Probabilistički algoritmi

Podsećanje

Karakteristika determinističkih algoritama je da za isti ulaz i isti način izvršavanja, program daje isti izlaz. Kod probabilističkih algoritama, postoje koraci koji ne zavise isključivo od ulaza, već i od nekih slučajnih događaja.

Probabilistički algoritmi
- Algoritmi koje čine slučajne odluke tokom svog izvršenja
- Primer: Quicksort s nasumičnim pivotom

Primena probabilističkih algoritama?
- Mnogi NP-teški problemi mogu biti lako rešivi za "karakteristične" ulazne primere
- Jedan pristup je koristiti heurističko rešavanje za određene ulazne test primere
- Drugi pristup je koristiti probabilistički metod (obrade ulaznih primera, ili probabilističke korake u algoritmu) kako bi se smanjila mogućnost česte obrade najgoreg slučaja

Monte Karlo algoritam: Verovatnoća dobijanja negativnog rezultata je jako mala, ali je vreme izvršavanja ovog tipa algoritma bolje od najboljeg determinističkog algoritma.

Las Vegas algoritam: Ovaj tip algoritama nikad ne daje pogrešan rezultat, ali mu vreme izvršavanja nije garantovano. On je koristan ako mu je očekivano vreme izvršavanja malo.

Primer Monte Karlo i Las Vegas algoritma koji rešavaju isti problem

PROBLEM: Dat je niz sa $n \geq 2$ elemenata u kom je polovina članova niza jednaka slovu *a,* polovina članova niza jednaka slovu *b*. Za takav niz nađi član jednak slovu *a*.

Las Vegas algoritam:

```
NadjiA_LV(niz A, n)
begin
   repeat
      Nasumično izaberite jedan od n elemenata niza A.
   until 'a' je nadjen
end
```

Verovatnoća da će algoritam dati korektan odgovor je 1. Ali, očekivano vreme izvršavanja je $\Theta(n)$.

Monte Carlo algoritam:

```
NadjiA_MC(niz A, n, k)
begin
   i=0
   repeat
      Nasumično izaberite jedan od n elemenata niza A.
      i = i + 1
   until i=k ILI 'a' je nadjen
end
```

Ako je elementa '*a*' nadjen, onda je algoritam uspešan i dao je korektan odgovor, a inače ne dobija se korektan odgovor. Nakon *k* iteracija, verovatnoća nalaženja elementa jednaka slovu '*a*' je: $p=1-(1/2)^k$, , što čak za relativno malo k daje izuzetno malu verovatnoću greške, i to nevezano za veličinu ulaza.

Ovaj algoritam, dakle ne garantuje uspeh (jer p<=1), ali je vreme izvršavanje fiksirano.
Izbor elemenata niza se najviše izvršava *k* puta, te otuda vreme izvršavanja je $\Theta(1)$.

Dakle, možemo da zaključimo:

Monte Karlo algoritam: Verovatnoća dobijanja negativnog rezultata je jako mala, ali je vreme izvršavanja ovog tipa algoritma bolje od najboljeg determinističkog algoritma.

Las Vegas algoritam: Ovaj tip algoritama nikad ne daje pogrešan rezultat, ali mu vreme izvršavanja nije garantovano. On je koristan ako mu je očekivano vreme izvršavanja malo.

Jasno je da Monte Carlo algoritam se može pretvoriti u Las Vegas algoritam pokretanjem Monte Carlo algoritma više puta dok se ne dobije tačan odgovor.

T1. Probabilistički algoritmi: nalaženje elementa iz gornje polovine

Klasičan algoritam za pronalaženje elementa iz gornje polovine niza je linearne složenosti. Ako niz ima n elemenata, prolazi se kroz prvih n/2+1 članova niza, i pamti se tekući maksimum. Najveći od n/2+1 elemenata je sigurno u gornjoj polovini.

Međutim, ukoliko niz ima jako veliki broj elemenata, ovo može biti neprihvatljivo sporo. Sa druge strane, postoji probabilistički algoritam, koji skoro uvek daje tačno rešenje, a radi u konstantnom vremenu, O(1), u samo nekoliko koraka. Ovo skoro uvek podrazumeva da postoji verovatnoća da će algoritam dati netačno rešenje, ali je ta verovatnoća ne samo mala, već i kontrolisana - moguće je ograničiti je.

Ukoliko na slučajan način izaberemo element niza, verovatnoća da je on u gornjoj polovini iznosi 1/2. Ovo je naravno neprihvatljivo velika verovatnoća greške, ali ako na isti način izaberemo dva elementa, verovatnoća greške pada na 1/4 (jer je verovatnoća da je maksimum od dva slučajno izabrana elementa u gornjoj polovini jednaka 3/4).

Nakon izabranih k elemenata, verovatnoća greške je $1/2^k$, što već za relativno malo k daje izuzetno malu verovatnoću greške, i to nevezano za veličinu ulaza.

Primera radi, ako je k=30, verovatnoća greške iznosi 1/1073741824 =0,00000000093132

T2. Probabilistički algoritmi; bojenje elemenata skupa S od n elemenata sa dve boje, tako da u datih k podskupova S veličine r postoje elementi obe boje;k $\leq 2^{r-2}$

Problem: Obojiti svaki element skupa S jednom od dve boje, crvenom ili plavom, tako da svaki podskup Si sadrži bar jedan crveni i bar jedan plavi element.

Bojenje koje zadovoljava taj uslov zvaćemo ispravnim bojenjenjem. Ispostavlja se da pod navedenim uslovima ispravno bojenje uvek postoji. Probabilistički algoritam se dobija prepravkom probabilističkog dokaza postojanja takvog bojenja:

Obojiti svaki element S slučajnom bojom, plavom ili crvenom, nezavisno od bojenja ostalih elemenata.

