Développement informatique

Cours 7 :

- -Algorithme glouton
- -Diviser pour régner
- -Programmation dynamique

Contexte

Optimiser un problème, c'est déterminer les conditions dans lesquelles ce problème présente une caractéristique spécifique.

Par exemple, déterminer le minimum ou le maximum d'une fonction est un problème d'optimisation. On peut également citer :

- **La répartition optimale de tâches suivant des critères précis**
- **Le problème du rendu de monnaie,**
- **Le problème du sac à dos,**
- **La recherche d'un plus court chemin dans un graphe,**
- **Le problème du voyageur de commerce.**
- **Un problème d'organisation (planification de RDV sous contraintes)**

Contexte

De nombreuses techniques informatiques sont susceptibles d’apporter une solution exacte ou approchée à ces problèmes.

Certaines de ces techniques, comme l’énumération exhaustive de toutes les solutions, ont un coût machine qui les rend souvent peu pertinentes au regard de contraintes extérieures imposées (temps de réponse de la solution imposé, moyens machines limités).

Contexte

Solutions :

-Les techniques de programmation dynamique ou d’optimisation linéaire → remise en cause des solutions déjà établies. Au lieu de se focaliser sur un seul sous-problème, elle explore les solutions de tous les sous-problèmes pour les combiner finalement de manière optimale.

- Les algorithmes gloutons constituent une alternative dont le résultat n’est pas toujours optimal → choix locaux, jamais remis en cause

Quelques définitions

Algorithme glouton : construit une solution de manière incrémentale, en optimisant un critère de manière locale.

Diviser pour régner : divise un problème en sous-problèmes indépendants (***qui ne se chevauchent pas***), résout chaque sous-problème, et combine les solutions des sous-problèmes pour former une solution du problème initial.

Programmation dynamique : divise un problème en sous-problèmes qui sont non indépendants (***qui se chevauchent***), et cherche (et stocke) des solutions de sous-problèmes de plus en plus grands

Algorithmes gloutons

Le principe de l'algorithme glouton (greedy algorithm) : faire toujours un choix localement optimal dans l'espoir que ce choix mènera a une solution globalement optimale.

Exemples classiques :

- **Le problème du rendu de monnaie**

  Si le rendu est de 7 euros et 30 centimes, on rend la plus grosse pièce à chaque fois.

  5 + 2 + 0.20 + 0.10

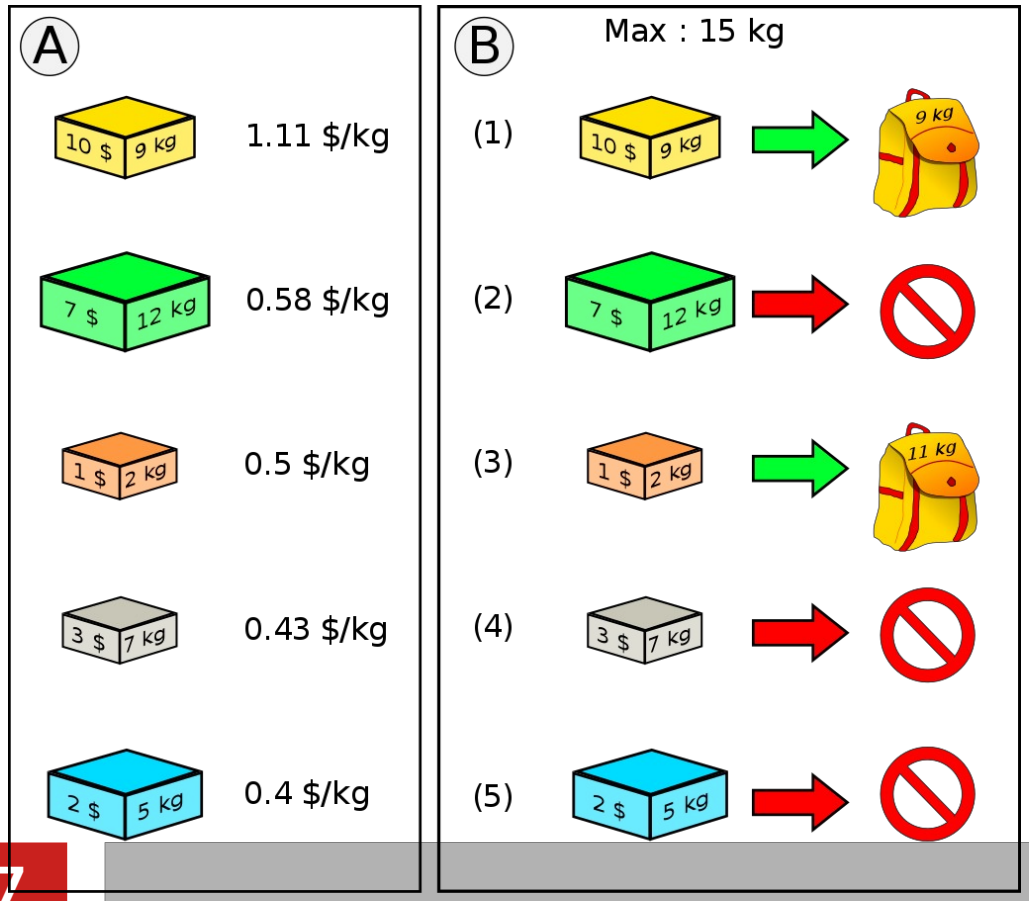
- **Problème du sac a dos**

  Remplir un sac à dos qui peut porter 15 kg maximum avec la plus grande valeur financière

  3kg de riz pour 2€, 10kg de sable pour 3€, ...

