SVEUČILIŠTE U ZAGREBU
PRIRODOSLOVNO – MATEMATIČKI FAKULTET
MATEMATIČKI ODSJEK

Projektni izvještaj – Oblikovanje i analiza algoritama

Usporedba sortiranja: klasični sort, quick sort

Mentor: doc. dr. sc. Matej Mihelčić

Dominik Horvat

26. listopada 2023.

Sadržaj

Uvod 1
Klasičan sort 1
Način rada 1
Primjer programskog koda 1
Složenost algoritma 2
Quick sort 3
Način rada 3
Primjer programskog koda 4
Analiza vremenske složenosti 5
Prosječna složenost quick sort algoritma 5
Najgora složenost quick sort algoritma 7
Klasičan sort i quick sort – usporedba 8
Empirijska mjerenja 8
Sortiranje nasumičnih nizova 8
Sortiranje već sortiranog niza 10
Prosječna složenost algoritma quick sort 12
Zaključak 14
Literatura 16

Uvod

U ovom projektnom izvještaju bit će opisana dva algoritma koja se koriste za sortiranje nizova, preciznije, bit će opisan algoritam klasičnog sortiranja i brzog sortiranja (eng. *quick sort*) Detaljno će biti objašnjeni algoritmi, načini na koji oni rade i sortiraju. Za svaki sort bit će priložen adekvatan programski kod u programskom jeziku C. Bit će analizirane njihove vremenske složenosti uz adekvatne dokaze istih. Na kraju projektnog izvještaja fokus će biti na primjerima kroz koje će se vidjeti odnos između njihovih vremenskih složenosti. Ukratko, biti će prikazani grafovi koji prikazuju vrijeme sortiranja u odnosu na određene duljine nizova. Izvještaj će omogućiti uvid u to koji je algoritam bolji za sortiranje nizova te neke prednosti i mane između ova dva algoritma.

Klasičan sort

Ovaj sort nam govori: „*Za sve indekse i, j takve da je $i < j$ provjeri je li $a_i > a_j$ (tj. jesu li brojevi a_i i a_j u pogrešnom redoslijedu): ako da, onda ih zamijeni.*“ (Iz [1], stranica 8) Ukratko rečeno, na većem indeksu mora biti veći element. Neka je x neki niz i neka su pripadni indeksi i, j. Tada vrijedi:

$$\forall\, i, j \colon (i < j) \Rightarrow x_i \leq x_j.$$

Ovime se dobiva uzlazno sortirani niz, ali je jasno da promjenom znaka usporedbe postižemo da konačan niz nakon sortiranja bude silazno sortiran.

Način rada

Za dani niz kojeg trebamo (uzlazno) sortirati krećemo od prvog člana niza. Promatramo prvi član u odnosu na svakog idućeg. Ako je prvi član veći od nekog na većem indeksu napravimo zamjenu tih dvaju članova. Nastavljamo s drugim indeksom (indeks *j*) po nizu dok je prvi još uvijek fiksiran na prvi član sve dok ne dođemo do kraja niza s drugim indeksom. Kada dođemo do kraja niza prvi član će biti najmanji. Zatim se pomičemo s indeksom *i* na drugi član niza i pratimo analogan postupak, promatramo ga u odnosu na preostale članove koji slijede nakon njega sve do kraja samog niza. I tako sve do kraja zadanog niza, konačan rezultat biti će sortiran niz.

Primjer programskog koda

Kod za klasičan sort u programskom jeziku C glasi:

```
#include<stdio.h>
int main(void){
    int niz[1000];
    int n,i,j;

    //ucitamo velicinu niza
    scanf("%d", &n);
    for(i=0;i<n;i++)
        scanf("%d", &niz[i]);

    for(i=0;i<n-1;i++)
        for(j=i+1;j<n;j++)
```

```
    if(niz[i]>niz[j]){
        int temp=niz[i];
        niz[i]=niz[j];
        niz[j]=temp;
    }

    //ispis sortiranog niza
        for(i=0;i<n;i++)
            printf(" %d\n", niz[i]);

    return 0;
}
```

Složenost algoritma

Klasičan sort će sortirati niz, ali to radi loše. Prvobitno zbog njegove vremenske složenosti. Za niz proizvoljne duljine n, varijabla i (kao što je gore u primjeru koda) poprima $n-1$ vrijednost. Dakle, $i = 0, 1, \dots, n-2 < n-1$. Taj i-ti element uspoređuje se s preostalih $n-1-i$ elemenata. Na primjer, za $i = 0$ (početni element) uspoređujemo s j-tim, za $j = 1, 2, \dots, n-1$ elemenata. Sve nas to dovodi do jako puno usporedbi što utječe na samu vremensku složenost ovog algoritma. Kao što se nazire iz samog koda svaki put se u $if()$ uspoređuje i-ti s j-tim elementom.

i	$n-1-i$
$\mathbf{0}$	$\boldsymbol{n-1}$
$\mathbf{1}$	$\boldsymbol{n-2}$
$\mathbf{2}$	$\boldsymbol{n-3}$
.	.
.	.
.	.
$\boldsymbol{n-2}$	$\mathbf{1}$

Tablica 1

Lijevi stupac *Tablice 1* predstavlja indeks onog elementa kojeg uspoređujemo sa svim elementima nakon njega, a desni stupac predstavlja koliko je tih usporedbi napravljeno. Za složenost algoritma gledamo desni stupac i zbrajamo od kraja prema vrhu. Dakle, to je ekvivalentno zbroju prvih $n-1$ prirodnih brojeva. Općenito, za zbroj prvih n prirodnih brojeva formula glasi $1 + 2 + \cdots + n = \frac{n \cdot (1+n)}{2}$. Primjenjujući ovu formulu na prvih $n-1$ prirodnih brojeva dobiva se sljedeće:

$$1 + 2 + \cdots + n - 1 = \frac{(n-1) \cdot (n-1+1)}{2} = \frac{(n-1) \cdot n}{2} = \frac{n^2 - n}{2} = \frac{1}{2} \cdot n^2 - \frac{1}{2} \cdot n$$

S obzirom da se u prethodnom računu pojavljuje $\frac{1}{2} \cdot n^2$ govorimo kako je složenost upravo jednaka $O(n^2)$, takozvana kvadratna složenost. Jasno je kako će za klasičan sort uvijek složenost biti kvadratna, neovisno o tome u kakvom su početnom redoslijedu elementi niza. Svejedno će se kretati od prvog elementa niza i uspoređivati sa svakim nakon, i tako sve do kraja samog niza.

Quick sort

Riječ je o rekurzivnom algoritmu za sortiranje. Karakteristike ovog algoritma su: dijeli zadano polje na dva manja polja, rekurzivnim pozivima sortiraju se dva manja polja te na kraju spaja dva manja sortirana polja u jedno.