Jasno je da ovakav algoritam ne daje uvek ispravno bojenje. Izračunaćemo verovatnoću neuspeha. Verovatnoća da su svi elementi Si obojeni crvenom bojom je 2^{-r}, a verovatnoća da su svi obojeni istom bojom je 2*2^{-r}

Slučajan događaj A: neki od skupova Si je neispravno obojen je unija svih slučajnih događaja Ai: skup Si je neispravno obojen, za i = 1, 2, ... , k.

P(A)=1/2

Time je dokazano da ispravno bojenje postoji (inače bi verovatnoća neispravnog bojenja bila 1). Pored toga, vidi se da je ovaj probabilistički algoritam dobar. Ispravnost zadatog bojenja se lako proverava: proveravaju se elementi svakog podskupa dok se ne nađu dva elementa koja su različito obojena. Verovatnoća uspešnog bojenja je p=1-P(A) je bar 1/2. Ako se u jednom pokušaju ne dobije ispravno bojenje, postupak se ponavlja. Očekivani broj pokušaja je manji ili jednak 2.

Opisani algoritam bojenja je **očigledno Las Vegas algoritam**, jer se bojenja proveravaju jedno za drugim, a sa traženjem se završava kada se naiđe na ispravno bojenje. Ne postoji garancija ispravnog bojenja u bilo kom fiksiranom broju pokušaja, ali je ovaj algoritam u praksi vrlo efikasan.

1. *(Probalisticka analiza algoritma i probalisticki metod resavanja problema)* Problem zapošljavanja: želimo da zaposlimo novog radnika koristeći usluge agencije za zapošljavanje. Agencija svakog dana šalje po jednog kandidata i nakon razgovora odlučujemo da li želimo da ga zaposlimo. Agenciji plaćamo malu naknadu za intervjuisanje kandidata, dok pri zapošljavanju plaćamo dosta veću naknadu - treba da otpustimo tekućeg radnika i da platimo visoku naknadu agenciji za zapošljavanje novog radnika. Želimo da u svakom trenutku imamo najbolju osobu za posao. Stoga, nakon svakog intervjua ako je novi kandidat bolje kvalifikovan od tekućeg radnika otpuštamo tekućeg radnika i zapošljavamo novog kandidata. Spremni smo da platimo cenu ove strategije, ali želimo da je procenimo.

Algoritam Zaposljavanje_radnika
Ulaz: niz od n kandidata
Izlaz: best - najbolji kandidat

```
begin
  best:=0;
  for i=1 to n
     intervjuisi kandidata(i);
     if bolji(i,best)
        best:=i;
        zaposli_kandidata(i);
    end
```

(a) Pod pretpostavkom da je cena intervjuisanja c_i, a cena zapošljavanja c_z, odrediti ukupnu cenu algoritma u najgorem slučaju.

Neka je broj zaposlenih jednak m.
Onda je ukupna cena algoritma $O(nc_i + mc_z)$.
Bez obzira koliko ljudi će se zaposliti, uvek se intervjuiše n kandidata i otuda uvek se pojavljuje cena intervjuisanja nc_i u proceni cene algoritma.

Zato ćemo se skocentrisati na cenu zapošljavanja mc_z.
Ova cena se menja svakim izvršavanjem algortma tj. proglašavanjem najboljeg kandidata.

U najgorem slučaju, zapošljava se svaki kandidat koji je intervjuisan. Na primer, ovo može da se dogodi ako se kandidati pojavljuju sortirani po kvalitetu tako da obrazuju rastući niz. Dakle, tada je cena zapošljavanja $O(nc_z)$, te je ukupna cena algoritma u najgorem slučaju $O(nc_i + nc_z)$. .

(b) Odrediti očekivani broj puta koji zapošljavamo novog radnika. (sa predavanja, CLR 5.2)

Neka X_i je slučajna promenljiva, tj. indikator pridružen događaju: *i*ti kandidat je zaposlen
Tada je $E(X_i)$= P{ *i*ti kandidat je zaposlen} (ovo sledi iz definicije indikator promenljive tj.
$E(X_i)$=1* P{ *i*ti kandidat je zaposlen} + 0* P{ *i*ti kandidat nije zaposlen}
=1/i

$$
\begin{aligned}
\mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n} X_i\right] && \text{(by equation (5.3))} \\
&= \sum_{i=1}^{n} \mathrm{E}[X_i] && \text{(by linearity of expectation)} \\
&= \sum_{i=1}^{n} 1/i && \text{(by equation (5.4))} \\
&= \ln n + O(1) && \text{(by equation (A.7))} .
\end{aligned}
$$

Zaključak1: Iako se intervjuiše n ljudi, zapravo se, u proseku, zaposli ln n.
Zaključak2: Pod ppretpostavkom da kandidati za posao dolaze u nasumičnom redosledu, algoritam Zaposljavanje_radnika ima ukupnu cenu zapošljavanja $O(c_z \ln n)$.

(c) Koja je verovatnoća samo jednog zapošljavanja?
Kako algoritam Zaposljavanje_radnika uvek zaposli kandidata sa rednim brojem 1, onda jedno zapošljavanje može da se desi AKKO nije se zaposlio niti jedan kandidat sem kandidata 1.
Ovaj događaj se javlja ako kandidat 1 je najbolji od svih n kandidata, te je ta verovatnoća $1/n$.

(d) Koja je verovatnoća zapošljavanja n puta?
Algoritam Zaposljavanje_radnika zapošljava n puta ako svaki kandidat je bolji od svih koji su prethodno bili intervjuisani (i zaposleni).

