Algoritmi generiranja slučajnih permutacija

Kovče, Tihana

SVEUČILIŠTE U ZAGREBU
PRIRODOSLOVNO–MATEMATIČKI FAKULTET
MATEMATIČKI ODSJEK

Tihana Kovče

ALGORITMI GENERIRANJA SLUČAJNIH PERMUTACIJA

Diplomski rad

Voditelj rada:
izv. prof. dr. sc. Saša Singer

Zagreb, rujan 2018.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

U ovom diplomskom radu obradit ću temu slučajnih permutacija i algoritama koji ih generiraju. Permutacije su često korištene u kombinatorici, kriptografiji, umjetnoj inteligenciji, a susrećemo ih i u svakodnevnom životu – u kartaškim igrama i glazbi.

Nakon navođenja uvodnih definicija potrebnih za razumijevanje rada, istaknut ću nekoliko primjera algoritama koji, iako često korišteni, zapravo ne generiraju slučajne permutacije ili to čine neoptimalno. Nastavit ću s povijesnim razvojem, navodeći algoritme koji uistinu generiraju slučajne permutacije, od kojih je najpoznatiji Fisher–Yatesov algoritam. Paralelno ću opisati korake algoritama, komentirati složenosti te prikazati njihov rad u koracima.

Objasnit ću zašto kažemo da su pojedini algoritmi bazirani na transpozicijama elemenata, a drugi na sortiranju slučajnog niza. Dodatno ću navesti algoritam koji ne generira sve permutacije dane n-torke, već samo one u kojima niti jedan element neće ostati na svojoj inicijalnoj poziciji.

Opisat ću najčešće uzroke pristranosti prilikom generiranja slučajnih permutacija, uz pojašnjenja kako ih izbjeći. Potom ću prikazati implementacije ranije navedenih algoritama u programskom jeziku $C\#$, kao i testove provedene na njima.

Konačno, navest ću neke od brojnih praktičnih primjena algoritama generiranja slučajnih permutacija, kao i primjer zašto slučajne permutacije nisu uvijek poželjne.

Poglavlje 1

Osnovne definicije

1.1 Slučajna permutacija

Definicija 1.1.1. *Neka je S proizvoljan skup. Funkciju $a : \mathbb{N} \to S$ zovemo* ***niz*** *u S. Za $n \in \mathbb{N}$ pišemo $a(n) = a_n$ i nazivamo ga n-tim članom niza. Niz označavamo s $(a_n)_{n\in\mathbb{N}}$ ili $(a_n)_n$ ili samo (a_n).*

Ako je domena proizvoljan konačan skup $D \subset \mathbb{N}$ tada funkciju $a : D \to S$ zovemo ***konačnim nizom*** *u S.*

Definicija 1.1.2. *Neka je S konačan skup. Bijekciju $\pi : S \to S$ nazivamo* ***permutacijom*** *skupa S.*

Alternativno, **permutacija** n-članog skupa je svaka uređena n-torka elemenata zadanog skupa.

Standardni zapis permutacije π skupa $\{x_0, x_1, \ldots, x_{n-1}\}$ je:

$$\begin{pmatrix} x_0 & x_1 & x_2 & \cdots & x_{n-1} \\ \pi(x_0) & \pi(x_1) & \pi(x_2) & \cdots & \pi(x_{n-1}) \end{pmatrix}.$$

Postoji i kraći, tzv. **ciklički zapis** permutacije. Definirajmo prvo pojam ciklusa.

Definicija 1.1.3. *Kažemo da je permutacija π* ***ciklus*** *ili* ***ciklička permutacija*** *ako vrijedi:*

$$x_0 \mapsto x_1 \mapsto \cdots \mapsto x_{n-1} \mapsto x_0,$$

gdje su $x_0, \ldots, x_{n-1}$ različiti elementi skupa S u nekom redoslijedu.

Ciklus zapisujemo kao:

$$(x_0\ x_1\ \ldots\ x_{n-1}). \tag{1.1}$$

Primijetimo da je ciklus (1.1) jednak ciklusu

$$(x_{n-1}\ x_0\ x_1\ \ldots\ x_{n-2})$$

pa zaključujemo da ciklički zapis nije jedinstven.

Propozicija 1.1.4. *Svaka je permutacija kompozicija disjunktnih ciklusa.*

Dokaz. Neka su permutacija $\pi : S \to S$ i $x \in S$ proizvoljni. Vrijedi $\pi^{k_x}(x) = x$ za neki k_x, gdje je k_x duljina ciklusa koji sadrži x. Očito π možemo zapisati pomoću cikličkog zapisa kao:

$$\big(u\ \pi(u)\ \pi^2(u)\ \ldots\ \pi^{k_u-1}(u)\big)\ \big(v\ \pi(v)\ \pi^2(v)\ \ldots\ \pi^{k_v-1}(v)\big)\ \cdots . \qquad \square$$

Napomena 1.1.5. *Dogovorno ćemo sa S_n označavati skup $\{0, 1, 2, \ldots, n-1\}$, a s N_n niz $0, 1, 2, \ldots, n-1$.*

Primjer 1.1.6. *Neka je dana jedna permutacija skupa S_{10}:*

$$\pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 7 & 9 & 6 & 5 & 4 & 0 & 8 & 3 & 1 & 2 \end{pmatrix}.$$

Jedan mogući ciklični zapis permutacije π je:

$$\big(0\ 7\ 3\ 5\big)\ \big(4\big)\ \big(1\ 9\ 2\ 6\ 8\big).$$

Teorem 1.1.7. *Broj različitih permutacija n-članog skupa je $n!$.*

Dokaz. Pretpostavimo da je dan n-člani skup $S_n = \{0, 1, \ldots, n-1\}$. Permutacija skupa S_n je uređena n-torka elemenata iz S_n. Postoji n mogućnosti izbora prvog elementa iz skupa, $n-1$ mogućnost izbora drugog elementa, itd. Konačno, postoji jedna opcija za posljednji element. Dakle, ukupno imamo $n!$ različitih kombinacija elemenata iz S_n. $\square$

Definicija 1.1.8. *Proces ili odabir ćemo nazivati **slučajnim** ako je vjerojatnost svih mogućih ishoda jednaka.*

*Proces ili odabir u kojem neki ishodi imaju veću vjerojatnost pojavljivanja od drugih, nazivat ćemo **pristranim**.*

Napomena 1.1.9. *Primijetimo, postupak odabira (ili generiranja) jedne od $n!$ permutacija danog skupa nazivamo **slučajnim** ako je vjerojatnost odabira svake od permutacija jednaka $1/n!$.*

Definicija 1.1.10. *Za broj, permutaciju, niz i sl. reći ćemo da su **slučajni** ako su dobiveni slučajnim odabirom.*

1.2 Algoritam

Pojam **algoritam** kroz povijest je mijenjao svoje značenje, a danas ga često smatramo sinonimom za recept, proces, metodu ili tehniku. Točnije bi, ipak, bilo definirati **algoritam** kao konačan skup pravila, odnosno niz operacija za rješavanje određene klase problema.

Svaki algoritam ima sljedećih pet svojstava ([7]):

- **konačnost** – algoritam mora stati nakon konačno mnogo koraka (primijetimo da broj koraka može biti proizvoljno velik);
- **određenost** (definiranost, nedvosmislenost) – svaki korak algoritma mora biti jednoznačno definiran;
- **efikasnost** – očekuje se da je algoritam efikasan, što u praksi znači da se svaki njegov korak može izvršiti relativno jednostavno ("korištenjem olovke i papira") i u konačnom vremenu;
- **ulaz** – algoritam može, ali ne mora imati ulazne podatke;
- **izlaz** – algoritam mora imati barem jedan izlazni podatak.

Na primjeru ću pokazati da jedan od najpoznatijih algoritama – *Euklidov algoritam* zadovoljava navedene uvjete. Poznato je da navedeni algoritam rješava problem nalaženja najvećeg zajedničkog djelitelja dvaju prirodnih brojeva.

Primjer 1.2.1. *Neka su m i n proizvoljni prirodni brojevi. Sljedećim postupkom nalazimo njihov najveći zajednički djelitelj (u oznaci:* $\mathrm{NZD}(m,n)$*). Bez smanjenja općenitosti možemo pretpostaviti da je $m \geq n$, inače ih zamijenimo.*

Koraci algoritma:

1. [nađi ostatak pri dijeljenju] *neka je r ostatak pri dijeljenju m sa n (očito je $0 \leq r < n$);*
2. [je li ostatak jedak nuli?] *ako je $r = 0$ algoritam staje i rezultat je n;*
3. [prilagodi (smanji) brojeve] $m \leftarrow n, n \leftarrow r$ *i sljedeći korak je 1.*

Vidimo da je algoritam konačan: r će u svakom koraku poprimiti vrijednost između 0 i n. Ako vrijedi $r \neq 0$, u idućem ćemo koraku smanjiti n, a znamo da će padajući niz prirodnih brojeva u konačno mnogo koraka doći do nule. Dakle, algoritam će stati. Određenost je, također, zadovoljena: jasno je što znači podijeliti dva prirodna broja, odrediti ostatak tog dijeljenja, usporediti ga s nulom te pridružiti nove vrijednosti brojevima. Sve navedeno je relativno jednostavno izvedivo što potvrđuje svojstvo efikasnosti algoritma. Dodatno, znamo da su ulazni podaci prirodni brojevi m i n te da će algoritam sigurno imati izlazni podatak (jer u svakom slučaju dolazimo do drugog koraka).

