TD – Partie 2. Techniques algorithmiques

5. Diviser pour régner

Exercice 1. *Le maître*

1. Analyser la complexité des algorithmes suivants, dans lesquels `<op_elem>` désigne une opération élémentaire de complexité $O(1)$.

ALGO1(n) :
1. Si $n = 1$: Renvoyer 0
2. Pour i de 0 à $n-1$:
3. Pour j de 0 à $n-1$:
4. `<op elem>`
5. ALGO1($\lceil n/2 \rceil$)
6. ALGO1($\lceil n/2 \rceil$)

ALGO2(n) :
1. Si $n = 1$: Renvoyer 0
2. ALGO2($\lceil n/2 \rceil$)
3. `<op elem>`
4. ALGO2($\lceil n/2 \rceil$)
5. Pour i de 0 à $n-1$: `<op elem>`
6. ALGO2($\lceil n/2 \rceil$)

2. Pour résoudre un problème donné, on dispose de trois algorithmes. Calculer la complexité de chacun d'après sa description, et comparer les résultats. Sur une entrée de taille n,

i. ALGOA divise le problème en 5 sous-problèmes de taille $\lceil n/2 \rceil$ et combine les solutions en temps $O(n)$;
ii. ALGOB divise le problème en 2 sous-problèmes de taille $n-1$ et combine les solutions en temps $O(1)$;
iii. ALGOC divise le problème en 9 sous-problèmes de taille $\lceil n/3 \rceil$ et combine les solutions en temps $O(n^2)$.

3. Combien de fois la ligne « `Toujours pas fini...` » est-elle affichée par le programme suivant, en fonction de l'entrée `n` ?

```
void affiche(int n) {
  if (n > 1) {
    printf("Toujours pas fini...\n");
    affiche(n/2);
    affiche(n/2);
  }
}
```

Exercice 2. *Température extrême*

On suppose qu'au cours d'une journée, les températures commencent par croître (strictement) jusqu'à la température maximale, puis décroissent (strictement) en fin de journée. On dispose d'un tableau des températures mesurées (disons toutes les minutes), et on souhaite savoir quelle a été la température maximale.

1. Proposer un algorithme de complexité linéaire pour résoudre le problème.
2. En utilisant une stratégie « diviser pour régner », proposer un algorithme de meilleure complexité pour ce problème. *On pourra se servir de* $T_{[\lfloor n/2 \rfloor]}$.

Exercice 3. *Exponentiation rapide*

Étant donné x et n, on souhaite calculer (rapidement !) x^n. On s'intéresse à la complexité en nombre de multiplications effectuées.

1. Donner un algorithme de complexité $O(n)$ pour calculer x^n.

On cherche maintenant à décrire un algorithme plus rapide.

2. Exprimer x^n en fonction de $x^{\lfloor n/2 \rfloor}$, en distinguant le cas n pair du cas n impair.
3. En déduire un algorithme récursif de calcul de x^n.
4. Exprimer la complexité de l'algorithme obtenu.
5. Quel problème peut poser le fait d'analyser la complexité simplement en comptant le nombre de multiplications ?

Exercice 4. *Maximum et minimum*

On veut calculer le minimum et le maximum d'un tableau T d'entiers. On note $n = \#T$ la taille de T. *Dans cet exercice, on s'intéresse à la complexité en* ***nombre exact*** *de comparaisons, c'est-à-dire qu'on s'interdit les* $O(\cdot)$.

1. i. Écrire un algorithme qui calcule uniquement le maximum, et analyser le nombre exact de comparaisons effectuées.
   ii. Si on combine l'algorithme avec un autre qui calcule le minimum, combien de comparaisons sont effectuées ?
   iii. Ne peut-on pas économiser une comparaison ?
2. On calcule maintenant directement le minimum et le maximum, en adoptant une stratégie *diviser-pour-régner* : on sépare le tableaux en deux sous-tableaux de mêmes tailles. *On suppose que n est une puissance de* 2.
   i. Écrire l'algorithme.
   ii. Écrire l'équation de récurrence satisfaite par le nombre $C(n)$ de comparaisons effectuées.
   iii. Trouver la solution de l'équation de récurrence sous la forme $C(n) = \alpha n + \beta$ où α et β sont deux constantes à déterminer. *Remplacer* $C(n)$ *par* $\alpha n + \beta$ *dans la récurrence et en déduire des conditions sur* α *et* β.
   iv. *(bonus)* Que peut-on dire quand n n'est pas une puissance de 2 ?

