Notions sur les réductions

Notes de cours de complexité des problèmes, M1

1 Objectif

Le but de ce chapitre est de montrer comment obtenir des informations sur un problème donné P, à partir d'informations qu'on connait déjà sur un autre problème Q. C'est un outil fondamental pour tout le cours : l'idée est de n'étudier en détails que certains problèmes clés, puis de se ramener à ceux là quand on tombe sur un nouveau problème.

Prenons l'exemple du problème `Distinctness`, qui consiste à décider si un tableau contient des valeurs deux à deux distinctes. On peut résoudre ce problème naîvement en temps $\mathcal{O}(n^2)$, en faisant deux boucles imbriquées. Mieux, on peut utiliser un algorithme de tri en $\mathcal{O}(n \log n)$ pour commencer : les doublons éventuels sont alors situés les un à côté des autres dans le tableau trié.

```
def distinctness(T):
  T.sort()
  for i in range(len(T)-1):
    if T[i] == T[i+1]:
      return False
  return True
```

Complexité : $\mathcal{O}(n) + \text{Tri} \to \mathcal{O}(n \log n)$

On dira qu'on a *réduit* en temps linéaire le problème `Distinctness` à un problème de tri (`Sort`). Et on va le noter

$$\texttt{Distinctness} \lll_{\mathcal{O}(n)} \texttt{Sort}.$$

2 Cadre général

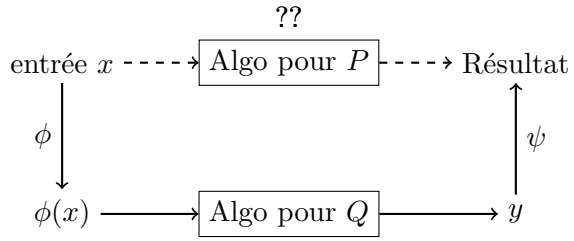
▶ **Définition.** Soient P et Q deux problèmes. On dit que P se réduit à Q en temps $\mathcal{O}(T_n)$ quand on peut résoudre P en utilisant un nombre borné de fois un algorithme qui résout Q, avec un surcoût en $\mathcal{O}(T_n)$. On le note : $P \lll_{\mathcal{O}(T_n)} Q$. ◀

Plus précisément, il faut procéder aux étapes suivantes :

1. Transformer l'entrée x du problème P en une entrée $\phi(x)$ du problème Q.
2. Exécuter l'algorithme $\mathcal{A}$ pour résoudre Q sur $\phi(x)$; on note y le résultat.
3. Construire la solution $\psi(y)$ de P à partir de la solution du point précédent.

La complexité de cet algorithme pour résoudre P dépend donc de la complexité de $\mathcal{A}$ pour des entrées de type $\phi(x)$ et du surcoût T_n, le coût pour calculer $\phi(x)$ et $\psi(y)$.

On peut représenter la réduction par un dessin :



Remarque. Dans l'exemple de réduction de `Distinctness` à `Sort`, on utilise le tableau x donné en entrée à `Distinctness` directement dans `Sort`. Donc ϕ est l'identité, et il n'y a pas de problème de changement de taille (autant d'éléments plus grands que de plus petits).

Quelques exemples sont listés ci-dessous. `Min` consiste à trouver le minimum d'un tableau et `Median` sa médiane.

- `Distinctness` $\lll_{\mathcal{O}(n)}$ `Sort`
- `Min` $\lll_{\mathcal{O}(1)}$ `Sort`
- `Median` $\lll_{\mathcal{O}(1)}$ `Sort`

Si on considère l'algorithme suivant :

```
def 2SUM(T):
  T.sort()
  return 2SUMSorted(T)
```

On peut le voir de deux façons différentes, selon que le considère la réduction à un problème de tri ou au problème `2SUMSorted`:

- On se ramène à un tri : `2SUM` $\lll_{\mathcal{O}(n)}$ `Sort`, car le post-traitement, l'appel à `2SUMSorted`, se fait en temps $\mathcal{O}(n)$, il y a donc bien un surcoût linéaire.
- On se ramène à un `2SUMSorted` : `2SUM` $\lll_{\mathcal{O}(n \log n)}$ `2SUMSorted`, car le pré-traitement, l'appel à un tri, se fait en temps $\mathcal{O}(n \log n)$.