1.3 Složenost algoritma

Primijetimo da svojstvo efikasnosti traži da svaki korak bude *"relativno jednostavno"* izvršiv. Jasno je da možemo imati više algoritama za rješavanje iste klase problema pa je potrebno odrediti neki oblik mjerenja koji je *"bolji"*. Analiziramo koliko je pojedini algoritam (kao kombinacija svojih koraka) zaista efikasan, odnosno koliko *"dobro"* rješava dani problem.

Definicija 1.3.1. *Količinu resursa potrebnu za dobivanje rezultata zadanim algoritmom nazivamo* ***složenost (kompleksnost) algoritma****. Preciznije,* ***složenost algoritma*** *je funkcija oblika $f : D \to \mathbb{R}$, gdje je D skup veličina ulaznih podataka svih problema koje algoritmi rješavaju. Intuitivno je jasno da je D odozgo neograničen podskup od npr. $\mathbb{R}$ ili $\mathbb{N}$.*

Za potrebe ovog rada ukratko ću navesti odabrane mjere složenosti koje ću kasnije koristiti za usporedbe algoritama. Uglavnom će nas zanimati vremenska i prostorna složenost.

Vremenska složenost je vrijeme potrebno za izvođenje algoritma izraženo u dogovorenim jedinicama.

Točno vrijeme izvođenja (u sekundama, minutama i sl.) ovisi o programskom jeziku, karakteristikama računala, kompajlera, itd. Zaključujemo, dakle, da izražavanje složenosti u takvom obliku nije objektivno niti pouzdano te ćemo iz tog razloga uspoređivati sam broj izvršenih instrukcija. Očito, niti taj podatak nije jednoznačan jer sigurno ovisi o ulaznim podacima. U ovom ću se radu koncentrirati na *najgori slučaj izvršenja algoritma.*

Definicija 1.3.2. *Neka su $f, g : D \to \mathbb{R}$ dvije funkcije na odozgo neograničenom skupu D.*

Kažemo da je g ***asimptotska gornja međa za*** *f ili da je f* ***manjeg reda veličine od*** *g ili da f* ***ne raste brže od*** *g, u oznaci*

$$f(x) = \mathcal{O}(g(x)),$$

ako postoje $c > 0$ i $x_0 \in D$ takvi da $\forall x \geq x_0$ vrijedi $f(x) \leq c \cdot g(x)$.

Prostorna složenost je računalna memorija potrebna na izvođenje algoritma izražena u dogovorenim jedinicama.

Iako se *bit* čini kao smislena jedinica za mjerenje, u praksi je vrlo nepraktičan zbog razlika u arhitekturama računala, odnosno načinu na koji zapisuju podatke u memoriju. Iz tog se razloga najčešće odlučujemo za mjerenje prostorne složenosti u

brojevima ili *riječima*. U ukupnu prostornu složenosti algoritma treba uračunati korišteni prostor za zapis instrukcija, ulaze, sve međurezultate, kao i izlaze, no dopuštamo da se pojedina memorija koristi više puta.

Upravo će, zbog zadnje činjenice, vrijediti da je *vremenska složenost uvijek veća od prostorne* (jer u više različitih koraka možemo koristiti istu memoriju).

Napomena 1.3.3. *Obično se složenost algoritma uspoređuje s nekom od sljedećih funkcija:*

$$1,\ \log_2 n,\ n,\ n\log_2 n,\ n^2,\ n^3,\ 2^n, \ldots$$

Primjer 1.3.4. *Ispišimo sve permutacije skupa S_n, $n \in \mathbb{N}$ zadan. Definiramo algoritam na način da permutacije generira jednu po jednu (nekim redoslijedom) te svaku generiranu odmah ispisuje. Dakle, memorija se može ponovno koristiti nakon ispisa.*

Precizna vremenska složenost ovisi o algoritmu, no jasno je da mora generirati $n!$ permutacija, dok je memorija potrebna maksimalno za dvije – prethodno ispisanu i iduću koju generira iz nje.

Primijetimo da prostorna složenost ipak ovisi o duljini ulaza, tj. ne može biti konstantna ($\mathcal{O}(1)$), no jednaka je točno memoriji potrebnoj za rad algoritma i za zapis dvije permutacije, tj.

$$f(n) = \mathcal{O}(n).$$

Poglavlje 2

Razvoj "shuffling" algoritama

2.1 Uvod

Podsjetimo se, permutaciju nazivamo slučajnom ako postupak njezinog dobivanja nije bio pristran, tj. vjerojatnost da upravo ona bude generirana bila je jednaka vjerojatnosti da bilo koja druga permutacija istog skupa bude generirana. Navedeni *postupak* generiranja ili odabira permutacija možemo zvati *algoritmom*.

Definicija 2.1.1. *Algoritme koji kao ulazni podatak primaju skup te u konačnom broju koraka generiraju slučajnu permutaciju nazivamo* ***algoritmima generiranja slučajnih permutacija*** *ili, kraće,* ***shuffling algoritmima****.*

Teorem 2.1.2. *Algoritam je shuffling algoritam ako i samo ako vrijedi:*

1. *za proizvoljan skup S, $|S| = n$, algoritam može generirati bilo koju od $n!$ permutacija,*
2. *vjerojatnost generiranja svake od permutacija je $1/n!$.*

Dokaz. Dokaz slijedi iz definicije 1.1.10 i teorema 1.1.7. □

Napomena 2.1.3. *U primjerima ćemo često komentirati permutacije konačnog niza (npr. permutacije špila karata ili polja u memoriji računala), no podrazumijeva se da je on u međukoraku generiran iz skupa, odnosno da su mu svi elementi različiti.*

U sljedećem ću poglavlju pokazati da nalaženje shuffling algoritma nije trivijalno.

2.2 Pristrani algoritmi generiranja slučajnih permutacija

Pogledajmo primjer (iz [5]) algoritma koji nasumično razmješta elemente polja.

Primjer 2.2.1. *Neka su u polje* `brojevi` *upisani elementi skupa koji želimo permutirati. Definiramo algoritam koji svaki element polja* `brojevi` *zamijeni s nekim slučajno odabranim elementom istog polja. Pseudokod navedenog algoritma:*

```
za svaki int i iz intervala [0, brojevi.duljina>
{
     zamijeni brojevi[i] i brojevi[random(brojevi.duljina)]
}
```

Funkcija `random(brojevi.duljina)` *vraća slučajno odabrani indeks danog polja, tj. slučajni broj iz intervala* $[0,$ `brojevi.duljina` $-1]$. *Pretpostavka je da će svaki indeks zaista biti odabran s vjerojatnošću* $1/$`brojevi.duljina`.

Primijenimo dani algoritam na jednom od najčešćih primjera iz prakse: miješanje špila karata. Jednostavnosti radi, neka špil ukupno ima tri karte. Neka su one međusobno različite i označene s A, B i C. Bez smanjenja općenitosti možemo ih fiksirati u inicijalni niz ABC. Analogno, u računalu možemo generirati polje `spil` (u obliku pseudokoda):

```
polje<charova> spil = {'A', 'B', 'C'};
```

Prikažimo rad algoritma u koracima.

1. Prvi element (A) zamjenjujemo s nekim nasumično odabranim, mogući rezultati su:
   - ABC – nazovimo ovaj poredak slučaj a),
   - BAC – nazovimo ovaj poredak slučaj b),
   - CBA – nazovimo ovaj poredak slučaj c).
2. Drugi element zamjenjujemo s nekim nasumično odabranim, mogući rezultati su:
   - nakon slučaja a):
     - BAC – nazovimo ovaj poredak slučaj aa),
     - ABC – nazovimo ovaj poredak slučaj ab),
     - ACB – nazovimo ovaj poredak slučaj ac),

- nakon slučaja b):
  - ABC – nazovimo ovaj poredak slučaj ba),
  - BAC – nazovimo ovaj poredak slučaj bb),
  - BCA – nazovimo ovaj poredak slučaj bc),
- nakon slučaja c):
  - BCA – nazovimo ovaj poredak slučaj ca),
  - CBA – nazovimo ovaj poredak slučaj cb),
  - CAB – nazovimo ovaj poredak slučaj cc).

3. Treći element zamjenjujemo s nekim nasumično odabranim, mogući rezultati su:
   - nakon slučaja aa): CAB, BCA, BAC,
   - nakon slučaja ab): CBA, ACB, ABC,
   - nakon slučaja ac): BCA, ABC, ACB,
   - nakon slučaja ba): CBA, ACB, ABC,
   - nakon slučaja bb): CAB, BCA, BAC,
   - nakon slučaja bc): ACB, BAC, BCA,
   - nakon slučaja ca): ACB, BAC, BCA,
   - nakon slučaja cb): ABC, CAB, CBA,
   - nakon slučaja cc): BAC, CBA, CAB.