Exercice 5. *Élément majoritaire*

On considère un tableau T de taille n contenant des entiers. On dit que l'élément p de T est *majoritaire dans T* s'il est contenu dans strictement plus de $n/2$ cases de T. Le but de l'exercice est d'écrire un algorithme qui retourne l'indice d'un élément majoritaire de T si celui-ci en contient un et -1 sinon.

1. On veut d'abord un algorithme simple pour résoudre le problème.

- **i.** Écrire un tel algorithme : pour chaque indice i de T ($0 \leq i \leq n-1$), compter le nombre d'éléments de T qui sont égaux à $T_{[i]}$, puis conclure.
- **ii.** Borner la complexité en temps de votre algorithme.

2. On veut maintenant élaborer un algorithme de type « diviser pour régner ».

- **i.** Montrer que T ne peut pas posséder d'élément majoritaire si ni $T_{[0,\lfloor n/2 \rfloor[}$ ni $T_{[\lfloor n/2 \rfloor,n[}$ n'en possèdent.
- **ii.** En déduire un algorithme récursif pour le problème.
- **iii.** Analyser la complexité de l'algorithme obtenu.

Exercice 6. *Transformée de Fourier rapide*

Soit $p \in \mathbb{C}[x]$ un polynôme de degré $< d$ et $\omega = e^{2i\pi/d}$. La *transformée de Fourier discrète* de p est le vecteur $\mathrm{DFT}_\omega^d(p) = (p(\omega^0), p(\omega^1), \ldots, p(\omega^{d-1})) \in \mathbb{C}^d$. On suppose que p est représenté par le vecteur de ses d coefficients, et les opérations élémentaires sont les opérations sur les complexes.

1.
- **i.** Décrire un algorithme qui étant donné p et $\alpha \in \mathbb{C}$ calcule $p(\alpha)$. Analyser sa complexité.
- **ii.** En déduire un algorithme qui étant donné p et $\omega = e^{2i\pi/d}$ calcule $\mathrm{DFT}_\omega^d(p)$. Analyser sa complexité.

On décrit maintenant une stratégie « diviser pour régner » pour calculer $\mathrm{DFT}_\omega^d(p)$ plus rapidement. *On suppose que d est une puissance de 2.*

2. Si $p = \sum_{i=0}^{d-1} p_i x^i$, on définit $p^{(0)} = \sum_{i=0}^{d/2} p_{2i} x^i$ et $p^{(1)} = \sum_{i=0}^{d/2} p_{2i+1} x^i$.

- **i.** Exprimer p à l'aide de $p^{(0)}$ et $p^{(1)}$. Combien coûte le calcul de $p^{(0)}$ et $p^{(1)}$ à partir de p ?
- **ii.** Exprimer les coefficients de $\mathrm{DFT}_\omega^d(p)$ en fonction de ceux de $\mathrm{DFT}_{\omega^2}^{d/2}(p^{(0)})$ et $\mathrm{DFT}_{\omega^2}^{d/2}(p^{(1)})$. *Indication. Justifier et utiliser le fait que $\omega^{2k} = \omega^{2k-d}$ pour $k \geq d/2$.*
- **iii.** En déduire un algorithme de type « diviser pour régner » pour calculer $\mathrm{DFT}_\omega^d(p)$ et analyser sa complexité.

6. Recherche exhaustive

Exercice 7. *Parcours d'objets*

Les algorithmes de recherche exhaustive nécessitent de parcourir différents types d'objets. Le but de cet exercice est d'écrire quelques uns de ces parcours.

1. On s'intéresse aux n-uplets d'entiers *strictement décroissants* entre 0 et $k-1$.

- **i.** Montrer qu'il faut forcément que $n \leq k$.
- **ii.** Si on prend l'ordre d'énumération des n-uplets vu en cours, quel est le premier qui soit strictement décroissant ? et le dernier ?
- **iii.** Soit U un n-uplet strictement décroissant d'entiers entre 0 et $k-1$. Quelle est la valeur maximale de $U_{[i]}$ (en fonction de i) ?
- **iv.** Écrire un algorithme pour passer d'un n-uplet strictement décroissant au suivant.

