DIU EIL 2019 / Bloc 2

Séance 5 – Algorithmes gloutons

UNIVERSITÉ DE NANTES

Glouton ou avare ?

- En anglais « greedy » = glouton / avare

- Principe algorithmique :
  - Résoudre un problème pas à pas
  - À chaque pas, faire le choix de moindre coût (ou meilleur gain)

- Moindre/meilleur ⇒ problème d'optimisation

Problème d'optimisation

- Définition : Problème dont la solution consiste à minimiser/ maximiser un/des critères (appelés *objectifs*)

- Exemples
  - Plus court chemin (GPS)
  - Plus petit temps d'exécution (Ordonnancement de processus)
  - Meilleure occupation mémoire (Allocateur mémoire)
  - … (proposez-en quelques-uns)

Problème d'optimisation

- Spécification :
  - Rôle = … le plus/moins …
  - Entrées/sorties : …
  - Pré-condition : …
  - Post-condition : sortie = max/min …

Activité – Rendu de monnaie / spécification

Énoncé : Rendre la monnaie sur une somme S payée avec un montant M en utilisant le moins de pièces possibles

- Donner une version non-contextuelle (et la plus générale possible) de ce problème
- Donner une spécification précise de ce problème
- Souligner les éléments qui en font un problème d'optimisation

Activité – Rendu de monnaie / spécification

- Version non-contextuelle :
  - Réaliser un certain entier N en additionnant un minimum d'entiers pris, avec répétition, parmi un ensemble fini $P=\{p_1,\ldots,p_k\}$
- Spécification :
  - Rôle : cf. ci-dessus
  - Entrées : Entier N, Ensemble d'entiers $P=\{p_1,\ldots,p_k\}$
  - Sortie : Multiplicités $M=\{m_1,\ldots,m_k\}$
  - Précondition : $N>0,\ \forall i\in[1..k]\ p_i > 0$
  - Postcondition : $\forall i\in[1..k]\ m_i\geqslant 0$ et $M\cdot P=N$ et $\forall M'$ tq $M'\cdot P=N,\ |M'|\geqslant|M|$

    avec $M\cdot P=m_1p_1+\ldots+m_kp_k$ et $|M|=\sum_i m_i$

Stratégie de force brute (énumérative)

- L'approche naïve (non-gloutonne, mais sûre) pour résoudre un problème d'optimisation consiste à passer en revue toutes les options possibles et retenir la meilleure ; on l'appelle aussi « générer et tester » *(generate-and-test)*
- Variations possibles :
  - S'arrêter au premier choix_faits complet et satisfaisant (problème de satisfaction, e.g., sudoku)
  - Filtrer/classer les choix possibles en cours de génération (heuristiques de recherche, e.g., meilleur d'abord)
- Exemples :
  - Donner tous les lancers de 3 dés à 6 faces qui font 12.
  - Donner toutes les frises de faïences de N cm (cf. séance 4)

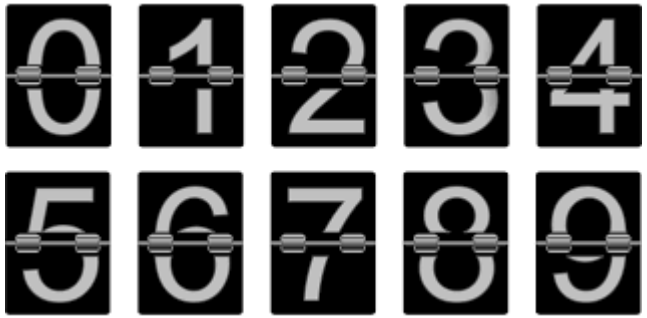
Algorithme d'énumération

- Propriétés de l'approche :
  - Termine ssi le nombre de choix possibles à une étape donnée est fini, et que le nombre d'étapes de choix est borné
    ⇒ Le nombre de combinaisons de choix est fini
  - Est partiellement correct ssi toutes les combinaisons de choix possibles sont bien testées
  - Prend un temps proportionnel au nombre de choix possibles multiplié par le coût des opérations à chaque choix (tests de satisfaction/optimalité, mise à jour du choix / de la solution)
  - Prend un espace mémoire dépendant des données locales utilisées dans les différentes opérations, et éventuellement des empilements pour appels récursifs

