TD12 - Algorithmes probabilistes (éléments de réponse)

Exercice 1

Question 1. La fonction swap échange deux éléments d'un tableau.

```
void swap(int a[], int i, int j) {
  int tmp = a[i];
  a[i] = a[j];
  a[j] = tmp;
}
```

Puis on suit l'algorithme donné dans l'énoncé. On note qu'on démarre à l'indice 1, car il est inutile d'échanger le premier élément avec lui-même. Il est important, pour que soit un bon mélange, d'aller jusqu'à i inclus.

```
void knuth_shuffle(int a[], int n) {
  for (int i = 1; i < n; i++) {
    swap(a, i, rand() % (i+1));
  }
}
```

Question 2. Soient x_ℓ les valeurs initiales du tableau et y_k ses valeurs finales. Montrons que, après l'étape i, pour $0 \leqslant k, \ell \leqslant i$, on a :

$$\mathbb{P}(y_k = x_\ell) = \frac{1}{i+1}$$

On procède par récurrence sur i. C'est clair pour $i = 0$, car $k = \ell = 0$. Supposons le résultat pour $i-1$ et montrons-le pour i. Dans la suite, on note j la valeur tirée dans $[0, i]$.

- Pour $k = i$,

$$\begin{array}{rrcll} & \mathbb{P}(y_i = x_i) & = & \frac{1}{i+1} & \text{(pas d'échange)} \\ \ell < i, & \mathbb{P}(y_i = x_\ell) & = & \sum_{0 \leqslant j < i} \frac{1}{i+1} \mathbb{P}(y_j = x_l) & \\ & & = & \sum_{0 \leqslant j < i} \frac{1}{i+1} \times \frac{1}{i} & \\ & & = & \frac{1}{i+1} & \end{array}$$

- Pour $k < i$,

$$\begin{array}{rrcll} & \mathbb{P}(y_k = x_i) & = & \frac{1}{i+1} & \text{(échange)} \\ \ell < i, & \mathbb{P}(y_k = x_\ell) & = & \frac{i}{i+1} \times \mathbb{P}(y_k = x_\ell) & \\ & & = & \frac{i}{i+1} \times \frac{1}{i} & \\ & & = & \frac{1}{i+1} & \end{array}$$

Après la dernière étape, où $i = n - 1$, on a donc bien $\mathbb{P}(y_k = x_\ell) = \frac{1}{n}$, résultat attendu.

Exercice 2

Question 1. La fonction suivante répond à la question. Sa complexité est linéaire en la taille du tableau argument a.

```
let sampling k a =
  if k < 0 || k > Array.length a
  then invalid_arg "sampling";
  let r = Array.sub a 0 k in
  for i = k to Array.length a - 1 do
    let j = Random.int (i + 1) in
    if j < k then r.(j) <- a.(i)
  done;
  r
```

Question 2. On peut commencer par examiner le comportement du programme sur des cas limites. Pour $k = 1$ et $n = 2$, on initialise $r = [a_0]$ puis on effectue une itération pour $i = 1$ en tirant j dans $[0, 2[$. On a donc bien une chance sur deux de remplacer a_0 par a_1 (cas $j < 1$).

Plus généralement, montrons que tous les éléments de a ont la même probabilité d'être sélectionnés. Utilisons l'invariant de boucle proposé dans l'énoncé.

Initialement, $i = k$ et l'invariant est trivialement établi car les k premières valeurs sont sélectionnées avec probabilité 1.

Supposons à présent l'invariant établi pour $i \geqslant k$ et considérons l'itération i de l'algorithme. Soit $0 \leqslant \ell < i + 1$.

- Pour $\ell = i$, on a :
$$\mathbb{P}\left[a_i \text{ est sélectionné}\right] = \frac{k}{i+1}$$
car on a tiré j dans $[0, i]$ et on a conservé a_i si et seulement si $i < k$.
- Pour $\ell < i$, on a :
$$\begin{aligned}\mathbb{P}\left[a_\ell \text{ est sélectionné}\right] &= \mathbb{P}\left[a_\ell \text{ était sélectionné}\right] \times \mathbb{P}\left[a_\ell \text{ pas écrasé}\right]\\ &= \frac{k}{i} \times \frac{i}{i+1} \quad \text{par H.R.}\\ &= \frac{k}{i+1}\end{aligned}$$
car pour écraser a_ℓ il faut tirer exactement l'indice où se trouve a_ℓ actuellement, parmi $i + 1$ valeurs.

L'invariant est donc bien préservé pour $i + 1$. À l'issue de l'algorithme, c'est-à-dire $i = n$, on en déduit que chaque élément de a est sélectionné avec probabilité $\frac{k}{n}$, ce qui est bien le résultat attendu.

Exercice 3

Il s'agit d'appliquer le principe de l'échantillonnage dans le cas $k = 1$. On parcourt donc la liste en maintenant dans une variable `r` notre candidat et dans une variable `n` le nombre de valeurs vues jusqu'à présent. On remplace le candidat `r` par la tête de liste avec probabilité $1/$n.

```
let random_element l =
  let rec scan r n = function
    | []      -> r
    | x :: l -> scan (if Random.int n = 0 then x else r) (n + 1) l
  in
  match l with
    | []      -> invalid_arg "random_element"
    | x :: l -> scan x 2 l
```

C'est un exemple d'*algorithme en ligne*, parfois aussi appelé algorithme incrémental, qui reçoit des données en continu sans en connaître le nombre total, et qui doit prendre des décisions au fur et à mesure. Un cadre classique est celui dans lequel l'algorithme doit répondre à des requêtes les unes après les autres, sans connaître les requêtes à venir. Il s'oppose au concept d'*algorithme hors ligne* qui reçoit d'un seul coup les données qu'il a à considérer, et prend ses décisions en fonction de cette entrée.