2. On s'intéresse aux sous-ensembles à n éléments de l'ensemble $\{0, \ldots, k-1\}$.

- **i.** Montrer qu'on peut représenter un tel sous-ensemble, de manière unique, par un n-uplet strictement décroissant d'entiers entre 0 et $k-1$.
- **ii.** En déduire le nombre de n-uplets strictement décroissants d'entiers entre 0 et $k-1$.

3.
- **i.** Comment parcourir les sous-ensembles à t éléments d'un ensemble fini ?
- **ii.** Comment parcourir tous les sous-ensembles à au plus t éléments d'un ensemble fini ?
- **iii.** Comment parcourir l'ensemble des mots binaires de longueur s qui comportent exactement t bits à 1 ?

Exercice 8. *Somme et sac-à-dos*

Étant donné un tableau d'entiers T et une cible c, on cherche à déterminer s'il existe un sous-tableau de T dont la somme des éléments vaut c exactement. *Par exemple, si* $T = [-5, 2, 7, -3, 2, -6]$ *et* $c = 5$*, alors il suffit de prendre* $[2, 7, 2, -6]$ *dont la somme bien* 5*. Par contre si* $c = 10$ *ou* $c = -13$*, il n'y a aucune solution.*

1. Décrire une façon de représenter et parcourir tous les sous-tableaux de T.
2. Écrire un algorithme pour résoudre le problème et analyser sa complexité.

Le *problème du sac-à-dos* est le suivant : étant donné n objets $(t_0, v_0), \ldots, (t_{n-1}, v_{n-1})$ et une taille S, on cherche un sous-ensemble $I \subset \{0, \ldots, n-1\}$ qui maximise la *valeur* totale $\sum_{i \in I} v_i$ tout en respectant la contrainte $\sum_{i \in I} t_i \leq S$.

3. Adapter l'algorithme précédent à ce problème et analyser sa complexité.

Exercice 9. *Huit reines*

Le *problème des huit reines* demande s'il est possible de placer 8 reines sur un échiquier 8×8, sans qu'elles se menacent l'une l'autre : deux reines se menacent

si elle sont sur la même ligne, la même colonne, ou la même diagonale (ou anti-diagonale). C'est en effet possible, et il y a même plusieurs solutions. Ce qui nous intéresse ici est de calculer toutes les solutions possibles. De plus, on généralise le problème au cas des *n reines* : on cherche à placer n reines sur un échiquier $n \times n$, avec toujours les mêmes conditions.

1. **i.** Montrer que le problème n'admet aucune solution pour $n = 2$ et $n = 3$.
ii. Pourquoi ne peut-on pas mettre strictement plus de n reines sur un échiquier $n \times n$?
iii. Démontrer que dans toute solution, chaque ligne de l'échiquier contient exactement une reine.

On représente une solution par un tableau Q de taille n, tel que $Q_{[i]} = j$ s'il y a une reine en case (i, j) de l'échiquier.

2. **i.** Quelle propriété de Q est impliquée par le fait deux reines ne peuvent pas appartenir à la même colonne ?
ii. Écrire une fonction qui teste si Q est une solution valide et analyser sa complexité.
iii. En déduire un algorithme pour le problème et analyser sa complexité.

Exercice 10. *Codes de Gray*

Un code de Gray est une énumération de tous les mots binaires de longueur n dans un ordre particulier, afin que pour passer d'un mot au suivant, seul un bit ait besoin d'être modifié. Par exemple, l'énumération suivante est un code de Gray sur 2 bits : 00, 01, 11, 10. Les codes de Gray permettent une énumération très efficace des mots binaires, et donc des sous-ensembles d'un ensemble donné.

1. Écrire un code de Gray sur 3 bits.
2. Quelle est la longueur d'un code de Gray ?
3. Soit T un tableau contenant un code de Gray sur n bits. On définit les tableaux T^0 et T^1 par $T^0_{[i]} = 0 \cdot T_{[i]}$ et $T^1_{[i]} = 1 \cdot T_{[i]}$: on concatène 0 ou 1 en tête de $T_{[i]}$.
i. Montrer que tous les mots binaires sur $n + 1$ bits sont dans $T^0 \cup T^1$.
ii. Montrer comment construire *facilement* un tableau T^+ contenant un code de Gray sur $n + 1$ bits à partir des tableaux T^0 et T^1.
iii. En déduire un algorithme de construction d'un code de Gray sur n bits et analyser sa complexité.

