Algoritmi za sortiranje u programskom jeziku C++

Jakus, Alen

SVEUČILIŠTE U RIJECI
FILOZOFSKI FAKULTET U RIJECI
ODSJEK ZA POLITEHNIKU

Algoritmi za sortiranje u programskom jeziku C++

Završni rad

Mentor završnog rada: doc. dr. sc. Marko Maliković

Student: Alen Jakus

Rijeka, 2016.

SVEUČILIŠTE U RIJECI
Filozofski fakultet
Odsjek za politehniku

Rijeka, Sveučilišna avenija 4

Povjerenstvo za završne i diplomske ispite

U Rijeci, 07. travnja, 2016.

ZADATAK ZAVRŠNOG RADA

(na sveučilišnom preddiplomskom studiju politehnike)

Pristupnik: Alen Jakus

Zadatak: Algoritmi za sortiranje u programskom jeziku C++

Rješenjem zadatka potrebno je obuhvatiti sljedeće:

1. Napraviti pregled algoritama za sortiranje.
2. Opisati odabrane algoritme za sortiranje.
3. Dijagramima prikazati rad odabranih algoritama za sortiranje.
4. Opis osnovnih svojstava programskog jezika C++.
5. Detaljan opis tipova podataka, izvedenih oblika podataka, naredbi i drugih elemenata iz programskog jezika C++ koji se koriste u rješenjima odabranih problema.
6. Opis rješenja koja su dobivena iz napisanih programa.
7. Cjelokupan kôd u programskom jeziku C++.

U završnom se radu obvezno treba pridržavati **Pravilnika o diplomskom radu i Uputa za izradu završnog rada sveučilišnog dodiplomskog studija**.

Zadatak uručen pristupniku: 07. travnja 2016. godine

Rok predaje završnog rada: _____

Datum predaje završnog rada: _____

Zadatak zadao:

Doc. dr. sc. Marko Maliković

FILOZOFSKI FAKULTET U RIJECI
Odsjek za politehniku

U Rijeci, 07. travnja 2016. godine

ZADATAK ZA ZAVRŠNI RAD

(na sveučilišnom preddiplomskom studiju politehnike)

Pristupnik: **Alen Jakus**

Naslov završnog rada: **Algoritmi za sortiranje u programskom jeziku C++**

Kratak opis zadatka: Napravite pregled algoritama za sortiranje. Opišite odabrane algoritme za sortiranje. Dijagramima prikažite rad odabranih algoritama za sortiranje. Opišite osnovna svojstva programskog jezika C++. Detaljno opišite tipove podataka, izvedene oblike podataka, naredbe i druge elemente iz programskog jezika C++ koji se koriste u rješenjima odabranih problema. Opišite rješenja koja ste dobili iz napisanih programa. Priložite cjelokupan kôd u programskom jeziku C++.

Zadatak uručen pristupniku: **07. travnja 2016. godine**

Ovjera prihvaćanja završnog rada od strane mentora: _____

Završni rad predan: _____

Datum obrane završnog rada: _____

Članovi ispitnog povjerenstva: 1. predsjednik - _____

2. mentor - _____

3. član - _____

Konačna ocjena: _____

Mentor

_____

Doc. dr. sc. Marko Maliković

Sažetak

U ovom radu bit će prikazano nekoliko rješenja problema sortiranja podataka u jednodimenzionalnom nizu elemenata. Kako se u praksi često pojavljuje potreba za sortiranjem podataka od velike je koristi pronaći što brži i efikasniji algoritam. Prikazat će se i praktične izvedbe algoritama za sortiranje koje će biti testirane na različitim nasumičnim skupovima podataka sa više ili manje elemenata. Kako bi lakše shvatili brzinu rada algoritma priložena će biti tablica u kojoj možemo lagano zaključiti koji je od algoritama koliko brz odnosno koliko mu je vremena potrebno da obavi posao sortiranja niza. Samo testiranje biti će realizirano u programskom jeziku C++, gdje je napisan i sam kôd. Isto tako biti će objašnjeni svi korišteni tipovi podataka, naredbe i ostali elementi kôda.

Ključne riječi:

- Algoritam
- Sortiranje
- Složenost algoritma
- Usporedba algoritama sortiranja

Kazalo

1. Uvod

Kao uvod u algoritme obradit ćemo nekoliko osnovnih načina koji su prikladni za sortiranje manjih nizova i datoteka. Iz više je razloga bolje analizirati jednostavnije metode nego neke kompleksne kao npr., što na jednostavan način možemo naučiti terminologiju sortiranja te nam to omogućava lakše proučavanje kompleksnijih algoritama.

Za jedan problem moguće je imati više različitih algoritama koji riješavaju taj problem. Uobičajeno je da nisu svi ti algoritmi jednako dobri u smislu brzine izvođenja. Algoritmi sortiranja kojima ćemo se baviti u ovom radu:

- Bubble sort
- Insertion sort
- Heap sort
- Shell sort
- Quick sort

Neke od tih metoda pripadaju osnovnoj vrsti algoritama, npr., bubble sort i insertion sort, dok drugi algoritmi pripadaju složenijoj vrsti algoritama poput quick sorta.

Analiziramo algoritme odnosno njihove složenosti kako bi odabrali najbolji mogući algoritam za riješavanje zadanog problema. Za to koristimo dvije analize:

- A priori analiza - učinkovitost algoritma se provjerava korištenjem matematičkih metoda
  - analizira algoritam neovisno o implementaciji
  - može se koristiti i prije nego što je algoritam implementiran
- A posteriori analiza - učinkovitost algoritama procjenjuje se pokretanjem tih algoritama na računalu

Mi u ovom radu iznosimo i A priori i A posteriori rezultate.

Kao cilj ovog završnog rada je usporediti spomenute algoritme sortiranja te vidjeti kako se ponašaju kroz različitit broj elemenata niza koje sortiramo.

2. Algoritmi sortiranja i njihova složenost

U računalnoj znanosti algoritmi za sortiranje su algoritmi koji na određeni način postavljaju elemente niza (polja, liste) u određeni redoslijed. Sortiranja koja se najviše koriste jesu *leksikografska* (sortiranje slova i riječi) te *brojevna* (sortiranje brojeva) kojima se posebno bavimo u ovom radu.

