Usporedba algoritama sortiranja

Đuranović, Tomislav

Undergraduate thesis / Završni rad

2017

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Pula / Sveučilište Jurja Dobrile u Puli**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:137:295159

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-04-24**

Repository / Repozitorij:

Digital Repository Juraj Dobrila University of Pula

Sveučilište Jurja Dobrile u Puli
Odjel za informacijsko – komunikacijske znanosti

TOMISLAV ĐURANOVIĆ

USPOREDBA ALGORITAMA SORTIRANJA

Završni rad

Pula, rujan, 2017. godine

Sveučilište Jurja Dobrile u Puli
Odjel za informacijsko – komunikacijske znanosti

TOMISLAV ĐURANOVIĆ

USPOREDBA ALGORITAMA SORTIRANJA

Završni rad

JMBAG: 0303054493

Studijski smjer: Informatika

Predmet: Strukture podataka i algoritmi
Znanstveno područje: Društvene znanosti
Znanstveno polje: Informacijske i komunikacijske znanosti
Znanstvena grana: Informacijski sustavi i informatologija
Mentor: doc. dr. sc. Tihomir Orehovački

Pula, rujan, 2017. godine

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani Tomislav Đuranović, kandidat za prvostupnika Informatike ovime izjavljujem da je ovaj Završni rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Završnog rada nije napisan na nedozvoljen način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

_____

U Puli, _____, _____ godine

IZJAVA

o korištenju autorskog djela

Ja, Tomislav Đuranović dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj završni rad pod nazivom “Usporedba algoritama sortiranja” koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, _____

Potpis

_____

SADRŽAJ

1. UVOD 1
2. UPOTREBA I PRIMJENA SORTIRANJA 3
3. ANALIZA SLOŽENOSTI ALGORITAMA 4
4. VRSTE SORTIRANJA 5
4.1 SORTIRANJE ZAMJENOM ELEMENATA 5
4.1.1 SORTIRANJE IZBOROM NAJMANJEG ELEMENTA 5
4.1.2 SORTIRANJE ZAMJENOM SUSJEDNIH ELEMENATA 7
4.1.3 KOKTEL SORTIRANJE 9
4.1.4 *COMB SORT* 11
4.2 SORTIRANJE UMETANJEM 13
4.2.1 JEDNOSTAVNO SORTIRANJE UMETANJEM 13
4.2.2 SORTIRANJE VIŠESTRUKIM UMETANJEM 15
4.3 REKURZIVNI ALGORITMI ZA SORTIRANJE (*MERGE SORT*) 17
5. USPOREDBA ALGORITAMA 20
5.1 A PRIORI ANALIZA 20
5.2 A POSTERIORI ANALIZA 21
6. ZAKLJUČAK 28
LITERATURA 30
SAŽETAK 33
ABSTRACT 33

1. UVOD

Kod razvoja računalnih programa osnovna podjela je na „statički“ i „dinamički“ dio programa. Statički dio čine podaci, odnosno, ono s čime se radi, dok dinamički dio čine algoritmi (engl. *algorithms*) koji definiraju ono što se radi. Neslužbena definicija algoritma iz knjige *Introduction to Algorithms*, (Cormen et al., 2009, str. 5) glasi: „... algoritam je bilo koji dobro definirani računalni postupak koji uzima neku vrijednost ili skup vrijednosti kao ulaz, te stvara neku vrijednost ili skup vrijednosti kao izlaz.“. Kako bi se algoritmi mogli koristiti i razvijati potrebno ih je definirati, odnosno, zapisati. Svaki algoritam je moguće napisati prirodnim govornim jezikom, npr. engleski, njemački, pseudojezikom koji se sastoji od strukturiranih rečenica s jasnim značenjem i programskim jezicima, npr. C++, PHP i sl. Jedna od primjerna algoritama je sortiranje. Problem koji rješava algoritam uzlaznog sortiranja je od liste koji se sastoji od *n* (a1, a2, a3, ... , an) elemenata, permutacijom dobiti listu tako da vrijedi a1 <= a2 <= a3 <= ... <= a_n (Manger, 2013). Primjer uzlaznog sortiranja cijelih brojeva je sljedeći: brojevi prije sortiranja = 4,8,5,2,1; brojevi nakon sortiranja = 1,2,4,5,8. Dakle, kod uzlaznog sortiranja, podaci su poredani od najmanjeg prema najvećem. a kod silaznog načina sortiranja obratno, gdje je uređaj <= dovoljno zamijeniti s >=. Niz elemenata je moguće sortirati s različitim algoritama, a ispravan je onaj koji rješava problem sortiranja. Kako bi se moglo odrediti koji algoritam je bolji, potrebno ih je analizirati, odnosno procijeniti resurse. Pri tome se misli na resurse koji su potrebni algoritmu za izvršavanje potrebnog posla. Dva najvažnija resursa su prostor i vrijeme. Analiza složenosti vremena se izražava funkcijom koja je izračun svi potrebnih operacija algoritma pri obavljanju posla (usporedbe, dodjela vrijednosti, aritmetičke operacije i sl.). Kod vremenske procjene algoritama sortiranja čest slučaj je da funkcija ne ovisi samo o veličini skupa podataka potrebnih za obradu već i o vrijednostima, odnosno njihovom početnom redoslijedu (Manger, 2013). „Algoritam za sortiranje možda brže sortira “skoro sortirani” niz brojeva, a sporije niz koji je “jako izmiješan”.“ (Ibid, str. 17).

Prema tome, u ovom završnom radu, algoritmi su analizirani u različitim slučajevima ulaznih podataka. Analizirano je sedam algoritama sortiranja (*Selection sort, Insert sort, Bubble sort, Cocktail sort, Comb sort, Shell sort* i *Merge sort*). U poglavlju „Upotreba i primjena sortiranja“ su opisane potrebe za sortiranjem kao i primjeri istih. Zatim, poglavlje „Analiza složenosti algoritama“ opisuje vrste analiza, klasifikaciju algoritama prema složenosti te notacije. U četvrtom poglavlju je svaki algoritam zasebno obrađen što uključuje opis, primjer na nizu brojeva, analizu složenosti te programski kod napisan u jeziku C++. Nakon toga, u petom poglavlju se nalazi usporedba prethodno navedenih algoritama, te na kraju zaključak svega navedenog.

2. UPOTREBA I PRIMJENA SORTIRANJA

Sortiranje u stvarnom životu je aktivnost koja se svakodnevno koristi. Primjerice, sortirati se može novac, karte i slično. Pretraživanje i korištenje stvari, odnosno podataka je lakše ukoliko su oni sortirani, te je upravo to razlog za sortiranje. Sortirati je moguće prema početnom slovu, broju, boji i sl., kao i uzlazno i silazno. Primjer sortiranja je prikazan na sljedećoj slici (Slika 1.).

Slika 1. Primjer sortiranja (Dhanvani, *freefeast*, 2013)

Primjena sortiranja je veoma bitna i u računalnoj znanosti kako navodi Cormen et. al (2009, str. 5 - 6): „Zato što ga mnogi programi koriste kao prijelazni korak, sortiranje je temeljna aktivnost u računalnoj znanosti, te kao rezultat toga, imamo veliki broj dobrih algoritama sortiranja na raspolaganju.“. Potreba za sortiranjem je veoma važna jer olakšava svakodnevne aktivnosti. „Na primjer, kako bi se pripremili izvodi računa, banke trebaju sortirati čekove prema njihovom broju.“ (Ibid, str. 124). Također, programski paketi poput *Microsoft Office* koriste algoritme sortiranja, isto tako i operativni sustavi imaju mogućnost sortiranja, npr. za datoteke prema nazivu, datumu, memoriji, tipu podatka i sl.

3. ANALIZA SLOŽENOSTI ALGORITAMA

Algoritme je moguće analizirati kako bi se utvrdilo koji algoritam je bolji, odnosno, koji algoritam troši manje resursa u odnosu na drugi, a da pri tome bude funkcionalan i učinkovit. Dva osnovna resursa kojima se promatra algoritam su prostor i vrijeme. Prostor koji algoritam koristi pri izvođenju je količina memorije računala, a vrijeme potrebno algoritmu od početka do kraja izvođenja se mjeri u broju vremenskih jedinica. Vrijeme se označava kao funkcija $T(n)$, gdje je n veličina ulaznog niza. Ukoliko se analizira algoritam koji sortira niz brojeva, tada je n duljina tog niza. Procjena vremena izvršavanja algoritma odgovara broju operacija koje su potrebne algoritmu. Funkcija $T(n)$ nije u potpunosti precizna, razlog tome je što neke od operacije imaju dulje vrijeme izvršavanja u odnosu na druge (Manger, 2013). Dakle, procjena vremena izvršavanja algoritma pripada tzv. „a priori“ analizi složenosti, te se ona obavlja prije stvarnog mjerenja. Suprotno tome, „a posteriori“ analiza daje precizniju procjenu stvarne razine vjerojatnosti. „A posteriori“ analiza izražava veću vjerojatnost manjeg odstupanja rezultata od „a priori“ analize (Tempo, Calafiore, Dabbene, 2005). Kako ulazni, odnosno početni podaci ne ovise samo o veličini, nego i o drugim atributima (u slučaju sortiranja o početnom redoslijedu elemenata), postoje tri slučaja učinkovitosti:

- Najgora učinkovitost - najgori slučaj ulaznih podataka (trajanje algoritma je najdulje)
- Najbolja učinkovitost - najbolji slučaj ulaznih podataka (trajanje algoritma je najkraće)
- Prosječna učinkovitost - nasumični ulazni podaci (Levitin, 2012).