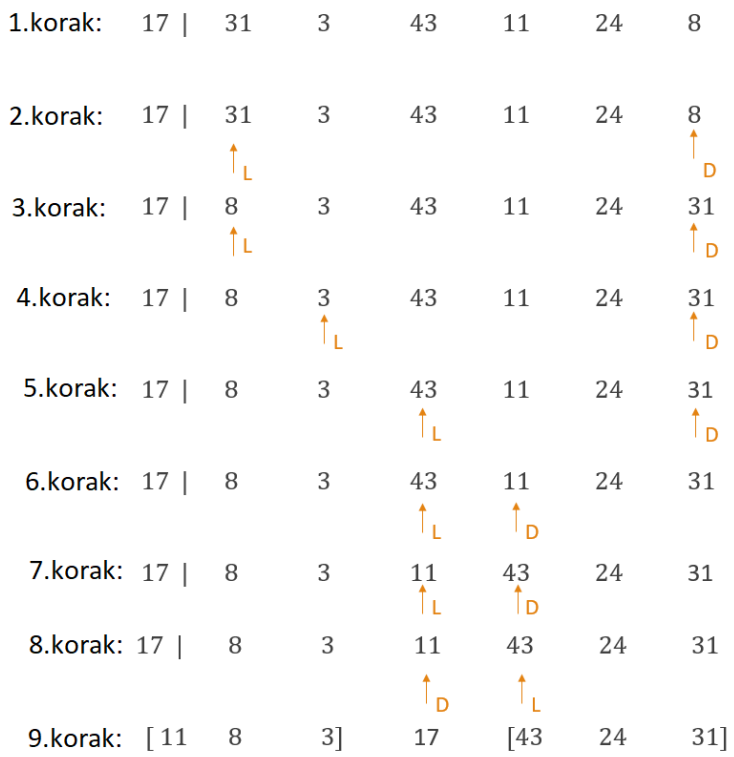
Način rada

Prvobitno odaberemo jedan element polja koji nazivamo stožer. Najčešće primjenjujemo dvije mogućnosti odabira stožera:

- Za stožer uzimamo početni element polja
- Za stožer uzimamo medijan izabran između tri elementa koja se nalaze na početku, kraju odnosno sredini polja.

Zatim se svi ostali elementi razvrstavaju ispred odnosno iza stožera. To gledamo tako je li element polja $\leq$ od stožera, odnosno $\geq$ od stožera.

Nakon provedenog postupka, to jest razvrstavanja elemenata na gore opisan način, stožer se nalazi na svom završnom mjestu u traženom sortiranom poretku. Kako bi se sortiralo dano polje nakon ovog koraka, dovoljno je zasebno sortirati novonastala pod-polja ispred, odnosno iza stožer. To može biti na dva načina: rekurzivnim pozivima istog algoritma ili na trivijalan način za pod-polja duljine 0 ili 1. Način samog rada uz dodatno detaljno objašnjenje biti će prikazano pomoću *Slike 1* i popratnog teksta.



Slika 1 Primjer sortiranja quick sorta

U ovom primjeru ćemo za stožer izabrati početni element (1.korak). Sa strelicama, to jest $\uparrow_L$ i $\uparrow_D$ označavamo koje elemente gledamo u odnosu na naš izabran stožer. Sa simbolom $\uparrow_L$ kojeg ćemo u nastavku zvati kazaljka označavamo elemente polja koje gledamo u ovisnosti od stožera, a trebaju biti $\leq$ naspram stožera. Slično s kazaljkom $\uparrow_D$ gdje elementi polja na tom

mjestu moraju biti ≥ od stožera. Vidimo kako je u 2. koraku na poziciji $\uparrow_L$ element veći od stožera pa gledamo element na koji pokazuje $\uparrow_D$. Kako kazaljka $\uparrow_D$ pokazuje na manji element od stožer radimo zamjenu kao što je vidljivo u 3. koraku. Četvrti i peti korak nam ukazuju kako se kazaljka $\uparrow_L$ pomiče u desno tražeći sljedeći element polja koji će biti veći ili jednak od stožera. U 6. koraku vidimo kako se kazaljka $\uparrow_D$ pomaknula dva mjesta u lijevo na element 11 koji je manji od stožera. U 7. koraku dogodila se zamjena elemenata na koje su pokazivale kazaljke u 6. koraku. Ključan je 8. korak gdje je kazaljka $\uparrow_L$ traži dalje veći element od stožer i pomaknula se za jedno polje u desno, a kazaljka $\uparrow_D$ se zatim pomakla za jedno polje u lijevo. Samim tim iz *Slike 1* uočljivo je kako su se kazaljke prekrižile u 9. koraku. Ovdje naš algoritam staje s pomicanjem kazaljki i mijenja element koji je označen kao stožer s elementom na koji pokazuje kazaljka $\uparrow_D$. U 9. koraku nastaju dva nova pod-polja koja treba sortirati rekurzivnim pozivom ovog algoritma. Postupak sortiranja pod-polja je analogan upravo opisanom postupku.

Primjer programskog koda

Quick sort se može implementirati na razne načine ovisno o izboru stožera. Navedena je implementacija quick sorta u programskom jeziku C gdje se u svakom rekurzivnom pozivu kao stožer bira početni element pod-polja koje treba sortirati. Ukratko, poziv funkcije `QuickSort()` sortira dio polja `a[]` čiji elementi imaju indekse od `l` do `r`.

```
void QuickSort(int a[], int l, int r){
    int i,j;

    if(l>=r) return;

    //razvrstavaju se elementi s obzirom na stozer
    i=l+1;
    j=r;
    while((i<=j) && (i<=r) && (j>l)){
        while((a[i]<=a[l]) && (i<=r)) i++;
        while((a[j]>=a[l]) && (j>l)) j--;
        if(i<j)
            swap(&a[i], &a[j]);
    }
    if(i>r){
        //stozer je najveci u polju
        swap(&a[r],&a[l]);
        QuickSort(a,l,r-1);
    }
    else if(j<=l){
        //stozer je najmanji u polju
        QuickSort(a,l+1,r);
    }
    else{
        //stozer se nalazi negdje u sredini
        swap(&a[j],&a[l]);
        QuickSort(a,l,j-1);
        QuickSort(a,j+1,r);
    }
```

```
}
```

Napomena 1.: Za zamjenu elemenata korištena je funkcija `swap`. Implementacija ove funkcije u programskom jeziku C izgleda ovako:

```
void swap(int *a, int *b){
int temp;
    temp = *a;
    *a = *b;
    *b = temp;
return;
}
```

Definiramo ju prije funkcije `QuickSort()`.

Napomena 2.: Poziv koji bi sortirao cijelo polje `a[]` duljine n izgleda ovako: `QuickSort(a,0,n-1)`.

Analiza vremenske složenosti

Quick sort algoritam u najgorem slučaju ima složenost jednaku $O(n^2)$. Ovaj slučaj nastupa kada je polje već sortirano i stožer je početni element polja. Usprkos tome postoji i matematički dokaz da je prosječno vrijeme izvršavanja quick sorta jednako $O(n \cdot \log n)$. Naslućujemo kako quick sort postiže dvije složenosti: $O(n^2)$ i $O(n \cdot \log n)$. Praktična iskustva primjene ovog algoritma u stvarnosti pokazuju njegovu izuzetnu brzinu. Na ponašanje samog algoritma u velikoj mjeri utječe odabir stožera. Ističu se dvije mogućnosti za odabir koje su u ovom izvještaju prethodno navedene.