Odmah je jasno da ovaj algoritam ne može biti *"pošten"*. Naime, skup od tri elementa (A, B, C) ima šest različitih permutacija, a algoritam je generirao dvadeset i sedam (27) jednako mogućih permutacija. S obzirom da $6 \nmid 27$ očito je da se neke permutacije pojavljuju češće od drugih. Točnije:

Permutacija:	ABC	ACB	BAC	BCA	CAB	CBA
Broj ponavljanja:	4	5	5	5	4	4

Tablica 2.1: Broj pojavljivanja svake od šest permutacija

Zaključujemo da algoritam iz primjera 2.2.1 ne spada u shuffling algoritme.

Što bi se dogodilo da nismo imali unaprijed generirani niz, već da smo samo karte (ili, općenito, podatke) upisivali na slučajno odabrano mjesto u polju?

Primjer 2.2.2. *Neka je dan algoritam koji elemente danog skupa S upisuje na slučajno odabrano mjesto u nizu.*

Pseudokod:

```
Definiramo:
    staroPolje: sadrži sve elemente iz S
    novoPolje: prazno polje jednake dimenzije

za svaki int i iz intervala [0, staroPolje.duljina>
{
    ponavljaj:
    {
         t = random(0, staroPolje.duljina)
    } dok je novoPolje[t] != prazno;

    novoPolje[t] = staroPolje[i];
}
```

Čak i bez prikaza rada s konkretnim podacima lako vidimo da ovakav algoritam ne želimo koristiti u praksi.

Porast broja ulaznih podataka značajno povećava složenost danog algoritma. Naime, algoritam će ponavljati generiranje slučajnog broja iz intervala [0, `brojPodataka`−1] dok ne nađe broj takav da je njegovo redno mjesto u polju prazno. Za prve će elemente nalaženje takvog mjesta biti relativno brzo, dok će kasnije vjerojatnost nalaženja praznog mjesta biti sve manja. Štoviše, nemamo garanciju da će proces uopće stati.

Lako možemo zaključiti da nalaženje optimalnog algoritma generiranja slučajnih permutacija nije trivijalno.

2.3 Fisher–Yatesov algoritam

Vratimo se kratko algoritmu iz primjera 2.2.1. Zanima nas uzrok pristranosti algoritma. Vidimo da je pristranost nastala jer nisu svi elementi imali jednaku vjerojatnost zamjene, odnosno pojedini su elementi bili zamijenjeni više puta od ostalih. Dakle, algoritam bi potencijalno bio shuffling kada bismo onemogućili da se isti elementi zamjenjuju više puta. Upravo su takav algoritam opisali R. Fisher i F. Yates u [3] i on se smatra prvim poznatim shuffling algoritmom. Opisao ga je i Knuth u [6]. Iz tog se razloga algoritam ponekad naziva i *Knuthovim algoritmom*.

Koraci algoritma

1. Za ulazni skup S, $|S| = n$, definiraj ekvivalentan niz *stariNiz* duljine n i *noviNiz* koji je u startu prazan.
2. Neka je k slučajno odabran broj iz intervala $[0, \mathit{stariNiz.duljina} - 1]$.
3. Postavi $\mathit{noviNiz.add}(\mathit{stariNiz}[k])$, gdje funkcija $\mathit{add}(x)$ dodaje x na kraj niza.
4. Ukloni $\mathit{stariNiz}[k]$ iz *stariNiz*.
5. Ako $\mathit{stariNiz.duljina} \neq 0$ vrati se na korak 2.
6. *noviNiz* je izlazna slučajna permutacija.

Prikažimo sada rad navedenog algoritma na stvarnim podacima:

Primjer 2.3.1. *Neka je dan niz N_5. Prikažimo korake algoritma:*

1. $\mathit{stariNiz} = [0, 1, 2, 3, 4]$, $\mathit{noviNiz} = [\ ]$,
2. *pretpostavimo da je odabran* $k = 2$,
3. $\mathit{stariNiz} = [0, 1, 3, 4]$, $\mathit{noviNiz} = [2]$,
4. *pretpostavimo da je odabran* $k = 3$,
5. $\mathit{stariNiz} = [0, 1, 3]$, $\mathit{noviNiz} = [2, 4]$,
6. *pretpostavimo da je odabran* $k = 0$,
7. $\mathit{stariNiz} = [1, 3]$, $\mathit{noviNiz} = [2, 4, 0]$,
8. *pretpostavimo da je odabran* $k = 1$,
9. $\mathit{stariNiz} = [1]$, $\mathit{noviNiz} = [2, 4, 0, 3]$,
10. *jedina mogućnost je* $k = 0$,
11. $\mathit{stariNiz} = [\ ]$, $\mathit{noviNiz} = [2, 4, 0, 3, 1]$.

Dakle, izlazna slučajna permutacija je

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 2 & 4 & 0 & 3 & 1 \end{pmatrix}.$$

Teorem 2.3.2. *Fisher–Yatesov algoritam je shuffling algoritam.*

Dokaz. Neka je S proizvoljan skup, $|S| = n$. Pretpostavimo da je odabir broja k zaista slučajan. Koristimo korolar 2.1.2 i [4].

Prvo dokažimo da algoritam može generirati svaku od $n!$ permutacija skupa S. Neka je π generirana permutacija. Napravimo kopiju konačnog polja *noviNiz* i nazovimo ju A. Dakle, u A je zapisana reprezentacija permutacije, tj.

$$A[i] = \pi(i).$$

Sortirajmo A algoritmom *selection sort*; rezultat će biti upravo početni niz N_n. Lako se vidi da je permutacija elemenata tijekom sortiranja identična permutaciji koju je ranije generirao algoritam (element na mjestu i u polju *noviNiz* upisali smo na mjesto $\pi(i)$ u polju A). Preciznije:

$$noviNiz[i] = A(\pi(i)).$$

Primijetimo sada da, neovisno o inicijalnom poretku elemenata u polju *staroPolje*, opisani algoritam može kao rezultat generirati uzlazno sortirani niz, odnosno permutaciju

$$\begin{pmatrix} x_0 & x_1 & x_2 & \dots & x_n \\ \pi(x_0) & \pi(x_1) & \pi(x_2) & \dots & \pi(x_n) \end{pmatrix}$$

takvu da vrijedi

$$\pi(x_0) \leq \pi(x_1) \leq \pi(x_2) \leq \cdots \leq \pi(x_n).$$

Konkretno, da bi algoritam generirao upravo takvu permutaciju trebalo bi se dogoditi da je na svakom slučajno odabranom mjestu k upravo najmanji element preostalog (još nepermutiranog) polja. Vjerojatnost je mala, ali postoji. U tom bi slučaju algoritam implementirao upravo *selection sort*. Dakle, moguć je slučaj da polje *staroPolje* reprezentira upravo permutaciju π.

Pokazali smo da algoritam ima strogo pozitivnu vjerojatnost da će bilo koje ulazno polje uzlazno sortirati, a kao što je ranije pokazano, taj je postupak je upravo jednak primjeni permutacije π na polju *staroPolje*.

Zaključujemo da za svaku permutaciju postoji strogo pozitivna vjerojatnost da će ju algoritam primijeniti na ulazni niz.

Dokažimo sada da je vjerojatnost generiranja svake permutacije upravo $1/n!$. Znamo da u prvom koraku slučajan k biramo između n vrijednosti, u drugom između $n-1$ itd. Dakle, broj mogućih izbora slijeda k-ova je:

$$n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1 = n! \tag{2.1}$$

Ranije smo pokazali da algoritam može generirati svih $n!$ permutacija danog skupa.

Dakle, postoji točno jedan slijed k-ova koji generira svaku od permutacija. Vjerojatnost odabira točno određene vrijednosti za k u prvom koraku je $1/n$, u drugom $1/(n-1)$, itd. Slijedi da je vjerojatnost odabira točno određenog slijeda od n k-ova (što odgovara točno jednoj permutaciji) jednaka:

$$\frac{1}{n} \cdot \frac{1}{n-1} \cdot \frac{1}{n-2} \cdots \frac{1}{2} \cdot 1 = \frac{1}{n!}. \tag{2.2}$$

Zaključujemo da su permutacije skupa S uniformno distribuirane. □

Složenost algoritma

- Prvi će korak (zbog definiranja inicijalnog niza *stariNiz*) biti izvršen u vremenu $\mathcal{O}(n)$.
- Pretpostavljamo da je za izbor slučajnog k iz zadanog intervala, kao i za funkciju *add*, dovoljno $\mathcal{O}(1)$ operacija.
- Uklanjanje elementa iz polja izvršavat će se u linearnom vremenu (jer sve elemente iza onog kojeg uklanjamo treba "pomaknuti" unaprijed).
- Operacija *stariNiz.duljina* $\neq 0$ izvršava se u linearnom vremenu, no može biti optimizirana uvođenjem varijable *duljinaNiza*.