7. Algorithmes probabilistes

Exercice 11. *Échantillonnage par rejet*

Pour tirer un point aléatoire dans un disque $\mathcal{D}$ de centre $(0, 0)$ et de rayon 1, on tire $x, y \in [-1, 1]$ uniformément : si $(x, y) \in \mathcal{D}$, on le renvoie ; sinon on recommence.

1. **i.** Comment tester si $(x, y) \in \mathcal{D}$?
   ii. Quelle est la probabilité que $(x, y) \in \mathcal{D}$?
   iii. En déduire l'espérance du nombre de tirages nécessaires pour obtenir un point dans le disque.
2. Afin de tirer moins de réels, une personne astucieuse commence par tirer x, puis tire des y tant que $(x, y) \notin \mathcal{D}$. Est-ce une bonne idée ?

On généralise l'algorithme, pour des ensembles finis : soit X un ensemble fini dans lequel on sait tirer un élément uniformément, et $S \subset X$; supposons qu'on sache tester, pour $x \in X$, si $x \in S$; enfin, on note $p = \#S/\#X$.

3. **i.** À quoi correspondent X, S et p dans le cas du disque ?
   ii. Décrire l'algorithme permettant de tirer un élément dans S.
   iii. Quelle est l'espérance du nombre d'éléments de X à tirer avant d'avoir un élément de S ?
   iv. Justifier que l'algorithme renvoie un point *uniforme* dans S, c'est-à-dire que chaque point de S est renvoyé avec probabilité $1/\#S$.
4. On souhaite tirer un nombre premier uniformément parmi les premiers $\leq N$. On suppose disposer d'un algorithme déterministe de test de primalité. *On rappelle que le nombre de premiers $\leq N$ est $\pi(N) \geq N/\ln N$ (pour $N \geq 17$).* Calculer l'espérance du nombre d'entiers à tirer avant d'en obtenir un premier.

Exercice 12. *Preuve de l'algorithme de Freivalds*

L'algorithme de Freivalds prend en entrée $A, B, C \in \mathbb{K}^{n\times n}$ pour un certain corps $\mathbb{K}$. Il tire un vecteur $\vec{v}$ aléatoire uniforme dans $\{0,1\}^n$ et répond VRAI si $A\times B\times\vec{v} = C\times\vec{v}$, FAUX sinon.

1. Justifier que la complexité de l'algorithme est $O(n^2)$.
2. Montrer que si $C = A \times B$, alors l'algorithme répond toujours VRAI.
3. On suppose que $C \neq A \times B$ et on note $D = C - A \times B$.
   i. Justifier qu'il existe (i, j) tel que $D_{i,j} \neq 0$.
   ii. Montrer que si $A \times B \times \vec{v} = C \times \vec{v}$, alors $v_j = -\frac{1}{D_{i,j}} \sum_{k\neq j} D_{i,k} v_k$
   iii. En déduire que $\Pr[A \times B \times \vec{v} = C \times \vec{v}] \leq \frac{1}{2}$. Que conclut-on ?
4. Quel type d'algorithme est l'algorithme de Freivalds ?

Exercice 13. QUICKSELECT

L'algorithme QUICKSELECT est une variante du tri rapide, pour résoudre le problème suivant : étant donné un tableau T de taille n et un entier $k \in \{1, \ldots, n\}$, on souhaite trouver le $k^{\text{ème}}$ plus petit élément de T, noté $T^{(k)}$. *Par exemple, si $k = 1$, $T^{(1)}$ est le minimum de T.* Le principe de l'algorithme est le même que pour le tri rapide : choisir aléatoirement un pivot p dans T ; calculer T_{INF} et T_{SUP} qui contiennent les éléments de T *strictement* inférieurs et *strictement* supérieurs à

p, respectivement ; si nécessaire effectuer un appel récursif sur T_{INF} ou T_{SUP}. *On suppose tous les éléments distincts dans T.*

