Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Uroš Hekić

Primerjava algoritmov porazdeljevanja pri hitrem urejanju

diplomsko delo

interdisciplinarni univerzitetni
študijski program prve stopnje
računalništvo in matematika

Mentor: doc. dr. Jurij Mihelič

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Hitro urejanje je eden izmed najbolj znanih in raziskovanih algoritmov urejanja in posledično daje vtis, kot da o njem že skoraj vse vemo. Kljub temu se je v zadnjih letih pojavilo kar nekaj novih odkritij, povezanih predvsem z večpivotnim porazdeljevanjem, ki se je v teoriji dolgo smatralo za neučinkovito, praktično testiranje pa je pokazalo, da se še kako splača. V okviru diplomske naloge preštudirajte in opišite različne načine porazdeljevanja tako klasične enopivotne kot večpivotne različice. Preštudirajte tudi različne načine tvorjenja testnih primerov in implementirajte program za njihovo generiranje. Na koncu algoritme porazdeljevanja še eksperimentalno ovrednotite in primerjajte.

Zahvaljujem se mentorju doc. dr. Juriju Miheliču za ideje, znanje in pomoč pri izdelavi diplomske naloge. Zahvalil bi se tudi Lenki in družini za podporo ter vzpodbudne besede.

Kazalo

Povzetek

Naslov: Primerjava algoritmov porazdeljevanja pri hitrem urejanju

Avtor: Uroš Hekić

Diplomska naloga obravnava problem urejanja in opisuje različne načine porazdeljevanja pri hitrem urejanju ter morebitne izboljšave le-teh. Opravljena je bila eksperimentalna primerjava algoritmov, ki implementacije različnih algoritmov primerja na podlagi časa izvajanja, števila primerjav, števila premikov in števila rekurzivnih klicev v odvisnosti od velikosti vhodnih podatkov. Predstavljen in implementiran je nov model generiranja testnih primerov, ki pokrije obstoječe testne scenarije iz analiz drugih avtorjev ter doda nove. Ker je hitrost urejanja spremenljiv problem, na katerega močno vpliva arhitektura strojne opreme, je ponovljivo in primerljivo testiranje algoritmov na enakih ali večjih testnih scenarijih ključnega pomena.

Ključne besede: urejanje, hitro urejanje, porazdeljevanje.

Abstract

Title: Comparison of quicksort partitioning algorithms

Author: Uroš Hekić

The thesis deals with the sorting problem, various ways of quicksort partitioning, and possible optimizations thereof. We carried out an experimental analysis comparing different algorithms based on the execution time, number of comparisons, number of moves and number of recursive calls with respect to the length of input data. The thesis also presents and implements a new model for generating test cases that covers existing test scenarios from previous analyses, as well as adds new ones. The sorting speed is an ever-changing problem influenced by hardware architecture, which is why it is important to have a repeatable and comparable way of testing algorithms on the same or larger test scenarios.

Keywords: sorting, quicksort, partitioning.

Poglavje 1

Uvod

Med programiranjem velikokrat naletimo na problem, ki zahteva, da so podatki urejeni v določenem vrstnem redu. Urejanje se pojavlja na vseh nivojih razvoja, npr. v uporabniških aplikacijah, podatkovnih bazah, pri upravljanju s procesi itd. Zelo zanimiv je algoritem hitrega urejanja, saj se algoritmi, ki gradijo na tej ideji, v praksi uporabljajo še danes – po več kot 50 letih od odkritja.

Večina člankov, ki opisujejo na novo odkrite algoritme, kvečjemu primerja svojo implementacijo z osnovno verzijo hitrega urejanja ali pa z verzijo, ki so jo nadgradili. Avtorji posameznih člankov prav tako uporabljajo različne testne primere za ovrednotenje hitrosti, zato je primerjava različnih algoritmov skoraj nemogoča. Prav tako se pri primerjavi uporabi zadnja implementacijo algoritma, ki že vsebuje optimizacije na procesorskem nivoju. Majhne spremembe v algoritmu lahko pohitrijo čas izvajanja ali pa ga upočasnijo, saj se določene optimizacije med seboj izključujejo. Nas pa je zanimalo, kako različni načini porazdeljevanja vplivajo na hitrost, št. primerjav, št. zamenjav in št. rekurzivnih klicev. Cilj naloge je bil tako zbrati različne algoritme hitrega urejanja, jih implementirati in na jih na enotnih testnih podatkih ovrednotiti.

1.1 Problem urejanja

Za definiranje problema urejanja potrebujemo definicijo delne urejenosti, ki jo najdemo v [4].

Problem urejanja je problem, kjer za poljuben vhodni seznam dolžine n, tj. $a = (a_1, a_2, ..., a_n)$, in dano delno urejenost $\leq$ iščemo njegovo permutacijo $a' = (a'_1, ..., a'_n)$, da velja $a'_1 \leq a'_2 \leq ... \leq a'_n$.

Seznam bomo predstavili s podatkovno strukturo tabele (angl. array), ki je indeksirana od 0 do $n-1$. Tabelo bomo označevali z a, njene elemente pa z $a[i]$, kjer velja $0 \leq i \leq n-1$.

1.2 Zgodovina

Charles Antony Richard Hoare je algoritem hitrega urejanja razvil leta 1959 [13] med služenjem obveznega vojaškega roka v britanski kraljevi mornarici. V tem času je bil gostujoč študent na moskovski univerzi in je delal na projektu za avtomatsko strojno prevajanje ruščine v angleščino v sklopu nacionalnega fizikalnega laboratorija (National Physical Laboratory, UK).

V tistih časih so bili prevodi besed shranjeni v abecednem vrstnem redu na magnetnem traku. Zato se je splačalo, preden so vsako besedo poiskali v slovarju, besede v stavku najprej urediti po abecedi. S tem so lahko poiskali vse besede v slovarju z enim sprehodom po magnetnem traku. Njegova prva ideja je bil algoritem, ki ga danes imenujemo urejanje z mehurčki (angl. bubble sort), ampak je zaradi počasnosti algoritma razmišljal dalje.

