Vizualizacija algoritama sortiranja za C++

Kovač, Marijan

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Marijan Kovač

VIZUALIZACIJA ALGORITAMA SORTIRANJA ZA C++

ZAVRŠNI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Marijan Kovač

JMBAG: 0016142261

Studij: Informacijski sustavi

VIZUALIZACIJA ALGORITAMA SORTIRANJA ZA C++

ZAVRŠNI RAD

Mentor:

Doc. dr. sc. Robert Kudelić

Varaždin, rujan 2022.

Marijan Kovač

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Rad obuhvaća pregled temeljnih algoritama za sortiranje te vizualiziranje njihova rada. U tu svrhu izrađeno je programsko rješenje za vizualni prikaz algoritama sortiranja u programskom jeziku C++, bazirano na Qt alatima, koje predstavlja temelj ovoga rada. Programsko rješenje podijeljeno je u dvije cjeline – vizualizacija i usporedba algoritama. Dio vizualizacije omogućava generiranje niza brojeva različite duljine i oblika te vizualiziranje sortiranja generiranoga niza jednim od ponuđenih algoritama. Dio usporedbe algoritama također omogućava generiranje niza, ali većih duljina te nudi mjerenje performansi algoritama bez vizualizacije kako bi se dočarala njihova učinkovitost u stvarnom okruženju. Glavni cilj rada je na vizualan način olakšati razumijevanje osnovnih algoritama za sortiranje te samog pojma sortiranja kao jednog od najvažnijih problema u računarstvu uopće.

Ključne riječi: algoritam; sortiranje; vizualizacija; programiranje; C++; Qt; performanse; statistika; usporedba

Sadržaj

1. Uvod

Sortiranje je proces permutiranja niza vrijednosti ili objekata kako bi isti bili poredani nekim logičnim redoslijedom. [1] Problem sortiranja pojavio se pojavom prvih računala te je odmah privukao mnogo istraživanja, ponajviše zbog složenosti njegova učinkovita rješavanja, unatoč tomu što se možda na prvu čini kao jednostavan problem. [2] Daljnjim razvojem računala, kao i napretkom informacijskih tehnologija, došlo je do gomilanja podataka, a prvi korak u njihovu organiziranju je upravo sortiranje. Rješenje problema sortiranja leži u algoritmima za sortiranje, čije implementacije imaju svi računalni sustavi današnjice, bilo za korištenje od strane samih sustava ili njihovih korisnika. [1]

Postoji mnogo različitih algoritama za sortiranje. Ipak, većina njih su različite modifikacije temeljnih algoritama, nastale s ciljem da se postojećim algoritmima poboljšaju učinkovitost i performanse. Stoga ćemo se u ovom radu fokusirati na one jednostavne i temeljne algoritme sortiranja, koji predstavljaju osnovu za shvaćanje i ostalih, mnogo kompleksnijih algoritama. Ti algoritmi su redom – sortiranje izborom, sortiranje umetanjem, sortiranje zamjenom, mjehuričasto sortiranje te dvostruko mjehuričasto sortiranje.

Glavna tema ovoga rada je vizualizirati navedene algoritme sortiranja, što je postignuto izradom programskog rješenja koje sadrži njihove implementacije, prilikom čijeg izvršavanja se vizualizira njihov rad. Osim toga, programsko rješenje omogućava i vizualni prikaz mjerenja performansi navedenih algoritama, kako bi se jasnije mogla vidjeti njihova različitost.

Ova tema je značajna ponajprije iz razloga što algoritmi, zbog svoje apstraktne prirode, često nisu lako razumljivi, posebice onima koji se tek susreću s programiranjem, a osim toga, algoritmi za sortiranje najčešće su polazna točka u učenju programiranja. Stoga je cilj ovoga rada na vizualan način demistificirati rad algoritama za sortiranje, kako bi se bolje shvatio njihov princip, ali i učinkovitost u različitim problemima koje moraju riješiti.

Moja motivacija za odabir ove teme javila se još za vrijeme polaganja kolegija Programiranje 1, upravo iz prethodno opisanih razloga. Naime, tada sam primijetio kako je dosta teško pokušavati razumjeti algoritam na način da se gleda u kod i pokušava u glavi „vrtjeti“ brojeve. To me najprije potaknulo da skiciram polje i bilježim stanje nakon svakog prolaza. Međutim, čak i u tom slučaju vrlo lako se izgubi „nit vodilja“, nakon čega se rodila ideja za vizualiziranjem promjena stanja u svakom koraku.

2. Metode i tehnike rada

Za izradu programskog rješenja korišten je jezik C++ u sklopu Qt (izgovara se kao eng. *cute*) razvojnog okvira, koji sadrži čitav niz alata za pojednostavljeno stvaranje aplikacija i korisničkih sučelja za različite platforme – *desktop*, ugradbene ili pak mobilne. Karakterizira ga višeplatformnost, zbog toga što aplikacije koje su upogonjene istim, rade na različitim *softverskim* i *hardverskim* platformama poput Linux-a, Windows-a, macOS-a, Android-a i ostalima, uz male do nikakve promjene u osnovnome kodu, bez utjecaja na mogućnosti, brzinu i korisničko iskustvo. [3], [4]

Qt koriste brojne poznate *open-source* i višeplatformske aplikacije kao što su Autodesk, Monero, OBS, qBittorent, Scribus, Teamviewer, VirtualBox, VLC media player, Wireshark i tako dalje. Osim toga, koriste ga mnogobrojne kompanije, a neke od poznatijih su BMW, Electronic Arts, LG, Microsoft, Samsung, Phillips, Tesla, Volvo, HP, Valve i slično. [4]

Kako bih se upoznao sa radom u alatu Qt, koristio sam njegovu službenu dokumentaciju, integriranu u sam *Qt Creator*. Ipak, sama dokumentacija ponekad nije bila dovoljna za razumijevanje karakterističnih potreba, tako da su dodatno korišteni različiti web-izvori poput platforme Udemy, Youtube servisa, Stack Overflow-a, Qt Foruma i druge.

3. Algoritmi sortiranja

U ovome će se poglavlju najprije proanalizirati već ranije spomenuti algoritmi sortiranja, prilikom čega će se koristiti tehnika bilježenja stanja polja u svakom prolazu, tj. iteraciji algoritma.

Kao primjer ćemo uzeti niz od 7 nasumičnih brojeva, koji su u rasponu od 1 do 50. Početno stanje polja izgledalo bi ovako (prema [5]):

Tablica 1: Početno stanje polja

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
17	31	3	43	11	24	8

Navedeni ćemo primjer koristiti za sve algoritme koje ćemo analizirati.

3.1. Sortiranje izborom

Sortiranje izborom jedan je od najjednostavnijih algoritama za sortiranje, a radi na način da se u polju pronađe element s najvećom vrijednošću, te da se vrijednost tog elementa zamijeni sa vrijednošću posljednjeg elementa u polju. To znači da će nakon prvog prolaza u posljednjem elementu polja biti element s najvećom vrijednošću, te da taj element više ne trebamo razmatrati. Postupak se ponavlja sve dok ne preostane zadnji element koji će na kraju zapravo ostati sortiran, što znači da je za sortiranje potrebno n – 1 prolazaka. [5]

Implementacija ovog algoritma u jeziku C++ izgledala bi ovako (prema [7]):

```
void swap(int *x, int *y){
      int tmp = *x;
      *x = *y;
      *y = tmp;
}

void SelectionSort (int A[], int N){
      for(int i = N-1; i > 0; i--){
            int max = 0;
            for(int j = 1; j <= i; j++)
                  if(A[j] > A[max]) max = j;
            swap(&A[i], &A[max]);
      }
}
```

Pogledajmo sada što se događa u svakom koraku ovoga algoritma (Tablica 2).

Tablica 2: Primjer sortiranja izborom

i	max	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
6	3	17	31	3	43	11	24	8
5	1	17	31	3	8	11	24	43
4	1	17	24	3	8	11	31	43
3	0	17	11	3	8	24	31	43
2	1	8	11	3	17	24	31	43
1	0	8	3	11	17	24	31	43
0	-	3	8	11	17	24	31	43

Svaki redak tablice predstavlja jedan korak **vanjske** petlje algoritma. U svakom se koraku bilježi trenutna pozicija *i*, pozicija najvećeg elementa *max*, te stanje u elementima polja **prije** nego je došlo do zamjene. Dodatno, zadnji element koji promatramo označit ćemo crvenom bojom, element s najvećom vrijednošću zelenom, dok ćemo sortirani dio polja označiti žutom bojom.

U prvoj iteraciji, najveća vrijednost nalazi se u elementu na poziciji 3. Poziciju tog elementa pronalazi unutarnja petlja algoritma, te se nakon njenog završetka vrši zamjena pronađenog elementa s najvećom vrijednošću i zadnjeg promatranog elementa polja. Taj zadnji element polja određuje vanjska petlja, koja ujedno predstavlja i trenutni korak.

Nakon prve iteracije, u zadnjem elementu polja nalazi se element s najvećom vrijednošću, što znači da u idućim koracima taj element više ne razmatramo. Na taj način ovaj algoritam zapravo razdvaja polje na **sortirani** i **nesortirani** dio. Postupak se, dakle, ponavlja, na način da se u svakom idućem koraku element s najvećom vrijednošću traži samo u **nesortiranom** dijelu, sve dok ne preostane samo jedan element.

Pogledajmo još stanje nakon zadnje iteracije, tj. konačno stanje. Možemo primijetiti kako je preostali element na poziciji 0 zapravo već sortiran s obzirom na sortirani dio polja.

Što se tiče složenosti, ovaj je algoritam poprilično neučinkovit. Naime, u prvom prolasku potrebno je n – 1 usporedbi, dok se u svakom idućem prolasku smanjuje za 1, što znači da je ukupan broj usporedbi n(n – 1)/2. Osim toga, u svakom se prolasku vrši i zamjena elemenata, tako da imamo još dodatnih 3(n – 1) operacija pridruživanja. Dakle, složenost ovog algoritma je $n(n – 1)/2 + 3(n – 1) = O(n^2)$. [5]

Osim velike vremenske složenosti, nedostatci ovog algoritma su i obavljanje iste količinu posla bez obzira na početno stanje polja te nepotrebno vršenje zamjene elemenata u slučaju da je element na zadnje promatranoj poziciji najveći. [5]

3.2. Sortiranje zamjenom

Ovaj je jednostavni algoritam, po svojoj ideji, vrlo sličan prethodnome algoritmu. Naime, i ovom se algoritmu vrijednost posljednjeg elementa zamjenjuje s vrijednošću najvećeg elementa, ali ovaj puta čim se takav element pronađe. Na taj će se način u zadnjem elementu polja, nakon prvog prolaska, također nalaziti element s najvećom vrijednošću, kao što je to slučaj kod sortiranja izborom. Postupak se tako ponavlja sve dok ne preostane zadnji nesortirani element.

Implementaciju ovog algoritma u jeziku C++ možemo izvesti na sljedeći način (prema [7]):

```
void swap(int *x, int *y){
      int tmp = *x;
      *x = *y;
      *y = tmp;
}

void ExchangeSort (int A[], int N){
      for(int i = N-1; i > 0; i--){
            for(int j = 0; j < i; j++)
                  if(A[j] > A[i]) swap(&A[j], &A[i]);

      }
}
```

Da bismo bolje razumjeli kako ovaj algoritam radi, pokazat ćemo primjer sortiranja niza na istom primjeru kao u prethodnome algoritmu (Tablica 1). S obzirom da algoritam vrši više zamjena u jednom prolazu, ovaj ćemo put prikazati promjene stanja samo u njegovom prvom prolazu.

