Laborator 3 - Algoritmi aleatori

I. Generarea numerelor aleatoare

În R există o vastă colecţie de funcţii dedicate generării de obiecte aleatoare (numere, permutări etc).

Generarea numerelor aleatoare întregi. Pentru generarea numerelor aleatoare întregi se poate utiliza funcţia *sample*(); de exemplu pentru generarea unui număr aleator întreg între 1 şi 300 sau a cinci numere aleatoare întregi cuprinse între 200 şi 250:

```
> x = sample(300,1)
> x
> [1] 68
> x = sample(200:250,5)
> x
> [1] 182 36 234 204 170 286
```

Putem genera numere aleatoare întregi cu repetiţie (implicit *replace* este FALSE):

```
> x = sample(30, 6, replace=T)
> x
> [1] 8 25 25 29 8 2
> x = sample(20:40, 5, replace=T)
> x
> [1] 20 30 37 24 30
```

Funcţia*sample*() poate genera un eşantion format din elemente ale unui vector indicat:

```
> x = c(2.1, 3.2, 2.3, 2.5, 3.1, 2.9, 2.6, 2.2, 3.3)
> sample(x, 5)
> [1] 3.1 2.2 3.2 2.9 3.2
> sample(x, 5, replace=T)
> [1] 3.1 2.2 3.2 2.9 3.2
```

Poate fi folosită şi pentru a genera permutări aleatoare:

```
> sample(10)
> [1] 3 2 5 7 8 10 6 9 1 4
> [sample(15)
> [1] 9 6 7 4 11 15 10 3 12 2 1 13 8 5 14
```

Puteţi trece în revistă şi alte funcţii similare, cum ar fi *shuffle(), din pachetul permute.*

Generarea de numere aleatoare reale. Pentru generarea de numere reale aleatoare uniforme dintr-un interval dat se foloseşte funcţia *runif()*; de exemplu pentru a genera k numere aleatoare uniforme cuprinse între a şi b se foloseşte *runif(k, a, b)*:

```
> runif(10,2, 4.5) # zece valori aleatoare uniforme dintre 2 şi 4.5
> [1] 3.802909 3.072721 3.615275 2.378011 3.281129 4.269154
> [7] 2.611120 4.297596 2.418020 2.536075
> x = runif(4, 0, 1) # patru numere aleatoare uniforme dintre 0 şi 1
> x
> [1] 0.9979809 0.6081421 0.4032731 0.8214655
```

II. Algoritmi aleatori.

Exerciţiu rezolvat. (Monte Carlo) Scrieţi o funcţie care să implementeze algoritmul aleator de verificare a înmulţirii a două matrici. Funcţia va avea ca parametri matricile A, B şi C. Folosind această funcţie scrieţi o altă funcţie care utilizând metoda amplificării reduce probabilitatea de a greşi a algoritmului sub 2^{-k}.

Soluţie. Funcţia *matrix(data, nrow, ncol, byrow)* crează o matrice (*data* este lista elementelor pe linie, *byrow* este implicit FALSE):

```
> x = c(1, 3, 1, 4, 12, 7)
> M = matrix(x, 3, 2)# creates a matrix 3x2
> N= matrix(x, 2, 3)# creates a matrix 2x3
```

Următoarea funcţie verifică dacă $ABr = Cr$ pentru un vector uniform generat, r, de componente 0 sau 1.

```
matrix_product = function(A, B, C) {
  n = nrow(A);
  r = matrix( , nrow = n, ncol = 1);
  x = matrix( , nrow = n, ncol = 1);
  y = matrix( , nrow = n, ncol = 1);
  r = sample(0:1, n, replace = TRUE);
  for(i in 1:n) {# x = Br
    x[i] = 0;
    for(j in 1:n)
      x[i] = (x[i]+ B[i,j]*r[j])%%2;
  }
  for(i in 1:n) {# y = Ax = ABr
    y[i] = 0;
    for(j in 1:n)
      y[i] = (y[i]+ B[i,j]*x[j])%%2;
  }
  for(i in 1:n) {# x = Cr
    x[i] = 0;
    for(j in 1:n)
      x[i] = (x[i]+ C[i,j]*r[j])%%2;
  }
  for(i in 1:n) {# verify if ABr==Cr
    if(y[i] !=x[i])
      return(FALSE);
  }
  return(TRUE);
}
```

Funcţia de mai jos dă o probabilitate a erorii sub 2^{-k}.

```
matrix_product_reduce = function(A, B, C, k) {
  for(i in 1:k)
    if(!matrix_product(A, B, C))
      return(FALSE);
  }
  return(TRUE);
}
```

Exerciţiu rezolvat. (Las Vegas.) Scrieţi o funcţie care evaluează un arbore (”game tree”) de adâncime $2h$; arborele este dat ca un vector care conţine cele 4^h valori din frunze.

