Partie 6

Résolution de problèmes

Plan

Méthodes de résolution de problèmes

Quelques approches génériques pour aborder la résolution d'un problème :

- Approche par force brute : résoudre directement le problème, à partir de sa définition ou par une recherche exhaustive
- Diviser pour régner : diviser le problème en sous-problèmes, les résoudre, fusionner les solutions pour obtenir une solution au problème original
- Programmation dynamique : obtenir la solution optimale à un problème en combinant des solutions optimales à des sous-problèmes similaires plus petits et se chevauchant
- Approche gloutonne : construire la solution incrémentalement, en optimisant de manière aveugle un critère local

Approche par force brute (brute-force)

- Consiste à appliquer la solution la plus directe à un problème
- Généralement obtenue en appliquant à la lettre la définition du problème
- Exemple simple :
  - Rechercher un élément dans un tableau (trié ou non) en le parcourant linéairement
  - Calculer a^n en multipliant a n fois avec lui-même
  - Implémentation récursive naïve du calcul des nombres de Fibonacci
  - ...
- Souvent pas très efficace en terme de temps de calcul mais facile à implémenter et fonctionnel

Exemple : tri

Approches par force brute pour le tri :

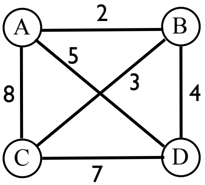
- Un tableau est trié (en ordre croissant) si tout élément est plus petit que l'élément à sa droite
- $\Rightarrow$ tri à bulle : parcourir le tableau de gauche à droite en échangeant toutes les paires d'éléments consécutifs ne respectant pas cette définition
- Complexité : $O(n^2)$
- $\Rightarrow$ tri par sélection : trouver le minimum du tableau, l'échanger avec le premier élément, répéter pour trier le reste du tableau
- Complexité : $\Theta(n^2)$

Recherche exhaustive

- Une solution par force brute au problème de la recherche d'un élément possédant une propriété particulière
- Générer toutes les solutions possibles jusqu'à en obtenir une qui possède la propriété recherchée
- Exemple pour le tri :
  - Générer toutes les permutations du tableau de départ (une et une seule fois)
  - Vérifier si chaque tableau permuté est trié. S'arrêter si c'est le cas.
  - Complexité : $O(n! \cdot n)$
- Généralement utilisable seulement pour des problèmes de petite taille
- Dans la plupart des cas, il existe une meilleure solution
- Dans certains cas, c'est la seule solution possible

Problème du voyageur de commerce

- Etant donné *n* villes et les distances entre ces villes
- Trouver le plus court chemin qui passe par toutes les villes exactement une fois avant de revenir à la ville de départ



Tour	Coût
A-B-C-D-A	17
A-B-D-C-A	21
A-C-B-D-A	20
A-C-D-B-A	21
A-D-B-C-A	20
A-D-C-B-A	17

- Recherche exhaustive : $O(n!)$
- On n'a pas encore pu trouver un algorithme de complexité polynomiale (et il y a peu de chance qu'on y arrive)

Force brute/recherche exhaustive

Avantages :

- Simple et d'application très large
- Un bon point de départ pour trouver de meilleurs algorithmes
- Parfois, faire mieux n'en vaut pas la peine

Inconvénients :

- Produit rarement des solutions efficaces
- Moins élégant et créatif que les autres techniques

Dans ce qui suit, on commencera la plupart du temps par fournir la solution par force brute des problèmes, qu'on cherchera ensuite à résoudre par d'autres techniques

Méthodes de résolution de problèmes

Quelques approches génériques pour aborder la résolution d'un problème :

- Approche par force brute : résoudre directement le problème, à partir de sa définition ou par une recherche exhaustive
- Diviser pour régner : diviser le problème en sous-problèmes, les résoudre, fusionner les solutions pour obtenir une solution au problème original
- Programmation dynamique : obtenir la solution optimale à un problème en combinant des solutions optimales à des sous-problèmes similaires plus petits et se chevauchant
- Approche gloutonne : construire la solution incrémentalement, en optimisant de manière aveugle un critère local

Plan

Approche diviser-pour-régner *(Divide and conquer)*

Principe général :

- Si le problème est trivial, on le résoud directement
- Sinon :
  1. Diviser le problème en sous-problèmes de taille inférieure (Diviser)
  2. Résoudre récursivement ces sous-problèmes (Régner)
  3. Fusionner les solutions aux sous-problèmes pour produire une solution au problème original

Exemples déjà rencontrés

- Merge sort :
  1. Diviser : Couper le tableau en deux sous-tableaux de même taille
  2. Régner : Trier récursivement les deux sous-tableaux
  3. Fusionner : fusionner les deux sous-tableaux

  Complexité : $\Theta(n \log n)$ (force brute : $\Theta(n^2)$)
- Quicksort :
  1. Diviser : Partionner le tableau selon le pivot
  2. Régner : Trier récursivement les deux sous-tableaux
  3. Fusionner : /

  Complexité en moyenne : $\Theta(n \log n)$ (force brute : $\Theta(n^2)$)
- Recherche binaire (dichotomique) :
  1. Diviser : Contrôler l'élement central du tableau
  2. Régner : Chercher récursivement dans un des sous-tableaux
  3. Fusionner : trivial

  Complexité : $O(\log n)$ (force brute : $O(n)$)

Exemple 1 : Calcul du minimum/maximum d'un tableau

- Approche par force brute pour trouver le minimum ou le maximum d'un tableau

```
MIN(A)
min = A[1]
for i = 2 to A.length
    if min > A[i]
        min = A[i]
return min
```

```
MAX(A)
max = A[1]
for i = 2 to A.length
    if max < A[i]
        max = A[i]
return max
```

- Complexité : $\Theta(n)$ ($n-1$ comparaisons)
- Peut-on faire mieux ?

Exemple 1 : Calcul du minimum/maximum d'un tableau

- Approche par force brute pour trouver le minimum ou le maximum d'un tableau

```
MIN(A)
min = A[1]
for i = 2 to A.length
        if min > A[i]
                min = A[i]
return min
```

```
MAX(A)
max = A[1]
for i = 2 to A.length
        if max < A[i]
                max = A[i]
return max
```

- Complexité : $\Theta(n)$ ($n-1$ comparaisons)
- Peut-on faire mieux ?
  - Non, pas en notation asymptotique (le problème est $\Theta(n)$)
  - Par contre, on peut diminuer le nombre total de comparaisons pour calculer à la fois le minimum et le maximum

Calcul simultané du minimum et du maximum

- Approche diviser-pour-régner pour le calcul simultané du minimum et du maximum

```
MAX-MIN(A, p, r)
if r - p ≤ 1
    if A[p] < A[r]
        return (A[r], A[p])
    else return (A[p], A[r])
q = ⌊(p+r)/2⌋
(max1, min1) = MAX-MIN(A, p, q)
(max2, min2) = MAX-MIN(A, q+1, r)
return (MAX(max1, max2), MIN(min1, min2))
```

Appel initial : MAX-MIN$(A, 1, A.length)$

- Correct ? Oui (preuve par induction)
- Complexité ?

Analyse de complexité

- En supposant que n est une puissance de 2, le nombre de comparaisons $T(n)$ est donné par :

$$T(n) = \begin{cases} 1 & \text{si } n = 2 \\ 2T(n/2) + 2 & \text{sinon} \end{cases}$$

qui se résoud en :

$$\begin{array}{rcl} T(n) & = & 2T(n/2) + 2 \\ & = & 4T(n/4) + 4 + 2 \\ & = & 8T(n/4) + 8 + 4 + 2 \\ & = & 2^i T(n/2^i) + \sum_{j=1}^{i} 2^j \\ & = & 2^{\log_2(n)-1} T(2) + \sum_{j=1}^{\log_2(n)-1} 2^j \\ & = & 3/2n - 2 \end{array}$$

- C'est-à-dire 25% de comparaisons en moins que les méthodes séparées

Exemple 2 : Recherche de pics

$-\infty$	1	2	6	5	3	7	4	$-\infty$

- Soit un tableau $A[1 \,..\, A.length]$. On supposera que $A[0] = A[A.length + 1] = -\infty$.
- Définition : $A[i]$ est un pic s'il n'est pas plus petit que ses voisins :

$$A[i-1] \leq A[i] \geq A[i+1]$$

  ($A[i]$ est un maximum local)
- But : trouver un pic dans le tableau (n'importe lequel)
- Note : il en existe toujours un

Approche par force brute

- Tester toutes les positions séquentiellement :

```
PEAK1D(A)
for i = 1 to A.length
      if A[i − 1] ≤ A[i] ≥ A[i + 1]
            return i
```

- Complexité : $\Theta(n)$ dans le pire acas

Approche par force brute 2

- Le maximum global du tableau est un maximum local et donc un pic

```
PEAK1D(A)
m = A[0]
for i = 1 to A.length
    if A[i] > A[m]
        m = i
return m
```

- Complexité : $\Theta(n)$ dans tous les cas

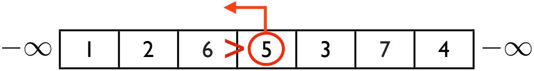
Une meilleure idée

Approche diviser-pour-régner :

- Sonder un élément $A[i]$ et ses voisins $A[i-1]$ et $A[i+1]$
- Si c'est un pic : renvoyer i
- Sinon :
  - les valeurs doivent croître au moins d'un côté

$$A[i-1] > A[i] \text{ ou } A[i] < A[i+1]$$

  - Si $A[i-1] > A[i]$, on cherche le pic dans $A[1 \mathinner{.\,.} i-1]$
  - Si $A[i+1] > A[i]$, on cherche le pic dans $A[i+1 \mathinner{.\,.} A.\mathit{length}]$



- A quel position i faut-il sonder ?

Algorithme

```
PEAK1D(A, p, r)
q = ⌊(p+r)/2⌋
if A[q − 1] ≤ A[q] ≥ A[q + 1]
      return q
elseif A[q − 1] > A[q]
      return PEAK1D(A, p, q − 1)
elseif A[q] < A[q + 1]
      return PEAK1D(A, q + 1, r)
```

Appel initial : PEAK1D$(A, 1, A.length)$

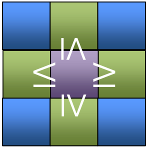
Analyse

- Correction : oui
  - On doit prouver qu'il y aura un pic du côté choisi
  - Preuve par l'absurde :
    - Supposons que $A[q+1] > A[q]$ et qu'il n'y ait pas de pic dans $A[q+1\,..\,r]$
    - On doit avoir $A[q+2] > A[q+1]$ (sinon $A[q+1]$ serait un pic)
    - On doit avoir $A[q+3] > A[q+2]$ (sinon $A[q+2]$ serait un pic)
    - ...
    - On doit avoir $A[r] > A[r-1]$ (sinon $A[r-1]$ serait un pic)
    - Comme $A[r] > A[r+1] = -\infty$, $A[r]$ est un pic, ce qui contredit l'hypothèse

- Complexité :
  - Dans le pire cas, on a $T(n) = T(n/2) + c_1$ et $T(1) = c_2$ (idem recherche binaire)
  - $\Rightarrow T(n) = O(\log n)$

Extension à un tableau 2D

- Soit une matrice $n \times n$ de nombres
- Trouver un élement plus grand ou égal à ses 4 voisins (max)



9	3	5	2	4	9	8
7	2	5	1	4	0	3
9	8	9	3	2	4	8
7	6	3	1	3	2	3
9	0	6	0	4	6	4
8	9	8	0	5	3	0
2	1	2	1	1	1	1

(Demaine & Leiserson)

- Approche par force brute : $O(n^2)$
- Recherche du maximum : $\Theta(n^2)$

Approche diviser-pour-régner

- Chercher le maximum global dans la colonne centrale
- Si c'est un pic, le renvoyer
- Sinon appeler la fonction récursivement sur les colonnes à gauche (resp. droite) si le voisin à gauche (resp. droite) est plus grand

9	3	5	2	4	9	8
7	2	5	1	4	0	3
9	8	9	3	2	4	8
7	6	3	1	3	2	3
9	0	6	0	4	6	4
8	9	8	0	5	3	0
2	1	2	1	1	1	1

9	9	9	3	5	9	8

(Demaine & Leiserson)

