Algorithmique 1
Algorithmes gloutons

Hajer Akid & Sandrine Julia

Semestre 4

Des problèmes à forte combinatoire

Des problèmes nombreux et variés

- les problèmes d'**exploration**
  - graphes réels
  - graphes de solutions (jeux)
- les problèmes d'**optimisation combinatoire**
  - problèmes de *Sac-à-dos*
  - problèmes de *Bin-packing*
  - problèmes d'*ordonnancement*
  - problèmes d'*allocation de ressources*
    - planification de tâches
    - emploi du temps
    - distribution

Forte combinatoire

Une explosion du nombre de cas à traiter

1. énumération de sous-ensembles
2. croissance exponentielle de l'arbre des solutions
3. calcul des permutations des éléments d'un ensemble

Exemple

1. problème de *Sudoku*
2. problème des *8 reines*
3. problème du *Voyageur de commerce*

Quizz

Un problème informatique est résolu dès qu'on a un algorithme correct pour le résoudre ? ☐

Actuellement, il existe des problèmes utiles et simples à poser sans algorithme efficace ? ☐

La puissance des ordinateurs peut toujours compenser l'inefficacité d'un algorithme ? ☐

Peut-on espérer trouver des algorithmes efficaces pour des problèmes sans ? ☐

Tout problème dont on peut vérifier une solution facilement peut-il être résolu facilement ? ☐

Les problèmes sont classés en fonction des meilleurs algorithmes qui les résolvent ? ☐

Utiliser plus d'espace peut permettre de trouver un algorithme plus efficace en temps pour résoudre un problème ? ☐

Réponses à discuter ...

Un problème informatique est résolu dès qu'on a un algorithme correct pour le résoudre ?	☒
Actuellement, il existe des problèmes utiles et simples à poser sans algorithme efficace ?	☑
La puissance des ordinateurs peut toujours compenser l'inefficacité d'un algorithme ?	☒
Peut-on espérer trouver des algorithmes efficaces pour des problèmes sans ?	☑
Tout problème dont on peut vérifier une solution facilement peut-il être résolu facilement ?	☒
Les problèmes sont classés en fonction des meilleurs algorithmes qui les résolvent ?	☑
Utiliser plus d'espace peut permettre de trouver un algorithme plus efficace en temps pour résoudre un problème ?	☑

P = NP ? (une brève évocation)

Une question majeure à 1 million de dollars

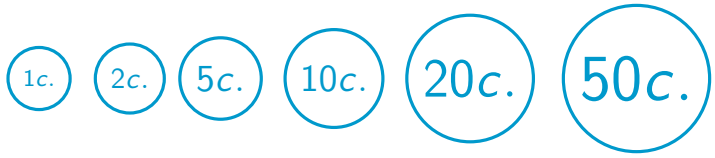
- sur la recherche de solution(s) dans un ensemble exponentiel de possibilités
- *une solution que l'on peut vérifier facilement peut-elle être trouvée facilement ?*
- en informatique, *facilement* veut dire en **temps polynomial** sur une machine *déterministe*
- on appelle **P** *(Polynomial time)* la classe des problèmes informatiques solubles ainsi
- les problèmes dont on peut **vérifier** une solution en temps polynomial sur une machine *déterministe* ou, de façon équivalente, dont une solution peut être trouvée en temps polynomial sur une machine *non-déterministe*, forment la classe **NP** *(Non-deterministic Polynomial time)*
- il est communément admis que P$\neq$ NP mais ce n'est pas démontré
- une réponse dans un sens ou dans l'autre aurait des conséquences innombrables

Remarque

- Le problème de la satisfaisabilité d'une formule booléenne **SAT** figure parmi les problèmes les plus durs de la classe NP

Problème du retour de monnaie

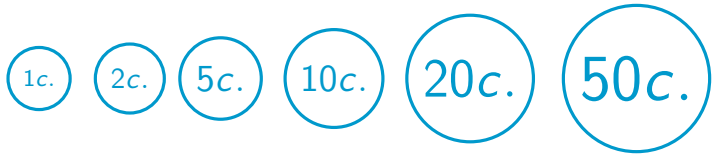
Comment réaliser une somme donnée avec un nombre minimal de pièces ?