Stratégie de force brute (énumérative)

- Limites de l'approche :
  - La solution optimale est souvent unique (ou peu fréquente) parmi toutes les options possibles
  - Il n'est souvent pas possible de savoir si on a déjà trouvé la solution optimale ou pas encore (e.g., recherche du maximum d'un tableau)
    ⇒ impossible d'arrêter l'énumération de façon précoce
  - Le nombre d'options explose souvent *combinatoirement* (produit des choix à chaque étape = nombre exponentiel/factoriel de combinaisons), et le temps de calcul est alors déraisonnable

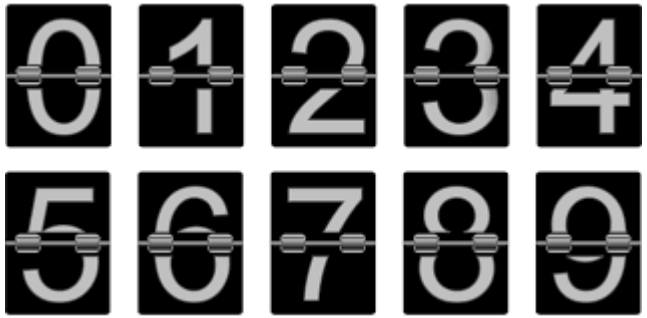
Algorithme d'énumération itératif



- Algorithme basé sur le principe du *compteur mécanique* : considère à chaque itération un choix complet possible, teste s'il est satisfaisant/meilleur, puis passe au suivant selon un ordre total sur les choix possibles

```
fonction énumérer(C)   # Choix par étape C=[C1,…,Ck]
  choix_faits ← [1,…,1] # Choix indicés de 1 à Ck
  solution ← [0,…,0] # Solution factice en cas d'absence de solution
  fini ← Faux
  tant que non fini faire
    si choix_faits est satisfaisant/meilleur…
      alors solution ← choix_faits # ; fini ← Vrai
    fin si
    fini ← non choix_suivant(choix_faits,C)
  fin tant que
  retourner solution
```

Algorithme d'énumération itératif



- Algorithme basé sur le principe du *compteur mécanique* : considère à chaque itération un choix complet possible, teste s'il est satisfaisant/meilleur, puis passe au suivant selon un ordre total sur les choix possibles

```
# Passe au choix suivant s'il existe et retourne vrai dans ce cas
fonction choix_suivant(choix_faits,C)
  e ← 0 ; E ← Taille(C) #<=> Taille(choix_faits) <=> nombre d'étapes
  tant que e<E et choix_faits[e]=C[e] faire # évaluation paresseuse
    choix_faits[e] ← 1 ; e ← e+1
  fin tant que
  si e<E alors
    choix_faits[e] ← choix_faits[e]+1
  fin si
  retourner e<E
```

Algorithme d'énumération itératif : exemple

Exemple : si énumérer(C=[2,3,4]) (2 choix * 3 choix * 4 choix)

- Initialement, choix_faits = [1,1,1]
- Au fil des itérations, il évolue comme suit :
  - Itération 1 : il prend la valeur [2,1,1]
  - Itération 2 : [1,2,1] (gestion du *débordement* par `choix_suivant` => 2e choix modifié)
  - Itération 3 : [2,2,1]
  - Itération 4 : [1,3,1] (gestion du *débordement* par `choix_suivant` => 2e choix modifié)
  - Itération 5 : [2,3,1]
  - Itération 6 : [1,1,2] (gestion du *débordement* par `choix_suivant` => 3e choix modifié)
  - Itérations suivantes : [2,1,2] ; [1,2,2] ; [2,2,2] ; [1,3,2] ; [2,3,2] ; [1,1,3] ; [2,1,3] ; [1,2,3] ; [2,2,3] ; [1,3,3] ; [2,3,3] ; [1,1,4] ; [2,1,4] ; [1,2,4] ; [2,2,4] ; [1,3,4] ; [2,3,4]
  - fin avec choix_faits=[1,1,1] et fini=Vrai

⇒ l'algorithme a bien énuméré les 2*3*4 combinaisons possibles

Activité – Rendu de monnaie / énumération itérative

- Proposer un algorithme d'énumération itératif pour le problème de rendu de monnaie
- Argumenter sur sa terminaison / correction partielle.
- Quelle est sa complexité temporelle/spatiale ?