1. Soit n_{INF} et n_{SUP} les tailles de T_{INF} et T_{SUP}.
   - **i.** Si $n_{\text{INF}} = k-1$, que vaut $T^{(k)}$?
   - **ii.** Si $n_{\text{INF}} \geq k$, exprimer $T^{(k)}$ comme $T^{(\ell)}_{\text{INF}}$ pour un certain ℓ.
   - **iii.** Si $n_{\text{INF}} < k-1$, exprimer $T^{(k)}$ comme $T^{(\ell)}_{\text{SUP}}$ pour un certain ℓ.
2. Écrire complètement l'algorithme. *Il n'est pas interdit d'être délicat et d'éviter de construire des tableaux inutiles.*
3. Combien de comparaisons sont effectuées, si tous les choix probabilistes sont mauvais ?
4. On s'intéresse maintenant l'espérance du nombre C_n de comparaisons effectuées pour un tableau de taille n.
   - **i.** Montrer que $\mathbb{E}[C_n] = \sum_{j=1}^{n} \mathbb{E}[C_n | p = T^{(j)}] \Pr[p = T^{(j)}]$, où p est le pivot.
   - **ii.** Montrer que si $p = T^{(j)}$, l'algorithme effectue un appel récursif sur un tableau de taille $t \leq \max(j-1, n-j)$.
   - **iii.** En déduire que $\mathbb{E}[C_n | p = T^{(j)}] \leq n + E_m$ où $m = \max(j-1, n-j)$.
   - **iv.** Montrer par récurrence sur n que $\mathbb{E}[C_n] \leq 4n$.

Exercice 14. *Compteur probabiliste*

Un compteur entier qui va de 0 à n utilise un espace mémoire $O(\log n)$. On va voir une technique probabiliste qui permet de fournir un compteur approximatif qui utilise un espace exponentiellement plus petit. L'idée est d'avoir une fonction INCRÉMENT qui soit probabiliste.

1. On définit une fonction INCRÉMENT(c) qui incrémente c avec probabilité p et qui ne fait rien avec probabilité $1-p$. On initialise c à 0 et on effectue n appels à INCRÉMENT.
   - **i.** Quelle est l'espérance de la valeur de c ?
   - **ii.** Quelle valeur faut-il renvoyer pour avoir une valeur approchée de n ?
   - **iii.** Borner la probabilité que la valeur renvoyée soit $\geq 2n$.

On modifie la fonction INCRÉMENT, pour que INCRÉMENT(c) incrémente c avec probabilité $1/2^c$ et ne fasse rien avec probabilité $1-1/2^c$.

2. - **i.** On suppose avoir accès à une fonction BITALÉATOIRE() qui renvoie 0 avec probabilité $\frac{1}{2}$ et 1 avec probabilité $\frac{1}{2}$. Écrire explicitement la fonction INCRÉMENT.
   - **ii.** Quelle est l'espérance du nombre d'appels effectués à BITALÉATOIRE ?

On note C_k la variable aléatoire qui décrit la valeur du compteur après k appels à INCRÉMENT, et $p_{k,c} = \Pr[C_k = c]$ la probabilité que le compteur ait la valeur c après k appels à INCRÉMENT.

3. **i.** Calculer $p_{0,0}$, $p_{0,c}$ pour $c > 0$ et $p_{k,0}$ pour $k > 0$.
ii. Calculer $\sum_{c \geq 0} p_{k,c}$.
iii. Montrer que pour $k, c > 0$, $p_{k,c} = \frac{1}{2^{c-1}} p_{k-1,c-1} + (1 - \frac{1}{2^c}) p_{k-1,c}$. *Indication. Si le compteur vaut c après k appels à* INCRÉMENT*, quelle peut être sa valeur après $k-1$ appels ?*

Finalement, on décide de renvoyer la valeur $v = 2^c$. On cherche à calculer l'espérance de la valeur finale de v après n appels à INCRÉMENT. On note $V_k = 2^{C_k}$ la variable aléatoire qui décrit la valeur v renvoyée par l'algorithme.

4. **i.** Exprimer $\mathbb{E}[V_k]$ en fonction de $p_{k,c}$, pour k, $c \geq 0$.
ii. En déduire que $\mathbb{E}[V_k] = \mathbb{E}[V_{k-1}] + 1$.
iii. En déduire que $\mathbb{E}[V_n] = n$.

8. Programmation dynamique

Exercice 15. *Coefficients binomiaux*

Les coefficients binomiaux $\binom{n}{k}$ vérifient la récurrence suivante : $\binom{n}{0} = \binom{n}{n} = 1$ et $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ pour $0 < k < n$.