Tablica 3: Primjer prvog prolaza kod sortiranja zamjenom

i	j	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
6	0	17	31	3	43	11	24	8
6	1	8	31	3	43	11	24	17
6	2	8	17	3	43	11	24	31
6	3	8	17	3	43	11	24	31
6	4	8	17	3	31	11	24	43
6	5	8	17	3	31	11	24	43

U Tablici 3 svaki redak predstavlja jedan korak **unutarnje** petlje algoritma. U svakom koraku bilježimo poziciju *j* te stanja u elementima polja, i to **prije** zamjene elemenata. Kako bismo lakše uočili promjene, zelenom su bojom istaknuti elementi s najvećom vrijednošću, dok je zadnji promatrani element istaknut crvenom bojom.

Svaka unutarnja iteracija ovoga algoritma provjerava je li element na trenutnoj poziciji veći od zadnje promatranog elementa, pa će tako već u prvoj unutarnjoj iteraciji ovoga primjera vrijednost elementa na poziciji 0 biti veća od zadnje promatranog elementa na poziciji 6, te će se odmah izvršiti zamjena ovih dvaju elemenata.

Nakon prve unutarnje iteracije, provedena je zamjena elemenata, te se nastavlja dalje sa elementom na sljedećoj poziciji, koja se također uspoređuje sa trenutno zadnje promatranim elementom, i tako dalje sve do elementa koji prethodi posljednjem elementu.

Ukoliko pak element u tekućoj iteraciji nije veći od zadnje promatranog elementa, zamjena se neće izvršiti, te se nastavlja dalje s izvođenjem petlje. Primjer takve situacije je druga iteracija.

Nakon što završi jedan takav prolaz vanjske iteracije, u zadnjem će se elementu polja, kao i u algoritmu sortiranja izborom, nalaziti element s najvećom vrijednošću, te ga više ne treba razmatrati. Korak vanjske petlje umanjuje se za jedan, te se ponavlja gore opisan postupak, sve dok ne preostane jedan element koji će tada već biti sortiran.

Poput algoritma sortiranja izborom, i u ovom se algoritmu polje zapravo izdvaja na **sortirani** i **nesortirani** dio.

Složenost ovog algoritma ista je kao i kod prethodnog, dakle $O(n^2)$. Međutim, ovaj je algoritam neučinkovitiji od algoritma sortiranja izborom, jer vrši drastično više zamjena, pa je u konačnici sporiji od istog.

3.3. Sortiranje umetanjem

Za razliku od prethodno opisanih algoritama, ovaj je algoritam sofisticiraniji. U ovom se algoritmu nesortirani element premješta na odgovarajuću poziciju u svakom koraku. To se radi na način da se stvara prostor za umetanje trenutnog elementa pomicanjem elemenata s većom vrijednošću za jedno mjesto udesno. Ovaj se princip rada može povezati s razvrstavanjem karata u bridžu – razmatraju se karte jedna po jedna, te se umeću na odgovarajuće mjesto među one karte koje su već razmatrane. [1]

Ovaj algoritam u jeziku C++ možemo implementirati na sljedeći način (prema [1]):

```
void swap(int *x, int *y){
      int tmp = *x;
      *x = *y;
      *y = tmp;
}

void InsertionSort (int A[], int N){
      for(int i = 1; i < N; i++){
            for(int j = i; j > 0 && A[j] < A[j-1]; j--)
                  swap(&A[j], &A[j-1]);
      }
}
```

Rad ovog algoritma također ćemo pokazati na primjeru niza nasumičnih brojeva prema Tablici 1.

Tablica 4: Primjer sortiranja umetanjem

i	j	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
1	-	17	31	3	43	11	24	8
2	0	17	31	3	43	11	24	8
3	-	3	17	31	43	11	24	8
4	1	3	17	31	43	11	24	8
5	3	3	11	17	31	43	24	8
6	1	3	11	17	24	31	43	8
-	-	3	8	11	17	24	31	43

U svakom retku ove tablice bilježimo korak vanjske petlje *i* koji određuje nesortirani element kojeg trenutno promatramo, zatim posljednji korak unutarnje petlje *j* koji određuje gdje se promatrani element premjestio nakon usporedbi, te stanja elemenata polja **prije** nego je došlo do zamjena.

S obzirom da se stanja u elementima polja bilježe prije zamjena, u tablici nam zadnji redak zapravo predstavlja stanje nakon što su završili svi koraci algoritma.

Osim navedenog, posebno su istaknuti ključni elementi, pa su tako već sortirani elementi označeni žutom bojom, element koji se trenutno promatra i za kojeg nema zamjena označen je crvenom bojom, dok je zelenom bojom označen element koji se trenutno promatra i za kojeg temeljem usporedbi postoje zamjene te će kao takav biti premješten na drugu poziciju.

Promotrimo sada što se događa u pojedinom koraku **vanjske** petlje ovoga algoritma. Korak kreće, dakle, od drugog elementa polja, jer se za prvi element pretpostavlja da je on već sortiran, te za njega promatramo elemente koji mu prethode i vršimo usporedbu. U ovome primjeru, trenutni element koji promatramo (element na poziciji 1) nije **manji** od svog jedinog prethodnika (na poziciji 0), te algoritam nastavlja dalje.

U drugom koraku promatramo sljedeći element i ponovno vršimo uspoređivanje s već sortiranim elementima. Sada možemo uočiti kako je ovaj element manji od svog prethodnika, pa ga pomičemo ulijevo, te još jednom provjeravamo uvjet petlje. Element je ponovno manji od svog prethodnika, te ga ponovno pomičemo ulijevo. U ovom trenutku **unutarnja** petlja staje, jer smo došli do početka polja, te je konačna pozicija elementa kojeg smo pomicali u tekućem koraku 0, koja je zabilježena kao zadnja iteracija varijable *j*.

Pogledajmo još peti korak iteracije. Kao što smo već opisali, element pomičemo sve dok je on manji od svog prethodnika. U ovom će slučaju element na poziciji 5 biti premješten na poziciju 1, jer je bio manji od svakog svog prethodnika sve dok nije došao na poziciju 1. U tom je trenutku **unutarnja** petlja završila izvršavanje, te je zadnji korak iteracije zabilježen u tablici u varijabli *j*.

Za ovaj algoritam također možemo primijetiti kako tokom svoga rada polje dijeli na **sortirani** i **nesortirani** dio, i to na način da u svakom koraku iteracije zapravo uvećava sortirani dio za 1, dok nesortirani dio smanjuje za 1. [5] Valja napomenuti kako sortirani dio polja, za razliku od prethodnih algoritama, nije na svojim konačnim pozicijama sve dok **vanjska** petlja ne dođe do kraja, budući da se tokom svake njene iteracije može naići na još elemenata s najmanjom vrijednošću za koje će se morati napraviti mjesto za premještanje. [1]

Kao što smo već na početku spomenuli, ovaj je algoritam sofisticiraniji od svojih prethodnika, no bez obzira na to, ima istu složenost kao i njegovi prethodnici, dakle $O(n^2)$. Razlog tomu je što broj operacija koje ovaj algoritam mora odraditi u najgorem slučaju iznosi $n(n – 1)$. [5]

3.4. Mjehuričasto sortiranje

Ovaj je algoritam varijanta algoritma sortiranja zamjenom, a po svojoj je ideji vrlo jednostavan, zbog čega ga ljudi najčešće brzo nauče. Radi na način da se prolaskom kroz polje svaki element uspoređuje sa svojim susjednim elementom te pritom vrši zamjena ukoliko je vrijednost elementa sljedbenika manja od vrijednosti trenutno promatranog elementa. [6]

U ovom će se algoritmu, nakon prvog prolaska, kao i u algoritmu sortiranja izborom i algoritmu sortiranja zamjenom, element s najvećom vrijednošću nalaziti na posljednjem mjestu u polju, te da ga više nećemo morati razmatrati. Postupak se tako ponavlja, kao i kod spomenutih dvaju algoritama, na način da se smanjuje duljina promatranog polja za 1, no razlika je u tome što se u ovome algoritmu može stati i ranije, ukoliko tijekom prolaza nije došlo do zamjene elemenata, jer je u tom slučaju polje već zapravo sortirano.

Kod ovoga algoritma u programskom jeziku C++ bi izgledao ovako (prema [7]):

```
void swap(int *x, int *y){
      int tmp = *x;
      *x = *y;
      *y = tmp;
}

void BubbleSort (int A[], int N){
      bool swapped = true;
      for(int i = N-1; i > 0 && swapped; i--){
            swapped = false;
            for(int j = 0; j < i; j++){
                  if(A[j] > A[j+1]){
                        swap(&A[j], &A[j+1]);
                        swapped = true;
                  }
            }

      }
}
```

S obzirom da ovaj algoritam ima velik broj koraka i zamjena elemenata, poput algoritma sortiranja zamjenom, na primjeru ćemo pokazati što se događa samo u prvom prolazu ovoga algoritma (Tablica 5).

Tablica 5: Primjer prvog prolaza kod mjehuričastog sortiranja

i	j	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
6	0	17	31	3	43	11	24	8
6	1	17	31	3	43	11	24	8
6	2	17	3	31	43	11	24	8
6	3	17	3	31	43	11	24	8
6	4	17	3	31	11	43	24	8
6	5	17	3	31	11	24	43	8
6	6	17	3	31	11	24	8	43

U Tablici 5, svaki redak označava jedan korak **unutarnje** petlje algoritma. Pri tome se bilježi korak **vanjske** petlje *i* koji određuje zadnji element niza do kojeg će se kretati **unutarnja** petlja algoritma, čije korake bilježimo u stupcu *j*. Kao i uvijek, bilježi se i stanje svih elemenata polja u svakom koraku, u ovom slučaju **unutarnje** petlje, i to **prije** nego je došlo do zamjene.

Za potrebe ovog primjera, kako bismo bolje uočili ključne elemente u svakom koraku, crvenom su bojom označeni oni elementi koji ne zadovoljavaju kriterij usporedbe, odnosno u kojem trenutni element *j* nije veći od svog sljedbenika *j+1*, dok su zelenom bojom označeni elementi koji zadovoljavaju kriterij, odnosno u kojem trenutni element *j* jest veći od svog sljedbenika *j+1*. Valja napomenuti kako će ti, zeleno označeni elementi, biti zamijenjeni neposredno nakon ispitivanja uvjeta, a njihovo je promijenjeno stanje prikazano u idućem koraku. Osim navedenog, u zadnjem je prolazu žutom bojom istaknut zadnji element, koji predstavlja sortirani dio, odnosno najveći element polja koji je smješten na začelje.

Proučimo sada što se događa tijekom prve iteracije **vanjske** petlje algoritma. U prvoj **unutarnjoj** iteraciji promatramo element na poziciji 0, te ga uspoređujemo s njemu susjednim elementom sljedbenikom na poziciji 1. Možemo primijetiti kako je vrijednost promatranog elementa na poziciji 0 manja od vrijednosti njegovog elementa sljedbenika na poziciji 1, te da se ovdje ne vrši zamjena.

