L3-INFO Algorithmique et Modélisation

Quick – 26 février 2021 – durée 1 h

Sont interdits : les documents, les ordinateurs, les téléphones (incluant smartphone, tablettes,... tout ce qui contient un dispositif électronique).
Seuls les dictionnaires papier pour les personnes de langue étrangère sont autorisés.
Il sera tenu compte de la qualité de la rédaction et de la clarté de la présentation (2 pts).
Le barème indicatif : Exercice 1 : 5 pts ($\sim 20mn$) ; Exercice 2 : 13 pts ($\sim 45mn$).

1 Tableau circulairement trié

Un tableau T de taille n d'éléments comparables est dit *circulairement trié* par ordre croissant si il existe un indice i_{min} avec $1 \leqslant i_{min} \leqslant n$ tel que

$$T[i_{min}] \leqslant T[i_{min}+1] \leqslant T[i_{min}+2] \leqslant \cdots \leqslant T[n] \leqslant T[1] \leqslant T[2] \leqslant T[i_{min}-1].$$

Par exemple, pour le tableau suivant de taille $n = 10$

1	2	3	4	5	6	7	8	9	10
23	42	99	128	1024	2	3	10	13	19

l'indice i_{min} correspondant est 6.

1. Écrire un algorithme naïf en $\mathcal{O}(n)$ opérations de comparaisons d'éléments, qui renvoie l'indice i_{min} d'un tableau de n éléments **distincts** trié circulairement.

Éléments de réponse : Il suffit de parcourir le tableau jusqu'à une rupture de l'ordre croissant. Attention au cas des bords. Un exemple d'algorithme (ici en python)

```
def recherche_imin_seq(T):
    i = 0
    while (i < len(T)-1) and (T[i] < T[i+1]):
        i = i+1
    return (i + 1) % len(T)
```

Cet algorithme parcourt le tableau, chaque élément du tableau est accédé au plus 2 fois en lecture et on effectue au plus n comparaisons (si le tableau est de taille n), donc on a un algorithme de complexité (n).
Lorsque l'on sort de la boucle l'une des deux conditions est fausse, si la 2ième est fausse alors $T[i] > T[i+1]$ (on a aussi que $T[0] < T[1] \cdots < T[i]$ et l'indice $i+1$ est l'indice du minimum, si la première est fausse alors i est l'indice de. fin de tableau et comme on sait que le tableau est circulairement trié l'indice du minimum est 0

2. Écrire un algorithme qui résout le même problème mais avec une complexité en $\mathcal{O}(\log n)$. On justifiera précisément cet algorithme (preuve et analyse de la complexité).

Éléments de réponse : Il suffit d'utiliser le principe de la recherche dichotomique. En comparant avec le milieu de la plage considérée avec une des extrémités de la plage on sait si imin est à gauche ou à droite du milieu

```
def recherche_imin_iter(T):
    i = 0
    j = len(T)-1
    while j-i > 1:
        milieu = (i + j) // 2
        if (T[milieu] < T[j]):
            j = milieu
        else :
            i = milieu
    return i if T[i] < T[j] else j
```

```
def recherche_imin_rec(T,i=0,j=None):
    j = len(T)-1 if j is None else j
    if  (j - i) <= 1  :
        return i if T[i] < T[j] else j
    else:
        milieu = (i + j) // 2
        if (T[milieu] < T[j]):
            return recherche_imin(T,i,milieu)
        else :
            return recherche_imin(T,milieu,j)
```

La preuve se fait en 2 étapes :
- invariant : la rupture est toujours contenue dans l'intervalle $[i, j]$
- variant $j - i$ est strictement décroissant et on s'arrête à 1 ou 0

et on met ce qui va bien autour : pré-conditions, post conditions .... çà marche

pour le coût de l'algorithme, les bornes sont un peu pénibles, mais on est de l'ordre de $\log n$, c'est presque la même chose que la dichotomie.

