SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
V A R A Ž D I N

Ivan Užarević

FISHER - YATES ALGORITAM

SEMINARSKI RAD IZ KOLEGIJA ALGORITMI

Varaždin, 2016.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Ivan Užarević

Matični broj: 41945/13-R

Studij: Informacijski sustavi

FISHER - YATES ALGORITAM

SEMINARSKI RAD IZ KOLEGIJA ALGORITMI

Mentor:

Prof.dr.sc.Alen Lovrenčić

Varaždin, siječanj 2016.

Sadržaj

1. UVOD

"*Fisher-Yates*" algoritam, u engleskoj literaturi kolokvijalno pod imenom "*Fisher-Yates shuffle*" predstavlja algoritam za miješanje elemenata nekoga niza. U prijevodu riječ je o algoritmu koji generira nasumičnu permutaciju konačnoga niza elemenata. Sam pojam mogao bi se pojasniti i ispreplitanjem nekoga niza. Algoritam je po predmetu promatranja vezan za kombinacijske algoritme, odnosno skupinu algortama vezanih za rad sa nizovima permutacija (Prema: Wikipedia, 2015.). Treba naglasiti i glavnu prednost ovoga algoritma i zašto je njegova primjena izrazito važna kod programera, ali i šire u računalnoj znanosti. Glavna prednost leži u činjenici da je riječ o algoritmu koji ima karakteristiku "*unbiased*". U prijevodu ovaj algoritam je uravnotežen, nepolariziran, odnosno omogućuje izbor permutacije sa istom vjerojatnošću i to za svaku moguću premutaciju. Zbog toga sam algoritam ima svojstvo nedeterminiranosti. Osim "*unbiased*" svojstva, algoritam je lako razumljiv, dokazan i testiran u raznim implementacijama što ga čini najboljim izborom za programera. Početnici u programiranju često se služe takozvanim "naivnim" vrstama algoritama za mješanje što ponekad zna dovesti do toga da algoritam ne radi ono što bi trebao raditi, ne ispunjava svrhu koju je programer prvenstveno zamislio. To je najčešće zbog toga što takozvani "naivni" algoritmi imaju svojstvo "*biased*", odnosno polarizirani su, neuravnoteženi i s time njihova efikasnost je manja.

Sam algoritam originalno je zamišljen od strane znanstvenika Ronalda Fishera i Franka Yatesa još iz 1938. godine gdje su ga objasnili u svojoj knjizi "*Statistical tables for biological, agricultural and medical research*". Ta metoda je znana pod imenom originalna metoda ili "*Pencil-and-paper method*"(engl.). Veliku ulogu u razvoju ovoga algoritma su imali i Richard Durstenfeld te Donald E. Knuth koji je najviše pridonijeo popularizaciji algoritma zahvaljujući svojoj knjizi "*The Art of Computer Programming*". Zbog toga sam algoritam još nosi i naziv "*Knuth Shuffle*" iliti u samoj knjizi pod nazivom "*Algoritam P*". No osim popularizacije, Durstenfeld i Knuth su ovaj algoritam uspjeli i modificirati te poboljšati te je on još poznat u obliku takozvane moderne metode samoga algoritma. (Prema: Wikipedia, 2015)

S obzirom na navedeno, ovaj seminarski rad obuhvatiti će rad ovoga algoritma na primjeru metoda kojima se služi te usporedbu i važnost ovoga algoritma u odnosu na ostale algoritme istoga tipa. Uz prinicip rada i uspredbu potrebno je dotaknuti se i primjene samog algoritma u praksi.

2. FISHER - YATES ALGORITAM

2.1. Fisher - Yates originalna metoda

Za uvođenje u rad samoga algoritma može se specifirati i izreći intuitivno objašnjenje na koji način algoritam funkciniora. Npr. pretpostavi se da postoji šešir u kojem se nalaze kuglice sa brojevima, npr. kuglice za bilijar. Kuglice unutar šešira se dobro izmješaju. Postepeno se izvlače kuglice iz šešira i postavljaju redom kojim su izvađene iz šešira. Na taj način formira se niz kuglica. S obzirom da se ne može točno odrediti koja će se kuglica točno izvući, vjerojatnost da će se izvući određena kuglica je jednaka naspram svih ostalih koje su ostale u šeširu. Nakon što se iz šešira izvuku sve kuglice rezutat je niz kuglica koje zapravo predstavljaju nasumičnu permutaciju kuglica koje su se nalazile u šeširu. (Prema: Eli's Benderskys website, 2010). Naravno ovo objašnenje ne predstavlja matematički dokaz funkcioniranja algoritma, ali intuitivno se može raspoznati o čemu je zapravo riječ. Ipak logika koja stoji iza ovoga objašnjenja je obrađena u ostalim segmentima seminarskoga rada. Prije svega, za početak potrebno je nešto reći o originalnoj metodi samoga algoritma.

Fisher - Yates originalna metoda vezana je uz znastvenike R. Fishera i F. Yatesa. Koristila se običnim papirom i olovkom te je još nazvana "*Pencil-and-paper method*"(engl.). Kao vrsta algoritma zasniva se na 5 koraka:

1. Zapisati brojeve od 1 do N
2. Izabrati nasumični broj k između broja 1 i trenutne duljine niza
3. Izabrati k-ti broj u nizu od lijeva prema desno, prekrižiti ga i zapisati ga na drugo mjesto (npr. novi niz)
4. Ponavljati 2. i 3. korak sve dok dok nisu izbačeni svi brojevi iz niza
5. Niz brojeva koji su zapisani u koraku 3 predstavljaju novi niz koji predstavlja nasumičnu permutaciju početnoga niza

(Prema: Wikipedia, 2015.)