Konačno, s obzirom na to da uklanjanje elementa treba provesti za svaki element, dobivamo kvadratnu vremensku složenost originalnog algoritma, odnosno vremenska složenost je $\mathcal{O}(n^2)$.

Primijetimo da smo, osim instrukcija samog algoritma, u memoriju spremali dva polja (čija duljina ovisi o ulaznom n) i varijable. Lako zaključujemo da je prostorna složenost algoritma $\mathcal{O}(n)$.

2.4 Durstenfeldov algoritam

U vrijeme kada je originalni Fisher–Yatesov algoritam opisan, on se uglavnom provodio pomoću olovke i papira i to na način da su se odabrani elementi u inicijalnom polju križali. Taj pristup u praksi nije zahtijevao da se ostali elementi prepisuju, odnosno "pomiču" unaprijed pa se navedeni korak izvršavao u konstantnom vremenu. Po dolasku računala, vrlo se brzo otkrilo da uklanjanje elementa iz polja u računalnoj memoriji nije operacija konstantne vremenske složenosti, već linearne. Slijedilo je da algoritam, zapravo, nije optimalan. Fisher–Yatesov originalni algoritam prilagodio je Durstenfeld u [2], a naziva ga se još i *modernim Fisher–Yatesovim algoritmom.*

Koraci algoritma

1. Za ulazni skup S, $|S| = n$, definiraj ekvivalentni niz A, $|A| = n$.
2. Postavi: $j \leftarrow |A| - 1$.
3. Neka je k slučajno odabran broj iz intervala $[0, j]$.
4. Postavi: $A[k] \leftrightarrow A[j]$.
5. Postavi: $j \leftarrow j - 1$.
6. Ako $j > 0$ vrati se na korak 3.
7. U A je upisana izlazna slučajna permutacija.

Prikažimo sada rad navedenog algoritma na stvarnim podacima.

Primjer 2.4.1. *Neka je dan niz N_5. Prikažimo korake algoritma (napomena: podebljano ću označiti fiksirane elemente):*

1. $A = [0, 1, 2, 3, 4],\ j = 4$,
2. *pretpostavimo da je odabran* $k = 1$,
3. $A = [0, 4, 2, 3, \mathbf{1}],\ j = 3$,
4. *pretpostavimo da je odabran* $k = 2$,
5. $A = [0, 4, 3, \mathbf{2}, \mathbf{1}],\ j = 2$,
6. *pretpostavimo da je odabran* $k = 0$,
7. $A = [3, 4, \mathbf{0}, \mathbf{2}, \mathbf{1}],\ j = 1$,
8. *pretpostavimo da je odabran* $k = 1$,
9. $A = [3, \mathbf{4}, \mathbf{0}, \mathbf{2}, \mathbf{1}],\ j = 0$.

Dakle, izlazna slučajna permutacija je

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 3 & 4 & 0 & 2 & 1 \end{pmatrix}.$$

Teorem 2.4.2. *Durstenfeldov algoritam je shuffling algoritam.*

Dokaz. Neka je S proizvoljan skup, $|S| = n$. Pretpostavimo da je odabir broja k zaista slučajan.

Tvrdimo da algoritam može generirati svaku od $n!$ permutacija skupa S. Dokaz te tvrdnje analogan je dokazu teorema 2.3.2 (primijetimo da u dokazu nije korišteno inicijalno polje).

Nadalje tvrdimo da je vjerojatnost generiranja svake permutacije jednaka $1/n!$. Tvrdnja se dokazuje analogno dokazu teorema 2.3.2, do na mogućnosti izbora k u (2.1) i (2.2). Moderna verzija algoritma nema posljednji korak (no tu je vrijednost 1, što ne mijenja rezultate). □

Složenost algoritma

- Prvi će korak (zbog definiranja inicijalnog niza A) biti izvršen u vremenu $\mathcal{O}(n)$.
- Pretpostavljamo da se naredbe:
  - izbor slučajnog k iz zadanog intervala,
  - $j \leftarrow |A| - 1$,
  - $A[k] \leftrightarrow A[j]$,
  - $j \leftarrow j - 1$,

– $j > 0$,

izvršavaju u konačnom broju ($\mathcal{O}(1)$) operacija.

Lako je vidjeti da algoritam (zbog ponavljanja operacija konstantne vremenske složenosti za svaki od elemenata inicijalnog polja) ima ukupnu vremensku složenost $\mathcal{O}(n)$.

Količina memorije potrebne za rad modernog algoritma slična je količini memorije potrebne za rad originalnog algoritma. Vidimo da moderna verzija algoritma koristi jedno umjesto dva polja te varijablu j u kojoj je spremljena duljina još nepermutiranog polja. Ukupna prostorna složenost algoritma je $\mathcal{O}(n)$.

2.5 Sattolin algoritam

Algoritam Sandre Sattolo, opisan u [9], poznat još i kao *Sattolin ciklički algoritam* ne spada u algoritme generiranja slučajnih permutacija. Naime, on permutira dani niz tako da niti jedan element ne ostane na svom početnom položaju, odnosno generira **cikličke permutacije**.

Koraci algoritma

1. Za ulazni skup S, $|S| = n$, definiraj ekvivalentni niz A, $|A| = n$.
2. Postavi: $j \leftarrow |A| - 1$.
3. Neka je k slučajno odabran broj iz intervala $[0, j-1]$.
4. Postavi: $A[k] \leftrightarrow A[j]$.
5. Postavi: $j \leftarrow j - 1$.
6. Ako $j > 0$ vrati se na korak 3.
7. U A je upisana izlazna slučajna permutacija.

Primijetimo da je implementacija vrlo slična Durstenfeldovom algoritmu. Točnije, razlikuju se jedino u intervalu mogućih vrijednosti slučajno odabranog broja k. Naime, skraćivanjem navedenog intervala za 1 onemogućena je zamjena elementa sa samim sobom, što osigurava cikličnost permutacije. Dokažimo tu tvrdnju.

Propozicija 2.5.1. *Sattolin algoritam generira cikličke permutacije.*

Dokaz. Neka je S proizvoljan skup, $|S| = n$. Pretpostavimo da je odabir broja k zaista slučajan. Indukcijom dokazujemo da je svaka permutacija koju algoritam generira ciklička.

Baza $n = 1$ je trivijalan slučaj. Primijenimo algoritam na dvočlani skup:

1. $A = [x_1, x_2]$, $j = 1$,
2. jedina mogućnost je $k = 0$,
3. $A = [x_2, \boldsymbol{x_1}]$, $j = 0$.

Algoritam je generirao permutaciju $(x_2\ x_1)$ koja je ciklička. Pretpostavimo da algoritam uvijek generira cikličku permutaciju za ulazni skup kardinalnosti $n-1$. Zapravo, pretpostavljamo da će algoritam, nakon prve iteracije (u kojoj premjesti jedan element), tijekom preostalih iteracija generirati cikličku permutaciju od prvih $n-1$ elemenata.

Promotrimo što se dogodilo: zadnji element inicijalnog niza u prvom je koraku premješten na neku od prvih $n-1$ pozicija, nazovimo ju p_1 (jer algoritam onemogućuje elementu zamjenu sa samim sobom).

Po pretpostavci: u koracima koju su slijedili generirana je ciklička permutacija svih elemenata osim zadnjeg. Dakle, kada bismo promatrali na koju je poziciju postavljen element koji je bio na poziciji p_1, nazovimo ju p_2, zatim na koju je poziciju postavljen element s pozicije p_2 i tako dalje, vratili bismo se na posljednju poziciju polja tek nakon analiziranih svih ostalih pozicija.

Time smo dokazali da algoritam generira samo cikličke permutacije. □

Teorem 2.5.2. *Sattolin algoritam generira* $(n-1)!$ *različitih slučajnih permutacija.*

Dokaz. Neka je dan skup

$$S = \{s_0, s_1, \ldots, s_{n-1}\}.$$

Fiksiramo ekvivalentni niz

$$N = s_0, \ldots, s_{n-1}.$$

Sattolin algoritam će svaki element premjestiti na poziciju različitu od njegove početne. Dakle, na prvo mjesto mogu biti odabrani svi elementi osim inicijalnog (njih $n-1$), na drugo svi osim inicijalnog i onog na prvom mjestu (njih $n-2$), itd. Zaključujemo da je broj različitih izbora jednak:

$$(n-1) \cdot (n-2) \cdots 2 \cdot 1 = (n-1)!.$$ □

Primjer 2.5.3. *Neka je dan niz* N_5*. Prikažimo korake algoritma (napomena: podebljano ću označiti fiksirane elemente):*

1. $A = [0, 1, 2, 3, 4]$, $j = 4$,
2. *pretpostavimo da je odabran* $k = 0$,
3. $A = [4, 1, 2, 3, \mathbf{0}]$, $j = 3$,
4. *pretpostavimo da je odabran* $k = 2$,
5. $A = [4, 1, 3, \mathbf{2}, \mathbf{0}]$, $j = 2$,

6. *pretpostavimo da je odabran* $k = 0$,
7. $A = [3, 1, \mathbf{4}, \mathbf{2}, \mathbf{0}]$, $j = 1$,
8. *jedina moguća opcija je* $k = 0$,
9. $A = [1, \mathbf{3}, \mathbf{4}, \mathbf{2}, \mathbf{0}]$, $j = 0$.