Njegova naslednja ideja je bilo hitro urejanje. Napisal je algoritem za porazdeljevanje, ni pa znal napisati programa, ki bi poskrbel za neurejeni podtabeli. Kasneje je odkril članek o rekurziji v programskem jeziku ALGOL, in tako leta 1961 objavil članek ter izvorno kodo v reviji Communications of the Association for Computing Machinery.

Avtor je v intervjuju leta 2009 [13] dejal, da meni, da je hitro urejanje edini zanimiv algoritem, ki ga je kadarkoli razvil.

1.3 Hitro urejanje

Algoritem hitrega urejanja (angl. quicksort) oz. urejanje s porazdeljevanjem je algoritem za urejanje podatkov, ki ga je leta 1959 razvil in leta 1961 objavil Sir Charles Richard Antony Hoare.

Algoritem temelji na metodi načrtovanja algoritmov *deli in vladaj*, kjer večji problem rešimo tako, da ga razdelimo na več manjših podproblemov, ki jih rekurzivno rešimo, iz njihovih rešitev pa sestavimo rešitev prvotnega problema. Algoritem je tako sestavljen iz treh faz deli, vladaj in združi, ki jih predstavimo v nadaljevanju.

Deli Quicksort razdeli vhodno tabelo na dva dela, glede na predhodno izbrano vrednost pivota (ki je lahko ena izmed vrednosti iz tabele), na *spodnjo* in *zgornjo* tabelo. V spodnji tabeli so vsi elementi, ki so manjši od pivota, v zgornji pa vsi, ki so večji od pivota. Elementi, enaki pivotu so ponavadi lahko v poljubnem delu.

Vladaj Obe podtabeli rekurzivno uredimo po enakem postopku, dokler ne pridemo do prazne tabele ali enega elementa (ali dovolj majhne tabele). Te so trivialno urejene, njihovo reševanje oz. obvladovanje pa je preprosto, zato tej fazi pravimo vladaj.

Združi Ker sta podtabeli vsaka zase urejeni, elementi pa so pravilno porazdeljeni glede na pivot, je celotna tabela že urejena.

Algoritem hitrega urejanja tabele a dolžine n, ki uredi elemente na mestih $left, ..., right$, opisuje algoritem 1.1. Če je indeks $left$ večji ali enak indeksu $right$, ustavimo urejanje (kar vidimo v vrsticah 2 in 3). V vrstici 4 tabelo s funkcijo Partition porazdelimo na 2 dela (kot je opisano zgoraj v koraku deli) in s tem dobimo indeks pivotnega elementa, tj. $pivotIndex$, vrednost pivota pa je $a[pivotIndex]$. Nato v vrsticah 5 in 6 rekurzivno uredimo podtabeli levo in desno od pivotnega elementa. V splošnem lahko algoritmi uporabljajo $s-1$ pivotov in s tem tabelo porazdelijo na s podtabel, ki jih na koncu rekurzivno uredimo.

```
1 fun Quicksort(a, left, right)
2     if left ≥ right then
3         return
4     pivotIndex = Partition(a, left, right)
5     Quicksort(a, left, pivotIndex − 1)
6     Quicksort(a, pivotIndex + 1, right)
```

Algoritem 1.1: Algoritem hitrega urejanja

Po vsakem porazdeljevanju ostaneta dve tabeli, ki ju je treba urediti. Če je katera izmed tabel prazna ali sestavljena iz enega samega elementa, jo ignoriramo in proces nadaljujemo na drugi tabeli. Tudi če v tabeli ostane dovolj majhno število elementov (odvisno od lastnosti računalnika), lahko za njihovo urejanje uporabimo program, prilagojen za urejanje manjšega števila elementov (npr. urejanje z vstavljanjem). V primeru, ko sta obe tabeli dovolj veliki, bo potrebno odložiti urejanje ene, dokler popolnoma ne uredimo druge. Med tem pa moramo (na skladu rekurzije) hraniti indekse prvega in zadnjega elementa podtabele.

Časovna zahtevnost hitrega urejanja v povprečnem primeru je $\Theta(n \log n)$, kjer n predstavlja število elementov v vhodni tabeli. Časovna zahtevnost v najslabšem primeru je $\Theta(n^2)$.

Kljub temu da obstajajo algoritmi za urejanje s časovno zahtevnostjo $\Theta(n \log n)$ v najslabšem primeru (npr. urejanje z zlivanjem in urejanje s kopico), pa je v praksi hitro urejanje še vedno hitrejše, saj ima manjšo konstanto, ki v asimptotični notaciji časovne zahtevnosti ni vidna.

Pri implementaciji algoritmov bomo elemente urejali v nepadajočem vrstnem redu. Omejili se bomo na urejanje celih števil.

1.4 Izboljšave

Implementacije hitrega urejanja se razlikujejo glede na izbiro pivota, glede na število pivotov, glede na porazdeljevanja, glede na število podtabel in druge

optimizacije, ki jih bomo predstavili v tem razdelku.

1.4.1 Majhne tabele

Quicksort ni primeren za majhne vhodne podatke, še posebej zaradi rekurzije. Že Hoare je zato predlagal uporabo bolj učinkovite metode za manjše število elementov v tabeli. Ena izmed metod, ki so učinkovite pri manjših vhodnih podatkih, je urejanje z vstavljanjem (angl. insertion sort) [12]. Skozi čas se je meja, pri kateri preidemo na urejanje z vstavljanjem, zaradi povečevanja hitrosti procesorjev spreminjala.

1.4.2 Rekurzija

Pri rekurzivni implementaciji porazdeljevanja največjo težavo predstavlja velika količina potrebnega pomnilnika (prostora) za (implicitni) sklad rekurzije. Če je del tabele $a[0]$, ..., $a[n-1]$, se bo program rekurzivno izvedel do globine n in s tem porabil dodatni prostor v odvisnosti od n. Že Hoare je izpostavil, da se temu lahko izognemo tako, da najprej rekurzivno kličemo krajšega izmed dveh tabel. Globina rekurzije je s tem omejena na $\log_2 n$, z ročno implementacijo sklada pa lahko prihranimo še več časa.