Algoritam nastavlja dalje, tako da je sada promatrani element na poziciji 1, i u ovom slučaju je vrijednost njegova sljedbenika manja, tako da će se u ovome trenutku izvršiti zamjena tih dvaju elemenata. Tu zamjenu možemo vidjeti u trećoj iteraciji, u kojoj se ponavlja situacija iz prve iteracije, tako da ni ovaj puta ne dolazi do zamjene elemenata.

Pogledajmo još četvrti korak, koji je posebno zanimljiv jer je u tom trenutku zapravo pronađen najveći element niza. S obzirom na već opisani postupak, možemo uočiti kako će se

u ovom koraku, kao i u svim idućim koracima do kraja niza, taj element zapravo premještati. Razlog tomu je što će vrijednost tog elementa u svakom idućem koraku biti veća od svojih sljedbenika, čime je on u konačnici smješten na poziciju posljednjeg elementa polja i čime zapravo postaje sortiran te ga više ne treba razmatrati.

Nakon završetka ovoga postupka, algoritam nastavlja dalje istim principom, ali ne nužno do prvog elementa polja. Naime, zbog svoje karakterističnosti, ovaj se algoritam može i ranije završiti, na način da se u svakom koraku **vanjske** petlje ispituje jesu li elementi zamijenjeni, te ako nisu, završi s izvršavanjem **vanjske** petlje algoritma.

Bez obzira na spomenuto svojstvo, ovaj je algoritam daleko najsporiji od svih koji su u ovome radu opisani. Razlog tomu jest što je, u slučaju da se element s velikom vrijednošću nalazi pri početku polja, potreban velik broj zamjena kako bi taj element došao na svoje mjesto. Uz to, ako se element s malom vrijednošću nalazi pri kraju polja, on će se vrlo sporo premještati prema svome mjestu. Nadalje, ako se element u početnom polju nalazi iza mjesta na kojem bi trebala biti u sortiranom polju, nastaje problem jer se u svakom prolazu element može premjestiti samo za jedno mjesto unaprijed. [7]

Navedeni problem poznat je kao problem zečeva i kornjača, a kako bi se on izbjegao, koristi se algoritam dvostrukog mjehuričastog sortiranja, koji će biti opisan u nastavku. [7]

Što se složenosti ovog algoritma tiče, ona je, bez obzira na sve, ista kao u svim do sada opisanim algoritmima, jer ukupan broj operacija u najgorem slučaju iznosi $2n(n – 1)$, dakle $O(n^2)$. [5]

3.5. Dvostruko mjehuričasto sortiranje

Dvostruko mjehuričasto sortiranje, poznato i kao *cocktail sort* ili *shaker sort*, inačica je algoritma mjehuričastog sortiranja. U ovom se algoritmu, za razliku od mjehuričastog sortiranja, elementi ne premještaju samo s početka polja prema kraju, već se u svakom drugom prolazu premještaju i s kraja prema početku. [7]

S obzirom na navedeno, u ovom će se algoritmu nakon prvog prolaza element s najvećom vrijednošću nalaziti na kraju polja, a nakon drugog prolaza će element s najmanjom vrijednošću biti na početku polja i tako dalje sve dok se polje u cijelosti ne sortira. [7]

Kako bi se implementirao ovaj algoritam, potrebne su dvije varijable – jedna u kojoj će biti smještena pozicija elementa od kojeg započinju nesortirane vrijednosti i druga u kojoj one završavaju. [7]

Pogledajmo sada kako bi ta implementacija izgledala u jeziku C++ (prema [7]):

```
void swap(int *x, int *y){
	int tmp = *x;
	*x = *y;
	*y = tmp;
}

void CocktailSort (int A[], int N){
	bool swapped = true;
	int start = 0, end = N-1;
	for(int i = N-1; i > 0 && swapped; i--){
		swapped = false;
		if(!(i%2)){
			for(int j = 0; j < end; j++){
				if(A[j] > A[j+1]){
					swap(&A[j], &A[j+1]);
					swapped = true;
				}
			}
			end--;
		}
		else{
			for(int j = end; j > start; j--){
				if(A[j] < A[j-1]){
					swap(&A[j], &A[j-1]);
					swapped = true;
				}
			}
			start++;

		}
	}
}
```

Zbog velikog broja koraka, kao i u prethodnome algoritmu, i za ovaj algoritam nećemo prikazivati sve prolaze, već samo prva dva prolaza, kako bismo vidjeli promjene stanja u oba smjera.

Tablica 6: Primjer prvog prolaza kod dvostruko mjehuričastog sortiranja

i	j	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
6	0	17	31	3	43	11	24	8
6	1	17	31	3	43	11	24	8
6	2	17	3	31	43	11	24	8
6	3	17	3	31	43	11	24	8
6	4	17	3	31	11	43	24	8
6	5	17	3	31	11	24	43	8
6	6	17	3	31	11	24	8	43

Ova tablica poznata nam je iz prošlog poglavlja, jer je prvi prolaz, barem u ovom slučaju, identičan kao i kod mjehuričastog sortiranja. Naime, kako je polje veličine 7 elemenata, a početni element na poziciji N – 1, smjer kretanja će biti od početka prema kraju, jer je broj pozitivan. S obzirom na to, ovu tablicu nećemo posebno opisivati, nego ćemo samo reći kako, za razliku od prethodnog algoritma, varijabla *end* označava krajnji element koji se uzima u obzir, a on je ujedno i početni element za idući korak algoritma.

Pogledajmo sada kako izgleda taj drugi korak.

Tablica 7: Primjer drugog prolaza kod dvostruko mjehuričastog sortiranja

i	j	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
5	5	17	3	31	11	24	8	43
5	4	17	3	31	11	8	24	43
5	3	17	3	31	8	11	24	43
5	2	17	3	8	31	11	24	43
5	1	17	3	8	31	11	24	43
5	0	3	17	8	31	11	24	43

Kao što smo već spomenuli, u drugom prolazu algoritam kreće od zadnjeg elementa, tj. elementa na poziciji smještenoj u varijabli *end*, i to prema početku polja, odnosno do pozicije smještene u varijabli *start*. Princip je isti kao i u prvome koraku, a razlika je jedino u tome što se sada provjerava je li vrijednost trenutnog elementa **manja** od vrijednosti **prethodnika**, te se u tom slučaju vrši njihova zamjena. Pogledajmo sada još zadnji redak Tablice 7. Primijetimo kako je sada na početku polja element s **najmanjom** vrijednošću, te da ga isto tako ne trebamo više razmatrati. To znači da se varijabla *start* uvećava, te da se u idućem koraku kreće od pozicije koju ona sadrži.

Opisani se postupak ponavlja sve dok svi elementi ne budu sortirani, a s obzirom na korisno svojstvo ovoga algoritma, može se stati i ranije, kao i kod osnovnog algoritma mjehuričastog sortiranja.

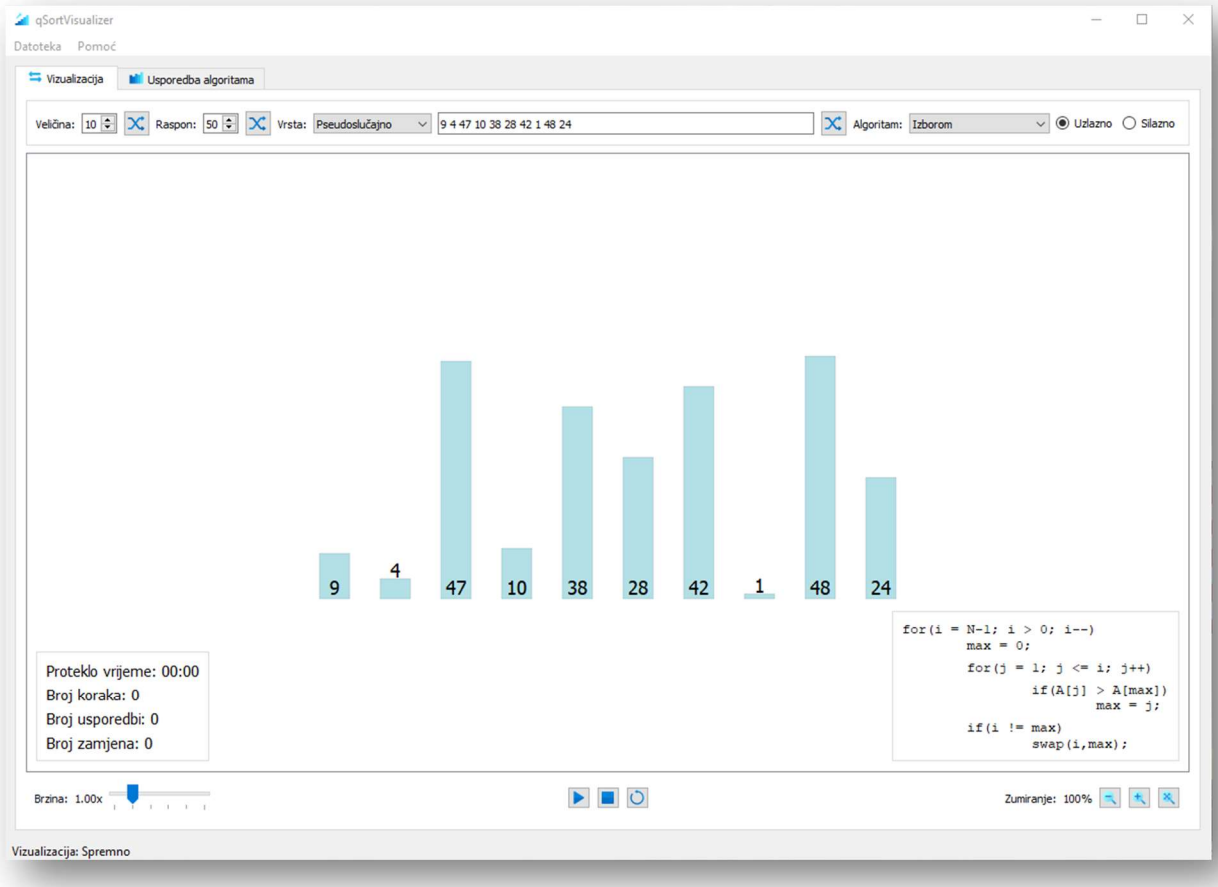
Jasno je kako i ovaj algoritam radi na način da polje dijeli na **sortirani** i **nesortirani** dio, s time da će u ovom algoritmu sortirani dio biti i na početku i na kraju niza. No, bez obzira na to, ovaj je algoritam i dalje najsporiji od prva tri koja smo proučavali. Razlog tomu je što je, iako se sada elementi s manjim vrijednostima brže premještaju prema početku polja, i dalje potreban velik broj zamjena kako bi se elementi premještali na veće udaljenosti. [7]

4. Programsko rješenje

U prethodnome smo poglavlju mogli primijetiti kako nam statička vizualizacija može olakšati shvaćanje rada algoritma, no unatoč tomu, lako se može dogoditi da izgubimo trag, a osim toga, postupak jednog takvog vizualiziranja često je dugotrajan, posebice u algoritmima koji imaju velik broj koraka, odnosno promjena.

