Algorithmique avancée – Examen Réparti 2 – janvier 2013 – Éléments de corrigé

1 Questions diverses [8 points]

Question 1. [2 points] L'arbre de Huffman est un peigne et donne les codages suivants :

$$a : 0000000 \quad b : 0000001 \quad c : 000001 \quad d : 00001 \quad e : 0001 \quad f : 001 \quad g : 01 \quad h : 1.$$

Cas où les fréquences sont les n premiers nombres de Fibonacci (a_i de fréquence F_i), les codages sont :

$$a_1 : 0^{n-2}.0 \qquad a_2 : 0^{n-2}.1 \qquad a_i : 0^{n-i}.1$$

Question 2. [3 points]
On a 256 caractères $a_1, \ldots, a_{256}$ de fréquences $f_1 \leq f_2 \leq \ldots f_{256}$, avec l'hypothèse : $f_{256} < 2f_1$. Comme $f_{256} \leq f_1 + f_2$, les 128 premières étapes de la construction de l'arbre de Huffman donnent 128 arbres de poids $f_1 + f_2 \leq f_3 + f_4 \leq \ldots \leq f_{255} + f_{256}$, avec $f_{255} + f_{256} \leq 2(f_1 + f_2)$, ces arbres sont de hauteur 1 et sont complets à tous les niveaux. Les 64 étapes suivantes donnent 64 arbres de hauteur 2 complets à tous les niveaux, dont les poids ont toujours la même propriété. En réitérant la construction, on obtient finalement 1 arbre de hauteur 8 complet à tous les niveaux. Les 256 caractères ont donc tous des codes de 8 bits.
Question 3. [3 points]
Avec Huffman : $a : 0$, $b : 1$, d'où un texte compressé de 2000 bits, donc 250 octets.
Avec LZW, le texte se décompose en $a - b - (ab) - (ab)a - (ba) - (ba)b - (ab)^2 - (ab)^2a - (ba)^2 - (ba)^2b - (ab)^3 - (ab)^3a - (ba)^3 - (ba)^3b - \ldots - (ab)^k - (ab)^ka - (ba)^k - (ba)^kb - \ldots$. On a besoin de :
- 1 code pour le premier a,
- 1 code pour le premier b,
- pour les 1998 caractères suivants, 4 codes pour chaque facteur $(ab)^k.(ab)^ka.(ba)^k.(ba)^kb$.

Chacun des facteurs est de longueur $8k + 2$, le nombre de facteurs est majoré par n tel que :

$$\sum_{k=1}^{n-1}(8k + 2) \leq 1998 \leq \sum_{k=1}^{n}(8k + 2).$$

La valeur de n est 23 et le nombre de codes nécessaires pour le texte est majoré par $2 + 4 * 23 = 94$.
Avec LZW, 94 octets suffisent pour compresser $(ab)^{1000}$.
Conclusion : LZW est meilleur que Huffman.

2 Compression par anti-dictionnaire [12 points]

Question 1. [1 point] Plein de possibilités !
Question 2. [3 points]
$T = 01101010$ et $AD(T) = \{00, 111, 1011\}$.

v	TC	facteur dans $AD(T)$
0	0	00
01	0	—
011	01	111
0110	01	00
01101	01	1011
011010	01	00
0110101	01	1011
01101010	01	

$T = 01101010$ et $AD(T) = \{00, 0111\}$.

v	TC	facteur dans $AD(T)$
0	0	00
01	0	—
011	01	0111
0110	01	00
01101	01	—
011010	010	00
0110101	010	—
01101010	0100	

Question 3. [2 points] Dans l'algorithme $T[1..n]$ est le texte à compresser, $E = AD(T)$, TC est le texte compressé, k est le max des longueurs des mots de E.

Algorithm 1 Compression par anti-dictionnaire

```
for i = 1 to n do
  j ← i
  while j ≥ 1 do
    if i − j + 1 > k then
      j ← 0
    else if estDans(T[j..i].¬T[i + 1], E) then
      j ← −1
    else
      j ← j − 1
    end if
  end while
  if j = 0 then
    TC ← TC + T[i + 1]
  end if
end for
```

Complexité : $O(nk)$.

Question 4. [2 points] Dans l'exemple où $AD(T) = \{00, 111, 1011\}$ et le texte compressé est 01, si on ne fournit pas la taille du texte T, 01 peut se décompresser en 011 ou en n'importe quel texte de la forme $011(01)^m$ ou $011(01)^m0$.