Attention : pour l'algorithme suivant, on *n'a pas réduit* `3SUM` à `2SUM` car on appelle `2SUM` un nombre non borné de fois.

```
def 3SUM(T):
  T.sort()
  for z in T:
    if 2SUM(T,-z):
      return True
  return False
```

Plusieurs appels à un algorithme pour Q : On veut résoudre le problème qui consiste à vérifier à la fois les deux conditions sur un tableau de nombres :

- Il y a au moins un doublon parmi les nombres strictement négatifs ;
- Il n'y a pas de doublon parmi les positifs ou nuls.

```
def f(T):
  P, N = [], []
  for x in T:
    if x < 0:
        N.append(x)
    else:
      P.append(x)
  return Distinctness(P) and not Distinctness(N)
```

On a utilisé deux fois `Distinctness` pour le résoudre, c'est valide pour une réduction (nombre borné de fois) : $P \lll_{\mathcal{O}(n)} \texttt{Distinctness}$.

Changement significatif de la taille de l'entrée : Comme il arrive que $\phi(x)$ soit de taille significativement plus grande que x, on peut vouloir le préciser en écrivant :

$$P(n) \lll_{\mathcal{O}(T_n)} Q(f_n),$$

pour dire qu'on peut résoudre P sur des entrées de tailles n en utilisant un algorithme pour résoudre Q sur des entrées de taille f_n, avec un surcoût en $\mathcal{O}(T_n)$. Par exemple, si on cherche à résoudre `3SUM` en créant un tableau trié avec toutes les sommes possibles de deux éléments, puis à chercher par dichotomie si $-x$ est dedans pour tout x dans T:

```
def 3SUM(T):
  TT = [x+y for x in T for y in T]
  TT.sort()
  for z in T:
    if Dichotomie(TT,-z):
      return True
  return False
```

On a $\texttt{3SUM}(n) \lll_{\mathcal{O}(n \log n)} \texttt{Sort}(n^2)$, puisque l'on trie un tableau qui contient n^2 éléments, le surcoût étant constitué des n recherches dichotomiques en $\mathcal{O}(\log n^2) = \mathcal{O}(\log n)$. Il est important ici de noter que le tri s'effectue sur un tableau de taille n^2, car du coup cela va couter au moins $\Omega(n^2 \log n)$ (par comparaisons).

3 Utilisation des réductions (important!)

Plaçons nous dans le cas où $P \lll_{\mathcal{O}(T_n)} Q$ et où T_n est petit devant les autres complexité. On a deux utilisations possibles de cette information :

- L'utilisation classique est de trouver une solution au problème P en utilisant une solution au problème Q. Si Q se résout en $\mathcal{O}(f_n)$, alors P se résoud en $\mathcal{O}(f_n + T_n) = \mathcal{O}(f_n)$. On cherche donc **des informations sur** P.
- La nouvelle utilisation est dans l'autre sens. Si on sait que P est difficile à résoudre, en $\Omega(f_n)$, alors on peut en déduire que Q est aussi en $\Omega(f_n)$. On cherche donc **des informations sur** Q.

Par exemple, si $P \lll_{\mathcal{O}(n)} Q$ et que le problème P a une borne inférieure de complexité en $\Omega(n \log n)$, alors Q aussi: par l'absurde, s'il y avait un algorithme pour résoudre Q dont la complexité n'est pas dans $\Omega(n \log n)$, alors on pourrait s'en servir pour faire un algorithme pour P dont la complexité n'est pas en $\Omega(n \log n)$; c'est impossible, cela contredirait la borne inférieure sur le problème P. Donc Q a une borne inférieure en $\Omega(n \log n)$.