U tu se svrhu, u ovome radu, nudi programsko rješenje, koje će na dinamičan način vizualizirati promjene u polju u svim koracima rada algoritma. Osim rada algoritama, program dodatno omogućava i vizualni prikaz statističkih mjerenja koja se mogu provesti za polja većih veličina, kako bi se jasno mogla vidjeti njihova složenost.

Programsko je rješenje, pod nazivom *qSortVisualizer*, razvijeno unutar Qt razvojnog okruženja prilikom čega je korišten programski jezik C++. S obzirom na to, implementacije svih prethodno opisanih algoritama u programskom su rješenju implementirane na isti način.



Slika 1: Početni zaslon *qSortVisualizer-a*

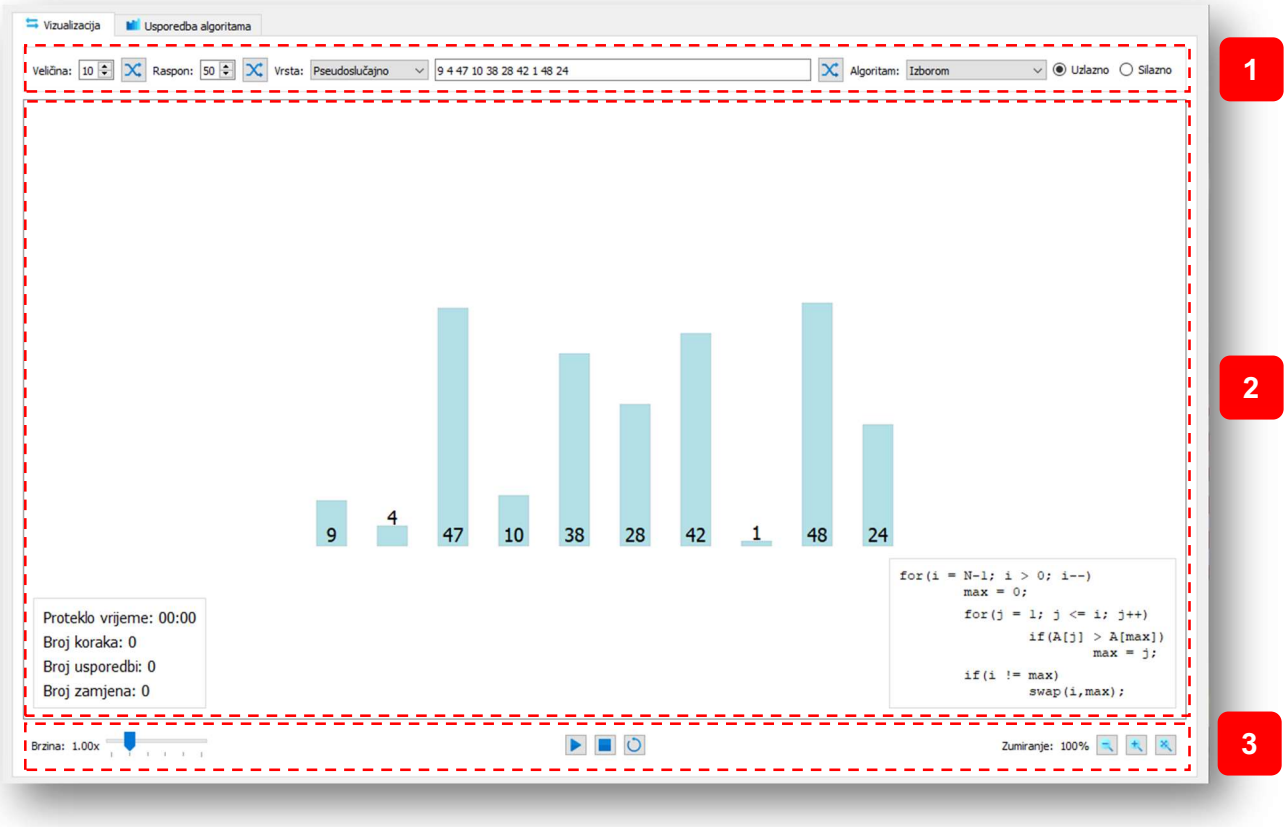
Dizajn sučelja programa vrlo je jednostavan. Program se pokreće u veličini 800x600 kako bi mogao raditi i na zaslonima manjih rezolucija. Prilikom pokretanja, prikazuje se kartični izbornik s otvorenom karticom **Vizualizacija** (Slika 1).

Kao što je već ranije spomenuto, program je podijeljen na dva dijela: vizualizacija i usporedba algoritama. Dio vizualizacije, koji se odmah prikazuje prilikom pokretanja, omogućava vizualiziranje algoritama sortiranja na poljima različitih vrsta, veličina i raspona, dok dio usporedbe algoritama nudi mogućnost statističkih mjerenja tokom rada algoritama, s poljima većih veličina i raspona. U nastavku će se detaljnije opisati ove dvije cjeline programa.

4.1. Vizualizacija

Kartica **Vizualizacija** sastoji se od 3 glavna dijela:

1. podešavanje postavki polja i odabir algoritma
2. grafička scena za vizualizaciju s brojačima i programskim kodom algoritma
3. upravljanje vizualizacijom i grafičkom scenom



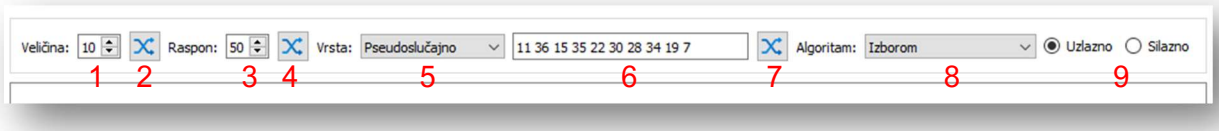
Slika 2: Vizualizacija

4.1.1. Postavke polja i odabir algoritma

Ova sekcija služi za podešavanje izgleda početnog polja koje će se vizualizirati, te omogućava odabir algoritama sortiranja za vizualizaciju kao i poredak sortiranja. Prilikom mijenjanja ovih postavki, scena s objektima se ažurira automatski.

Sekcija se sastoji od nekoliko kontrola (Slika 3):

1. kontrola za ručno podešavanje veličine niza
2. gumb za nasumično podešavanje veličine niza
3. kontrola za ručno podešavanje raspona niza
4. gumb za nasumično podešavanje raspona niza
5. padajući izbornik za odabir vrste niza (pseudoslučajno, uzlazno sortirano, silazno sortirano, skoro sortirano, uniformno, Gaussova krivulja, prilagođeno)
6. tekstualni obrazac za pregled generiranog niza ili prilagođeno uređivanje niza
7. gumb za nasumično generiranje novih brojeva prema prethodno zadanim postavkama
8. padajući izbornik za odabir algoritma (izborom, zamjenom, umetanjem, mjehuričasto, dvostruko mjehuričasto)
9. opcije za odabir vrste sortiranja (uzlazno ili silazno)

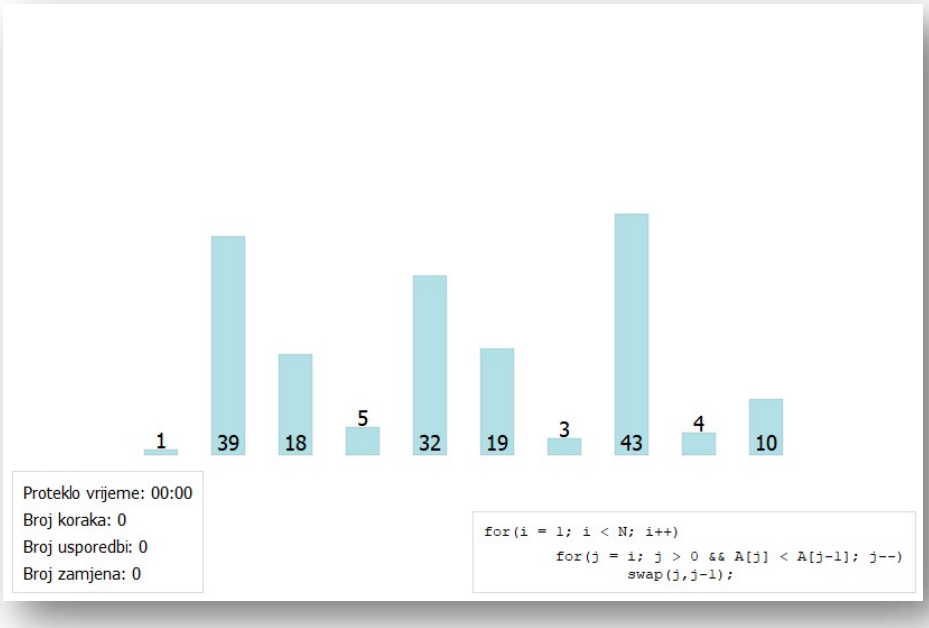


Slika 3: Kontrole postavki polja i odabira algoritama

Kako bismo podesili **veličinu niza**, potrebno je ručno odabrati željenu veličinu (1). Raspon veličine niza ograničen je na brojeve između 3 i 20. Razlog tomu je što s jedne strane nema smisla vizualizirati sortiranje za manje od 3 broja, a s druge strane više od 20 objekata niza ne bi bilo moguće smjestiti na ekran standardne rezolucije. Ukoliko pak ne želimo ručno podešavati veličinu, možemo ju podesiti koristeći gumb **Nasumično** (2) koji će generirati nasumičan broj u navedenom rasponu. Na sličan način možemo ručno (3) ili nasumično (4) podesiti i **raspon niza**, koji može biti između 1 i 80.

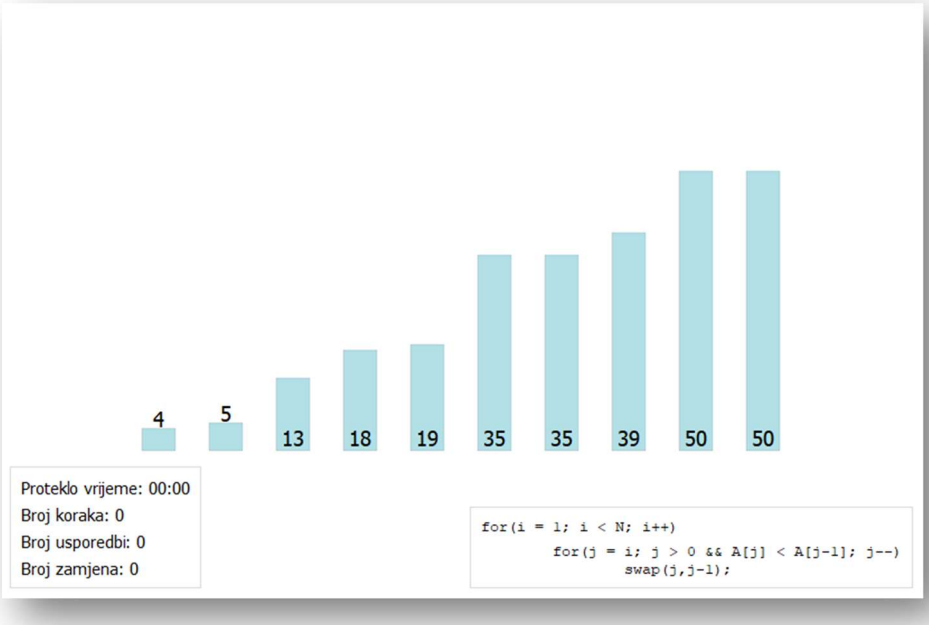
Za podešavanje **elemenata niza** imamo nekoliko mogućnosti. Najprije iz padajućeg izbornika (5) odabiremo jednu od ponuđenih opcija:

- Pseudoslučajno – niz se generira pomoću generatora pseudoslučajnih brojeva prema prethodno zadanoj veličini niza



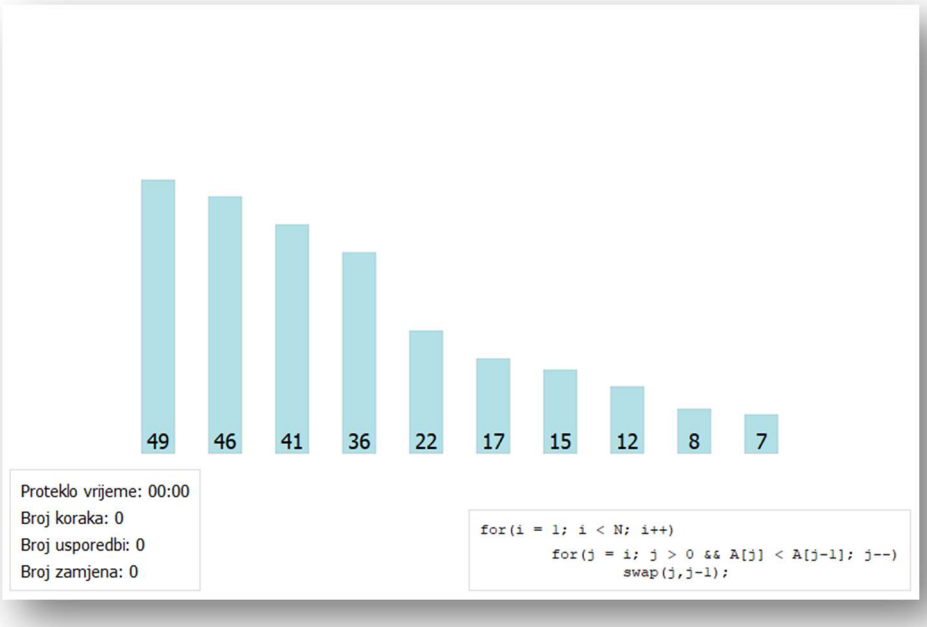
Slika 4: Pseudoslučajni niz

- Sortirano (uzlazno) – niz se generira pomoću generatora pseudoslučajnih brojeva prema prethodno zadanoj veličini niza i zatim sortira **uzlaznim** poretkom



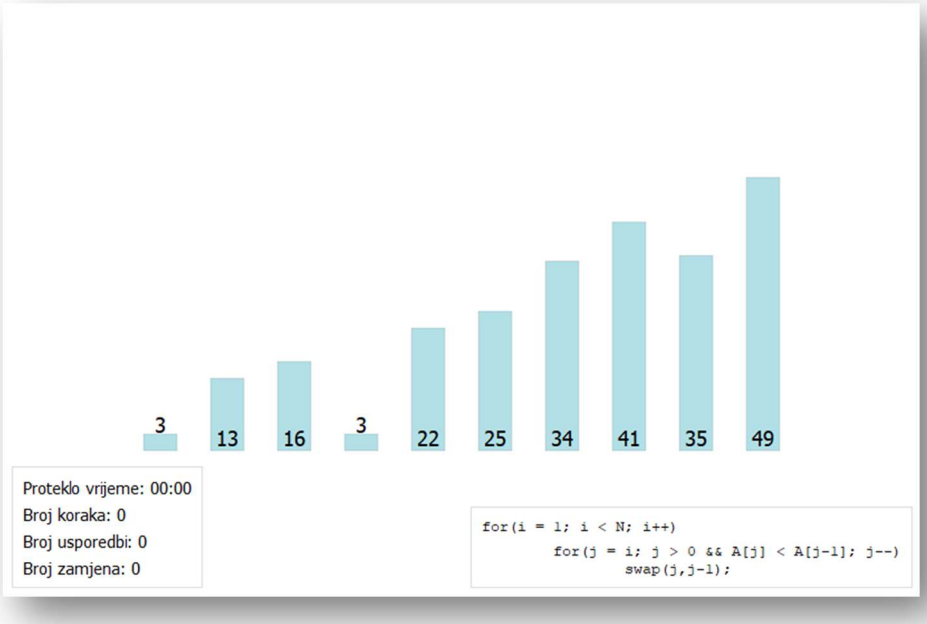
Slika 5: Uzlazno sortirani niz

- Sortirano (silazno) – niz se generira pomoću generatora pseudoslučajnih brojeva prema prethodno zadanoj veličini niza i zatim sortira **silaznim** poretkom



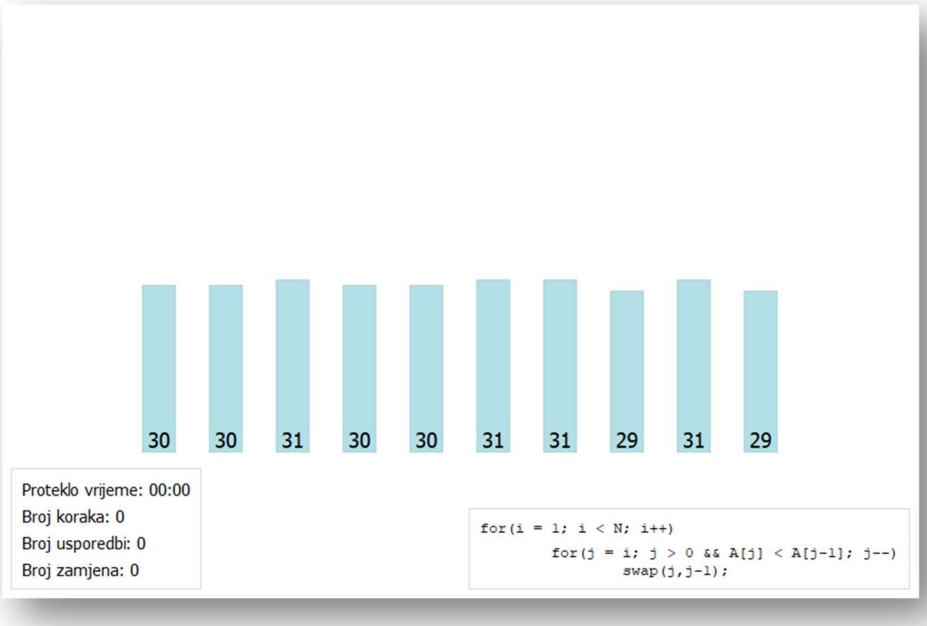
Slika 6: Silazno sortirani niz

- Skoro sortirano – niz se generira pomoću generatora pseudoslučajnih brojeva prema prethodno zadanoj veličini niza i zatim djelomično sortira



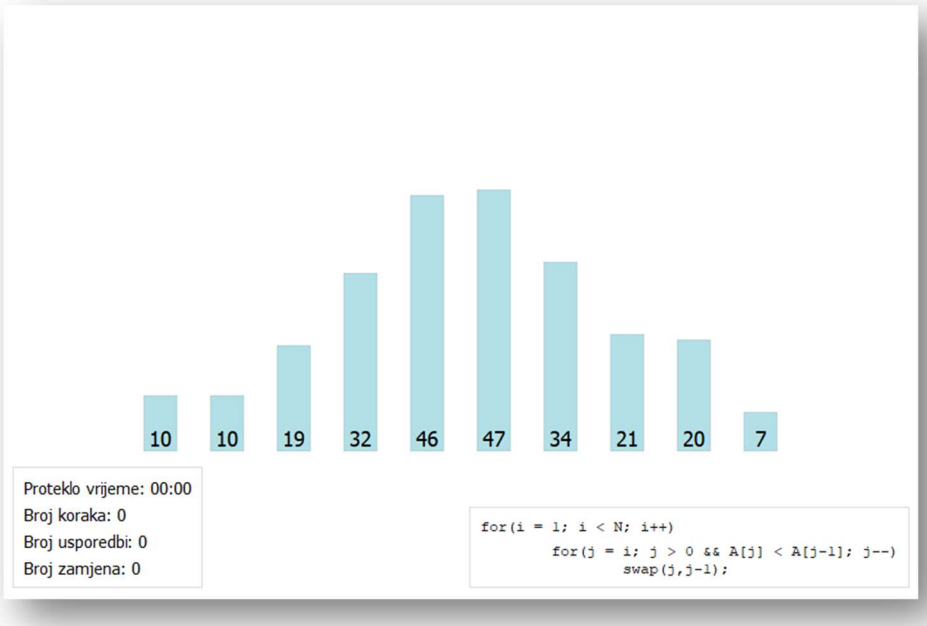
Slika 7: Skoro sortirani niz

- Uniformno – niz se generira tako da se pojavljuje više dupliciranih i približno istih vrijednosti po nasumičnom razmještaju



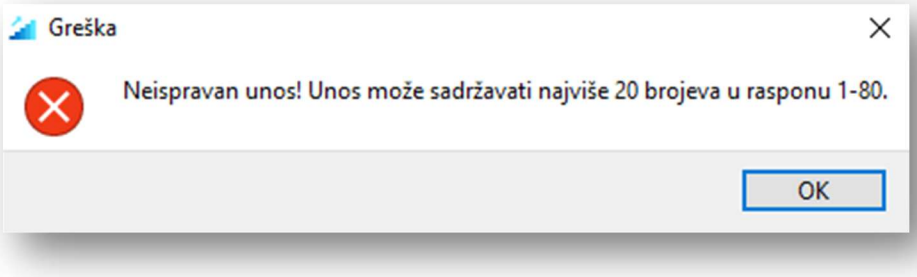
Slika 8: Uniformni niz

- Gaussova krivulja – niz se generira tako da formira oblik Gaussove krivulje (normalna raspodjela)



Slika 9: Niz u obliku Gaussove krivulje

- Prilagođeno – omogućuje ručni unos željenih vrijednosti odvojenih razmakom u obrazac (6), pri čemu vrijedi već ranije spomenuto ograničenje o rasponu brojeva. Ukoliko je unos neispravan, program odmah javlja grešku te vraća posljednje ispravan unos (Slika 10). Valja napomenuti kako je u ovoj opciji onemogućena kontrola za ručno podešavanje veličine i raspona niza, gumbovi za nasumično generiranje veličine i raspona niza, kao i gumb za generiranje novih nasumičnih brojeva.