Navedene učinkovitosti, odnosno trajanja algoritma moguće je povezati s notacijama koje se koriste pri izračunu analize složenosti. One su dio asimptotske analize čija je svrha izraziti složenost algoritma bez obzira na računalo na kojem se on izvodi, već matematički, putem notacija. Tri notacije koje se koriste su: Ω (*omega*), O (*big-O*) i Θ (*theta*), (Rathi, *greeksforgeeks*, n.d.). Kako je prethodno spomenuto, funkcija $T(n)$ nije u potpunosti točna, stoga je dovoljno odrediti njezin red veličine. Ako je procjenom složenosti utvrđeno da je $T(n)=5n^3+2n^2+3n$, tada funkcija $T(n)$ proporcionalno raste s

$5n^3$, a ostali članovi se zanemaruju, odnosno $T(n)=O(n^3)$, (Manger 2013). Definicija notacije „veliko *O*“ glasi: „Neka su $T(n)$ i $f(n)$ dvije realne funkcije definirane na skupu prirodnih brojeva. Kažemo da je $T(n) = O(f(n))$ ako postoji konstanta $c > 0$ i prirodni broj n_0 tako da za sve prirodne brojeve $n \geq n_0$ vrijedi: $|T(n)| \leq c \cdot |f(n)|$.“, (Ibid, str. 18)

Theta notacija glasi: za danu funkciju $g(n)$ označenu kao $\Theta(g(n))$ vrijedi $\Theta(g(n)) = f(n)$ ako postoje pozitivne konstante c_1, c_2 i n_0 tako da $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ za svaki $n \geq n_0$. *Omega* notacija glasi: za danu funkciju $g(n)$ označenu kao $\Omega(g(n))$ vrijedi $\Omega(g(n)) = f(n)$ ako postoje pozitivne konstante c i n_0 tako da $0 \leq cg(n) \leq f(n)$ za svaki $n \geq n_0$, Cormen et. al (2009). „A priori“ analize algoritama obrađenih u ovom završnom radu su definirane notacijama koje mogu biti jedne od sljedećih redova veličine:

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < \ldots < O(2^n) < O(n!)$, (Manger, 2013).

4. VRSTE SORTIRANJA

Algoritmi sortiranja se prema načinu, odnosno, vrsti sortiranja mogu podijeliti u skupine koje se međusobno razlikuju, te u kojima se nalaze različiti algoritmi. Prema Mangeru (2013), vrste algoritama sortiranja su sljedeće:

- Sortiranje zamjenom elemenata
- Sortiranje umetanjem
- Rekurzivni algoritmi za sortiranje

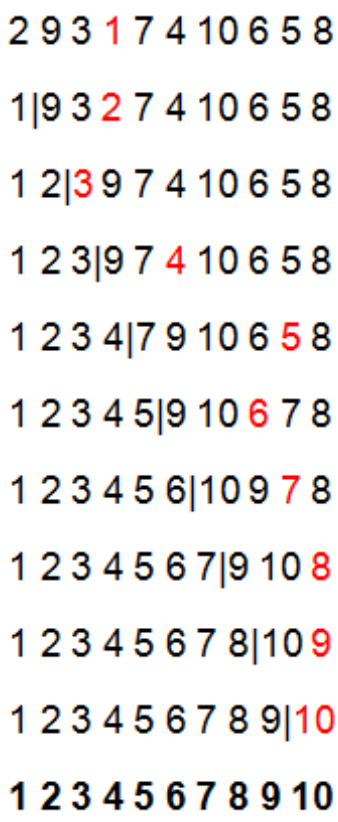
4.1 SORTIRANJE ZAMJENOM ELEMENATA

Sortiranje pri kojem elementi, ako je potrebno, mijenjaju pozicije nakon usporedbe spadaju u skupinu algoritama sortiranja zamjenom elemenata. U sljedećim se potpoglavljima nalaze četiri algoritma koji odgovaraju takvom opisu.

4.1.1 SORTIRANJE IZBOROM NAJMANJEG ELEMENTA

Sortiranje izborom najmanjeg elementa (engl. *Selection sort*) je algoritam koji sortira niz elemenata tako da prolazi poljem i odabire element s najmanjom vrijednosti.

Nakon toga, odabrani element mijenja poziciju s početnim elementom. Svakim sljedećim prolazom kroz polje izostavljaju se već zamijenjeni, odnosno sortirani elementi. Za polje veličine *n*, potrebno je *n*-1 prolazaka kroz polje (Manger, 2013). Na sljedećoj slici (Slika 2.) je prikazano sortiranje cijelih brojeva primjenom algoritma sortiranja izborom najmanjeg elementa. Polje se sastoji od 10 elemenata koji su prema vrijednosti nesortirani. Elementi, odnosno, brojevi označeni crvenom bojom su brojevi s najmanjom vrijednosti u nesortiranom dijelu polja (desno od okomite crte), te oni mijenjaju poziciju s prvim elementom u nesortiranom dijelu polja. Nakon devet prolazaka kroz polje elemenata, elementi su uzlazno sortirani.



Slika 2. Primjer sortiranja izborom najmanjeg elementa

Programski kod algoritma sortiranja izborom najmanjeg elementa je prikazan na sljedećoj slici (Slika 3.). Algoritam se sastoji od funkcije *SelectionSort* koja prima dva argumenta, polje cijelih brojeva, te veličinu tog polja. Na početku funkcije inicijalizirane su cjelobrojne varijable koje su korištene pri sortiranju. Funkcija *swap* služi za zamjenu pozicija dva elementa koji se navode kao argumenti te funkcije. U primjerima ovog

završnog rada argumenti funkcije su elementi polja koje je potrebno sortirati. Funkcija je korištena iz standardne biblioteke programskog jezika C++.

```
void SelectionSort(int polje[], int n){
    int i,j,min;
    for(i=0; i<n; i++){
        min=i;
        for(j=i+1; j<n; j++){
            if(polje[j] < polje[min]) min=j;
            swap(polje[i], polje[min]);
        }
    }
}
```

Slika 3. Sortiranje izborom najmanjeg elementa - C++ kod (Prilagođeno prema Mangeru, 2013, str. 104)

Vremenska analiza složenosti, odnosno, ukupni broj operacija ovisi o broju usporedbi i o broju pridruživanja. Broj usporedbi pri prvom prolasku iznosi n-1, dok se svakim sljedećim prolaskom smanjuje, dakle broj usporedbi je $n(n-1)/2$. Prilikom prolaska kroz polje elementi mijenjaju pozicije, stoga operacija pridruživanja ima $3(n-1)$. Ukupni broj operacija iznosi: $n(n-1)/2+3(n-1)=O(n^2)$, (Manger, 2013).

4.1.2 SORTIRANJE ZAMJENOM SUSJEDNIH ELEMENATA

Algoritam za sortiranje zamjenom susjednih elemenata (engl. *Bubble sort*) sortira tako da prilikom prolaska kroz polje od početka prema kraju, uspoređuje susjedne elemente. Ukoliko se usporedbom vrijednosti dva elementa utvrdi da njihov redoslijed nije ispravan, mijenjaju se pozicije elementima. Nakon prvog prolaska kroz polje, element s najvećom vrijednosti se nalazi na kraju polja, te se on smatra sortiranim. Svakim sljedećim prolaskom kroz polje, broj elemenata za usporedbu se smanjuje. Kao poboljšanje ovog algoritma moguće je u implementaciju dodati logičku varijablu koja će zaustaviti algoritam ako je ustanovljeno da nije potrebna više niti jedna zamjena (Manger, 2013). Sortiranje zamjenom susjednih elemenata sadrži problem „kornjača“ i „zečeva“. „Kornjače“ su elementi s relativnom malom vrijednošću te su smješteni pri kraju polja, dok su „zečevi“ elementi s velikom vrijednošću i smješteni su na početku

polja. Pošto *Bubble sort* algoritam uspoređuje dva susjedna elementa, „kornjače“ se jednim prolaskom mogu samo za jedno mjesto pomaknuti. Takvim se sortiranjem elementi s malom vrijednosti polako pomiču polako prema početku liste, a elementi s velikom vrijednosti brže prema kraju (Lacey i Box, 1991). Primjer sortiranja zamjenom susjednih elemenata je prikazan na sljedećoj slici (Slika 4.). Elementi označeni bojama su elementi čije se vrijednosti uspoređuju, ukoliko je njihova boja plava, nema potrebe za zamjenom, inače se njihove pozicije mijenjaju. Na primjeru je prikazan samo prvi prolazak, te se može vidjeti da je element s najvećom vrijednosti nalazi na kraju polja. Sljedećim se prolaskom kroz polje taj element više ne uspoređuje.