Soluţie. Intrarea este un vector de dimensiune of 4^h ca acesta de mai jos (pentru $h = 2$)

```
> leaves = c(0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0)
```

Se observă că un nod MAX se află pe un nivel impar, iar un nod MIN pe un nivel par. Astfel un nod i satisface

$$[\log_2(i)] \equiv \begin{cases} 1\,(mod\,2), & \text{dacă } i \text{ este MAX} \\ 0\,(mod\,2), & \text{dacă } i \text{ este MIN} \end{cases}$$

Vom evalua fiecare nod cu excepţia frunzelor; un nod de pe penultimul sau ultimul nivel, i, este caracterizată de următoarea inegalitate: $\log_2(i) \geqslant (2h-1)$.

Nodurile vor fi numerotate de la rădăcină (nodul 1) spre frunze şi de la stânga la dreapta. Funcţia de mai jos evaluează recursiv un nod i care nu este o frunză (funcţia trebuie completată cu un *if* corespunzând nodurilor MAX):

```
tree_eval = function(i, leaves) {
   a = runif(1, 0, 1); len = length(leaves);
   if(log(i,2) >= log(len,2) - 1) { # copiii nodului i sunt frunze
      if(a <= 0.5) {
         if(leaves[2*i - len + 1] == 0)
            return(leaves[2*i +1 -len + 1];
         return(1);}
      else {
         if(leaves[2*i + 1 -len + 1] == 0)
            return(leaves[2*i +1 -len + 1];
         return(1);}
   }
   if((floor(log(i,2))%% 2 == 0)){ # nodul i este de tip MIN
      if(a <= 0.5) {
         if(tree_eval (2*i, leaves) == 1)
            return(tree_eval (2*i + 1, leaves));
         return(0);
      else {
         if(tree_eval (2*i +1, leaves) == 1)
            return(tree_eval (2*i, leaves));
         return(0);
   }
   ......
}
```

După descrierea completă a funcţiei ea poate fi folosită astfel

```
game_tree_eval = function(leaves) {
   return(tree_eval (1, leaves));
}
```

Verificaţi rezultatul calculând direct valoarea arborelui.

Exerciţii propuse.

1. Scrieţi o funcţie care simulează o variabilă aleatoare finită

$$X : \begin{pmatrix} x_1 & x_2 & \dots & x_k \\ p_1 & p_2 & \dots & p_k \end{pmatrix}$$

Temă pentru acasă.

14 puncte [7p: B1 sau B3] + [7p: B2 sau B4]

B1. (Las Vegas.) Scrieţi o funcţie care implementează algoritmul aleator Quick Sort. Funcţia va avea mulţimnea S ca singur parametru (dat ca vector).

B2. (Monte Carlo.) Scrieţi o funcţie care să implementeze algoritmul lui Karger.
Hint. Presupunem că graful este dat cu matrice de adiacenţă. De exemplu:

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

B3. (Las Vegas.) Implementaţi algoritmul aleator pentru determinarea medianei.

B4. (Monte Carlo.) Un *BPP* este un arbore binar în care fiecare nod x este etichetat cu o regiune a planului $R(x)$, în plus, un nod interior mai are o etichetă: o dreaptă $L(x)$. Pentru un nod interior, x, dreapta $L(x)$ împarte regiunea asociată, $R(x)$, în două sub-regiuni, $R_1(x)$ şi $R_2(x)$, care sunt regiunile din plan asociate copiilor săi. În acest fel o regiune asociată unui nod este delimitată de toate dreptele care se găsesc înodurile de pe drumul către rădăcină. Regiunile asociate frunzelor formează partiţia arborelui BPP.

Să presupunem că avem o familie de segmente din plan care nu se intersectează $S = \{s_1, s_2, \ldots, s_n\}$. Dorim să determinăm un BPP astfel ca fiecare regiune din partiţia BPP-ului să conţină cel mult un segment sau o porţiune a unui segment din S.

Pentru un segment s_i, dreapta sa suport, $l(s_i)$, se obţine extinzând s_i la ambele capete către infinit.

RandomPartition(S) **{**
    alege π *o permutare a mulţimii* $\{1, 2 \ldots, n\}$ *uniform şi aleator*;
    while($\exists R$ *regine care conţine mai mult de un segment*) **{**
        alege $i = \min\{\pi(j) \ : \ s_{\pi(j)} \in R, j = \overline{1, n}\}$;
        taie R *cu* $l(s_i)$;
}

Scrieţi o funcţie care implementează algoritmul RandomPartition de mai sus. Funcţia va avea mulţimea S drept unic parametru (dat ca vector). (În rădăcină regiunea poate fi planul întreg.)

Rezolvările acestor exerciţii (funcţiile R şi apelurile lor) vor fi redactate într-un script R.