Dakle, izlazna slučajna permutacija je

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 3 & 4 & 2 & 0 \end{pmatrix},$$

u cikličkom zapisu: (1 3 2 4 0).

Složenost algoritma

Kao što sam već spomenula, jedina razlika u odnosu na Durstenfeldov algoritam je interval iz kojeg odabiremo slučajni k. U kontekstu složenosti – razlike nema. Dakle, i vremenska i prostorna složenost Sattolinog algoritma su $\mathcal{O}(n)$.

2.6 Generiranje permutacija sortiranjem

Ako bolje promotrimo modernu verziju Fisher–Yatesovog algoritma i Sattolinog algoritma, možemo primijetiti da su bili bazirani na *slučajnim transpozicijama.*

Definicija 2.6.1. *Permutaciju koja zamjenjuje točno dva elemente nazivamo* ***transpozicijom****. Točnije, transpozicija je permutacija oblika:*

$$\pi = \begin{pmatrix} 0 & 1 & 2 & \cdots & i & \cdots & j & \cdots & n-1 \\ 0 & 1 & 2 & \cdots & j & \cdots & i & \cdots & n-1 \end{pmatrix},$$

gdje su $i, j \in \{0, 1, \ldots, n-1\}$, $i \neq j$, *proizvoljni.*

Gornja tvrdnja slijedi iz činjenice da oba opisana algoritma imaju korak $A[k] \leftrightarrow A[j]$, što je zapravo transponiranje elemenata na pozicijama k i j, ako $k \neq j$.

Drugi način generiranja permutacija je *sortiranjem slučajnog niza.* Navest ću primjer shuffling algoritma baziranog na sortiranju slučajnog niza.

Koraci algoritma

1. Za ulazni skup S kardinaliteta n definiraj ekvivalentni niz A, $|A| = n$ i niz $B = \{\underbrace{0, 0, \ldots, 0}_{n}\}$.

2. Definiraj $i \leftarrow 0$.
3. Neka je $k > 0$ slučajno odabran.
4. Ako B sadrži k vrati se na korak 3.
5. $B[i] \leftarrow k, i \leftarrow i + 1$
6. Ako je $i < |A|$ vrati se na korak 3.
7. Sortiraj polje B uz ekvivalentne promjene polja A.
8. U novom A zapisan je prikaz rezultantne permutacije.

Primijetimo da smo interval odabira broja k ostavili nedefiniran. Prvo, nužno je u polje B upisati međusobno različite vrijednosti k-ova jer u suprotnom, nakon sortiranja, nećemo dobiti slučajne rezultate. Dakle, da bi algoritam stao u konačno koraka, broj mogućih izbora za k mora biti barem $|A|$. Prvi logičan izbor bio bi interval $[1, |A|]$.

No, lako vidimo da je gornji izbor intervala daleko od optimalnog, iz očitog razloga: u kasnijim će koracima biti vrlo teško naći "neiskorišteni" k.

Algoritam možemo ubrzati definiranjem proizvoljno većeg intervala.

Primjer 2.6.2. *Neka je dan niz $A = N_3$. Ukratko prikažimo rad algoritma:*

1. $A = [0, 1, 2]$, $B = [0, 0, 0]$,
2. *pretpostavimo da su redom odabrani* $k = 5,\ 2,\ 3$,
3. $A = [0, 1, 2]$, $B = [5, 2, 3]$,
4. *nakon sortiranja dobivamo:* $A = [1, 2, 0]$, $B = [2, 3, 5]$.

Izlazna permutacija je (1 2 0).

Složenost algoritma

Primijetimo da se algoritam sastoji od nekoliko dijelova:

- korak 1 inicijalizira proizvoljni niz u vremenu $\mathcal{O}(n)$,
- koraci 2–6 generiraju slučajni niz B jednake duljine (n) kao i A,
- korak 7 sortira B i simultano generira slučajnu permutaciju niza A,
- korak 8 definira izlazni podatak.

Korake 2–6 lako možemo zamijeniti generatorom slučajnog niza duljine n složenosti $\mathcal{O}(n)$. Preostaje, dakle, vidjeti kolika je složenost sortiranja n-članog niza. Znamo da najbrže poznato sekvencijalno sortiranje zahtijeva $\mathcal{O}(n \log n)$ operacija, no možemo implementirati i paralelno sortiranje složenosti $\mathcal{O}(n)$ (npr. *Radix soritranje*).

Zaključno, algoritam koji permutira skup sortiranjem slučajnog niza može se optimizirati do vremenske složenosti $\mathcal{O}(n)$.

Prostorna složenost nema značajnu ovisnost o izboru algoritma sortiranja. U većini slučajeva ćemo, osim memorije za instrukcije, koristiti memoriju za dva polja i nekoliko varijabli. Prostorna složenost je tada, također, $\mathcal{O}(n)$.

Poglavlje 3

Mogući uzroci pristranosti

3.1 Pristran odabir brojeva

Primijetimo da svi navedeni algoritmi koriste "slučajno odabran k", često iz zadanog intervala. U dokazima da dani algoritmi zaista generiraju slučajne permutacije uvijek smo pretpostavljali da će k zaista biti odabran na slučajan način, no to često nije slučaj.

U dokumentacijama programskih jezika često možemo vidjeti da svoje funkcije *random()* nazivaju pseudorandom generatorima brojeva (*Pseudo Random Number Generators* ili *PRNGs*). Do danas su *PRNG*-ovi prilično napredovali, no navest ću jedan mogući uzrok pristranosti:

Primjer 3.1.1. *Pretpostavimo da želimo slučajan broj iz intervala* $[0, 4]$. *Kod za dobivanje slučajnog broja u npr. programskom jeziku C# izgledao bi ovako:*

```
int slucajanBroj = new Random().Next();
Console.WriteLine("Maksimalni broj: {0},
     \n Odabrani broj: {1}",
     Int32.MaxValue, slucajanBroj);
```

Jedan mogući rezultat je:
Maksimalni broj: 2147483647
Odabrani broj: 1072264799

Rezultat nam ne odgovara jer želimo broj iz intervala $[0, 4]$.

Tipična greška bila bi napisati:

```
int trazeniBroj = slucajanBroj % 5;
```

Naime, iako ćemo zaista dobiti rezultat iz zadanog intervala, on ipak nije potpuno slučajan.

Primijetimo da je `slucajanBroj` *bio odabran iz intervala* $[0, 2147483647\rangle$, *odnosno* $[0, 2147483646]$.

U intervalu $[0, 2147483644]$ *ima točno jednako brojeva koji pri dijeljenju s* 5 *daju ostatke* 0–4. *Dodamo li u tom intervalu brojeve* 2147483645 *i* 2147483646 *dobivamo da će vjerojatnost dobivanja ostataka* 0 *i* 1 *biti ipak malo veća od vjerojatnosti dobivanja ostataka* 2–4.

U sličnim se situacijama preporuča koristiti predefinirane funkcije korištenog programskog jezika. U situaciji iz primjera 3.1.1 poželjno je koristiti naredbu

```
int slucajanBroj = new Random().Next(5);
```

jer će funkcija `Next(5)` generirati slučajan broj iz intervala $[0, 4]$.

3.2 Ograničenja veličina brojeva

Kao što smo vidjeli u prethodnom poglavlju, postoji maksimalni broj koji generator slučajnih brojeva može producirati. Njegova vrijednost ovisi o programskom jeziku, načinu zapisa koji odaberemo (Int32, Int64, Double i sl.), količini memorije koju imamo na raspolaganju itd., no uvijek je konačan. Trebamo biti svjesni da u određenim situacijama možemo trebati izbor iz većeg skupa brojeva nego što uvjeti dozvoljavaju. Tada je potrebno prilagoditi algoritam ili metodu generiranja brojeva jer u protivnom algoritam neke od rezultata neće generirati nikada.