Slika 4. Primjer sortiranja zamjenom susjednih elemenata

Programski kod algoritma sortiranja zamjenom susjednih elemenata vidljiv je na sljedećoj slici (Slika 5.). Algoritam se sastoji od funkcije *BubbleSort* koja prima dva

argumenta, polje cijelih brojeva, te veličinu tog polja. Algoritam pomoću dvije *for* petlje prolazi kroz polje uspoređujući dva susjedna elementa, te, ukoliko prethodni element ima veću vrijednost nego sljedeći, dolazi do zamjene pozicija.

```
void BubbleSort(int polje[], int n){
    int i,j;
    for(i=0; i<n-1; i++){
        for(j=0; j<n-1-i; j++){
            if(polje[j+1] < polje[j])
            swap(polje[j+1], polje[j]);
        }
    }
}
```

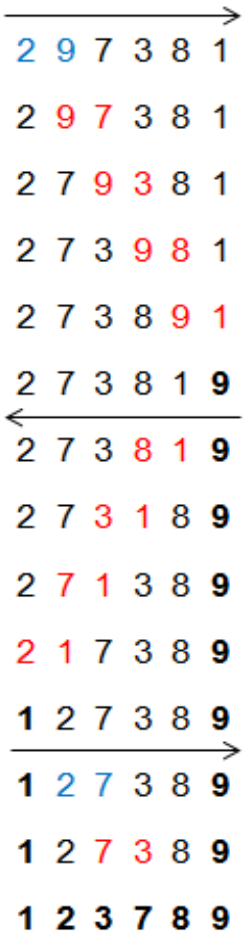
Slika 5. Sortiranje zamjenom susjednih elementa - C++ kod (Prilagođeno prema Mangeru, 2013, str. 105)

Složenost u najgorem slučaju znači da je prilikom svakog prolaska kroz polje potrebno svakom elementu zamijeniti poziciju. Dakle, prvim bi prolaskom bilo n-1 usporedbi i n-1 zamjena, drugim prolaskom n-2 usporedbi i n-2 zamjena, itd. Broj operacija u takvom slučaju iznosi: $4(n-1)+4(n-2)+...+4*1=4n(n-1)/2=2n(n-1)$, odnosno, $O(n^2)$, (Manger, 2013).

4.1.3 KOKTEL SORTIRANJE

Koktel sortiranje (engl. *Cocktail sort*) je izvedena verzija sortiranja zamjenom susjednih elemenata. Princip sortiranja je veoma sličan sortiranju zamjenom susjednih elemenata, no, razlika je u prolascima kroz polje. Koktel sortiranje prolazi kroz polje i uspoređuje dva susjedna elementa, te, nakon što se element s najvećom vrijednosti pozicionira na kraj polja, algoritam se „vraća“ na početak polja pronalazeći element s najmanjom vrijednosti. Sljedećim prolaskom kroz polje, algoritam započinje usporedbu elemenata od drugog do pretposljednjeg elementa. Svakim idućim prolaskom, prilikom usporedbe elemenata, izostavljaju se već sortirani elementi, te se sortiranje bazira prema sredini polja (Gupta i Jaroliya, 2008.). Primjer koktel sortiranja se nalazi na sljedećoj slici (Slika 6.). Sortiran je niz od sedam brojeva, crvenom bojom su označeni elementi kojima je potrebna zamjena, a plavom ako zamjena nije potrebna. Strelice

određuju smjer sortiranja (prema kraju polja i prema početku polja). Primjer odgovara poboljšanoj verziji algoritma koji zaustavlja sortiranje ukoliko više nema zamjena.



Slika 6. Primjer koktel sortiranja

Programski se kod koktel sortiranja nalazi na sljedećoj slici (Slika 7.). Funkcija *CocktailSort* sadrži jednu *while* petlju koja se izvršava sve dok ima zamjena. Ukoliko nema zamjena algoritam se zaustavlja. Nakon prolaska do kraja polja, ono se smanjuje za jedan (--kraj), odnosno pri povratku se poveća (++pocetak). *For* petlje služe za prolaske kroz polja, te za usporedbu elemenata.

```
void CocktailSort(int polje[], int n){
    bool zamjena = true;
    int pocetak = 0;
    int kraj = n-1;

    while(zamjena){
        zamjena = false;
        for(int i = pocetak; i < kraj; ++i){
            if(polje[i] > polje[i + 1]){
                swap(polje[i], polje[i+1]);
                zamjena = true;
            }
        }
        if(!zamjena)
            break;
        zamjena = false;
        --kraj;
        for(int i = kraj - 1; i >= pocetak; --i){
            if(polje[i] > polje[i + 1]){
                swap(polje[i], polje[i+1]);
                zamjena = true;
            }
        }
        ++pocetak;
    }
}
```

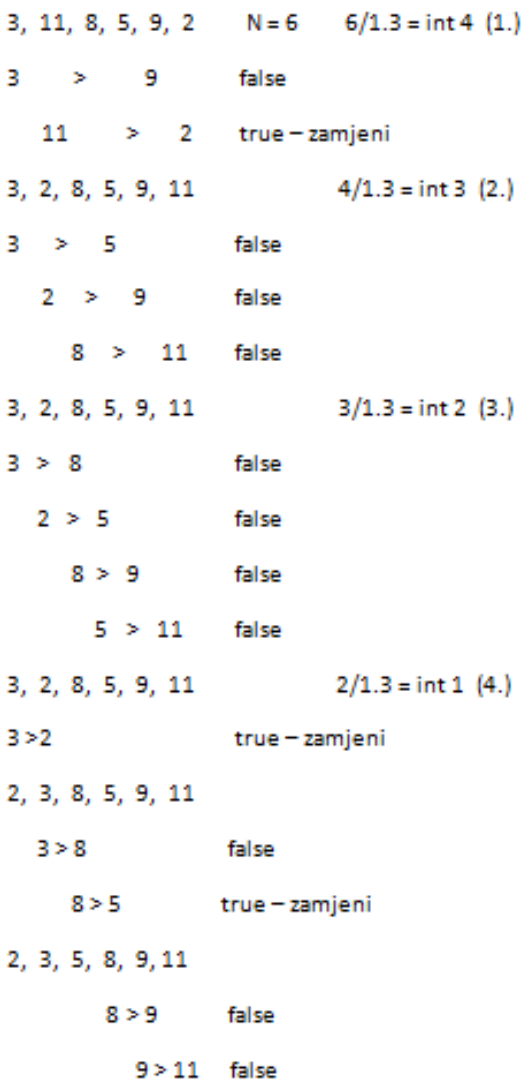
Slika 7. Koktel sortiranje - C++ kod (Prilagođeno prema Agrawal, *geeksforgeeks*, n.d.)

Kako navode Alnihoud J. i Mansi R. (2010), složenost algoritma koktel sortiranja je ista kao i kod *Bubble Sort* algoritma iz istih razloga. Izračun broja usporedbi i broja zamjena elemenata iznosi $O(n^2)$.

4.1.4 *COMB SORT*

Engl. *Comb sort* je također kao i koktel sortiranje jedna od izvedenih, odnosno poboljšanih verzija sortiranja zamjenom susjednih elemenata. Poboljšana je u smislu rješavanja problema „kornjača“ i „zečeva“. *Comb sort* uspoređuje udaljene elemente uz pomoć „koraka“ koji predstavlja razmak između dva elementa. „Korak“ se odnosi na cijelo polje elemenata, dakle omogućuje da se elementi s velikom vrijednosti koji su na početku polja pozicioniraju na kraj polja, i obrnuto za male vrijednosti. Optimalan faktor za izračun razmaka iznosi 1.3. Prvi „korak“ se izračuna dijeljenjem veličine polja s faktorom 1.3, a svaki sljedeći dijeljenjem trenutnog koraka s istim faktorom. Ukoliko je izračunat „korak“ od 9 ili 10, tada će on iznositi 11. To je poboljšanje algoritma

izbjegavanjem „kornjača“ pri razmaku od jedan (Lacey i Box, 1991). Primjer *Comb* sortiranja je prikazan na sljedećoj slici (Slika 8.).



Slika 8. Primjer Comb sortiranja

Programski kod napisan u C++ jeziku je prikazan na sljedećoj slici (Slika 9.). Uz glavnu *CombSort* funkciju korištena je i pomoćna funkcija „noviKorak“ koja služi za izračun novog razmaka, odnosno koraka. Funkcija prima jedan argument, a to je trenutni korak te isti dijeli s faktorom 1.3. Ako je rezultat manji od jedan, korak će iznositi jedan. A ako je korak 9 ili 10, tada će iznositi 11. U funkciji *CombSort* se nalazi jedna *do-while* petlja koja se izvršava sve dok je „korak“ veći od jedan. U toj petlji se poziva i pomoćna

funkcija koja izračunava novi „korak“, te *for* petlja u kojoj se uspoređuju elementi i ukoliko je potrebno mijenjaju njihove pozicije.

```
int noviKorak(int korak){
    korak=korak/1.3;
    if(korak < 1){
        korak=1;
    }
    if(korak==9 || korak==10){
        korak=11;
    }
    return korak;
}

void CombSort(int polje[], int n){
    int i,j;
    int korak=n;
    do{
        korak=noviKorak(korak);
        for(i=0; i<n-korak; i++){
            if(polje[i]>polje[i+korak]){
                swap(polje[i], polje[i+korak]);
            }
        }
    }
    while(korak>1);
}
```

Slika 9. Comb sortiranje - C++ kod (Prilagođeno prema Karira, IDeserve, n.d.)

Prema Gagalowicz i Philips (2009) izračun analize složenosti *Comb* sortiranja u najgorem i prosječnom slučaju iznosi $O(n \log n)$.

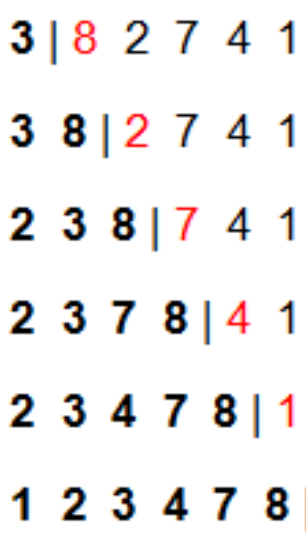
4.2 SORTIRANJE UMETANJEM

Kako navodi Nyhoff (2005., str. 728): „Sortiranje umetanjem se temelji na ideji konstantnog umetanja elementa u već sortiranu listu, tako da nakon umetanja elementa lista i dalje ostane sortirana.“.

4.2.1 JEDNOSTAVNO SORTIRANJE UMETANJEM

Jednostavno sortiranje umetanjem (engl. *Insertion sort*) radi na način da elemente iz nesortiranog dijela polja ubacuje u sortirani dio polja. Na početku sortiranja prvi element se smatra sortiranim, a svi ostali elementi nesortirani. Svakim prolaskom

kroz polje, algoritam uzima prvi element iz nesortiranog dijela polja i ubacuje ga u sortirani dio polja, na točno određenu poziciju. Nakon svakog prolaska, sortirani dio polja se poveća za jedan, a nesortirani dio smanji za jedan. Polje je sortirano nakon što su svi elementi umetnuti na pravo mjesto (Manger, 2013). Primjer sortiranja umetanjem nalazi se na sljedećoj slici (Slika 10.). Brojevi podebljani crnom bojom označuju sortirani dio niza, a brojevi crvene boje označuju element koji je sljedeći na redu za umetanje u sortirani dio polja.



Slika 10. Primjer jednostavnog sortiranja umetanjem

Algoritam *InsertionSort* napisan u programskom jeziku C++ je prikazan na sljedećoj slici (Slika 11.). Funkcija *InsertionSort* sadrži dvije *for* petlje koje svaki element iz polja umeću na pravo mjesto.

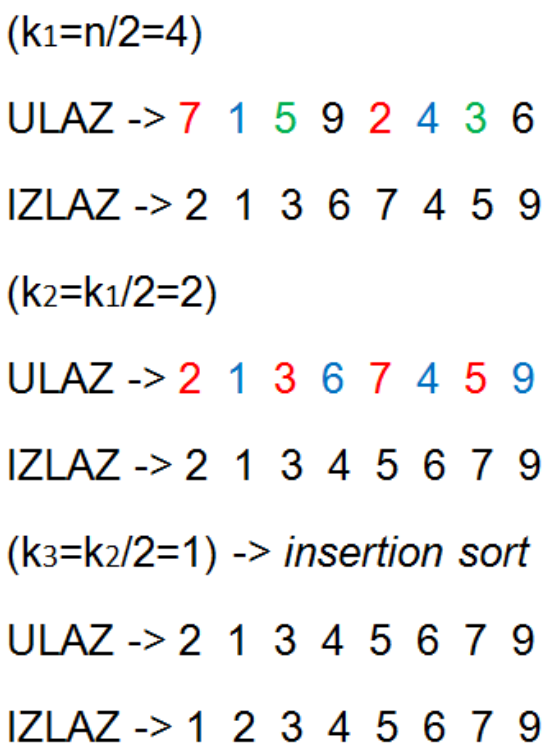
```
InsertionSort(int polje[], int n){
    int i,j,x;
    for(i=1; i<n; i++){
        x=polje[i];
        for(j=i; j>=1 && polje[j-1]>x; j--)
            polje[j]=polje[j-1];
            polje[j]=x;
    }
}
```

Slika 11. Jednostavno sortiranje umetanjem - C++ kod (Prilagođeno prema Mangeru, 2013, str. 108)

Analiza složenosti ovog algoritma je kvadratna. U *n*-tom prolasku unatrag kroz sortirani dio polja, element koji je potrebno umetnuti na određenu poziciju se uspoređuje s drugim elementima koji se pomiču za jedno mjesto. Dakle, u najgorem slučaju postoji *n* usporedbi i otprilike isto toliko pridruživanja. Broj operacija iznosi: 2*1+2*2+...+2*(*n*-1)=*n*(*n*-1), što iznosi O(n^2), (Manger, 2013).

4.2.2 SORTIRANJE VIŠESTRUKIM UMETANJEM

Sortiranje višestrukim umetanjem (engl. *Shell sort*) je algoritam koji sortira na način da se nizu veličine *n* odredi korak, odnosno pomak s kojim se kasnije uspoređuju vrijednosti elemenata. Korak *k* se izračuna tako da se veličina niza podjeli s brojem dva, pa zatim pri svakom sljedećem prolazu se podijeli korak s dva. Elementi se uspoređuju uz pomoć koraka, odnosno, prvi element se uspoređuje s elementom koji slijedi nakon onoliko elemenata kolika je vrijednost koraka. U zadnjem prolazu kroz niz, korak iznosi jedan, te je postupak isti kao i kod jednostavnog sortiranja umetanjem (Manger, 2013). Primjer sortiranja višestrukim umetanjem je prikazan na sljedećoj slici (Slika 12.). Sortiran je niz od osam brojeva. Prvi korak je rezultat dijeljenja veličine niza s brojem dva, zatim drugi korak je izračun prvog koraka i broja dva, a treći izračun drugog koraka i broja dva. „Ulaz“ označuje poredak elemenata prije sortiranja, dok „Izlaz“ označuje elemente nakon. Bojama su prikazani elementi za usporedbu.



Slika 12. Primjer sortiranja višestrukim umetanjem

Programski kod algoritma sortiranja višestrukim umetanjem je prikazan na sljedećoj slici (Slika 13.). Kod sadrži tri *for* petlje koje određuju novi „korak“, te ubacuju elemente na određeno mjesto u polju.

```
void ShellSort(int polje[], int n){
    int i,j,korak;
    int temp;
    for(korak=n/2; korak>0; korak/=2){
        for(i=korak; i<n; i++){
            temp=polje[i];
            for(j=i; j>=korak && polje[j-korak]>temp; j-=korak)
            polje[j]=polje[j-korak];
            polje[j]=temp;
        }
    }
}
```

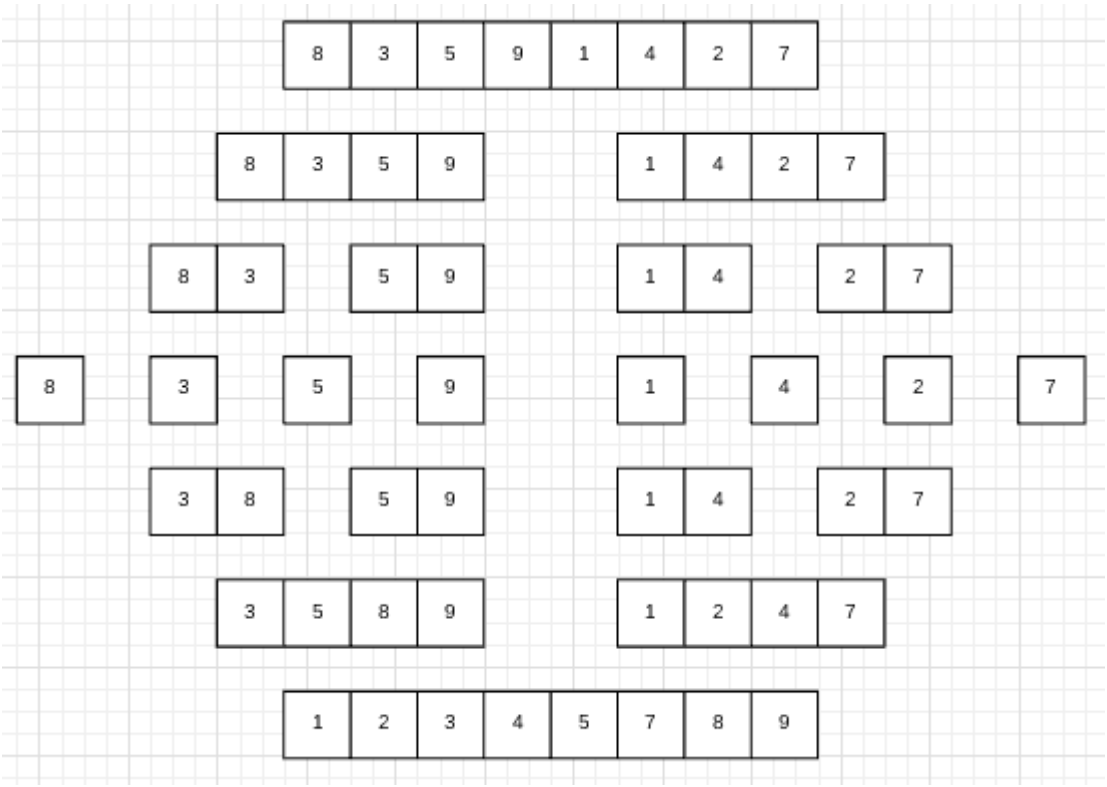
Slika 13. Sortiranje višestrukim umetanjem - C++ kod (Prilagođeno prema Mangeru, 2013, str. 110)

Vremenska analiza složenosti algoritma *Shell sort* ovisi o implementaciji, odnosno definiranju „koraka“ za usporedbu brojeva. Smatra se da je vrijeme izvršavanja algoritma između $O(n)$ i $O(n^2)$, stoga točna procjena je i dalje problem. Originalna sekvenca izračuna „koraka“ je $n/2$, $n/4$, ... , 1. Postoje i druge sekvence određivanja veličine prema kojima je analiza složenosti drugačija, npr. Hibbardov inkrement: 1, 3, 7, …, $2^k - 1$, Knuthov inkrement: 1, 4, 13, …, $(3^k - 1)/2$ (Mangal, P., 2016).

4.3 REKURZIVNI ALGORITMI ZA SORTIRANJE (*MERGE SORT*)

U knjizi *A Practical Introduction to Data Structures and Algorithm Analysis*, Clifford A. Shaffer (2010, str. 36) navodi da je: „ ... algoritam rekurzivan ukoliko poziva sam sebe za obavljanje pojedinog dijela posla. Kako bi pristup „pozivanja samog sebe“ bio uspješan, mora se podijeliti na manji problem nego početno zadani.“.

Sortiranje pomoću sažimanja (engl. *Merge sort*) koristi metodu oblikovanja algoritma „podijeli pa vladaj“. Ideja sortiranja pomoću sažimanja je da se niz podjeli na dva dijela i rekurzivno sortira lijevi i desni dio, te na kraju sortirane dijelove spoji u jedan niz. Dakle, rekurzivnom metodom se dijeli niz sve dok je to moguće, odnosno dok se svaki element ne nalazi zasebno. Tada se smatra da je taj element sortiran te se elementi uspoređuju i spajaju u jedan niz (Baumgartner i Poljak, 2005). Primjer sortiranja se nalazi na sljedećoj slici (Slika 14.). Početni niz brojeva se rekurzivno podjeli na podnizove. Zatim, spajanjem elemenata se dobije sortirani niz.



Slika 14. Primjer sortiranja pomoću sažimanja

Algoritam sortiranja pomoću sažimanja u programskom jeziku C++ je prikazan na sljedećoj slici (Slika 15.). Isti se sastoji od dvije funkcije. Funkcija *merge* služi sa sažimanje potpolja u polje, dok funkcija *MergeSort* rekurzivno dijeli polje na potpolja i poziva funkciju sa sažimanje.

```
void merge(int polje[], int pomocno[], int l, int k, int r){
    int i, k1, k2, brojac;
    k1 = k-1;
    k2 = l;
    brojac = r-l+1;
    while(l <= k1 && k <= r){
        if(polje[l] <= polje[k]) pomocno[k2++] = polje[l++];
        else pomocno[k2++] = polje[k++];
    }
    while(l <= k1)
        pomocno[k2++] = polje[l++];
    while(k <= r)
        pomocno[k2++] = polje[k++];
    for(i = 0; i < brojac; i++, r--)
        polje[r] = pomocno[r];
}

void MergeSort(int polje[], int pomocno[], int lijevo, int desno){
    int sredina;
    if(lijevo < desno){
        sredina = (lijevo + desno) / 2;
        MergeSort(polje, pomocno, lijevo, sredina);
        MergeSort(polje, pomocno, sredina+1, desno);
        merge(polje, pomocno, lijevo, sredina+1, desno);
    }
}
```

Slika 15. Sortiranje pomoću sažimanja - C++ kod (Prilagođeno prema Mangeru, 2013, str. 112 - 113)

Analiza složenosti je sljedeća, prilikom rekurzivnog poziva algoritma, vrijeme je proporcionalno duljini polja koje će nastati. Svi rekurzivni pozivi algoritma rade s poljima čija je duljina jednaka početnom polju. Stoga, izračun vremena za sve rekurzivne pozive je $O(n)$, razina ima $\log_2 n+1$, pa je ukupna složenost u najgorem slučaju $O(n \log n)$, (Manger, 2013).

5. USPOREDBA ALGORITAMA

Usporedba prethodno navedenih algoritama je izrađena prema „a priori“ vremenskoj analizi složenosti u tri slučaja kao i „a posteriori“ analiza. Prva analiza odgovara najboljem slučaju, odnosno sortiranju već sortiranog polja. Druga analiza je procjena, odnosno testiranje u prosječnom slučaju kada se polje sastoji od nasumično poredanih elemenata. Zadnja analiza je u najgorem slučaju što znači sortiranje obrnuto sortiranog polja.

5.1 A PRIORI ANALIZA

A priori analiza procjene vremena prikazana je za sedam algoritama sortiranja obrađenih u ovom završnom radu, te u tri prethodno navedena slučaja. Na sljedećoj tablici (Tablica 1.) su prikazane procjene složenosti algoritama. U najgorem slučaju *Shell Sort* ima složenost $n(\log n)^2$, *Merge* i *Comb Sort* linearno logaritamsku, a ostali imaju kvadratnu složenost. Kod prosječnog slučaja su rezultati isti. Kao najbolji slučaj samo *Selection Sort* ima kvadratnu složenost, a ostali algoritmi ili linearnu ili linearno logaritamsku.

Tablica 1. Procjene složenosti algoritama (Dharmajee i Ramesh, 2012; Gagalowicz i Philips, 2009)

	Vremenska	analiza	složenosti
ALGORITAM	Najbolji slučaj	Prosječni slučaj	Najgori slučaj
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Cocktail Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Shell Sort	$\Omega(n)$	$\Theta(n (\log n)^2)$	$O(n (\log n)^2)$
Merge Sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$
Comb Sort	-	$\Theta(n \log n)$	$O(n \log n)$

5.2 A POSTERIORI ANALIZA

Za a posteriori vremensku analizu korišten je programski jezik Dev C++ 5.11, a funkcije za mjerenje vremena su iz biblioteke „biblioteka_vrijeme.cc“. Analiza obuhvaća mjerenje vremena za različiti broj elemenata u polju (500, 1.000, 10.000, 50.000 i 100.000). Rezultati mjerenja su izraženi u sekundama, a performanse računala na kojemu je provedena analiza su sljedeće:

- Procesor: Intel Core i3-4030U, 1.90GHz
- RAM: 4,00 GB
- Operacijski sustav: Windows 7 (64 bit)

