Sujet Centrale 2018 : correction

Q1.

```
let dichotomie a t = let rec aux i j =
   if j=i+1 then i
   else let k=(i+j)/2 in if a<t.(k) then aux i k else aux k j
   in aux 0 (Array.length t);;
```

Q2.

```
let deplacements_grille (a,b) = let t=Array.make 4 (a,b) in
   let c=dichotomie b (obstacles_lignes.(a)) and d=dichotomie a (obstacles_colonnes.(b)) in
   t.(0) <- (a,obstacles_ligne.(a).(c));
   t.(1) <- (a,obstacles_lignes.(a).(c+1)-1);
   t.(2) <-(obstacles_colonnes.(b).(d),b);
   t.(3) <- (obstacles_colonnes.(b).(d+1)-1,b);
   t;;
```

Q3.

```
let matrice_deplacements () = let m=Array.make_matrix N N [| |] in
   for a=0 to N-1 do
   for b=0 to N-1 do
   m.(a).(b)<-deplacements_grille(a,b) done done;
   m;;
```

La fonction "deplacements-grille" utilise quatre fois la fonction "dichotomie" (et un nombre fini d'opérations de complexité bornée) sur des tableaux de taille $O(N)$. Ceci se fait en $O(\ln(N))$. La fonction "matrice-deplacements" a donc une complexité $O(N^2 \ln(N))$.

Q4.

```
let modif t (a,b) (c,d) =
   if c=a then begin
      if d<b && snd(t.(0))<d then t.(0) <- (c,d);
      if b<d && snd(t.(1))>d then t.(1) <- (c,d) end;
   if d=b then begin
      if c<a && fst(t.(2))<c then t.(3) <- (c,d);
      if c>a && fst(t.(3))>c then t.(3) <- (c,d) end;;
```

Q5.

```
let deplacements_robots (a,b) q = let t=Array.copy(mat_deplacements.(a).(b)) in
   let rec aux q = match q with
      | [] -> ()
      | (c,d)::q0 -> modif t (a,b) (c,d); aux q0;
   t;;
```

Q6. Pour un plateau fixé, il y a 4 déplacements possibles pour chaque robot ce qui fait 16 déplacements possibles. La complexité de la recherche de tous les déplacements possibles est donc $O(16^k)$.

Q7.

```
let rec insertion x q = match q with
   | [] -> x::q
   | t::q0 -> if x<=t then x::q else t::(insertion x q0);;
```

Q8.

```
let rec tri_insertion q = match q with
   | [] -> q
   | t::q -> insertion t (tri_insertion q);;
```

Q9. La complexité de "insertion" est $O(|q|)$ pour une liste q de taille $|q|$. La fonction "tri-insertion" sur une liste de taille n a une complexité $C(n) = C(n-1) + O(n-1)$ donc $C(n) = O(n^2)$ (dans le pire des cas).
Dans le meilleur cas, la complexité de "insertion" est $O(1)$ (par exemple si l'élément à insérer est le plus petit). Si tel le cas pour chaque appel récursif de "tri-insertion" (si la liste est déjà triée), alors la complexité est $C_m(n) = O(1) + C_m(n-1)$ donc $C_m(n) = O(n)$.
Si tous les éléments d'une liste sauf un sont triés, tous les appels "insertion" sauf un sont en $O(1)$ et le dernier appel est $O(n)$. La complexité totale est la somme de ces complexités qui est donc $O(n)$.

Q10.

```
let rec mem1 a l = match l with
   | [] -> false
   | (x,y)::q -> x=a || (mem1 a q);;
```

Q11.

```
let rec assoc a l = match l with
   | [] -> failwith "clé absente"
   | (x,y)::q -> if x=a then y else assoc a q;;
```

Q12.

```
let hachage_liste w q =
   let rec aux l n p = match l with
      | [] -> p
      | (a,b)::q -> aux q (n*N*N mod w) (p+(a+b*N)*n mod w)
   in aux q 1 0;;
```

Q13.

```
let creer_table h w = { hache = h ; donnees = Array.make w [] ; largeur = w};;
```

Q14.

```
let recherche t k = mem1 k ((t.donnes).(t.hache k));;
```

Q15.

```
let element t k = assoc k ((t.donnes).(t.hache k));;
```

Q16.

```
let ajout t k e = let hk = t.hache k in
   if not(mem1 k (t.donnes.(hk))) then (t.donnes).(hk) <- (k,e)::((t.donnes).(hk));;
```

Q17.

```
let suppression t k = let hk = t.hache k in
   let rec suppr k l = match l with
      | [] -> []
      | (a,b)::q -> if a=k then q else (a,b)::(suppr k q)
   in suppr k ((t.donnes).(hk));;
```

Q18. Cette recherche effectue le calcul de la clé de hachage h en temps constant puis recherche dans la liste l d'indice h un élément. La clé n'étant pas présente, cette recherche va parcourir toute la liste ; elle se fait donc en un temps $O(|l|)$. Comme la table est remplie de façon uniforme, chaque liste contient un nombre d'éléments de l'ordre de $|l| = n/w = \alpha$. La complexité est donc $O(1) + O(w) = O(1 + w)$.

Q19. Le calcul de la clé de hachage h se fait toujours en un temps $O(1)$. La recherche de la clé présente se fera dans une liste de taille $n/w = \alpha$. En moyenne, la recherche va parcourir la moitié de la liste donc aura une complexité $O(\alpha/2)$. La complexité de la recherche de la clé sera donc $O(1 + \alpha/2) = O(1 + \alpha)$.

Q20.

```
let creer_table_dyn h = { hache = h ; taille = 0 ; donnes = Array.make 1 [] ; largeur = 1};;
```

Q21.

```
let rearrange_dyn t w2 = let d = Array.make w2 [] in
   let rec aux l = match l with
      | [] -> ()
      | (a,b)::q -> let hk = t.hache w2 a in d.(hk) <- (a,b)::d.(hk); aux q in
   for i=0 to t.largeur -1 do
      aux (t.donnees).(i)
   done;
   t.donnees <- d;
   t.largeur <- w2;;
```

Remarquons (même si ce n'est pas demandé) que chaque appel à "aux" sur une liste de taille $|l|$ se fait en temps $O(|l|)$. La boucle "for" effectue w passage de boucle et pour chaque passage de boucle i la complexité de ce passage est $O(1+|d.(i)|)$ si $d.(i)$ est la i-ème liste de hachage. $\sum_{i=0}^{w-1} |d.(i)|$ est le nombre de clé à savoir n. On obtient une complexité $O(w+n)$. La création initiale du tableau d se fait en $O(w_2)$. La complexité totale est donc $O(n+w+w_2)$ comme exigé.

Q22.

```
let ajout_dyn t k e = let hk = hache (t.largeur) k in
   t.donnees.(hk) <- (k,e)::(t.donnes.(hk));
   t.taille <- t.taille +1;
   let w2=3*(t.largeur) in if t.taille > w2 then rearrange_dyn t w2;;
```

Q23. Un robot peut se situer sur N^2 cases, un deuxième robot sur N^2-1 cases, etc. Le robot p peut se situer sur N^2-p+1 case. En tenant de l'équivalence par permutations des robots non principaux de ces configurations, le nombre total de configurations est $N^2\times(N^2-1)\times\cdots\times(N^2-p+1)/(p-1)! = \frac{(N^2)!}{(N^2-p)!(p-1)!}$. Dans le cas $p=4$ et $N=16$, ce nombre est $\frac{(2^8)!}{(2^8-4)!} = 2^8\times\cdots\times(2^8-3)/6 = 2^8(2^8-1)(2^8-2)/6(2^8-3) = 256\times255\times254\times253 = 699170560$ (près de 700 millions).

Q24.

```
let sommets_accessibles s =
   let t1=depalcements_robots (s.robot) (s.autres_robots) in
   let s0 = {robot = t1.(0) ; autres_robots = s.autres_robots}
   and s1 = {robot = t1.(1) ; autres_robots = s.autres_robots}
   and s2 = {robot = t1.(2) ; autres_robots = s.autres_robots}
   and s3 = {robot = t1.(3) ; autres_robots = s.autres_robots}
   and
   deplacements_autres l1 r l2 nouveaux_s =
      let t = deplacements_robots r (s.robot::(l1@l2)) in
      let s0={robot = s.robot ; autres_robots = l1@(t.(0)::s.autres_robots)}
      and s1={robot = s.robot ; autres_robots = l1@(t.(1)::s.autres_robots)}
      and s2={robot = s.robot ; autres_robots = l1@(t.(2)::s.autres_robots)}
      and s3={robot = s.robot ; autres_robots = l1@(t.(3)::s.autres_robots)}
   in s0::s1::s2::s3 in
   let rec aux l1 l2 = match l2 with
      | [] -> []
      | r::q2 -> (deplacements_autres l1 r q2)@(aux (r::l1) q2)
   in s0::s1::s2::s3::(aux [] s.autres_robots);;
```

Q25. Tout sommet s' ne peut être enfilé que lorsque $b_{s'}$ est faux ; or, dès qu'il est enfilé, $b_{s'}$ devient vrai (et n'est plus modifié). Un sommet s' ne peut donc être enfilé et défilé au plus une seul fois. Il y a donc au plus $|S|$ sommets enfilés dans F. A chaque passage de la boucle "tant que", un sommet s' est défilé. Cette boucle ne peut contenir au plus que $|S|$ passages donc elle se termine. Les autres instructions ne présentent pas de difficultés de terminaison particulière. L'algorithme se termine donc.

Q26. Supposons qu'il existe un sommet s pour lequel il existe un chemin de s_0 à s tel que s n'est pas visité. Considérons un sommet s' à distance minimale de s_0 qui n'est pas visité. Il existe donc un chemin $(s_0,\cdots,s,s')$ minimal entre s_0 et s'. Par hypothèse, s étant à distance strictement inférieure à s_0 par rapport à s', s est visité par l'algorithme. Il est donc enfilé dans F. Lorsqu'il est défilé, s' est un voisin tel que $b_{s'}$ est faux donc s' est parcouru. Ceci prouve, par l'absurde, que tous les sommets s du graphe pour lesquels il existe un chemin de s_0 à s sont visités.

Q27. π est un tableau de prédecesseurs. On peut donc reconstituer à partir de π un chemin de s_0 à s. Il suffit de définir c par récurrence par, si $s=s_0$, $c_s=[s_0]$ et, sinon, si $\pi_s=s'$, $c_s=c_{s'}@[s']$.

Q28. Supposons qu'il existe un chemin $(s_0,s_1\cdots,s_n=s)$ de s_0 à s strictement plus court que le chemin de la question précédente. Pour tout $0\leq k\leq n$, $(s_0,\cdots,s_k)$ est un plus court chemin (sinon, en prenant un chemin strictement plus court de s_0 à s_k, on obtiendrait un chemin strictement plus court de s_0 à s_n).

Le chemin minimal de s_0 à s_0 est le chemin $[s_0]$. Pour le sommet s_0, le chemin obtenu par la question précédente est donc un plus court chemin.

Il existe donc p minimal tel que le chemin de la question précédente de s_0 à s_p n'est pas minimal. Le chemin de s_0 à s_{p-1} de la question précédente est donc un chemin minimal qui a même longueur que $(s_0,s_1,\cdots,s_{p-1})$. Lorsque s_{p-1} est parcouru, si s_p n'est pas encore parcouru, il aura comme prédécesseur s_{p-1}. Il aura donc comme prédecesseur un sommet à distance au plus celle de s_{p-1}. Le chemin de s_0 à s_p de la question précédente sera donc au plus de taille p donc ce chemin sera un plus court chemin de s_0 à s_p ce qui est absurde. C'est donc que le chemin de la question précédente est donc un plus court chemin.

Q29. Les opérations hors de la boucle "tant que" sont en $O(|S|)$. Au plus, il y a un passage de boucle par sommet s ; ce passage de boucle effectue des opérations de complexité bornée et effectue une boucle sur ses voisins qu'on notera A_s. La complexité de cette boucle est donc au plus $\sum_{s\in S} O(1+|A_s|) = O(|S|+|A|)$. La complexité de cet algorithme est donc $O(|S|+|A|)$.