Ovaj događaj se javlja kada je lista kandidata na ulazu uređena po njihovom rangu <1, 2, . . . , *n*>, što se dešava sa verovatnoćom $1/n!$.

(e) Koja je verovatnoća zapošljavanja 2 puta?

2. Na raspolaganju nam je procedura koja za proizvoljno *k*, 1 <= *k* <= *n*, generiše slučajne brojeve iz opsega od 1 do *k* sa uniformnom raspodelom verovatnoća. Konstruisati algoritam za generisanje slučajne permutacije brojeva {1, 2,..., *n*} sa uniformnom raspodelom verovatnoća.

Rešenje:
Algoritam možemo konstruisati indukcijom po *n*.
Pretpostavimo da znamo da konstruišemo permutaciju $\{p_1,\ldots,p_{n-1}\}$ skupa {1,2,…,n-1} sa uniformnom raspodelom verovatnoća.
Da bi se dobila slučajna permutacija brojeva {1,2,…,n} treba najpre izgenerisati slučajni broj i, 1<=i<=n koga treba staviti na poslednje mesto. Zatim se na n-1 brojeva 1,2,..,**i-1,i+1,**…,n primenjuje permutacija $p_1,p_2,\ldots,p_{n-1}$
Ovo nam daje da će se sve permutacije reda n dobiti sa jednakom verovatnoćom.
Složenost algoritma je $O(n^2)$.

3. Probabilistički metod koji proverava da li je dati broj prost

Klasičan primer primene Monte Carlo algoritma je provera da li je broj n složen. Ako se odlučimo za k prirodnih brojeva $p_1, p_2, \ldots, p_k$ ($2 <= p_i <= \sqrt{n}$) i ako je n deljiv s nekim od njih, sledi da je n složen. Tako smo dobili vrlo jednostavan Monte Carlo algoritam:

Algoritam 1

```
for (i = 1,...,k)
  if (n % (random(sqrt(n))+2) == 0) {
      return n_je_slozen;
   }
return n_je_prost;
```

4. Monte Karlo probabilistički metod koji proverava da li je dati broj n prost

```
#include <stdio.h>
#include <stdlib.h>

const unsigned n = 7919; /* provera da li je broj n prost */
const unsigned k = 10;
/* broj opita u Monte Karlo algoritmu sa slucajnom osnovom a*/

/* izracunati a^t mod n; */
unsigned long bigmod(unsigned long a, unsigned long t, unsigned long n)
{ return (t == 1) ? (a % n) : (bigmod(a, t - 1, n) * (a % n)) % n;
}

char strongRandom(unsigned long n, unsigned long a)
{ unsigned long s = 1, t = n - 1, x, i;

  /* specijalni slucaj */
  if (n < 2) return 0;
  if (n == 2) return 1;

  /* korak 1) */
  while (t % 2 != 1) {
    s++;
```

```
    t /= 2;
  }
  /* korak 2) x = a^t mod n; */
  x = bigmod(a, t, n);
  if (1 == x) return 1;
  /* korak 3 */
  for (i = 0; i < s; i++) {
    if (x == n - 1) return 1;
    x = x * x % n;
  }
  return 0;
}

char isPrime(unsigned long n)
{ unsigned i;
  for (i = 1; i <= k; i++) {      /* k opita za proveru primalnosti brja n */
    long int a = rand()%(n - 3) + 2;
    if (!strongRandom(n, a)) return 0;
  }
  return 1;
}

int main(void) {
  printf("Broj %d je %s.\n", n, (isPrime(n)) ? "prost" : "slozen");
  return 0;
}
```

5. Probabilistički metod (Monte Karlo) za određivanje broja PI

1. nacin

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#define SEED 35791246

main(int argc, char* argv)
{
   int niter=0; /* broj iteracija, unosi se sa stdin */
   double x,y;
   int i,count=0;
/* count=broj tacaka u 1. kvadrantu jedinicnog kruga */
   double z;
   double pi;

   printf("Unesite broj iteracija za određivanje broja PI: ");
   scanf("%d",&niter);

   /* inicijalizacija slucajnih brojeva */
   srand(SEED);
   count=0;
   for ( i=0; i<niter; i++) {
      x = (double)rand()/RAND_MAX;
      y = (double)rand()/RAND_MAX;
      z = x*x+y*y;
      if (z<=1) count++;
```

```
    }
  pi=(double)count/niter*4;
  printf("# iteracija= %d ,  procenjenja vrednost pi= %g \n",niter,pi);
}
```

2. nacin

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(void)
{ long t = 1000000;  /* broj testova */
  long r = 200;  /* radijus kruznice */
  long r2 = r / 2;
  long k = 0, i;

  for (i = 0; i < t; i++) {
    long a = (rand()%r) - r2 + 1;
    long b = (rand() %r) - r2 + 1;
    if (sqrt(a * a + b * b) <= r2) k++;
  }

  printf("Priblizno p = %.2f\n", (4.0 * k) / t);
  return 0;
}
```

6. Za date tri matrice $A \in R^{m \times p}$, $B \in R^{p \times n}$, $C \in R^{m \times n}$ proveri da li je $C = A \cdot B$

Naivni algoritam: izračunati i proveriti jednakost. Za sada (jul 2014), najefikasniji algoritam za množenje matrica, modifikacija Coppersmith–Winograd algoritam, (http://theory.stanford.edu/~virgi/matrixmult-f.pdf) ima vremensku složenost $O(n^{2.373})$

Rešenje: Probabilistički metod (Monte Carlo algoritam)

Algoritam (A,B,C,m,n,p)
Input: Matrice $A \in R^{m \times p}$, $B \in R^{p \times n}$, $C \in R^{m \times n}$.
Output: TRUE, ako je $C = A \cdot B$;
FALSE, ako je $C \neq A \cdot B$
begin
  i=1
  repeat
    Nasumično izabrati n-torku brojeva $r=(r_1,...,r_n) \in \{0,1\}^n$
    Izračunati $C \cdot r$ i izračunati $A \cdot (B \cdot r)$
    if $C \cdot r \neq A \cdot (B \cdot r)$
      return FALSE
    endif
    i = i + 1
  until i=k
  return TRUE
end

Vremenska složenost algoritma je $O(\boldsymbol{kn^2})$ (u pitanju su proizvodi matrica*vektor, u praksi postoje veoma brzi algoritmi izračunavanja takvog proizvoda).