$$C(n) = C\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathcal{O}(1)$$

3. Démontrer que cet algorithme est optimal en nombre de comparaisons d'éléments.

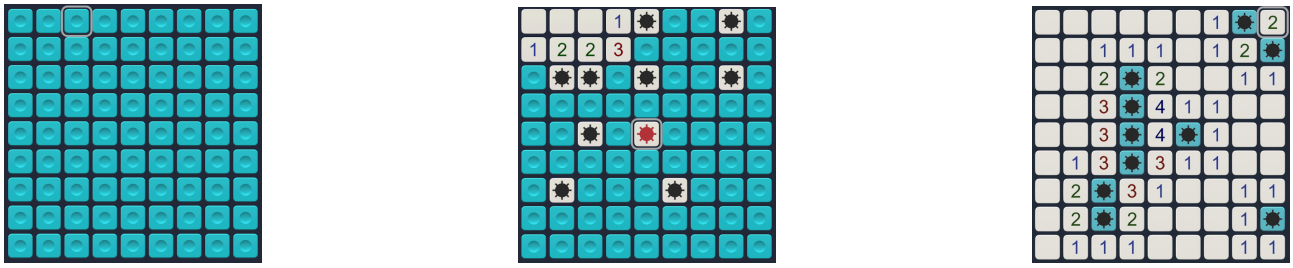
Éléments de réponse : Éléments de réflexion :
- le problème est d'identifier un élément ($imin$) parmi $1, \cdots, n$, ici l'élément est unique, car les éléments sont distincts.
- $imin$ peut prendre toutes les valeurs entre 1 et n, pour tout k il existe une donnée telle que $imin = k$).
- la seule opération pour pouvoir identifier $imin$ est la comparaison de 2 éléments, cette opération fournit une réponse binaire Vrai/Faux (0 ou 1)
- deux arguments possibles pour conclure : construction d'un arbre de décision associé à l'algorithme ou argument sur la quantité d'information fournie par une comparaison

avec k bits je ne peux identifier que 2^k éléments différents (feuilles de l'arbre de décision). Ainsi, pour identifier un élément parmi n j'aurai donc besoin d'au moins $\log_2 n$ comparaisons.

Donc on obtient une borne inférieure sur la complexité du problème en $\Omega(\log n)$. Comme il existe un algorithme de complexité $\mathcal{O}(\log n)$ on en déduit que la complexité du problème est en $\Theta(\log n)$.

2 Démineur, des bombes dans une grille

Dans le jeu du démineur, l'ordinateur propose au joueur une grille de $N = n \times n$ cases, k cases, cachées initialement au joueur contiennent des bombes. Celui-ci doit découvrir les bombes en tentant sa chance sur différentes cases...



(a) Grille initiale (b) Partie perdue (c) Positions des mines

FIGURE 1: Grille $N = 9 \times 9$ avec $k = 10$ mines

Pour initier le jeu l'ordinateur doit générer une grille $N = n \times n$ avec exactement k bombes cachées. Pour que le joueur humain ne soit pas avantagé (que la grille soit imprédictible) on souhaite construire une grille aléatoire et que toutes les grilles possibles aient la même probabilité d'apparition.

On dispose d'une fonction `random()` qui génère un nombre réel dans l'intervalle $[0, 1[$, on suppose que les appels successifs à `random()` sont modélisés par une suite de variables aléatoires indépendantes de même loi uniforme sur $[0, 1[$.

1. En une phrase que fait la fonction `piece` définie ci-dessous :

```
Fonction piece (p)
  Données : p un nombre réel 0 ⩽ p ⩽ 1
  Résultat : un entier (booléen) ...

  if random() < p
   └ Renvoie 1
  else
   └ Renvoie 0
```

Éléments de réponse : La fonction `piece` (p) renvoie une valeur, soit 0 soit 1 on pourrait dire qu'elle simule un lancer d'une pièce de monnaie (pile est représenté par 1, face par 0), la pièce est biaisée, c'est à dire que la probabilité de tomber sur pile est p (et sur face $1-p$)
De plus, plusieurs appels à `piece` (p) correspondent à des lancers de pièce inépendants.