Algoritam se može objasniti i na primjeru:

Potrebno je prema 1. koraku zapisati 10 brojeva od 1 do N koji će predstavljati početni niz. Npr. neka je riječ o brojevima 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

(1 2 3 4 5 6 7 8 9 10)

Prema 2. koraku potrebno je izabrati nasumični broj k. U ovom slučaju on može biti između (1 - 10) jer trenutno postoji 10 elemenata u nizu i taj broj ujedno predstavlja poziciju u trenutno promatranom nizu. Npr. **k = 4**.

Prema 3. koraku potrebno je k-ti broj u nizu prekrižiti i zapisati ga u novi niz. A to je broj 4.

(1 2 3 5 6 7 8 9 10) (IZBAČEN: 4)

Novi niz izgleda ovako:

(4)

Ponavljaju se koraci 2 i 3. Broj k sada može biti izabran između (1-9) jer trenutno postoji 9 elemenata u nizu. Neka je **k = 7**. Pa je broj koji treba izbaciti i dodati u novi niz 8.

(1 2 3 5 6 7 9 10) (IZBAČEN: 8)

Novi niz izgleda ovako:

(4 8)

Ponavljaju se koraci 2 i 3. Broj k sada može biti izabran između (1-8) jer trenutno postoji 8 elemenata u nizu. Neka je **k = 4**. Pa je broj koji treba izbaciti i dodati u novi niz 5.

(1 2 3 6 7 9 10) (IZBAČEN: 5)

Novi niz izgleda ovako:

(4 8 5)

Ponavljaju se koraci 2 i 3. Broj k sada može biti izabran između (1-7) jer trenutno postoji 7 elemenata u nizu. Neka je **k = 2**. Pa je broj koji treba izbaciti i dodati u novi niz 2.

(1 3 6 7 9 10) (IZBAČEN: 2)

Novi niz izgleda ovako:

(4 8 5 2)

Ponavljaju se koraci 2 i 3. Broj k sada može biti izabran između (1-6) trenutno postoji 6 elemenata u nizu. Neka je **k = 6**. Pa je broj koji treba izbaciti i dodati u novi niz 10.

(1 3 6 7 9) (IZBAČEN: 10)

Novi niz izgleda ovako:

(4 8 5 2 10)

Ponavljaju se koraci 2 i 3. Broj k sada može biti izabran između (1-5) trenutno postoji 5 elemenata u nizu. Neka je **k = 3**. Pa je broj koji treba izbaciti i dodati u novi niz 6.

(1 3 7 9) (IZBAČEN: 6)

Novi niz izgleda ovako:

(4 8 5 2 10 6)

Ponavljaju se koraci 2 i 3. Broj k sada može biti izabran između (1-4) jer trenutno postoje 4 elementa u nizu. Neka je **k = 3**. Pa je broj koji treba izbaciti i dodati u novi niz 7.

(1 3 9) (IZBAČEN: 7)

Novi niz izgleda ovako:

(4 8 5 2 10 6 7)

Ponavljaju se koraci 2 i 3. Broj k sada može biti izabran između (1-3) jer trenutno postoje 3 elementa u nizu. Neka je **k = 1**. Pa je broj koji treba izbaciti i dodati u novi niz 1.

(3 9) (IZBAČEN: 1)

Novi niz izgleda ovako:

(4 8 5 2 10 6 7 1)

Ponavljaju se koraci 2 i 3. Broj k sada može biti izabran između (1-2) jer trenutno postoje 2 elementa u nizu. Neka je **k = 1**. Pa je broj koji treba izbaciti i dodati u novi niz 3.

(9) (IZBAČEN: 3)

Novi niz izgleda ovako:

(4 8 5 2 10 6 7 1 3)

Ponavljaju se koraci 2 i 3. Broj k sada jedino može biti 1 jer imamo samo 1 elemenat u nizu. **k = 1** pa je broj koji treba izbaciti i dodati u novi niz 9.

Početni niz je prazan. (IZBAČEN: 9)

Novi niz izgleda ovako:

(4 8 5 2 10 6 7 1 3 9)

S obzirom da više nema elemenata u promatranom nizu, prema koraku 5 utvrđuje se da novostvoreni niz (4 8 5 2 10 6 7 1 3 9) predstavlja nasumičnu permutaciju početnoga niza (1 2 3 4 5 6 7 8 9 10).

Prednost ove metode je što je ona poprilično jednostavna i prihvatljiva čovjeku, ali naravno kao takva ima svoje nedostatke. Problem ove metode proizlazi iz toga što se mora koristiti dodatna struktura za pohranu elemenata nasumične permutacije što dodatno troši memoriju. Naravno još jedan od problema predstavlja i kvadratna složenost $O(n^2)$.