Primer primene algoritma:
Neka su nam date matrice A, B, C tako da želimo da proverimo

$$AB = \begin{bmatrix} 2 & 3 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 2 \end{bmatrix} \stackrel{?}{=} \begin{bmatrix} 6 & 5 \\ 8 & 7 \end{bmatrix} = C.$$

Inače, jasno da A·B= $\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$

1. iteracija

Neka smo nasumično izabrali vektor $\vec{r} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

Tada je

$$\begin{aligned} A \times (B\vec{r}) - C\vec{r} &= \begin{bmatrix} 2 & 3 \\ 3 & 4 \end{bmatrix} \left(\begin{bmatrix} 1 & 0 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right) - \begin{bmatrix} 6 & 5 \\ 8 & 7 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 2 & 3 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \end{bmatrix} - \begin{bmatrix} 11 \\ 15 \end{bmatrix} \\ &= \begin{bmatrix} 11 \\ 15 \end{bmatrix} - \begin{bmatrix} 11 \\ 15 \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \end{aligned}$$

Dakle, dobili smo nula vektor, te postoji mogućnost da je A·B = C.

2. iteracija

Neka smo nasumično izabrali vektor $\vec{r} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

Tada je

$$A \times (B\vec{r}) - C\vec{r} = \begin{bmatrix} 2 & 3 \\ 3 & 4 \end{bmatrix} \left(\begin{bmatrix} 1 & 0 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right) - \begin{bmatrix} 6 & 5 \\ 8 & 7 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \end{bmatrix}.$$

Dakle, nismo dobili nula vektor, te postoji mogućnost da je A·B ≠ C.

Ako je n=2, onda postoje cetiri razlicita izbora za 0/1 vektor r ($r=(0,0)^T$, $r=(0,1)^T$, $r=(1,0)^T$, $r=(1,1)^T$)

Dva od ta cetiri izbora ($r=(0,0)^T$, $r=(1,1)^T$) daju nula vektor u jednakosti $A \times (B\vec{r}) - C\vec{r}$

Dakle, verovatnoca da smo u k=2 iteracije izabrali ove vektore (i pogresno zakljucili da A·B = C) je (1/2)*(1/2)=1/4

Dakle, u opstem slucaju, verovatnoca izbora onih vektora r koji dovode do pogresnog zakljucka da A·B = C je manja od ½.

Što je veći broj opita (iteracija), verovatnoća greške je sve manja. (Na primer, ako je k=30, verovatnoća greške iznosi 1/1073741824 =0,0000000093132)

Teorema: Algoritam daje tačan odgovor sa verovatnoćom koja je barem $1 - (\frac{1}{2})^k$.

Dokaz:
Dokažimo da ako $A\cdot B \neq C$, onda $p[A\cdot B\cdot r=C\cdot r] <= 1/2$

Ako $A\cdot B \neq C$, onda važi i da $D= A\cdot B-C \neq 0$.

Bez smanjenja opštosti, pretpostavimo da u matrici D element $d_{11} \neq 0$.

S druge strane $p[A\cdot B\cdot r=C\cdot r] = p[(A\cdot B-C)\cdot r=0] = p[D\cdot r=0]$

Ako $D\cdot r=0$, onda je prva vrste proizvoda $D\cdot r$ jednaka 0, tj.

$$\sum_{j=1}^{n} d_{1j} r_j = 0$$

Kako po pretpostavci vazi da $d_{11} \neq 0$, onda se može naći resenje za r_1:

$$r_1 = \frac{-\sum_{j=2}^{n} d_{1j} r_j}{d_{11}}$$

Ako fiksiramo sve vrednosti r_j sem vrednosti za r_1, onda gornja jednakost važi za najviše jedan od dva moguća izbora vrednosti za $r_1 \in \{0,1\}$ (vektori kao u gore datom primeru ($r=(0,0)^T$, $r=(1,1)^T$))
Dakle, $p[A\cdot B\cdot r=C\cdot r] <= 1/2$

Ciklus repeat until se ponavlja *k* puta.

1. slučaj: Ako $C= A\cdot B$, onda je algoritam uvek korektan;

2. slučaj: Ako $C \neq A\cdot B$, verovatnoća dobijanja tačnog odgovora nakon k ponavljanja je barem $1-(\frac{1}{2})^k$.

7. Neka je *E* zadati niz brojeva $x_1, x_2, \ldots, x_n$. Višestrukost broja *x* u *E* je broj pojavljivanja broja *x* u *E*. Odrediti element *x* višestrukosti veće od *n/*2 (”preovlađujući element”) ili ustanoviti da takav element ne postoji. Složenost algoritma treba da je *O*(*n*).

Na primer: 2 3 2 5 5 5 5 2 5 1 5 4 5 = > 5 je preovlađujući

Ideja 1 : izvrši se sortiranje, te ako postoji preovlađujući broj, on je u sredini i izvrši se za njega provera pozicije u sortiranom nizu. Može li efikasnije?

Ideja 2: Odrediti medijanu (videti u knjizi poglavlje o rangovskim statistikama - medijana je n/2-ti najmanji element), te je nađen kandidat za proveru. Može li efikasnije?

PODSEĆANJE: *Formulacija problema rangovskih statistika: Za zadati niz elemenata S sa n članova i broj k, 0 <= k < n, odrediti k-ti najmanji elemenat u S.* Kao i kod sortiranja razdvajanjem, loš izbor pivota vodi kvadratnom algoritmu.

Ideja 3: Ako je x[i] različito od x[j] i ako se izbace oba ova elementa iz niza, onda ako postoji *y* koji je preovlađujući element starog niza, onda je on preovlađujući element i novog niza (obratno ne važi).

U algoritmu se u jednom prolazu koriste promenljive *C*, *M*, gde *C* je jedini kandidat za preovlađujući element u nizu x[0],x[1],..,x[i-1].
M je broj pojavljivanja elementa *C* u nizu x[0], x[1],...,x[i-1], bez onih elemenata C koji su izbačeni.

Ako je x[i]==C, onda se M uvećava za 1, a inače smanjuje za 1. (tada se smatra da x[i] može biti izbačen zajedno sa nekim članom jednakim C).
Ako M postane 0, onda C dobije novu vrednost x[i] i time i M postane 1.

```
Algoritam Preovladjuje (x,n):
ulaz: x, n /* niz pozitivnih brojeva x, dimenzije n*/
izlaz: preovlada /* preovladjujuci alaement (ako postoji) ili -1 */

C=x[0];
M=1;
for(i=1; i < n; i++)
   if (M==0)
   {C=x[i]; M=1;}
   else
      if (x[i]==C) M++; else M--;

if (M==0) preovlada=-1;   /*nema preovladjujuceg elementa*/
else brojac=0;

for (i=0; i < n; i++)
   if (x[i]==C) brojac++;

if (brojac > n/2) preovlada=C; else preovlada=-1;
```

8. Napisati program (na programskom jeziku C) vremenske složenosti O(n) koji učitava sa standardnog ulaza niz realnih brojeva a, dimenzije n i pronalazi i ispisuje na standardni izlaz član niza čiji broj pojava u nizu je veći od n/4.

Na primer u nizu: 1 2 3 4 1 2 3 4 1 2 3 4 2
rešenje je 2

Rešenje:
Algoritam iz prethodnog zadatka modifikujemo tako da pamtimo tri različita umesto jednog kandidata.
Novi element poredimo sa tri tekuća kandidata, te ako je različit od svih, njihove tekuće vrednosti višestrukosti smanjujemo za 1. Ako, pri tome, višestrukost nekog od njih padne na 0, on ispada iz spiska kandidata.
Ako trenutno ima manje od 3 kandidata, onda se novi element uzima za novog kandidata sa tekućem višestrukošću 1.
Na kraju se preostali kandidati porede sa svim ostalim elementima da bi im se odredile tačne višestrukosti.

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX 100
#define NUM 4

int main()
{
  int a[MAX],i,n,j,k,prazno,x[NUM],p[NUM];
  printf("Unesite dimenziju niza! ");
  scanf("%d",&n);
  printf("\nUnesite clanove niza! ");
  for(i=0;i<n;i++) scanf("%d",a+i);

  for(j=0;j<NUM;j++) p[j]=-1;

  for(i=0;i<n;i++)
  {
   prazno=-1;

   for(j=0;j<NUM;j++)
   {
      if ((p[j]>=0) && (x[j]==a[i])) p[j]+=3;
      else p[j]-=1;
```

```
      if(p[j]<=0) prazno=j;
    }

    if(prazno>=0)
    {x[prazno]=a[i];
      p[prazno]=3;
    }
  }

  for(j=0;j<NUM;j++) p[j]=0;

  for(i=0;i<n;i++)
   for(j=0;j<NUM;j++)
     if(a[i]==x[j]) p[j]++;

   k=(n/4)+1;

   for(j=0;j<NUM;j++)
     if(p[j]>=k)
     {printf("%d ",x[j]); return 0;}


  printf("\nNe postoji takav element\n");


   return 0;
}
```

9. Zadat je niz rekurzivno tako da je: **F[n] = A*F[n-1] + B*F[n-2] + C*F[n-3],**. Date su početne vrednosti niza **F[1], F[2], F[3]** i konstante **A, B, C. Svaki od učitanih brojeva je iz intervala [0, 10000]. Konstruisati algoritam složenosti O(log n) koji će pronaći vrednost niza (F[n], F[n] mod 10000)** za zadato veliko *n* iz intervala **[1, 2000000000].**.

Rešenje:
Jasno je da postoji algoritam reda O(n).
Na primer, aко je n>4 onda: **for (i=4;i <n;i++) f[i]=a*f[i-1]+b*f[i-2]+c*f[i-3].**

Vremensko poboljšanje je moguće ako se problem svede na množenje matrica 4x4.

$$\begin{bmatrix} f4 & f3 & f2 & f1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} A & 1 & 0 & 0 \\ B & 0 & 1 & 0 \\ C & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} f5 & f4 & f3 & f2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Analogno,

$$\begin{bmatrix} f5 & f4 & f3 & f2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} A & 1 & 0 & 0 \\ B & 0 & 1 & 0 \\ C & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} f6 & f5 & f4 & f3 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Dakle, $\begin{bmatrix} f4 & f3 & f2 & f1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} A & 1 & 0 & 0 \\ B & 0 & 1 & 0 \\ C & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}^{n-4} = \begin{bmatrix} f_n & f_{n-1} & f_{n-2} & f_{n-3} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$

Komapktnije to možemo zapisati: $F_4 \times M^{n-4} = F_n$

Dakle, algoritam se sastoji od stepenovanja matrice M^{n-4}, šo je reda $O(\log_2 n)$. Da li se slažete sa tom složenošću? Obrazložite!!!

Nakon toga se rezultujuća matrica množi sa leve strane vektorom F_4. Rezultat koji se traži bio bi prva koordinaa vektora F_n.