Sortirati možemo:

- uzlazno – od manjeg prema većem
- silazno – od većeg prema manjem

Definiran je jako velik broj algoritama koji se mogu implementirati u različitim programskim jezicima pa da nabrojimo nekoliko:

- Bubble sort, Insertion sort, Selection sort, Heap sort, Quick sort, Radix sort, Counting sort, Introsort, Timsort, Cubesort, Shell sort, Library sort, Cocktail sort, Smooth sort, Block sort, Gnome sort, Patience sorting, Strand sort, Cycle sort...

Neki od njih (npr. quick sort) postoji već implementiran kao gotova funkcija u biblioteci C programskog jezika.

U okvirima programiranja, složenost algoritama ovisna je o broju naredbi, odnosno instrukcija koje program treba izvršiti kako bi se dobio konačan rezultat dok kod petlji ovisi o broju iteracija. Za većinu problema broj iteracija ovisi o broju elemenata koji se procesira kao npr., duljina liste koja se sortira, duljina liste koja se pretražuje, itd...Ako taj broj označimo sa *n*, tada je složenost algoritma neka funkcija od *n*. Drugim riječima, broj naredbi koje izvršava neka funkcija za *n* elemenata je funkcija od *n* koju označujemo sa *f(n).*

Za mjerenje brzine odnosno učinkovitosti algoritma nije bitno imati točan broj iteracija već samo tzv. *O – notaciju (big - O notation)* koja označava red veličine složenosti algoritma.

Složenosti algoritama poredane od najmanje do najveće:

- $\log n$, n, $n*\log n$, n^2, n^3,..., n^k , 2^n, $n!$.

Opširnije o složenosti algoritama slijedi u nastavku rada.

2.1. Bubble sort

Sortiranje u valovima ili mjehuričasto sortiranje (engl. Bubble sort) je jedan od najjednostavnijih algoritama za sortiranje. Može se krenuti sortirati od početka ali isto tako i od kraja liste. Osnovna ideja algoritma je usporedba svaka dva susjedna elementa liste (polja) te im se mjenjaju mjesta ako poredak nije dobar. Nakon jednog prolaska kroz listu najveći element biva "izguran" na kraj liste odnosno ispliva kao mjehurić otkud dolazi i sam naziv algoritma. Takav postupak se ponavlja sve dok cijela lista nije sortirana. Lista se u svakom trenutku dijeli na dvije podliste, sortiranu i nesortiranu, odvojene „konceptualnim zidom".

Slijedi primjer (tablica 1.) koja prikazuje korake sortiranja bubble sorta.

16	**11**	5	8	10	→ zamjena
11	**16**	**5**	8	10	→ zamjena
11	5	**16**	**8**	10	→ zamjena
11	5	8	**16**	**10**	→ zamjena
11	5	8	10 ‖‖	16	→ gotov 1. prolaz; dio liste sortiran
11	**5**	8	10 ‖‖	16	→ zamjena
5	**11**	**8**	10 ‖‖	16	→zamjena
5	8	**11**	**10** ‖‖	16	→zamjena
5	8	10 ‖‖	11	16	→ gotov 2. prolaz; dio liste sortiran
5	**8**	10 ‖‖	11	16	→ nema zamjene
5	**8**	**10** ‖‖	11	16	→ nema zamjene
5	8 ‖‖	10	11	16	→ gotov 3. prolaz; dio liste sortiran
5	**8** ‖‖	10	11	16	→ nema zamjene
5 ‖‖	8	10	11	16	→ gotov 4. prolaz; **lista sortirana**

Tablica 1. Princip rada bubble sorta

2.1.1. Složenost Bubble sorta

Bubble sort stoji jako loše kad je u pitanju složenost algoritama jer ima najlošiju složenost koju označavamo sa O(n^2) gdje je n broj elemenata koji sortiramo. Takva složenost dolazi od toga što za implementaciju koristimo dvije *for* petlje od kojih jedna služi za usporedbu svaka dva susjedna elementa dok druga prolazi preko cijelog djela nesortiranog niza. Čak i drugi algoritmi sa istom složenosti, kao npr. insertion sort, imaju potencijal da budu puno bolji i efikasniji od bubble sorta. Dakle, za veliki n ovaj algoritam nam nije toliko praktičan koliko i kad imamo samo par elemenata za sortiranje pa tu ima prednost zbog svoje jednostavnosti. Isto tako velika prednost koju ima u odnosu na algoritme sa boljom složenosti je sposobnost otkrivanja da imamo sortirani niz ugrađen u algoritam. U tom slučaju kad je na ulazu sortirani niz složenost bubble sorta postaje O(n) što je i najbolji slučaj.

U određivanju složenosti samog algoritma značajnu ulogu imaju pozicije elemenata u nizu pa se tako mali elementi sa kraja niza pomiču na početak jako sporo dok je situacija sa velikim elementima sa početka niza situacija drukčija jer se oni lako kreću prema kraju i mjenjaju.

Programeri su razvili više poboljšanih verzija bubble sorta kao što su *cocktail sort i comb sort* sa čime su dobili puno na efikasnosti i brzini samog algoritma.

2.2. Insertion sort

Sortiranje umetanjem (engl. Insertion sort) je jednostavan algoritam koji je puno brži i efikasniji od ranije spomenutog bubble sorta.

Kod sortiranja umetanjem niz se dijeli na dva dijela, sortirani i nesortirani. Osnovna ideja je uzeti element iz nesortiranog dijela niza te smjestiti na odgovarajuću poziciju u sortiranom dijelu niza. Počinje se gledati od prvog elementa liste te se svi elementi koji su veći od elementa kojeg smještamo u sortiranu listu pomiču za jedno mjesto u desno. Ako imamo listu od n elemenata, bit će potrebno najviše n-1 prolaza kroz listu da bi se podaci sortirali.

U nastavku slijedi primjer (tablica 2.) koji prikazuje korake sortiranja niza. U svakom koraku element koji se razmatra je podebljan. Dio koji je u prethodnom koraku pomaknut ili ostavljen na mjestu zato što je dotad najveći razmatrani bit će podvučen.