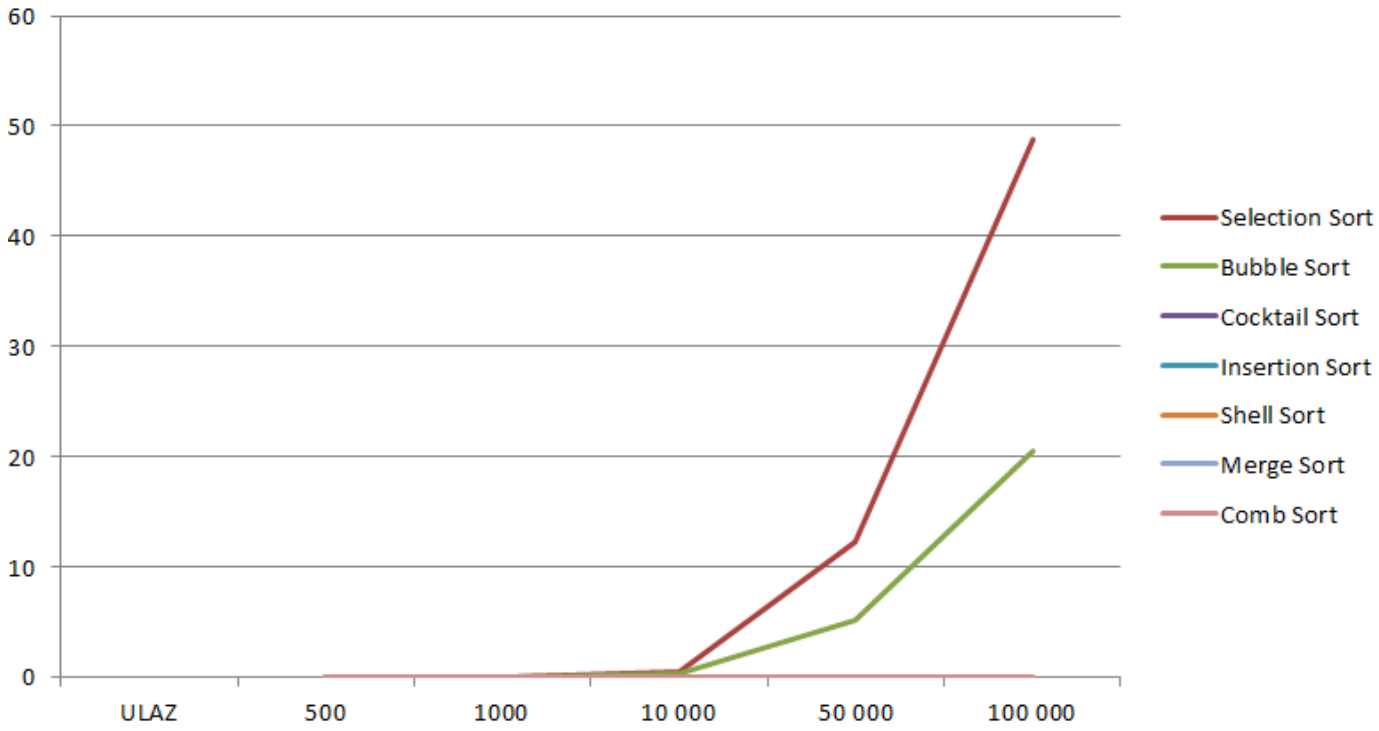
Sljedeće tablice prikazuju analizu u prethodno navedenim slučajevima. Druga tablica prikazuje vremensku analizu složenosti u najboljem slučaju za sedam algoritama. Vidljivo je da *Selection* i *Bubble Sort* odstupaju od ostalih algoritama po izmjerenom vremenu. Složenost *Cocktail Sort* algoritma je u najboljem slučaju ista kao i npr. *Bubble Sort* algoritmu, no vrijeme sortiranja je puno manje *Cocktail* algoritmu. Razlog tomu je što je on poboljšana verzija *Bubble* algoritma ne samo u prolascima kroz polje u oba smjera, već i u zaustavljanju sortiranja ako pri prolasku više nema zamjena.

Tablica 2. Analiza algoritama u najboljem slučaju

	ULAZ	n=500	n=1000	n=10 000	n=50 000	n=100 000
ALGORITAM						
Selection Sort		0,001	0,005	0,491	12,21	48,773
Bubble Sort		0,001	0,002	0,205	5,113	20,472
Cocktail Sort		<0,001	<0,001	<0,001	<0,001	0,001
Insertion Sort		<0,001	<0,001	0,001	0,001	0,001
Shell Sort		<0,001	<0,001	0,001	0,005	0,011
Merge Sort		<0,001	<0,001	0,001	0,008	0,026
Comb Sort		<0,001	<0,001	0,001	0,015	0,017

Podaci iz prethodne tablice odgovaraju sljedećem grafikonu (Grafikon 1.). X os predstavlja broj elemenata pri sortiranju, a Y os vrijeme potrebno za sortiranje, odnosi

se na sve grafikone. Vidljivo je da se algoritmi *Selection Sort* i *Bubble Sort* pri veličini polja od 10.000 počinju isticati u odnosu na druge, dok je pri veličini polja do 10.000 teško odrediti koji ima manje ili veće vrijeme.



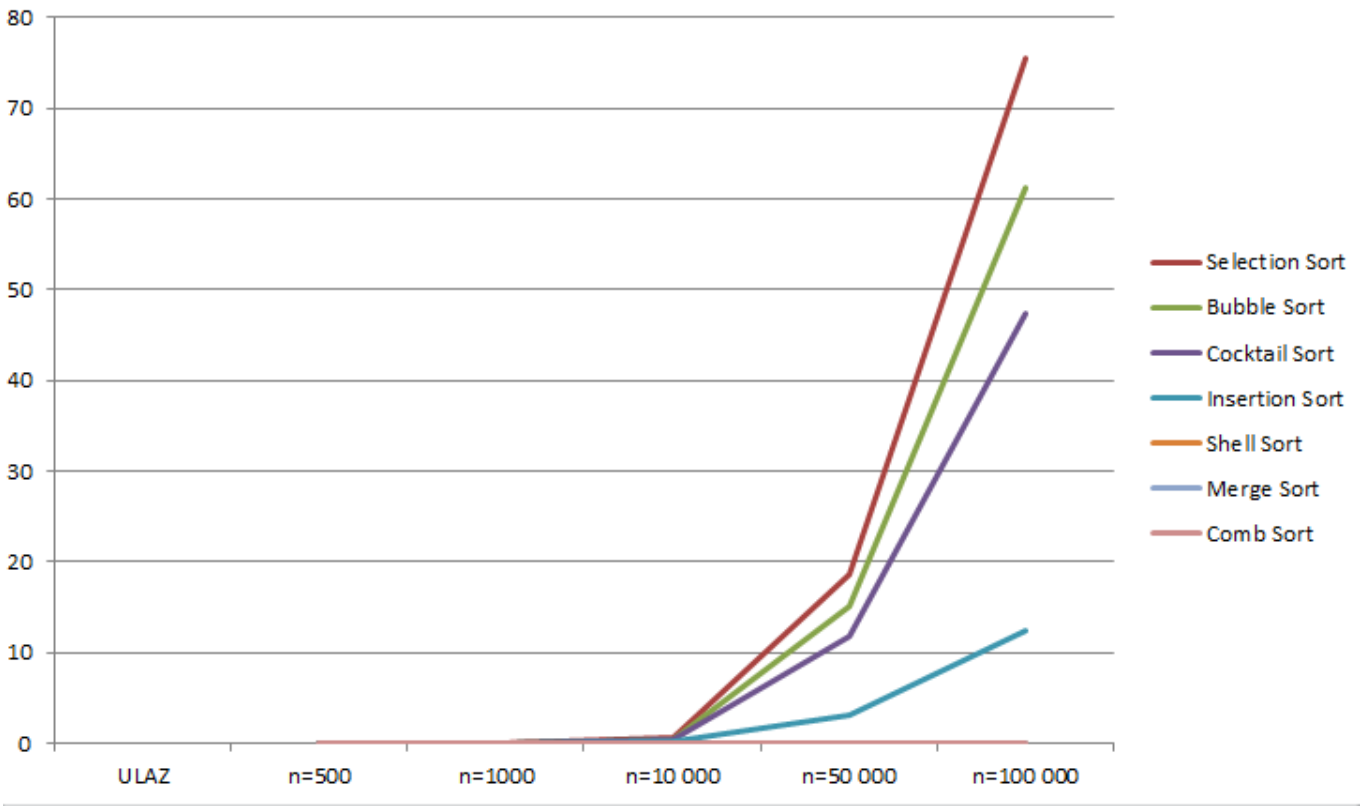
Grafikon 1. Usporedba algoritama u najboljem slučaju

Na sljedećoj tablici (Tablica 3.) je prikazana analiza algoritama u prosječnom slučaju. Pri ovoj analizi elementi su nasumično poredani u polju, te zatim sortirani uzlazno. S najmanjim izmjerenim vremenom ističu se algoritmi *Shell Sort, Merge Sort* i *Comb Sort*, a kao dva najlošija *Bubble Sort* i *Selection Sort*.

Tablica 3. Analiza algoritama u prosječnom slučaju

	ULAZ	n=500	n=1000	n=10 000	n=50 000	n=100 000
ALGORITAM						
Selection Sort		0,003	0,007	0,734	18,648	75,523
Bubble Sort		0,002	0,004	0,535	15,055	61,304
Cocktail Sort		0,001	0,004	0,440	11,783	47,300
Insertion Sort		0,001	0,001	0,129	3,118	12,435
Shell Sort		<0,001	0,001	0,003	0,022	0,047
Merge Sort		<0,001	<0,001	0,002	0,015	0,031
Comb Sort		<0,001	<0,001	<0,001	0,015	0,031

Grafički prikaz podataka iz tablice 2. je prikazan na sljedećem grafikonu (Grafikon 2.). Također, kao i na prethodnom grafikonu, tek se pri sortiranju 10.000 elemenata uočavaju razlike u izmjerenom vremenu.