4 Retour sur `3SUM`

4.1 Réduire `3SUM` à un autre problème

Ce problème va nous servir d'exemple pour obtenir des bornes inférieures. Il est souvent utilisé pour cela, notamment en géométrie algorithmique. Supposons que `3SUM` en en $\Omega(n^2)$. Alors si on a un problème P tel que pour un $0 \leq \alpha < 2$

$$\texttt{3SUM} \lll_{\mathcal{O}(n^\alpha)} P,$$

alors P est en $\Omega(n^2)$. Donc, **sous l'hypothèse que `3SUM` est quadratique**, on peut s'en servir pour obtenir des bornes inférieures quadratiques.[1]

Du coup `3SUM` va être un de nos problèmes de base pour obtenir des bornes inférieures. Il y en aura d'autres, comme par exemple le tri par comparaisons, dont on admet pour l'instant qu'il admet une borne inférieure en $\mathcal{O}(n \log n)$. Faire des réductions de l'un ou l'autre de ces problèmes va consister en les réduire (`3SUM` ou `Sort`) en le problème P qu'on souhaite étudier. Autrement dire, trouver un encodage des donnés du tableau de sorte que, si on avait une solution efficace pour P, on pourrait s'en servir pour résoudre `3SUM` ou `Sort`.

4.2 Un exemple de réduction

▶ **Définition.** Le problème `3SUM'` consiste, étant donné trois ensembles de nombres A, B et C tous de taille n, à déterminer s'il existe $a \in A$, $b \in B$ et $c \in C$ tels que $a + b = c$. ◀

Vous avez vu en TD que `3SUM'` $\lll_{\mathcal{O}(n)}$ `3SUM` et `3SUM` $\lll_{\mathcal{O}(n)}$ `3SUM'`. On va s'intéresser à. une restriction de ce problème :

▶ **Définition.** Le problème `3SUM'01` consiste, étant donné trois ensembles de nombres de $]0, 1[$ nommés A, B et C, tous de taille n, à déterminer s'il existe $a \in A$, $b \in B$ et $c \in C$ tels que $a + b = c$. ◀

On a la réduction : `3SUM'` $\lll_{\mathcal{O}(n)}$ `3SUM'01`. En effet, on utilise le programme suivant.

```
def 3SUMp(A,B,C):
  m = 1-min(min(A),min(B),min(C))
  AA = [x+m for x in A]
  BB = [x+m for x in B]
  CC = [x+2*m for x in C]
  M = max(max(AA),max(BB),max(CC))+1
  AAA = [x/M for x in AA]
  BBB = [x/M for x in BB]
  CCC = [x/M for x in CC]
  return 3SUMp01(AAA,BBB,CCC)
```

On vérifie facilement que AAA, BBB et CCC ne contiennent que des nombres entre 0 et 1. De plus,

$$\begin{aligned} &A[i] + B[i] = C[i] \Leftrightarrow \\ &(A[i] + m) + (B[i] + m) = (C[i] + 2m) \\ &\quad \Leftrightarrow AA[i] + BB[i] = CC[i] \Leftrightarrow \\ &\frac{AA[i]}{M} + \frac{BB[i]}{M} = \frac{CC[i]}{M} \\ &\quad \Leftrightarrow AAA[i] + BBB[i] = CCC[i] \end{aligned}$$

Passons au problème qui nous intéresse :

▶ **Définition.** Le problème `EqDist` consiste, étant donné deux ensembles de nombres P et Q tous deux de taille n, si il existe $p_1 \neq p_2 \in P$ et $q_1 \neq q_2 \in Q$ tels que $p_2 - p_1 = q_2 - q_1$. ◀

On veut montrer le résultat suivant :

Théorème 1 *On a la réduction `3SUM'` $\lll_{\mathcal{O}(n)}$ `EqDist`.*

D'après ce qui précède, il suffit de montrer que `3SUM'01` $\lll_{\mathcal{O}(n)}$ `EqDist`.

On part donc de trois tableaux A, B et C de $]0, 1[$ qui contiennent chacun n nombres. Voilà un exemple **qui ne marche pas** : prendre

$$\begin{aligned} P &= A \cup \{-b \mid b \in B\} \\ Q &= C \cup \{0\} \end{aligned}$$

Ca peut sembler une bonne idée car si $a+b = c$, alors on peut prendre $p_1 = -b$, $p_2 = a$ dans P et $q_2 = c$, $q_1 = 0$ dans Q et obtenir $p_2 - p_1 = q_2 - q_1$. Donc si `3SUM'01` admet une solution, `EqDist`

[1] En fait cette hypothèse, longtemps conjecturée, a été prouvée fausse en 2014, voir plus loin. Mais on peut remplacer n^2 par $n^{2-\epsilon}$ avec $\epsilon > 0$ aussi petit que l'on veut pour avoir une hypothèse plus raisonnable (on ne l'a pas fait pour ne pas alourdir le propos, qui est vraiment cette idée de réduction pour montrer des bornes inférieures).