1. Écrire un algorithme de programmation dynamique pour calculer $\binom{n}{k}$ et analyser sa complexité.
2. Écrire une variante de l'algorithme qui minimise l'espace mémoire utilisé.

Exercice 16. *Stratégie étudiante*

Une étudiante prévoit son programme de révision. Chaque jour peut être *tranquille* ou *soutenu*, ou un jour de *repos*. Mais avant un jour *soutenu*, il lui faut un jour de *repos*. Si le jour i est *tranquille*, cela lui fera gagner t_i points sur sa moyenne, s'il est *soutenu* ce sera s_i points, et aucun si c'est un jour de *repos*.*Par exemple, sur la semaine suivante, l'optimal est d'être en repos les jours 1 et 4, en travail tranquille le jour 3, et en travail soutenu les jours 2 et 5, ce qui lui rapportera* $2,5$ *points.*

Jour	1	2	3	4	5
t	0,3	0,5	0,4	0,6	0,2
s	0,5	1,2	1	0,8	0,9

1. Montrer que l'algorithme suivant n'est pas optimal (n est le nombre de jours) :

```
1 i ← 1
2 Tant que i ≤ n :
3     Si i ≤ n − 1 et s_{i+1} > t_i + t_{i+1} :
4         Repos le jour i et travail soutenu le jour i + 1
5         i ← i + 2
6     Sinon :
7         Travail tranquille le jour i
8         i ← i + 1
```

2. Proposer une formule récursive pour calculer le nombre p_i maximal de points obtenu en travaillant jusqu'au jour i.
3. Décrire un algorithme de programmation dynamique pour calculer p_n et analyser sa complexité en temps et en espace.
4. Peut-on réduire sa complexité en espace ?
5. Décrire un algorithme de calcul d'une stratégie optimale.

Exercice 17. *Le retour du sac-à-dos*

On rappelle le *problème du sac-à-dos* : étant donné n objets (t_0, v_0), …, (t_{n-1}, v_{n-1}) et une taille S, on cherche un sous-ensemble $I \subset \{0, \dots, n-1\}$ qui maximise la *valeur* totale $\sum_{i \in I} v_i$ tout en respectant la contrainte $\sum_{i \in I} t_i \leq S$. On note $V_{\max}$ la valeur maximale qu'on peut atteindre.

1. Pour $m < n$ et $t \leq S$, on note $V(m,t)$ la valeur maximale que l'on peut atteindre en ne prenant que des objets parmi (t_0, v_0), …, (t_m, v_m), et avec une taille totale maximale $\leq t$.
   - **i.** Exprimer $V_{\max}$ avec $V(m,t)$ pour un m et un t bien choisis.
   - **ii.** Que vaut $V(m,t)$ si $t < t_m$? Et si $m = 0$?
   - **iii.** Donner une formule récursive pour $V(m,t)$. *Distinguer deux cas : on choisit l'objet m ou non.*
   - **iv.** En déduire un algorithme de calcul de $V_{\max}$ et analyser sa complexité.
   - **v.** Comparer le résultat avec l'approche par recherche exhaustive.
2. On souhaite effectuer un algorithme de reconstruction, pour obtenir une solution.
   - **i.** Modifier l'algorithme précédent pour calculer, pour tout m et t, un booléen $C_{[m,t]}$ qui indique si l'objet m est choisi pour atteindre la valeur $V(m,t)$.
   - **ii.** Utiliser le tableau des $C_{[m,t]}$ pour reconstruire la solution.
3. *(bonus)* Décrire une variante des algorithmes précédents qui calcule la taille $S(m,v)$ minimale d'un sac-à-dos de valeur v constitué des objets 0 à m, pour tout m et v.

Exercice 18. *Sous-vecteur de somme maximale*

Soit T un tableau de n entiers relatifs. On cherche un sous-tableau $T_{[i,j[}$ dont la somme des éléments est maximale. Formellement, on veut calculer $M = \max\{\sum_{k=i}^{j-1} T_{[k]} : 0 \leq i \leq j \leq n\}$. Par exemple, pour

$T =$	2	18	−22	20	8	−6	10	−24	13	3

la solution optimale est $T_{[3,7[}$ dont la somme vaut 32. *Remarque : pour* $T =$ | *-17* |, *la solution optimale est* $T_{[0,0[}$ *dont la somme vaut* 0.

