MARO 037 - Competitive Programming

Fabian Lecron - Arnaud Vandaele

Faculté Polytechnique - Services MIT et MARO

UMONS

Année académique 2024-2025

Séance 03 :
Complete search et Divide and Conquer

Séance 03 – Complete search et Divide and Conquer

Tables des matières

1 Les paradigmes algorithmiques

2 Complete search a.k.a. Brute Force

3 Divide and conquer

Plan

1. Les paradigmes algorithmiques

2. Complete search a.k.a. Brute Force
   - Présentation
   - Quelques outils en Python
   - Exercice
   - Produit cartésien

3. Divide and conquer
   - Présentation
   - Quelques exemples
   - Exercice

Paradigme ?

- Les *stratégies algorithmiques* désignent des approches générales par lesquelles beaucoup de problèmes peuvent être résolus.
- Ces problèmes peuvent être totalement différents les uns des autres
- Ces *stratégies algorithmiques* sont aussi couramment appelées des *paradigmes algorithmiques*

Les paradigmes algorithmiques sont donc des modèles génériques sur lesquels on se base pour construire des algorithmes.

- Plusieurs techniques peuvent souvent résoudre le même problème
- Souvent, une approche est meilleure qu'une autre (complexité)

Dans les deux prochains cours, nous allons passer en revue plusieurs paradigmes parmi les plus importants :

- Complete search (recherche exhaustive)
- Divide and conquer (diviser pour régner)
- Greedy (méthodes gloutonnes)
- Dynamic programming

Plan

1 Les paradigmes algorithmiques

2 Complete search a.k.a. Brute Force
- Présentation
- Quelques outils en Python
- Exercice
- Produit cartésien

3 Divide and conquer
- Présentation
- Quelques exemples
- Exercice

Complete search (recherche exhaustive) / Brute force

La technique *brute force* est une méthode consistant à parcourir l'entiereté de l'espace des solutions afin de trouver la meilleure solution d'un problème donné.

Démarche générale :

- Le problème possède un ensemble fini de solutions
- Parmi ces solutions, on souhaite trouver la (ou les) solution(s) satisfaisant des contraintes ou minimisant un objectif
- S'il est possible d'énumérer ces solutions de manière efficace et que le nombre de solutions n'est pas trop important (suivant le temps imparti), alors la méthode *brute force* peut être envisagée

Deux points importants :

- Trouver la manière la plus efficace d'énumerer ces solutions
- Comment mettre en oeuvre cette énumération ?

Complete search - propriétés

- Généralement, une approche brute force est obtenue lorsqu'on applique à la lettre la définition du problème
- Souvent, cette approche n'est pas efficace en terme de temps de calcul mais possède l'avantage d'être facile à implémenter et fonctionnelle
- La méthode brute force est la première à laquelle il faut penser, et la première à oublier si le nombre de solutions est trop important (comme dans le Reply Challenge (TP1))
- Dans la plupart des cas, il existe une meilleure technique pour obtenir la solution du problèmem, mais, dans certains cas, c'est la seule approche possible

Exemples d'approches par brute force pour le tri d'un tableau :

- Trouver le minimum du tableau, l'échanger avec le premier élement, répéter pour trier le reste du tableau : tri par sélection (complexité : $\mathcal{O}(n^2)$)
- Parcourir le tableau de gauche à droite en échangeant toutes les paires d'éléments consécutifs si nécessaire : tri à bulle (complexité : $\mathcal{O}(n^2)$)

Enumérer efficacement - exemple

Il vaut la peine de réfléchir à comment énumérer efficacement les solutions d'un problème

Enoncé : *Etant donné un vecteur d'entiers de taille n, on souhaite savoir s'il existe deux entiers dans ce vecteur dont la somme vaut S.*

Exemple : $v = [4, 9, 0, 34, 7, 22, 3, 10, 1, 5, 12, 55, 89, 32, 53, 62]$ et $S = 111$

- Solution 1 en $\mathcal{O}(n^2)$
- Solution 2 en $\mathcal{O}(n \log(n))$
- Solution 3 en $\mathcal{O}(n)$

Comment énumérer ? Des outils bien pratiques en Python

- Itérer sur deux listes à l'aide la fonction `zip` :

```
list(zip([1, 2, 3], ['a', 'b', 'c']))
```

- La fonction `map` permet d'appliquer une fonction à chaque élément :

```
list(map(len, ['abc', 'de', 'fghi']))
```

- Générer tous les subsets de k éléments d'un vecteur v (il y en a $\binom{n}{k}$) :

```
import itertools
v = [4, 9, 0, 34, 7]
k = 3
print(list(itertools.combinations(v,k)))
```

- Idem mais avec répétition (il y en a $\binom{n+k-1}{k}$) :

```
import itertools
v = [4, 9, 0, 34, 7]
k = 10
print(list(itertools.combinations_with_replacement(v, k)))
```

- Générer tous les subsets possibles d'un vecteur v (il y en a 2^n) :