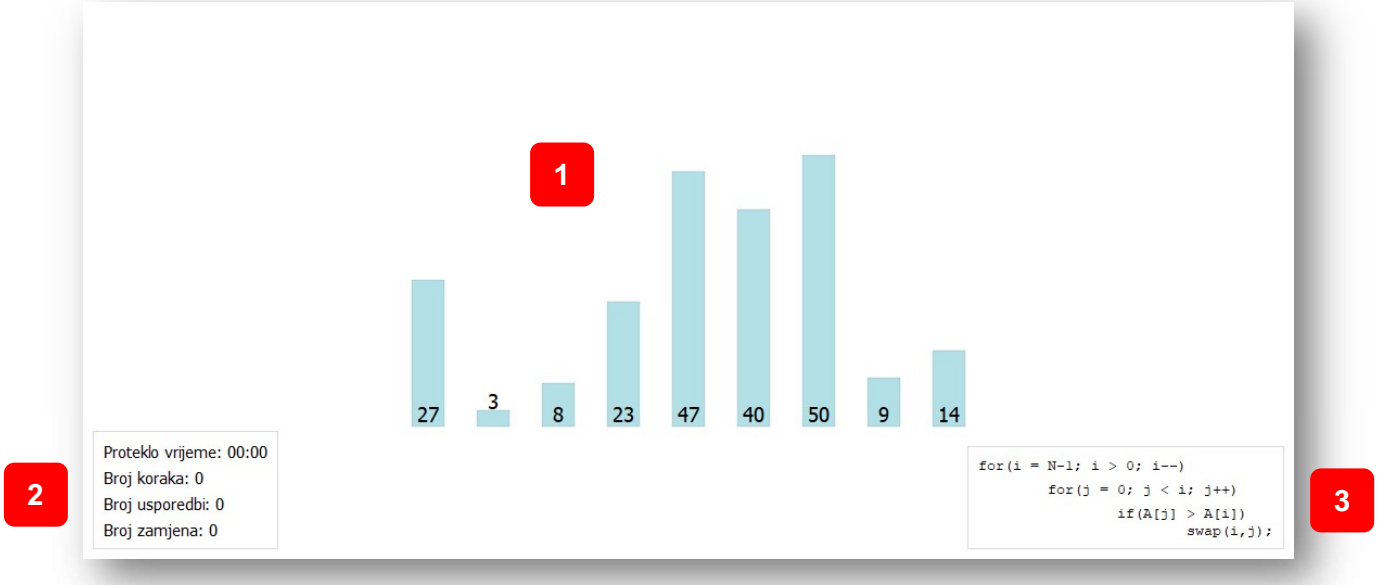
Slika 10: Greška prilikom prilagođenog unosa niza

Ukoliko je iz padajućeg izbornika odabrana bilo koja opcija osim **Prilagođeno**, moguće je koristiti gumb za generiranje novih nasumičnih brojeva (7), kako bi se generirao niz s novim vrijednostima za prethodno zadane postavke.

Nakon što je generiran niz, slijedi odabir algoritma sortiranja koji se obavlja putem padajućeg izbornika (8). Moguće je odabrati sljedeće algoritme sortiranja: izborom, umetanjem, zamjenom, mjehuričasto te dvostruko mjehuričasto. Dodatno je moguće odabrati i poredak za sortiranje koji može biti **uzlazno** (zadano) ili **silazno** (9).

4.1.2. Grafička scena

Grafička scena je područje u kojem su smješteni objekti za vizualizaciju. Svaki objekt predstavlja jedan broj iz generiranog niza, u obliku pravokutnika, čija je visina određena generiranim brojem (1). Dodatno, scenu prekrivaju i dva prozorčića (eng. *overlay*): brojači za vrijeme, broj koraka, broj usporedbi i broj zamjena (2) te programski kod odabranog algoritma i željenog poretka (3).



Slika 11: Grafička scena

4.1.3. Upravljanje vizualizacijom i grafičkom scenom

U ovoj se sekciji nalaze sljedeće kontrole kojima je moguće upravljati vizualizacijom i grafičkom scenom (Slika 12).

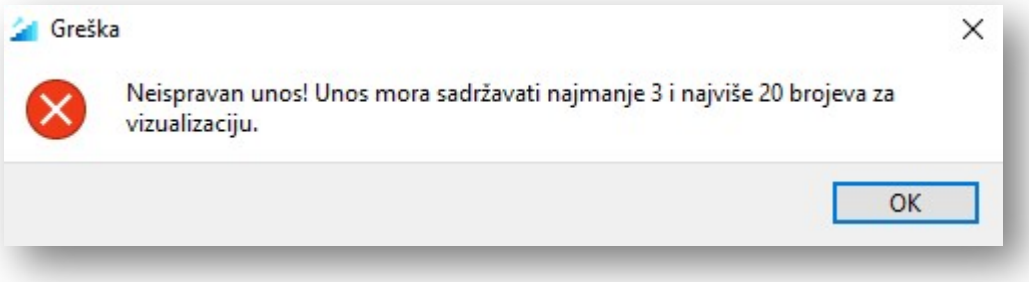
Slika 12: Kontrole upravljanja vizualizacijom i grafičkom scenom

Moguće su slijedeće opcije:

1. Brzina izvođenja – omogućava promjenu brzine izvođenja vizualizacije, u rasponu od 0.25x do 4.00x u odnosu na zadanu brzinu. Kontrolom je moguće upravljati čak i za vrijeme trajanja vizualizacije

2. Kontrole izvođenja – set gumbova za promjene stanja vizualizacije. Moguće su slijedeće opcije:
    - **Pusti/Pauziraj** – multifunkcijski gumb koji služi pokretanju vizualizacije ukoliko je ona zaustavljena. Nakon pokretanja gumb mijenja funkciju i postaje gumb za privremeno zaustavljanje vizualizacije (pauziranje). Osim toga, onemogućuju se sve kontrole postavki polja i odabira algoritama (Slika 3).
    - **Zaustavi** – gumb trajno zaustavlja vizualizaciju, čime se ponovno omogućavaju kontrole postavki polja i odabira algoritama (Slika 3), a onemogućuje se gumb **Pusti/Pauziraj** sve dok se ne odabere novi niz ili ne klikne na gumb **Ponovno pokreni**.
    - **Ponovno pokreni** – postavlja niz na početno stanje prije vizualiziranja. Omogućuje gumb **Pusti/Pauziraj** nakon što je vizualizacija zaustavljena, a ako je u tijeku izvođenja, automatski zaustavlja animaciju i ponovno postavlja niz.
3. Zumiranje – set gumbova koji omogućava promjene skaliranja grafičke scene u rasponu 25% - 300% u odnosu na zadano. Moguće je odabrati opcije: **Smanji**, **Povećaj**, **Zadano.**

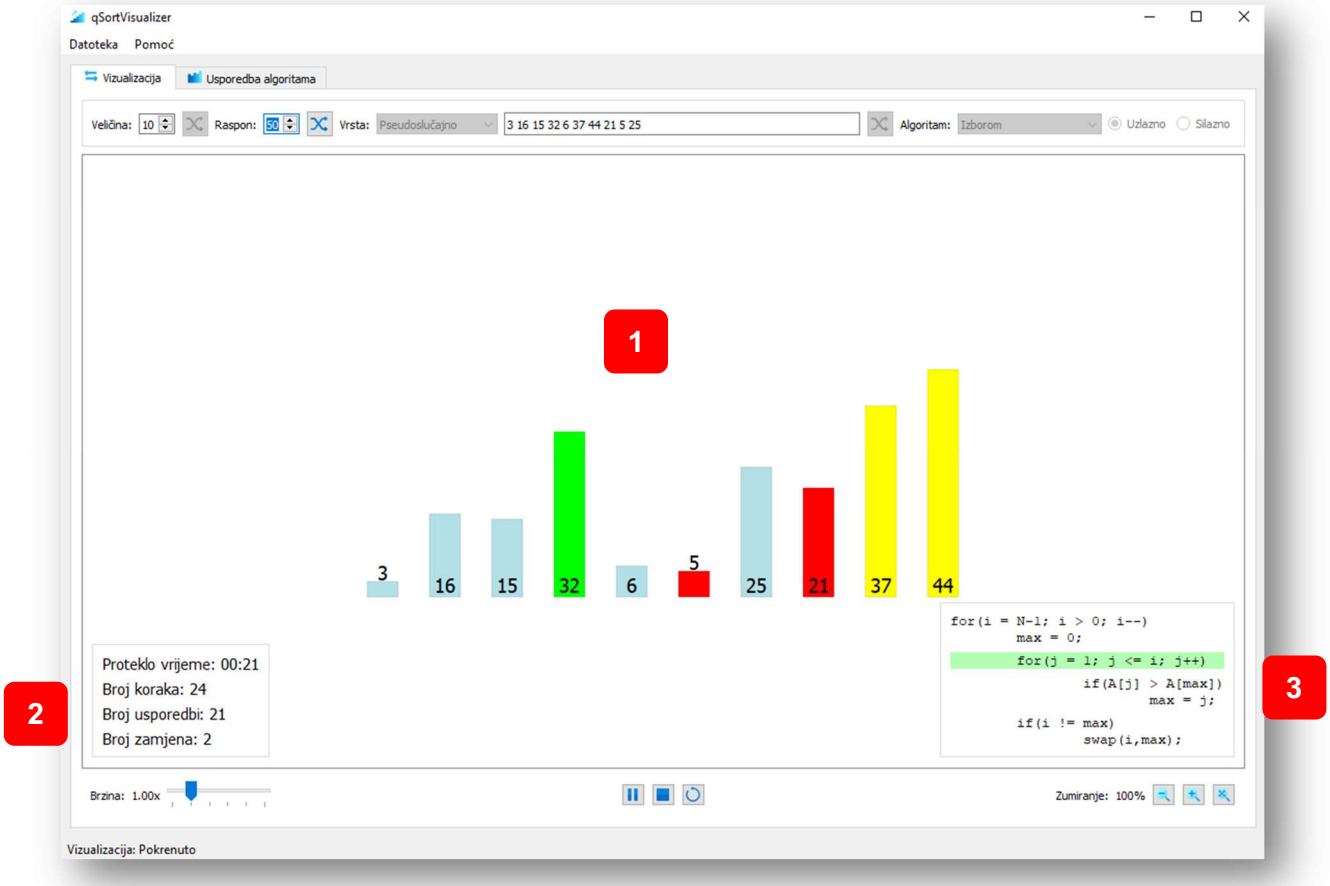
Ukoliko je odabrana opcija unosa elemenata **Prilagođeno**, a nije uneseno barem 3 elementa niza, prilikom klika na gumb **Pusti** program javlja grešku (Slika 13).



Slika 13: Greška prilikom pokretanja animacije

4.1.4. Primjer izvođenja vizualizacije

Pogledajmo sada kako izgleda vizualizacija na primjeru sortiranja izborom (Slika 14). Vizualizacija se vrši nad objektima smještenima na grafičkoj sceni, prilikom čega se različitim bojama posebno ističu ključni elementi (1). Tijekom rada bilježe se statistički podaci i proteklo vrijeme, koji se ažuriraju u stvarnom vremenu (2). Osim toga, paralelno s vizualizacijom se izvodi i kod algoritma, na način da se sjenčanjem označava trenutna linija koda koja se izvršava (3).