```
#include <cstdio>
#include <iostream>

using namespace std;

struct matrix {
    int p[5][5];

    void init (){
      for ( int i=0;i<4;++i )
        for ( int j=0;j<4;++j )
          p[i][j] = 0;
    }

    matrix operator * ( matrix x ){
        matrix ret;
        ret.init();
        for ( int i=0;i<4;++i ){
          for ( int j=0;j<4;++j ){
            int z=0;
          for ( int k=0;k<4;++k )
              z = ( z + (p[i][k]*x.p[k][j])%10000 ) % 10000;
        ret.p[i][j] = z%10000;
        }
      }
      return ret;
     }


    };

void print ( matrix x ){
      for ( int i=0;i<4;++i,printf("\n") )
      for ( int j=0;j<4;++j )
        printf ("%d ",x.p[i][j]);
        return;
    }

matrix power ( matrix mat, int exp ){
    if ( exp == 1 ) return mat;
    if ( !(exp%2) ){
      matrix tmp = power ( mat,exp/2 );
     return tmp*tmp;
```

```
    }
    else return ( mat * power(mat,exp-1) );
    }

int main(){
  matrix m,ret,f4;
  int b[4],f[5];
  int n,sol;
  ret.init();

  scanf("%d%d%d",&f[0],&f[1],&f[2]);
  scanf("%d%d%d",&b[2],&b[1],&b[0]);
  scanf("%d",&n);

  if ( n < 4 ) printf ("%d\n",f[n-1]);
  else {
    f4.init();
    m.init();
    f4.p[0][0] = ( f[0]*b[2] + f[1]*b[1] + f[2]*b[0] ) % 10000;
    for ( int j=0;j<3;++j ) f4.p[0][j+1] = f[2-j];
     for ( int i=0;i<3;++i ) { m.p[i][0] = b[i] % 10000; m.p[i][i+1] = 1; }
    m = power(m,n-4);
    ret = f4 * m;
    sol = ret.p[0][0]%10000;

    printf ("%d\n",sol);
   }

  return 0;
  }
```

Zaključak: Linearne rekurentne formule sa **κ** nezavisnih početnih koeficijenata i **κ** zadatih početnih vrednosti niza mogu da se svedu na $F_{k+1} \times M^{n-k-1} = F_n$, gde su sve matrice F **(k+1)x(k+1).**

10. Dat je skup M.
r-bojenje skupa M je preslikavanje $c : M \rightarrow \{1, 2, \ldots, r\}$. Kažemo da je podskup $X \subset M$ monohromatski ako su svi elementi iz X obojeni istom bojom, tj. $\forall x, y \in X$: važi: $c(x) = c(y)$.
Neka nam je data familija F = $\{F_1, F_2, \ldots, F_s\}$ tako da je $|F_i| = k$ i $F_i \subset M$, za svako $1 <= i <= s$. Dokazati da ako je $s < r^{k-1}$ onda postoji takvo r-bojenje da nijedan skup F_i nije monohromatski.

Rešenje:
Ideja je dodeliti boje na sledeći način i pokazati da za $s < r^{k-1}$ važi da je verovatnoća da nijedan F_i nije monohromatski veća od 0.
Dodelimo svaku od boja nezavisno i uniformno svakom elementu skupa M:
P[c(x)=j]=1/r, za svako x iz M, 1<=j<=r

Kolika je verovatnoća da F_i jeste monohromatski?
Postoji r mogućnosti za boju:

P[F_i jeste monohromatski j]= $r \cdot \left(\frac{1}{r}\right)^{|Fi|} = r \cdot \left(\frac{1}{r}\right)^{|K|} = r^{-k+1}$

Neka je E događaj da nijedan od skupova F_i nije monohromatski.
Važi da $P(E)=1-P(E^C)$

Da bismo pokazali da $P(E)>0$, dovoljno je pokazati da $P(E^C)<1$

P(E^C)=P($\bigcup_{i=1}^{S} Fi$ monohromatski) <= $\sum_{i=1}^{S} P(Fi$ monohromatski)=s* r^{-k+1}

Po našoj pretpostavci, $s < r^{k-1}$, te važi da P(E^C) <1
Odatle sledi da P(E)>0.

11. Ramsey-ev broj $R(k)$ broja k je najmanji prirodan broj takav da svaki graf sa najmanje $R(k)$ čvorova sadrži ili kompletan graf sa k čvorova ili nezavisan skup od k čvorova.
Dokazati da je: $R(k) >= 2^{k/2-1}$.

Problem svodimo na prethodni problem. Umesto grafova nad V, |V|=n, razmatramo 2-bojenja grana kompletnog grafa Kn nad V.
Za dati graf G nad V dodeljujemo crvenu boju svim granama u Kn koje postoje u G, a plavu onim granama koje ne postoje u G.

Jasno G sadrzi K-kliku ako bojenje Kn ima crvenu k-kliku.
Slicno, G sadrzi K-nezavisan skup AKKO bojenje Kn sadrzi plavu k-kliku.

Stoga, graf nad n cvorova koji ne sadrzi ni k-kliku, ni K-nezavisan skup postoji
⇔ se grane kompletnog grafa nad n cvorova mogu obojiti sa 2 boje bez stvaranja monohromatske klike.

M=E(Kn)
F – familija svih k-klika u Kn (gledamo kao skup grana, jer njih bojimo)
|F|=s=$\binom{n}{k}$
|Fi|=$\binom{k}{2}$

Zeljeno bojenje postoji ako je $\binom{n}{k} < 2^{\binom{K}{2}-1}$
Poslednja nejednakost vazi prema
$$\binom{n}{k} < n^k < 2^{\frac{(k-2)k}{2}} = 2^{\frac{k(k-1)}{2}-\frac{k}{2}} = 2^{\binom{k}{2}-\frac{k}{2}} < 2^{\binom{k}{2}-1}$$
koristeci $n < 2^{\frac{k}{2}-1}$, k>=2

12. Na raspolaganju nam je procedura *Pristrasni random* koja vraća 0 ili 1 i to: 1 sa verovatnoćom p, a 0 sa verovatnoćom $1 - p$, gde je $0 < p < 1$, pri čemu ne znamo vrednost za p. Konstruisati algoritam koji koristi datu proceduru kao primitivu i kao izlaz daje nepristrasni odgovor - tj. daje 0 sa verovatnoćom 1/2 i 1 sa verovatnoćom 1/2. Koje je očekivano vreme izvršavanja u funkciji od p?

Algoritam Nepristrasni_random
Izlaz: 0 sa verovatnocom 1/2 i 1 sa verovatnocom 1/2