2.2. Fisher - Yates moderna metoda

S obzirom da je originalna metoda od strane Fishera i Yatesa više služila kao teorijska osnova, Richard Durstenfeld 1964. godine opisuje modificiranu verziju ovoga algoritma s prvenstvenom uporabom za računala. Problem osnovne metode, kako je već navedeno, javlja se u kvadratnoj složenosti $O(n^2)$ i to zbog razloga što algoritam treba znati koliko je elemenata ostalo u početnom nizu nakon prebacivanja elementa iz promatranoga niza u novi niz. Odnosno, algoritam svoje vrijeme troši na prebrojavanje elemenata koji su preostali u početnom nizu (permutaciji). Da bi se to izbjeglo Durstenfeld uvodi malu, ali značajnu izmjenu za sam algoritam. Vrši se zamjena između elementa koji se "izbacuje" iz niza te zadnjeg elementa u promatranom nizu i to u svakoj iteraciji zamjene. To je omogućilo i razvoj ovoga algoritma na vrlo specifičan način. Zbog ove izmjene vremenska složenost je smanjena sa $O(n^2)$ na O(n). Složenost se može detaljnije pojasniti. Samo generiranje nasumičnoga broja izvršava se u konstantnom vremenu. S obzirom da je ideja da se kreće od zadnjeg elementa i taj element se zamjenjuje sa bilo kojim elementom u nizu uključujući i njega samog. Rapon niza se u sljedećem koraku nad kojim će se ponovno vršiti zamjena elemenata smanjuje za jedan i sve tako dok se ne dođe do prvoga elementa. Iz toga proizlazi da je složenost algoritma O(n). (Prema: GeeksforGeeks, 2014). Upravo ovaj primjer iskazuje kako male izmjene u algoritmu mogu dovesti do značajnih poboljšanja, ali također naglašava koliko je istraživanje samih algoritama važno za razvoj računalne znanosti. (Prema: Mička, 2015; Wikipedia, 2015)

Knuth(1997:145) u svojoj knjizi "*The Art of Computer Programming*", u svesku 2, navodi postupak za modernu verziju Fisher - Yates Algoritma. Pretpostavi se da postoji niz od *t* brojeva kojeg treba izmješati. Postupak se svodi na 4 koraka:

1. Inicijalizacija: Vrijednost *j* predstavlja početnu vrijednost broja elemenata nekoga niza i ona je jednaka vrijednosti *t.*
2. Generiranje broja *U*: *U* predstavlja nasumično generiran broj i to generiran tako da je vjerojatnost odabira svakog broja jednaka (engl: "unbiased") između 0 i 1.
3. Zamjena: Postavljanje vrijednosti *k*, *k = [j * U] + 1*, k je zbroj najvećeg cijelog umnoška brojeva *j* i *U* te broja 1 (Na ovaj način *k* predstavlja nasumičan broj između 1 i *j*). Mijenjaju se elementi na pozicijama *k* i *j*.
4. Dekrementirati j: Dekrementirati j za 1. Ako je j > 1, vratiti se na 2. korak

Algoritam se može objasniti i na primjeru:

Neka se početni niz sastoji od brojeva 1, 2, 3, 4, 5. S obzirom na zadani niz, t = 5.

Početni niz: (1 2 3 4 5)

Na temelju 1. koraka inicijalizira se vrijednost j na vrijednost t pa je **j = t = 5**.

Na temelju 2. koraka generira se broj U između 0 i 1, npr. 0.56

Na temelju 3. koraka računa se vrijednost k, odnosno k = [5 * 0.56] + 1 = [2.8] + 1, tj. **k = 3**. Vrši se zamjena elemenata na pozicijama k i j (k = 3, j = 5). Odnosno dolazi do zamjene između brojeva 3 i 5. Niz sada izgleda ovako:

(1 2 **5** 4 **3**)

Na temelju 4. koraka dekrementira se j, pa je **j = 4**. S obzirom da je j > 1 ponavlja se 2. korak.

Na temelju 2. koraka ponovno se generira broj U, npr 0.112

Na temelju 3. koraka računa se vrijednost k, odnosno k = [4 * 0.112] + 1 = [0.448] + 1, tj. **k = 1**. Vrši se zamjena elemenata na pozicijama k i j (k = 1, j = 4). Odnosno dolazi do zamjene između brojeva 1 i 4. Niz sada izgleda ovako:

(4 **2** 5 **1** 3)

Na temelju 4. koraka dekrementira se j, pa je **j = 3**. S obzirom da je j > 1 ponavlja se 2. korak.

Na temelju 2. koraka ponovno se generira broj U, npr 0.441

Na temelju 3. koraka računa se vrijednost k, odnosno k = [3 * 0.441] + 1 = [1.323] + 1, tj. **k = 2**. Vrši se zamjena elemenata na pozicijama k i j (k = 2, j = 3). Odnosno dolazi do zamjene između brojeva 2 i 5. Niz sada izgleda ovako:

(4 **5** **2** 1 3)

Na temelju 4. koraka dekrementira se j, pa je **j = 2**. S obzirom da je j > 1 ponavlja se 2. korak.

Na temelju 2. koraka ponovno se generira broj U, npr 0.654

Na temelju 3. koraka računa se vrijednost k, odnosno k = [2 * 0.654] + 1 = [1.308] + 1, tj. **k = 2**. Vrši se zamjena elemenata na pozicijama k i j (k = 2, j = 2). Odnosno ne dolazi do zamjene jer su k i j jednaki. Niz ostaje isti:

(4 **5** 2 1 3)

Na temelju 4. koraka dekrementira se j, pa je **j = 1**. S obzirom da je j = 1. Algoritam ovdje staje i više nije potrebno ponavljati 2. korak.

