Problema colecționarului de cupoane și algoritmul Quicksort

Obiectivul acestui laborator este de prezenta problema Colecționarului de cupoane și algoritmul randomizat QuickSort și de a determina timpul mediu de execuție a acestuia.

1 Problema colecționarului de cupoane

Să presupunem că fiecare cutie de cereale conține unul dintre cele n cupoane diferite existente. Odată ce o persoană a colecționat toate cele n cupoane poate să le trimită pentru a revendica un premiu. De asemenea, presupunem că fiecare cupon este ales uniform și independent din cele n posibilități existente și colecționarul nu colaborează cu alte persoane pentru a completa colecția. Întrebarea care se pune este câte cutii de cereale trebuie cumpărate, în medie, pentru a obține cel puțin unul din fiecare cupon?

Pentru a rezolva această problemă, să notăm cu X variabila aleatoare care descrie numărul de cutii de cereale cumpărate până când obținem toate cupoanele. Vrem să determinăm $\mathbb{E}[X]$.

Să notăm cu X_i numărul suplimentar de cutii de cereale pe care trebuie să le cumpărăm atunci când avem $i-1$ cupoane colecționate pentru a avea i cupoane diferite, $i \geq 1$. Astfel putem scrie

$$X = \sum_{i=1}^{n} X_i,$$

de unde $\mathbb{E}[X] = \sum_{i=1}^{n} \mathbb{E}[X_i]$. Rămâne să determinăm $\mathbb{E}[X_i]$ pentru $i \in \{1, 2, \ldots, n\}$.

Să remarcăm faptul că atunci când avem colectate $i-1$ cupoane ne mai rămân $n-i+1$ cupoane de colecționat, prin urmare probabilitatea de a alege un nou cupon este $p_i = \frac{n-i+1}{n}$ și cum $X_i \sim \text{Geom}(p_i)$ deducem că $\mathbb{E}[X_i] = \frac{1}{p_i}$.

Avem

$$\mathbb{E}[X] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} \frac{1}{p_i} = \sum_{i=1}^{n} \frac{n}{n-i+1} = n \sum_{i=1}^{n} \frac{1}{i} = nH_n$$

ceea ce implică $\mathbb{E}[X] = n\left(\log(n) + \mathcal{O}(1)\right)$ deoarece $H_n = \log(n) + O(1)$[1]).

Următorul cod R simulează problema colecționarului de cupoane:

```
simcollect = function(n) {

coupons = 1:n # multimea cupoanelor
collect = numeric(n)

nums = 0
```

[1] A se vedea pagina de wikipedia armonic_number

```
while (sum(collect)<n)
{
  # extragere cu intoarcere
    i = sample(coupons, 1)
    collect[i] = 1
    nums = nums + 1
}
return(nums)
}
```

Pentru calculul mediei vom considera $n = 20$ și vom repeta procedeul $N = 10000$ de ori. Vom compara rezultatul empiric cu cel teoretic:

```
## Calculul mediei
n = 20
Nrep = 10000
simlist = replicate(Nrep, simcollect(n))

# media empirica
mean(simlist)
[1] 71.8905

# media teoretica
n*sum(1/(1:n))
[1] 71.95479
```

Pentru calculul variației avem că $Var(X_i) = \frac{1-p_i}{p_i^2}$ și cum X_i sunt independente rezultă că

$$Var(X) = \sum_{i=1}^{n} Var(X_i) = \sum_{i=1}^{n} \frac{1-p_i}{p_i} = n \sum_{i=1}^{n} \frac{i-1}{(n-i+1)^2}.$$

Pentru compararea rezultatului empiric cu cel teoretic avem:

```
# varianta empirica
var(simlist)
[1] 559.212

# varianta teoretica
n*sum((0:(n-1))/((n:1)^2))
[1] 566.5105
```

O aplicație a problemei colecționarului de cupoane este următoarea: să presupunem că într-un parc național din India, o cameră video automată fotografiază n tigrii care trec prin dreptul ei pe parcursul unui an și analizând fotografiile se constată că t dintre aceștia sunt diferiți. Ne propunem să estimăm (aproximăm) numărul total de tigrii din parc. În contextul problemei colecționarului de cupoane, presupunem că avem n cupoane diferite și că am cumpărat t cutii de cereale și ne întrebăm care este numărul mediu de cupoane distincte din cele t.