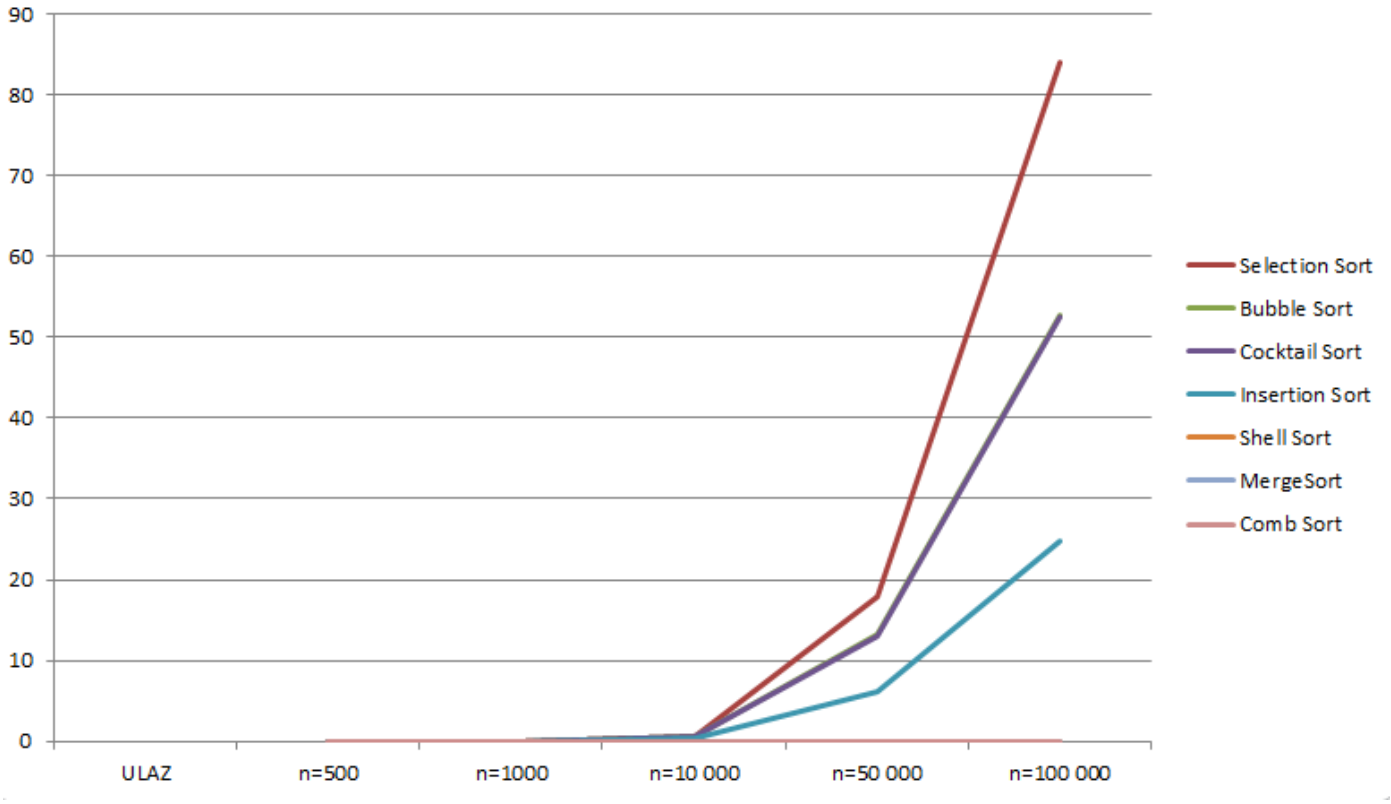
Grafikon 2. Usporedba algoritama u prosječnom slučaju

Posljednja tablica (Tablica 4.) je prikaz rezultata mjerenja u najgorem slučaju sortiranja, odnosno, pri suprotno sortiranom polju. Vrijeme izvođenja pojedinih algoritama je u odnosnu na prethodne slučajeve povećano pri većoj količini podataka. Kao najbrži, pokazali su se algoritmi *Shell Sort* i *Merge Sort*, a kao najsporiji *Selection Sort*.

Tablica 4. Analiza algoritama u najgorem slučaju

	ULAZ	n=500	n=1000	n=10 000	n=50 000	n=100 000
ALGORITAM						
Selection Sort		0,001	0,006	0,608	17,894	83,989
Bubble Sort		0,003	0,008	0,524	13,187	52,866
Cocktail Sort		0,002	0,008	0,515	13,099	52,623
Insertion Sort		<0,001	0,003	0,249	6,209	24,867
Shell Sort		<0,001	<0,001	0,002	0,008	0,016
MergeSort		<0,001	<0,001	0,001	0,008	0,018
Comb Sort		<0,001	<0,001	0,001	0,010	0,016

Grafički prikaz vremenske analize sortiranja u najgorem slučaju je prikazan sljedećim grafikonom (Grafikon 3.). Vrijeme potrebno za sortiranje se povećava u odnosu na prošla dva slučaja sortiranja za pojedine algoritme. *Bubble Sort* i *Cocktail Sort* imaju slično izmjereno vrijeme, pa se teško uočavaju razlike na grafikonu.



Grafikon 3. Usporedba algoritama u najgorem slučaju

Funkcija za mjerenje vremena iz biblioteke „biblioteka_vrijeme.cc“ se nalazi na sljedećoj slici (Slika 16.). Korištene su funkcije „vrijeme_pocetak()“, „vrijeme_kraj()“ i „vrijeme_proteklo()“.

```
#include<time.h>
#include<iostream>
using namespace std;
clock_t vrijeme1,vrijeme2;
double razlika;
double vrijeme_pocetak(){
  vrijeme1=clock();
    return (double)vrijeme1;
};
double vrijeme_kraj(){
    vrijeme2=clock();
    return (double)vrijeme2;
};
double vrijeme_proteklo(){
    razlika=vrijeme2-vrijeme1;
    return razlika;
};
```

Slika 16. Funkcije za mjerenje vremena (Orehovački, n.d.)

Glavna funkcija koja je korištena pri testiranju algoritama u programskom jeziku C++ je prikazana na sljedećoj slici (Slika 17.). Na početku je inicijalizirano cjelobrojno polje veličine 500 (pri testiranju algoritama na drugim veličinama, polju je izmijenjena veličina kao i u ostatku koda). Prva *for* petlja služi za popunjavanje polja nasumičnim brojevima, funkcija *rand* je korištena iz standardne biblioteke „<cstdlib>“ programskog jezika. Druga for petlja služi za ispisivanje elemenata polja prije sortiranja. Nakon toga između funkcija „vrijeme_pocetak()“ i „vrijeme_kraj()“ koje određuju početak i kraj mjerenja vremena, se nalazi funkcija, odnosno algoritam *CombSort* koji je definiran iznad *main* funkcije. Treća *for* petlja je ista kao i druga, te ispisuje brojeve nakon sortiranja. Za ispis proteklog vremena sortiranja služi funkcija „vrijeme_proteklo()“. Povratna vrijednost te funkcije je podijeljena s 1000 kako bi se vrijeme izmjereno za manje veličine polja moglo izraziti. Kod sortiranja u slučaju već sortiranih i obrnuto sortiranih polja, nakon *for* petlje u kojoj se polje „puni“ nasumičnim brojevima, nalazi se funkcija koja sortira polje onako kako to određuje slučaj analize. Odnosno, ta funkcija „priprema“ polje za sljedeće sortiranje (ono čije se vrijeme izvršavanja mjeri). Ukoliko je potrebno sortirati silazno, implementaciju algoritama je moguće jednostavno prilagoditi zamjenom znakova <= s >=.

```
int main(){

    int polje[500];

    for(int i=0; i<500; i++){
        polje[i]=rand()%500;
    }

    cout << "Brojevi prije sortiranja: " << endl;
    for(int i=0; i<500; i++){
        cout << polje[i] << ", ";
    }

    vrijeme_pocetak(); // pocetak mjerenja
    CombSort(polje, 500);
    vrijeme_kraj(); // kraj mjerenja

    cout << endl;
    cout << "Brojevi nakon sortiranja: " << endl;
    for(int i=0; i<500; i++){
        cout << polje[i] << ", ";
    }
    cout << endl;
    cout << "Vrijeme potrebno za sortiranje polja: " << vrijeme_proteklo()/1000 << endl;
    cout << endl;

    system("pause");
    return 0;
}
```

