Méthodes Algorithmiques

PSI/PSI⋆

I Problèmes Algorithmiques

❦ Définition.

Un *problème algorithmique* est un problème qui peut être résolu par un algorithme. Il est défini par :

— les entrées du problème
— la pré-condition
— la sortie du problème
— la post-condition

Remarque : La spécification d'un algorithme défini le problème que l'algorithme résous.

1.1 Taxonomie

Voici une liste non exhaustive de catégories de problèmes :

1.1.1 Problèmes de recherche

❦ Définition.

Un *problème de décision* est un problème algorithmique dont la réponse est une élément, dans un ensemble fixé, vérifiant une propriété donnée.

Forme générique : On pose $\mathcal{E}$ un ensemble d'entrée et $\mathcal{P}$ une propriété :

Soit $e \in \mathcal{E}$, trouver $x \in e$ tel que $\mathcal{P}(x)$?

Méthodes Algorithmiques Classiques

La recherche la plus simple se fait par un parcours de chaque élément de l'ensemble $\mathcal{E}$ d'entrée.

```
for x in E : # ensemble d'entrée
    if P(x) : # fonction booléenne P
        return x
return None
```

Parmis les autres options, en exploitant les propriétés de la structure de E on peut :

▷ restreindre le domaine de recherche, par élimination ou sélection d'une partie des éléments.
▷ essayer de construire un élément solution.

1.1.2 Problèmes de décision

❦ Définition.

Un *problème de décision* est un problème algorithmique dont la réponse est un *booléen* : vrai ou faux, oui ou non.

Forme générique : On pose $\mathcal{E}$ un ensemble d'entrée et $\mathcal{P}$ une propriété :

Soit $e \in \mathcal{E}$, est-ce que $\mathcal{P}(e)$?

Méthodes Algorithmiques Classiques

Certains problèmes de décision sont dérivés d'un problème de recherche :

Soit $e \in \mathcal{E}$, est-ce qu'il existe $x \in e$ t.q. $\mathcal{P}(x)$?

ou dans sa version négative :

Soit $e \in \mathcal{E}$, est-ce que pour tout $x \in e$, $\mathcal{P}(x)$?

Il suffit alors de renvoyer un booléen au lieu d'une valeur, `True` si on a trouvé une valeur, `False` sinon. Pour la version négative, on se ramène au premier cas en remarquant que $\forall x \in e, \mathcal{P}(x) \iff \nexists x \in$ E, $\neg\mathcal{P}(x)$, on obtient :

```
for x in e :
    if P(x) :
        return True
return False
```

Pour la version négative :

```
for x in e :
    if not P(x) :
        return False
return True
```

1.1.3 Problèmes de dénombrement

❦ Définition.

Un *problème de dénombrement* est un problème algorithmique dont la réponse consiste à compter le nombre de réponses possible à un problème de recherche.

Forme générique : On pose $\mathcal{E}$ un ensemble d'entrée et $\mathcal{P}$ une propriété :

Soit $e \in \mathcal{E}$, compter le nombre de $x \in e$ tel que $\mathcal{P}(x)$?

Méthodes Algorithmiques Classiques

On peut adapter l'algorithme solution du problème de recherche pour incrémenter un compteur et continuer la recherche. Dans le cas d'un parcours des éléments, il suffit de le continuer. Une solution générique consisterait à :

1. Réaliser une recherche la recherche
2. Si on trouve un élément x :
   (a) On incrémente un compteur
   (b) On supprime x de la structure
   (c) On relance une recherche
3. Sinon on termine.

Dans le cas d'une recherche linéaire, l'algorithme s'adapte alors de la solution du problème de recherche :

```
n = 0
for x in E :
    if P(x) : # Condition sur x, ou fonction booléenne P
        n += 1
return n
```

1.1.4 Problèmes de d'optimisation

❦ Définition.

Un *problème d'optimisation* est un problème algorithmique dont la réponse consiste à trouver la meilleure réponse possible à un problème de recherche.