Fie Y variabila aleatoare care ne dă numărul de cupoane distincte după t cutii cumpărate. Atunci putem scrie

$$Y = I_1 + I_2 + \cdots + I_n$$

unde pentru $j \in \{1, 2, \ldots, n\}$

$$I_j = \begin{cases} 1, \text{cuponul j se află printre cele t} \\ 0, \text{altfel} \end{cases}$$

Astfel,

$$\mathbb{E}[I_j] = \mathbb{P}(\text{cuponul j se află printre cele t}) = 1 - \mathbb{P}(\text{cuponul j nu se află printre cele t}) = 1 - \left(1 - \frac{1}{n}\right)^t$$

ceea ce conduce la

$$\mathbb{E}[Y] = \sum_{j=1}^{n} \mathbb{E}[I_j] = n\left[1 - \left(1 - \frac{1}{n}\right)^t\right].$$

Pentru a determina numărul de tigrii din parc trebuie să găsim valoarea lui n știind numărul t de tigrii fotografiați de camera automată și de numărul mediu de tigrii distincți d determinați manual, deci trebuie să rezolvăm ecuația:

$$d \approx \mathbb{E}[Y] = n\left[1 - \left(1 - \frac{1}{n}\right)^t\right]$$

Ecuația nu are o soluție explicită dar poate fi determinată numeric folosind funcția `uniroot` care permite găsirea unei rădăcini într-un interval dat a unei funcții reale:

```
f = function(n, t = 100, d = 50){
  n*(1 - (1 - 1/n)^t) - d
}

# pentru 100 de tigrii fotografiati dintre care 50 diferiti
ans = uniroot(f, c(50, 200), t = 100, d = 50)
ans$root
[1] 62.40844
```

Pentru $t = 100$ și $d = 50$ estimăm în jur de 62 tigrii in parc.

2 Timpul mediu de execuție al algoritmului Quicksort

În practică, algoritmul *Quicksort* este unul din cei mai rapizi și mai populari algoritmi de sortare. Unul dintre motivele acestui fapt este că algoritmul nu necesită memorie de stocare suplimentară. Cu toate acestea, algoritmul *Quicksort* este un algoritm destul de slab atunci când luăm în calcul scenariile teoretice de tipul *cel mai rău caz*. Vom vedea, în cele ce urmează, că versiunea randomizată a acestui algoritm are, în medie, o performanță foarte bună.

Algoritmul primește ca input o listă $S = \{x_1, \ldots, x_n\}$ de n numere care, pentru ușurință, vor fi presupuse diferite. Pseudocodul algoritmului este:

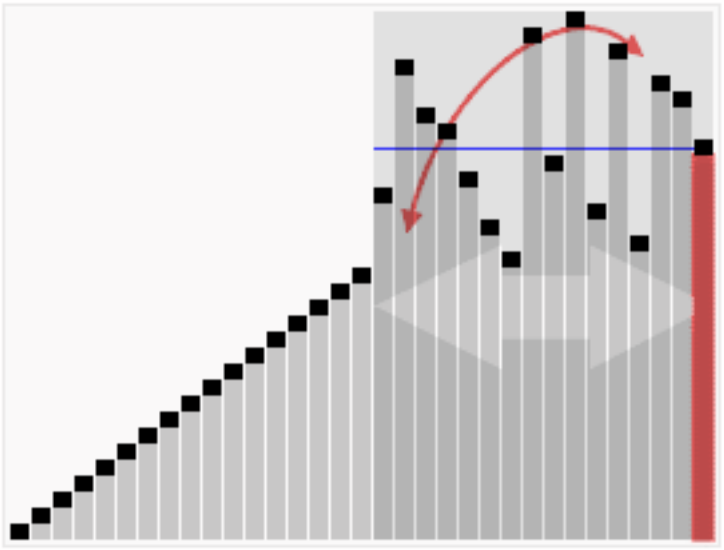
Input: O listă $S = \{x_1, \ldots, x_n\}$ de n elemente distincte pe o mulțime total ordonată.