Konačna nasumična permutacija niza izgleda ovako:

(4 5 2 1 3)

2.3. Izračun složenosti Fisher-Yates algoritma (karakteristična jednadžba)

Složenost Fisher-Yates algoritma koja se temelji na modernoj metodi može se izračunati preko karakteristične jednadžbe s obzirom da se ovaj algoritam može zapisati i u rekurzivnom obliku. Zapravo je riječ o nehomogenoj rekurzivnoj funkciji. Prvo ćemo definirati pseudokod za tu rekurzicnu funkciju:

```
Fisher-Yates(ar,N) {
    i = cijeli broj uniformno izabran između intervala [0,N]
    zamijeniti ar[N-1] sa ar[i]
    if (N>2)
        Fisher-Yates (ar,N-1)
}
```

$$T_{max}^{FY}(n) = c_1 + c_2 + T_{max}^{FY}(n-1)$$

Ovu jednadžu možemo skratiti tako da je $c_1+c_2 = c$ te da uvedemo malo drugačiju simboliku za T oznake.

$$a_n = c + a_{n-1}$$

Ovu nehomogenu rekurziju moramo svesti na homogenu, pa ćemo to i napraviti.

$$a_n - a_{n-1} = c$$

$$a_{n-1} - a_{n-2} = c/(-1)$$

$$\overline{\phantom{a_n - 2a_{n-1} + a_{n-2} = 0}}$$

$$a_n - 2a_{n-1} + a_{n-2} = 0$$

Karakteristična jednadžba ove rekurzije je:

$$x^2 - 2x + 1 = 0$$

Odnosno bolje zapisano:

$$(x-1)^2 = 0$$

Vidimo da ova karakteristična jednadžba ima višestruke korijene (dvostruki korijen jednadžbe), tj. rješenje ove jednadžbe je

$$x_{1,2} = 1$$

Opće rješenje ove rekurzije je:

$$a_n = C_1 \cdot \ 1^n + \ C_2 \cdot n \cdot 1^n$$

Odnosno, kraće zapisano

$$a_n = C_1 + C_2 \cdot n$$

Iz posljednje jednadžbe može se zaključiti sljedeće

$$T_{max}^{FY}(n) = O(n)$$

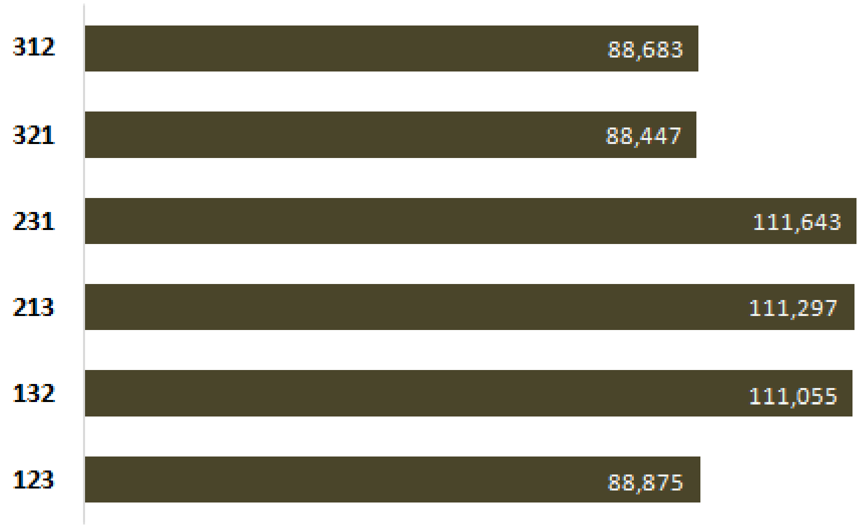
2.4. Odnos "naivnih" algoritama i Fisher - Yates algoritma

Naivni algoritmi za početnike u programiranju predstavljaju veliki problem. To su algoritmi koje programeri naivno prihvaćaju i koriste jer algoritam radi, no pravo pitanje koje proizlazi iz toga, predstavlja li taj algoritam uistinu ono što treba programeru, ili postoji nešto bolje.

Naivni algoritmi predstavljaju jednostavno rješenje problema, jednostavno se obrađuju i lagano implementiraju, ali problem je što troše puno više resursa kao što je vrijeme, prostor, memorija, pristup, itd. Primjer jednog takvog naivnog algoritma je mjehuričasto sortiranje (engl: "buble sort") koje je lagano za razumijevanje, ali ima složenost O(n^2). A zamjena za takav algoritam predstavlja npr. "quicksort" sa složenošću O(n * log n). (Prema: Atwood, 2007).

Ovaj odnos moguće je objasniti između "shuffle" algoritama kao što je to npr. Fisher - Yates algoritam, odnosno on predstavlja rješenje za takozvane "naivne shuffle" algoritme.

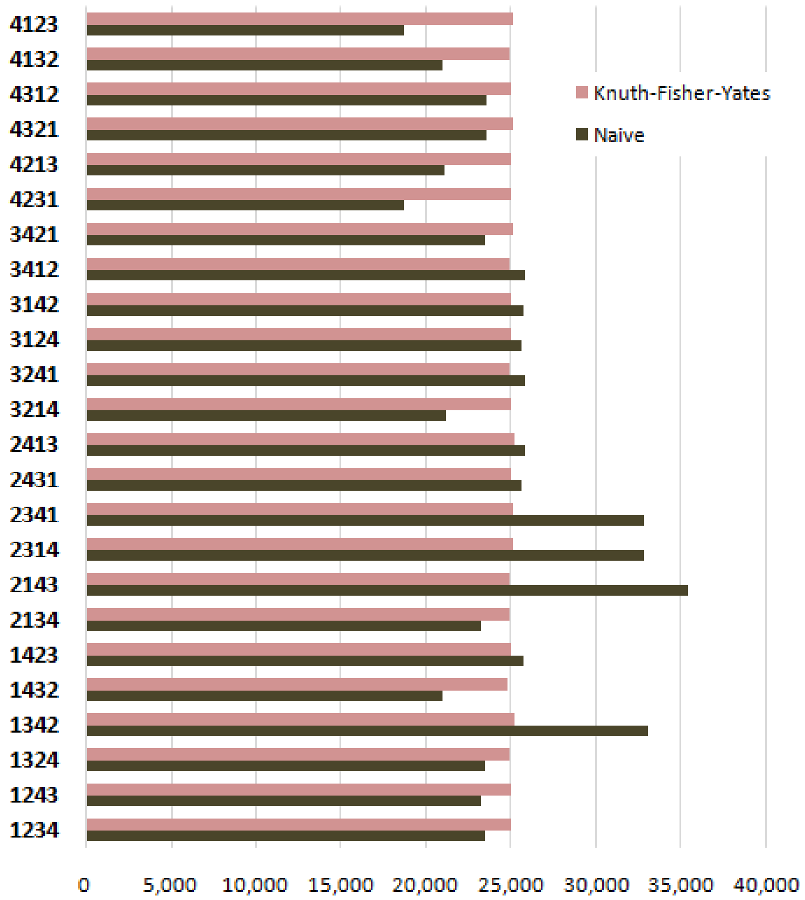
To je moguće utvrditi i na temelju primjera. Potrebno je promiješati niz od 3 elementa. Da bi se mogao utvrditi odnos, miješanje je potrebno izvesti više puta. Npr. neka je broj miješanja 600.000. Zahvaljujući permutacijama za niz od 3 elementa moguće je 3! odnosno 6 permutacija toga niza. (Prema: Atwood, 2007). Slika 1. prikazuje grafički prikaz permutacija u odnosu na broj miješanja, tj. koliko puta je pogođena određena permutacija.