On connaît $TC = 01$, $AD(T) = \{00, 111, 1011\}$ et on sait que la taille de T est 8. On lit $TC[1] = 0$ donc $T = 0$, 00 est interdit donc $T = 01$. Il n'y a aucun suffixe de 01 qui se complète en un mot interdit, on lit donc $TC[2] = 1$ et on l'écrit dans T, donc $T = 011$. 111 est interdit donc $T = 0110$, 00 est interdit donc $T = 01101$. 1011 est interdit donc $T = 011010$. 00 est interdit donc $T = 0110101$. 1011 est interdit donc $T = 01101010$. On arrête car on a atteint la taille de T.

Question 5. [2 points] Dans l'algorithme $TC[1..p]$ est le texte à décompresser, $E = AD(T)$, k est le max des longueurs des mots de E.

Algorithm 2 Décompression par anti-dictionnaire

```
i ← 1
m ← 1
while i ≤ p and m < n do
  j ← i
  while j ≥ 1 do
    if i − j + 1 > k then
      j ← 0
    else if estDans(T[j..i].0, E) then
      T ← T + 1
      j ← −1
    else if estDans(T[j..i].1, E) then
      T ← T + 0
      j ← −1
    else
      j ← j − 1
    end if
  end while
  if j = 0 then
    i ← i + 1
    T ← T + TC[i]
  end if
  m ← m + 1
end while
```

Complexité : $O(nk)$.
Question 6. [1 point] à voir...
Question 7. [1 point] à voir...

3 Triangulations d'un polygone simple [12 points]

Dans toutes les questions, on connaît un contour positif $(p_0, \ldots, p_{n-1})$ du polygone considéré.
Question 1. [1 point] Dans une liste initialement vide, on ajoute un à un les triangles $p_0p_ip_{i+1}$, pour i allant de 1 à $n-2$. En comptant les ajouts dans une liste, la complexité est en $\Theta(n)$.
Question 2. [3 points] Supposons que $p_{i-1}p_ip_{i+1}$ soit une oreille. Lorsqu'on coupe cette oreille, le nouveau contour est $(p_0, \ldots, p_{i-1}, p_{i+1}, \ldots, p_{n-1})$ (sauf dans des cas particuliers, traités juste après). Le seul côté susceptible d'intersecter un côté qui ne lui est pas adjacent est $[p_{i-1}, p_{i+1}]$, mais ce n'est pas le cas, par définition (bis) d'une oreille. Les cas particuliers :
- p_{i-1}, p_{i+1}, p_{i+2} alignés → le nouveau contour est $(p_0, \ldots, p_{i-1}, p_{i+2}, \ldots, p_{n-1})$, le seul côté susceptible d'intersecter un côté qui ne lui est pas adjacent est $[p_{i-1}, p_{i+2}]$, mais ce n'est pas le cas car :
  - ou bien p_{i+1} est entre p_{i-1} et p_{i+2}, et alors $[p_{i-1}, p_{i+2}]$ est l'union de $[p_{i-1}, p_{i+1}]$ (côté de l'oreille) et $[p_{i+1}, p_{i+2}]$ (côté du polygone simple initial),
  - ou bien p_{i+2} est entre p_{i-1} et p_{i+1}, et alors $[p_{i-1}, p_{i+2}]$ est contenu dans $[p_{i-1}, p_{i+1}]$ (côté de l'oreille),
  - le cas p_{i-1} entre p_{i+1} et p_{i+2} est exclu car le polygone initial est simple ;
- p_{i-2}, p_{i-1}, p_{i+1} alignés → le nouveau contour est $(p_0, \ldots, p_{i-2}, p_{i+1}, \ldots, p_{n-1})$, idem ;
- p_{i-2}, p_{i-1}, p_{i+1}, p_{i+2} alignés → le nouveau contour est $(p_0, \ldots, p_{i-2}, p_{i+2}, \ldots, p_{n-1})$, idem.

