant(L2)
  }
  retourner prem
}
```

Complexité : avec $n = card(E1) + card(E2)$, $T(n) = O(n)$

Ajout en queue (sous-fonction)

Cette version est une sous-fonction de la version itérative de Union(L1,L2) :

```
AjouteEnQueue(entier d){

  // allocation et mise-à-jour d'une nouvelle cellule
  new : pointeur sur nouveau(élément)
  donnee(new) ← d
  suivant(new) ← nil

  // ajout en queue
  si (prem = nil){
    prem ← new
    der ← new
  }
  sinon {
    suivant(der) ← new
    der ← suivant(der)
  }
}
```

Complexité : $O(1)$

Union de 2 ensembles en récursif

```
Union(liste_chaînee L1, liste_chaînee L2){
  // L1 et L2 listes triées modélisant les ensembles E1 et E2

  si (L1 = nil){
    retourner L2
  }
  si (L2 = nil){
    retourner L1
  }
  // L1 et L2 non vides

  si (donnee(L1) < donnee(L2)){
    retourner ajouteEnTête(donnee(L1),Union(suivant(L1),L2))
  }
  si (donnee(L1) > donnee(L2)){
    retourner ajouteEnTête(donnee(L2),Union(L1,suivant(L2)))
  }
  // égalité : on évite le doublon
  retourner ajouteEnTête(donnee(L1),Union(suivant(L1),suivant(L2)
  ))
  }
}
```

Complexité : avec $n = card(E1) + card(E2)$, $T(n) = T(n-1) + O(1) = O(n)$

Remarques

- la liste résultant de Union(L1,L2) est indépendante des listes L1 et L2 initiales
  - tous les éléments ont été recopiés
  - l'espace a été alloué par ajouteEnTête(d) ou ajouteEnQueue(d)

Il reste à écrire ajouteEnTête(d) en TD !

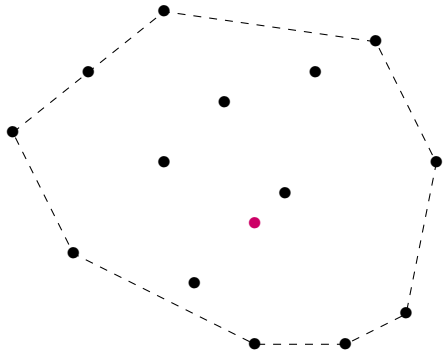
- le **tri-fusion** implanté sur les **listes chaînées** utilise l'algorithme Fusion(L1,L2) pour fusionner deux listes triées
  - le problème de l'union est similaire à celui de la fusion
  - *a priori* seul le cas d'égalité de données diffère
  - il y a cependant une différence majeure : la fusion doit être **sur place**
  - les algorithmes en seront donc un peu différents

Problème : enveloppe convexe

On considère les points de coordonnées entières dans le plan.

Entrée : un ensemble fini de points S

Problème : trouver leur enveloppe convexe

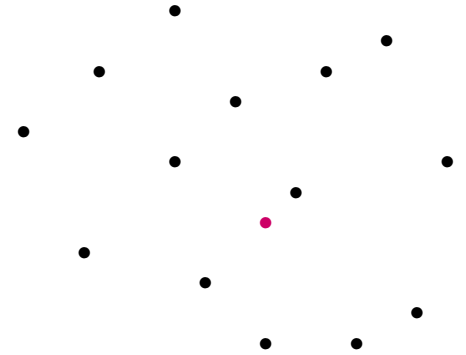


Problème : enveloppe convexe

On considère les points de coordonnées entières dans le plan.

Entrée : un ensemble fini de points S

Problème : trouver leur enveloppe convexe



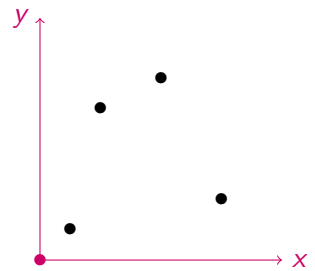
Problème : 2D-maxima (Front de Pareto)

Voici un nouveau problème, plus simple !

Plaçons-nous dans le quart de plan où les points sont de coordonnées positives :

Entrée : un ensemble fini de points S de coordonnées entières positives

Problème : trouver tous les points maxima de l'ensemble S