Slika 1. *Rezultat miješanja pomoću naivnog "shuffle" algoritma* (Izvor: http://blog.codinghorror.com/the-danger-of-naivete)

Iz slike se jasno vidi da permutacije (2 3 1), (2 1 3) i (1 3 2) iskaču iz prosjeka, odnosno pogođene su više puta unutar 600.000 miješanja niza. To ukazuje na činjenicu da je riječ o naivnom "shuffle" algoritmu koji je neuravnotežen (engl: "biased") i u pravilu neprikladan. Problem je u tome što naivni "shuffle" algoritam u ovoj situaciji mogu prikazati 3^3 odnosno 27 mogućih kombinacija permutacija , a matematički postoji samo 6 permutacija u nizu od 3 elementa. Tj. neke permutacije se ponavljaju jer se zamjena elemenata prilikom miješanja uvijek vrši na temelju početne duljine niza. Matematički gledano 27 nije djeljivo sa 6 i upravo iz toga razloga neke će permutacije biti pogođene više puta. Fisher - Yates uklanja taj problem jer on vrši zamjenu na takav način što u 1. iteraciji 3.element zamjenjuje u odnosu na 3 elementa, u 2. iteraciji 2. element u odnosu na 2 elementa, itd. (Prema: Atwood, 2007; Stackoverflow, 2009). Može se reći da se zamjena elemata unutar algoritma vrši na temelju imaginarne veličine niza koja se u svakoj iteraciji dekrementira za 1.

Moguće je napraviti usporedbu sa nizom koji se sastoji od 4 elementa i 600.000 miješanja. Slika 2. grafički prikazuje taj odnos. Fisher - Yates algoritam je svaku permutaciju pogodio 25.000 puta. To proizlazi iz činjenice da je 600.000 miješanja djeljivo sa 24 permutacije što daje 25.000, a taj broj prestavlja koliko je puta pogođena svaka permutacija. Naivni "shuffle" algortam je daleko neefikasniji po tom pitanju, tj. neuravnoteženo ispreplića niz pa su neke permutacije pogođene manje od 25.000 puta , a neke više. Stvar postaje još gora za veći broj elemenata i veći broj miješanja. Upravo zbog toga uporaba Fisher-Yates algoritma za ispreplitanje nizova je ispravan odabir u odnosu na ostale algoritme istoga tipa jer je vjerojatnost da ćete pogoditi određenu permutaciju jednaka i za sve ostale permutacije.



Slika 2. *Usporedba Fisher-Yates algoritma i "naivnog" shuffle algoritma* (Izvor: http://blog.codinghorror.com/the-danger-of-naivete)

Napravimo uspredbu na temelju programske implementacije naivnog algoritma i Fisher - Yates algoritma. Potrebno je obraditi pozornost na uvjet petlje i način odabira nasumične pozicije niza.

"*Fisher - Yates algoritam*"

```
for (int i = niz.Length - 1; i > 0; i--) {
	int n = rand.Next(i + 1);
	Swap(ref niz[i], ref niz[n]);
}
```

"*Naivni shuffle algoritam*"

```
for (int i = 0; i < niz.Length; i++) {
	int n = rand.Next(niz.Length);
	Swap(ref niz[i], ref niz[n]);
}
```

Ovaj programski kod može se objasniti na primjeru niza sa 3 elementa.

	Fisher - Yates algoritam			"Naivni" shuffle algoritam		
Redni broj koraka	1. korak	2. korak	3. korak	1. korak	2. korak	3, korak
Permutacija	(1,2,3)	(1,2,3)	(1,2,3)	(1,2,3)	(2,1,3)	(3,1,2), (2,3,1), (2,1,3)
		(2,1,3)	(2,1,3)		(1,2,3)	(3,2,1), (1,3,2), (1,2,3)
					(1,3,2)	(2,3,1), (1,2,3), (1,3,2)
Permutacija	(1,3,2)	(1,3,2)	(1,3,2)	(2,1,3)	(1,2,3)	(3,2,1), (1,3,2), (1,2,3)
		(3,1,2)	(3,1,2)		(2,1,3)	(3,1,2), (2,3,1), (2,1,3)
					(2,3,1)	(1,32,), (2,1,3), (2,3,1)
Permutacija	(3,2,1)	(3,2,1) (2,3,1)	(3,2,1) (2,3,1)	(3,2,1)	(231)	132, 213, 231
					(3,2,1)	123, 312, 321
					(3,1,2)	213, 321, 312