Activité – Rendu de monnaie / énumération itérative

- Le point de départ est de bien dénombrer les étapes et choix à chaque étape, par exemple :
  - Étapes = choisir la multiplicité au rang i
  - Choix = pour un rang i donné, m_i peut aller de **0** à N/P_i

  ⇒ nb étapes E=Taille(P) et nb choix possibles $C_i=N/P_i+1$
- Il faut ensuite décider ce qu'est une solution et comment comparer deux solutions
  - Solution = multiplicité M tel que $m_i>0$ et $M \cdot P=N$
  - Comparaison de solutions M et M' : $|M|<|M'|$

Activité – Rendu de monnaie / énumération itérative

- Une implémentation en Python :

```
def multiplicite_suivante(M,C):
    e = 0 ; E = len(C)
    while e<E and M[e]==C[e]:
        M[e] = 0 ; e += 1
    if e < E:
        M[e] += 1
    return e<E
```

- ```
  def enumerer_rendu(N,P):
      E = len(P) ; C = [N//p for p in P]
      S = C # solution factice
      M = [0]*E # choix initial = aucune pièce
      fini = False
      while not fini:
          print(". ",M)
          if sum([m*p for (m,p) in zip(M,P)]) == N:
          # choix satisfaisant
              if sum(M) < sum(S): # choix meilleur
                  S = M.copy() ; print("meilleur !")
          fini = not multiplicite_suivante(M,C)
      return S
  ```
  ```

Activité – Rendu de monnaie / énumération itérative

- Exemple d'exécution :

```
>>> print(" rendu = ",enumerer_rendu(8,[5,2]))

.   [0, 0]
.   [1, 0]
.   [0, 1]
.   [1, 1]
.   [0, 2]
.   [1, 2]
.   [0, 3]
.   [1, 3]
.   [0, 4]
meilleur !
.   [1, 4]
rendu = [0,4]
```

Activité – Rendu de monnaie / énumération itérative

- Terminaison / Correction partielle (arguments) :
  - On peut définir un ordre lexicographique (inverse) total sur les choix possibles : $[0,0,\ldots,0] < [1,0,\ldots,0] < [2,0,\ldots,0] < \ldots < [0,1,\ldots,0] < [1,1,\ldots,0] < [2,1,\ldots,0] < \ldots < [C_1,\ldots,C_k]$
  - On peut démontrer (~invariant) que l'algorithme part du premier choix possible dans cet ordre et les explore tous jusqu'au dernier

⇒ L'algorithme termine puisque le nombre de choix est fini

⇒ L'algorithme est correct puisqu'il évalue tous les choix et retient le meilleur selon l'objectif défini

- Complexité : le nombre de choix est en $O(\prod_{i=1..k} C_i)$
  - Le nombre de test de choix satisfaisant est donc du même ordre
  - Le nombre d'affectation est en $O(k * \prod_{i=1..k} C_i)$ (gestion du débordement)

Algorithme d'énumération récursif

- Aussi appelé *retour-arrière* ou *retour-sur-trace* (*backtracking* en anglais) : complète à chaque récursion une combinaison partielle (initialement vide) et, si elle est complète, teste si elle est satisfaisante/meilleure

```
fonction explorer(choix_faits,C,solution)
  si choix_faits est complet # À préciser
  alors
    si choix_faits est satisfaisant/meilleur que solution
      alors solution ← choix_faits
    fin si
  sinon
    pour chaque choix suivant faire # À préciser
      solution ← explorer(choix_faits ⊕ choix,C,solution)
                 # Opération d'agrégation ⊕ à préciser
    fin pour
  fin si
  retourner solution
```

Algorithme d'énumération récursif : exemple

Exemple : explorer(choix_faits=[],C=[2,3,4],solution=[0,0,0])

- Appel initial : [] est une combinaison incomplète ⇒ appels récursifs :
  - Explorer([1],C,solution) ; [1] incomplet ⇒ appels récursifs :
    - Explorer([1,1],…) ; [1,1] incomplet ⇒ appels récursifs :
      - Explorer([1,1,1],…) ; choix complet
      - explorer([1,1,2],…) ; choix complet
      - explorer([1,1,3],…) ; choix complet
      - explorer([1,1,4],…) ; choix complet
    - explorer([1,2],…) ; [1,2] incomplet ⇒ appels récursifs :
      - explorer([1,2,1],…) ; choix complet
      - explorer([1,2,2],…) ; choix complet
      - explorer([1,2,3],…) ; choix complet
      - Explorer([1,2,4],…) ; choix complet
    - …
  - Explorer([2],…) ; choix incomplet ⇒ appels récursifs :
    - …

⇒ l'algorithme énumère les 2*3*4 combinaisons possibles

Activité – Rendu de monnaie / énumération récursive

- Proposer un algorithme de retour-arrière pour résoudre le problème de rendu de monnaie.
- Argumenter sur sa terminaison / correction partielle.
- Quelle est sa complexité temporelle/spatiale ?

Activité – Rendu de monnaie / énumération récursive

- Une implémentation en Python (tenant compte des choix partiels déjà faits pour éviter les combinaisons pas solutions) :