12	9	4	99	120	1	3	10
12	**9**	4	99	120	1	3	10
9	12	**4**	99	120	1	3	10
4	9	12	**99**	120	1	3	10
4	9	12	99	**120**	1	3	10
4	9	12	99	120	**1**	3	10
1	4	9	12	99	120	**3**	10
1	3	4	9	12	99	120	**10**
1	3	4	9	10	12	99	120

Tablica 2. Princip rada insertion sorta

Kao što vidimo pri svakom prolasku uzima se jedan element na ulazu, te se povećava sortirana izlazna lista. Dakle, pri prolasku uklanja se jedan element na ulaznoj listi, pronalazi pripadajuće mjesto u sortiranoj listi te ga postavlja tamo. Proces se ponavlja sve dok na ulaznoj listi više nema podataka. Na svakom se prolasku kroz listu provjerava vrijednost ulaznog podatka te ga uspoređuje sa najvećom vrijednosti elementa niza u prethodnoj provjeri. Ako je veći ostavlja element na mjestu te prelazi na sljedeći, odnosno ako je manji pronalazi odgovarajuću poziciju u nizu, pomiče elemente više vrijednosti da napravi mjesto te ga tamo ubacuje.

Puno manja efikasnost kod dužih nizova u odnosu na složenije algoritme poput quick ili heap sorta. Međutim ima i neke prednosti:

- Jednostavna implementacija
- Učinkovitiji u praksi od ostalih jednostavnih algoritama sa istom složenosti (bubble sort, selection sort)
- Učinkovit na malim skupovima podataka
- Stabilan odnosno ne mijenja relativni redoslijed elemenata sa jednakim vrijednostima
- Prilagodljiviji za skupove podataka koji su gotovo sortirani u cijelosti
- Trenutan, online, sortira niz odmah po primanju

2.2.1. Složenost insertion sorta

Kada imamo na ulazu već sortiran niz što je ujedno i najbolji slučaj, tada imamo linearnu složenost O(n). Suprotno tome, jedan od najgorih slučajeva je kada imamo niz u obrnutom redoslijedu kada složenost postaje kvadratna (tj. O(n^2)). Kako imamo i u prosječnom slučaju kvadratnu složenost, sortiranje većih nizova postaje nepraktično. Međutim, što se tiče vrlo malih nizova sortiranje umetanjem slovi kao jedno od najboljih rješenja, odnosno najbržih.

2.3. Shell short

Višestruko sortiranje umetanjem poznato kao sortiranje po Shellu (engl. Shell sort). Osnovna zamisao (D. L. Shell, 1959.) je da se ne uspoređuju susjedni elementi, već da se definira veličina koja će odrediti razmak k_1 (gap) između elemenata koji će se uspoređivati. Prema veličini razmaka niz se dijeli na podnizove i untar svakog podniza elementi se sortiraju primjenjujući prethodno opisan princip jednostavnog sortiranja umetanjem (insertion sort). Nakon sortiranja svih podnizova za definirani razmak k, taj broj se smanjuje i ponavlja se postupak s brojem k_2. U svakoj novoj iteraciji razmak se smanji sve dok se ne dobije vrijednost $k_n = 1$. Za $k_n = 1$ postupak sortiranja svodi se na jednostavno sortiranje umetanjem, ali postupak puno kraće traje budući da je niz jednim dijelom sortiran tijekom prethodno izvedenih postupaka.

Postavlja se pitanje kako odrediti početni razmak k_1, pa ovisno o tome postoje različite varijante implementacije višestrukog sortiranja umetanjem. Prema Shellu početni razmak odnosno pomak određuje se kao *floor(n/2)* gdje je n broj elemenata niza, a svaki novi k se dobije dijeljenjem brojem 2. Funkcija *floor* daje najveći cijeli broj koji je rezultat djeljenja (n/2). U nastavku slijedi primjer sortiranja niza po Shellu (tablica 3).

1. Tri nesortirane podliste – sortira se svaka zasebno

54	26	93	17	77	31	44	55	20	→ podlista 1
54	26	93	17	77	31	44	55	20	→ podlista 2
54	26	93	17	77	31	44	55	20	→ podlista 3

2. Rezultat nakon sortiranja tri podliste

17	26	93	44	77	31	54	55	20	→ podlista 1 sortirana
54	26	93	17	55	31	44	77	20	→ podlista 2 sortirana
54	26	20	17	77	31	44	55	93	→ podlista 3 sortirana
17	26	20	44	55	31	54	77	93	

3. Pomaci za ostatak elementa

17	26	20	44	55	31	54	77	93	→ pomak br. 20 za 1 mjesto
17	20	26	44	55	31	54	77	93	→ pomak br. 31 za 2 mjesta
17	20	26	31	44	55	54	77	93	→ pomak br. 54 za 1 mjesto
17	20	26	31	44	54	55	77	93	→**sortirani niz**

Tablica 3. Princip rada shell sorta

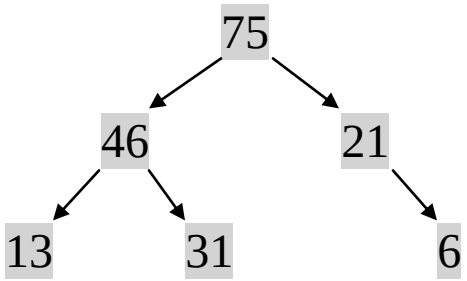
2.3.1. Složenost shell sorta

Ideja shell sortiranja je poopćenje prethodno opisanog jednostavnog sortiranja umetanjem. Ovakav način sortiranja pokušava izbjeći „sporo“ premještanje nekog elementa velike vrijednosti „mjesto po mjesto“, kako je implementirano u algoritmu bubble sort. U trenutku kada se počinju uspoređivati susjedni elementi, niz je već jednim dijelom dobro sortiran. Složenost ovog algoritma iznosi $O(n^{1.5})$, što je bolji rezultat od prethodno opisanih algoritama.

2.4. Heap sort

Sortiranje upotrebom gomile (engl. Heap sort) je algoritam koji radi na principu unutarnjeg sortiranja. Pod pojmom unutarnjeg sortiranja se podrazumijevaju algoritmi za čiju je realizaciju potrebno sve elemente liste smjestiti u operativnu memoriju a zatim ih sortirati.

Ideja je da se svi elementi niza smjeste u jednu vrstu stabla koja naizgled predstavlja binarno stablo koje zovemo gomila ili heap. Drugim riječima, gomila je potpuno binarno stablo gdje se čvorovi mogu uspoređivati nekom uređenom relacijom (npr. <=) i gdje je bilo koji čvor u smislu te relacije veći ili jednak od svoje djece naravno ako uopće postoje (slika 1).