Tablica 1. *Dobivanje permutacija za niz od 3 elementa na sve moguće načine u algoritmu*

"Naivni" shuffle algoritam će u svakom koraku petlje (itereacije) za svaku permutaciju imati mogućnost zamjene elementa na 3 načina. To ukupno daje 3 * 3 * 3 = 27 kombinacija, što je puno više u odnosu na mogućih 6 permutacija koje se mogu postići sa nizom od 3 elementa. Fisher - Yates algoritam će u 1. koraku iteracije imati mogućnost zamjene elemanta na 3 načina. Svaki od načina dobivenih u 1. koraku se u 2. koraku iteracije može izvršiti na 2 načina, a u 3. koraku svaki od načina dobivenih u 2. koraku samo na 1 način što daje ukupno 3 * 2 * 1 = 3! = 6 mogućih kombinacija.

Recimo da smo u prvom koraku pogodili permutaciju (1,3,2) kao što je prikazano u tablici (došlo je do zamjene elemanata na 3. i 2. poziciji, a treća pozicija se više ne promatra u nastavku algoritma). Znači dvojka ostaje na 3. poziciji. Sada element na 2. poziciji možemo zamijeniti sa elementom na 1. poziciji ili da uopće ne dođe do zamjene, to su jedine mogućnosti. Pretpostavimo da smo u 2. koraku pogodili permutaciju (3,1,2). Ostao je ješ samo jedan korak s o bzirom da se promatra samo jedan element, tj. element na 1. poziciji i njega više nemamo sa kime mijenjati te je nasumično dobicena permutacija (3,1,2).

Fisher - Yates algoritam će bez obzira na koji način bira permutaciju, sa istom vjerojatnošću proizvesti nasumičnu permutaciju iz početnoga niza elemenata. (Prema: Atwood, 2007; Nedrich, 2014)

2.5. Problem uporabe algoritama sortiranja za ispreplitanje nizova

Ispreplitanje nizova, odnosno konstrukcija nasumične permutacije niza osim Fisher - Yates algoritma mogla bi se također implementirati pomoću algoritama za sortiranje. U samoj praksi to baš ne predstavlja dobro rješenje. Na temelju primjera moguće je i potvrditi tu pretpostavku.

Prvi primjer je pokušaj miješanja uz pomoć funkcija *Array.sort()* iz prog. jezika Javascript i funckije *Math.round(Math.random()*2)-1* koja služi kao funckija uspredbe.

Start shuffling Array using `Array.sort()` and `Math.round(Math.random()*2)-1` comparison function.

Reset

Iterations	Column represents position of letters after a shuffle									
100	**C**	**A**	**D**	**E**	**B**	**F**	**G**	**H**	**I**	**J**
Letters	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
A	71	17	9	1	2	0	0	0	0	0
B	21	56	17	4	1	0	1	0	0	0
C	6	19	48	18	5	4	0	0	0	0
D	2	4	20	40	23	8	2	1	0	0
E	0	4	5	26	44	20	1	0	0	0
F	0	0	1	7	21	49	16	4	2	0
G	0	0	0	4	1	16	55	18	5	1
H	0	0	0	0	3	2	20	56	12	7
I	0	0	0	0	0	0	2	16	63	19
J	0	0	0	0	0	1	3	5	18	73

Slika 3. *Učestalost pojavljivanja slova na određenom indeksu niza nakon 100 iteracija* (Izvor: http://sroucheray.org/blog/2009/11/array-sort-should-not-be-used-to-shuffle-an-array)

Iz početnoga niza (A B C D E F G H I J) se nakon pokretanja funkcije 100, puta, tj. miješanja elemanta unutar niza, odnosno ispreplitanja niza, dobila permutacija (C A D E B F G H I J). Slika 3. predstavlja dvodimenzionalnu matricu elemenata niza (u ovom slučaju slova) i indeksa niza. Iz nje je moguće uočiti da je element niza ostao u najvećem broju slučaja na istom mjestu ili na susjednom. Na matrici je to vidljivo po tome što se najveće brojke nalaze na njenoj dijagonali ili na susjednim indeksima. Razlog tomu prije svega je taj što funkcija *Array.sort()* koristi algoritam mjehuričastog sortiranja. Unutar matrice vidljiv je velik broj nula što znači da niz nije dobro izmješan ni nakon 100 miješanja i to je pokazatelj da ovaj algoritam nije dobar za korištenje u praksi. (Prema: Lambda, 2009)

Prethodni algoritam je moguće poboljšati uz korištenje bolje komparacijske funkcije, npr. *Math.round(Math.random())*.

Start shuffling Array using `Array.sort()` and `Math.round(Math.random())` comparison function.

Reset

Iterations: 100

Column represents position of letters after a shuffle

	D	C	A	E	B	F	H	G	I	J
Letters	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
A	26	31	22	7	4	6	3	1	0	0
B	27	27	24	8	9	3	1	0	0	1
C	21	26	18	12	11	7	1	1	1	2
D	11	6	22	28	10	9	7	1	5	1
E	11	6	6	25	20	14	8	7	2	1
F	2	3	4	13	23	24	17	7	3	4
G	2	1	1	5	11	18	26	21	7	8
H	0	0	2	1	7	10	23	29	20	8
I	0	0	1	0	3	6	10	22	34	24
J	0	0	0	1	2	3	4	11	28	51

Slika 4. *Učestalost pojavljivanja slova na određenom indeksu niza nakon 100 iteracija* (Izvor: http://sroucheray.org/blog/2009/11/array-sort-should-not-be-used-to-shuffle-an-array)

