Complexité algorithmique

Mesure du temps d'exécution d'un algorithme

Il n'est pas possible de déterminer à l'avance le temps que prendra un ordinateur pour exécuter un algorithme : cette caractéristique dépend de trop nombreux paramètres, tant matériels que logiciels. En revanche, il est souvent possible d'évaluer l'*ordre de grandeur* du temps d'exécution en fonction des paramètres de l'algorithme. Par exemple, si n désigne la taille du paramètre d'entrée, un algorithme sera qualifié de *linéaire* lorsque le temps d'exécution croîtra proportionnellement à n et de *quadratique* lorsque le temps d'exécution croîtra proportionnellement à n^2. Durant cette séance de travaux pratiques nous allons écrire plusieurs algorithmes résolvant le même problème : le premier aura un temps d'exécution proportionnel à n^3, le second à n^2 et le troisième à n et nous constaterons la différence considérable qui peut exister concernant le temps d'exécution de chacune de ces trois fonctions sur des données de grandes tailles.

Pour mesurer le temps d'exécution, nous allons commencer par importer une fonction nommée `time` qui appartient à un module nommé lui aussi `time`. Votre code devra donc commencer par la ligne suivante :

```
from time import time
```

Une fois cette commande interprétée, vous disposerez d'une fonction `time()` vous donnant la durée exprimée en secondes depuis une date de référence qui dépend de votre système. Pour mesurer la durée d'exécution d'une portion de code, il vous suffira d'encadrer celle-ci de la façon suivante :

```
debut = time()
.......................
.... portion de code ....
.......................
fin = time()

print("durée d'exécution :", fin - debut)
```

Coupes minimales

Dans ce problème, on considère des listes d'entiers relatifs $a = [a_0, \dots, a_{n-1}]$, et on appelle *coupe* de a toute suite non vide d'éléments consécutifs de cette liste. Ainsi, une coupe est une liste de la forme $[a_i, \dots, a_{j-1}]$ avec $0 \leqslant i < j \leqslant n$ qu'on notera désormais $a[i : j]$.

À toute coupe $a[i : j]$ on associe la somme $s[i : j] = \sum_{k=i}^{j-1} a_k$ des éléments qui la composent. Le but de ce problème est de déterminer un algorithme efficace pour déterminer la valeur minimale des sommes des coupes de a.
À titre d'exemple, la somme minimale des coupes du tableau $a = [4, -4, 1, -1, -9, 8, -3, 8, -5, 5]$ est égale à -13, valeur atteinte pour la coupe $a[1 : 5]$.

Un générateur pseudo-aléatoire

On considère la suite $(u_n)_{n\in\mathbb{N}}$ définie par la donnée de $u_0 = 42$ et la relation de récurrence :

$$\forall n \in \mathbb{N}, \qquad u_{n+1} = \left(163\,811 \times u_n \mod 655\,211\right) - 327\,605$$

Pour expérimenter les différentes fonctions que nous allons écrire, nous allons avoir besoin de trois listes Python, de longueurs respectives 1 000, 10 000, 100 000, qu'on nommera `lst1`, `lst2` et `lst3`. Rédiger un script générant chacune de ces trois listes, avec pour contenu :

$$\texttt{lst1} = \left[u_i \,\middle|\, 1 \leqslant i \leqslant 1\,000\right] \quad \texttt{lst2} = \left[u_i \,\middle|\, 1\,001 \leqslant i \leqslant 11\,000\right] \quad \texttt{lst3} = \left[u_i \,\middle|\, 11\,001 \leqslant i \leqslant 111\,000\right]$$

Vérifiez que vous ne vous êtes pas trompés ; vous devez avoir

$$\texttt{lst1[500]} = 54\,009, \qquad \texttt{lst2[500]} = 10\,995, \qquad \texttt{lst3[500]} = -87\,244.$$

Par la suite, vous pourrez vérifier la validité de vos algorithme ; les sommes minimales des coupes de chacune de ces trois listes valent respectivement :

$$-9\,557\,159 \text{ pour } \texttt{lst1} \qquad -21\,577\,847 \text{ pour } \texttt{lst2} \qquad -42\,168\,128 \text{ pour } \texttt{lst3}$$

Question 1. un algorithme naïf

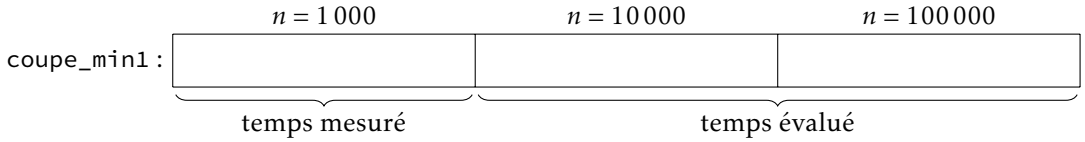
a) Définir une fonction `somme` prenant en paramètres une liste a et deux entiers i et j et retournant la somme $s[i : j]$.

b) En déduire une fonction `coupe_min1` prenant en paramètre une liste a et retournant la somme minimale d'une coupe de a.

Il est possible de montrer que le nombre d'additions effectuées par la fonction `coupe_min1` est proportionnel à n^3, ce qui nous permet de supposer que le temps d'exécution de cette fonction est lui aussi *grosso modo* proportionnel à n^3.

c) Mesurer le temps d'exécution de la fonction `coupe_min1` pour la liste `lst1`.

