Sujet F

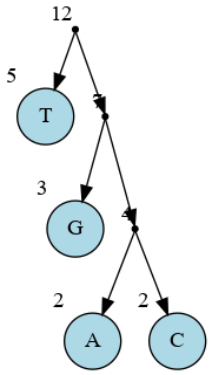
❑ Exercice : *type A*

On s'intéresse dans cet exercice à la compression de chaines de caractères représentant des séquences génétiques codées sur l'alphabet `{A, C, G, T}` avec l'algorithme de Huffmann. On rappelle que le principe de cet algorithme est d'attribuer aux caractères les plus fréquents un code plus court.

1. On veut compresser la séquence S=`ATGTGATGTCCT`, donner le nombre d'occurrences de chaque caractère dans cet séquence.

Caractère	A	C	G	T
Nombre d'occurrences	2	2	3	5

2. Construire l'arbre de Huffman associé à la compression de S.



3. Donner les codes obtenus pour chacun des quatre caractères `A, C, G` et `T`. En déduire le taux de compression de l'algorithme. On supposera qu'initialement la séquence est codée en ASCII et que donc chacun des caractères occupe un octet et on ne tiendra pas compte de la taille de l'arbre.

   — Code pour `T` = `0`
   — Code pour `G` = `10`
   — Code pour `A` = `110`
   — Code pour `C` = `111`

   Comme indiqué sur l'arbre `T` apparaît 5 fois, `G` 3 fois et `A` et `C` chacun deux fois. La taille finale du code est donc : $5 \times 1 + 3 \times 2 + 2 \times 3 + 2 \times 3 = 23$ bits, alors le code initial contenait 12 caractères codés sur 8 bits chacun donc 96 octets, le taux de compression est donc de $\frac{23}{96} \simeq 24\%$.

4. Quelle structure de données abstraite est utilisée pour implémenter cet algorithme ? Quelle en est l'implémentation usuelle ?

   On doit disposer d'une file de priorité afin d'y stocker les sous arbres avec le total du nombre d'occurrences qu'il contiennent. Une file de priorité est usuellement implémentée en utilisant un tas binaire min. Les opérations d'extraction et d'insertion d'un élément sont alors en complexité logarithmique de la taille du tas.

5. Proposer un type en OCaml permettant de représenter un arbre de Huffmann.

```
type abh =
  | Feuille of char
  | Noeud of abh*abh;;
```

❑ Exercice : *type B*

On dispose d'un *système monétaire* c'est-à-dire d'un ensemble de valeurs possibles pour les pièces et les

billets. Le problème du rendu de monnaie consiste à déterminer le nombre minimal de pièces à utiliser pour former une somme donnée. On supposera que les valeurs des pièces et des billets sont des entiers de même que la somme à rendre et que le système monétaire contient toujours la valeur 1. Les fonctions demandées dans cet exercice sont à écrire en OCaml et on donnera le système monétaire sous la forme de la liste *triée dans l'ordre décroissant* des valeurs des pièces. Par exemple, la liste `[500; 200; 100; 50; 20; 10; 5; 2; 1]` est un système monétaire correct et le nombre minimal de pièces pour rendre la somme `17` est `3` (obtenue en utilisant `10 + 5 + 2`.). D'autre part, on s'interdit dans cet exercice l'utilisation des aspects impératifs du langage OCaml (en particulier les boucles et les références.)

1. Ecrire une fonction `verifie : int list -> bool` qui prend en argument un systeme monétaire et renvoie un booléen indiquant si ce système est valide (c'est à dire la liste des valeurs est rangée dans l'ordre décroissant est se termine par 1).

```
let rec verifie systeme =
  match systeme with
  | [] -> false
  | p::[] -> p=1
  | p1::p2::rs -> p1>p2 && verifie (p2::rs);;
```

2. On considère dans un premier temps la méthode consistant à rendre toujours la pièce de plus forte valeur possible. A quelle famille d'algorithme appartient cette méthode ? Justifier

> Il s'agit d'un algorithme glouton car on effectue un choix local optimal (dans le sens où la somme à rendre diminue le plus possible) à chaque étape.

3. En utilisant un exemple de votre choix, montrer que cette méthode ne fournit pas toujours le nombre minimal de pièces.

> On peut par exemple considérer le système monétaire `[5; 4; 3; 1]` et former la somme 7, l'algorithme glouton utilise alors 3 pièces (5+1+1) alors qu'on peut n'en utiliser que 2 (4+3).

4. Donner une implémentation de cette méthode sous la forme d'une fonction `monnaie : int list -> int -> int` prenant en argument un système monétaire ainsi qu'une somme et renvoyant le nombre de pièces.

```
let rec monnaie systeme somme =
  match systeme, somme with
  | _, 0 -> 0
  | [], _ -> failwith "Impossible"
  | p::rs, s -> if p<=s then 1 + monnaie systeme (s-p) else monnaie rs s;;
```

5. On veut maintenant résoudre ce problème par programmation dynamique. On note $(p_i)_{0\leqslant i\leqslant n}$ les valeurs des pièces rangées dans l'ordre décroissant, et on note $m(S,k)$ le nombre minimal de pièces pour rendre la somme S en utilisant seulement les pièces $p_k,\dots,p_{n-1}$. On convient que $m(S,k)=+\infty$ si $S\neq 0$ et $k\geqslant n$ car on cherche alors à rendre une somme non nulle sans utiliser de pièces. Donner la relation liant $m(S,k)$ à $m(S,k+1)$ si on choisit de ne pas utiliser la pièces p_k. De même trouver une relation entre différentes instances du problème lorsqu'on choisit d'utiliser p_k (dans ce cas, on a nécessairement $S\geqslant p_k$).

> — si $S<p_k$ alors $m(S,k)=m(S,k+1)$
> — sinon, $m(S,k)=\min\{1+m(S-p_k,k),m(S,k+1)\}$
>
> D'autre part, $m(S,0)=0$.

6. Ecrire une fonction `dynamique : int list -> int -> int` qui résoud le problème par programmation dynamique, on pourra utiliser la valeur `Int.max` de OCaml afin d'indiquer que la résolution est impossible (nombre infini de pièces).

```
let rec dynamique systeme somme =
  match systeme, somme with
  | _, 0 -> 0
  | [], _ -> Int.max_int
  | pk::rs, s -> if pk>s then dynamique rs somme else
                 let sans = dynamique rs s in
                 let avec = 1 + dynamique systeme (s-pk) in
                 min avec sans;;
```