Iz početnoga niza, tj. (A B C D E F G H I J), se nakon pokretanja funkcije 100 puta, tj. miješanja elemanta unutar niza, odnosno ispreplitanja niza, dobila permutacija (D C A E B F H G I J). Dobiven je niz koji je u odnosu na početni niz puno bolje izmješan nego što je to bio slučaj sa prethodnim algoritmom. Funkcija za usporedbu sada vraća dvije vrijednosti pa je šansa da će doći do zamjene slova puno veća u odnosu na prethodnu komparacijsku funkciju koja je vraćala tri vrijednosti. Kompacijska funkcija je pridonijela Array.sort() funkciji, ali ne sa značajnim poboljšanjima. I dalje je većina pojavljivanja elemanata na istom indeksu kao i na početku ili na susjednim indeksima uz pojedini specijalni slučaj koji je omogućio malo bolju raspodjelu i ravnotežu između elemanta na dijagonali i oko nje. (Prema: Lambda, 2009)

Vidljivo je da Array.sort() i slične funkcije nije dobro koristiti za miješanje elemenata niza iako je implementacija moguća. Konačni niz takvoga ispreplitanja je jako loš, tj. sličan je početnom nizu i to u svakom slučaju ne predstavlja dobro rješenje. Eventualno se može primjenjivati za jako mali broj elemenata i jako mali broj miješanja.

Rješenje, kao i u svim usporedbama do sada je Fisher - Yates algoritam. Slika 5. koja predstavlja matrični prikaz testiranog primjera ukazuje kako je ovaj algoritam najbolje rješenje za ispreplitanje nizova, odnosno mješanje elemanta niza.

Start shuffling Array using Fisher-Yates algorithm.

Reset

Iterations: 100

Column represents position of letters after a shuffle

	D	**F**	**B**	**E**	**I**	**A**	**H**	**G**	**J**	**C**
Letters	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
A	11	10	10	7	9	12	14	12	9	6
B	10	9	13	6	12	12	14	8	8	8
C	15	9	6	10	10	8	9	12	6	15
D	13	7	8	14	9	10	9	7	13	10
E	7	15	8	12	8	7	10	8	13	12
F	9	8	9	16	14	13	7	9	10	5
G	9	10	8	12	4	6	9	14	13	15
H	11	13	15	11	5	11	12	7	7	8
I	4	9	10	7	18	12	8	7	11	14
J	11	10	13	5	11	9	8	16	10	7

Slika 5. *Učestalost pojavljivanja slova na određenom indeksu niza nakon 100 iteracija (Fisher - Yates algoritam)* (Izvor: http://sroucheray.org/blog/2009/11/array-sort-should-not-be-used-to-shuffle-an-array)

Fisher - Yates algoritam ravnomjerno nakon 100 prolazaka, tj. mješanja raspoređuje slova (elemente niza) ravnomjerno, tj uravnoteženo po matrici. Nema nijedne vrijednosti 0 u matrici što predstavlja dobar pokazatelj efikasnosti i preciznosti samoga algoritma u odnosu na ostale algoritme istoga tipa, upravo zahvaljujući uniformnoj distribuciji, tj.raspodjeli elemenata niza.

2.6. Implementacija Fisher - Yates algoritma u programskom jeziku C++

U nastavku je priložena jednostavna implementacija Fisher - Yates algoritma u programskom jeziku C++ (Slika 6.) uz sliku ekrana izvršavanja programa (Slika 7.).

Slika 6. (ispod) predstavlja implementaciju Fisher-Yates algoritma u programskom jeziku C++ koja je u ovom slučaju izvedena na vrlo jednostavan način. Na početku je unutar samoga programskoga koda inicijaliziran niz brojeva. U ovom slučaju riječ je o nizu od prvih 5 brojeva. Na temelju duljine niza ispisuje se početni niz. Zatim slijedi petlja koje predstavlja osnovu algoritma, tj. vrši se ispreplitanje niza, miješanje njegovih elemanata i na kraju se ispisuje isti niz na temelju duljine niza u novom, nasumičnom poretku elemenata. Sam algoritam je jednostavan za implementaciju i u bilo kojem drugom programskom jeziku prije svega zbog svoje jednostavnosti. Također program je modificiran kako bi se prikazalo izvođenje algoritma korak po korak. Na slici 7. moguće je vidjeti rezultat pokretanja implementacije. Moguće je uočiti da je rezultat izvođenja algoritma relevantan i ispravan, tj. uspješno je dobivena nasumična permutacija niza prvih 5 brojeva.

```cpp
#include <iostream>
#include <cmath>
#include <ctime>
#include <cstdlib>
#include <conio.h>
using namespace std;
int main() {
    srand(time(0));
    cout << "Fisher - Yates algoritam (Shuffle): " << endl;
    cout << "------------------------------------" << endl;
    int niz[] = {1,2,3,4,5};
    int duljina_niza = 5;
    int temp, k;
    cout << "Pocetni niz: ";
    for (int i = 0; i < duljina_niza; i++)
        cout << niz[i] << " ";
    cout << endl << endl;

    for (int i = duljina_niza-1; i > 0; i--) {
        //system("cls");
        k = rand () % i;
        cout << "Trenutno stanje niza: ";
        for (int j = 0; j < duljina_niza; j++)
            cout << niz[j] << " ";
        cout << endl << endl;
        cout << duljina_niza-i << ". korak: ";
        cout << "Izabran je element na poziciji " << k+1;
        temp = niz[i];
        niz[i] = niz[k];
        niz[k] = temp;
        cout << " -> Zamjena elemenata " << niz[i] << " i " << niz[k] << endl;
        cout << endl;
        cout << "Novo stanje niza: ";
        for (int j = 0; j < duljina_niza;j++)
            cout << niz[j] << " ";
        cout << endl << endl;
        cout << "ENTER ZA NASTAVAK";
        getch();
        cout << endl << endl;
    }
    cout << "Nakon mijesanja: ";
    for (int i = 0; i < duljina_niza; i++) {
        cout << niz[i] << " ";
    }
    return 0;
}
```

