# 1 Algorithmes d'approximation

**Exercice 1**

On considère le problème SAC À DOS ILLIMITÉ :

* **Instance** : une liste de poids $p_1, \ldots, p_n$, une liste de valeurs $v_1, \ldots, v_n$ et un poids maximal $P$, ces valeurs étant des entiers naturels non nuls.
* **Solution** : une liste d'entiers naturels $a_1, \ldots, a_n$ telle que $\sum_{i=1}^{n} a_i p_i \leqslant P$
* **Optimisation** : Maximiser $\sum_{i=1}^{n} a_i v_i$.

Ce problème correspond à la version classique du sac à dos où on suppose que chaque objet est disponible en quantité illimitée. On considère l'algorithme suivant :

**Début algorithme**
  $S \leftarrow 0$.
  **Pour** $i \in \{1, \ldots, n\}$ par ordre décroissant de $\frac{v_i}{p_i}$ **Faire**
    Poser $a_i$ maximal tel que $S + a_i p_i \leqslant P$.
    $S \leftarrow S + a_i p_i$.
  **Renvoyer** $a_1, \ldots, a_n$.

1. Déterminer la complexité temporelle de cet algorithme.
2. Montrer que cet algorithme est une $\frac{1}{2}$-approximation de SAC À DOS ILLIMITÉ.

**Exercice 2**

On considère le problème COUVERTURE PAR SOMMETS :

* **Instance** : un graphe non orienté $G = (S, A)$.
* **Solution** : une couverture des arêtes par les sommets, c'est-à-dire un ensemble $R \subseteq S$ tel que $\forall a \in A, a \cap R \neq \emptyset$.
* **Optimisation** : Minimiser $|R|$.

On propose l'algorithme suivant :

**Début algorithme**
  $R \leftarrow \emptyset$.
  **Tant que** il reste des arêtes non couvertes **Faire**
    Choisir $a = \{s, t\} \in A$ une arête non couverte.
    $R \leftarrow R \cup \{s\}$.
  **Renvoyer** $R$.

1. Montrer que l'algorithme précédent n'est pas une approximation de COUVERTURE PAR SOMMETS à un facteur constant.

On considère l'algorithme suivant :

**Début algorithme**
  $C \leftarrow \emptyset$.
  **Tant que** $A \neq \emptyset$ **Faire**
    Choisir $a = \{s, t\} \in A$.
    $C \leftarrow C \cup \{a\}$.
    Supprimer de $A$ toutes les arêtes incidentes à $s$ ou $t$.
  **Renvoyer** $C$.

2. Montrer que l'ensemble renvoyé par l'algorithme précédent est un couplage maximal.
3. En déduire une 2-approximation de `COUVERTURE PAR SOMMETS`.

On représente un graphe en OCaml par le type usuel de tableau de listes d'adjacence :

```
type graphe = int list array;;
```

4. Écrire une fonction `couverture_sommets : graphe -> int list` qui prend en argument un graphe et renvoie une liste de sommets qui est une couverture des arêtes par les sommets de taille au plus le double de la couverture minimale. Cette fonction ne devra pas modifier le graphe donné en argument et aura une complexité linéaire en le nombre de sommets et d'arêtes.

**Exercice 3**

On considère le problème d'ordonnancement de tâches suivant : `ORDONNANCEMENT` :

* **Instance** : $n$ tâches ayant des temps de traitement $t_1, t_2, \ldots, t_n$ et $m$ machines.
* **Solution** : une attribution des tâches sur les machines, sous la forme d'une partition $P = \{E_1, \ldots, E_m\}$ de $[\![1, n]\!]$ de taille $m$.
* **Optimisation** : minimiser le temps total de traitement, c'est-à-dire $\max\limits_{j=1}^{m} \sum\limits_{i \in E_j} t_i$.

On décrit un algorithme glouton de manière informelle : en considérant les tâches par ordre décroissant de durée, affecter la tâche $i$ à la première machine disponible. On note $T$ la durée totale de traitement obtenue avec un tel algorithme et $T^*$ la durée d'un ordonnancement optimal.

1. Montrer que $T^* \geqslant \max\limits_{i=1}^{n} t_i$ et $T^* \geqslant \frac{1}{m} \sum\limits_{i=1}^{n} t_i$.
2. Montrer que $T \leqslant \max\limits_{i=1}^{n} t_i + \frac{1}{m} \sum\limits_{i=1}^{n} t_i$.
3. En déduire que l'algorithme précédent est une 2-approximation de `ORDONNANCEMENT`.