Primjer 3.2.1. *Standardan špil karata sastoji se od pedeset i dvije* (52) *različite karte, dakle postoji* 52! *različitih permutacija špila karata. Prilikom generiranja slučajne permutacije Fisher–Yatesovim algoritmom prividno smo definirali bijekciju između permutacije i niza duljine* 52. *Svaki od elemenata navedenoga niza reprezentira jednu od karata iz špila. Da smo, umjesto između permutacije i polja, bijekciju pokušali definirati između permutacije i prirodnog broja, algoritam ne bi mogao generirati sve permutacije. Naime,*

$$52! \approx 2^{225.6}.$$

Odnosno, da bi mogao generirati različite prirodne brojeve za sve moguće permutacije špila od 52 *karte, generator random brojeva morao bi moći generirati minimalno* 52! *različitih brojeva. Dakle, generator bi morao imati preciznost zapisa od barem* 226 *bitova.*

Većina programskih jezika ima generatore preciznosti 32 *ili* 64 *bita, što je nedovoljno za generiranje dovoljno velikog broja različitih brojeva. U prethodnom smo poglavlju vidjeli da je maksimalni prikazivi broj u zapisu* 32*-bitne preciznosti jednak*

2147483647. *Primijetimo:*

$$12! = \ \ 479001600 < 2^{31} - 1 = 2147483647,$$
$$13! = 6227020800 > 2^{31} - 1 = 2147483647.$$

Zaključujemo da bi se algoritmom baziranim na bijekcijama između prirodnih brojeva i permutacija mogle generirati sve permutacije skupa kardinalnosti najviše 12.

3.3 Pogrešna implementacija

Pogrešna implementacija algoritama jedan je od češćih uzroka pristranosti rezultata. S obzirom na činjenicu da uglavnom baratamo poljima, tzv. *off-by-one-error*, u prijevodu *greška pomaka indeksa za jedan* česta je pojava.

Primjer 3.3.1. *Pretpostavimo da želimo implementirati algoritam generiranja slučajnih permutacija i odlučili smo se za Durstenfeldov. Smanjimo li, greškom, interval iz kojeg želimo slučajan broj* k *s* $[0,\ j]$ *na* $[0,\ j-1]$*, gdje je* $j = |A| - 1$*, zapravo ćemo implementirati Sattolin algoritam. Ranije smo dokazali da Sattoloin algoritam generira samo cikličke permutacije, odnosno naš će algoritam biti u mogućnosti generirati neku od* $(n-1)!$ *mogućih permutacija danog skupa, umjesto željenih* $n!$*.*

Poglavlje 4

Implementacije algoritama

U ovom ću poglavlju pokazati rad navedenih algoritama u praksi, na stvarnim podacima. S obzirom na činjenicu da je većina algoritama linearne složenosti te su implementacije nekih od algoritama slične, odlučila sam u fokus staviti rezultate, a ne brzinu u realnim vremenskim jedinicama. Iz istog sam razloga odlučila implementirati algoritme u programskom jeziku *C#*, koji ima dobru podršku za potrebne operacije, a ne u nekom od nižih programskih jezika (npr. u *C*-u).

4.1 Randomizator

Kao što je spomenuto u prethodnom poglavlju, jedan od koraka svih navedenih algoritama je odabir slučajnog broja. Prilikom njihove implementacije u jeziku *C#* koristit ću *.NET*-ov pseudorandom generator brojeva reprezentiran klasom

```
public class Random.
```

Naredba

```
Random rand = new Random();
```

inicijalizira *PRNG* pomoću stvarnog slučajnog broja kojeg nazivamo *sjeme* (eng. *seed*). S obzirom na činjenicu da je u ovom slučaju sjeme određeno sistemskim satom, objekti klase `Random` generirani u vremenskom razmaku manjem od 15 milisekundi često će imati isto sjeme. Iz tog se razloga, pri generiranju niza slučajnih brojeva, sjeme definira jednom, a potom se koristi naredba

```
Next();.
```

Naredba `Next()` generirat će slučajni cijeli broj iz intervala $[0,\ Int32.MaxValue]$. Alternativno, možemo koristiti neku od naredbi `Next(Int32)` ili `Next(Int32, Int32)`.

Npr. naredba `Next(max)`, odnosno `Next(min, max)` generirat će, redom, cijeli broj iz intervala $[0,\ \max - 1]$, odnosno $[\min,\ \max - 1]$.

Prije korištenja opisanog pseudorandom generatora u algoritmima, odlučila sam ga testirati sljedećom funkcijom:

```
public void TestirajRandomizator()
{
     int velicinaIntervala = Int32.Parse(Console.ReadLine());
     int[] interval = new int[velicinaIntervala];

     int brojRandomizacija = Int32.Parse(Console.ReadLine());
     Random random = new Random();
     int randomBroj;

     for(int i = 0; i < brojRandomizacija; i++)
     {
          randomBroj = random.Next(velicinaIntervala);
          interval[randomBroj]++;
     }
}
```

Primijetimo da prilikom svakog pokretanja algoritma imamo mogućnost odrediti broj `velicinaIntervala` takav da će slučajni brojevi `randomBroj` biti iz intervala $[0, velicinaIntervala - 1]$. Također, možemo odrediti koliko slučajnih brojeva želimo generirati. Rezultati će biti prikazani u polju `interval` na način da će vrijednost `interval[i]` biti ukupan broj generiranja broja i.

Algoritam sam pokrenula više puta s vrijednošću `velicinaIntervala = 5;`. Ovo su rezultati:

	Interval				
brojRandomizacija	0	1	2	3	4
1000	213	201	207	187	192
10000	2022	2026	1976	2012	1964
100000	20069	19924	20178	19809	20020

Tablica 4.1: Rezultati testiranja randomizatora za interval $[0,\ 4]$

Može se zaključiti da *.NET*-ov randomizator slučajne brojeve iz željenog intervala generira s gotovo uniformnom distribucijom.

4.2 Testovi

U sljedećim ću odjeljcima implementirati ranije opisane algoritme te tablično prikazati rezultate njihovog rada sa stvarnim podacima (skupovima S_n, za različite n-ove). Često ću koristiti randomizator iz prethodnog odjeljka te funkciju zamjene dvaju elemenata polja:

```
void zamijeni(ref int a, ref int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

Pristrani algoritam

Prvo ću prikazati rad algoritma iz primjera 2.2.1 kako bih provjerila jesu li dobivene vrijednosti konzistentne s vrijednostima u tablici 2.1.

```
public void JednostavnaZamjena()
{
    int brojPermutacija = Int32.Parse(Console.ReadLine());
    int duljinaNiza = Int32.Parse(Console.ReadLine());
    Random random = new Random();
    int randomBroj;

    Dictionary<string, int> rezultati = new
       Dictionary<string, int>();
    int[] niz = new int[duljinaNiza];
    int i, j;

    for (i = 0; i < brojPermutacija; i++)
    {
        //inicijalizacija niza
        for (j = 0; j < duljinaNiza; j++)
            niz[j] = j;
        //generiraj permutaciju
        for(j = 0; j < duljinaNiza; j++)
        {
            randomBroj = random.Next(duljinaNiza);
            zamijeni(ref niz[i], ref niz[randomBroj]);
```

```
        }
        //"spremi" permutaciju
        string permutacija = String.Join(",", niz.Select(p
           => p.ToString()).ToArray());

        if (!rezultati.ContainsKey(permutacija))
            rezultati.Add(permutacija, 1);
        else
            rezultati[permutacija]++;
    }
}
```

Prilikom pokretanja algoritma potrebno je odrediti brojeve `brojPermutacija` i `duljinaNiza` koji će redom definirati koliko permutacija će algoritam generirati te kardinalitet skupa kojeg permutiramo. Bez smanjenja općenitosti ćemo skup prezentirati nizom $N_{duljinaNiza}$.

Rezultati, tj. generirane permutacije bit će prikazane u rječniku `rezultati`.

Promotrimo dio koda koji "sprema" permutaciju: Naime, nakon generiranja, permutacija π

$$\pi = \begin{pmatrix} 0 & 1 & 2 & \cdots & velicinaNiza-1 \\ \pi(0) & \pi(1) & \pi(2) & \cdots & \pi(velicinaNiza-1) \end{pmatrix}$$

bit će spremljena u polje `niz` u obliku `niz[i]` $= \pi(i)$. Iz tog ćemo polja stvoriti `string permutacija` oblika:

$$\texttt{permutacija} = \texttt{niz[0]niz[1]}\ldots\texttt{niz[duljinaNiza-1]},$$

tj.

$$\texttt{permutacija} = \pi(0)\pi(1)\ldots\pi(\texttt{duljinaNiza}-1)$$

koji ćemo smatrati ekvivalentom permutacije π. Ako identičan `string` još ne postoji u rječniku `rezultati` zaključujemo da je ovo prva takva generirana permutacija. Tada u rječnik dodajemo par ključ–vrijednost `(permutacija, 1)`. Ako takva permutacija kao ključ u rječniku već postoji, njenu ćemo vrijednost povećati za jedan.

Nakon generiranja ukupno `brojPermutacija` permutacija, u rječniku će biti zapisane sve različite generirane permutacije, kao i ukupan broj koliko je puta pojedina bila generirana.

Algoritam sam pokrenula više puta i zadala različite `brojPermutacija`, no zbog jednostavnosti prikaza i usporedbe, testirala sam jedino skup S_3. Naime, skup S_2 ima samo dvije različite permutacije, a skup S_4 već ima njih $4! = 24$.

Podsjetimo li se primjera 2.2.1 vidjet ćemo da je očekivani broj pojava permutacija 021, 102 i 120 za otprilike 25% veći od broja pojava permutacija 012, 201 i 210. Dobiveni rezultati su:

brojRandomizacija	012	021	102	120	201	210
5000	751	927	930	955	722	715
50000	7418	9242	9295	9366	7281	7398
300000	44399	55512	55922	55286	44191	44690

Tablica 4.2: Rezultati testiranja algoritma iz primjera 2.2.1

Zaključujemo da su dobiveni rezultati testiranja dovoljno blizu očekivanima.

Fisher–Yatesov algoritam

Algoritam sam prvo implementirala u originalnoj verziji – dodavanjem elementa u novi niz i uklanjanjem iz starog. Slično kao i prethodni algoritam, i ovaj sam proširila tako da će generirati uneseni broj permutacija.