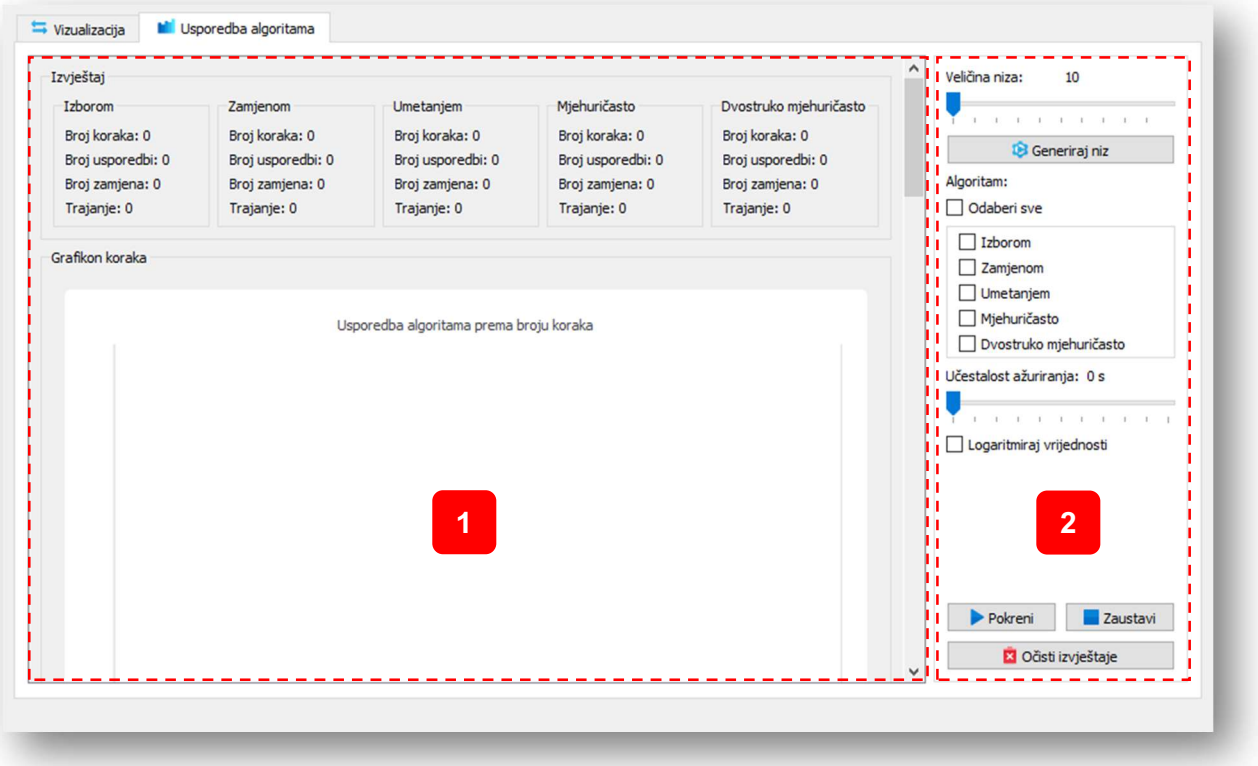


Slika 14: Primjer izvođenja algoritma sortiranja izborom

4.2. Usporedba algoritama

Kartica **Usporedba algoritama** sastoji se od 2 glavna dijela:

1. područje za ispis rezultata izvještaja s grafičkim prikazima
2. kontrole za upravljanje i generiranje niza, odabir algoritama te kontrole za provedbu mjerenja

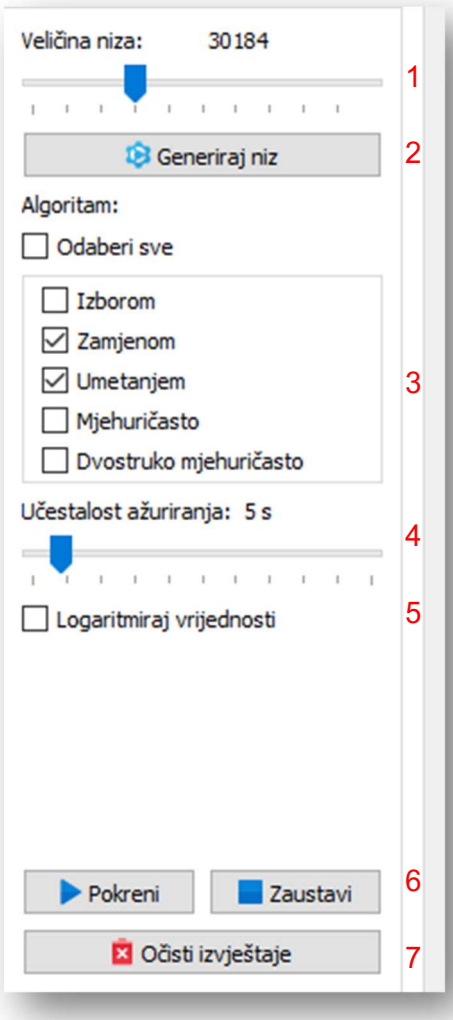


Slika 15: Usporedba algoritama

4.2.1. Podešavanje postavki mjerenja

Ova se sekcija sastoji od nekoliko kontrola i opcija (Slika 16):

1. podešavanje veličine niza
2. gumb za generiranje niza
3. odabir algoritama
4. podešavanje učestalosti ažuriranja rezultata mjerenja
5. odabir opcije za logaritmiranje vrijednosti na grafikonima
6. set gumbova za upravljanje mjerenjima
7. gumb za čišćenje rezultata mjerenja na izvještajima



Slika 16: Postavke mjerenja

Kako bismo proveli statističko mjerenje, najprije je potrebno podesiti veličinu niza. To se radi pomoću klizača, a raspon niza je od 10 do 100.000 elemenata (1). Nakon odabira veličine niza, možemo kliknuti na gumb **Generiraj niz**, prilikom čega će se generirati niz pseudoslučajnih elemenata zadane veličine (2).

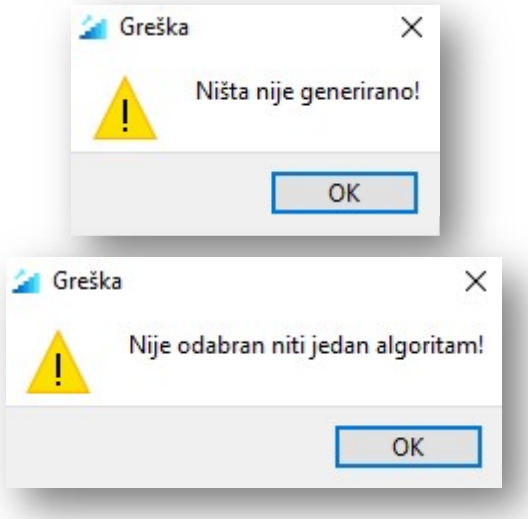
Nakon generiranja niza, potrebno je odabrati algoritme za koje želimo izvršiti testiranje. Pritom je moguće odabrati sve algoritme ili pojedinačno (3).

Prije nego se pokrene mjerenje, može se odabrati **Učestalost ažuriranja** pomoću klizača, u rasponu od 0 do 60 sekundi (4). Ako je odabrano 0 sekundi (zadano), rezultati će se prikazati tek prilikom završetka rada pojedinog algoritma, a ukoliko je odabrana bilo koja druga vrijednost, tada će se rezultati svaki puta osvježavati istekom zadanog perioda, tokom rada algoritama.

S obzirom da prilikom mjerenja rezultati dosežu vrlo velike brojeve, rezultati se mjerenja na vizualnom prikazu mogu logaritmirati pomoću odabira opcije **Logaritmiraj vrijednosti** (5)**.**

Kako bi se pokrenula mjerenja, potrebno je kliknuti na gumb **Pokreni**. Jednom kada je mjerenje pokrenuto, nije moguće ponovno pokrenuti mjerenje, sve dok se trenutno mjerenje ne zaustavi. Kako bismo zaustavili mjerenje, potrebno je kliknuti na gumb **Zaustavi** (6).

Ukoliko se pokuša kliknuti na gumb **Pokreni**, a da pritom nije generiran niz ili nije odabran niti jedan algoritam, program javlja grešku (Slika 17).



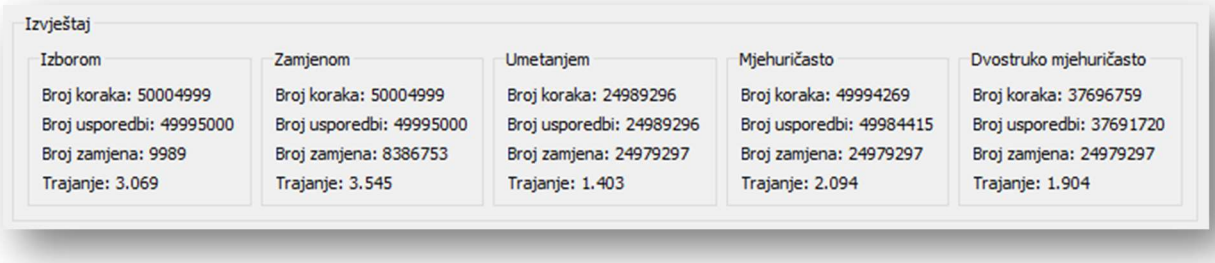
Slika 17: Greške kod pokretanja mjerenja

U slučaju da želimo očistiti rezultate izvještaja i prikaz na grafikonima, to možemo učiniti pomoću gumba **Očisti izvještaje** (7).

4.2.2. Primjer statističkog mjerenja

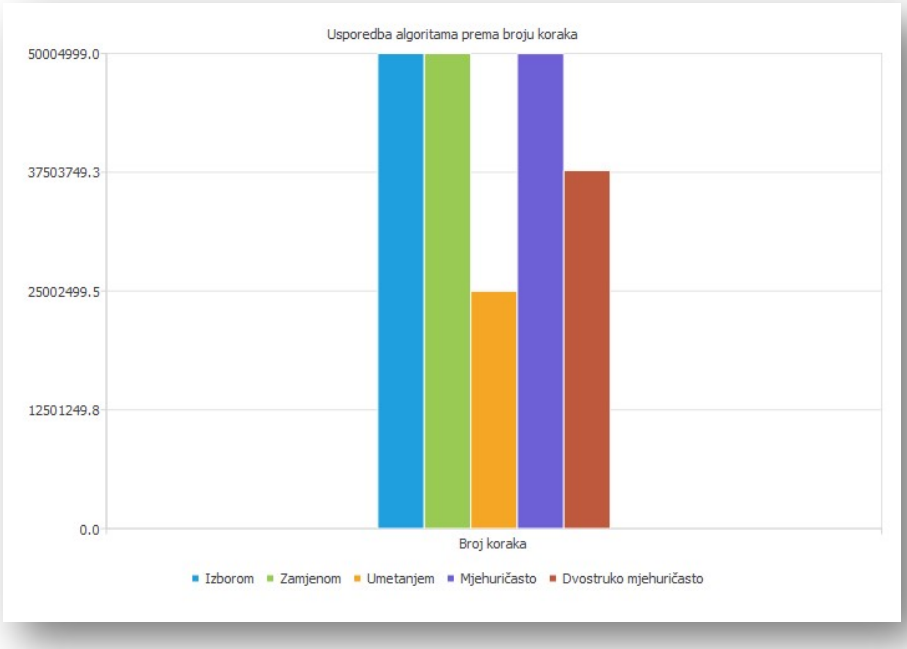
Za primjer ćemo koristiti niz od 10.000 nasumično generiranih brojeva, a za testiranje ćemo koristiti sve ponuđene algoritme. Test je proveden na računalu s i5 procesorom i 16 GB memorije.

Na Slici 18. možemo vidjeti primjer dobivenog izvještaja statističkih mjerenja.