  => On obtient le prix au kilo et on met dans le sac les plus gros prix au kilo

Solution du sac à dos avec une approche gloutonne



(1) + (3)
=> 11kg pour 11€

Solution optimale :
(1) + (5)
=> 14kg pour 12€

Le problème de la monnaie

On dispose de pièces de monnaie dont les montants sont des nombres entiers de l'unité de monnaie : $t_0 > t_1 > t_2 > \cdots > t_p$.

Pour payer une somme S (entier naturel quelconque), on aimerait utiliser le nombre minimal de pièces possibles

Le problème de la monnaie

Choix glouton :

- On donne la plus ”grosse” pièce possible.

- Puis à nouveau la plus grosse pièce possible pour le montant restant.

- Puis `a nouveau la plus grosse pièce possible pour le montant restant.

- Etc...

Le problème de la monnaie

Choix optimal ? : L'algorithme mène-t-il toujours à payer avec un nombre minimal de pièces ?

Un exemple :

On suppose que la liste des montants est : T= [18,7,1], c'est à dire que l'on peut disposer de pièces de t_0= 18€ , de pièces de t_1= 7€ et de pièces de t_3= 1€ .

L'algorithme conduit-il toujours à payer avec le nombre minimal de pièces ?

Le problème de la monnaie

Pour une somme à payer de 21€, l’algorithme choisit une pièce de18€ suivie de trois pièces de 1€. Soit 4 pièces alors que trois pièces de 7€ suffisent.

Diviser pour Régner

De nombreux problèmes auxquels on peut être confronté en informatique peuvent être subdivisés en sous-problèmes plus faciles à résoudre :

- **Jeu du « plus ou moins »**
- **Recherche par dichotomie**
- **Exponentiation rapide (c.f. RSA) (n^p = n^a1 * n² * n^a2 * n2)**
- **Tri Fusion**
- **Algorithme de Karatsuba (méthode de multiplication de grands nombres)**

Cette stratégie de création d'algorithmes porte le nom de divide and conquer aux États-Unis ou Veni, divisi,vici. En français, on la traduit souvent par Diviser pour régner, ce qui ne rend pas le sens de conquête d'une solution.

Diviser pour Régner

Trois conditions pour avoir un algorithme diviser-pour-régner efficace:

- Bien décider quand utiliser l'algorithme simple sur de petites instances plutôt que les appels récursifs

- La décomposition d'une instance en sous-instances et la recombinaison des sous-solutions doivent être efficaces

- Les sous-instances doivent être, autant que possible, environ de la même taille

Diviser pour Régner

Un exemple simple : calcul de x^n

(x peut être un nombre, une matrice ou d'un polynôme)

- Algorithme naif :

Y = x . x . x… → Complexité en O(n)

- Méthode binaire

$$x^n = \begin{cases} x^{n/2} \cdot x^{n/2} & \text{si } n \text{ est pair,} \\ x^{\lfloor n/2 \rfloor} \cdot x^{\lfloor n/2 \rfloor} \cdot x & \text{si } n \text{ est impair.} \end{cases}$$

→ Complexité en O(log(n))

Diviser pour Régner

- Méthode des facteurs

(La méthode des facteurs est basée sur la factorisation de n)

$$x^n = \begin{cases} (x^p)^q & \text{si } p \text{ est le plus petit facteur premier de } n \ (n = p \times q), \\ x^{n-1} \cdot x & \text{si } n \text{ est premier.} \end{cases}$$

Exemple : $x^{15} = (x^3)^5 = x^3.(x^3)^4$

→ Il existe une infinité de nombres pour lesquels la méthode des facteurs est meilleure que la méthode binaire

Bref historique

- Programmation dynamique : paradigme développé par Richard Bellman en 1953 chez RAND Corporation.

- « Programmation » = planification

- Technique de conception d'algorithme très générale et performante.

- Permet de résoudre de nombreux problèmes d'optimisation.

Revisitons Fibonacci...

Soit F_n = nombre de lapins au mois n

$F_1 = 1$

$F_2 = 1$

$F_n = F_{n-1} + F_{n-2}$

Ce sont les nombres de Fibonacci :

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Ils croissent très vite: $F_{30} > 10^6$!

En fait, $F_n = 2^{0.694n}$ → croissance exponentielle.