Si nous avions la mauvaise idée d'utiliser cette fonction pour la liste `lst2`, et en admettant que le temps d'exécution soit effectivement proportionnel à n^3, combien de temps peut-on prévoir d'attendre ? Et pour `lst3` ? On répondra à ces questions en remplissant le tableau ci-dessous :



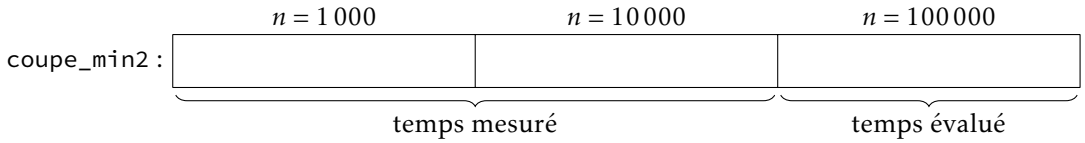
Question 2. un algorithme de coût quadratique

a) Définir (sans utiliser la fonction `somme`) une fonction nommée `mincoupe` prenant en paramètres une liste a et un entier i, et calculant la valeur minimale de la somme d'une coupe de a *dont le premier élément est* a_i.

Combien d'additions cette fonction effectue-telle ?

b) En déduire une fonction `coupe_min2` dont le temps d'exécution est proportionnel à n^2, prenant en paramètre une liste a et retournant la somme minimale d'une coupe de a.

c) Mesurer le temps d'exécution de la fonction `coupe_min2` pour les listes `lst1` et `lst2`. Les deux valeurs obtenues sont-elles compatibles avec une croissance quadratique ? Combien de temps peut-on prévoir d'attendre si nous utilisions cette fonction pour calculer la somme minimale d'une coupe de la liste `lst3` ?

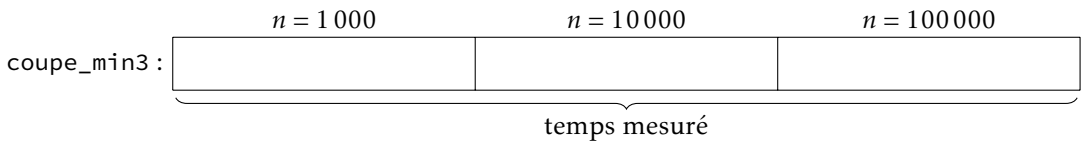


Question 3. un algorithme de coût linéaire

Étant donnée une liste a, on note m_i la somme minimale d'une coupe quelconque de la liste $a[0 : i]$ et c_i la somme minimale d'une coupe de $a[0 : i]$ *se terminant par* a_{i-1}.

a) Montrer que $c_{i+1} = \min(c_i + a_i, a_i)$ et $m_{i+1} = \min(m_i, c_{i+1})$ et en déduire une fonction `coupe_min3` de temps d'exécution linéaire calculant la valeur minimale de la somme d'une coupe de a.

b) Mesurer le temps d'exécution de la fonction `coupe_min3` pour les listes `lst1`, `lst2` et `lst3`. Ces valeurs sont-elles compatibles avec une croissance linéaire ?



Et pour les plus rapides

Question 4. un algorithme de coût quasi-linéaire

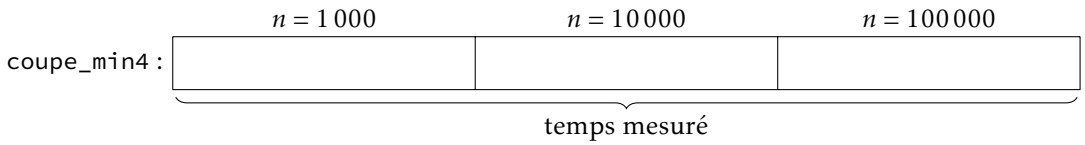
Un algorithme de type *diviser pour régner* est un algorithme qui scinde le problème initial en plusieurs problèmes de taille plus petite, par exemple en deux sous-problèmes de taille deux fois plus petite que le problème initial.

a) Soit $k = \lfloor n/2 \rfloor$. Observer que la coupe minimale de a est :

- soit entièrement contenue dans $a[0 : k]$;
- soit entièrement contenue dans $a[k : n]$;
- soit constituée de la concaténation d'une coupe $a[i_0 : k]$ minimale parmi celles de la forme $a[i : k]$ $(0 \leqslant i < k)$ et d'une coupe $a[k : j_0]$ minimale parmi celles de la forme $a[k : j]$ $(k < j < n)$;

et en déduire une fonction `coupe_min4` utilisant ce principe pour résoudre le problème de la coupe minimale.

b) Mesurer le temps d'exécution de cette fonction pour chacune des listes `lst1`, `lst2` et `lst3`.



Il est possible de montrer que ce temps d'exécution est *grosso modo* proportionnel à $n \log n$; Compte tenu des mesures de temps obtenues comprenez-vous la raison pour laquelle un algorithme ayant une telle complexité est qualifié de *quasi-linéaire* ?

Question 5. On cherche maintenant à déterminer le *gain maximal* du tableau a, à savoir la quantité $\max_{i<j}(a_j - a_i)$. Rédiger une fonction déterminant le gain maximal en temps linéaire.