Forme générique : On pose $\mathcal{E}$ un ensemble d'entrée et, γ une fonction de $\mathcal{P}$ une propriété :

Soit $e \in \mathcal{E}$, compter le nombre de $x \in e$ t.q. $\mathcal{P}(x)$ et $\gamma(x) = \min\{y \in e \mid \mathcal{P}(y)\}$?

Méthodes Algorithmiques Classiques

Une première approche consiste à trouver toutes les solutions du problème de recherche, puis à faire une recherche de minimum pour la fonction γ associée au problème. Facile à implémenter, mais souvent trop coûteuse.

Il existe deux autres techniques classiques :

▷ *Algorithmes Gloutons*

Souvent efficace, mais répond rarement au problème. On l'utilise pour obtenir une solution proche de l'optimale.

▷ *Programmation Dynamique*

Souvent plus efficace que la Force Brute, mais moins qu'un algorithme glouton. Coûteuse en espace.

1.1.5 Problèmes de classification

❦ Définition.

Un *problème de classification* est un problème algorithmique dont la réponse consiste à déterminer la catégorie à laquelle appartient l'entrée.

Forme générique : On pose $\mathcal{E}$ un ensemble d'entrée et k un nombre fini de catégorie :

Soit $e \in \mathcal{E}$, renvoyer $0 \leq i \leq k$ tel que e soit dans la catégorie k ?

Méthodes Algorithmiques Classiques

Lorsque le problème de classification peut se formuler sous la forme :

$$\text{Soit } e \in \mathcal{E}, \text{ renvoyer } c \text{ tel que } d(e,c) = \min\{d(e,i), 0 \leq i \leq k\}$$

avec $d(e,i)$ une notion de distance entre l'entrée et la catégorie, on peut se ramener à un problème d'optimisation.

Voici quelques outils classiques

- ▷ Arbres de décision.
- ▷ Algorithme des k-moyennes
- ▷ Algorithme des k-voisins
- ▷ Réseaux de neurones

1.2 Transformation et Réduction de problèmes

Étant donné un problème algorithmique, on peut parfois le ramener à un autre problème algorithmique dont on connaît la solution :

- ▷ Par transformation éventuelle de l'entrée (Transformation),
- ▷ Puis par reformulation du problème (Réduction),

1.3 Sous-problèmes

❦ Définition.

Étant donné un problème algorithmique, on définit un *sous-problème* comme étant un nouveau problème informatique *de même nature* et *de taille plus petite*.

Un très grand nombre d'algorithmes sont basés sur la décomposition d'un problème en un ensemble de *sous-problèmes* dont la résolution permet d'aboutir à la résolution du problème en lui-même.

⚛ Exemples.

Entrée : Une liste d'entiers L de taille n.
Problème : M = calculer le maximum de la liste.
Sous-problèmes : $\forall 0 \leq i \leq n$, M_i = calculer le maximum des i premiers éléments de L

Résolution :

— $M = M_n$,
— $M_0 = -\infty$
— $M_{i+1} = \max\{L[i+1], M_i\}$.

❧❧❧

❦ Définition.

Étant donnés un problème algorithmique et une décomposition en sous-problèmes on dit que deux *sous-problèmes,* R et P sont *dépendants* s'il existe un sous-problème Q nécessaire à la résolution de R et P. On dit alors qu'il y a *chevauchement*.

⚛ Exemples.

- Soit F la suite de Fibonacci. Le calcul de F_2 est nécessaire pour calculer F_3 et F_4. Il y a chevauchement.
- $\binom{n}{k}$ est nécessaire pour calculer $\binom{n+1}{k}$ et $\binom{n+1}{k+1}$. Il y a chevauchement.

❧❧❧

1.3.1 Approche Montante v.s. Descendante

❦ Définition.

Étant donnés un problème algorithmique et une décomposition en *sous-problèmes,* une résolution par *approche montante* consiste à partir d'un sous-problème petit et facile à résoudre, puis de résoudre des sous-problèmes de plus en plus grand en combinant les solutions des sous-problèmes déjà résolus.