Slika 1. Primjer gomile

Jedna od karakteristika gomile je ta da je vrijednost čvora uvijek veća od vrijednosti lijevog i desnog „sina“ te svaki element gomile odgovara elementima niza.

U implementaciji heap sorta koristimo slijedeće dvije funkcije:

- *Heapify* koja je ključna za očuvanje uređenosti gomile
- *HeapSort* koja sortira polje

Ideja za sortiranje niza je da se najveći element niza, koji je po definiciji heap strukture uvijek u korijenu pomiče na zadnju poziciju, a čitav se niz tretira kao da ima n-1 elemenata, te se ponovno sortira sa funkcijom heapify. Proces se ponavlja sve dok se ne dođe do dva elementa i nakon toga dobijamo rastući niz.

2.4.1. Složenost heap sorta

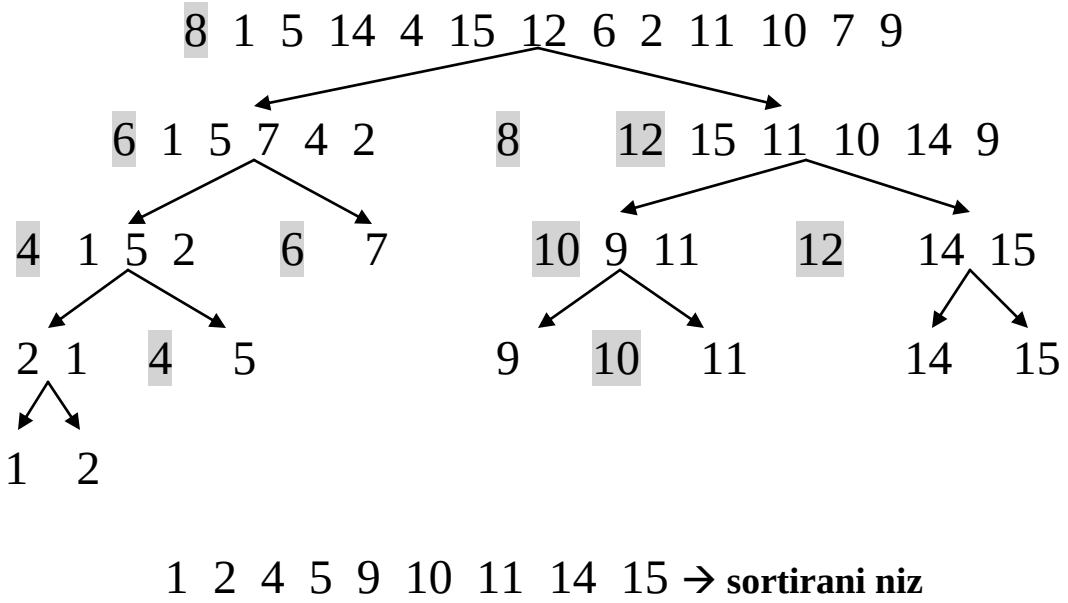
Složenost ovog algoritma je O(log*n*) te kako se to obavlja *n* puta tada je složenost heap sorta O(*n**log*n*).

2.5. Quick sort

Brzo sortiranje (engl. Quick sort) je jedan od najpopularnijih algoritama sortiranja što može zahvaliti svojoj velikoj brzini. Razvio ga je Toni Hor 1960. godine, a radi na principu tehnike *podijeli pa vladaj*. Glavni koraci ove metode su:

- *Podijeli:* ako je veličina problema (ulaza) prevelika da bi ga direktno rješavali, dijelimo problem na dva ili više disjunktnih potproblema
- *Savladaj:* rekurzivno primjenjujemo metodu podijeli i savladaj za rješavanje potproblema
- *Kombiniraj:* uzimamo dobivena rješenja potproblema i spajamo da bi dobili rješenje početnog problema

Osnovna ideja algoritma brzog sortiranja se sastoji u tome da se odabere jedan element sa liste koji će služiti kao „pivot“. To može biti bilo koji element liste te se u odnosu na vrijednost pivota izvorna lista dijeli na dva dijela. U prvom se dijelu nalaze svi elementi s vrijednostima manjim ili jednakim vrijednosti pivota, dok drugi dio liste čine svi elementi s vrijednostima većim od pivota. Tako dobijene liste se opet na isti način dijele dok se ne dobije lista sa samo jednim elementom te kao rezultat dobijemo sortirani niz. Kako bi sve to bolje razumjeli slijedi primjer na koji način radi algoritam quick sort (slika 2).



Slika 2. Princip rada quick sorta

Kod implementacije quick sorta u jeziku C++ imamo dvije funkcije:

- Quicksort - poziva funkciju podijeliNiz koja dijeli niz na dva dijela ovisno o pivotu te rekurzivno poziva funkcije za prvi kao i za drugi dio niza
- PodijeliNiz – funkcija prvo definira pivotni element, zatim pregledava sve elemente niza koje uspoređuje sa pivotom te po potrebi vrši zamjenu elemenata (engl. swap)

Postoje i druge implementacije ovog algoritma sortiranja koje sadrži istu ideju ali malo drugačiju izvedbu kao npr. da se podijeli niz ne na dva već na tri dijela. U nekim situacijama je takva podijela puno efikasnija nego kod klasičnog brzog sortiranja.

2.5.1. Složenost quick sorta

Složenost algoritma puno ovisi o načinu na koji imamo zadan niz te je kod najnepovoljnijeg slučaja složenost $O(n^2)$ dok je u prosječnom slučaju kad imamo uravnotežen niz sa n elemenata složenost $O(n\log n)$.

3. Programski jezik C++

C++ je razvio Bjarne Stroustrup u AT&T Bell laboratoriju 1980-tih godina. Poznat je kao jako moćan i snažan programski jezik, koji dopušta ogromnu kontrolu memorije, brzinu i efikasnost koda. Nekoliko programskih jezika više razine koji su mu prethodili jesu: Fortran, Cobol, Basic, Pascal, C... Strukturno programiranje je u početku bilo dovoljno za velike programe ali su s vremenom i oni postajali sve veći te se težilo razvoju novih odnosno naprednijih programskih jezika kako bi se olakšalo pisanje takvih ogromnih kodova. Tako je jezik C++ izveden iz C jezika. Pokrio je nekoliko njegovih nedostataka C te se polako uvelo *objektno orijentirano programiranje* čime se pojednostavilo pisanje kôda za složene programe. Razvijan je za programiranje složenih aplikacija u praksi te je imao velik utjecaj na razvoj drugih programskih jezika kao što su Java i C#.