Slika 17. Main funkcija

6. ZAKLJUČAK

Kao osvrt na prethodno navedeno, može se zaključiti da je sortiranje veoma bitno u svijetu informacijsko-komunikacijskih tehnologija, kao i u životu. Razlog tome je potreba za brzim pretraživanjem i snalaženjem pri korištenju određenog sadržaja. Danas postoje mnogi algoritmi koji rješavaju problem sortiranja na različite načine. Neki od njih sortiraju zamjenom elemenata, ubacivanjem istih ili rekurzivno, no, zajednička karakteristika im je što sortiraju na principu usporedbe. Dakle, neovisno da li je potrebno sortirati uzlazno ili silazno, znakove ili brojeve, oni se uspoređuju kako bi se znalo koji prethodi, odnosno, slijedi kojemu. Što se tiče funkcionalnosti, smatra se da je ispravan onaj algoritam koji od ulaznih podataka, sortiranih ili nesortiranih, stvori sortirane podatke. Kako bi se pojedini algoritam mogao poboljšati ili zamijeniti boljim, potrebno ga je analizirati prema određenim kriterijima, a to su vrijeme i memorijski prostor. U ovom završnom radu analizirano je sedam algoritama sortiranja koji su međusobno uspoređeni prema potrebnom vremenu za sortiranje određene veličine polja cijelih brojeva. Analizirani su u tri slučaja: najbolji slučaj (već sortirano polje), prosječni slučaj (nasumični redoslijed elemenata) i najgori slučaj (obrnuto sortirano polje), te s pet različitih veličina polja (500, 1.000, 10.000, 50.000 i 100.000). Pri najboljem slučaju sortiranja *Insertion Sort* je sortirao elemente u najmanjem vremenu iako je linearne složenosti kao i *Bubble Sort* koji je sortirao iste elemente za relativno duže vremena. U prosječnom su slučaju skoro svi algoritmi imali veće izmjereno vrijeme bez obzira na veličinu ulaznog polja. U najgorem slučaju, kao najveće izmjereno vrijeme pripada *Selection Sort*-u, a algoritam s manjim izmjerenim vremenom u odnosu na druge algoritme iz skupne sortiranja zamjenom elemenata je *Comb Sort*. On je poboljšana verzija *Bubble Sort* algoritma, koji i u praksi rješava problem „kornjače“ i „zeca“ tako što veliki brojevi brže dolaze na kraj polja, a mali na početak polja. Dakle, postoji mnogo algoritama sortiranja koji uspješno rješavaju zadani problem. Za neke od njih postoje poboljšanja kako bi se njihove performanse unaprijedile, te kako bi se učinkovitost povećala. Procjenom analize složenosti je moguće okvirno predvidjeti koji je algoritam bolji od kojega kako bi korisnik bio u mogućnosti odabrati zaista onaj koji je prigodan za pojedinu situaciju, no analiza stvarnog vremena to bolje pokazuje. Najbolji algoritam za

sortiranje je teško odrediti jer situacija u kojoj se on primjenjuje ovisi o puno karakteristika. Stoga, proučavanje algoritama sortiranja, implementacija novih, poboljšanja već postojećih i slično je i danas izazov mnogim stručnjacima.

LITERATURA

Agrawal, R. (n.d.) *Cocktail Sort*. Preuzeto s: http://www.geeksforgeeks.org/cocktail-sort/, Pristupljeno: 17.9.2017.

Alnihoud, J. i Mansi, R. (2010) *An Enhancement of Major Sorting Algorithms*. Jordan: Department of Computer Science, Al al-Bayt University. Preuzeto s: http://ccis2k.org/iajit/PDF/vol.7,no.1/9.pdf, Pristupljeno: 5.9.2017.

Baumgartner, A. i Poljak, S. (2005) *Sortiranje podataka*. Osijek: Elektrotehnički Fakultet, Sveučilište u Osijeku. Preuzeto s: http://hrcak.srce.hr/4056, Pristupljeno: 5.9.2017.

Cormen, H. T. et al. (2001) *Introduction to Algorithms*. London: The MIT Press

Dhanvani, P. (2013) *Insertion Sort | Pseudo Code of Insertion Sort | Insertion Sort in Data Structure*. Preuzeto s: http://freefeast.info/general-it-articles/insertion-sort-pseudo-code-of-insertion-sort-insertion-sort-in-data-structure/, Pristupljeno: 31.8.2017.

Dharmajee, R. i Ramesh, B (2012) *Experimental Based Selection of Best Sorting Algorithm*. International Journal of Modern Engineering Research (IJMER)

Gagalowicz, A. i Philips, W. (2009) Computer Vision/Compuer Graphics Collaboration Techniques. France: Springer

Gupta, I. C. i Jaroliya D. (2008) *IT Enabled Practices and Emerging Management Paradigms*, New Delhi: Prestige Institute of Management and Research

Karira, S. (n.d.), *Algorithm/Insights*. Preuzeto s: http://www.ideserve.co.in/learn/comb-sort, Pristupljeno: 13.9.2017.

Lacey, S. i Box, R. (1991) *A Fast Easy Sort*. BYTE Magazine

Levitin, A. (2012) *Introduction to The Design and Analysis of Algorithms*. USA: Pearson

Magner, R. (2014) *Strukture podataka i algoritmi*. Zagreb: Prirodoslovno - matematički fakultet u Zagrebu

Mangal, P. (2016) *SHELL SORT ALGORITHM- EXPLANATION, IMPLEMENTATION AND COMPLEXITY*. Preuzeto s: https://www.codingeek.com/algorithms/shell-sort-algorithm-explanation-implementation-and-complexity/, Pristupljeno: 14.9.2017.

Nyhoff, R. L. (2005) *ADTs, Data Structures, and Problem Solving with C++*. USA: Alan Apt

Orehovački, T. (n.d.) *Analiza složenosti algoritama*. Pula: Sveučilište Jurja Dobrile u Puli

Rathi, A. (n.d.) *Analysis of Algorithms | Set 3 (Asymptotic Notations)*. Preuzeto s: http://www.geeksforgeeks.org/analysis-of-algorithms-set-3asymptotic-notations/, Pristupljeno: 19.9.2017.

Shaffer, A. C. (2010.) *A Practical Introduction to Data Structures and Algorithm Analysis,* Blacksburg: Department of Computer Science, Virginia Tech

Tempo, R., Calafiore, G. i Dabbene, F. (2005) *Randomized Algorithms for Analysis and Control of Uncertain Systems*. USA: Springer - Verlag

PRILOZI

SLIKE:

Slika 1. Primjer sortiranja 3
Slika 2. Primjer sortiranja izborom najmanjeg elementa 6
Slika 3. Sortiranje izborom najmanjeg elementa - C++ kod 7
Slika 4. Primjer sortiranja zamjenom susjednih elemenata 8
Slika 5. Sortiranje zamjenom susjednih elementa - C++ kod 9
Slika 6. Primjer koktel sortiranja 10
Slika 7. Koktel sortiranje - C++ kod 11
Slika 8. Primjer Comb sortiranja 12
Slika 9. Comb sortiranje - C++ kod 13
Slika 10. Primjer jednostavnog sortiranja umetanjem 14
Slika 11. Jednostavno sortiranje umetanjem - C++ kod 15
Slika 12. Primjer sortiranja višestrukim umetanjem 16
Slika 13. Sortiranje višestrukim umetanjem - C++ kod 16
Slika 14. Primjer sortiranja pomoću sažimanja 18
Slika 15. Sortiranje pomoću sažimanja - C++ kod 19
Slika 16. Funkcije za mjerenje vremena 26
Slika 17. Main funkcija 27

TABLICE:

Tablica 1. Procjene složenosti algoritama 20
Tablica 2. Analiza algoritama u najboljem slučaju 21
Tablica 3. Analiza algoritama u prosječnom slučaju 23
Tablica 4. Analiza algoritama u najgorem slučaju 24

GRAFIKONI:

Grafikon 1. Usporedba algoritama u najboljem slučaju 22
Grafikon 2. Usporedba algoritama u prosječnom slučaju 23
Grafikon 3. Usporedba algoritama u najgorem slučaju 25

SAŽETAK

Usporedba algoritama sortiranja

Važnost algoritama sortiranja je velika jer se koriste svakodnevno. Primjerice, prilikom sortiranja datoteka na računalu njihov redoslijed može biti određen prema nazivu, datumu, veličini i sl. Postoje razni algoritmi sortiranja koji se razlikuju prema određenim karakteristikama. Kako bi se utvrdilo koji je bolji, odnosno kojemu se mogu poboljšati performanse, potrebno ih je analizirati, točnije procijeniti potrebne resurse. Takva se procjena naziva „a priori“ analiza složenosti, dok stvarni izračun pripada „a posteriori“ analizi. Dva osnovna resursa algoritama su prostor i vrijeme. Procjena potrebnog vremena algoritmu za rješavanje definiranog problema se izražava kroz funkciju $T(n)$ koja nije u potpunosti točna jer ne može odrediti stvarno vrijeme izvršavanja algoritma. Stoga, stvarno vrijeme izraženo u vremenskim jedinicama točnije određuje resurse. Usporedbom prema navedenim resursima je moguće utvrditi koji algoritam je bolji od ostalih, ali za određene situacije sortiranja koje ovise o početnom redoslijedu sadržaja, veličini i sl.

Ključne riječi: algoritam, sortiranje, analiza složenosti, usporedba

ABSTRACT

Comparison of sorting algorithms

The importance of sorting algorithms is significant because they are used on a regular basis. For instance, while sorting files on a computer, their sequence can be determined by name, date, size etc. There are a lot of sorting algorithms which differ from each other by certain characteristics. In order to determine which of them is better or whether some of them need their performances improved, they have to be analyzed and the needed resources have to be estimated. That kind of estimation is called „a priori“ complexity analysis, whereas the real calculation belong to the „a posteriori“ one. Two basic algorithm resources are space and time. The estimation of the time required for the algorithm to resolve a specifically defined problem is shown through the $T(n)$

function. This function isn't completely accurate because it can not define the real time of the algorithm implementation. Therefore, the real time, which is defined in time units, specifies resources more accurately. With the comparison towards the given resources, it is possible to define which algorithm is better but only for certain situations of sorting which depend on the initial sequence on content, size etc.

Key words: algorithm, sorting, complexity analysis, comparison