Slika 6. Implementacija Fisher - Yates algoritma u programskom jeziku C++

```
Fisher - Yates algoritam (Shuffle):
-----------------------------------
Pocetni niz: 1 2 3 4 5

Trenutno stanje niza: 1 2 3 4 5

1. korak: Izabran je element na poziciji 2 -> Zamjena elemenata 2 i 5

Novo stanje niza: 1 5 3 4 2

ENTER ZA NASTAVAK

Trenutno stanje niza: 1 5 3 4 2

2. korak: Izabran je element na poziciji 1 -> Zamjena elemenata 1 i 4

Novo stanje niza: 4 5 3 1 2

ENTER ZA NASTAVAK

Trenutno stanje niza: 4 5 3 1 2

3. korak: Izabran je element na poziciji 2 -> Zamjena elemenata 5 i 3

Novo stanje niza: 4 3 5 1 2

ENTER ZA NASTAVAK

Trenutno stanje niza: 4 3 5 1 2

4. korak: Izabran je element na poziciji 1 -> Zamjena elemenata 4 i 3

Novo stanje niza: 3 4 5 1 2

ENTER ZA NASTAVAK

Nakon mijesanja: 3 4 5 1 2
--------------------------------
Process exited after 4.658 seconds with return value 0
Press any key to continue . . .
```

Slika 7. Implementacija Fisher - Yates algoritma

Može se primjetiti, na slici 7, da se algoritam izvodi korak po korak. Ispisuje se trenutno stanje niza, prije zamjene elemenata. Izvodi se zamjena prema Modernoj metodi Fisher-Yatesa algoritma, tako što se nasumično odabrani element niza zamijeni sa zadnjim elementom u nizu te se taj posljedni element više ne uzima u obzir već se niz smanjuje za 1 i procedura se ponavlja. Naravno uz zamjenu elemata prikazuje se i novo stanje niza sa obzirom na trenutno, kako bi se uočila razlika.

3. ZAKLJUČAK

Fisher-Yates algoritam pokazao se kao odlično rješenje za problem ispreplitanja nizova, tj. mješanje elemenata u nekom redu što se zapravo svodi na biranje nasumične permutacije. Riječ je o metodi koja svojom jednostavnošću, lakoćom implemetacije, ali prije svega preciznošću je pravi primjer korištenja ispravnih algoritama.Zanimljiva činjenica je ta, da je spomenuti algoritam proizašao iz znanstvenoga rada koji se bavio tematikom statističkih tablica za biološka, agrokulturna i medicinska istraživanja, a danas se njegove varijante mogu prepoznati u različitim područjima računalne znanosti. Na primjer koriste ga neki kriptografski algoritmi, koristi se za različite vrste enkripcija (npr. enkripicija slika), koristi se u mrežama. Velika primjenu ovoga algoritma pokazala se u igrama na sreću (posebice se odnosi na kartaške igre). Riječ je o algoritmu koji je uspješno razvijan kroz povijest i primjer algoritma na kojem je radilo više znanstvenika iz područja računalnih znanosti. Primjer je algoritam Sandre Sattolo (Sattolo's algorithm). Također postoje i neke druge implementacije koje pokušavaju smanjiti složenost ovoga algoritma, ali u najgorem slučaju ipak daju složenost od O (n^2). Upravo detalji koji specifiraju sam algoritam predstavljaju raznolikost i korisnost poznavanja svijeta algoritama u svim svojim postojećim oblicima.

4. LITERATURA

[1] Atwood J. (2007). *The Danger of Naïveté*. Dostupno 7. prosinca 2007. sa http://blog.codinghorror.com/the-danger-of-naivete/

[2] Eli's Benderskys website (2010). *The intuition behind Fisher-Yates shuffling*. Dostupno 28. svibnja 2010. sa http://eli.thegreenplace.net/2010/05/28/the-intuition-behind-fisher-yates-shuffling/

[3] GeeksforGeeks (2014). *Shuffle a given array*. Preuzeto 4. siječnja 2016. http://www.geeksforgeeks.org/shuffle-a-given-array/

[4] Knuth D.E. (1997). The *Art of Computer Programming, Volume 2 / Seminumerical Algorithms* (3. izdanje), (145,156).

[5] Lambda (2009). *Array.sort() should not be used to shuffle an array*. Dostupno od 16. studenoga 2009. sa http://sroucheray.org/blog/2009/11/array-sort-should-not-be-used-to-shuffle-an-array/

[6] Mička P. (2015). *Fisher-Yates shuffle*. Preuzeto 4. siječnja 2016. s http://en.algoritmy.net/article/43676/Fisher-Yates-shuffle

[7] Nedrich M. (2014). *Fisher-Yates Shuffle – An Algorithm Every Developer Should Know*. Dostupno od 11. kolovoza 2014. sa http://spin.atomicobject.com/2014/08/11/fisher-yates-shuffle-randomization-algorithm/

[8] Stackoverflow (2009). *why does this simple shuffle algorithm produce biased results? what is a simple reason?* Dostupno 13. svibnja 2009. sa http://stackoverflow.com/questions/859253/why-does-this-simple-shuffle-algorithm-produce-biased-results-what-is-a-simple%20%282nd%20answer%29

[9] Wikipedia (2014). *Fisher–Yates shuffle*. Preuzeto 4. siječnja 2016. s http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle