Correction du TD 13

Julien Reichert

Recherche dans un texte - Algorithme naïf

Exercice 1

```
int* recherche_naive(char* s, char* m) {
  int ns = strlen(s);
  int nm = strlen(m);
  int c_f = 0;
  int* final = malloc(ns * sizeof(int));
  for (int i = 0 ; i <= ns - nm ; i++)
  {
    int count = 0;
    while (count < nm && s[i + count] == m[count]) count += 1;
    if (count == nm)
    {
      final[c_f] = i;
      c_f += 1;
    }
  }
  int* final2 = malloc((c_f+1) * sizeof(int));
  final2[0] = c_f;
  for (int i = 0 ; i < c_f ; i += 1) final2[i+1] = final[i];
  free(final);
  return final2;
}
```

Recherche dans un texte - Algorithme de Boyer-Moore

Exercice 2

	'C'	'H'	'E'	'R'	'Z'
0 ('C')	OK				
1 ('H')	+1	OK			
2 ('E')	+2	+1	OK		
3 ('R')	+3	+2	+1	OK	
4 ('C')	OK	+3	+2	+1	
5 ('H')	+1	OK	+3	+2	
6 ('E')	+2	+1	OK	+3	
7 ('Z')	+3	+2	+1	+4	OK

Exercice 3

Premier cas : Recherche de `"CHERCHEZ"` dans `"CHERCHER"` répété n fois.

Avec l'algorithme naïf : 8 comparaisons à partir de chaque "C" sauf le dernier qui est hors champ, 1 comparaison à partir de chaque autre lettre sauf les 7 dernières positions, d'où $(n-1) \times (8+1+1+1+8+1+1+1)+8$ soit $22n-14$ comparaisons.

Avec l'algorithme de Boyer-Moore : décalage de 4 crans à chaque étape jusqu'à atteindre la fin et échouer, d'où $2n-1$ comparaisons.

Deuxième cas : Recherche de `"CHERCHEZ"` dans `"CHERCHEZ"` répété n fois.

Avec l'algorithme naïf : 8 comparaisons à partir de chaque "C" dont le « numéro de l'occurrence » est pair, 4 à partir de chaque autre C sauf le dernier qui est hors champ, 1 comparaison à partir de chaque autre lettre sauf les 7 dernières positions, d'où $(n-1) \times (8+1+1+1+4+1+1+1)+8$ soit $18n-10$ comparaisons.

Avec l'algorithme de Boyer-Moore : complétion du motif et décalage d'un cran, puis décalage de 3 crans puis décalage de 4 crans pour tomber sur le motif suivant jusqu'à atteindre la fin et, d'où $(n-1) \times (8+1+1)+8$ soit $10n-2$ comparaisons.

Troisième cas : Recherche de `"CHERCHEZ"` dans `"CHEZCHEZ"` répété n fois.

Avec l'algorithme naïf : 4 comparaisons à partir de chaque "C" sauf le dernier qui est hors champ, 1 comparaison à partir de chaque autre lettre sauf les 7 dernières positions, d'où $(n-1) \times (4+1+1+1+4+1+1+1)+4$ soit $14n-10$ comparaisons.

Avec l'algorithme de Boyer-Moore : 4 comparaisons puis décalage de 4 crans à chaque étape jusqu'à atteindre la fin et échouer, d'où $10n-5$ comparaisons.

Exercice 4

Version optimisée pour `"CHERCHEZ"` : tous les décalages sauf la dernière ligne sont remplacés par des +8, car il est impossible de trouver le motif dans la zone si un `'Z'` y figure à la fin et que la reconnaissance échoue par ailleurs, puisque cette lettre est unique dans le motif.

Version optimisée pour `"CHERCHER"` :

	'C'	'H'	'E'	'R'
0 ('C')	OK	+4	+4	+4
1 ('H')	+4	OK	+4	+4
2 ('E')	+4	+4	OK	+4
3 ('R')	+4	+4	+4	OK
4 ('C')	OK	+8	+8	+8
5 ('H')	+8	OK	+8	+8
6 ('E')	+8	+8	OK	+8
7 ('R')	+3	+2	+1	OK

Exercice 5

```
let redirection motif =
  let n = String.length motif in
  let t = Array.init n (fun _ -> Hashtbl.create 8) in (* Pas make, c'est dépendant ! *)
  Hashtbl.add t.(0) motif.[0] 0;
  for i = 1 to n-1 do
    Hashtbl.iter (fun cle valeur -> Hashtbl.add t.(i) cle (valeur+1)) t.(i-1);
    Hashtbl.replace t.(i) motif.[i] 0;
  done;
  t;;

let boyer_moore chaine motif =
  let t_r = redirection motif in
  let nm = String.length motif in
  let nc = String.length chaine in
  let rec aux ic im =
    if ic > nc - nm then []
    else match Hashtbl.find_opt t_r.(im) chaine.[ic + im] with
      | Some 0 when im = 0 -> ic :: aux (ic + 1) (nm - 1)
      | Some 0 -> aux ic (im - 1)
      | Some a -> aux (ic + a) (nm - 1)
      | None -> aux (ic + im + 1) (nm - 1)
  in aux 0 (nm-1);;
```

Recherche dans un texte - Algorithme de Rabin-Karp

Exercice 6

Avec la convention du cours stipulant que le degré augmente de droite à gauche (plus facile pour l'algorithme final), le haché de `"mot"` est $116 + 111 \times 31 + 109 \times 31^2 = 108306$.

Exercice 7

Plus petit haché possible : 0 en utilisant le caractère de fin de chaîne trois fois (même si ce n'est pas censé se produire). Plus grand haché possible : 253215 en utilisant le caractère d'indice 255 trois fois. En raison des collisions, le nombre de hachés différents n'est pas de $256^3 = 16777216$ mais de 253216 (tous les entiers de l'intervalle, en fait).

Exercice 8

```
int hachage (char* s, int b, int p)
{
  int final = 0;
  for (int i = 0 ; s[i] != '\0' ; i += 1)
  {
    final = final * b + (int) s[i];
    if (p != 0) final = final % p;
  }
  return final;
}
```

Exercice 9

```
int hachage_suivant (int x, char* s, int b, int k, int bpkm1, int ind)
{
  return (x - s[ind] * bpkm1) * b + s[ind + k];
}
```

Exercice 10

```
int hachagebis (char* s, int b, int p, int taille)
{
  int final = 0;
  for (int i = 0 ; i < taille ; i += 1)
  {
    final = final * b + (int) s[i];
    if (p != 0) final = final % p;
  }
  return final;
}
```

La fonction ci-avant calcule le premier haché d'une tranche. Ci-après, l'algorithme proprement dit.

```
int* rabin_karp(char* s, char* m, int b, int p) {
  int ns = strlen(s);
  int nm = strlen(m);
  if (nm > ns) return NULL;
  int c_f = 0;
  int* final = malloc(ns * sizeof(int));
  int hm = hachage(m, b, p);
  int hs = hachagebis(s, b, p, nm);
  int bpkm1 = 1;
  for (int i = 1 ; i < nm ; i += 1) bpkm1 = p == 0 ? bpkm1 * b : (bpkm1 * b) % p;
  for (int i = 0 ; i <= ns - nm ; i++)
  {
    if (hm == hs)
    {
      int count = 0;
      while (count < nm && s[i + count] == m[count]) count += 1;
      if (count == nm)
      {
        final[c_f] = i;
        c_f += 1;
      }
    }
    hs = hachage_suivant(hs, s, b, nm, bpkm1, i);
    if (p != 0) hs = (hs % p + p) % p; // Risque de valeurs négatives ici !
  }
  int* final2 = malloc((c_f+1) * sizeof(int));
  final2[0] = c_f;
  for (int i = 0 ; i < c_f ; i += 1) final2[i+1] = final[i];
  free(final);
  return final2;
}

int main(int argc, char** argv)
{
  int b = atoi(argv[1]);
  int p = argc > 2 ? atoi(argv[2]) : 0;
  char m[] = "ABCD";
  char s[] = "ACBDACBDABACDBBDA";
  int* rk = rabin_karp(s, m, b, p);
  printf("Nombre d'occurrences du motif : %d.\n", rk[0]);
  printf("Première (seule en fait) occurrence : en %d.\n", rk[1]);
  free(rk);
  return 0;
}
```

Compression - Algorithme de Huffman

Exercice 11

L'arbre sera complet en pratique. C'était à prévoir. L'algorithme de Huffman n'est pas nécessaire ici et pour cause.

