

Rapport TP3 ACT

Gaspar Henniaux - Marwane Ouaret

github :(<https://github.com/pargass/ACT-TP/tree/main/tp3>)

1. Qu'est-ce qu'une propriété NP ?

Question 1

1.

Ici, un certificat est une association pour chaque objet à un sac.

Par exemple : pour une liste de 5 objets de poids [4,2,2,1,4] et 3 sac de capacité 6, un certificat pourrait être {1:1, 2:2, 3:2, 4:3, 5:1}.

Ici le premier objet est dans le sac 1, le deuxième et troisième dans le sac 2, le quatrième dans le sac 3 et le cinquième dans le sac 1.

On peut utiliser un dictionnaire dont les clés seront les objets et les valeurs les sacs.

Par conséquent, la taille d'un certificat est n , le nombre d'objet. Cette taille est bien bornée polynomialement par rapport à la taille de l'entrée car n est la taille de l'entrée.

```
fonction verif_sac(certificat, n, poids, c, k):  
  
    if len(certificat) != n:  
        retourner faux  
  
    somme : dictionnaire  
    pour chaque objet dans certificat:  
        si certificat[objet] n'est pas dans somme:  
            somme[certificat[objet]] = poids[objet]  
        sinon:  
            somme[certificat[objet]] += poids[objet]  
  
    if len(somme) != k:  
        retourner faux  
  
    pour chaque sac dans somme:  
        si somme[sac] > c:  
            retourner faux  
  
    retourner vrai
```

PROF

On passe n fois dans la boucle pour remplir le dictionnaire somme, et k fois pour vérifier que chaque sac ne dépasse pas la capacité c . La complexité de cette fonction est donc en $O(n + k)$.

Question 2

2.1.

```
fonction generer_certificat(n, k):  
    certificat : dictionnaire  
  
    pour i allant de 1 à n:  
        certificat[i] = random(1, k)  
  
    retourner certificat
```

Cet algorithme génère les certificats de manière uniforme car chaque objet est associé à un sac de manière aléatoire. Chaque certificat a donc la même probabilité d'être généré.

La complexité de cet algorithme est en $O(n)$ car on passe n fois dans la boucle

2.2.

```
certificat = generer_certificat(n, k)  
verif_sac(certificat, n, poids, c, k)
```

cet algorithme génère un certificat aléatoire et vérifie s'il est valide. Il est non déterministe car il dépend de la génération aléatoire du certificat et polynomiale car la complexité de la génération du certificat est en $O(n)$ et la complexité de la vérification du certificat est en $O(n + k)$.

Question 3

3.1.

Pour n et k fixés, le nombre de certificats possibles est k^n . En effet, pour chaque objet, on a k choix de sacs possibles.

3.2.

Pour ordonner les certificats, on peut les trier par ordre lexicographique.

pour $k = 3$ et $n = 3$, les certificats dans l'ordre seraient :

```
{1:1, 2:1, 3:1}  
{1:1, 2:1, 3:2}  
{1:1, 2:1, 3:3}  
{1:1, 2:2, 3:1}  
{1:1, 2:2, 3:2}  
{1:1, 2:2, 3:3}  
{1:1, 2:3, 3:1}  
{1:1, 2:3, 3:2}  
{1:1, 2:3, 3:3}
```

```
{1:2, 2:1, 3:1}
...
```

3.3.

Pour tester si le problème a une solution ou non, on peut tester tous les certificats possibles dans l'ordre. Si un certificat est valide, alors le problème a une solution.

La complexité de cet algorithme est en $O(k^n * (n + k))$. En effet, on teste tous les certificats possibles, et pour chaque certificat, on vérifie s'il est valide en $O(n + k)$. Il y a k^n certificats possibles.

Question 4 - Implémentation

voir binPack.py

2. Réduction polynomiale

Question 1

1.

Réduction polynomiale de Partition vers Bin Pack :

a) Toutes instances de Partition (I) se réduit en une instance de BinPack (red(I)) par un algorithme polynomiale de cette façon :

```
BinPack      Partiton
n             <-  n
Xi            <-  Xi
c = sum(Xi) / 2
k = 2
```

Cet algo est polynomiale car il ne fait que des opérations en $O(1)$.

PROF

b.1) Montrer que si I valide \Rightarrow red(I) valide

Supposons que I valide alors il suffit de placer tout les objet appartenant au sous-ensemble qui correspond à la moitié de la somme (x_i tel que $i \in J$) dans un sac et le reste dans un autre sac (x_i tel que i n'appartient pas J).

b.2) Montrer que red(I) valide \Rightarrow I valide

Supposons que red(I) valide alors il suffit de prendre un des 2 sac tel que tous les objets de ce sac soit considérer comme J ($i \in J$ tel que $x_i \in k_1$)

1.1.

```
function reduction (nb_objet, liste_objet)
    capacite_sac = somme de liste_objet divisé par 2
```

```
nombre_sac = 2
return nb_objet, liste_objet, capacite_sac, nombre_sac
```

1.2.

On a déjà prouvé que binPack est un problème NP en montrant qu'il existe un algorithme polynomial pour vérifier si un certificat est valide.