3.1. Osnovna svojstva jezika C++

- Jezik opće namjene srednje do visoke razine
- **Imperativni jezik** – program se realizira kroz niz naredbi koje mijenjaju stanje programa odnosno varijabli
- **Objektno orijentirani pristup** – ideja: modelirati probleme i situacije iz stvarnog života pomoću objekata. Pri tome je važno što objekt radi, a ne kako on to radi. To omogućava da se pojedini objekt može po potrebi izbaciti i zamijeniti drugim koji će jednu te istu zadaću obaviti na uspješniji način.
- Podržava i proceduralno programiranje (Jezik C)

3.2. Elementi, naredbe i tipovi podataka korišteni u riješenjima problema

U ovom će poglavlju biti detaljno opisani svi tipovi podatka, naredbe kao i ostali elementi korišteni u riješavanju problema sortiranja.

3.2.1. Tipovi podataka

U jeziku C++ tipove podataka prema vrsti razlikujemo:

- Cijelobrojni tipovi podataka – int
- Realni tipovi podataka – float (obični) i double (dvostruke preciznosti)
- Znakovni tipovi podataka – char i string
- Logički tip podataka - bool

Svaki od njih zauzima određenu količinu memorije u računalu te ima određeni raspon vrijednosti. Sad ćemo to i prikazati u tablici 4., gdje su podebljanim slovima označeni tipovi podataka korišteni u samom kôdu.

Tip podatka	***Veličina u bitovima***	***Raspon vrijednosti***
Char	8	-128 do 127
short int	16	-32768 do 32767
unsigned int	16	0 do 65535
Int	**16**	**-32768 do 32767**
Long	**32**	**-2147483648 do 2147483647**
Float	32	$3{,}4x10^{-38}$ do $3{,}4x10^{38}$
Double	**64**	**$1{,}7x10^{-308}$ do $1{,}7x10^{308}$**
long double	80	$3{,}4x10^{-4932}$ do $1{,}1\ x10^{4932}$

Tablica 4. Tipovi podataka u jeziku C++

Osnovni tip cijelog broja je *int*. Može sadržavati pozitivnu i negativnu vrijednost unutar prikazanih granica. Kada nam je za varijablu potrebno veće područje kao u našem slučaju tada koristimo tip podataka *long*. Realni brojevi sadrže decimalni dio a njihov osnovni tip je *float*, dok smo mi koristili *double* koji daje mogućnost smještanja većeg broja znamenaka sa još većom preciznošću.

3.2.2. Naredbe

Linije programa koje počinju znakom # su pretprocesorske naredbe prevodiocu (engl. compiler) koje se izvršavaju prije prevođenja programa.

- **#include<iostream>** poziva „iostream.h“ datoteku koja sadrži skup naredbi za komunikaciju s programom
- **#include<cstdlib>** poziva „cstdlib.h“ datoteku koja sadrži opis funkcija ulaza/izlaza
- **#include<ctime>** poziva „ctime.h“ datoteku koja sadrži funkciju za računanje realnog vremena

Naredba **using namespace std;** govori prevoditelju da će program koristiti standardne nazive naredbi.

- **Cout** – izlazni tok koji šalje podatke na ekran
- **Cin –** ulazni tok koji prima podatke preko tipkovnice

Svaka naredba u programu završava sa točkom-zarezom (;), osim uvjetne naredbe *if-else*.

- Naredba **if –** omogućava uvjetno grananje toka programa ovisno o tome je li zadovoljen uvjet iza ključne riječi if. Ako je tome tako (logička istina) onda se izvodi blok naredbi koje slijede iza izraza dok u drugom slučaju (logička neistina) se blok naredbi unutar if naredbe preskače te se izvođenje nastavlja od prve naredbe iza bloka.
- Naredba **else** – uvodi se kad želimo da se izvede određen dio programa ukoliko je neispunjen uvjet if naredbe.

Kada imamo više rezultata za neki izraz uvjeta, a za svaki treba provesti različite odsječke programa, u tom je slučaju često preglednije koristiti *switch* grananje.

- Naredba **switch –** prvo se izračunava neki izraz koji daje cijelobrojni rezultat te se ovisno o njemu tok programa preusmjerava na neku od grana unutar switch bloka naredbi

Spomenute naredbe if, else i switch se izvršavaju samo jednom, ali u praksi se često pojavljuje potreba da se neke naredbe moraju izvršiti više puta te se u tom slučaju koriste petlje.

U kôdu smo koristili dvije petlje:

- **For petlja -** koristi se za ponavljanje segmenta kôda koji je zadan unutar bloka naredbi u slučaju kada je unaprijed poznat točan broj ponavljanja bloka naredbi
  - Uvodi se varijabla cjelobrojnog tipa koja predstavlja brojač, uobičajeno se naziva i (j, k...)
  - U for petlji je uobičajena praksa koristiti unarne operatore u izrastu prirasta:
    - ++ za inkrementaciju (uvećavanje za 1); i=i+1 piše se kao *i*++
    - -- za deinkremetnaciju (smanjivanje za 1); i=i-1 piše se kao *i*—
- **While petlja** – koristi se u situacijama kada je potrebno neke naredbe ponavljati sve dok se ne ispuni određeni uvjet
  - Broj izvođenja naredbi unutar petlje nije unaprijed poznat već ovisi o logičkom uvjetu
  - Prvo se provjerava rezultat uvjeta izvođenja kao logički izraz koji ako je jednak logičkoj istini izvodi se blok naredbi te se program vraća ponovo na početak while naredbe dok u suprotnom slučaju ako je rezultat logičkog izraza jednak logičnoj neistini preskače se blok naredbi te se izvođenje nastavlja od prve naredbe iza bloka

Naredbom **system ("pause");** zaustavljamo program dok ne pritisnemo neku tipku što nam je važno prilikom izvođenja programa.

Naredbom **return 0;** glavni program vraća pozivnom programu broj 0 što je poruka operacijskom sustavu da je program uspješno okončan što god on radio.