Prosječna složenost quick sort algoritma

U ovom dijelu projektnog izvještaja biti će iznesen dokaz za prosječnu složenost quick sort algoritma koja je jednaka $O(n \cdot \log n)$.

Pretpostavimo da su sve permutacije polja kojeg treba quick sort algoritam sortirati jednako vjerojatne. Uvest ćemo oznaku $T_{avg}(n)$ kojom označavamo očekivano vrijeme za sortiranje polja duljine n elemenata. Sam dokaz će se zasnivati na sljedećoj ocjeni: pokazat ćemo da postoji neka konstanta k takva da za $\forall n \geq 2$ vrijedi sljedeće:

$$T_{avg}(n) \leq k \cdot n \cdot \ln n\,.$$

Kada pozovemo quick sort algoritam da sortira dano polje duljine n, algoritam najprije bira stožer (kao što je navedeno u samom načinu rada algoritma). Zatim razvrstava ostale elemente polja ispred, odnosno iza stožera. Nakon ovog procesa algoritam stavlja stožer na j-to mjesto u polju. Nakon toga rekurzivnim pozivima sortira novo dobivena pod-polja, to jest pod-polja ispred, odnosno iza početno odabranog stožera. Duljina pod-polja ispred stožera je upravo j, a duljina potpolja iza stožera je $n - j - 1$. Izbor samog stožera te razvrstavanje elemenata zahtjeva linearno vrijeme oblika $c \cdot n$, gdje je c neka konstanta. Ako je $T_{avg}(n)$ očekivano vrijeme za sortiranje polja duljine n, to jest polja od n elemenata tada očekivano vrijeme za dva rekurzivna poziva iznosi upravo $T_{avg}(j) + T_{avg}(n - j - 1)$. Budući da zbog naše početne pretpostavke j može poprimiti bilo koju vrijednost između 0 i $n - 1$, i to s jednakom vjerojatnošću, $T_{avg}(n)$ se dobiva „usrednjavanjem“ po svim mogućim j-ovima. To dovodi do toga da za $n \geq 2$ vrijedi sljedeće:

$$T_{avg}(n) \leq c \cdot n + \frac{1}{n} \cdot \sum_{j=0}^{n-1} \Big(T_{avg}(j) + T_{avg}(n-j-1) \Big).$$

Suma $\sum_{j=0}^{n-1} \Big(T_{avg}(j) + T_{avg}(n-j-1) \Big)$ se može rastaviti na dvije sume koje će biti jednake (zbog toga što je redoslijed sumiranja obrnut). Radi toga ova ista formula može se zapisati na jednostavniji način:

$$T_{avg}(n) \leq c \cdot n + \frac{2}{n} \cdot \sum_{j=0}^{n-1} T_{avg}(j)\,.$$

Zatim odabiremo neku konstantu b tako da vrijedi $T_{avg}(0) \leq b$ i $T_{avg}(1) \leq b$ te uzimamo $k = 2 \cdot (b + c)$. Uz ovako odabrane konstante možemo dokazati da za $n \geq 2$ zaista vrijedi prije spomenuta nejednakost $T_{avg}(n) \leq k \cdot n \cdot \ln n$. Dokaz provodimo matematičkom indukcijom po $n \in \mathbb{N}\backslash\{1\}$:

- Baza: Za $n = 2$ zbog nejednakosti $T_{avg}(n) \leq c \cdot n + \frac{2}{n} \cdot \sum_{j=0}^{n-1} T_{avg}(j)$ te zbog činjenice da je $\ln 2 \approx 0.69314$ imamo sljedeće: $T_{avg}(2) \leq 2 \cdot c + 2 \cdot b = k \leq k \cdot 2 \ln 2$
- Pretpostavka: Za bilo koji $m, 2 \leq m < n$, vrijedi: $T_{avg}(m) \leq k \cdot m \cdot \ln m$
- Korak: Kombinirajući $T_{avg}(n) \leq c \cdot n + \frac{2}{n} \cdot \sum_{j=0}^{n-1} T_{avg}(j)$ i pretpostavku matematičke indukcije te zbog izbora b dobivamo:

$$T_{avg}(n) \leq c \cdot n + \frac{4 \cdot b}{n} + \frac{2}{n} \cdot \sum_{j=2}^{n-1} T_{avg}(j)$$

$$\leq c \cdot n + \frac{4 \cdot b}{n} + \frac{2 \cdot k}{n} \cdot \sum_{j=2}^{n-1} j \cdot \ln j.$$

Zadnju sumu možemo shvatiti kao donju aproksimaciju površine ispod krivulje funkcije $x \cdot \ln x$ na intervalu od 2 do n. Točnu površinu dobivamo računanjem integrala pa vrijedi sljedeće:

$$T_{avg}(n) \leq c \cdot n + \frac{4 \cdot b}{n} + \frac{2 \cdot k}{n} \int_2^n x \cdot \ln x \; dx = (*)$$

Nakon sređivanja integrala $\int_2^n x \cdot \ln x \; dx$ dobivamo sljedeće:

$$(*) = c \cdot n + \frac{4 \cdot b}{n} + \frac{2 \cdot k}{n} \left[\frac{x^2}{2} \cdot \ln x - \frac{x^2}{4} \right]_2^n$$

$$= c \cdot n + \frac{4 \cdot b}{n} + \frac{2 \cdot k}{n} \left[\frac{n^2}{2} \cdot \ln n - \frac{n^2}{4} - \frac{2^2}{2} \cdot \ln 2 + \frac{2^2}{4} \right]$$

$$\textit{(...Vrijedi da je } -\frac{2^2}{2} \cdot ln\, 2 + \frac{2^2}{4} < 0 \textit{...)}$$

$$\leq c \cdot n + \frac{4 \cdot b}{n} + k \cdot n \cdot \ln n - \frac{k \cdot n}{2}$$

$$\textit{(...Koristimo da je } k = 2 \cdot (b + c) \textit{...)}$$

$$= c \cdot n + \frac{4 \cdot b}{n} + k \cdot n \cdot \ln n - \frac{2 \cdot (b + c) \cdot n}{2}$$

$$= c \cdot n + \frac{4 \cdot b}{n} + k \cdot n \cdot \ln n - b \cdot n - c \cdot n$$

$$\textit{(...Vrijedi da je } \frac{4 \cdot b}{n} \leq 2 \cdot b \textit{ i } b \cdot n \geq 2 \cdot b \textit{...)}$$

$$\leq 2 \cdot b + k \cdot n \cdot \ln n - 2 \cdot b = k \cdot n \cdot \ln n.$$

$$\Rightarrow T_{avg}(n) \leq k \cdot n \cdot \ln n$$

Time smo dokazali korak indukcije. (Dokaz je napravljen pomoću [3])

Zaključak: Prosječna složenost algoritma quick sort iznosi $O(n \cdot \log n)$.