On a aussi montré que Partition se réduit polynomialement à binPack en montrant que l'on peut transformer une instance de partition en une instance de binPack en temps polynomial.

Etant donné que Partition est NP-complet, il est également NP-dur, c'est à dire que tout problème NP se réduit polynomialement à partition. Par transitivité, tout problème NP se réduit polynomialement à binPack. binPack est donc NP-dur et NP. Il est donc NP-complet.

1.3

Nous ne pensons pas que BinPack se réduise polynomialement dans Partition car toutes instances de BinPack ne permettent pas d'avoir une instance de partition du au nombre de sac fixé à 2 et la capacité du sac fixé aussi. Uniquement certains cas de Binpack permet une instance de Partition.

Question 2

Partition peut être vu comme un cas particulier de Sum avec c égale à la moitié des poids de tous les objets. On peut en déduire que Partition peut se réduire polynomialement en une instance de Sum en fixant $c = \text{total} / 2$.

Question 3

Réduction polynomiale de Sum vers Partition :

a) Toutes instances de Sum (I) se réduit en une instance de Partition ($\text{red}(I)$) par un algorithme polynomiale de cette façon

```
Partition      Sum
n               <- n + 1
Xi             <- Xi + [2 * c - sum(Xi)]
```

Cet algo est polynomiale car il ne fait que des opérations en $O(1)$.

b.1) Montrer que si I valide $\Rightarrow \text{red}(I)$ valide

b.2) Montrer que $\text{red}(I)$ valide $\Rightarrow I$ valide

```
def sum_to_partition(nb_entiers ,cible ,list_entiers):

    el1 = 2*(somme de list_entiers) - cible
    el2 = somme de list_entiers + cible
    rep = ajouter à list_entier l'élément el1 et el2
```

```
return rep, nb_entiers + 2
```

Question 4

Grâce aux réductions faites aux questions 1 et 3, on peut réduire polynomialement Sum en BinPack par transitivité.

Dans un premier temps on réduit n'importe quelle instance de Sum en une instance de Partition, puis on réduit cette instance de Partition en une instance de BinPack.

Question 5

On doit ajouter des objets de la manière suivante :

On sélectionne le sac de capacité max et on construit un tableau en soustrayant la capacité max avec la capacité de tous les autres sacs et on ajoute cette liste d'objets aux autres.

Ensuite on définit la capacité ci la plus grande comme capacité maximale de la liste de l'instance de base et le nombre de sacs ne change pas.

Cette transformation permet d'ajouter des objets au sac n'ayant pas pour capacité la capacité maximale pour ainsi avoir que des ensembles d'objets ayant la capacité maximale.

3. Optimisation versus Décision

Question 1

Supposons que BinPackOpt1 soit de P alors il existerait un algorithme polynomial Aopt1 qui résoudrait le problème de BinPackOpt1.

On pourrait à partir de cet algorithme implémenter BinPack de cette façon :

```
BinPack(n , xi, c, k):  
    kmin = Aopt1(n, xi, c)  
    if kmin > k:  
        return false  
    else:  
        return true
```

Donc BinPack serait de classe P, car l'algorithme serait polynomial. BinPack est polynomial car il ne fait qu'appeler Aopt1 et faire une comparaison en $O(1)$.

Supposons que BinPackOpt2 soit de P alors il existerait un algorithme polynomial Aopt2 qui résoudrait le problème de BinPackOpt2.

On pourrait à partir de cet algorithme implémenter BinPack de cette façon :

```
BinPack(n , xi, c, k):  
    sac = Aopt2(n, xi, c) #liste d'objet correspondant à chaque sac
```

```

if sac.lenght > k:
    return false
else:
    return true

```

Donc BinPack serait de classe P, car l'algorithme serait polynomial.

Or BinPack est NP-complet et donc NP ce qui implique que $P = NP$

BinPack étant le problème de décision de BinPackOpt1 et 2. Leur complexité serait au moins aussi difficile que BinPack.

Question 2

Supposons que BinPack soit de P alors il existerait un algorithme polynomial Adec qui résoudrait le problème de BinPackOpt1.

On pourrait à partir de cet algorithme implémenté BinPackOpt1 de cette façon :

```

BinPackOpt1(n, xi c):
    for k de 1 à n:
        si Adec(n,xi,c,k) == true
            return k
    return -1 # valeur pour dire qu'il n'y a pas de mise en sac
    possible.

```

La complexité serait de $O(n \cdot Adec)$ avec Adec un algorithme polynomial, la complexité serait donc polynomiale donc BinPackOpt1 serait P.

Question 3

Supposons que BinPack soit de P alors il existerait un algorithme polynomial Adec qui résoudrait le problème de BinPackOpt2.

On pourrait à partir de cet algorithme implémenté BinPackOpt2 de cette façon :

```

BinPackOpt2(n, xi c):
    k = -1
    for i de n à 1:
        si Adec(n,xi,c,i) == true
            k = i
    reponse = refaire Adec(n,xi,c,k) mais prendre le tableau aff qui
    correspond à l'affectation d'un objet dans un sac.
    return reponse

```

La complexité serait de $O(n \cdot Adec + Adec)$ avec Adec un algorithme polynomial, la complexité serait donc polynomiale donc BinPackOpt2 serait P.