```
def explorer_rendu(N,P):
  return explorer_rendu_rec(N,P,[],[N//p for p in P])


def explorer_rendu_rec(N,P,M,S):
  e = len(M) # rang du prochain choix à faire
  print("."*e,M) # affichage des choix explorés
  if e == len(P): # choix complet
    if N == 0 and sum(M) < sum(S): # solution meilleure
      S = M.copy()
  else:
    for m in range(N//P[e]+1): # choix suivants possibles
      S = explorer_rendu_rec(N-m*P[e],P,M+[m],S)
  return S
```

Activité – Rendu de monnaie / énumération récursive

- Exemple d'exécution :

```
>>> print(" rendu = ",explorer_rendu(8,[5,2]))

 []
. [0]
.. [0, 0]
.. [0, 1]
.. [0, 2]
.. [0, 3]
.. [0, 4]
. [1]
.. [1, 0]
.. [1, 1]
rendu = [0,4]
```

Activité – Rendu de monnaie / énumération récursive

- Terminaison / Correction partielle (arguments) :
  - M, initialement vide, augmente d'un choix à chaque récursion ⇒ fini par atteindre la taille de P
  - À chaque récursion non terminale, M est augmenté de chaque (répétitive pour) choix encore possible vis à vis du reste de la somme à rendre

  ⇒ L'algorithme termine puisque le nombre de choix est fini

  ⇒ L'algorithme est correct puisqu'il évalue tous les choix possibles (n'excédant par N) et retient le meilleur selon l'objectif défini
- Complexité :
  - Temporelle : la majoration de l'algorithme itératif tient
    Un ordre de grandeur plus précis est difficile à calculer du fait de la dépendance des choix partiels faits sur les choix futurs possibles …
  - Spatiale : maximum k appels récursifs imbriqués, et un état mémoire de e+3 entiers (M,N,e,m, mais pas P ni S partagés) à empiler ⇒ $O(k^2)$

Algorithmes d'énumération – Conclusion

- Les algorithmes d'énumération permettent une exploration exhaustive des combinaisons possibles ⇒ Il garantissent de trouver une solution satisfaisante/optimale s'il en existe une, et prouvent qu'il n'en existe pas autrement.
- Exemples d'applications :
  - Résolution de casse-tête (labyrinthe, sudoku, N reines, …)
  - Optimisation combinatoire (rendu de monnaie, sac à dos, …)

  MAIS, il existe souvent de meilleurs algorithmes dédiés !

Algorithmes gloutons

- Principe :
  - Partant d'un choix initial vide, construit une séquence de choix
  - À chaque étape, évalue les choix possibles et retient le meilleur (selon les données disponibles à cette étape, principe de *localité*)
  - **Aucun choix fait n'est jamais remis en cause**

⇒ La séquence de choix conduit à une unique combinaison

  - Soit elle est satisfaisante/optimale : l'algorithme glouton résout bien le problème
  - Soit elle ne l'est pas : l'algorithme glouton est une *heuristique* qui fournit un résultat approché (parfois utilisé comme point de départ pour une exploration plus exhaustive)

Activité – Rendu de monnaie / glouton

- Proposer un algorithme glouton pour traiter ce problème
- Exprimer ses conditions de succès
- Argumenter sa terminaison / correction partielle
- Déterminer sa complexité temporelle / spatiale

Activité – Rendu de monnaie / glouton

- Algorithme glouton :