Exercice 12

L'arbre sera un peigne selon une définition alternative et au changement d'ordre des fils de chaque nœud près. En tout cas chaque nœud interne aura au moins une feuille pour fils. Ici, l'efficacité est maximale pour l'algorithme de Huffman.

Exercice 13

[Chacun pourra faire la correction comme il l'entend.]

Exercice 14

La preuve se fait par récurrence sur le nombre n de lettres différentes du texte, donc le nombre de feuilles de l'arbre de Huffman, avec une initialisation pour $n = 2$ (un seul arbre possible à l'ordre près des fils de la racine, donc l'optimalité est garantie).

Concernant l'hérédité, on isole les deux lettres, notées x et y, qui sont les moins fréquentes et ont donc fusionné dans la première étape de l'algorithme et on considère une pseudo-lettre appelée z qui n'est pas dans le texte et à laquelle on attribuerait pour « poids » la somme des poids (nombre d'occurrences) de x et de y. Par récurrence, l'arbre de Huffman pour la nouvelle distribution des lettres est optimal, et le nombre de bits utilisé pour représenter le texte adapté (x et y étant remplacés par z) est noté T. Remplacer dans cet arbre la feuille de z par un nœud ayant x et y pour enfants occasionne l'augmentation du nombre de bits pour écrire le texte d'une unité par apparition de x ou y (les deux feuilles en question sont à un niveau de profondeur plus bas que l'ancienne feuille pour z), cette différence qu'on note δ porte donc le total à $T + \delta$.

Considérons un codage préfixe optimal. Alors les feuilles marquées par x et y ne seront pas forcément deux frères, mais les échanger avec deux feuilles à la profondeur maximale ne peut pas empirer le code (et vu l'hypothèse cela ne l'améliore pas non plus). Alors la même transformation du nœud ayant à présent ces feuilles pour fils donne un code qui a économisé δ bits au total de manière analogue à ce qu'on a vu.

L'hypothèse de récurrence permet de conclure à partir de l'optimalité du code ayant utilisé z, car le codage préfixe optimal après fusion utilise au moins T bits pour le texte, donc sans la fusion au moins $T + \delta$, or c'est exactement ce qu'utilise le codage de Huffman.

Compression - Algorithme de Lempel-Ziv-Welch

Exercice 15

Considérons une chaîne avec n répétitions du caractère `'A'`.

Alors le premier `'A'` sera représenté par le code `65`, les deux suivants par le nouveau code `256`, les trois suivants par le nouveau code `257`, etc.

Au total, il y aura k codes pour un texte de taille $n = \frac{k(k+1)}{2}$ et entre deux valeurs de cette forme on fait un arrondi supérieur.

Le nombre de bits utilisés sera de l'ordre de la racine de la taille du texte, en négligeant la taille de chaque code qui sera de huit bits plus une valeur logarithmique en le nombre de codes.

Exercice 16

Considérons une chaîne avec n répétitions de la sous-chaîne `"AB"`.

Alors le premier `'A'` sera représenté par le code `65`, le premier `'B'` par le code `66`, le deuxième `"AB"` en entier par le nouveau code `256`, puis c'est plus technique car le code suivant, à savoir `258`, représentera le facteur `"ABA"` qui suit, après quoi le facteur suivant est `"BA"`, représenté par le code `257`...

Quoi qu'il en soit, on observe que les tailles de facteur évoluent selon une tendance linéaire, pour un résultat qui restera de l'ordre de la racine de la taille de départ.

Exercice 17

Une chaîne utilisant une et une seule fois chaque caractère ne sera pas réduite, et en finissant par les deux caractères du début on se force à utiliser un nouveau code, ce qui garantit que le nombre total de bits sera supérieur au nombre de bits dans la représentation naturelle.