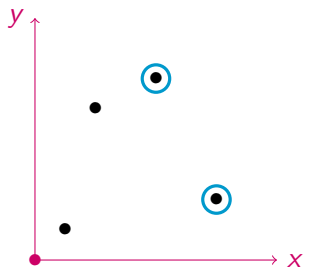
Problème : 2D-maxima (Front de Pareto)

Voici un nouveau problème, plus simple !

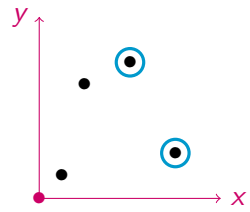
Plaçons-nous dans le quart de plan où les points sont de coordonnées positives :

Entrée : un ensemble fini de points S de coordonnées entières positives

Problème : trouver tous les points maxima de l'ensemble S



Problème : 2D-maxima



Definition

Dans le plan, un point (x_1, y_1) **domine** un autre point (x_2, y_2) si :

$$x_1 \geq x_2 \quad \textbf{et} \quad y_1 \geq y_2$$

Definition

Dans le plan, un point est un **maximum** s'il n'est dominé par aucun autre.

Algorithme 2D-maxima

Comment faire pour trouver les points maximaux ?

- doit-on trier les points de l'ensemble S ?
- si oui, dans quel ordre ?

- peut-on séparer efficacement les points dominés des autres ?
- si oui, selon quel fil-rouge ?

- et en quelle complexité ?
- est-ce possible en temps linéaire ?

Réponses en TD !

Tri par base des points

Trions dans l'**ordre cartésien** n points du plan à coordonnées entières **comprises entre 0 et 9** à l'aide de 2 tableaux de listes chaînées $L = [L_i]_{0 \leq i \leq 9}$ et $K = [K_i]_{0 \leq i \leq 9}$:

$(3,9)\ (4,2)\ (0,3)\ (5,8)\ (2,6)\ (2,0)\ (7,1)\ (9,3)\ (6,2)\ (8,3)\ (7,4)\ (0,2)\ (8,1)\ (5,8)\ (3,7)\ (4,8)$

- parcours des ordonnées
  ajout en queue des points (x, y) aux listes L_y :

  $0 \to (2,0)$
  $1 \to (7,1) \to (8,1)$
  $2 \to (4,2) \to (6,2) \to (0,2)$
  $3 \to (0,3) \to (9,3) \to (8,3)$
  $4 \to (7,4)$
  $5 \to$
  $6 \to (2,6)$
  $7 \to (3,7)$
  $8 \to (5,8) \to (4,8)$
  $9 \to (3,9)$

- parcours des abcisses dans l'ordre des $(L_i)_{0 \leq i \leq 9}$
  ajout en queue des points (x, y) aux listes K_x :

  $0 \to (0,2) \to (0,3)$
  $1 \to$
  $2 \to (2,0) \to (2,6)$
  $3 \to (3,7) \to (3,9)$
  $4 \to (4,2) \to (4,8)$
  $5 \to (5,8)$
  $6 \to (6,2)$
  $7 \to (7,1) \to (7,4)$
  $8 \to (8,1) \to (8,3)$
  $9 \to (9,3)$

Tri par base des points (suite)

- le tableau de listes K précédent :

  $0 \rightarrow (0,2) \rightarrow (0,3)$
  $1 \rightarrow$
  $2 \rightarrow (2,0) \rightarrow (2,6)$
  $3 \rightarrow (3,7) \rightarrow (3,9)$
  $4 \rightarrow (4,2) \rightarrow (4,8)$
  $5 \rightarrow (5,8)$
  $6 \rightarrow (6,2)$
  $7 \rightarrow (7,1) \rightarrow (7,4)$
  $8 \rightarrow (8,1) \rightarrow (8,3)$
  $9 \rightarrow (9,3)$

- ultime parcours pour obtenir le tri cartésien :

  $(0,2)$ $(0,3)$ $(2,0)$ $(2,6)$ $(3,7)$ $(3,9)$ $(4,2)$ $(4,8)$ $(5,8)$ $(6,2)$ $(7,1)$ $(7,4)$ $(8,1)$ $(8,3)$ $(9,3)$

- si l'ajout en queue dans les listes est en O(1) -

Complexité : O(n)