29 *cents* : 2*x* 1*x* 1*x*

Problème du retour de monnaie

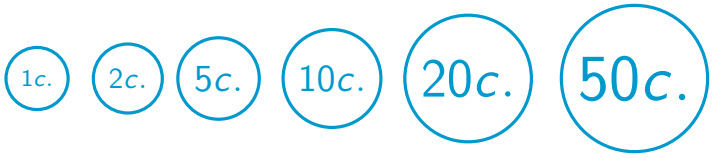
Comment réaliser une somme donnée avec un nombre minimal de pièces ?



43 *cents* : 1*x* 1*x* 2*x*

Problème du retour de monnaie

Comment réaliser une somme donnée avec un nombre minimal de pièces ?



97 *cents* : 1x 1x 2x 1x

Algorithme

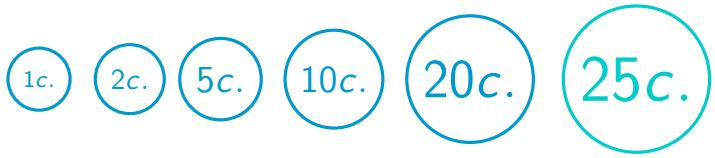
```
nombrePiecesMin (entier P [], entier S){
  n ← longueur(P)
  Res : entier tableau[n]
  pour (i←1, i≤ n, i++){
    Res[i] ← 0
    }
  i ← n
  tant que (S>0) {
    si (P[i] ≤ S) {
      S ← S − P[i]
      Res[i] ← Res[i] + 1
    }
    sinon {
      i ← i−1
    }
  }
  retourner Res
}
```

Complexité : $O(S + n)$

Attention ! Cet algorithme glouton ne donne pas toujours le meilleur résultat ...

Contre-exemple

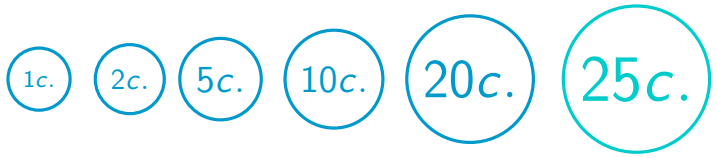
Appliquons l'algorithme précédent sur ce nouvel ensemble de pièces :



29 *cents* : 2x 1x

Contre-exemple

Appliquons l'algorithme précédent sur ce nouvel ensemble de pièces :



43 *cents* : 1x 1x 1x 1x 1x

Meilleure solution ? 1x 1x 2x

Preuve que l'algorithme précédent est faux !

En vrai, il est correct pour certains jeux de monnaie, pas pour d'autres.

Algorithme “glouton“ (greedy algorithm)

Qu'est-ce ?

- un algorithme qui se fonde sur une stratégie **heuristique**
- à chaque étape, le fil rouge choisi est un **optimum** (local)
- l'algorithme construit une solution **incrémentale** et ne revient jamais sur ses choix
- dans le meilleur des cas, il fournit à la fin une solution **optimale** (globale donc)

- un tel algorithme est souvent linéaire
- si il fournit une solution optimale, il faut le **prouver**
- il évite de traiter tous les cas comme le ferait un algorithme **exhaustif** (*brute-force*) en temps exponentiel

Un problème de planification de tâches

Chaque tâche s'exécute sur une machine pendant un laps de temps précis :

Données

- un tableau T des tâches T_i $(1 \leq i \leq n)$, pour chaque tâche on donne le couple (d_i, f_i) tel que :
  - d_i est la date de début
  - f_i est la date de fin
  - $d_i < f_i$
- on dispose *a priori* d'un nombre illimité de machines (!)

Problème

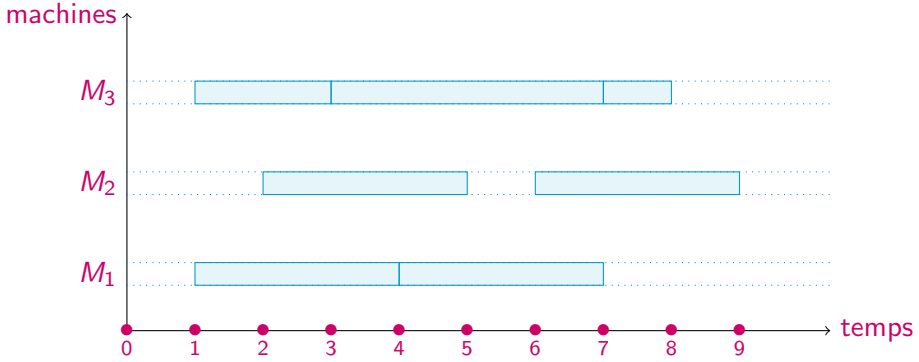
- planifier chacune des tâches sur une machine donnée de façon non conflictuelle
- utiliser un nombre minimal de machines