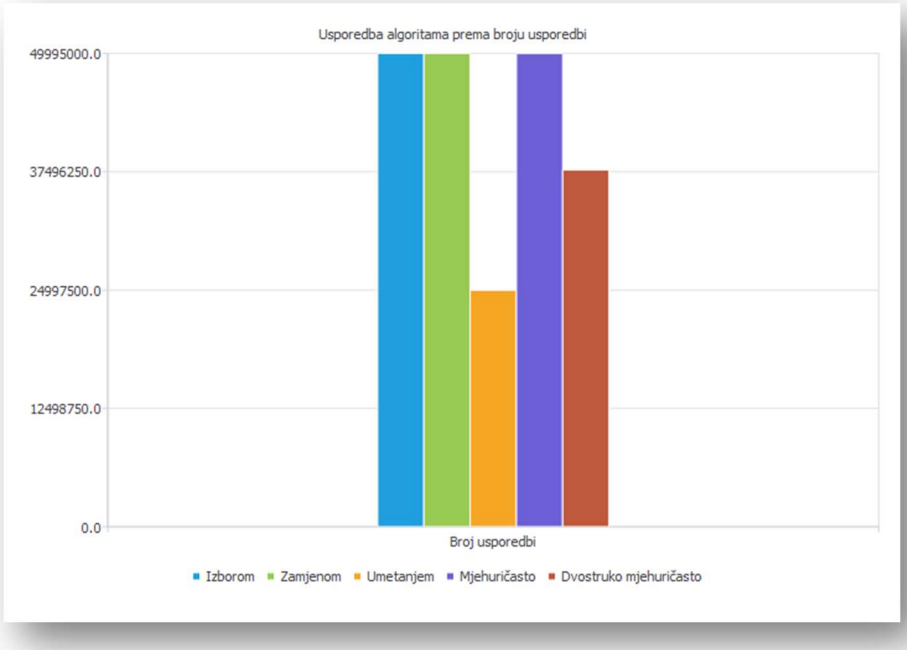


Slika 18: Prikaz izvještaja za dobivena mjerenja

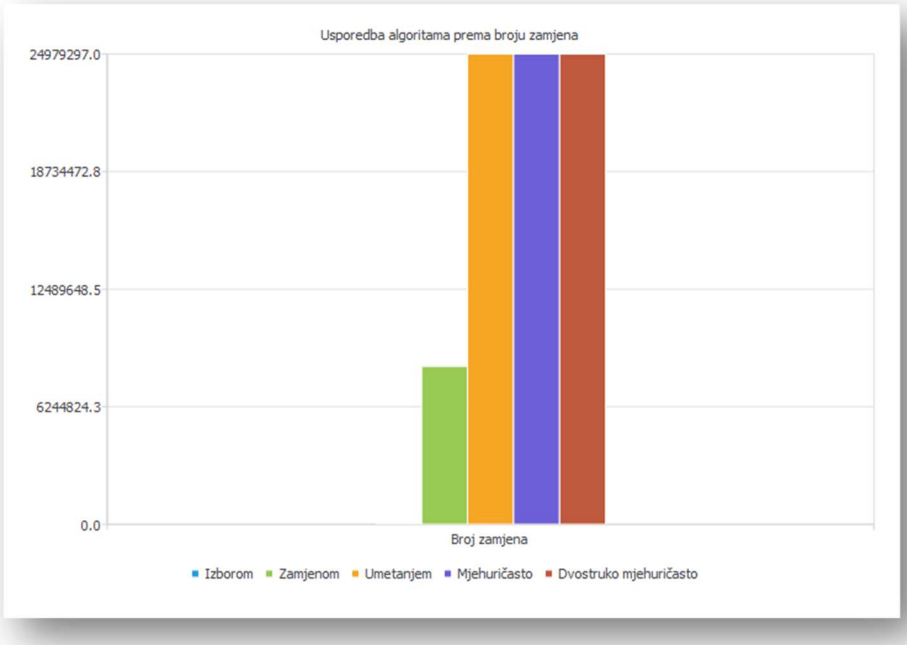
Kako bismo bolje uočili razlike u dobivenim rezultatima za odabrane algoritme, dodatno se koristi vizualni prikaz dobivenih podataka pomoću stupčastih grafikona. Primjere takvih grafikona možemo vidjeti na slikama u nastavku.



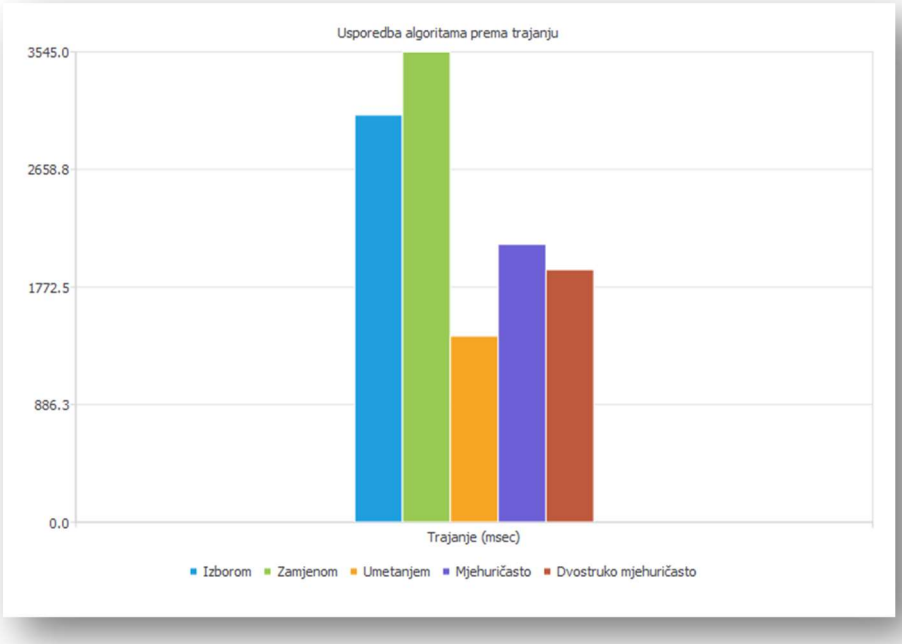
Slika 19: Grafički prikaz usporedbe algoritama prema broju koraka



Slika 20: Grafički prikaz usporedbe algoritama prema broju usporedbi



Slika 21: Grafički prikaz usporedbe algoritama prema broju zamjena

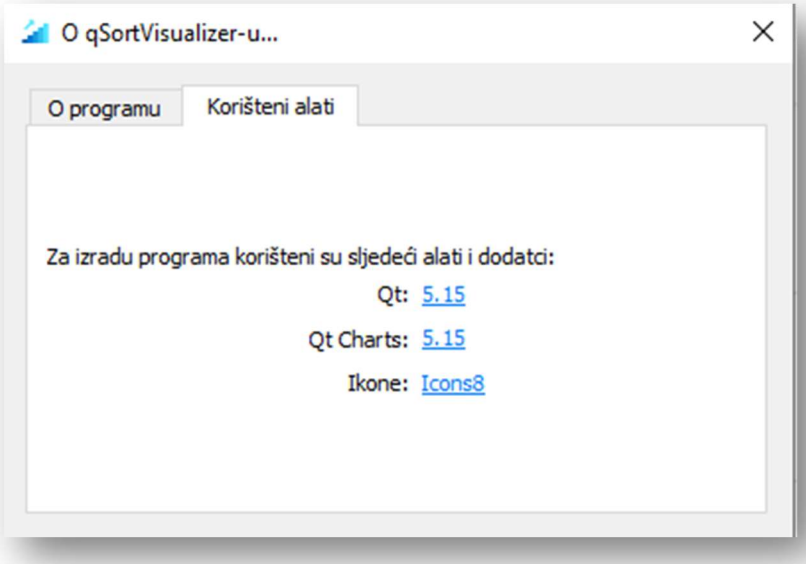


Slika 22: Grafički prikaz usporedbe algoritama prema trajanju

4.3. O programu

Informacije o programu moguće je vidjeti ako se iz alatne trake odabere opcija **Pomoć** i zatim **O programu**. Prilikom odabira otvara se dijaloški okvir s dvije kartice – **O programu** i **Korišteni alati**.

Slika 22: Informacije o programu



Slika 23: Informacije o korištenim alatima

5. Zaključak

Cilj ovoga rada bio je proanalizirati osnovne algoritme sortiranja, pokazati statičku vizualizaciju njihova rada, a potom izraditi rješenje kojim bi se vizualizacija rada algoritma prikazivala na dinamičan način u stvarnom vremenu.

Kako je sortiranje možda i najvažniji problem u računarstvu uopće, nezaobilazan je predmet izučavanja prilikom učenja programiranja. Stoga je implikacija ovoga programskog rješenja prije svega u obrazovne svrhe, odnosno namijenjena je svima onima koji žele naučiti kako ti algoritmi rade, koji od njih su bolji ili gori u različitim okolnostima i zašto.

Izrada ovoga programskog rješenja nije bio nimalo lak zadatak. Iako se program, po svojoj zamisli, čini vrlo jednostavnim, za njegovu je realizaciju bilo potrebno puno istraživanja, truda i vremena. Jedan od većih problema bio je osmisliti i implementirati vlastiti objekt koji će predstavljati element polja i koji će se vizualizirati, a kako se puno stvari odvija istovremeno, bilo je potrebno koncipirati i višedretvenost, kako bi sve funkcioniralo kako treba.

Ipak, veliko mi je zadovoljstvo da sam uspio, gotovo u potpunosti, realizirati zamisao koju sam osmislio još kada sam se tek upoznavao s programiranjem, a do sad stečeno znanje uvelike mi je pomoglo u tome.

Iako je program idejno doveden do savršenstva, kao i svaki drugi program ima svoje manjkavosti i potencijalna poboljšanja, a ona se prvenstveno odnose na računala sa slabijim performansama. Naime, kako je sortiranje općenito vrlo zahtjevan proces za obradu, dodatna vizualizacija i grafički efekti na takvim računalima mogu narušiti rad samog programa. S obzirom na to, dobro bi bilo implementirati, primjerice, OpenGL grafičko sučelje, kako bi se postiglo hardverski ubrzano prikazivanje.

Ovim radom, osim što sam još jednom detaljno proučio osnovne algoritme sortiranja, uvelike sam proširio znanje, kako o programiranju općenito, tako i o jeziku C++, koji mi je zbog svojih nepreglednih mogućnosti zasigurno najdraži.

Popis literature

[1] R. Sedgewick, K. Wayne, *Algorithms, Fourth Edition*. Pearson Education, 2011.

[2] Wikipedia (bez dat.), *Sorting algorithms*. Dostupno na: https://en.wikipedia.org/wiki/Sorting_algorithm, pristupano 27. 08. 2022.

[3] Qt službena stranica (bez dat.). Dostupno na: https://www.qt.io/, pristupano 27. 08. 2022.

[4] Wikipedia (bez dat.), *Qt (software)*. Dostupno na: https://en.wikipedia.org/wiki/Qt_(software), pristupano 27. 08. 2022.

[5] R. Manger, *Strukture podataka i algoritmi*, 1. izdanje, Zagreb: Element, 2015.

[6] R. Sedgewick, *Algorithms in C++, Third Edition*. Priceton University, 2009.

[7] A. Lovrenčić, *Apstraktni tipovi podataka i algoritmi*, *Dio 1: Uvod u složenost algoritama i struktura podataka s primjerima pretraživanja i sortiranja*, Sveučilište u Zagrebu, Fakultet organizacije i informatike: Varaždin, 2018.

Popis slika

Popis tablica

Prilozi

1. Github repozitorij: https://github.com/mkovac700/zavrsni-rad