Najgora složenost quick sort algoritma

U ovom dijelu projektnog izvještaja biti će iznesen dokaz za najgoru složenost quick sort algoritma koja je jednaka $O(n^2)$. Dokaz izgleda ovako:

S $T(n)$ označimo najgore vrijeme za postupak quick sorta za ulaznu veličinu n. S obzirom da smo stavili stožer na pravo mjesto i podijelili $n-1$ elemenata na dva pod-polja, dobivamo sljedeću rekurziju

$$T(n) = \max\{T(q) + T(n-1-q) : 0 \leq q \leq n-1\} + \theta(n).$$

$\theta(n)$ ukazuje na to koliko imamo elemenata za razvrstati i ubacivanje stožera na pravo mjesto. Izraz unutar vitičastih zagrada govori nam vremena pod-polja. U jednom pod-polju se nalazi q elemenata, dok u drugom je $n-1-q$, gdje je q cijeli broj između 0 i $n-1$. Dalje pretpostavljamo kako je $T(n) \leq c \cdot n^2$, za neku konstantu $c > 0$. Supstituirajući ovo u

$$T(n) = \max\{T(q) + T(n-1-q) : 0 \leq q \leq n-1\} + \theta(n),$$

dobivamo sljedeće:

$$T(n) \leq \max\{c \cdot q^2 + c \cdot (n-1-q)^2 : 0 \leq q \leq n-1\} + \theta(n)$$

$$= c \cdot \max\{q^2 + (n-1-q)^2 : 0 \leq q \leq n-1\} + \theta(n).$$

Promotrimo sada izraz unutar vitičastih zagrada. Za $q = 0,1,\dots,n-1$, imamo sljedeće:

$$q^2 + (n-1-q)^2 = q^2 + (n-1)^2 - 2 \cdot q \cdot (n-1) + q^2$$

$$= (n-1)^2 + 2 \cdot q \cdot \big(q - (n-1)\big)$$

$$\leq (n-1)^2$$

Zadnja nejednakost nastaje jer znamo kako je $0 \leq q \leq n-1$, jasno je za uočiti kako iz $q \leq n-1$ prebacivanjem $n-1$ na lijevu stranu dobivamo $q-n+1 \leq 0$. Samim tim zaključujemo kako je izraz $(n-1)^2 + 2 \cdot q \cdot \big(q-(n-1)\big)$ neovisno o $2 \cdot q$ manji ili jednak od $(n-1)^2$. Drugim riječima za bilo koji q, maksimumu je $\leq (n-1)^2$. Nastavljajući dalje s našom analizom $T(n)$ uz prethodno dokazano dobivamo sljedeće:

$$T(n) \leq c \cdot (n-1)^2 + \theta(n)$$

$$\leq c \cdot n^2 - c \cdot (2 \cdot n - 1) + \theta(n)$$

$$\leq c \cdot n^2,$$

za dovoljno veliku konstantu c, $\theta(n)$ će postati manji od $c \cdot (2 \cdot n - 1)$.

$\Rightarrow T(n) = O(n^2)$.

Klasičan sort i quick sort – usporedba

Najbolju usporedbu ova dva sorta možemo dati kroz njihove vremenske složenosti. Kao što je prije izneseno u projektnom izvještaju, klasičan sort neovisno o rasporedu n elemenata imat će složenost $O(n^2)$. S druge strane, kod quick sorta prosječna i najgora složenost nisu iste. Prosječna složenost je jednaka $O(n \cdot \log n)$, a najgora složenost je jednaka $O(n^2)$. Na složenost utječu specifične situacije u početnom rasporedu elemenata, ali i izboru stožera. Jasno je za naslutiti kako je quick sort bolji algoritam sortiranja naspram klasičnog sorta koji ima uvijek istu složenost $O(n^2)$. Općenito, bitno je za istaknuti kako je quick sort u praksi jedan od najbržih algoritama za sortiranje. Klasičan sort ima vrlo jednostavnu implementaciju kao što se i vidi po samom priloženom programskom kodu. S druge strane, quick sort ima složeniju implementaciju i nije lak za razumijevanje kao klasičan sort. Što se tiče memorije između ova dva algoritma dolazi do velike razlike. Klasičan sort ne zahtijeva dodatnu memoriju osim one koja se koristi za pohranu samih podataka. Problem nastaje kod quick sorta. Ovaj algoritam može zahtijevati dodatnu memoriju za svoje rekurzivne pozive i Stog za pohranu podataka. Bitno je za istaknuti primjer kada se pokušava sortirati već sortirani niz algoritmom quick sort. Naime, za sortirane nizove duljine n veće od 30 000 kompajler javlja grešku (u slučaju mojih testiranja). Pretpostavka je da dolazi do preopterećenja memorije i stoga uslijed značajnog broja rekurzivnih poziva. U empirijskim mjerenjima gdje se provodilo sortiranje već sortiranog niza može se uočiti kako se testiralo do nizova duljine $n = 30000$.

Empirijska mjerenja

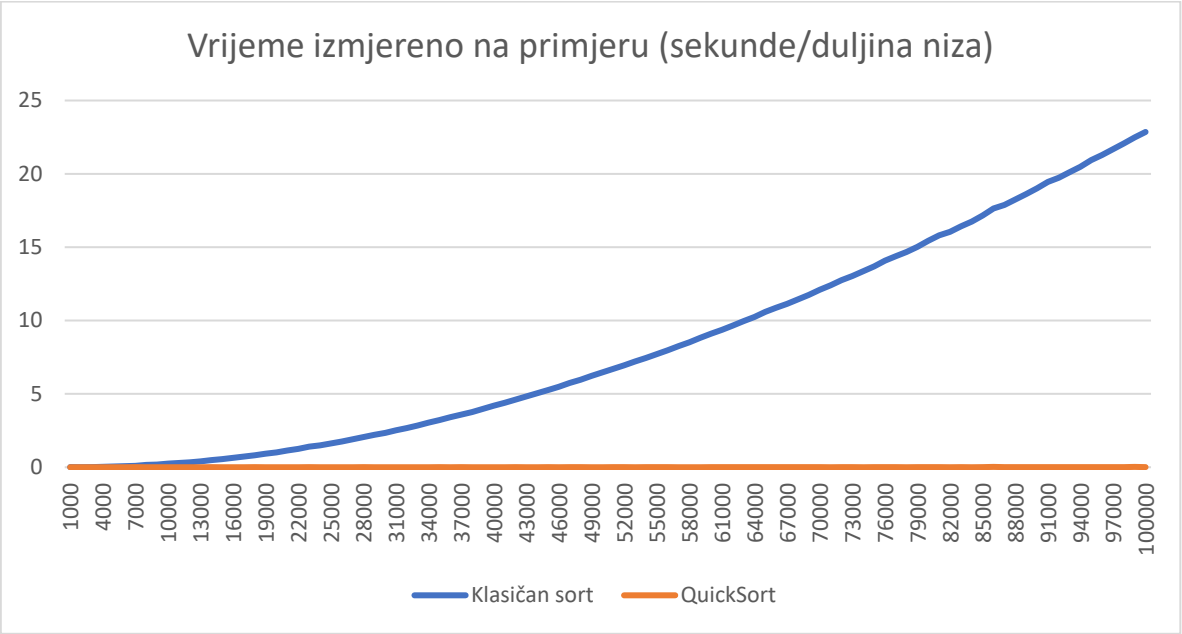
Sortiranje nasumičnih nizova

U ovom dijelu projektnog izvještaja biti će prezentirana empirijska mjerenja vezana uz klasičan sort i quick sort. Računalo na kojem su testiranja izvršena ima sljedeće specifikacije:

- Procesor – Intel® Core™ i3-1005G1 CPU @ 1.20GHz 1.19GHz
- Instalirana memorija (RAM) – 8,00 GB (7,79 GB iskoristivo)
- Vrsta sustava: 64-bitni operacijski sustav, procesor x64

Kodovi su napisani u programu Code::Blocks u programskom jeziku C. Za prikaz grafova korišten je Microsoft Excel. U prvom mjerenju generirani su slučajni prirodni brojevi

uključujući i nulu. Pozivima quick sorta i klasičnog sorta sortirao se niz duljine $n \in \{1000, 2000, \ldots, 100000\}$.



Graf 1 Usporedba klasičnog sorta i quick sorta

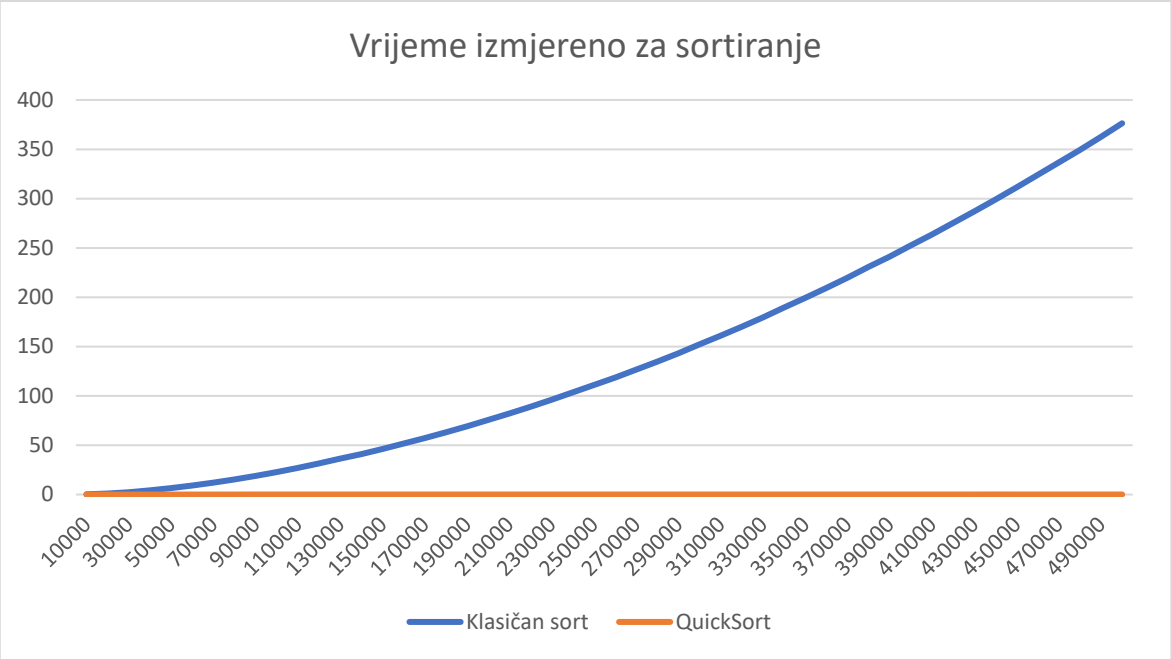
Graf 1 prikazuje testiranje jednog prolaza za nizove duljine n. Horizontalnom skalom prikazana veličina nizova duljine n koja se sortira, a vertikalnom skalom vrijeme u sekundama potrebno za sortiranje određene duljine niza. Jasno je za uočiti kako je već uz prije navedeno quick sort veoma brz u praksi što se vidi i na priloženom grafu. Klasičnom sortu će već za nizove duljine n veće od 90000 trebati više od 19 sekundi, dok quick sortu treba oko 16 milisekundi što je vrlo impresivno. Najbolje možemo to uočiti iz sljedeće tablice za isti ovaj primjer:

Duljina niza (n)	*Vrijeme za klasičan sort (s)*	*Vrijeme za quick sort (s)*
1000	0,006	0
2000	0	0
5000	0,053	0
10000	0,185	0
25000	1,495	0
50000	6,459	0
100000	22,857	0,015

Tablica 2 prikaz vremena u sekundama za sortiranje

Iz tablice je vidljivo kako quick sort tek pri većim duljinama niza ima vrijeme sortiranja jednako 0,015 sekundi, to jest 15 milisekundi (*ms*) što čini veliku razliku u vremenu sortiranja ova dva algoritma.

Ponovno za nasumično generirane prirodne brojeve uključujući i nulu napravljeno je testiranje (jedan prolaz) vremena potrebnog za sortiranje između quick sorta i klasičnog sorta. Ali ovaj put za duljine nizova $n \in \{10000, 20000, \ldots, 500000\}$. Dobiveni rezultati potvrđuju ponovo brzinu i efikasnost primjene quick sorta.



Graf 2 prikaz vremena u sekundama za sortiranje

Ponovo isti zaključak, vrijeme sortiranja za klasičan sort za dovoljno velike n-ove raste kvadratno (što se može i uočiti iz prikazanog grafa), dok quick sort i dalje ostaje na vrlo malim izmjerenim vremenima, i to čak u milisekundama.