Exemple

La donnée de 7 tâches sous la forme de couples date de début et de fin :
$T = (2,5), (1,3), (1,4), (7,8), (3,7), (4,7), (6,9)$

Une solution optimale

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



Scénario de l'algorithme

- on construit un tas_min :
  - on y conserve pour chaque machine la date où elle se libère
  - la racine doit concerner une machine où la **date de fin** de la dernière tâche est **minimale**
- on trie le tableau des tâches selon les **dates de début croissantes**
- pour chaque tâche :
  - soit on peut la planifier sur la machine libre au plus tôt
  - soit elle nécessite l'usage d'une nouvelle machine avant d'être planifiée

Algorithme

```
Task_scheduling (couples T []){// T : couples de dates (d,f)
  n ← longueur(T) ;   i ← 1
  m ← 0                          // nombre de machines
  Res : entier tableau[n]   // table résultat
  H ← tas_min_vide() ; Heappush(H,(1,0)) // initialisation tas
  T ← tri(T)                     // tri tâches sur date début
  tant que (i≤n) {
    d,f ← T[i]
    M, min_f ← Heappop(H)      // M machine finissant au plus tôt
    si (d < min_f) {
      m ← m+1                    // nouvelle machine requise
      Heappush(H,(m,f))
      Res[i] ← m
    } sinon {
      Heappush(H,(M,f))
      Res[i] ← M
    }
    i ← i+1
  }
  retourner Res,m
}
```

Complexité : $O(nlog(n))$

Correction de l'algorithme

Démonstration :

- considérons que l'algorithme a utilisé k machines
- supposons qu'il existe une meilleure solution avec $k-1$ machines :
  - soit i la première tâche programmée sur la machine k
  - la tâche i est en conflit temporel avec $k-1$ autres tâches
  - les tâches ont été ordonnées par date de début donc les tâches en conflit avec la tâche i ont toutes leur date de début antérieure ou égale à d_i et leur date de fin strictement supérieure à d_i
  - donc ces tâches ne sont pas seulement en conflit avec la tâche i mais sont toutes en conflit les unes avec les autres
  - comme k tâches sont en conflit, on a au moins besoin de k machines
- contradiction donc notre supposition est fausse : il n'y a pas de solution avec $k-1$ machines
- conclusion : l'algorithme est **correct**

Limite des algorithmes "gloutons"

Heuristique

- méthode de calcul qui fournit rapidement une solution réalisable pour un problème d'optimisation difficile
- elle ne fournit pas nécessairement une solution optimale ou exacte
- quand le choix de l'élément fil-rouge depend des choix ultérieurs, elle ne peut pas conduire à une solution optimale

D'autres méthodes à essayer :

- la **programmation dynamique**
- le *rétro-parcours* plus connu sous le nom de **backtracking**
- les algorithmes *probabilistes*
- la *programmation par contraintes*
- les *algorithmes génétiques*

Remarque

Pas d'algorithme glouton pour le problème des *8 reines* ou celui du *voyageur de commerce*

Rétro-parcours ou *backtracking*

Ce principe, en totale opposition à celui des algorithmes gloutons, est aussi très répandu :

- l'algorithme construit **incrémentalement** une solution valide
- à chaque étape, il tente de prolonger son début de solution
  - pour toute possibilité valide, il continue d'essayer de la prolonger
  - si c'est impossible, il abandonne ce début de solution

Exemple

Nous allons l'appliquer au problème des 8 reines ...

Problème des 8 reines : algorithme

```
booléen choixCorrect (entier Y[], entier k){
  // vrai si ajout nouvelle reine en case (k,Y[k]) valide
  i ← 0
  tant que ((i < k) et Y[i]≠Y[k] et (k − i) ≠| Y[k] − Y[i] |) {
       i ← i+1
  }
  retourner i = k
}

Prolonge (entier Y[], entier k){// indices débutant à 0
  // Antécédent: k reines placées dans cases (i,Y[i]) pour i<=k
  // Rôle si k<8 : tente de placer une reine dans une case (k,j)
  k ← k + 1
  si (k = 8) {
    afficher(Y)   // affichage d'une solution
  }
  sinon {
    pour (j←0; j < 8; j++){
      Y[k] ← j
      si choixCorrect(Y,k) {
        Prolonge(Y,k)
      }
} } }
```


Avec 4 reines, 2 solutions :