Sauvegarde des solutions

L'approche montante pose naturellement la question de la sauvegarde des solutions : il faut sauvegarder les solutions des petits sous-problèmes pour calculer les problèmes plus gros.

```
# Approche montante
def maximum(l) :
    m = -float("inf") # Premier sous-problème
    # Invariant : m contient la solution du
    # sous problème de taille i
    for i in range(n) :
        # Résolution du sous-problème i+1 à partir
        # du sous-problème i déjà calculé
        if l[i] > m :
            m = l[i]
    # Arrêt au sous-problème n
    return m
```

❦ Définition.

Étant donnés un problème algorithmique et une décomposition en *sous-problèmes*, une résolution par *approche descendante* consiste à partir du problème lui-même, puis de déterminer les sous-problèmes nécessaires à la résolution du problème, d'en calculer les solutions, immédiatement ou de manière descendante, puis de combiner ces solutions pour résoudre le problème.

Sauvegarde des solutions

L'approche descendante élude naturellement cette question. La résolution des sous-problèmes de tailles inférieures est laissée aux appels récursifs et chaque appel ne se concentre que sur le calcul de la solution du problème à partir des sous-problèmes identifiés.

```
# Approche descendante
def maximum(l) :
    # Résolution du sous-problème 0,
    #   exécuté uniquement au dernier appel récursif
    if l = [] :
        return -float("inf")
    # Résolution du sous-problème i-1
    m = maximum(l[1:])
    # Résolution du problème à partir du sous-problème plus petit
    if l[0] > m :
        return l[0]
    else :
        return m
```

1.3.2 Dépendance et Mémoïsation

Étant donnés un problème algorithmique et une décomposition en *sous-problèmes*, quelque soit l'approche de résolution choisie, la solution à un sous-problème est obtenue en combinant la solution de sous-problèmes plus petits.

Supposons que les sous-problèmes ne soient pas tous indépendant. Il peut alors arriver que la solution d'un sous-problème soit calculée deux fois pour la résolution de deux sous-problèmes plus grand différents.

On peut réduire considérablement la complexité de la solution à un problème algorithmique en sauvegardant la solution d'un sous-problème déjà résolu.

Approche Montante

L'approche montante par du principe que la solution des sous-problèmes plus petits a déjà été calculée et sauvegarder. Il est donc naturel de ne pas recalculer deux fois la solution problème.

Approche Descendante et Mémoïsation

Le problème du recalcul de solutions apparaît très rapidement lors de l'approche descendante vu qu'elle élude naturellement la question de la sauvegarde des solutions aux sous-problèmes.

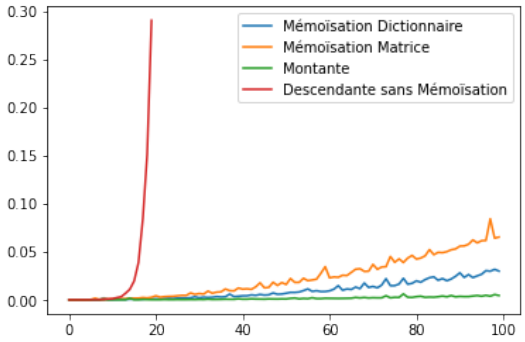
❦ Définition.

> La mémoïsation consiste à sauvegarder la sortie d'une fonction dans une structure adaptée pour éviter de la relancer deux fois sur la même entrée.
>
> Dans le cadre d'une fonction récursive, la mémoïsation évite que deux appels récursifs soient lancé sur la même entrée.

Pour être mise en oeuvre, la mémoïsation nécessite une structure de donnée *persistante* à travers tous les appels de la fonction, *i.e* une structure de donnée qui existe en dehors de la fonction et que la fonction peut modifier.

En Python on utilisera une liste, un tableau ou un dictionnaire que *l'on passera en paramètre de toutes nos fonctions*. La structure doit pouvoir associer à chaque entrée possible, *i.e* chaque sous-problème, deux informations : est-ce que le sous-problème a déjà été résolu ? si oui quel est sa solution.