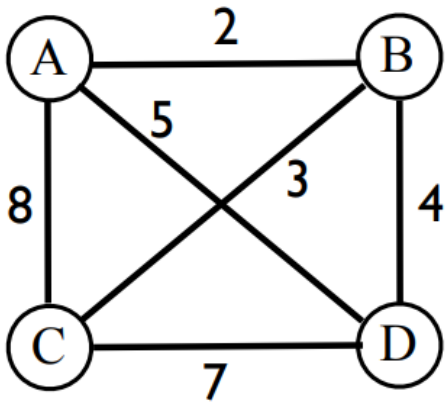
```
import itertools
v = [4, 9, 0, 34, 7]
for k in range(len(v)+1):
    for subset in itertools.combinations(v,k):
        print(subset)
```

- `permutations` pour générer les $n!$ permutations

Enumération - Complexité exponentielle / polynomiale

Le problème du voyageur de commerce

- Etant donnés n villes et les distances entre ces villes
- On souhaite trouver le tour le plus court passant par toutes les villes exactement une fois



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

- Par recherche exhaustive, nombre de solutions à tester : $(n-1)!$
- Personne n'a encore trouvé un algorithme de complexité polynomiale ($\mathcal{O}(n^7)$, $\mathcal{O}(n^2)$, ...) et il y a peu de chance qu'on y arrive

Enumération - Exercices

- *On possède 3 billets de 20 dollars, 5 billets de 5 dollars et 5 billets d'un dollar.*

```
billets = [20, 20, 20, 10, 10, 10, 10, 10, 5, 5, 1, 1, 1, 1, 1]
```

1. Listez les combinaisons uniques si on prend trois billets dans le portefeuille.
2. Listez toutes les sommes possibles
3. Listez toutes les combinaisons possibles menant à 100 dollars.
4. Listez toutes les combinaisons uniques possibles menant à 100 dollars.

- Si on possède autant de billets de 50, 20, 10, 5 et 1 dollars que l'on souhaite, listez toutes les manières de faire 50 dollars. Et 100 dollars ?

Produit cartésien en Python

`itertools.product` renvoie tous les tuples possibles en prenant un élément de chaque liste passée en paramètre.

Exemple simple :

```
import itertools
liste1 = ['Audi','Peugeot','Opel']
liste2 = ['rouge', 'bleue']
for elem in itertools.product(liste1, liste2):
    print(elem)
```

Exemple avec $*$ (unpacking operator) :

```
import itertools
listes = [[1,2,3],[3,4,5],[7,8]]
for elem in itertools.product(*listes):
    print(elem)
```

Exemple avec le paramètre `repeat` :

```
import itertools
for elem in itertools.product([0, 1], repeat=10):
    print(elem)
```

Plan

1 Les paradigmes algorithmiques

2 Complete search a.k.a. Brute Force
- Présentation
- Quelques outils en Python
- Exercice
- Produit cartésien

3 Divide and conquer
- Présentation
- Quelques exemples
- Exercice

Divide and conquer

De nombreux algorithmes sont *récursifs*, c-à-d qu'ils s'appellent eux-mêmes une ou plusieurs fois pour traiter des sous-problèmes très similaires.

Généralement, ces algorithmes suivent une approche *diviser pour régner* : le problème est divisé en plusieurs sous-problèmes plus petits mais semblables au problème initial.

Le principe général est le suivant :

- Si le problème est trivial, il est résolu directement
- Sinon :
  - **Diviser :** le problème est divisé en sous-problèmes de plus petites tailles
  - **Régner :** les sous problèmes sont résolus de façon récursive
  - **Combiner :** on combine les solutions des sous-problèmes pour produire la solution du problème original

Complexité et récurrence : le Master Theorem

(quelques approximations sont faites dans ce slide, voir Master Theorem)

Soit $T(n)$ la complexité d'un problème de taille n. Si $T(n)$ satisfait la récurrence :

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Dans le cas où

- $f(n) = \mathcal{O}(n^c)$ avec $c < \log_b(a)$, alors
$$T(n) = \mathcal{O}(n^{\log_b(a)})$$
- $f(n) = \mathcal{O}(n^c \log^k(n))$ avec $c = \log_b(a)$ et $k \geq 0$, alors
$$T(n) = \mathcal{O}(n^c \log^{k+1}(n))$$

Exemples :

Relation de récurrence	Complexité
$T(n) = T\left(\frac{n}{2}\right) + \mathcal{O}(1)$	$\mathcal{O}(\log(n))$
$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(1)$	$\mathcal{O}(n)$
$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n)$	$\mathcal{O}(n\log(n))$