Duljina niza (n)	*Vrijeme za klasičan sort (s)*	*Vrijeme za quick sort (s)*
10000	0,247	0,015
20000	1,087	0,000
50000	6,444	0,000
100000	22,864	0,023
250000	110,940	0,038
400000	252,400	0,062
500000	376,341	0,069

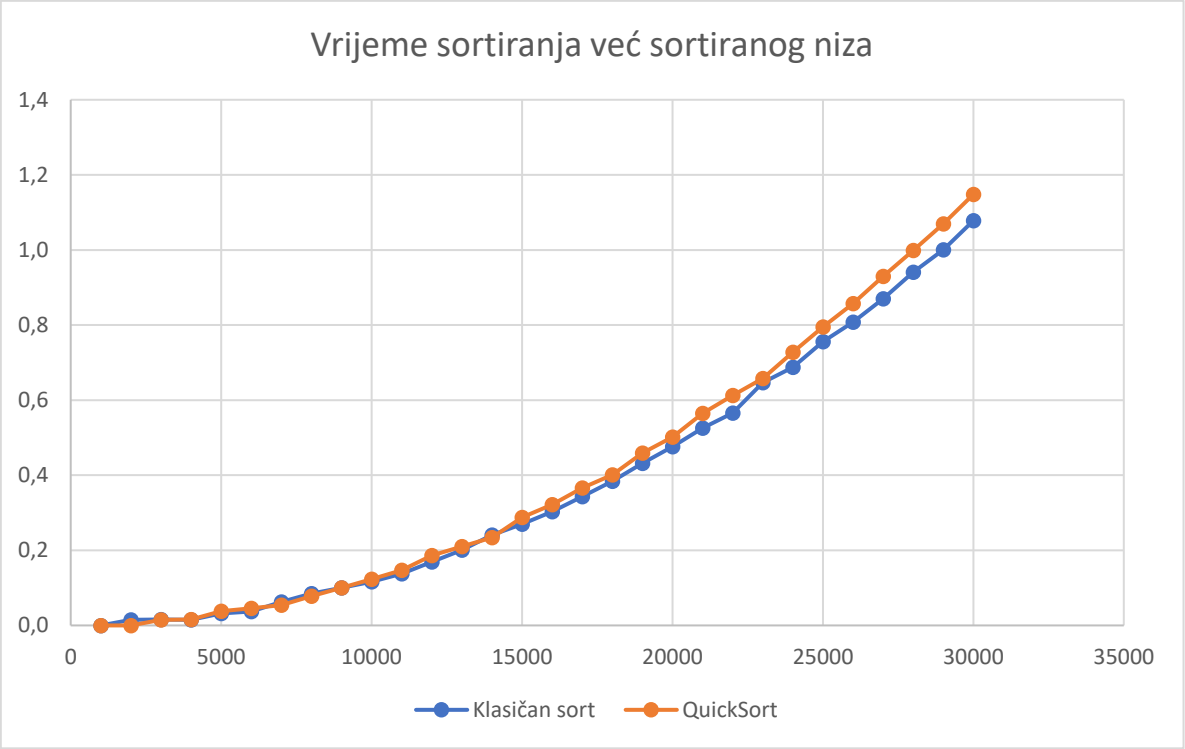
Tablica 3 prikaz vremena u sekundama za sortiranje

Vremenska razlika između sortiranja niza duljine $n = 10000$ i $n = 500000$ je velika. Za niz duljine 10000 razlika iznosi 0,247 sekundi, to jest 247 milisekundi, a za niz duljine 500000 iznosi 376,341 sekundi, to jest 376341 milisekundi. Naspram quick sorta kojemu je trebalo 15 milisekundi za sortiranje nasumičnog niza od 10000 elemenata, a za niz od 500000 elemenata 62 milisekunde. Ovo je samo još jedan od primjera koliko je quick sort brzi algoritam sortiranja. Važno je za istaknuti kako bi se primjenom uzastopnih sortiranja nekoliko datoteka sa generiranim nasumičnim cijelim (ne negativni) brojevima postigla bolja vizualizacija prosječne složenosti quick sorta koja iznosi $O(n \cdot \log n)$ te klasičnog sa složenosti $O(n^2)$. U podnaslovu „Prosječna složenost algoritma quick sort“ biti će prikazana testiranje za quick sort upravo takvog mjerenja gdje će biti grafički jasno uočljivo da je to stvarno jednako složenosti $O(n \cdot \log n)$.

Sortiranje već sortiranog niza

U podnaslovu iznad mogli smo vidjeti vrijeme potrebno za sortiranje elemenata nasumičnih nizova. Sada će biti iznijeta vremena sortiranja niza koji je već sortiran (elementi niza su cijeli ne negativni brojevi od 0 do krajnje duljine niza kojeg želimo sortirati, to jest do $n - 1$). U

ovom slučaju jasno će biti za naslutiti da quick sort algoritam neće biti više tako efikasan i brz. Mjerenja su se kretala za niz duljine $n \in \{1000, 2000, \ldots, 30000\}$. Sljedeći grafovi će prikazivati jedan takav prolaz ovih algoritama za sortiranje za određene nizove duljine n. Analogno bi se dobilo i za više testiranja.



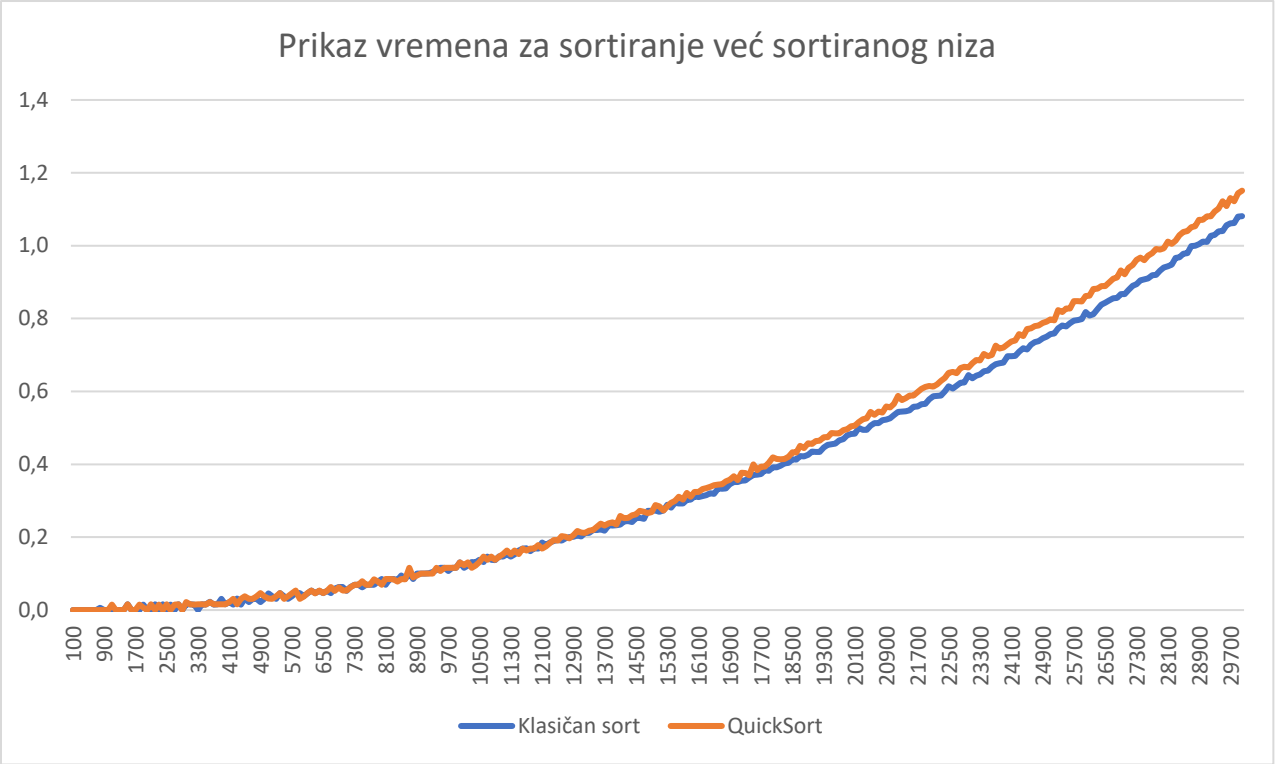
Graf 3 mjerenje sortiranja već sortiranog niza