Analyse : correction

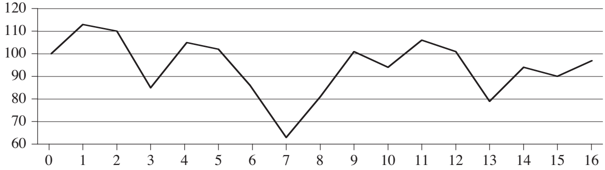
- On doit prouver qu'il y a bien un pic du côté choisi
- Preuve par l'absurde :
  - Supposons qu'il n'y ait pas de pic
  - Soient $A[i, j]$ le maximum de la colonne centrale et $A[i, k]$ le voisin le plus grand ($k = j - 1$ ou $k = j + 1$)
  - $A[i, k]$ doit avoir un voisin $A[p_1, q_1]$ avec une valeur plus élevée (sinon, ce serait un pic)
  - $A[p_1, q_1]$ doit avoir un voisin $A[p_2, q_2]$ avec une valeur plus élevée (sinon, ce serait un pic)
  - ...
  - Le voisin doit toujours rester du même côté de la colonne centrale (puisque $A[i, k] > A[i, j]$ et $A[i, j]$ est le maximum de la colonne j)
  - A un certain point, on va manquer de points
  - Il doit donc y avoir un pic

Analyse : complexité

- $\Theta(n)$ pour trouver le maximum d'une colonne
- $O(\log n)$ itérations
- $O(n \log n)$ au total

- Peut-on faire mieux ? Oui, il est possible de proposer un algorithme en $O(n)$ (pas vu dans ce cours)

Exemple 3 : Achat/vente d'actions



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97

- Soit le prix d'une action au cours de n jours consécutifs (prix à la fermeture)
- On aimerait déterminer rétrospectivement :
  - à quel moment, on aurait dû acheter et
  - à quel moment, on aurait dû vendre

  de manière à maximiser notre gain

Exemple 3 : Achat/vente d'actions

Première stratégie :

- Acheter au prix minimum, vendre au prix maximum

Exemple 3 : Achat/vente d'actions

Première stratégie :

- Acheter au prix minimum, vendre au prix maximum
- Pas correct : Le prix maximum ne suit pas nécessairement le prix minimum

Exemple 3 : Achat/vente d'actions

Première stratégie :

- Acheter au prix minimum, vendre au prix maximum
- Pas correct : Le prix maximum ne suit pas nécessairement le prix minimum

Deuxième stratégie :

- Soit acheter au prix minimum et vendre au prix le plus élevé qui suit
- Soit vendre au prix maximum et acheter au prix le plus bas qui précède

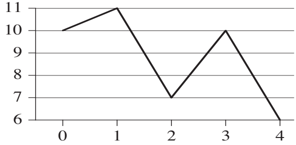
Exemple 3 : Achat/vente d'actions

Première stratégie :

- Acheter au prix minimum, vendre au prix maximum
- Pas correct : Le prix maximum ne suit pas nécessairement le prix minimum

Deuxième stratégie :

- Soit acheter au prix minimum et vendre au prix le plus élevé qui suit
- Soit vendre au prix maximum et acheter au prix le plus bas qui précède
- Pas correct :



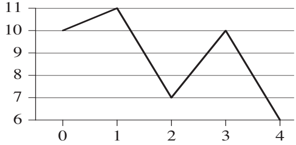
Exemple 3 : Achat/vente d'actions

Première stratégie :

- Acheter au prix minimum, vendre au prix maximum
- Pas correct : Le prix maximum ne suit pas nécessairement le prix minimum

Deuxième stratégie :

- Soit acheter au prix minimum et vendre au prix le plus élevé qui suit
- Soit vendre au prix maximum et acheter au prix le plus bas qui précède
- Pas correct :



Troisième stratégie :

- Tester toutes les paires (force brute)
- Correct ? Complexité ?