On peut faire une analyse plus fine de cet algorithme. En effet, l'algorithme est optimal si $n \leqslant m$. Supposons pour la suite que $n > m$. Sans perte de généralité, supposons de plus que $t_1 \geqslant t_2 \geqslant \ldots \geqslant t_n$.

4. Montrer que $T^* \geqslant 2t_{m+1}$.
5. En considérant la dernière tâche $i$ attribuée à la machine $j$ de temps maximal parmi celles qui effectuent au moins deux tâches dans l'algorithme glouton, montrer que $t_i \leqslant \frac{T^*}{2}$. En déduire que l'algorithme glouton est une $\frac{3}{2}$-approximation de `ORDONNANCEMENT`.

**Remarque**

On peut en fait montrer que c'est une $\frac{4}{3}$-approximation.

# 2 Branch and Bound

**Exercice 4**

La version du problème du sac à dos où chaque objet est limité est souvent appellé `SAC À DOS 0-1` :

* **Instance** : une liste de poids $p_1, \ldots, p_n$, une liste de valeurs $v_1, \ldots, v_n$ et un poids maximal $P$, ces valeurs étant des entiers naturels non nuls.

* **Solution** : une liste d'entiers naturels $(a_1, \ldots, a_n) \in \{0,1\}^n$ telle que $\sum_{i=1}^{n} a_i p_i \leqslant P$
* **Optimisation** : Maximiser $\sum_{i=1}^{n} a_i v_i$.

On a déjà vu que la version décisionnelle de ce problème était NP-complète. On considère une variante où les coefficients $a_i$ ne sont pas limités aux entiers 0 et 1, mais à toute valeur entre 0 et 1 : `SAC À DOS FRACTIONNAIRE` :

* **Instance** : une liste de poids $p_1, \ldots, p_n$, une liste de valeurs $v_1, \ldots, v_n$ et un poids maximal $P$, ces valeurs étant des entiers naturels non nuls.
* **Solution** : une liste de rationnels $(a_1, \ldots, a_n) \in [0,1]^n$ telle que $\sum_{i=1}^{n} a_i p_i \leqslant P$
* **Optimisation** : Maximiser $\sum_{i=1}^{n} a_i v_i$.

1. Montrer que `SAC À DOS FRACTIONNAIRE` $\in$ P en proposant un algorithme glouton qui le résout en complexité $\mathcal{O}(n \log n)$.

On cherche à résoudre le problème `SAC À DOS 0-1` par un algorithme de type Branch and Bound. Pour $i \in [\![0, n]\!]$, une solution partielle du problème sera un $i$-uplet $(a_1, a_2, \ldots, a_i) \in \{0,1\}^i$ (vide si $i = 0$).

2. Dans quel ordre semble-t-il pertinent de ranger les objets avant de lancer l'algorithme BnB résolvant ce problème ? Avec cet ordre, quelle est l'heuristique de branchement à considérer ?

On cherche à déterminer une heuristique d'évaluation pour une solution partielle. On propose d'utiliser le problème de `SAC À DOS FRACTIONNAIRE` pour cela. Pour $\tilde{y} = (a_1, \ldots, a_i)$ une solution partielle, on pose $h(\tilde{y})$ comme la valeur maximale d'une solution à ce problème pour l'instance $((p_j)_{j>i}, (v_j)_{j>i}, P_{\tilde{y}})$ où $P_{\tilde{y}} = P - \sum_{j=1}^{i} a_j p_j$.

3. Montrer que $h$ est admissible, c'est-à-dire que $h(\tilde{y})$ est toujours supérieur ou égale à la valeur totale d'une solution complétée à partir de $\tilde{y}$.

On suppose disposer de tableaux en C `int* p` et `int* v` de même taille `int n` contenant les poids et les valeurs des objets, déjà triés dans l'ordre de la question 2.

4. Écrire une fonction `double h(int* p, int* v, int* a, int n, int i, int P)` qui prend en argument les tableaux de poids $(p_j)_j$, de valeurs $(v_j)_j$ et des coefficients $(a_j)_j$, ainsi que la taille $n$ identique pour ces trois tableaux, un entier $i$ et un poids maximal $P$ et calcule la valeur $h(\tilde{y})$ où $\tilde{y} = (a_1, \ldots, a_i)$. On ne modifiera aucun des trois tableaux.