```
begin
while(1)
  do
     x:=Pristrasni_random();
     y:=Pristrasni_random();
     if (x!=y) return x;
```

end

Rešenje:
U ovom problemu sa bi se dobio bit (0 ili 1) nepristrasnog odgovora potrebno je pozvati Pristrasni_random() dva puta. I ponavljamo (while 1) pozive sve dok x==y. Kada dva poziva vrate različit rezultat x, y, onda vratimo kao rezultat prvi bit x

Da bi videli da li ***Nepristrasni_random*** vrati 0 ili 1 sa verovtanoćom 1/2, posmatrajmo verovatnoću da tekuća iteracija vrati 0
Pr $\{x = 0 \text{ and } y = 1\} = (1 - p)p$,
dok verovatnoća da tekuća iteracija vrati 1 je
Pr $\{x = 1 \text{ and } y = 0\} = p(1 - p)$.
(Očekujemo da bitovi koje vraća Pristrasni_random(); su nezavisni)

Otuda, verovatnoća da data iteracija daje 0 je jednaka verovatnoći da ta iteracija daje 1.
Kako ne postoji drugi način da Nepristrasni_random da rezultat, onda je verovatnoća rezultat 0 ista kao i rezultat 1, tj. 1/2.
Pretpostavljajuči da svaka iteracija ima vremensku složenost $O(1)$ time, onda je očekivana vremenska složenost ***Nepristrasni_random*** linearna po očekivanom broju iteracija. Možemo na svaku iteraciju gledati kao na Bernoulli pokus, gde status uspeha je da iteracija vrati rezultat. Verovatnoća uspeha je jednala verovatnoći da je vraćena 0 plus verovatnoća da je vraćena 1. Dakle, to je $2p(1 - p)$.
Broj pokusa dok se ne pojavi uspeh je dat geometrijskom distribucijom.
Znamo da očekivanje za geometrijsku distribuciju je

$$
\begin{aligned}
\mathrm{E}[X] &= \sum_{k=1}^{\infty} k q^{k-1} p \\
&= \frac{p}{q} \sum_{k=0}^{\infty} k q^{k} \\
&= \frac{p}{q} \cdot \frac{q}{(1-q)^2} \\
&= 1/p \,.
\end{aligned}
$$

Dakle, u našem primeru, očekivani broj pokusa je $1/(2p(1 - p))$.

Otuda sledi da očekivanio vreme izvršavanja Nepristrasnog_random je $\Theta(1/(2p(1 - p))$.

13. Neka je A niz od n različitih brojeva. Ako je $i < j$ i $A[i] > A[j]$ onda par(i, j) zovemo inverzijom od A. Pod pretpostavkom da je svaki element iz A izabran nezavisno na slučajan način i uniformno na domenu 1 do n izračunati očekivani broj inverzija.

14. Permutacija celih brojeva od 1 do n je niz $a_1, a_2, a_3, \ldots, a_n$, takav da svaki celi broj od 1 do n se pojavljuje u nizu tačno jednom. Dva cela broja u permutaciji obrazuju inverziju kada god veći broj se pojavljuje pre manjeg broja. Na primer, u permutaciji 4 2 7 1 5 6 3, postoji ukupno 10 inverzija. To su sledeći parovi: 4–2, 4–1, 4–3, 2–1, 7–1, 7–5, 7–6, 7–3, 5–3, 6–3. Napišite program koji računa broj inverzija u datoj permutaciji.

Ulaz
U prvoj liniji standardnog ulaza nalazi se prirodan broj n $(2 \le n \le 1000000)$. Permutacija je zadata u drugoj liniji: n brojeva, razdvojenih blanko karakterom.

Izlaz
Napišite broj inverzija na standardni izlaz.

Primer

Ulaz