On montre qu'un polygone simple ayant n sommets admet au moins une triangulation et que cette triangulation a au plus $n-2$ triangles. Si $n = 3$, c'est vrai. Si $n > 3$ le polygone est l'union, sans recouvrement, d'une oreille et d'un polygone simple ayant au plus m sommets, avec $m \leq n-1$. Par hypothèse de récurrence, il existe une triangulation pour m sommets et elle compte au plus $m-2 \leq n-3$ triangles. Il existe donc une triangulation du polygone simple à n sommets et elle compte au plus $n-2$ triangles.
Remarque : toutes les triangulations d'un polygone n'ont pas le même nombre de triangles. Par exemple, considérons un polygone simple de contour $(p_0, p_1, p_2, p_3, p_4, p_5)$ tel que $p_2p_3p_4$ est une oreille et p_1, p_2, p_4, p_5 sont alignés dans cet ordre. Il admet une triangulation ayant 2 triangles : $(p_0p_1p_5, p_2p_3p_4)$, une triangulation ayant 4 triangles : $(p_0p_1p_2, p_0p_2p_4, p_0p_4p_5, p_2p_3p_4)$, et des triangulations ayant 3 triangles.
Question 3. [1 point] Pour tester si $p_ip_{i+1}p_{i+2}$ est une oreille :
- tester si $(p_i, p_{i+1}, p_{i+2}$ est un tour gauche,
- puis tester si `intersecte?`$(p_i, p_{i+2}, p_j, p_{j+1})$ pour j allant de $i+3$ à $i-2$.

En comptant les appels à `intersecte?`, la complexité est en $0(n)$.
Pour déterminer une oreille, on teste si $p_ip_{i+1}p_{i+2}$ est une oreille, pour i allant de 0 à $n-1$. Complexité en $0(n^2)$.
Question 4. [1 point] Pour trianguler un polygone simple :
- déterminer une oreille,
- calculer une triangulation du polygone simple obtenu en coupant cette oreille,
- ajouter l'oreille à cette triangulation.

Avec cette méthode on a une complexité en $0(n^3)$.
Question 5. [1 point] Quand on enlève une pointe p_i à une montagne, on lui coupe une oreille donc le polygone obtenu est simple. La pointe n'est ni p_0 ni p_1 donc le nouveau contour est $(p_0, p_1, \ldots)$, avec $abscisse(p_0) < abscisse(p_1)$ et les abscisses qui décroissent de p_1 à p_0. Le nouveau polygone est donc une montagne.

Question 6. [2 points]
Principe de l'algorithme, calqué sur la fin de l'algorithme de Graham : éliminer les tours gauches.
On connaît un contour positif de la montagne et on dispose des fonctions *pred* et *succ*.

Algorithm 3 Triangulation d'une montagne

```
L ← ∅
p ← succ(p_0)
while p ≠ p_0 do
  if (p, succ(p), succ(succ(p))) n'est pas un tour gauche then
    p ← succ(p)
  else
    supprimer succ(p) du contour
    ajouter (p, succ(p), succ(succ(p))) à L
    if pred(p) ≠ p_0 then
      p ← pred(p)
    end if
  end if
end while
```

Remarque : dans le cas où $(p, succ(p), succ(succ(p)))$ est un tour gauche, on pourrait aussi supprimer p du contour si $pred(p)$, p et $succ(succ(p))$ sont alignés.
Comme dans l'algorithme de Graham, on a une complexité linéaire.

Question 7. [1 point]

1. Le polygone $(p_0, p_1, p_m, p_{m+1}, \ldots, p_{n-1})$ est une montagne.
2. Le sommet p_k fait partie du contour $(p_1, p_2, \ldots, p_{m-1}, p_m)$. Les abscisses de p_1, ... , p_k croissent et les abscisses de p_k, p_{k+1}, ... , p_{n-1}, p_1 décroissent. Le polygone $(p_1, p_2, \ldots, p_{m-1}, p_m)$ est donc monotone.

Question 8. [1 point] Si $abscisse(p_{n-1}) = abscisse(p_1)$ alors le polygone initial est l'union, sans recouvrement, du triangle $p_0p_1p_{n-1}$ et d'un polygone monotone.

Question 9. [1 point]

– Si $abscisse(p_{n-1}) = abscisse(p_1)$, on ajoute le triangle $p_0p_1p_{n-1}$ à la triangulation du polygone monotone $(p_1, \ldots, p_{n-1})$
– Si $abscisse(p_{n-1}) < abscisse(p_1)$, on détermine le plus petit indice m tel que $abscisse(p_m) < abscisse(p_1)$ en reculant dans la liste ($n-m$ tests). On obtient une montagne $(p_0, p_1, p_m, p_{m+1}, \ldots, p_{n-1})$, que l'on triangule avec l'algorithme de triangulation d'une montagne (complexité en $n - m$) et un polygone monotone $(p_1, p_2, \ldots, p_{m-1}, p_m)$.
– Si $abscisse(p_{n-1}) > abscisse(p_1)$: analogue au cas précédent.

Complexité linéaire.