   *Attention : les indices dans un tableau commencent à 0 contrairement à ceux des n-uplets considérés dans l'exercice.*
5. En déduire une fonction `int* sac_a_dos_01(int* p, int* v, int n, int P)` qui applique l'algorithme BnB permettant de résoudre ce problème. La fonction renverra le tableau des $a_i$.

# 3 Algorithmes probabilistes

**Exercice 5**

Un flux de $n$ valeurs défilent successivement avec les défauts suivants :

– $n$ est trop grand pour qu'on puisse garder en mémoire toutes les valeurs déjà vues ;
– $n$ n'est pas connu à l'avance.

On peut imaginer qu'il s'agit d'une liste très grande qu'on ne peut parcourir qu'une seule fois.

Écrire un algorithme qui choisit $k$ éléments parmi les $n$, de manière équiprobable. La seule génération d'aléatoire dont on dispose est la sélection uniforme d'un entier compris entre 0 et $i - 1$, pour tout

entier $i$.

**Exercice 6: Quickselect**

On cherche à résoudre le problème suivant : étant donné un **tableau** $t = [t_0, t_1, \ldots, t_{n-1}]$ et un entier $k$, quel est le $k$-ème plus petit élément de $t$ ?

1. Décrire en français un algorithme de complexité temporelle dans le pire cas $\mathcal{O}(n \log n)$ et de complexité spatiale $\mathcal{O}(1)$ résolvant le problème (en s'autorisant à modifier le tableau).
2. En s'inspirant de l'algorithme de tri rapide randomisé, décrire un algorithme de type « diviser pour régner » permettant de résoudre le problème.
3. Déterminer sa complexité dans le pire des cas.
4. Écrire une fonction `int partition(int* t, int i, int j)` qui prend en argument un tableau `t` de taille $n$ et deux indices $0 \leqslant i < j \leqslant n$ et partitionne les éléments de `t` dont les indices sont compris dans $\{i, i+1, \ldots, j-1\}$. L'algorithme choisira un pivot aléatoirement parmi ces éléments et renverra l'indice où ce pivot ce trouve à la fin de l'exécution de l'algorithme. On rappelle que `rand()` renvoie un entier choisi aléatoirement et uniformément entre 0 et `RAND_MAX`.
5. En déduire une fonction `void selection_rapide(int* t, int n, int k)` qui renvoie le $k$-ème plus petit élément d'un tableau $t$ de taille $n$. La fonction pourra modifier le tableau.

Pour $0 \leqslant i < j < n$, on note $T_{ij} = \{t_i, t_{i+1}, \ldots, t_j\}$ et $X_{ij}$ la variable aléatoire qui vaut 1 si $t_i$ est comparé à $t_j$ pendant l'exécution de l'algorithme et 0 sinon. On note $X$ la variable aléatoire correspondant au nombre total de comparaisons effectuées par l'algorithme.

6. Montrer que $\mathbb{P}(X_{ij} = 1) = \frac{2}{\max(j,k) - \min(i,k) + 1}$.
7. En distinguant trois cas : $k \leqslant i$, $i < k < j$ et $j \leqslant k$, montrer que $\mathbb{E}(X) = \mathcal{O}(n)$.

**Exercice 7**

On considère un arbre correspondant à un graphe de jeu à deux joueurs où les coups alternent entre les deux joueurs. On suppose que les états terminaux sont gagnants pour l'un ou l'autre des joueurs (pas de match nul). On cherche à déterminer si la racine de l'arbre est gagnante pour 1 ou pour 2. On se limite au cas d'un arbre binaire où il reste $k$ coups à jouer pour chacun des deux joueurs, en commençant par le joueur 1.

1. Quelle est la complexité temporelle en fonction de $k$ d'un algorithme déterministe pour trouver qui est le joueur gagnant ?
2. Montrer qu'il n'est pas nécessaire de déterminer l'état de chaque feuille de l'arbre pour trouver qui est le joueur gagnant.
3. En déduire un algorithme Las Vegas simple permettant de déterminer le joueur gagnant.
4. Montrer que l'espérance du nombre de feuilles dont l'état est déterminé par l'algorithme précédent est inférieur ou égal à $3^k$.
5. En déduire que la complexité temporelle de cet algorithme est en $\mathcal{O}(n^{0.8})$, où $n$ est le nombre de nœuds.