1.4.3 Robni primeri

Izvirna implementacija je neučinkovita v nekaterih primerih, ki se pogosto pojavijo v praksi.

Predpostavimo, da so števila $a[0], a[1], ..., a[n-1]$ že urejena. Potem bo $a[0]$ prvi pivotni element, leva podtabela bo prazna, desna podtabela pa bo sestavljena iz elementov $a[1], ..., a[n-1]$. Enako se bo zgodilo v naslednji iteraciji – desna podtabela bo vedno manjša samo za en element. Algoritem se bo izvajal nad tabelami z velikostjo: $n, n-1, n-2, ...$ in časovna zahtevnost izvajanja bo $O(n^2)$. Enako velja za tabelo, ki je urejena v nasprotnem vrstnem redu.

Obstaja veliko načinov, kako zmanjšati verjetnost, da se znajdemo v robnem primeru. Namesto da uporabimo prvi element polja kot pivotni element, lahko poskusimo uporabiti kakšen drug fiksni element, npr. sredinski. Hoare je predlagal izbiro naključnega elementa, kar bi nam zagotovilo, da do robnega primera (z zelo veliko verjetnostjo) v praksi ne bo prišlo, vendar je za generiranje naključnih števil po drugi strani potrebno veliko časa. S to metodo se sicer izognemo nekaj robnim primerom, vendar po nepotrebnem podaljšujemo čas izvajanja.

1.4.4 Mediana treh vrednosti

Naslednja metoda pospeši povprečni čas izvajanja algoritma, hkrati pa poskrbi za to, da se algoritem ne bo izrodil v najslabši robni primer. Hitro urejanje se najbolje obnaša, ko je pivotni element čim bližje sredini tabele oz. leva in desna podtabela vsebujeta čim bolj izenačeno število elementov.

Dober pivotni element lahko zato približno ocenimo z mediano tabele. Približek mediane lahko dobimo z izbiro manjšega vzorca iz polja. Idejo je predlagal že Hoare, ampak je ni razvijal naprej, saj je težko ocenil časovni prihranek. Največ časa prihranimo z uporabo vzorca treh elementov pri vsaki fazi porazdeljevanja. Večji vzorci sicer dajo boljši približek, vendar bistveno ne zmanjšajo časa izvajanja.

Z uporabo vzorčenja se zavarujemo pred izbiro pivotnega elementa, ki bi bil na robu polja. Povprečni čas izvajanja se zmanjša, če izberemo katere koli tri elemente. Singleton je leta 1969 predlagal uporabo prvega, sredinskega in zadnjega.

1.4.5 Porazdeljevanje

Razlike v času izvajanja prinesejo tudi različni načini porazdeljevanja, ki si jih bomo ogledali v naslednjem poglavju.

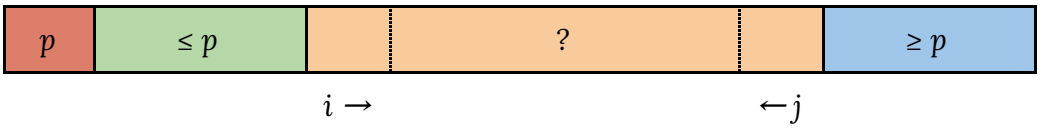
Poglavje 2

Porazdeljevanje

2.1 Hoarejevo porazdeljevanje

Imamo izbran pivot p. Hoare je vzel skrajni levi element tabele oz. dela tabele, ki ga trenutno obdelujemo. Dela tabele omejujeta levi indeks $left$ in desni indeks $right$, ki sta na začetku $left = 0$, $right = n - 1$, kjer n predstavlja število vseh elementov v tabeli.

Med porazdeljevanjem je tabela ločena na tri dele, kot je prikazano na sliki 2.1: *spodnji* del, ki vsebuje elemente, manjše ali enake pivotu, *neobdelani del* ter *zgornji del*, ki vsebuje elemente, večje ali enake pivotu.



Slika 2.1: Splošno stanje tabele pri Hoarejevem porazdeljevanju

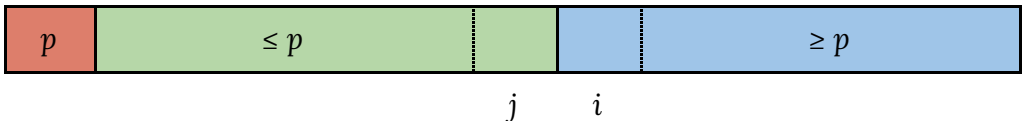
Potek porazdeljevanja je opisan v algoritmu 2.1. S pomočjo dveh indeksov pregledujemo elemente tabele, to sta spodnji indeks i in zgornji indeks j. Kot vidimo v vrsticah 5 in 6, na začetku spodnji indeks i inicializiramo na začetek podtabele, ki jo trenutno urejamo (indeks $left$), zgornji indeks j pa na konec tabele (indeks $right$). Na začetku sta tako zgornji kot spodnji del prazna. Spodnji indeks se pomakne navzgor, če je vrednost v tabeli na indeksu i, $a[i]$,

manjša ali enaka pivotu[1], tj. $a[i] \leq p$, in se pomika navzgor, dokler ne najde elementa, ki je večji od pivota. V tem primeru se spodnji indeks ustavi, kar lahko vidimo v vrstici 8.

Z iteriranjem nadaljuje (kot je opisano v 9. vrstici) zgornji indeks, ki se pomakne navzdol, če je vrednost v tabeli večja ali enaka pivotu, tj. $a[j] \geq p$, in se pomika navzdol, dokler ne najde elementa, ki je manjši od pivota.

Na tej točki smo prepričani, da sta elementa na indeksih i in j na napačnih straneh tabele, zato ju zamenjamo. Po zamenjavi oba indeksa premaknemo za ena v pripadajočo smer, spodnji indeks pa nadaljuje iteriranje. Postopek ponavljamo, dokler se indeksa ne prekrižata, tj., ko spodnji indeks kaže na višje[2] mesto v tabeli od zgornjega indeksa, $i > j$. To je razlog, da Hoarejevo porazdelitev imenujemo tudi *tehnika križanja kazalcev*.

Algoritem se konča v enem izmed dveh možnih stanj (sliki 2.2 in 2.3). Pri prvem možnem stanju se meji spodnjega in zgornjega dela prekrijeta, tj. spodnji indeks kaže na višje mesto kot zgornji, $i > j$. V tem primeru je potrebno pivot (ki se nahaja na začetku tabele $a[left]$) zamenjati z j-tim elementom [8].

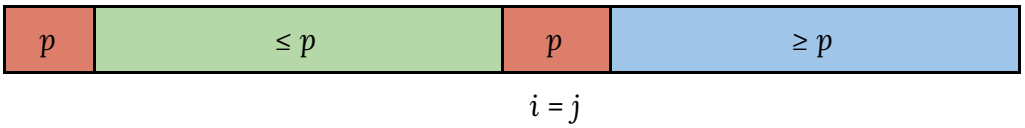


Slika 2.2: Prvo izmed dveh končnih stanj tabele pri Hoarejevem porazdeljevanju

Pri drugem možnem stanju se meji zgornjega in spodnjega dela dotakneta, tj. spodnji indeks je enak zgornjemu, $i = j$. Element na tem mestu je manjši ali enak pivotu in hkrati večji ali enak pivotu, torej je enak pivotu. V tem

[1] V originalnem članku Hoare pivot imenuje meja (angl. bound), indeksi so kazalci (angl. pointers), indeks, kjer pivot na koncu particioniranja loči spodnje in zgornje elemente, je ločnica (angl. dividing line), namesto o tabeli pa govori o datotekah (angl. files) in poddatotekah (angl. subfiles) ali segmentih.

[2] Spodnji indeks lahko kaže na višje ali enako mesto v tabeli od zgornjega indeksa, torej $i \geq j$.



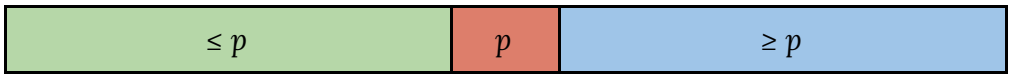
Slika 2.3: Drugo izmed dveh končnih stanj tabele pri Hoarejevem porazdeljevanju

primeru zamenjava elementov na i-tem in j-tem mestu ni potrebna, saj je pivot že na pravem mestu [8]. V spodnji del vključimo še pivot na prvem mestu, da dobimo enako stanje na sliki 2.4.

Porazdeljevanje je s tem sicer končano, urejanje tabele pa še ne. Zato rekurzivno pokličemo hitro urejanje nad zgornjim in spodnjim delom tabele, kar vidimo v vrsticah 13 in 14.

```
1  fun QuicksortHoare(a, left, right)
2      if left ≥ right then
3          return
4      p = a[left]
5      i = left
6      j = right + 1
7      while true do
8          do i = i + 1 while i < right and a[i] ≤ p
9          do j = j − 1 while j > left and a[j] ≥ p
10         if i ≥ j break
11         swap(a, i, j)
12     swap(a, left, j)
13     QuicksortHoare(a, left, j − 1)
14     QuicksortHoare(a, j + 1, right)
```

Algoritem 2.1: Hoarejevo hitro urejanje

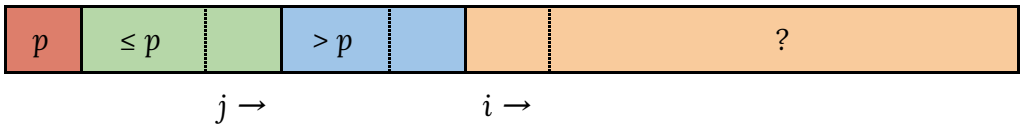


Slika 2.4: Končno stanje tabele s pivotom na pravilnem mestu

2.2 Lomutojevo porazdeljevanje

Spodaj opisano particioniranje pripisujejo Nicu Lomutu, populariziral pa ga je Jon Bentley v knjigi *Programming Pearls* [5] in kasneje Cormen et al. v knjigi *Introduction to Algorithms* [7, 2].

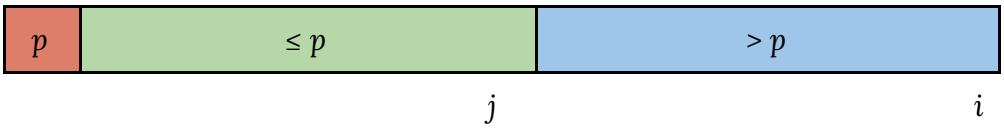
Ideja algoritma je, da se v vsaki fazi porazdeljevanja sprehodi po vseh elementih tabele z eno zanko, zato ga imenujemo tudi *enozančno* porazdeljevanje. Prvotna implementacija algoritma za pivotni element vedno izbere skrajni desni element, mi pa bomo zaradi konsistentnosti z ostalimi algoritmi izbrali skrajni levi element. Tekom porazdeljevanja je tabela razdeljena na štiri dele, podobno kot pri Hoarejevem hitrem urejanju, različen je le vrstni red. Splošna shema tabele je prikazana na sliki 2.5. Na začetku so od indeksa q do (vključno) i elementi, manjši (ali enaki) od pivota p (na indeksu r), od $i+1$ do $j-1$ so elementi, večji od pivota, med j in $r-1$ sledijo neurejeni elementi, na zadnjem, r-tem mestu pa je pivot [7].



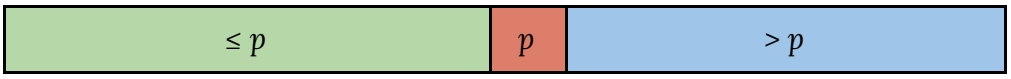
Slika 2.5: Splošno stanje tabele pri Lomutojevem porazdeljevanju

Lomutojevo porazdeljevanje je opisano v algoritmu 2.2. Algoritem inicializira vrednosti tako, da sta spodnji in zgornji del prazna, neurejeni del pa vsebuje celotno tabelo razen pivotnega elementa. Iteracijski indeks i se sprehodi čez tabelo in v vsakem koraku primerja vrednost v tabeli na i-tem mestu, $a[i]$, s pivotom, ugotovi, ali element pade v spodnji ali zgornji del, in izvede ustrezne zamenjave. Če je element manjši od pivota, $a[i] < p$, potem pade v spodnji del. Zato, kot je opisano v vrstici 8, inkrementiramo

j (povečamo spodnji del), zamenjamo i-ti in j-ti element (začasno ga uvrstimo v zgornji del), nato pa povečamo še i (i-ti element se iz zgornjega dela prestavi v spodnji del). Sedaj je i-ti element na pravem mestu. Če pa je element večji ali enak pivotu, $a[i] \geq p$, pa samo povečamo i (s tem povečamo zgornji del) in nadaljujemo z naslednjim elementom. Po končani zanki je neobdelani del prazen, pivot pa je še vedno na prvem oz. napačnem mestu, kot je prikazano na sliki 2.6. Zamenjamo ga z zadnjim elementom spodnjega dela, tj. $a[j]$, kar vidimo v vrstici 9.



Slika 2.6: Končno stanje tabele s pivotom na prvotnem mestu



Slika 2.7: Končno stanje tabele s pivotom na pravilnem mestu

```
1  fun QuicksortLomuto
2      if left ≥ right then
3          return
4      p = a[left]
5      j = left - 1
6      for i = left + 1 to right do
7          if a[i] < pivot then
8              swap(a, ++j, i)
9      swap(a, left, j)
10     QuicksortLomuto(a, left, j - 1)
11     QuicksortLomuto(a, j + 1, right)
```

Algoritem 2.2: Lomutojevo hitro urejanje

Porazdeljevanje je končano, algoritem mora pa še rekurzivno urediti spodnji in zgornji del, kar se zgodi v vrsticah 10 in 11. Končno stanje je predstavljeno na sliki 2.7.

2.3 Sedgewickovo porazdeljevanje

Sedgewick je leta 1975 v svoji doktorski tezi predstavil naslednji algoritem, ki prav tako temelji na tehniki križanja kazalcev [12]. V bistvu gre za spremenjeno različico Hoarejevega hitrega urejanja z nekaterimi razlikami. Splošna shema je enaka kot shema 2.1 pri Hoarejevem porazdeljevanju.

V primerjavi s Hoarejevo različico se notranja zanka pri primerjavi s pivotom ustavi pri elementu, ki je enak pivotu oz. se pivot in trenutni element primerjata s strogim enačajem (namesto z neenačajem ali enako).

Sedgewickovo porazdeljevanje je opisano v algoritmu 2.3. Avtor se znebi dodatne primerjave trenutnega elementa s spodnjim čuvajem (angl. sentinel) v vrstici 8. Zaradi primerjave pivota s strogim neenačajem funkcijo čuvaja prevzame pivotni element.

Zunanja zanka je zapisana v obliki do-while, namesto while-do.

Avtor je v originalnem algoritmu za pivotni element vedno izbral skrajni desni element (Hoare je izbral skrajni levi element).

```
1  fun QuicksortSedgewick(a, left, right)
2      if left ≥ right then
3          return
4      p = a[right]
5      i = left − 1
6      j = right
7      do
8          do i = i + 1 while i < right and a[i] < p
9          do j = j − 1 while a[j] > p
10         if j > i then
11             swap(a, i, j)
```

```
12      while j ≥ i
13      swap(a, i, right)
14      QuicksortSedgewick(a, left, i − 1)
15      QuicksortSedgewick(a, i + 1, right)
```

Algoritem 2.3: Sedgewickovo hitro urejanje

Naša implementacija zaradi konsistentnosti z ostalimi algoritmi kot pivotni element izbere skrajni levi element.

2.4 Wirthovo porazdeljevanje

Niklaus Wirth je v svojem učbeniku o algoritmih in podatkovnih strukturah iz leta 1985 opisal svojo različico hitrega urejanja. Wirthov algoritem [15] prav tako temelji na tehniki porazdeljevanja s križanjem. Splošna shema je enaka kot shema 2.1 pri Hoarejevem porazdeljevanju. Ena izmed razlik je, da se notranja zanka pri elementu, ki je enak pivotu (oziroma zamenja pivot s trenutnim elementom), ne pomakne naprej in algoritem s tem opravi več zamenjav. V primeru, ko urejamo n enakih elementov, zato opravimo $n/2$ zamenjav. Nepotrebnim zamenjavam se lahko izognemo tako, da v notranjih zankah (v vrsticah 6 in 7 sheme 2.4) trenutni element v tabeli s pivotom primerjamo z neenačajem ali enako: $a[i] \leq p$ in $p \leq a[j]$. Vendar v tem primeru pivot p ne bi služil več kot čuvaj in zanka bi se lahko pomaknila izven tabele. Wirth je ugotovil, da preprostost ustavitvenih pogojev odtehta nepotrebne zamenjave, saj se le te redko pojavijo v povprečnem naključnem vhodnem primeru.