Output: Elementele listei S sortate crescător.

1. Dacă S nu are niciun element sau are un element atunci întoarce S. Altfel continuă.
2. Alege un element din S ca pivot; să-l numim x

3. Compară toate elementele din S cu x pentru a împărți lista în două subliste: a) S_1 conține toate elementele din S mai mici decât x. b) S_2 conține toate elementele din S mai mari decât x.
4. Folosește recursiv Quicksort pentru a sorta crescător sublistele S_1 și S_2.
5. Întoarce lista $\{S_1, x, S_2\}$

Următoarea animație este ilustrativă (doar în versiunea HTML):



Să observăm că există situații (de tipul *cel mai rău caz*) în care algoritmul Quicksort necesită $\Omega(n)$[2] operații de comparare. De exemplu să presupunem că lista de input este $S = \{x_1 = n, x_2 = n-1, \ldots, x_n = 1\}$ și să presupunem că pentru alegerea pivotului adoptăm regula ca acesta să fie primul element din listă. Prin urmare primul pivot ales este n și algoritmul necesită $n-1$ comparații. În urma diviziunii, rezultă două subliste, una de lungime 0 (care nu necesită nicio operație suplimentară) și una de lungime $n-1$ (ce elementele $n-1, n-2, \ldots, 1$). La pasul doi, următorul pivot ales este $n-1$ iar algoritmul necesită $n-2$ comparații și întoarce sublista cu elemente $n-2, \ldots, 1$. Continuând procedeul deducem că algoritmul Quicksort efectuează

$$(n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2} \quad \text{operații.}$$

Din exemplul de mai sus, este clar că alegerea pivotului influențează puternic numărul de operații pe care le efectuează algoritmul. O alegere mai bună a pivotului ar consta în determinarea unui element, la fiecare pas, care să împartă lista în două subliste cam de aceeași mărime ($\lceil n/2 \rceil$ elemente).

Întrebarea care se pune este cum putem garanta că algoritmul alege un pivot bun suficient de des ? O modalitate ar fi să alegem pivotul aleator, de manieră uniformă între elementele disponibile. Această abordare face ca algoritmul Quicksort să devină randomizat.

Să presupunem că ori de câte ori un pivot este ales pentru algoritmul *Quicksort randomizat*, acesta este ales independent și uniform din mulțimea elementelor posibile. Arătați că numărul mediu de comparări ale algoritmului este de $2n\log(n) + O(n)$. Scrieți o funcție care implementează algoritmul *Quicksort randomizat* cu pivot ales uniform.

[2] A se vedea pagina de wikipedia Big_O_notation

Fie $y_1, \ldots, y_n$ elementele $x_1, \ldots, x_n$ ordonate crescător. Pentru $i < j$, fie X_{ij} variabila aleatoare care ia valoarea 1 dacă elementele y_i și y_j au fost comparate pe parcursul rulării algoritmului și valoarea 0 altfel. Atunci numărul total de comparări X satisface relația

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

și din proprietatea de liniaritate a mediei

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}].$$

Cum X_{ij} este o variabilă aleatoare de tip Bernoulli care ia doar valoarea 0 și 1, $\mathbb{E}[X_{ij}] = \mathbb{P}(X_{ij} = 1)$ prin urmare trebuie să determinăm probabilitatea ca elementele y_i și y_j să fie comparate pe parcursul algoritmului. Să observăm că elementele y_i și y_j sunt comparate dacă și numai dacă oricare dintre cele două elemente sunt alese ca pivot din mulțimea $A_{ij} = \{y_i, y_{i+1}, \ldots, y_j\}$. Acest lucru se datorează faptului că dacă y_i (sau y_j) a fost primul pivot ales din mulțimea A_{ij} atunci elementele y_i și y_j rămân în aceeași sublistă, deci vor fi comparate ulterior. În mod similar, dacă niciunul din elementele y_i și y_j nu este primul pivot ales din mulțimea A_{ij} atunci cele două elemente vor face parte din subliste separate și nu vor mai fi comparate.