3.2.3. Funkcije

Zbog preglednosti programa, manje složenosti kôda te lakšeg ispravljanja grešaka poželjno je svaku zasebnu cijelinu izdvojiti kao funkciju. Funkcije su blokovi naredbi koji se izdvajaju iz programa kao zasebne cijeline.

- **main –** glavna funkcija programa
- **iostream, cstdlib, ctime, swap** – funkcije koje su unaprijed definirane u postojećim bibliotekama
- **void** – tip funkcije koji ne daje povratnu vrijednost

U programu su definirane funkcije za svako sortiranje o kojima smo ranije pričali kao i funkcija za generiranje slučajnih brojeva koje ćemo poslije sortirati pomoću svakog algoritma te usporediti rezultate. Pisanje koda na ovakav način donijelo je bolju preglednost i manju složenost kôda.

3.2.4. Niz (polje)

Polje ili niz (engl. array) se definira kao konačan niz istovrsnih podataka. Niz ima svoje ime, tip i veličinu odnosno zauzima neki prostor u memoriji računala. Kod polja možemo objediniti više podataka pod isti naziv i deklarirati kao jedinstvenu varijablu koja sadrži niz podataka. Svakom se pojedinom podatku može pristupiti preko brojčanog indeksa.

Postoje jednodimenzionalna, dvodimenzionalna i višedimenzionalna polja. U našem programu koristili smo samo jednodimenzionalna pa ćemo o njima nešto više reći.

Jednodimenzionalna polja - najjednostavniji oblik polja kod kojeg se elementi polja dohvaćaju preko samo jednog indeksa koji označava njegovu udaljenost od početnog elementa. Isto tako elementi polja su složeni linearno. U nastavku slijedi primjer jednog takvog polja.

Primjer: Polje **o** u koje možemo pohraniti podatke za 10 ocjena

o[0]	o[1]	o[2]	o[3]	o[4]	o[5]	o[6]	o[7]	o[8]	o[9]
3	3	5	4	2	2	4	1	3	5

4. Usporedba algoritama sortiranja

Mnogi algoritmi iste učinkovitosti nemaju istu brzinu za isti ulaz. Postoji nekoliko važnih kriterija za prosuđivanje algoritama. Prvo, algoritme treba uspoređivati na temelju prosječnog, najboljeg i najgoreg slučaja. Neki algoritmi, kao npr. quick sort koji slovi kao jedan od najboljih algoritama, izuzetno dobro obavlja posao za većinu ulaza, dok je za neke druge dosta lošiji. Sa druge strane, algoritmi kao npr. heap sort, ne zavisi toliko o ulaznim podacima.

Drugi kriterij je memorijski prostor koji označava količinu pomoćnog prostora za pohranu i izmjene niza elemenata. Tu imamo tzv. *in-place* algoritme koji ne zahtijevaju nikakav dodatni prostor za obavljanje zadatka (insertion sort, shell sort, bubble sort), dok drugi algoritmi (quick sort) zahtjevaju dodatno mjesto na stogu.

Treći je kriterij stabilnost algoritma. Algoritam je stabilan ako zapisi s jednakim ključevima ostaju u originalnom poretku, odnosno drugim riječima ako iste elemente ostavlja u postojećem poretku.

U tablici 5., *n* je broj elemenata niza koji se sortira. Stupci „Prosječan slučaj“ i „Najgori slučaj“ dati će nam uvid u vremensku složenost svakog sortiranja posebno koju smo prethodno opisali a sad je navodimo samo radi usporedbe.

	Složenost algoritma				
Ime algoritma	Prosječan slučaj	Najbolji slučaj	Najgori slučaj	Memorijski prostor	Stabilnost algoritma
Bubble sort	$O(n^2)$	$O(n)$	$O(n^2)$	Konstantan	Stabilan
Insertion sort	$O(n^2)$	$O(n)$	$O(n^2)$	Konstantan	Stabilan
Heap sort	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	Konstantan	Nije stabilan
Shell sort	$O(n^{1.5})$	$O(n^{1.25})$	$O(n^2)$	Konstantan	Nije stabilan
Quick sort	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	log *n*- prosječni slučaj, *n*- najgori slučaj	Nije stabilan

Tablica 5. Usporedba algoritama sortiranja

4.1. Eksperimenalni rezultati

U ovom radu testirali smo spomenutih pet algoritama sortiranja na nasumičnim nizovima različite duljine. Testiranje je napravljeno kao i sam kôd u programskom okruženju Microsoft Visual Studio 2015 odnosno programskom jeziku C++. Korišteno je računalo sa 1.7 GHz Intel Core 3 procesorom, 4 GB RAM-a memorije, te operacijskim sustavom Windows 10 Pro.

U slijedećoj će tablici biti prikazani izmjereni rezultati brzine izvođenja algoritama.

Broj elemenata (*n*)	Bubble sort	Insertion sort	Heap sort	Shell sort	Quick sort
1000	4 ms	2 ms	5 ms	0,5 ms	0,4 ms
10000	497 ms	214 ms	54 ms	7 ms	4 ms
100000	57,006 sek	26,175 sek	414 ms	63 ms	51 ms
500000	1458,2 sek	620,003 sek	2,456 sek	445 ms	487 ms
1000000	5802,7 sek	1830,2 sek	5,306 sek	1,374 sek	1,116 sek

Tablica 6. Brzine izvođenja algoritama

Iz priloženog se vidi kako za kraće nizove (n=1000) razlika u brzini izvođenja i nije toliko velika koliko je kod dužih nizova (n>10000). Kako bubble sort i insertion sort imaju kvadratnu složenost tako se povećava vrijeme sortiranja. Drugim rječima, njihova se efikasnost drastično smanjuje na nizovima sa više podataka. Općenito, insertion sort je bolji algoritam te je relativno učinkovit za male nizove te se koristi kao dio sofisticiranijih programa. Heap sort nije toliko efikasan za kraće nizove, ali kako ima logaritamsku složenost vidimo da se povećanjem elemenata niza, otprilike u istom rangu povećava i vrijeme izvođenja. To potvrđuje prijašnju konstataciju kad smo govorili o složenosti algoritama. Preostala dva sortiranja shell sort i quick sort pokazali su se vrlo efikasnim posebno na velikim nizovima. Od svih pet testiranih algoritama, najbolja svojstva i najbrža vremena sortiranja pripadaju naravno quick sortu.