Ponovo vertikalna skala grafa pokazuje vrijeme sortiranje u sekundama, a horizontalna duljinu niza n koji se treba sortirati. Iz *Grafa 3* vidimo kako za sortirani niz quick sort ima slično vrijeme sortiranja određene duljine niza n kao i klasičan sort. Sljedeća tablica to najbolje prikazuje:

Duljina niza(n)	*Vrijeme za klasičan sort (s)*	*Vrijeme za quick sort (s)*
1000	0,000	0,000
2000	0,015	0,000
5000	0,032	0,038
10000	0,116	0,123
15000	0,270	0,288
20000	0,476	0,459
25000	0,756	0,795
30000	1,078	1,148

Tablica 4

Iz prethodnog grafa i tablice vidimo kako je quick sort u sortiranju niza što se tiče vremena malo lošiji od klasičnog sorta, ako je niz sortiran. Pogledajmo još jedan primjer koji će istaknuti upravo odnos vremena između quick sorta i klasičnog sorta za sortiranje već sortiranog niza. Ovaj put se testiralo na primjeru gdje je bila duljina nizova $n \in \{100, 200, 300, \ldots, 30000\}$.



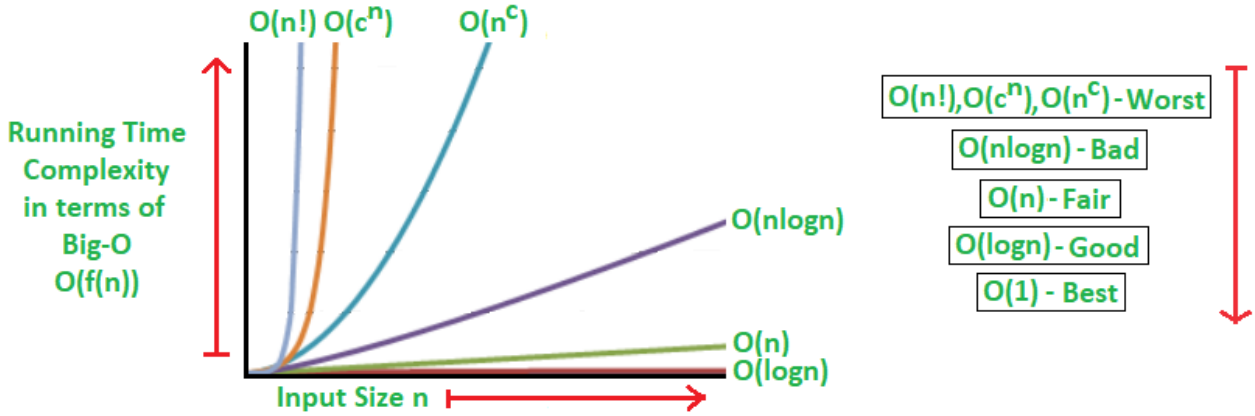
Graf 4

Napomena 3.: Provedena su i testiranja za n-ove poput 10, ali u tim sortiranjima klasičan sort i quick sort za već sortirani niz imaju vrijeme sortiranja 0 milisekundi. U većini slučajeva vrijeme koje je može iščitati u milisekundama se javlja za n-ove veće od 1000.

Napomena 4.: Za quick sort algoritam u mjerenjima primjenjivao se uvijek onaj koji će sortirati cijeli niz od početka do kraja, a za stožer se uzimao uvijek početni element tog niza.

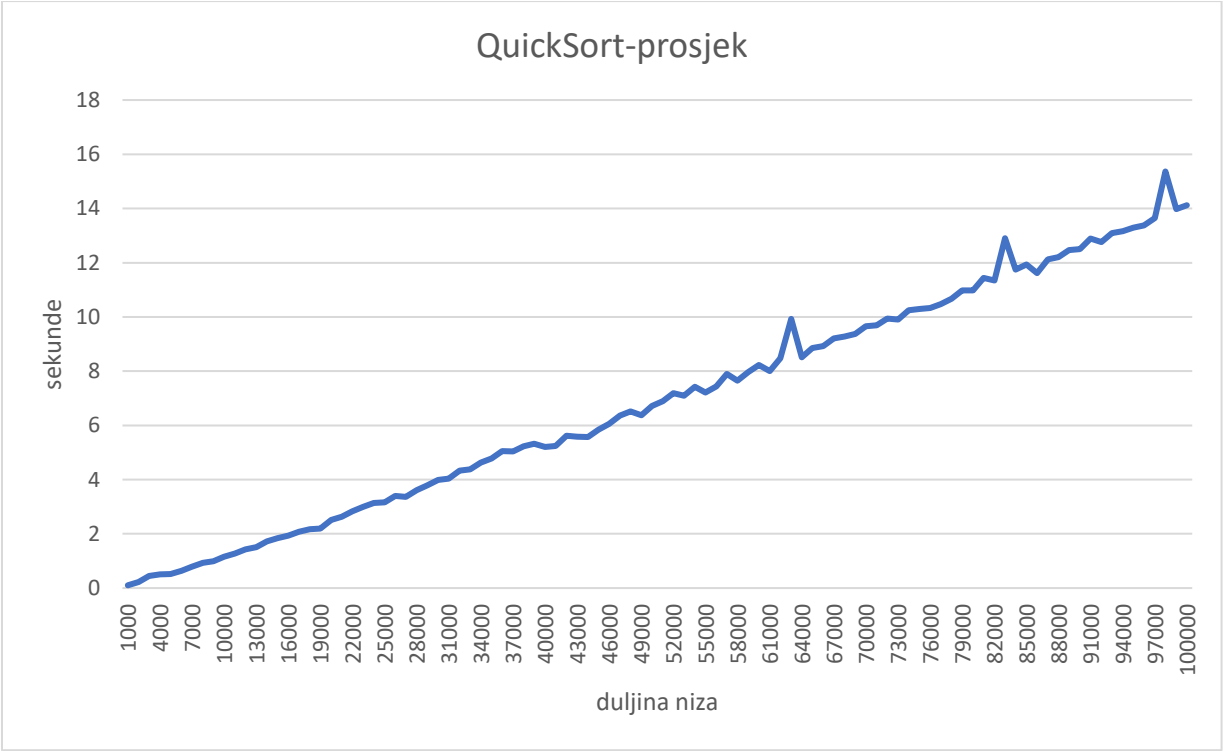
Prosječna složenost algoritma quick sort