```
public void FisherYates()
{
    int brojPermutacija = Int32.Parse(Console.ReadLine());
    int duljinaNiza = Int32.Parse(Console.ReadLine());
    Random random = new Random();
    int randomBroj;

    Dictionary<string, int> rezultati = new
       Dictionary<string, int>();
    int i, j, tempDuljinaNiza;

    for (i = 0; i < brojPermutacija; i++)
    {
        tempDuljinaNiza = duljinaNiza;
        List<int> stariNiz = new List<int>();
        List<int> noviNiz = new List<int>();

        //inicijalizacija pocetnog niza
        for (j = 0; j < duljinaNiza; j++)
            stariNiz.Add(j);
```

```
            //generiranje permutacije
            while(tempDuljinaNiza != 0)
            {
                randomBroj = random.Next(tempDuljinaNiza);
                noviNiz.Add(stariNiz.ElementAt<int>(randomBroj));
                stariNiz.RemoveAt(randomBroj);
                tempDuljinaNiza--;
            }
            //(...)"spremi" permutaciju
        }
}
```

Spremanje permutacije u rječnik `rezultati` ekvivalentno je dijelu koda iz prethodnog odjeljka. Algoritam ću testirati na skupovima S_2 (tablica 4.3) i S_3 (tablica 4.4).

brojRandomizacija	01	10
100	54	46
10000	4949	5051
500000	250678	249322

Tablica 4.3: Rezultati testiranja Fisher–Yatesovog algoritma na skupu S_2

brojRandomizacija	012	021	102	120	201	210
1000	178	160	164	165	162	171
30000	4969	5005	5024	5001	5040	4961
500000	83456	83611	83239	83235	83593	82866

Tablica 4.4: Rezultati testiranja Fisher–Yatesovog algoritma na skupu S_3

Durstenfeldov algoritam

Sada ću pokazati implementaciju *moderne verzije Fisher–Yatesovog algoritma*. Možemo primijeniti da će implementacija ipak biti jednostavnija zbog korištenja jednog polja podataka, umjesto dvije liste te ranije navedene funkcije `zamijeni()` umjesto funkcija `Add()` i `RemoveAt()`. Dio koda koji brine za spremanje permutacija u rječnik `rezultati` ponovno neću eksplicitno navoditi.

```
public void Durstenfeld()
{
    int brojPermutacija = Int32.Parse(Console.ReadLine());
    int duljinaNiza = Int32.Parse(Console.ReadLine());
    Random random = new Random();
    int randomBroj;

    Dictionary<string, int> rezultati = new
       Dictionary<string, int>();
    int[] niz = new int[duljinaNiza];
    int i, j, tempDuljinaNiza;

    for (i = 0; i < brojPermutacija; i++)
    {
        tempDuljinaNiza = duljinaNiza;

        //inicijalizacija pocetnog niza
        for (j = 0; j < duljinaNiza; j++)
            niz[j] = j;

        //generiranje permutacija
        while (tempDuljinaNiza > 0)
        {
            randomBroj = random.Next(tempDuljinaNiza);
            swap(ref polje[tempDuljinaNiza -1], ref
               polje[randomBroj]);
            tempDuljinaNiza--;
        }
        //(...)"spremi" permutaciju
    }
}
```

Algoritam ću testirati na skupovima S_3 (tablica 4.5) i S_4 (tablica 4.6).

Za skup S_3 algoritam ću pokrenuti tri puta s različitim ukupnim brojevima permutacija, a za skup S_4 jednom – algoritam će generirati milijun permutacija.

brojRandomizacija	012	021	102	120	201	210
10000	1633	1715	1732	1632	1624	1664
300000	49830	50003	50311	49879	49957	50020
1000000	166569	166396	167200	166686	166898	166251

Tablica 4.5: Rezultati testiranja Durstenfeldovog algoritma na skupu S_3

permutacija	ukupno	permutacija	ukupno	permutacija	ukupno
0123	41692	1203	41405	2301	41656
0132	41424	1230	41837	2310	41650
0213	41308	1302	41937	3012	41623
0231	41553	1320	42083	3021	41312
0312	41790	2013	41968	3102	41288
0321	42177	2031	41735	3120	41650
1023	41487	2103	41544	3201	41904
1032	41836	2130	41483	3210	41658

Tablica 4.6: Rezultati testiranja Durstenfeldovog algoritma na skupu S_4

Usporedba vremena izvođenja

Zanimalo me koliko će zaista moderna verzija Fisher–Yatesovog algoritma (linearne vremenske složenosti) brže generirati permutacije od originalne verzije (kvadratne vremenske složenosti). Odlučila sam ranije navedene algoritme dopuniti naredbama za mjerenje vremena u stvarnim vremenskim jedinicama. U tu ću svrhu koristiti klasu `public class Stopwatch`.

Nakon koda gdje definiram željeni broj permutacija i duljinu niza koji permutiram, a prije početka generiranja permutacija, u kod sam dodala naredbe:

```
Stopwatch sw = new Stopwatch();
sw.Start();
```

Preciznosti radi, za ovaj sam test uklonila kod koji generirane permutacije sprema u rječnik `rezultati`. Nakon generiranja permutacija izvršit će se kod:

```
sw.Stop();
Console.WriteLine("Vrijeme izvrsavanja: {0} ms. ",
   sw.ElapsedMilliseconds);
```

koji će ispisati koliko je milisekundi bilo potrebno za izvršavanje pokrenutog algoritma. Dobiveni rezultati prikazani su u sljedećoj tablici 4.7:

duljina niza	broj permutacija	Fisher–Yates (ms)	Durstenfeld (ms)
20	10000	23	7
20	500000	1195	338
50	500000	2622	892
100	1000000	9981	3584

Tablica 4.7: Usporedba brzine originalne i moderne verzije Fisher–Yatesovog algoritma

Vidimo da je Durstenfeldov algoritam zaista značajno brži. S obzirom na njegovu linearnu složenost i ranije komentirano definiranje ekvivalencije između polja u memoriji (ili stringa) i permutacije, a ne broja i permutacije, možemo zaključiti da je Durstenfeldov algoritam optimalni sekvencijalni generator slučajnih permutacija čak i velikih skupova brojeva.

Sattolin algoritam

Neću posebno navoditi implementaciju algoritma jer, kao što je ranije spomenuto, od Durstenfeldovog se algoritma razlikuje jedino u gornjoj granici intervala iz kojeg odabiremo slučajan broj. Preciznije, dovoljno je naredbu