Un premier exemple connu : binary search / recherche dichotomique

Enoncé : *trouver si un élement x est présent dans un tableau trié*

- Si l'élement central du tableau est x, stop. Sinon :
  - **Diviser :** comparaison de la valeur de l'élément central et de x
  - **Régner :** suivant la valeur de x, on cherche récursivement soit dans le sous-tableau de gauche, soit dans le sous-tableau de droite
  - **Combiner :** soit l'élément est trouvé, soit il n'est pas trouvé

```
def binarySearch(tableau,x):
    premier = 0
    dernier = len(tableau)-1
    found   = False
    while premier<=dernier and not found:
        pos = 0
        milieu = (premier + dernier)//2
        if tableau[milieu] == x:
            pos   = milieu
            found = True
        else:
            if x < tableau[milieu]:
                dernier = milieu-1
            else:
                premier = milieu+1
    return (pos, found)
```

Un deuxième exemple : recherche de pics

Pour un vecteur v de taille n,
l'entrée `v[i]` est un pic si `v[i]`$\geq$`v[i+1]` et `v[i]`$\geq$`v[i-1]`.

- On estime que `v[-1]`$=-\inf$ et `v[n+1]`$=-\inf$
- Il existe toujours au moins un pic
- Le but est d'en trouver un et de renvoyer sa position

Facilement, on construit une approche brute force :

- Toutes les positions sont testées séquentiellement
- La complexité est $\mathcal{O}(n)$
- Autre idée en brute force : le maximum de `v` est un pic. Mais $\mathcal{O}(n)$...
- Peut-on faire mieux ? Oui!

Divide and conquer pour la recherche de pics

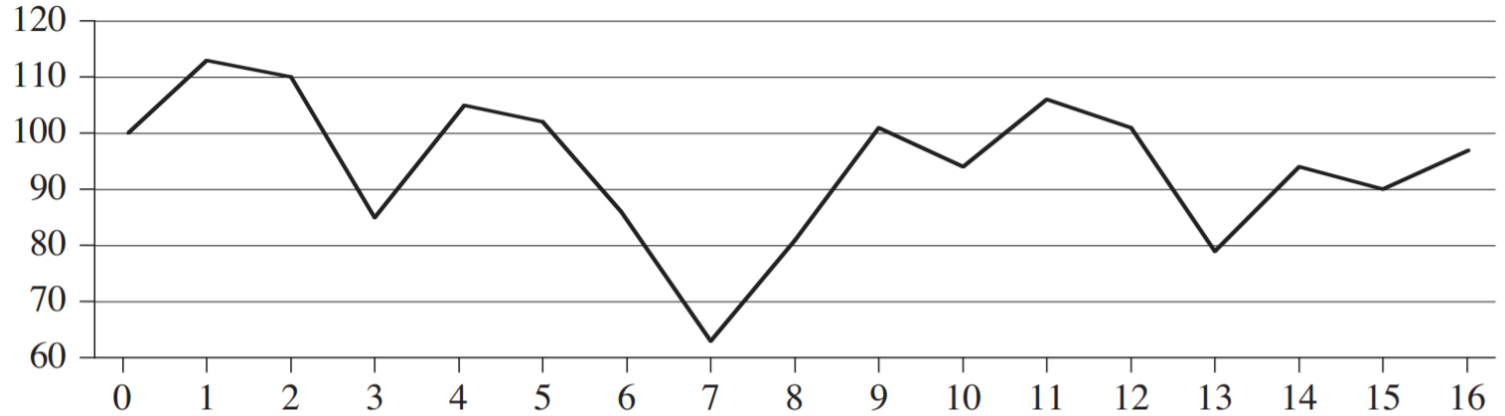
- Si l'élement central du tableau est un pic, stop. Sinon :
  - **Diviser :** comparaison de la valeur de l'élément central avec ses voisins
  - **Régner :** on cherche récursivement dans le sous-tableau correspondant au voisin le plus grand
  - **Combiner :** trivial, un pic sera forcément trouvé

Complexité :

- Idem que la recherche binaire : $\mathcal{O}(\log(n))$

Exemple 3 : achat/vente d'actions

Soit le prix d'une action au cours de n jours consécutifs :



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

Comment déterminer au bout de ces n jours

- à quel moment on aurait du acheter et
- à quel moment on aurait du vendre

de manière à maximiser le gain.

Sous-séquence de somme maximale : brute force

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

Le problème revient à trouver la sous-séquence continue de somme maximale dans un tableau.
Implémentation brute force :

- On génère toutes les sous-séquences, on calcule leur somme et on renvoie celle de somme maximale
- De façon naïve, il y a $\mathcal{O}(n^2)$ sous-séquences et faire la somme coûte $\mathcal{O}(n)$
- Il est quand même possible d'implémenter cette idée en $\mathcal{O}(n^2)$

Exercice : approche divide and conquer pour le problème de sous-séquence de somme maximale

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

Approche divide and conquer :

- **Diviser :** diviser le tableau en deux sous-tableaux
- **Régner :** trouver récursivement les sous-tableaux maximaux dans ces deux sous-tableaux
- **Combiner :** rechercher un sous-tableau maximum contenant le point de jonction et choisir la meilleure solution parmi les 3