Exercice 4

Fait en classe.

Exercice 5

Question 1. Commençons par définir une structure.

```
struct Bloom {
  int k;
  int *seeds; // tableau de taille k
  int m;
  bool *bits; // tableau de taille m
};
```

`seeds` est un tableau de `k` entiers tirés aléatoirement. Pour écrire les opérations élémentaires, on commence par définir une fonction de hachage en s'inspirant de celle fournie dans l'énoncé.

```
int bloom_hash(int seed, char *s) {
  int h = 0;
  char c;
  while ((c = *s++) != 0) {
    h = seed * h + c;
  }
  return h & INT_MAX;
}
```

On suppose ici que `bloom_hash` renvoie un résultat positif ou nul, ce qui assure que le modulo est bien dans $0..m-1$. La fonction `bloom_bit` calcule $h_i(s) \pmod m$.

```
int bloom_bit(bloom *b, int i, char *s) {
  return bloom_hash(b->seeds[i], s) % b->m;
}
```

On peut alors définir deux fonctions pour créer et supprimer un filtre.

```
bloom *bloom_create(int k, int m) {
  bloom *b = malloc(sizeof(struct Bloom));
  b->k = k;
  b->seeds = calloc(k, sizeof(int));
  for (int i = 0; i < k; i++) b->seeds[i] = rand();
  b->m = m;
  b->bits = calloc(m, sizeof(bool));
  return b;
}

void bloom_delete(bloom *b) {
  free(b->bits);
  free(b);
}
```

Et deux fonctions pour ajouter un caractère et tester la présence d'un caractère.

```
void bloom_add(bloom *b, char *s) {
  for (int i = 0; i < b->k; i++) {
    b->bits[bloom_bit(b, i, s)] = true;
  }
}

bool bloom_contains(bloom *b, char *s) {
  for (int i = 0; i < b->k; i++) {
    if (!b->bits[bloom_bit(b, i, s)])
      return false;
  }
  return true;
}
```

On pourrait avantageusement se servir d'un tableau de bits pour économiser de l'espace. En effet, un tableau C de type `bool[]` de taille n occupe n octets. On peut diminuer cet espace d'un en stockant huit éléments par octets. On peut proposer une structure de *tableau de bits* qui représente un tableau de n booléens à l'aide d'un tableau de $\lceil n/32 \rceil$ entiers 32 bits de type `uint32_t`.

```
typedef struct Bitarray {
  int size;
  uint32_t *bits;
} bitarray;
```

On peut alors implémenter les opérations de création, d'accès et de modification.

```
// W = nombre de bits par élément du tableau
#define W (8 * sizeof(uint32_t))

bitarray *bitarray_create(int size, bool b) {
  assert(size >= 0);
  bitarray *a = malloc(sizeof(struct Bitarray));
  a->size = size;
  int n = size / W;
  if (n % W > 0) n++;
  a->bits = calloc(n, sizeof(uint32_t));
  if (b)
    for (int i = 0; i < n; i++)
      a->bits[i] = ~0;
  return a;
}

int bitarray_size(bitarray *a) {
  return a->size;
}

void bitarray_set(bitarray *a, int i, bool b) {
  assert(0 <= i && i < a->size);
```

```
  int j = i / W, k = i % W;
  if (b)
    a->bits[j] |= 1 << k;
  else
    a->bits[j] &= ~(1 << k);
}

bool bitarray_get(bitarray *a, int i) {
  assert(0 <= i && i < a->size);
  int j = i / W, k = i % W;
  return (a->bits[j] & (1 << k)) != 0;
}

void bitarray_delete(bitarray *a) {
  free(a->bits);
  free(a);
}
```

En OCaml, un tableau de type `bool array` de taille n occupe $8n$ octets car chaque booléen est représenté par un mot mémoire de 64 bits. Là encore, on peut proposer une représentation plus compacte, avec l'idée ci-dessus, c'est-à-dire avec un tableau d'entiers dont les bits représentent les booléens. Il y a cependant une petite subtilité, car les entiers d'OCaml sont des entiers 63 bits (un bit est en effet réservé à l'usage du GC). Il faut donc faire de l'arithmétique modulo 63, ou s'orienter plutôt vers un tableau d'octets avec le module `Bytes`.

Question 2. Avec un dictionnaire sous la forme d'un fichier texte contenant 346 200 mots, on obtient les résultats suivants.

k	m	faux p.	% faux p.
3	300 000	20 098	$14,40$ %
5	500 000	6 663	$4,77$ %
7	1 000 000	937	$0,67$ %

Comme on le constate, la proportion de faux positifs tombe rapidement. Et pour autant un tableau de $m = 10^6$ bits n'occupe que 122 Kio si on utilise un tableau de bits (`bitarray`) comme présenté plus haut, ce qui est bien moins que les 1,5 Mio qu'utilise le fichier `/usr/share/dict/french`.