C'est souvent l'approche par dictionnaire que l'on privilégiera.



Évolution du temps de calcul du binôme de Newton en fonction de n

II Paradigmes Algorithmiques

2.1 Recherche Exhaustive, Méthode Naïve ou Méthode par Force Brute

On considère un problème algorithmique dont l'algorithme solution doit renvoyer, pour chaque entrée, un élément parmi un ensemble fini d'éléments $\mathcal{S}$. On suppose alors qu'il est facile, étant donné une entrée e et un élément $s \in \mathcal{S}$ de vérifier si s est la sortie correcte sur l'entrée e.

Une méthode consiste alors à tester pour tout $s \in \mathcal{S}$ si s est la sortie correcte associée à e, puis de la renvoyer.

Cette technique se résume donc à *tester toutes les sorties possibles pour chaque entrée jusqu'à trouver la bonne*. Elle porte plusieurs noms dans la littérature :

- ▷ Recherche exhaustive de solutions
- ▷ Méthode naïve
- ▷ Méthode par force brute, de l'anglais *brute force*.

Cette méthode est souvent inefficace, l'ensemble des sorties possible étant trop grand. Toutefois, il n'existe parfois pas de meilleure solution.

Les problèmes pour lesquels la méthode est envisageables sont les problèmes de recherche, d'optimisation ou de dénombrement.

Analyse

- ▷ *Terminaison* : Terminaison de la vérification et ensemble de sorties testées fini.
- ▷ *Correction* : Preuve que la sortie est dans $\mathcal{S}$ nécessairement. Preuve de correction de la vérification.
- ▷ *Complexité* : Complexité de la vérification multipliée par le cardinal de l'ensemble de sorties possibles

2.2 Méthode Itérative

On considère un problème algorithmique $\mathcal{P}$, que l'on peut découper en sous-problèmes $\mathcal{P}_i$ de telle manière que :

- ▷ Pour chaque entrée, $\exists n \in \mathbb{N}, \mathcal{P} = \mathcal{P}_i$
- ▷ On peut calculer $\mathcal{P}_0$ et $\mathcal{P}_{i+1}$ à partir de $\mathcal{P}_i$

La méthode itérative consiste alors à calculer, de manière itérative ou récursive selon qu'on utilise l'approche montante ou descendante, la solution à chaque sous-problème $\mathcal{P}_i$ un par un jusqu'à obtenir $\mathcal{P}$.

Analyse

- ▷ *Terminaison* : Terminaison du calcul de $\mathcal{P}_{i+1}$ à partir de $\mathcal{P}_i$ et preuve que $\exists n \in \mathbb{N}, \mathcal{P} = \mathcal{P}_i$
- ▷ *Correction* : Invariant de boucle ou récurrence selon l'approche montante ou descendante. Le but est d'arriver à montrer que le calcul à chaque itération permet de passer de la solution de $\mathcal{P}_i$ à la solution de $\mathcal{P}_{i+1}$
- ▷ *Complexité* : Complexité du calcul de $\mathcal{P}_{i+1}$ à partir de $\mathcal{P}_i$ multipliée n.

2.3 Diviser Pour Régner

On considère un problème algorithmique $\mathcal{P}$, que l'on peut découper en sous-problèmes de telle manière que :

- ▷ On puisse résoudre un ensemble de sous-problèmes de taille minimale.
- ▷ Chaque sous-problème $\mathcal{S}$ puisse être divisé en un nombre k de sous-problème $\mathcal{P}_1, \dots, \mathcal{P}_k$, dont il suffit de combiner les solutions pour résoudre le sous-problème $\mathcal{S}$.

Pour que la méthode soit efficace on ajoutera la condition suivante :

- ▷ $\mathcal{P}_1, \dots, \mathcal{P}_k$ sont indépendants.

La méthode *diviser pour régner* consiste alors à utiliser une approche descendante pour résoudre le problème :

1. On découpe le problème en k sous-problèmes.
2. On résout récursivement les k sous-problèmes qui sont indépendants, avec la même approche.