Cum pivoții sunt aleși de manieră independentă și uniform din fiecare sublistă de elemente, prima dată când un pivot este ales din mulțimea A_{ij} acesta are aceeași șansă să fie oricare element. Prin urmare probabilitatea ca y_i sau y_j să fie primul pivot ales este $\frac{2}{j-i+1}$. Astfel obținem

$$\begin{aligned}
\mathbb{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} = \sum_{k=2}^{n} \sum_{i=1}^{n+1-k} \frac{2}{k} \\
&= \sum_{k=2}^{n} (n+1-k)\frac{2}{k} = (2n+2)\sum_{k=1}^{n} \frac{1}{k} - 4n.
\end{aligned}$$

Prin urmare $\mathbb{E}[X] = 2(n+1)H_n - 4n = 2n\log(n) + O(n)$ (am folosit faptul că $H_n = \log(n) + O(1)$.

Următorul cod implementează algoritmul *Quicksort randomizat*:

```
quickSort <- function(vect) {
  # Args:
  #  vect: Vector numeric

  # daca lungimea este <= 1 stop
  if (length(vect) <= 1) {
    return(vect)
  }

  # alege pivotul
  ide = sample(1:length(vect),1)
  element = vect[ide]
  partition = vect[-ide]

  # Imparte elementele in doua subliste (< pivot si >= pivot)
```

```
  v1 = partition[partition < element]
  v2 = partition[partition >= element]

  # Aplica recursiv algoritmul
  v1 = quickSort(v1)
  v2 = quickSort(v2)
  return(c(v1, element, v2))
}

n = 25
S = sample(1:n, n, replace = FALSE)
# lista neordonata
S
 [1]  9 23  7  4  3 13 17 10 15 24 12 19  2 20  5 22  8  1 21  6 18 11 14 16 25
# lista ordonata
quickSort(S)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

Numărul mediu de comparații pe care le efectuează algoritmul *Quicksort randomizat*, versiunea empirică versus cea teoretică de mai sus, este ilustrat în figura următoare:

```
countQuickSort <- function(vect) {
  # Args:
  #  vect: Vector numeric

  # daca lungimea este <= 1 stop
  if (length(vect) <= 1) {
    return(vect)
  }

  count <<- count + length(vect) - 1 # imi intoarce numarul de comaparatii efectuate

  # alege pivotul
  ide = sample(1:length(vect),1)
  element = vect[ide]
  partition = vect[-ide]

  # Imparte elementele in doua subliste (< pivot si >= pivot)
  v1 = partition[partition < element]
  v2 = partition[partition >= element]

  # Aplica recursiv algoritmul
  v1 = countQuickSort(v1)
  v2 = countQuickSort(v2)
  return(c(v1, element, v2))
}

N = 1000
y = rep(0, N)

for (i in 1:N){
  S = sample(1:i, i, replace = FALSE)

  count = 0
```

```
  S_sort = countQuickSort(S)

  y[i] = count
}

# Functia care calculeaza numarul de operatii teoretice
T_n = function(n){
  return(2*(n+1)*sum(1/(1:n))-4*n)
}

theo_T = sapply(1:N, function(x){T_n(x)})

# Graficul
plot(1:N, y, type = "l",
     col = "grey80",
     bty = "n",
     main = "Algoritmul Quicksort",
     xlab = "Numar de elemente de comparat",
     ylab = "Numar de comparatii",
     lty = 3,
     lwd = 0.5)
points(1:N, y,
       col = "royalblue",
       pch = 16,
       cex = 0.6)
lines(1:N, theo_T, col = "brown3", lwd = 3, lty = 2)
legend('bottomright',
       legend = c("Empiric", "Teoretic"),
       fill = c("royalblue", "brown3"),
       bty = "n")
```