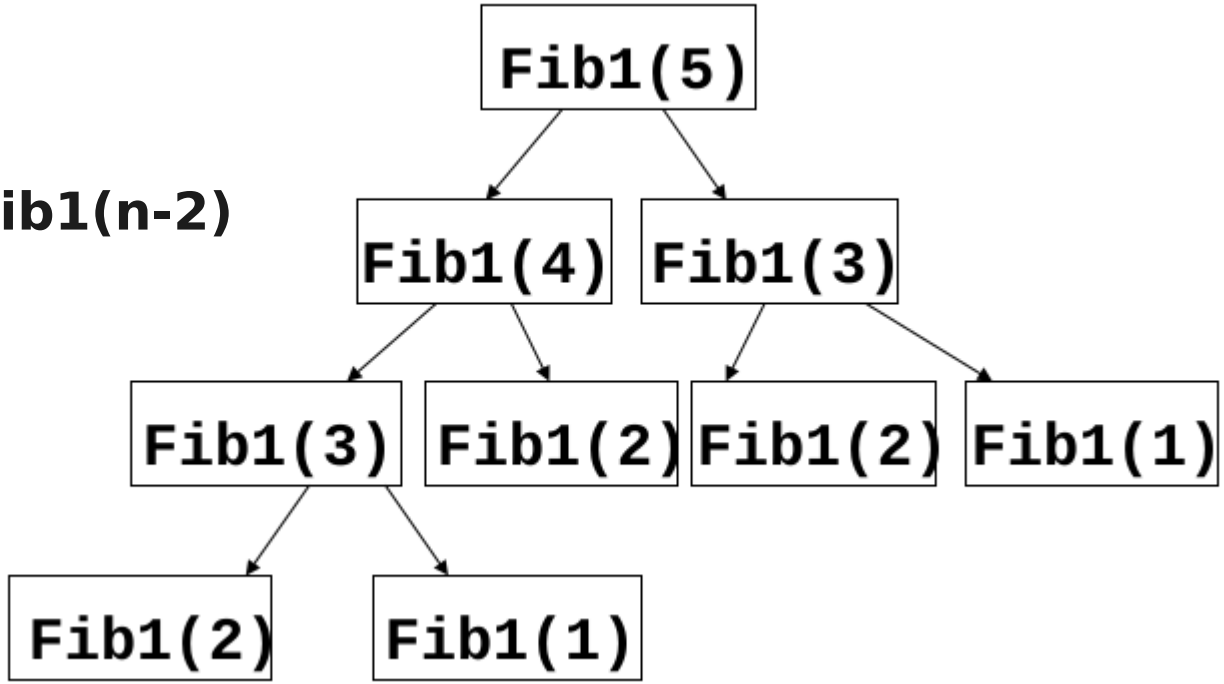
Algorithme récursif inefficace

fonction Fib1(n)

si n = 1 retourner 1

si n = 2 retourner 1

retourner Fib1(n-1) + Fib1(n-2)



Premiere approche : Algorithme récursif avec mémoïsation

```
fonction Fib1mem(n)
si mémo[n] est défini retourner mémo[n]
si n ≤ 2 alors F = 1
sinon F = Fib1mem(n-1) + Fib1mem(n-2)
    mémo[n] = F
retourner F
```

Mémoïser = conserver à la fin de l'exécution d'une fonction le résultat associé aux arguments d'appels, pour ne pas avoir à recalculer ce résultat lors d'un autre appel récursif.

Seconde approche : Approche « du bas vers le haut »

Dans cette approche, on remplit un tableau :

```
fonction Fib2(n)
Créer un tableau fib[1..n]
fib[1] = 1
fib[2] = 1
pour i = 3 à n:
   fib[i] = fib[i-1] + fib[i-2]
retourner fib[n]
```

Mêmes calculs que dans la version mémoïsée. Il faut toutefois identifier un ordre dans lequel résoudre les sous-problèmes.

Conception d’une procédure de programmation dynamique

Quatre étapes :

- **Définir les sous-problèmes.**
- **Identifier une relation de récurrence entre les solutions des sous-problèmes.**
- **En déduire un algorithme récursif avec mémoïsation ou une approche du bas vers le haut**
- **Résoudre le problème original à partir des solutions des sous-problèmes**

Exemples d’algorithmes

- Huit américain
- Plus grand carré uni
- Ordonnancement d’intervalles pondérés (ou son équivalent connu sous « problème de découpe de planches »
- Distance d'édition (Comment calculer la distance d'édition entre deux chaînes de caractères)
- Alignement optimal (entre deux séquences d’ADN par exemple)
- Calcul matriciel (Multiplications chaînées de matrices.)
- Plus court chemin dans un graphe
- Plus longue sous-séquence croissante

En résumé

La programmation dynamique est souvent utilisée lorsqu'une solution récursive se révèle inefficace. Deux approches sont possibles :

- **Solution récursive: Approche Haut→ bas avec mémoïsation.**
- **Programmation dynamique: Bas → haut.**

On évite de calculer plusieurs fois la même chose.