```
randomBroj = random.Next(tempDuljinaPolja);
```

zamijeniti naredbom

```
randomBroj = random.Next(tempDuljinaPolja - 1);.
```

Kao što je dokazano u teoremu 2.5.2, navedeni algoritam generira cikličke permutacije danog skupa i to s točno međusobno jednakom vjerojatnošću.

Algoritam ću testirati na skupovima S_3, S_4 i S_5, očekujemo da će algoritam generirati redom $2! = 2$, $3! = 6$ i $4! = 24$ različite permutacije. Rezultati testova prikazani su u sljedećim tablicama:

brojRandomizacija	120	201
100000	49922	50078
1000000	500044	499956

Tablica 4.8: Rezultati testiranja Sattolinog algoritma na skupu S_3

brojRandomizacija	1230	1302	2031	2310	3012	3201
50000	8384	8375	8314	8244	8340	8343
300000	49795	50358	49683	50281	49874	50009
1000000	166337	167017	167121	166145	167027	166353

Tablica 4.9: Rezultati testiranja Sattolinog algoritma na skupu S_4

permutacija	ukupno	permutacija	ukupno	permutacija	ukupno
12340	41473	23140	41911	34012	41419
12403	41442	23401	41820	34120	41780
13042	41604	24103	41542	40123	42042
13420	41398	24310	42049	40312	41977
14023	41949	30142	41544	42013	41774
14302	41401	30421	41524	42301	41422
20341	42015	32041	41412	43021	41739
20413	41971	32410	41421	43102	41371

Tablica 4.10: Rezultati testiranja Sattolinog algoritma na skupu S_5

Poglavlje 5

Primjene

Algoritmi generiranja slučajnih permutacija imaju vrlo široku primjenu. Navest ću nekoliko primjera:

- U strojnom učenju često susrećemo skupove podataka koji će biti korišteni za treniranje modela, testiranje i sl. Nužno je da su takvi podaci kvalitetno (slučajno) permutirani kako prilikom rada s njima ne bi bio prisutan određen uzorak, tj. kako ne bi bili pristrani. Posljedica je unaprijeđenje modela i njegovih performansi.
- Shuffling algoritmi korisni su za miješanje špila u brojnim kartaškim igrama; pokeru, bridžu, beli i sl. *Pokerstars*, jedna od najvećih online platformi za igranje pokera, u opisu svog dijeljenja karata ([1]) navodi da u svakoj igri karte "promiješa" unaprijed te se njihov poredak tijekom igre ne mijenja. Dakle, njihov algoritam prije početka igre generira jednu permutaciju špila karata. Mijenjanje permutacije tijekom igre (ili generiranje samo prvih nekoliko karata na početku) dovelo bi u pitanje integritet igre, ali i cijele platforme.
- Nedavno je na snagu stupila *Opća uredba o zaštiti podataka (GDPR)* koja, između ostalog, zabranjuje javno korištenje osobnih podataka. Da bi zaštitile podatke, tvrtke su često koristile različite softvere te namjene. Dok se neki od njih baziraju na tzv. *maskiranju*, odnosno kriptiranju podataka, drugi koriste algoritme generiranja slučajnih permutacija. Time podaci, iako su i dalje prisutni (u npr. bazi podataka), nisu međusobno povezivi, tj. nije moguće npr. broj bankovnog računa povezati s imenom ili OIB-om osobe.
- Prilikom slušanja glazbe nerijetko želimo slučajan redoslijed pjesama. Danas je u širokoj primjeni aplikacija *Spotify* putem koje korisnik ima pristup milijunima pjesama. Jedan od inženjera aplikacije objasnio je (u [8]) algoritam za shuffling pjesama implementiran u aplikaciji. On, naime, navodi kako su od prve verzije aplikacije koristili Fisher–Yatesov algoritam u svrhu savršenog slučajnog oda-

bira redoslijeda pjesama. Međutim, nedavno su algoritam prilagodili. Navest ću primjer gdje slučajna permutacije skupa ipak nije poželjno rješenje.

Primjer 5.0.1. *Ponovimo, u aplikaciji je za generiranje slučajnog redoslijeda odabranih pjesama korišten Fisher–Yatesov algoritam – algoritam generiranja slučajnih permutacija. Ipak, korisnici nisu bili zadovoljni. Inženjeri Spotifyja istražili su problem i otkrili sljedeće:*

Pretpostavimo da je korisnik odabrao skup od deset različitih pjesama koje želi slušati. Označimo pjesme slovima A, B, C i D, gdje su dvije pjesme označene istim slovom ako imaju istog autora. Pretpostavimo da se odabrani skup od deset pjesama sastoji od četiri pjesme A, tri pjesme B, dvije pjesme C i jedne pjesme D.

Primijenimo li Fisher–Yatesov algoritam na navedeni skup vjerojatnost dobivanja sljedeće dvije permutacije je jednaka:

$$ABACBADBAC \qquad AAAABBBCCA.$$

Naravno, korisnici nisu bili zadovoljni redoslijedom u drugoj permutaciji jer im se, zbog slijeda pjesama istog autora, činilo da skup pjesama zapravo nije "izmiješan". Kako bi zadovoljili korisnike, inženjeri su onemogućili generiranje svih permutacija danih skupova pjesama, odnosno uveli su dodatni uvjet da pjesme istog autora moraju biti maksimalno međusobno "udaljene" u slijedu.

Bibliografija

[1] J. Canny, *Monte–Carlo vs. Las Vegas*, https://people.eecs.berkeley.edu/~jfc/cs174/lecs/lec2/lec2.pdf.

[2] R. Durstenfeld, *Algorithm* 235*: Random permutation*, Communications of the ACM **7** (1964), br. 7, 420.

[3] R. A. Fisher i F. Yates, *Statistical tables for biological, agricultural and medical research*, 2. izd., Oliver & Boyd, Edinburgh, 1943.

[4] *Integrity of the shuffle*, https://www.pokerstars.com/help/articles/bb-shuffle-basic/10727/.

[5] M. James, *How Not To Shuffle – The Knuth Fisher–Yates Algorithm*, https://www.i-programmer.info/programming/theory/2744-how-not-to-shuffle-the-kunth-fisher-yates-algorithm.html, posjećeno 16. 8. 2017.

[6] D. E. Knuth, *The Art of Computer Programming, Volume* 2*: Seminumerical Algorithms*, 2. izd., Addison-Wesley, 1981.

[7] ———, *The Art of Computer Programming, Volume* 1*: Fundamental Algorithms*, 3. izd., Addison-Wesley, 1997.

[8] L. Poláček, *How to shuffle songs?*, https://labs.spotify.com/2014/02/28/how-to-shuffle-songs/, posjećeno 28. 2. 2014.

[9] S. Sattolo, *An algorithm to generate a random cyclic permutation*, Information Processing Letters **22** (1986), br. 6, 315–317.

Sažetak

Permutacija je uređeni niz svih elemenata danog skupa. Slučajne permutacije nalaze široku primjenu u brojnim granama matematike i računalnih znanosti, npr. u kombinatorici i umjetnoj inteligenciji, ali i u svakodnevnom životu; npr. za miješanje špilova karata i popisa pjesama. Cilj ovog diplomskog rada bio je navesti i analizirati algoritme generiranja slučajnih permutacija, tzv. shuffling algoritme.

Prvi se dio rada fokusirao na definiranje algoritama generiranja slučajnih permutacija i opisivanje kasnije korištenih metoda mjerenja i usporedbe njihovih složenosti. Također, naveli smo primjere pristranih i neoptimalnih algoritama, s ciljem prezentiranja loše prakse prilikom generiranja slučajnih permutacija.

U drugom su dijelu nabrojani shuffling algoritmi, opisane njihove instrukcije, procijenjene složenosti i u koracima prikazana njihova izvođenja na testnim skupovima. Među opisanim algoritmima je i Fisher–Yatesov algoritam generiranja slučajnih permutacija, jedan od najpoznatijih algoritama u računalnoj znanosti. Dodatno, osvrnuli smo se na česte uzroke pristranosti pri generiranju slučajnih permutacija te istovremeno komentirali kako ih uspješno izbjeći.

Konačno, algoritme smo implementirali u programskom jeziku *C#* te ocijenili njihovo izvođenje na kvantitativno značajnim podacima. Rezultati su prezentirani i uspoređeni, uz zaključak da je moderna verzija Fisher–Yatesovog algoritma istovremeno primjenjiva na dovoljno velikom skupovima, kao i zadovoljavajuće brza.

Summary

Permutation is a way of arranging all the elements of a given set into a sequence or an order. Random permutations are commonly used in different areas of mathematics and computer science, such as combinatorics and artificial intelligence, as well as in the everyday life; for arranging decks of cards, music play-lists, etc. The aim of this master's thesis was to describe and analyze algorithms for generating random permutations, also known as shuffling algorithms.

The first part of the thesis focused on defining shuffling algorithms and describing techniques later utilized for comparing their complexities. We also presented examples of how-not-to-shuffle, showing algorithms that are either biased or nonoptimal thus should not be used in data shuffling.

The second part enumerated shuffling algorithms by identifying their steps, calculating complexities and displaying their step-by-step execution on a given example set. Among others, we mentioned the Fisher–Yates shuffle, one of the most widely recognized algorithms in computer science. Furthermore, we addressed frequent causes of bias in shuffling algorithms, simultaneously providing their solutions.

Finally, the thesis validates given algorithms by implementing them in the programming language *C#* and executing the code with substantial input data. The outcomes are presented and compared, resulting in a conclusion that the modern, in-place version of the Fisher–Yates shuffle is both fast and applicable on arbitrarily large data sets.

Životopis

Rođena sam 25. studenog 1994. godine u Karlovcu. Završila sam Osnovnu školu "Ivan Goran Kovačić" Duga Resa te Srednju školu Duga Resa, smjer Opća gimnazija, s odličnim uspjehom. Tijekom osnovnoškolskog i srednjoškolskog obrazovanja sudjelovala sam na natjecanjima iz matematike, hrvatskog jezika, kemije, geografije i stolnog tenisa.

Po završetku srednjoškolskog obrazovanja upisujem preddiplomski sveučilišni studij Matematika na Prirodoslovno–matematičkom fakultetu Sveučilišta u Zagrebu. Isti završavam 2016. godine postavši tako sveučilišna prvostupnica matematike. U rujnu iste godine upisujem diplomski sveučilišni studij Računarstvo i matematika na istom fakultetu.

U travnju 2017. postajem aktivna u Udruzi studenata Europe u sklopu koje sam pohađala, a i samostalno održala razne radionice, neke od kojih su upravljanje projektima, rad u timu, komunikacija i konfliktologija itd.

Od rujna 2017. godine zaposlena sam na poziciji softverskog inženjera u tvrtci Ekobit d. o. o., gdje, u sklopu projekta za inozemnog partnera, radim na razvoju i održavanju softvera za automatizaciju pametnih poslovnih zgrada.