```
fonction rendre_monnaie(N,P)
  trier(P) // par ordre décroissant
  M ← [0,…,0] ; i ← 0
  tant que M·P < N et i < k faire
    si N−M·P ≥ P[i]
      alors M[i] ← M[i]+1
      sinon i ← i+1
    fin si
  fin tant que
  retourner M
```

- Conditions pour que le résultat soit :
  - Satisfaisant : $M \cdot P = N$ (garantit si $p_k=1$ et $N \in \mathbb{N}$, hypothèses usuelles)
  - Optimal : $|M| \leq |M'| \ \forall\ M'$ tq $M' \cdot P = N$ (dépend fortement de P)

Activité – Rendu de monnaie / glouton

- Exemples :
  - P = [5,2,1], N=8
    - Énumération → M_{opt}=[1,1,1], $|M_{opt}|$=3
    - Glouton → M=[1,1,1], |M|=3 solution optimale
  - P = [6,4,1], N=8
    - Énumération → M_{opt}=[0,2,0], $|M_{opt}|$=2
    - Glouton → M=[1,0,2], |M|=3 solution correcte mais sous-optimale
  - P=[5,2], N=8
    - Énumération → M_{opt}=[0,4], $|M_{opt}|$=4
    - Glouton → M=[1,1], |M|=2 solution incorrecte

Activité – Rendu de monnaie / glouton

- Terminaison :
  - Variant : N-M·P + k-i
    - Toujours positif ou nul (sous hypothèses usuelles)
    - Strictement décroissant à chaque itération :
      - N et k sont constants ;
      - soit M[i] est incrémenté ⇒ M·P augmente strictement ;
      - soit i est incrémenté.

- Complexité : O(k*(N+k)) + coût du tri *(combien ?)*
  - Taille des données : N ? k ? … Les deux !
  - Nombre d'itérations majoré par N+k (sous hypothèses usuelles)
    - Forme des données au pire : $p_i>N$ $\forall i\in[1..k-1]$, $p_k=1$
  - Coût d'une itération : O(k) à cause de M·P … améliorable ?

Activité – Rendu de monnaie / glouton

- Correction partielle :
  - Inutile en théorie puisqu'on a montré des contre-exemples : l'algorithme peut rendre un résultat incorrect ou sous-optimal
  - En fait, on dit que l'algorithme glouton répond correctement et optimalement à ce problème ssi l'ensemble P est canonique *;* et il a été démontré à la fin du 20e siècle qu'il est possible de vérifier qu'un ensemble P donné est canonique en temps polynomial : $O(|P|^3)$
    (par chance la plupart des systèmes monétaires, dont l'Euro, le sont … mais pas tous, cf. l'ancien système britannique)
  - Une procédure de vérification simple (mais non-polynomiale) consiste à comparer la solution gloutonne à la solution optimale (obtenue par une méthode énumérative) pour tous les N compris entre $p_{k-2}+2$ et p_1+p_2-1
    Exemple : pour P={6,4,1}, il suffit de comparer les solutions gloutonnes et optimales pour N entre 8 (6+2) et 9 (6+4-1) !

Activité – Rendu de monnaie / glouton

- Variante plus efficace (?) :

```
fonction rendre_monnaie(N,P)
    trier(P)
    M ← [0,…,0] ; i ← 0
    tant que M·P < N et i < k faire
        M[i] ← (N−M·P) div P[i]
        i ← i+1
    fin tant que
    retourner M
```

Reprendre les preuves et l'analyse de complexité

Activité – Rendu de monnaie / glouton

- Une implémentation en Python :

```
def glouton_rendu2(N,P):
    P.sort(reverse=True)
    M = [0]*len(P) ; i = 0 ; MP = 0
    while MP<N and i<len(P):
        print(". ",M)
        M[i] = (N-MP) // P[i] ; MP += M[i]*P[i]
        i += 1
    return M
```

- Exécution :