Algoritem inicializira $l = left, r = right$ in inkrementira/dekrementira i ter j v do-while zanki v vrsticah 6–11. Prav tako ne izvede zadnje zamenjave pred rekurzivnim klicem. Algoritem se razlikuje tudi po tem, da za pivot izbere naključni element. V naši implementaciji bomo vzeli skrajni levi element, saj nas zanimajo samo izboljšave v porazdeljevanju.

```
1   fun QuicksortWirth(a, left, right)
```

```
2       if left ≥ right then return
3       p = a[left]
4       i = left, j = right
5       do
6           while a[i] < p do i = i + 1
7           while p < a[j] do j = j − 1
8           if i <= j then
9               swap(a, i, j)
10              i = i + 1
11              j = j − 1
12      while i > j
13      QuicksortWirth(a, left, j)
14      QuicksortWirth(a, i, right)
```

Algoritem 2.4: Wirthovo hitro urejanje

2.5 Aho-Hopcroft-Ullman porazdeljevanje

Aho, Hopcroft in Ullman v knjigi *The Design and Analysis of Computer Algorithms*, ki je bila izdana leta 1974 [4], opišejo različico hitrega urejanja, ki temelji na tehniki križanja kazalcev. Tudi različica hitrega urejanja, ki so jo opisali zgoraj omenjeni avtorji, temelji na tehniki križnega porazdeljevanja. Splošna shema je enaka kot shema 2.1 pri Hoarejevem porazdeljevanju.

Porazdeljevanje je opisano v algoritmu 2.5. V vrsticah 8 in 9 lahko vidimo, da algoritem začne z zgornjim kazalcem pri zadnjem elementu in se pomika navzdol, dokler ne najde elementa, manjšega od pivotnega elementa (ali pride do začetka tabele). Potem pa skeniranje začne spodnji kazalec, ki se pomika navzgor, dokler ne najde elementa, ki je večji ali enak pivotnemu elementu (ali pa pride do konca tabele). Če je indeks spodnjega kazalca strogo manjši od indeksa zgornjega kazalca, ju zamenja, kar pomeni, da se zgornji kazalec ne ustavlja pri elementih, ki so enaki pivotu, spodnji kazalec pa se ustavlja. To lahko vidimo v pogojih zanke v vrsticah 8 in 9. Algoritem nima čuvajev,

notranji zanki v vsaki iteraciji preverjata, ali se indeks že nahaja izven tabele. Prav tako algoritem ne potrebuje zamenjave elementov po koncu zunanje zanke.

```
1  fun QuicksortAHU(a, left, right)
2      if left ≥ right then
3          return
4      p = a[left]
5      i = left
6      j = right
7      while i ≤ j do
8          while a[j] ≥ p and j ≥ left do j = j − 1
9          while a[i] < p and i ≤ right do i = i + 1
10         if i < j then
11             swap(i, j)
12             i = i + 1
13             j = j + 1
14     if i == left then j = j + 1
15     QuicksortAHU(a, left, i − 1)
16     QuicksortAHU(a, j + 1, right)
```

Algoritem 2.5: Aho-Hopcroft-Ullman hitro urejanje

2.6 Sedgewickovo dvopivotno porazdeljevanje

Namesto da tabelo porazdelimo na dve particiji z enim pivotom, jo lahko razdelimo na s particij s $s-1$ pivoti. Sedgewick je v svojem doktoratu obravnaval primer s tremi particijami $s = 3$ in predlagal sledeči algoritem. Izkazalo pa se je, da je le-ta veliko slabši od klasičnega Hoarejevega porazdeljevanja [14].

Algoritem izbere dva pivotna elementa p in q, pri čemer velja $p \leq q$. Za spodnji pivot p izberemo skrajni levi element, za zgornji pivot q pa izberemo skrajni desni element. Kot lahko vidimo na shemi 2.8, algoritem razdeli tabelo z elementi med indeksi $left$, ..., $right$ na 5 delov. *Spodnji del* na

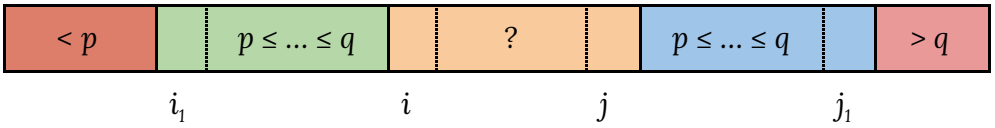
indeksih $left, ..., i_1 - 1$ vsebuje elemente, ki so manjši od spodnjega pivota p. *Spodnji srednji del*, na indeksih $i_1, ..., i - 1$, vsebuje elemente, ki so večji ali enaki od spodnjega pivota p ali manjši ali enaki zgornjemu pivotu q. *Neobdelani del*, $i, ..., j$, vsebuje vse neobdelane elemente. *Zgornji srednji del*, $j + 1, ..., j_1$, prav tako vsebuje elemente, ki so večji ali enaki p ali manjši ali enaki q. *Zgornji del*, $j_1 + 1, ..., right$, pa vsebuje elemente, ki so večji od zgornjega pivota q.

Algoritem poženemo z vhodnima parametroma $left = 0$ in $right = n-1$, kjer n predstavlja število elementov v tabeli. Algoritem inicializira $i = i_1 = left$ in $j = j_1 = right$. Za levi pivot p izbere skrajni levi element $p = a[left]$, za desni pivot q pa skrajni desni element $q = a[right]$. Če je levi pivot večji od desnega, ju zamenjamo, tako da je $p \leq q$.

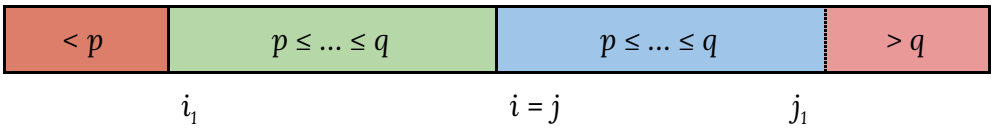
Najprej iteriramo po indeksu i, ki se pomika navzgor po tabeli, dokler ne najde elementa, večjega od zgornjega pivota, $a[i] > q$. Če je $i \geq j$, sta se kazalca prekrižala in prekinemo zunanjo zanko. Če je trenutni element manjši od pivota, potem i-ti element shranimo na mesto i_1-tega elementa, povečamo i_1 ter na i-to mesto shranimo $a[i_1]$ (elementov ne zamenjujemo, ampak jih zapišemo na prejšnje mesto od njihovega partnerja za zamenjavo, tako ne rabimo začasno shranjevati elementa). Sicer se samo premaknemo za ena navzgor. Iteriranje (v vrstici 12, shema 2.6) nadaljuje zgornji indeks j, ki se pomika navzdol po tabeli in se obnaša simetrično kot zgoraj.