Kad smo govorili o A priori i A posteriori analizama rekli smo da su A priori matematičke analize koje se mogu izvodit neovisno o implementaciji algoritma, dok se A posteriori izvodi testiranjem na računalu kao što smo i napravili u ovom radu. Vodeći se tim putem prvo smo usporedili A priori analize algoritma te bi kao rezultat toga bila njegova složenost. Tako imamo algoritme kvadratne složenoti Bubble sort i Insertion sort koji imaju najlošiju složenost te po tom pitanju su najlošiji od analiziranih algoritama, dok je međusobno insertion sort ipak malo efikasniji. Uspoređujući dalje A priori analize vidimo da malo bolju složenost od spomenutih ima poboljšana verzija insertion sorta, shell sort ($O(n^{1.5})$). Ali ipak, algoritmi logaritamske složenosti heap sort i quick sort su u samom vrhu, odnosno najbolji gledajući A priori rezultate.

S druge strane, gledajuću A posteriori rezultate uvjerili smo se da ako je algoritam bolji A priori, ne mora nužno biti bolji i A posteriori. To nam upravo pokazuje slučaj usporedbe shell i heap sorta gdje vidimo kako heap sort ima bolju složenost ali rezultati testiranja to ne potvrđuju te shell algoritam postaje puno brži odnosno efikasniji pa skoro i kao quick sort koji je ipak nedostižan. Ostali algoritmi ostaju u istom poretku kako i A priori, tako i A posteriori.

5. Programski kôdovi

Slijedi prikaz implementacije programskog kôda za svaki pojedini algoritam sortiranja detaljnije opisanih u poglavlju br. 2 kao i ostalih dijelova programa.

5. 1. Implementacija Bubble sorta

```
void bubble_sort(long niz[], int n) {
	int temp;
	for (int i = n - 1; i > 0; i--)
		for (int j = 0; j < i; j++)
			if (niz[j] > niz[j + 1]) {
				temp = niz[j];
				niz[j] = niz[j + 1];
				niz[j + 1] = temp;}
	return;}
```

5. 2. Implementacija Insertion sorta

```
void insertion_sort(long niz[], int n) {
	int temp;
	int j;
	for (int i = 1; i<n; i++) {
		j = i;
		temp = niz[i];
		while (j>0 && temp < niz[j - 1]) {
			niz[j] = niz[j - 1];
			j--;}
		niz[j] = temp;}
	return;}
```

5. 3. Implementacija Shell sorta

```
void insertion_sort(long niz[], int n) {
	int temp;
	int j;
	for (int i = 1; i<n; i++) {
		j = i;
		temp = niz[i];
		while (j>0 && temp < niz[j - 1]) {
			niz[j] = niz[j - 1];
			j--;}
		niz[j] = temp;}
	return;}
```

5. 4. Implementacija Heap sorta

```
void heapify(long niz[], int n, int i)
{
	int najveci = i;
```

```
	int l = 2 * i + 1;
	int r = 2 * i + 2;
	if (l < n && niz[l] > niz[najveci])
		najveci = l;
	if (r < n && niz[r] > niz[najveci])
		najveci = r;
	if (najveci != i) {
		swap(niz[i], niz[najveci]);
		heapify(niz, n, najveci);
	}
}
void heap_sort(long niz[], int n) {
	for (int i = n / 2 - 1; i >= 0; i--)
		heapify(niz, n, i);
	for (int i = n - 1; i >= 0; i--) {
		swap(niz[0], niz[i]);
		heapify(niz, i, 0);}}
```

5. 5. Implementacija Quick sorta

```
long podijeli(long niz[], int p, int q) {
	long x = niz[p];
	int i = p, j;
	for (j = p + 1; j<q; j++) {
		if (niz[j] <= x) {
			i = i + 1;
			int temp = niz[i];
			niz[i] = niz[j];
			niz[j] = temp;
		}
	}
	int tempa = niz[i];
	niz[i] = niz[p];
	niz[p] = tempa;
	return i;
}
void quick_sort(long niz[], int p, int q) {
	int r;
	if (p<q) {
		r = podijeli(niz, p, q);
		quick_sort(niz, p, r);
		quick_sort(niz, r + 1, q);}}
```

5.6. Funkcija za generiranje slučajnih brojeva

```
long gen_brojeva(long niz[], int a) {
	srand(time(NULL));
	rand();
	int x;
	for (x = 0; x < a; x++) {
		niz[x] = rand() + 1;
	}
	return x;
}
```

5.7. Glavna funkcija programa (main)

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

const long max_br = 1000000;
long br = 1000;
long niz_brojeva[max_br];

int main() {
	double t1, t2;
	int n;
	cout << "Koji algoritam zelis koristiti?" <<'\n';
	cout << "1 - Bubble sort\n" << "2 - Insertion sort\n" << "3 - Heap sort\n" << "4 -
Shell sort\n"<<"5 - Quick sort\n";
	cin >> n;
	switch (n) {
	case 1:
		cout << "Odabrali ste algoritam Bubble sort!\n";
		for (br = 1000; br <= max_br;) {
			cout << "\nDuljina niza: " << br << '\n';
			gen_brojeva(niz_brojeva, br);
			t1 = clock();
			bubble_sort(niz_brojeva, br);
			t2 = clock();
			cout << "Vrijeme sortiranja: " << (t2 - t1) / CLK_TCK << " sek\n";
			switch (br) {
			case 1000:
				br = 10000;
				break;
			case 10000:
				br = 100000;
				break;
			case 100000:
				br = 500000;
				break;
			case 500000:
				br = 1000000;
				break;
			case 1000000:
				br = 1000001;
				break;
			}
		}
		break;
	case 2:
		cout << "Odabrali ste algoritam Insertion sort!\n";
		for (br = 1000; br <= max_br;) {
			cout << "\nDuljina niza: " << br << '\n';
			gen_brojeva(niz_brojeva, br);
			t1 = clock();
			insertion_sort(niz_brojeva, br);
			t2 = clock();
			cout << "Vrijeme sortiranja: " << (t2 - t1) / CLK_TCK << " sek\n";
```