Pored iznad iznesenih mjerenja, grafova i tablica, ovdje će biti još predočena prosječna vremenska složenost quick sort algoritma. Prema već navedenom prosječna složenost jednaka je $O(n \cdot \log n)$. Sljedeći grafovi i mjerenja ostvarena su na način da je generirano 1000 različitih „ *.txt* “ datoteka s 1000000 cijelih ne negativnih brojeva. Zatim se iz svake te datoteke uzimao niz početnih elemenata duljine n i sortirao quick sortom. Mjereno je vrijeme potrebno za sortiranje svakog niza duljine n i napravljena suma svih tih vremena. Na kraju je ta suma podijeljena s 1000 za određeni n. Za vizualizaciju općenito složenost $O(n \cdot \log n)$ grafički izgleda ovako (ljubičasta linija):



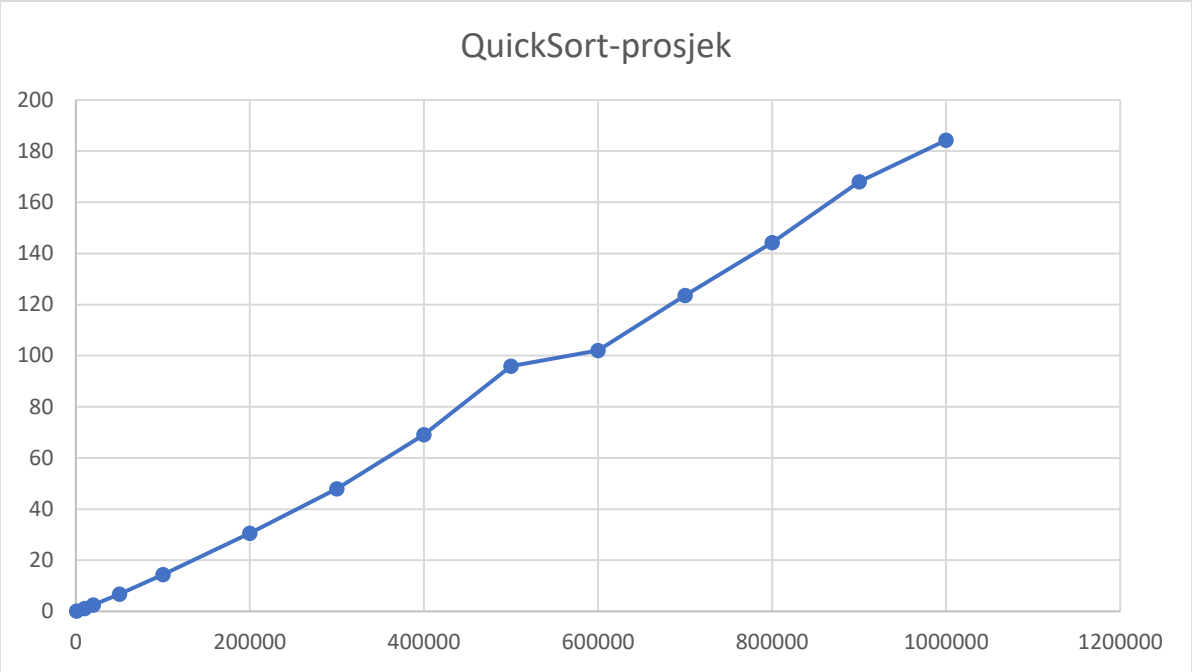
Slika 2 prikaz složenosti

Prvo testiranje se odnosilo na nizove duljine $n \in \{1000, 2000, \ldots, 100000\}$. Prikaz tog grafa je sljedeći:



Graf 5

Može se naslutiti kako kretanje linije iz grafa iznad nalikuje na onu iz *Slike 2* koja reprezentira $O(n \cdot \log n)$. Također, pored ovog grafa napravljen je graf gdje su duljine niza n bile sljedeće $\{1000, 10\,000, 20\,000, 50\,000, 100\,000, 200\,000, \ldots, 1\,000\,000\}$. Sve vrijednosti prosjeka mjerenja su dane u tablici ispod grafa.



Graf 6

Tablica vezana uz *Graf 6*:

Duljina niza (n)	*Prosjek u sekundama za 1000 pokretanja*
1000	0,102
10000	1,16
20000	2,516
50000	6,712
100000	14,39
200000	30,604
300000	47,98
400000	69,12
500000	95,966
600000	102,068
700000	123,545
800000	144,267
900000	167,982
1000000	184,262

Tablica 5

Zaključak

Kroz projektni izvještaj moglo se utvrditi ono što je navedeno više puta, quick sort je brži i učinkovitiji algoritam naspram klasičnog sorta. Postoji nekolicina slučajeva kada će i quick sort imati složenost $O(n^2)$, što je upravo i složenost klasičnog sorta.. Najbolje se to vidi kroz podnaslov „Sortiranje već sortiranog niza“ gdje se vrijeme mjerenja za quick sort i klasičan sort skoro pa podudaraju. Dakle, za opću primjenu sortiranja bilo bi poželjno koristiti quick sort algoritam. Iako je teži za implementaciju u programskom jeziku C (ali i u drugim programskim jezicima) naspram klasičnog sorta koji je lakše za implementirati. Vidimo kako empirijska analiza i svi priloženi grafovi pokazuju tu „cijenu“ isplativosti tog algoritma. Treba uvelike

istaknuti kako je klasičan sort neefikasan za sortiranje nizova velike proizvoljne duljine n. Najbolje se to može iščitati iz *Tablice 2* gdje za $n = 500000$ klasičnom sortu treba više od šest minuta, dok quick sortu treba samo 69 milisekundi. I kroz sam podnaslov „Prosječna složenost quick sort algoritma“ vidimo upravo tu efikasnost da neće se povećavati složenost algoritma, već će upravo biti jednaka krivulji sa Slike 2 koje prezentira složenost $O(n \cdot \log n)$. Intuitivno se nameće da bez prethodno testiranja istog načina za klasičan sort dobivamo složenost $O(n^2)$ što nas opet dovodi do zaključka kako je quick sort brži algoritam.

Literatura

[1] Šego, V. Programiranje 2 (vježbe) https://web.math.pmf.unizg.hr/nastava/prog2/materijali/prog2-vjezbe.pdf (datum pristupa: 4.10.2023.)

[2] Šego, V. Programiranje 1 (vježbe) https://web.math.pmf.unizg.hr/nastava/prog1/materijali/prog1-vjezbe.pdf (datum pristupa: 4.10.2023.)

[3] Manger, R. Strukture podataka i algoritama, Element, Zagreb, 2014.

[4] Cormen, T. H., Leiserson C. E., Rivest, R. L., Stein, C. Introduciton to algorithms 4th edition. Str. 185-205

[5] Horvat, S. 11.22. Sortiranje – 3. dio – Složenost klasičnog i prikaz sorta izborom najmanjeg elementa https://www.youtube.com/watch?v=9rHQtJD1lo4&t=439s (datum pristupa: 4.10.2023.)

Ostalo:

[6] Slika 2 - https://media.geeksforgeeks.org/wp-content/uploads/20230302113306/mypic.png