Če nismo izstopili iz zunanje zanke, nato v vrsticah 22–24 postavimo j-ti element v spodnji del, i-ti element pa v zgornji del. Ko izstopimo iz zunanje zanke, pa potem v vrstici 26 popravimo vrednosti i_1 in j_1, ki ju uporabimo za razdelitev tabele med pivoti. Končno stanje tabele vidimo na sliki 2.9

Na koncu samo še rekurzivno pokličemo funkcijo nad spodnjim delom $(left, ..., i_1 - 1)$, združenima srednjima deloma $(i_1, ..., j_1)$ in zgornjim delom $(j_1 + 1, ..., right)$.



Slika 2.8: Splošno stanje tabele pri Sedgewickovem dvopivotnem porazdeljevanju



Slika 2.9: Končno stanje tabele pri Sedgewickovem dvopivotnem porazdeljevanju

```
1  fun DualPivotQuicksortSedgewick(a, left, right)
2      if left ≥ right then return
3      i = left; i1 = left; j = right; j1 = right
4      p = a[left]; q = a[right]
5      if p > q then
6          swap(a, p, q)
7      while true do
8          i = i + 1
9          while a[i] <= q do
10             if i >= j then break outer while
11             if a[i] < p then
12                 a[i1] = a[i]; i1 = i1 + 1; a[i] = a[i1]
13             i = i + 1
14
15         j = j - 1
16         while a[j] >= p do
17             if a[j] > q then
18                 a[j1] = a[j]; j1 = j1 - 1; a[j] = a[j1]
19             if i >= j then break outer while
```

```
20              j = j - 1
21
22          a[i1] = a[j]; a[j1] = a[i]
23          i1 = i1 + 1; j1 = j1 - 1
24          a[i] = a[i1]; a[j] = a[j1]
25
26      a[i1] = p; a[j1] = q
27
28      DualPivotQuicksortSedgewick(a, left , i1 - 1)
29      DualPivotQuicksortSedgewick(a, i1 + 1, j1 - 1)
30      DualPivotQuicksortSedgewick(a, j1 + 1, right )
```

Algoritem 2.6: Sedgewickovo dvopivotno hitro urejanje

2.7 Trosmerno porazdeljevanje

Jon Louis Bentley in Malcolm Douglas McIlroy nista bila zadovoljna z obstoječo implementacijo algoritma *qsort* v standardni C knjižnici, zato sta se odločila napisati svojo. Objavila sta jo leta 1993 v članku "Engineering a sort function". Njuna sodelavca sta dve leti pred tem našla primer, ki bi ga algoritem moral urediti v nekaj minutah, izvajal pa se je več ur. Če ga ne bi ustavila, bi se izvajal še več tednov [6].

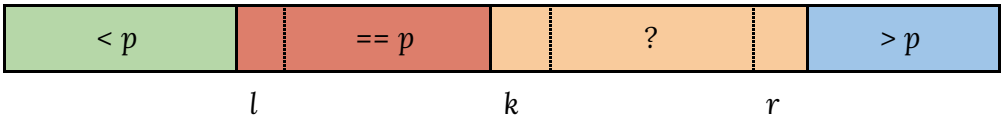
Njun pristop temelji na problemu nizozemske zastave (Dutch National Flag), kjer urejamo tabelo poljubne dolžine, ki je sestavljena samo iz 3 različnih elementov: 0, 1, 2.

Prejšnje verzije hitrega urejanja predpostavljajo, da so vsi elementi različni oz. ne uporabijo informacije o elementih, ki so enaki pivotu. V primeru, da imamo elemente, ki so enaki pivotu, jih večkrat po nepotrebnem primerjamo. Trosmerno hitro urejanje upošteva elemente, enake pivotu – več kot jih je, bolje bo deloval. Porazdelitev s pivotom tabelo razdeli na tri dele: elemente, manjše od pivota, enake pivotu in strogo večje od pivota.

2.7.1 Poenostavljena različica

Tabela je sestavljena iz štirih delov, kot je prikazano na sliki 2.10:

- *spodnji del*: vsebuje elemente, manjše od pivota
- *srednji del*: vsebuje elemente, enake pivotu
- *neobdelani del*: po njem se sprehodi iteracijski element
- *zgornji del*: elementi, večji od pivota



Slika 2.10: Splošno stanje tabele pri trosmernem porazdeljevanju

Spodnji in zgornji del omejujeta indeksa l in r, indeks k (sprehajalni element) označuje položaj trenutnega elementa in mejo med srednjim ter neobdelanim delom. Na začetku izvajanja sta spodnji in zgornji del prazna, srednji del vsebuje pivot, neobdelani del pa vsebuje vse preostale elemente. Psevdokoda algoritma [11] je predstavljena v algoritmu 2.7.

```
1  fun Quicksort3Way(a, left, right)
2      if right ≤ left then return
3      p = a[left]
4      l = left; k = left + 1; r = right
5      while k ≤ r do
6          if a[k] < p then swap(a, l++, k++)
7          else if a[k] > p then swap(a, k, r--)
8          else k++
9      Quicksort3Way(a, left, l - 1)
10     Quicksort3Way(a, r + 1, right)
```

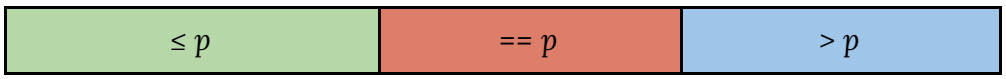
Algoritem 2.7: Trosmerno hitro urejanje, osnovna različica

Z indeksom k iteriramo po neobdelanem delu do zgornjega dela in v vsaki iteraciji primerjamo k-ti element s pivotom.