```
                switch (br) {
                case 1000:
                        br = 10000;
                        break;
                case 10000:
                        br = 100000;
                        break;
                case 100000:
                        br = 500000;
                        break;
                case 500000:
                        br = 1000000;
                        break;
                case 1000000:
                        br = 1000001;
                        break;
                }
        }
                break;
case 3:
        cout << "Odabrali ste algoritam Heap sort!\n";
        for (br = 1000; br <= max_br;) {
                cout << "\nDuljina niza: " << br << '\n';
                gen_brojeva(niz_brojeva, br);
                t1 = clock();
                heap_sort(niz_brojeva, br);
                t2 = clock();
                cout << "Vrijeme sortiranja: " << (t2 - t1) / CLK_TCK << " sek\n";
                switch (br) {
                case 1000:
                        br = 10000;
                        break;
                case 10000:
                        br = 100000;
                        break;
                case 100000:
                        br = 500000;
                        break;
                case 500000:
                        br = 1000000;
                        break;
                case 1000000:
                        br = 1000001;
                        break;
                }
        }
        break;
case 4:
        cout << "Odabrali ste algoritam Shell sort!\n";
        for (br = 1000; br <= max_br;) {
                cout << "\nDuljina niza: " << br << '\n';
                gen_brojeva(niz_brojeva, br);
                t1 = clock();
                shell_sort(niz_brojeva, br);
                t2 = clock();
                cout << "Vrijeme sortiranja: " << (t2 - t1) / CLK_TCK << " sek\n";
                switch (br) {
                case 1000:
```

```
                        br = 10000;
                        break;
                case 10000:
                        br = 100000;
                        break;
                case 100000:
                        br = 500000;
                        break;
                case 500000:
                        br = 1000000;
                        break;
                case 1000000:
                        br = 1000001;
                        break;
                }
        }
                break;
    case 5:
        cout << "Odabrali ste algoritam Quick sort!\n";
        for (br = 1000; br <= max_br;) {
                int p = 0;
                cout << "\nDuljina niza: " << br << '\n';
                gen_brojeva(niz_brojeva, br);
                t1 = clock();
                quick_sort(niz_brojeva, p, br);
                t2 = clock();
                cout << "Vrijeme sortiranja: " << (t2 - t1) / CLK_TCK << " sek\n";
                switch (br) {
                case 1000:
                        br = 10000;
                        break;
                case 10000:
                        br = 100000;
                        break;
                case 100000:
                        br = 500000;
                        break;
                case 500000:
                        br = 1000000;
                        break;
                case 1000000:
                        br = 1000001;
                        break;
                }
        }
        break;
    default: cout << "Unesen krivi broj!\n";
    }
    system("pause");
    return 0;
}
```

5. Zaključak

Kako i u realnom životu tako i u programiranju algoritmi sortiranja su nam mnogo važni ponajviše zbog toga što nam uvelike olakšavaju pronalazak prave informacije te ujedno skraćuju dragocjeno vrijeme. Važan su dio upravljanja podataka koje možemo razlikovati na osnovu njihove složenosti. Svaki algoritam ima određene prednosti i nedostatke u odnosu na ostale, te se to uglavnom projicira na vremensku zahtjevnost algoritma. Promatrajući A priori rezultate možemo poredati razmatrane algoritme od najlošijeg do naboljeg, pa tako redom imamo: bubble sort, insertion sort, shell sort, heap i quick sort.

Osobno smatram da je kod algoritama kvadratne složenosti bubble i insertion sorta relativno jednostavno shvatiti princip rada te se mogu dosta lako i brzo implementirati pa bi tako oni bili moj izbor za upoznavanje nekog sa takvom vrstom programiranja ko se prvi put susreće s tim. Na taj se način dobija dobra podloga za daljnje proučavanje složenijih algoritama poput heap sorta koji radi na principu gomile ili quick sort koji koristi rekurziju te sadrže mnogo složenije kôdove za implementaciju. Uostalom, iz eksperimentalnih rezultata da se zaključiti kako je za nizove do 1000 elemenata bolje čak koristiti jednostavnije algoritme koji se lakše i brže implementiraju. Gledajući cjelokupnu sliku, mislim da je programski jezik C++ prilično pogodan za pisanje ovakvih algoritama jer se u nekoliko minuta uz malo osnovnog znanja o programskom jeziku, može implementirat algoritam koji sortira recimo milijun brojeva za manje od sekunde.
Prateći odjednom A priori i A posteriori rezultate vidimo kako se u velikoj mjeri podudaraju, ali ne u potpunosti. Tu dolazi do izražaja razlika u broju elemenata na ulazu te kako se koji algoritam prema tome ponaša.

6. Literatura

[1] Motik, B., Šribar, J. (1997). *Demistificirani C++* (Četvrto izdanje 2014). Element d. o. o.

[2] Sedgewick, R. (2002). *Algorithms in C++*.

[3] Oljica, M. (2014). *Vizualizacija osnovnih algoritama za sortiranje*. Split: Sveučilište u Splitu.

[4] Baumgartner, A., Poljak S. (2005). *Sortiranje podataka*. Osječki matematički list 5, 21-28. Osijek: Sveučilište u Osijeku.

[5] Deljac, S. (2008). *Osnove programiranja u C++*. Zagreb: Željeznička tehnička škola. Preuzeto, 01.08.2016. s http://web1.ss-amkaramaneo-vis.skole.hr/

[6] Goretić, A. *Algoritmi sortiranja*. Preuzeto, 01.08.2016. s http://www.slideshare.net/adosvdc/algoritmi-sortiranja-u-c

[7] Sareej, P. (2013). *Comparison of Sorting Algorithms (On the Basis of Average Case)*. Preuzeto, 03.08.2016. s http://www.ijarcsse.com

[8] Allain, A. (2011). *Sorting algorithm comparison*. Preuzeto, 03.08.2016. s http://www.cprogramming.com/

[9] Khairullah, Md. (2013). *Enhancing Worst Sorting Algorithms*. Preuzeto, 05.08. 2016. s http://www.sersc.org

[10] Shaffer, C.A. (2010). *A Practical Introduction to Data Structures and Algorithm Analysis.* Preuzeto, 05.08.2016. s https://people.cs.vt.edu

[11] Niemann, T. *Sorting and Searching Algorithms: A Cookbook*. Preuzeto, 06.08.2016. s https://www.cs.auckland.ac.nz/

[12] Wikipedia. *Sorting algorithm*. Preuzeto, 06.08.2014. s https://en.wikipedia.org