3. On combine les solutions des k sous-problème pour obtenir la solution du problème.

Analyse

- ▷ *Terminaison* : Immédiate si la tailles des sous-problèmes est plus petite.
- ▷ *Correction* : Par récurrence. Il faut prouver la correction des cas initiaux et de la combinaison des solutions.
- ▷ *Complexité* : Résolution de l'équation

$$\mathrm{T}(n) = \begin{cases} \Theta(1) & \text{si } n = 0 \\ \sum_{i=0}^{k} \mathrm{T}(n_i) + \mathrm{D}(n) + \mathrm{C}(n) & \text{sinon} \end{cases}$$

Avec n_i la taille du sous-problème i. D la complexité de la division en k sous-problèmes et C la complexité de la combinaison. Dans le cas où les k sous-problèmes sont de tailles égales p on se ramène à

$$\mathrm{T}(n) = \begin{cases} \Theta(1) & \text{si } n = 0 \\ k\,\mathrm{T}(p) + \mathrm{D}(n) + \mathrm{C}(n) & \text{sinon} \end{cases}$$

2.4 Algorithmes Gloutons

On considère un problème *d'optimisation* $\mathcal{P}$, que l'on peut découper en sous-problèmes de telle manière que :

- ▷ On puisse résoudre un ensemble de sous-problèmes de taille minimale.
- ▷ Le problème vérifie la propriété de **sous-problème optimal**, *i.e* la solution optimal à un sous-problème peut être obtenue à partir de la solution optimale de sous-problèmes de taille inférieure.
- ▷ La solution optimale globale peut-être obtenue par une approche descendante en faisant un choix optimal local quant au sous-problème qui doit être résolu à l'étape suivante. (∗)

Si (∗) n'est pas vérifiée, l'algorithme obtenu n'est pas solution du problème. Il est toutefois courant que l'algorithme glouton permette d'approcher la solution optimale, tout en étant plus efficace qu'une recherche exhaustive ou une méthode par programmation dynamique. On appelle alors cet algorithme : *heuristique gloutonne*.

La méthode *gloutonne* consiste alors à construire de manière itérative la solution optimale en faisant toujours un choix optimal à chaque étape.

⚛ Exemples.

Algorithmes gloutons optimaux :

— Le problème du rendu de monnaie

— Le problème de choix d'activité

⚛ Exemples.

Heuristiques gloutonnes :

— ordonnancement
— codage de Huffman
— problème du voyageur de commerce

Analyse

▷ *Terminaison* : Montrer que l'itération termine.
▷ *Correction* : Montrer (∗)
▷ *Complexité* : Nombre d'itération multiplié par la complexité du choix optimal à chaque étape

2.5 Programmation Dynamique

On considère un problème **d'optimisation** $\mathcal{P}$, que l'on peut découper en sous-problèmes de telle manière que :

▷ On puisse résoudre un ensemble de sous-problèmes de taille minimale.
▷ Le problème vérifie la propriété de **sous-problème optimal**, *i.e* la solution optimal à un sous-problème peut être obtenue à partir de la solution optimale de sous-problèmes de taille inférieure.
▷ Il y a chevauchement des sous-problèmes

La méthode *par programmation dynamique* consiste alors à construire de manière itérative la solution optimale en combinant les solutions optimales des sous-problèmes et en sauvegardant les solutions de chaque sous-problème déjà résolus. On utilise souvent une approche descendante qui permet de choisir les sous-problèmes nécessaire au calcul de la solution optimale, et une technique de mémoïsation pour gérer les chevauchement de sous-problèmes.

Analyse

▷ *Terminaison* : Montrer que l'itération termine.
▷ *Correction* : Montrer que la combinaison des solutions mènent bien à la solution optimale.
▷ *Complexité* : Nombre d'itération multiplié par la complexité du calcul de la combinaison des solutions. Il faut ajouter à cela le temps de construction de la structure de sauvegarde des solutions aux sous-problème.