2. Démontrer que la fonction `piece` fait bien ce que l'on attend d'elle.

Éléments de réponse : Il faut avant toute chose se fixer des notations pour être rigoureux. On a une hypothèse sur la fonction `random` qu'il faut également expliciter et on doit spécifier clairement la propriété attendue de l'algorithme.
Pour l'hypothèse sur `random`, on suppose qu'une séquence d'appels à la fonction `random` est modélisée par une suite de variables aléatoires $U_1, U_2, \cdots, U_n, \cdots$ ces variables aléatoires sont indépendantes les unes des autres et ont la même loi uniforme sur $[0,1]$, c'est à dire

$$\mathbb{P}(U_i \in [a,b]) = b-a, \text{ pour } 0 \leqslant a \leqslant b \leqslant 1.$$

Cela correspond à ce qui est attendu des fonctions `random` que l'on retrouve dans presque tous les langages de programmation.
Pour la spécification de la fonction `piece` (p) on fixe d'abord les notations. Une séquence d'appels à la fonction `random` est modélisée par une suite de variables aléatoires $X_1, X_2, \cdots, X_n, \cdots$ (évidemment ce sont des variables aléatoires car dépendent d'appels à `random` qui est aléatoire). On veut montrer que

i Les variables X_i sont indépendantes.
En effet, comme les appels successifs à la fonction `piece` font, en interne, des appels différents à la fonction `random` et comme les variables modélisant les appels à la fonction `random` sont indépendants, les variables $X_1, X_2, \cdots, X_n, \cdots$ sont indépendantes. □

ii Les X_i ont la même loi de probabilité.
En effet, comme le code de la fonction `piece` est le même pour tous les appels lorsque le paramètre p est fixé, et que la loi des appels à `random` est la même pour tous les appels, on en déduit que les X_i ont tous la même loi. □

iii Les X_i suivent la même loi de probabilité $\mathcal{B}(p)$
En effet, X_i peut prendre uniquement la valeur 0 ou 1, et donc on calcule la probabilité que la valeur de retour soit 1, puis 1

$$\begin{aligned}
\mathbb{P}(X_i = 1) &= \text{ probabilité que la valeur de retour soit } 1 && (1)\\
&= \text{ probabilité que la condition } (\texttt{random} \leqslant p) && (2)\\
&= \mathbb{P}(U_i \leqslant p) && (3)\\
&= p
\end{aligned}$$

et on montre de la même façon que $\mathbb{P}(X_i = 0) = 1-p$. Ainsi X_i suit une loi de Bernoulli de paramètre p. □

On remarque que la génération d'une grille à $N = n \times n$ cases revient à numéroter toutes les cases de 1 à N et à construire un tableau de N bits, chaque bit indiquant s'il y a une bombe sur la case ou pas, et ayant exactement k bits à 1. On propose l'algorithme :

```
Fonction genereA (N, k)
  Données : N, k entiers, 0 ⩽ k ⩽ N
  Résultat : un tableau de N bits indicé de 1 à N
  p = k/N
  G = tableau de N bits
  for n = 1 to N
    G[i] = piece (p)
  Renvoie G
```

3. Que pensez-vous de ce générateur de grille `genereA` ? Justifier votre réponse.

Éléments de réponse : Il est clair que l'on génère bien une grille de N cases puisque on affecte une valeur 0 ou 1 à chaque case. Le problème vient de générer une grille avec exactement k cases à 1. Or, pour cet algorithme il n'y a pas de contraintes entre les cases donc potentiellement on pourrait avoir un nombre arbitraire de cases à 1. La probabilité d'observer par exemple que des 1 est égale à p^N qui est certes très très petit mais qui est strictement positif. On a

donc une probabilité non nulle d'avoir une grille qui ne répond pas à la spécification. Cependant, la loi de $S_N(p)$ nombre de cases à 1 est la loi binomiale $\mathcal{B}in(N,p)$, le nombre moyen de cases à 1 est alors $N.p = N.\frac{k}{N} = k$, donc en moyenne on a k bits à 1 dans la grille. Cela pourrait être une méthode pas tout à fait juste....