1. Soit $\Sigma_{[i,j[} = \sum_{k=i}^{j-1} T_{[k]}$, et $M_j = \max\{\Sigma_{[i,j[} : 0 \leq i < j\}$.
   i. Que vaut M_1 ?
   ii. Exprimer M_j en fonction de M_{j-1} et $T_{[j-1]}$.
   iii. Exprimer M en fonction des M_j.
2. En déduire un algorithme de programmation dynamique qui calcule M. *L'algorithme doit utiliser le moins d'espace possible.*
3. Modifier l'algorithme précédent pour qu'il renvoie également les indices i et j du sous-tableau de somme maximale.

9. Algorithmes d'approximation

Exercice 19. *Partition*

Notation. *Pour un ensemble S d'entiers, on note $\Sigma_S = \sum_{s\in S} s$.*

Étant donné un ensemble de n entiers positifs $A = \{a_0, \ldots, a_{n-1}\}$, on cherche une partition $A = X \sqcup Y$ équilibrée, c'est-à-dire telle que $\Sigma_X \simeq \Sigma_Y$. Plus précisément, on cherche à minimiser $\max(\Sigma_X, \Sigma_Y)$. On propose l'algorithme glouton suivant.

PARTITION(A) :

```
1 (X, Y, Σ_X, Σ_Y) ← (∅, ∅, 0, 0)
2 Pour i = 0 à n − 1 :
3   Si Σ_X < Σ_Y : X ← X ∪ {a_i}, Σ_X ← Σ_X + a_i
4   Sinon : Y ← Y ∪ {a_i}, Σ_Y ← Σ_Y + a_i
5 Renvoyer (X, Y)
```

1. i. PARTITION fournit-il une solution optimale sur l'entrée $A = \{4, 2, 3, 2, 7\}$?
   ii. Quelle est sa complexité ?

Pour une entrée A, soit $(X^\star, Y^\star)$ une solution optimale et $\text{OPT} = \max(\Sigma_{X^\star}, \Sigma_{Y^\star})$. Soit (X, Y) la solution renvoyée par PARTITIONGLOUTON, et on suppose sans perte de généralité $\Sigma_X \geq \Sigma_Y$.

2. i. Montrer que pour tout i, $\text{OPT} \geq a_i$.
   ii. Montrer que $\text{OPT} \geq \frac{1}{2}\Sigma_A$.
3. On considère le dernier élément a_k ajouté par PARTITION à X.
   i. Montrer que $\Sigma_X - a_k \leq \frac{1}{2}(\Sigma_A - a_k) \leq \text{OPT} - \frac{1}{2}a_k$.
   ii. En déduire que PARTITION est une $\frac{3}{2}$-approximation pour le problème.
   iii. Construire un exemple pour lequel PARTITION fournit une solution égale exactement à $\frac{3}{2}\text{OPT}$.
4. *(bonus)* On modifie très légèrement PARTITION en triant les a_i en ordre décroissant : $a_0 \geq a_1 \geq \cdots \geq a_{n-1}$. On garde les mêmes notations que précédemment.
   i. Montrer que sur l'entrée $\{10, 10, 9, 9, 2\}$, l'algorithme n'est pas optimal.
   ii. Montrer que si $\#X = 1$, alors la solution renvoyée est optimale.

iii. On suppose que $\#X \geq 2$. Montrer que le dernier élément a_k ajouté par PARTITION à X vérifie $a_k \leq \frac{2}{3}\text{OPT}$.

iv. En déduire que PARTITION avec tri est une $\frac{4}{3}$-approximation.

v. Construire un exemple pour lequel PARTITION avec tri fournir une solution exactement égale à $\frac{7}{6}\text{OPT}$. [1]

Exercice 20. *Coupe maximale*

Soit $G = (S, A)$ un graphe. Une *coupe* de G est une partition des sommets $S = X \sqcup Y$ en deux sous-ensembles disjoints non vides. La *taille* d'une coupe $X \sqcup Y$ est le nombre d'arêtes dont une extrémité est dans X et l'autre dans Y. On cherche à calculer une coupe $X \sqcup Y$ de taille maximale.

On utilise l'algorithme probabiliste simpliste suivant : chaque sommet $s \in S$ est affecté indépendamment à X avec probabilité $\frac{1}{2}$ et à Y avec la même probabilité.