```
>>> glouton_rendu2(8,[5,2,1])
.  [0, 0, 0]
.  [1, 0, 0]
.  [1, 1, 0]
rendu = [1, 1, 1]
```

Algorithmes gloutons – Conclusion

- L'approche gloutonne peut répondre efficacement à un problème d'optimisation
- Elle nécessite souvent un pré-traitement consistant à ordonner les choix possibles afin d'identifier le meilleur
- La solution gloutonne n'est pas toujours optimale
- Ce n'est même pas toujours une solution exacte

Activité – Sac à dos

Énoncé : (version « Lupin ») Remplir une besace avec les objets les plus précieux sans excéder la capacité du sac.

Donner :

- un énoncé non-contextuel aussi général et précis que possible ;
- sa spécification ;
- des exemples d'instances et leurs solutions ;
- un algorithme énumératif, sa preuve et sa complexité ;
- un algorithme glouton, sa preuve et sa complexité.

Activité – Sac à dos / Spécification

- Un énoncé non-contextuel possible :
  Quel est le sous-ensemble d'un ensemble de couples (objets) $O=\{o_1, \ldots, o_k\}$, où chaque $o_i=(p_i,v_i)$ est un couple de nombres positifs (poids, valeur), qui maximise la somme de v_i tout en ayant une somme de p_i inférieure à nombre C.
- Spécification :
  - Rôle : cf. supra
  - Entrées : ensemble O de couples de nombres positifs, nombre C
  - Sorties : sous-ensemble O' de O
  - Précondition : $\forall i \in [1,k]\ p_i>0$ et $v_i>0$; $C>0$
  - Post-condition : $O' \subseteq O$ et $\Sigma_{i\in[1,k]}\ p_i' \leq C$
    et $\forall\ O'' \subseteq O$ tq $\Sigma_{i\in[1,k]}\ p_i'' \leq C$, $\Sigma_{i\in[1,k]}\ v_i'' \leq \Sigma_{i\in[1,k]}\ v_i'$

Activité – Sac à dos / Instances

- $O=\{o_1=(1.5kg,200€),o_2=(2kg,500€),o_3=(0.3kg,400€)\}$, $C=2kg$
  $\Rightarrow O' = \{o_1,o_3\}$

- $O=\{o_1=(1.5kg,200€),o_2=(2kg,500€),o_3=(0.3kg,300€)\}$, $C=2kg$
  $\Rightarrow O' = \{o_1,o_3\}$ ou $\{o_2\}$

- $O=\{o_1=(1.5kg,200€),o_2=(2kg,500€),o_3=(0.3kg,300€)\}$, $C=1.8kg$
  $\Rightarrow O' = \{o_1,o_3\}$

- $O=\{o_1=(1.5kg,200€),o_2=(2kg,500€),o_3=(0.3kg,300€)\}$, $C=3.5kg$
  $\Rightarrow O' = \{o_2,o_3\}$

Activité – Sac à dos / Énumération

- Un choix consiste à décider si on prend l'objet i ou pas : c'est une donnée binaire, un booléen peut la modéliser
  ⇒ choix_faits est un tableau de k booléens
  ⇒ 2^k combinaisons possibles, donc complexité exponentielle
- Pour l'approche itérative (compteur mécanique) :
  - Le choix initial consiste à ne prendre aucun objet
  - Le choix suivant consiste à considérer Faux=0 et Vrai=1 pour la gestion du débordement
- Pour l'approche récursive (backtracking) :
  - Le choix partiel initial est toujours vide
  - Augmenter le choix ne se fait que de 2 façons : en ajoutant Faux ou Vrai à la fin de choix_faits ⇒ on peut simplifier la répétitive pour

Activité – Sac à dos / Glouton

- Plusieurs choix possibles :
  - Prendre toujours l'objet de plus grande valeur n'excédant pas la capacité restante ⇒ trier préalablement par valeur décroissante

    Exemple :
    O={o_1=(1.5kg,200€),o_2=(2kg,500€),o_3=(0.3kg,400€)}, C=2kg
    ⇒ $O_{glouton}$ = {o_2} sous-optimal !
  - Prendre toujours l'objet de plus faible poids ⇒ trier préalablement par poids croissant

    Exemple :
    O={o_1=(1.5kg,200€),o_2=(2kg,500€),o_3=(0.3kg,300€)}, C=3.5kg
    ⇒ $O_{glouton}$ = {o_3,o_1} sous-optimal !
  - En fait ce problème est *NP-difficile* (cf. Bloc 5) : toute approche gloutonne est vouée à l'échec !