On peut également calculer la probabilité d'avoir exactement k 1 et rejeter si cela ne convient pas. La probabilité d'acceptation est donc

$$p_{accept} = \binom{n}{k} p^k (1-p)^{N-k} = \binom{n}{k} \left(\frac{k}{N}\right)^k \left(1-\frac{k}{N}\right)^{N-k}$$

Pour $N = 81$ et $k = 10$ on obtient $p_{accept} \simeq 0.13$ (valeur à vérifier), ce qui n'est pas trop petit, en exécutant en moyenne 8 fois l'algorithme on obtient une grille correcte.

À partir de la fonction `random()` on construit la fonction `alea(n)` suivante :

Fonction `alea` (N)
Données : N un entier
Résultat : un entier k ...

```
k = floor( N * random() ) +1 /* floor (x) est la partie entière inférieure
 du nombre réel x, floor (3.14) renvoie l'entier 3  */
Renvoie k
```

4. En une phrase, que fait la fonction `alea` ?

Éléments de réponse : La fonction `alea` simule un "dé" avec n face, c'est à dire génère un entier entre 1 et n et la loi est uniforme (chaque valeur a la probabilité $\frac{1}{n}$ d'être générée).

5. Démontrer que la fonction `alea` (N) fait bien ce que vous venez de spécifier.

Éléments de réponse : On fixe une notation, par exemple on note $A_1, A_2, \cdots, A_n, \cdots$ les variables aléatoires modélisant les appels successifs à la fonction `alea`.

La démarche est exactement la même que pour la pièce, les arguments d'indépendance et de même loi sont exactement les mêmes. Les variables $A_1, A_2, \cdots, A_n, \cdots$ sont indépendantes et ont toutes la même loi., à valeur dans $[1, N]$

Pour une valeur l fixée $1 \leqslant\leqslant N$

$$\begin{aligned}
\mathbb{P}(A_i = k) &= \text{ probabilité que la valeur de retour de } \texttt{alea} \text{ (N) soit } l && (4)\\
&= \text{ probabilité que la valeur de la variable } \texttt{k} \text{ soit } l && (5)\\
&= \mathbb{P}\left(\texttt{floor}(N * \texttt{random}()) + 1 = l\right) && (6)\\
&= \mathbb{P}\left(\texttt{floor}(N * \texttt{random}()) = l - 1\right) && (7)\\
&= \mathbb{P}\left(N * \texttt{random}()) \in [l-1, l[\right) && (8)\\
&= \mathbb{P}\left(\texttt{random}() \in \left[\frac{l-1}{N}, \frac{l}{N}\right[\right) && (9)\\
&= \mathbb{P}\left(U_i \in \left[\frac{l-1}{N}, \frac{l}{N}\right[\right) && (10)\\
&= \frac{l}{N} - \frac{l-1}{N} && (11)\\
&= \frac{1}{N}
\end{aligned}$$

Cette probabilité ne dépend pas de l donc toutes les valeurs l possibles ont la même probabilité d'être générée. La variable A_i suit donc une loi uniforme sur l'ensemble $\{1, 2, \cdots, N\}$. □

On utilise l'algorithme suivant pour générer une grille

```
Fonction genereB (N, k)
  Données : N, k entiers, 0 ⩽ k ⩽ N
  Résultat : un tableau de N bits ...
  bomb = 0
  G = tableau de N bits initialisé à 0
  while bomb < k
  |  i = alea (N)
  |  if G[i] ≠ 1
  |  |  G[i] = 1
  |  |  bomb = bomb + 1
  Renvoie G
```