1. Soit $a = \{u, v\}$ une arête de G. Calculer la probabilité que a soit *coupée*, c'est-à-dire qu'une de ses extrémités appartienne à X et l'autre à Y.

2. Quelle est l'espérance de la taille de la coupe renvoyée par l'algorithme probabiliste ? *Utiliser la linéarité de l'espérance.*

3. En déduire que l'espérance de la taille de la coupe renvoyée est $\geq \frac{1}{2}\text{OPT}$ où OPT est la taille d'une coupe maximale.

Exercice 21. MAXSAT

On considère des *formules sous forme normale conjonctive* (formule CNF), comme par exemple $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_3) \wedge (x_2 \vee x_3)$: c'est une *conjonction de clauses*, chaque clause est une *disjonction de littéraux* et chaque littéral est soit une variable booléenne soit sa négation. Le problème SAT consiste, étant donné une formule CNF, à décider s'il existe une affectation qui satisfait la formule.

Dans cet exercice, on s'intéresse à une variante du problème : MAXSAT. Étant donné la formule CNF, il s'agit de trouver l'affectation qui satisfait *le plus possible de clauses*.

1. Justifier que si on sait résoudre de manière exacte MAXSAT, alors on peut résoudre SAT.

2. Vérifier que dans la formule de l'exemple, au plus 3 clauses sur 4 peuvent être satisfaites simultanément.

3. Quel algorithme (déjà vu) peut-on modifier pour résoudre MAXSAT ? Quelle est sa complexité ?

4. On propose l'algorithme suivant : on renvoie une affectation des variables choisie uniformément, c'est-à-dire qu'on choisit, pour chaque variable, la valeur VRAI avec probabilité $\frac{1}{2}$ ou FAUX avec probabilité $\frac{1}{2}$. *On suppose que chaque variable apparaît au plus une fois dans chaque clause.*

1. *Remarque. On peut en fait montrer (mais c'est difficile) que l'algorithme glouton avec tri renvoie toujours une solution* $\leq \frac{7}{6}\text{OPT}$.

i. Quelle est la complexité de cet algorithme ?
ii. Soit C une clause de taille k. Montrer que la probabilité que C soit satisfaite est $1 - 1/2^k$.
iii. En déduire que l'espérance du nombre de clauses satisfaites est $\geq m/2$ où m est le nombre total de clauses.
iv. Que peut-on dire sur le facteur d'approximation ?

10. Recherche exhaustive rapide

Exercice 22. 3-SAT

Soit φ une formule 3-CNF et ℓ un littéral apparaissant dans φ.

1. Montrer que si $\neg\ell$ apparaît dans φ, alors il existe une clause de taille ≤ 2 dans $\varphi_{|\ell}$.
2. En déduire qu'à part pour la formule d'origine, on peut supposer dans les appels récursifs que φ a une clause de taille ≤ 2.
3. En déduire un algorithme et analyser sa complexité.

Exercice 23. *Indépendance*

Soit $G = (S, A)$ un graphe. Un *indépendant* de G est un sous-ensemble de sommets $I \subset S$ tel qu'il n'existe aucune arête entre eux : $\forall s, t \in I,\ \{s, t\} \notin A$. On cherche, dans un graphe donné, la taille du plus grand indépendant possible.

1. Quelle est la complexité d'un algorithme de recherche exhaustive pour ce problème ?

On veut exprimer l'algorithme de recherche exhaustive de manière récursive. Pour un sommet s, on note $V(s)$ l'ensemble de ses *voisins*, c'est-à-dire l'ensemble des sommets t qui ont une arête reliée à s : $V(s) = \{t \in S : \{s, t\} \in A\}$. L'algorithme récursif est alors le suivant : si G n'a aucun sommet, renvoyer 0 ; sinon soit $s \in S$ (quelconque) : soit l'indépendant maximal contient s et aucun de ses voisins (appel récursif sur $G \setminus \{s\} \cup V(s)$) soit il ne contient pas s (appel récursif sur $G \setminus \{s\}$).

2. **i.** Écrire l'algorithme.
   ii. Analyser sa complexité.
3. **i.** Que peut-on faire des sommets *isolés*, c'est-à-dire sans voisin ?
   ii. En déduire une variante de l'algorithme et analyser sa complexité.
4. Améliorer l'algorithme en examinant le cas des sommets ayant un seul voisin.