aussi. Malheureusement la réciproque est fausse : pour $A = \{\frac{1}{5}\}$, $B = \{\frac{1}{5}\}$ et $C = \{\frac{3}{5}, 1\}$, on a $P = \{\frac{1}{5}, -\frac{1}{5}\}$ et $Q = \{0, \frac{3}{5}, 1\}$ et

$$\frac{1}{5} - \left(-\frac{1}{5}\right) = 1 - \frac{3}{5}$$

Donc `EqDist` retourne vrai et pourtant `3SUM'01` retourne faux : un tel algorithme ne calcule pas ce qu'il faut, ce n'est pas une réduction correcte.

Une réduction correcte consiste à construire les deux ensembles P et Q suivants :

$$P = \{100i \mid i \in \{0, ..., n-1\}\} \cup \{100i + 3 - C[i] \mid i \in \{0, ..., n-1\}\}$$
$$Q = A \cup \{3 - b \mid b \in B\}$$

Si $a + b = c$ alors on a bien, avec $c = C[i]$,

$$a - (3 - b) = 100i - (100i + 3 - c).$$

Réciproquement supposons $p_2 - p_1 = q_2 - q_1$. Comme les nombres de A, B et C sont dans $]0, 1[$, on a que

$$-3 \leq q_2 - q_1 \leq 3$$

Et si on prend $i \neq j$ l'écart entre $100i$ et $100j$ est au moins 100, l'écart entre $100i + 3 - C[i]$ et $100j + 3 - C[j]$ est au moins 99 et entre $100i$ et $100j + 3 - C[j]$ au moins 97. Donc l'hypothèse implique que $p_1, p_2 \in \{100i, 100i + 3 - C[i]\}$ pour un certain i. Et donc

$$p_2 - p_1 = \pm(3 - C[i]) \Rightarrow |p_2 - p_1| > 2 \Rightarrow |q_2 - q_1| > 2$$

Cette dernière inégalité implique que q_1 et q_2 ne peuvent pas venir tous les deux de A ou tous les deux de B. Donc il existe $a \in A$ tel que $q_2 = a$ et $b \in B$ tel que $q_1 = 3 - b$ (ou le contraire). Dans les deux cas, on a bien que $a + b = c$.

4.3 Un autre exemple, géométrique

▶ **Définition.** Le problème `3ALIGN` consiste, étant donné une liste de n points P, à déterminer s'il existe trois points distincts de P qui sont alignés. ◀

Rappel: trois points A, B et C sont alignés ssi leur produit vectoriel $\vec{AB} \wedge \vec{AC}$ est nul. Comme les autres composantes du produit vectoriel sont nulles, il suffit de regarder la troisième coordonnée, qui vaut $(x_B - x_A)(y_C - y_A) - (x_C - x_A)(y_B - y_A)$. On en déduit le résultat suivant :

Théorème 2 *Les points distincts (a, a^3), (b, b^3) et (c, c^3) sont alignés ssi $a + b + c = 0$.*

Démonstration : D'après la formule du produit vectoriel, ils sont alignés ssi

$$\begin{aligned}(b-a)(c^3-a^3) = (c-a)(b^3-a^3) &\Leftrightarrow (b-a)(c-a)(c^2+ac+a^2) = (b-a)(c-a)(b^2+ba+a^2)\\ &\Leftrightarrow c^2+ac+a^2 = b^2+ba+a^2\\ &\Leftrightarrow c^2-b^2+ac-ba = 0\\ &\Leftrightarrow (c-b)(b+c)+a(c-b) = 0\\ &\Leftrightarrow b+c+a = 0\end{aligned}$$

□

On peut s'en servir pour l'algo de réduction:

```
def 3SUM(T):
  if 0 in T: # cas a=b=c
        return True
  T.sort() # cas a=b
  for x in T:
    if Dichotomie(T,-2*x):
      return True
  return 3ALIGN([(x,x**3) for x in T]
```

Il faut juste traiter à part les cas (autorisés dans `3SUM`) où l'on prend un même indice plusieurs fois, car le théorème n'est vrai que pour a, b et c distincts.