6. Expliquer en une phrase le principe de cet algorithme (ne pas démontrer sa correction).

Éléments de réponse : La condition de sortie du **while** ,
└ a
vec le bon invariant (bomb $\leqslant k$) impose que le nombre de bombes est égal à k. De plus on a un algorithme basé sur le rejet et comme les tirages sont de loi uniforme sur la grille on a l'intuition qu'au final toutes les grilles auront la même probabilité d'être générées

7. Donner une expression de son coût moyen (on pourra s'inspirer du problème du "Coupon collector" pour obtenir au moins un majorant).

Éléments de réponse : On va exprimer le coût moyen en fonction de la valeur de k et on le note $C_N(k)$.
Lorsque $k-1$ bombes sont déjà placées (il y a alors $N-k+1$ cases vide) la probabilité de tirer une case vide est $\frac{N-k+1}{N}$. Le nombre de tentatives pour tirer une case vide suit donc une loi géométrique de paramètre $\frac{N-k+1}{N}$. Il faut donc en moyenne $\frac{N}{N-k+1}$ tentatives.
D'où l'équation de récurrence

$$
\begin{aligned}
C_N(k) &= C_N(k-1) + \frac{N}{N-k+1} && (12)\\
&= \frac{N}{N} + \frac{N}{N-1} + \frac{N}{N-1} + \cdots + \frac{N}{N-k+1} && (13)\\
&= N\left(\frac{1}{N} + \frac{1}{N-1} + \cdots + \frac{1}{N-k+1}\right) && (14)\\
&= N\left[\left(\frac{1}{N} + \frac{1}{N-1} + \cdots + \frac{1}{2} + \frac{1}{1}\right) - \left(\frac{1}{N-k} + \frac{1}{N-k-1} + \cdots + \frac{1}{1}\right)\right] && (15)\\
&\simeq N(\log N - \log(N-k)) = N \log \frac{N}{N-k} = -N \log(1 - \frac{k}{N})
\end{aligned}
$$

8. Que se passe-t'il lorsque k est petit ? lorsque k est proche de N ? Proposer une optimisation de cet algorithme.

Éléments de réponse : Lorsque k est petit, "on gagne presque à tous les coups", et on un coût de l'ordre de k, le développement limité (eh oui, ils peuvent servir,) nous dit que pour $\frac{k}{N}$ petit $\log(1-\frac{k}{N}) \simeq -\frac{k}{N}$ et on en déduit l'ordre de grandeur pour k petit.
Lorsque k est grand, proche de N "on aura du mal sur la fin" et le coût sera proche de $N\log(N)$, ce qui est sans doute trop.
Lorsque k est plus grand que $\frac{N}{2}$ plus de bombes que de cases vides, on a intérêt à partir d'une grille remplie de bombes et de retirer des bombes avec la probabilité $1-p$ qui est plus petite que $\frac{1}{2}$, le coût dans ce cas est majoré par $N\log 2 \simeq 0.7N$. On est sous les N tirages aléatoires.

9. Dans le jeu du démineur, combien de grilles différentes sont-elles possibles en fonction de k et N ?

Éléments de réponse : C'est le nombre de façon de choisir k éléments parmi N, c'est à dire

$$\binom{N}{k}$$

10. On souhaite enregistrer au fur et à mesure les différentes grilles générées et ne proposer que de nouvelles grilles. Proposer une structure de donnée permettant de stocker les grilles et de vérifier rapidement qu'une configuration a déjà été utilisée. Justifier votre réponse.

Éléments de réponse : Comme le nombre de grilles est très grand, on ne peut pas avoir un adressage direct. On pourrait proposer de stocker les grilles déjà faites dans une table de hachage, la fonction de hachage doit prendre en compte l'ensemble des bits de la grille, on pourrait calculer le nombre modulo un nombre premier assez grand.