Achat/vente d'actions : transformation

Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

- Transformation du problème :
  - Calculer le tableau $A[i] =$ (prix du jour i)-(prix du jour i-1) (de taille $A.length = n$ en supposant qu'on démarre avec un prix au jour 0)
  - Déterminer la sous-séquence non vide contiguë de somme maximale dans A
  - Soit $A[i \mathinner{..} j]$ cette sous-séquence. Il aurait fallu acheter juste avant le jour i (juste après le jour $i-1$) et vendre juste après le jour j.
- Exemple dans le tableau ci-dessus : $A[8 \mathinner{..} 11]$ est la sous-séquence maximale de somme 43 $\Rightarrow$ acheter juste avant le jour 8 et vendre juste après le jour 11.
- Si on peut trouver la sous-séquence maximale dans un tableau, on aura une solution à notre problème d'achat/vente d'actions

Approche par force brute

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

maximum subarray

- Implémentation naïve :
  - On génère tous les sous-tableaux
  - On calcule la somme des éléments de chaque sous-tableau
  - On renvoie les bornes du (d'un) sous-tableau de somme maximale
- Complexité : $\Theta(n^2)$ sous-tableaux et $O(n)$ pour le calcul de la somme d'un sous-tableau $\Rightarrow O(n^3)$
- On peut l'implémenter en $\Theta(n^2)$

Approche par force brute

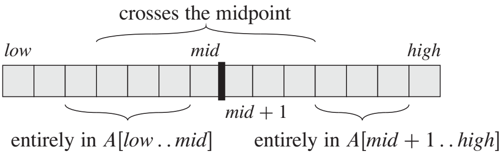
```
MAX-SUBARRAY-BRUTE-FORCE(A)
n = A.length
max-so-far = −∞
for l = 1 to n
        sum = 0
        for h = l to n
                sum = sum + A[h]
                if sum > max-so-far
                        max-so-far = sum
                        low = l
                        high = h
return (low, high)
```

Complexité : $\Theta(n^2)$
Peut-on faire mieux ?

Approche diviser-pour-régner

- Nouveau problème :
  - trouver un sous-tableau maximal dans $A[low\,..\,high]$
  - fonction MAXIMUM-SUBARRAY$(A, low, high)$
- Diviser :
  - diviser le sous-tableau en deux sous-tableaux de tailles aussi proches que possible
  - choisir $mid = \lfloor (low + high)/2 \rfloor$
- Régner :
  - trouver récursivement les sous-tableaux maximaux dans ces deux sous-tableaux
  - appeler MAXIMUM-SUBARRAY(A, low, mid) et MAXIMUM-SUBARRAY$(A, mid + 1, high)$
- Fusionner : ?

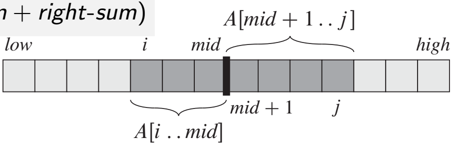
Approche diviser-pour-régner



- Fusionner :
  - Rechercher un sous-tableau maximum qui traverse la jonction
  - Choisir la meilleure solution parmi les 3
- MAX-CROSSING-SUBARRAY$(A, low, mid, high)$
  - Force brute : $\Theta(n^2)$ (car n/2 choix pour l'extrémité gauche, n/2 choix pour l'extrémité droite)
  - Meilleure solution : on recherche indépendamment les extrémités gauche et droite

MAX-CROSSING-SUBARRAY

```
MAX-CROSSING-SUBARRAY(A, low, mid, high)
left-sum = −∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
right-sum = −∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
return (max-left, max-right, left-sum + right-sum)
```



Complexité : $\Theta(n)$

Max-subarray

```
Max-subarray(A, low, high)
if high == low
    return (low, high, A[low])
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) = Max-subarray(A, low, mid)
    (right-low, right-high, right-sum) = Max-subarray(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        Max-crossing-subarray(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Analyse

- Si on suppose que n est un multiple de 2, le nombre d'opérations $T(n)$ est donné par :

$$T(n) = \begin{cases} c_1 & \text{si } n = 1 \\ 2T(n/2) + c_2 n & \text{sinon} \end{cases}$$

- Même complexité que le tri par fusion $\Rightarrow \Theta(n \log n)$
- Peut-on faire mieux ? On verra plus loin que oui

Diviser pour régner : résumé

- Mène à des algorithmes très efficaces
- Pas toujours applicable mais quand même très utile

- Applications :
  - Tris optimaux
  - Recherche binaire
  - Problème de sélection
  - Trouver la paire de points les plus proches
  - Recherche de l'enveloppe convexe (convex-hull)
  - Multiplication de matrice (méthode de Strassens)
  - ...