**Exercice 8: Algorithme de Karger**

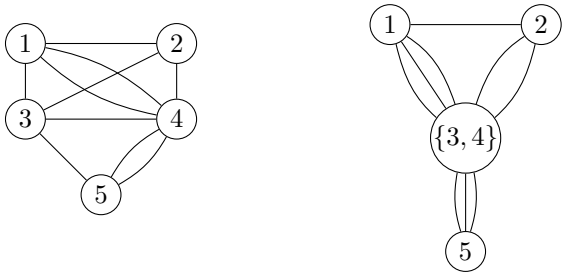
**Définition**

Un **multigraphe** est un couple $G = (S, A)$ tel que $A$ est un multi-ensemble d'éléments de $\mathcal{P}_2(S)$, c'est-à-dire un graphe où il peut y avoir plusieurs arêtes entre les mêmes sommets. Une **coupe** d'un multigraphe $G = (S, A)$ est une partition de $S$ en $S_1, S_2$. Une coupe $(S_1, S_2)$ est dite **minimale** si elle minimise le nombre d'arêtes entre $S_1$ et $S_2$.

Soit $G = (S, A)$ un multigraphe et $s \neq t$ deux sommets de $S$. La **contraction** de $\{s, t\}$ dans $G$, notée $G/\{s, t\}$, est le multigraphe formé à partir de $G$ où $s$ et $t$ ont été fusionnés, où les arêtes entre $s$ et $t$ ont disparu, et toute arête entre $s$ et un sommet $u$ devient une arête entre $\{s, t\}$ et $u$ (et de même pour une arête entre $t$ et $u$).

**Exemple**

Un multigraphe et sa contraction de $\{3, 4\}$ :




On cherche, étant donné un multigraphe connexe $G = (S, A)$, à trouver une coupe minimale par un algorithme randomisé.

1. On suppose qu'il existe $k$ coupes minimales. On tire aléatoirement et uniformément une coupe. Déterminer la probabilité qu'elle soit minimale en fonction de $n = |S|$ et $k$.

On propose l'algorithme suivant :

**Début algorithme**
  **Tant que** $|S| > 2$ **Faire**
    Choisir $\{s, t\} \in A$ aléatoirement et uniformément.
    $G \leftarrow G/\{s, t\}$.
  **Renvoyer** $S$.

2. En détaillant les structures de données choisies, déterminer la complexité temporelle de l'opération de contraction.
3. Montrer que l'algorithme précédent renvoie bien une coupe.
4. Déterminer la probabilité que l'algorithme renvoie une coupe minimale. On pourra se contenter d'une borne inférieure. Comment utiliser cet algorithme pour augmenter cette probabilité ?
5. Déterminer le nombre maximal de coupes minimales dans un multi-graphe d'ordre $n$.

**Exercice 9**

Dans cet exercice, on veut montrer que la borne $\frac{1}{3}$ de taux d'erreur dans la définition de BPP peut être choisie comme $\leqslant \frac{1}{2^{|x|^d}}$ pour une entrée $x$, où $d \in \mathbb{N}$ est un entier quelconque. On considère $A \in$ BPP et $\mathcal{A}$ un algorithme de Monte Carlo qui résout $A$ avec une probabilité d'erreur $p < \frac{1}{3}$. On considère l'algorithme suivant, où $\alpha$ est une constante qui sera définie plus tard :

**Entrée :** $x \in I_A$
**Début algorithme**
  $V \leftarrow 0$.
  $F \leftarrow 0$.
  **Pour** $i = 0$ à $i = \alpha|x|^d$ **Faire**
    **Si** $\mathcal{A}(x)$ **Alors**
      $V \leftarrow V + 1$.
    **Sinon**
      $F \leftarrow F + 1$.
  **Si** $V > F$ **Alors**
    **Renvoyer** Vrai.
  **Sinon**
    **Renvoyer** Faux.

On admet la propriété suivante :

**Proposition : Bornes de Chernoff**

Soit $p \in [\![0, 1]\!]$ et $X_1, X_2, \ldots, X_n$ des variables aléatoires indépendantes de Bernoulli de paramètre $p$. Soit $X = \sum_{i=1}^{n} X_i$. Alors :

$$\mathbb{P}(X \geqslant n/2) \leqslant \left(\frac{1+p}{\sqrt{2}}\right)^n$$

1. Montrer que la probabilité d'erreur de l'algorithme précédent est inférieure ou égale à $\left(\frac{4}{3\sqrt{2}}\right)^{\alpha|x|^d}$ pour une entrée $x$.
2. En déduire la valeur de $\alpha$ à choisir pour que la probabilité d'erreur soit strictement inférieure à $\frac{1}{2^{|x|^d}}$.

**Exercice 10**

Montrer que PP = Majority-P.