En pratique, on peut chercher des cas plus grands, car une compression sans perte ayant toujours un taux supérieur à un n'existe pas.

Cependant, il y a une subtilité : le nombre de textes sur au plus n octets est de $\sum_{i=0}^{n} 256^i$, mais le nombre de résultats de l'algorithme de LZW sur au plus n octets est de $\sum_{j=0}^{8n} 2^j$ (environ le double), de par la possibilité de s'arrêter en plein milieu d'un octet qu'on complètera comme on pourra. Cette richesse supplémentaire peut faire perdre sa pertinence au paragraphe précédent...

On notera quand même que le nombre de résultats sur au plus n octets moins un bit n'est pas supérieur, et qu'un gain suppose la diminution d'au moins un octet complet, ce qui sauve la preuve en ayant bien gardé à l'esprit la subtilité !

Exercice 18

```
(* Tableaux redimensionnables sans suppression ni modification. *)

type 'a tr = { mutable taille : int; mutable donnees : 'a array };;

let creer_tr capacite initial = { taille = 0 ; donnees = Array.make capacite initial };;

let acces_tr t i =
  assert (i >= 0 && i < t.taille);
  t.donnees.(i);;

let redimensionne_tr t =
  let nv_taille = 2 * t.taille in
  let nv_tab = Array.make nv_taille t.donnees.(0) in
  for i = 1 to t.taille - 1 do nv_tab.(i) <- t.donnees.(i) done;
  t.donnees <- nv_tab;;

let append_tr t x =
  if t.taille = Array.length t.donnees then redimensionne_tr t;
  t.donnees.(t.taille) <- x;
  t.taille <- t.taille + 1;;

let soc c = String.make 1 c;;

let soi i = soc (char_of_int i);;

let lzw_compression texte =
  assert (texte <> ""); (* Pas de blague ! *)
  let dico = Hashtbl.create 512 in
  for i = 0 to 255 do Hashtbl.add dico (soi i) i done;
  let taille_dico = ref 256 in
  let buff = ref (soc texte.[0]) in
  let pseudosortie = creer_tr 8 0 in
  for i = 1 to String.length texte - 1 do
    let nv_buff = !buff ^ (soc texte.[i]) in
    if Hashtbl.mem dico nv_buff then buff := nv_buff
    else
    begin
      Hashtbl.add dico nv_buff !taille_dico;
      incr taille_dico;
      append_tr pseudosortie (Hashtbl.find dico !buff);
      buff := soc texte.[i]
    end
  done;
  append_tr pseudosortie (Hashtbl.find dico !buff);
  pseudosortie;;
```

Exercice 19

On exploite cette fois-ci la structure de tableau redimensionnable pour le dictionnaire de décodage aussi.

```
let lzw_decompression pseudosortie =
  assert (pseudosortie.taille <> 0); (* Même remarque... *)
  let dicorev = creer_tr 512 "" in
  for i = 0 to 255 do append_tr dicorev (soi i) done;
  let code_prec = ref (acces_tr pseudosortie 0) in
  let texte = ref (acces_tr dicorev !code_prec) in
  for i = 1 to pseudosortie.taille - 1 do
    let code = acces_tr pseudosortie i in
    if code < dicorev.taille then
    begin
      let decodage = acces_tr dicorev code in
      append_tr dicorev ((acces_tr dicorev !code_prec) ^ (soc decodage.[0]));
      texte := !texte ^ decodage
    end
    else
    begin
      let nv_code0 = acces_tr dicorev !code_prec in
      let nv_code = nv_code0 ^ (soc nv_code0.[0]) in
      append_tr dicorev nv_code;
      texte := !texte ^ nv_code
    end;
    code_prec := code
  done;
  !texte;;
```

Petite remarque : on peut faire un `append_tr` sur le code précédent puis remplacer cette nouvelle entrée du tableau par elle-même suivie du premier caractère de `acces_tr dicorev code` pour ne pas avoir besoin de distinguer les cas. Il fallait y penser !

Exercice 20

Test sur le premier paragraphe de Du côté de chez Swann : taille de 1941 caractères (mesure par OCaml), compression en un tableau de 929 codes.