```
7
4 2 7 1 5 6 3
```

Izlaz

```
10
```

Vremensko ograničenje 1s Memorijsko ograničenje 64 MB

Rešenje:

Naivno rešenje je direktna provera za svaki par indeksa i, j (tako da i<j) da li važi a[i] > a[j].

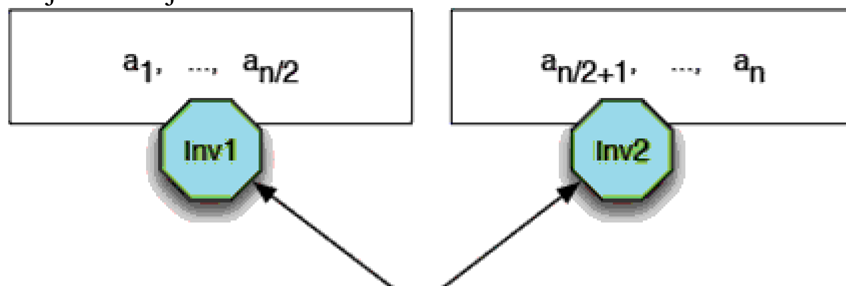
```
int InvCount(int a[], int n)
{
  int inv_count = 0;
  int i, j;

  for(i = 0; i < n - 1; i++)
    for(j = i+1; j < n; j++)
      if(a[i] > a[j])   inv_count++;

  return inv_count;
}
```

Ova direktna provera ima vremensku složenost $O(n^2)$ što je previše sporo, s obzirom na vremensko ograničenje u ovom zadatku.

Naime, kako je $n \leq 1000000$, potrebno je voditi računa da vremenska složenost rešenja $O(n^2)$ ne bi bila odgovarajuća za veliko n, tj. velike nizove.

Prebrojane inverzije pokazuju u kojoj meri je niz predsortiran. Ako je niz sortiran (u konkretnom zadatku u rastućem poretku), onda broj inverzija je 0. Ako je niz monotono opadajući, onda je broj inverzija maksimalan.

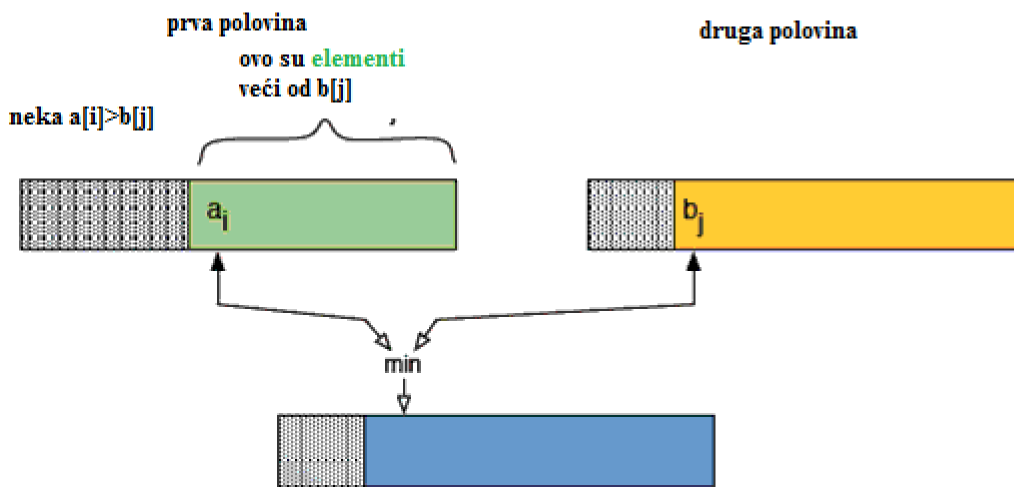
Pretpostavimo da znamo broj inverzija u levoj i desnoj polovini niza (sačuvanih redom u promenljivama inv1 i inv2). Koje inverzije eventualno nisu uračunate u zbiru inv1 + inv2?



Nisu uračunate verzije koje se pojavljuju pri spajanju polovina nizova.

Dakle, ukupan broj inverzija je jednak zbiru broju inverzija u levom i desnom podnizu i inverzija pri spajanju podnizova.

U procesu spajanja, neka se indeks i koristi za indeksiranje levog podniza i neka se j koristi za indeksiranje desnog podniza. U bilo kom koraku u modulu spajanja spojUredi(), ako važi da a[i] je veće od a[j] za i<j, onda postoji (sredina – i) inverzija, jer su levi i desni podniz sortirani, te preostali elementi u levom podnizu (a[i+1], a[i+2] … a[sredina]) su veći od a[j]




```
#include <iostream>
#include <stdio.h>
using namespace std;
int n, a[1000000];
long long r;
void spojUredi(int l, int d)
{    if(l>=d)      return;
     int s=(l+d)/2;
    static int b[1000000];
    int i=l,j=s+1,k=0;
    spojUredi(l,s);
    spojUredi(s+1,d);
   while(i<=s&&j<=d)
        if(a[i]<a[j])           b[k++]=a[i++];
        else
        {       b[k++]=a[j++];
               r+=s-i+1;
        }
    while (i<=s)        b[k++]=a[i++];
    while (j<=d)        b[k++]=a[j++];
    for(int p=0;p<=d-l;p++)    a[p+l]=b[p];
}
int main()
{    scanf("%d", &n);
     for(int i=0;i<n;i++)    scanf("%d", &a[i]);
    spojUredi(0,n-1);
    printf("%lld", r);

}
```

15. Posmatramo proces ubacivanja *n* identičnih loptica u *b* kutija (numerisanih sa 1, 2, . . ., *b*) na slučajan način. Ubacivanja su nezavisna i jednako je verovatno da loptica završi u svakoj od kutija.
(a) Koliki je očekivani broj loptica u svakoj od kutija?
(b) Koliki je očekivani broj ubacivanja loptica potreban da bi u datu kutiju bila ubačena loptica?
(c) Koliki je očekivani broj ubacivanja loptica potreban da bi svaka od kutija sadržala barem po jednu lopticu?