Če je k-ti element:

- *manjši* od pivota (vrstica 6): sodi v spodnji del, zamenjamo elementa na indeksih l in k ter indeksa inkrementiramo (prestavimo spodnji del in neobdelani del za 1 v desno),
- *enak* pivotu (vrstica 7): sodi v srednji del, zato je že na pravem mestu in inkrementiramo samo k (zmanjšamo neobdelani del),
- *večji* od pivota (vrstica 8): sodi v zgornji del, zamenjamo k in r, in r dekrementiramo (prestavimo začetek zgornjega dela za 1 v levo).

Na koncu zanke so vsi elementi v spodnjem delu manjši od pivota, v srednjem delu so enaki pivotu in v zgornjem delu večji od pivota, kar vidimo na sliki 2.11. Sedaj samo še rekurzivno uredimo spodnji in zgornji del.



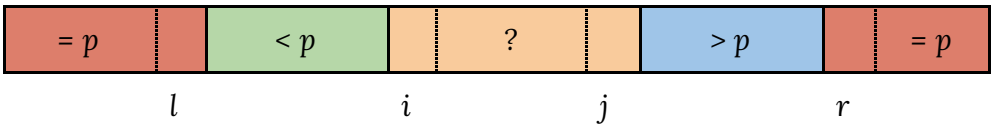
Slika 2.11: Končno stanje tabele pri trosmernem porazdeljevanju

Implementacija se dobro obnaša na zaporedjih z več enakimi elementi. Algoritem je zelo hiter, kadar se v zaporedju nahaja čim manjše število različnih elementov oz. posledično veliko število pivotu enakih elementov.

2.7.2 Bentley-McIlroy porazdeljevanje

V praksi je ponavadi manj elementov, podobnih pivotu, kot tistih, ki so različni od pivota. V tem primeru je bolj smiselno, da je *srednji del* iz poenostavljene verzije na robu tabele, saj pri tem naredimo manj zamenjav. Še bolj učinkovita je simetrična različica, kjer imamo elemente, enake pivotu, na obeh koncih tabele (prikazano na sliki 2.12).

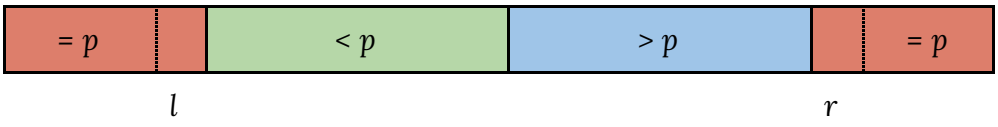
Bentley-McIlroy porazdeljevanje [11, 6] je opisano v algoritmu 2.8.



Slika 2.12: Splošno stanje tabele pri porazdeljevanju Bentley-McIlroy

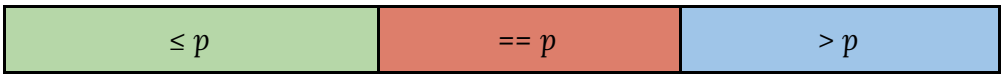
Spodnji del s pivotu enakimi elementi (navzgor) omejuje l, zgornji del pa (navzdol) omejuje r. Podobno kot pri Hoareu oz. porazdelitvi s križanjem kazalcev se iteracijski indeks i pomika navzgor, j pa navzdol. To najdemo v vrsticah 6 do 8 algoritma 2.8.

Ko obdelamo vse neobdelane elemente, sta zgornji in spodnji del pravilno porazdeljena, srednja dela pa se nahajata na robovih tabele. To prikazuje slika 2.13.



Slika 2.13: Končno stanje tabele pri porazdeljevanju Bentley-McIlroy s pivoti na prvotnih mestih

Zato ju moramo z zamenjavami prenesti na sredino (vrstici 13 in 14). Končno stanje je prikazano na sliki 2.14. Nato algoritem rekurzivno pokličemo nad spodnjim in zgornjim delom.



Slika 2.14: Končno stanje tabele pri porazdeljevanju Bentley-McIlroy

```
1  fun QuicksortBentleyMcIlroy(a, left, right)
2      if right <= left then return;
3      p = a[left];
4      l = left; i = left; j = right + 1; r = j
```

```
5       while true do
6           while a[++i] < p and i < right
7           while a[--j] > p
8           if (i >= j) then break
9           swap(a, i, j)
10          if a[i] == p then swap(a, ++l, i)
11          if a[j] == p then swap(a, --r, j)
12      i = j + 1
13      for k = left to l do swap(a, k, j--)
14      for k = right downto r do swap(a, k, i++)
15      QuicksortBentleyMcIlroy(a, left, j)
16      QuicksortBentleyMcIlroy(a, i, right)
```

Algoritem 2.8: Bentley-McIlroy hitro urejanje

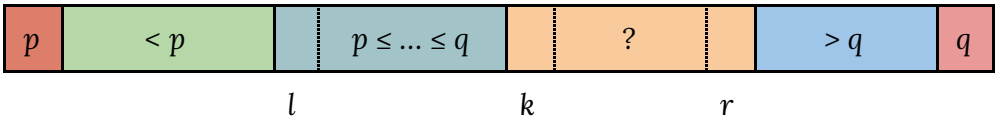
2.8 Porazdeljevanje Jaroslavskega

2.8.1 Osnovna različica

Shema razdeli tabelo na šest delov oz. na dva pivota in še štiri dele [16], kot je prikazano na sliki 2.15:

- *levi pivot* p
- *spodnji del*: elementi, manjši od p
- *srednji del*: elementi, med p in q
- *neobdelani elementi*
- *zgornji del*: elementi, večji od q
- *desni pivot* q

Algoritem predpostavlja, da je levi pivot manjši od desnega, $p < q$. Trenutni (še neobdelani) element označuje indeks k:



Slika 2.15: Splošno stanje tabele

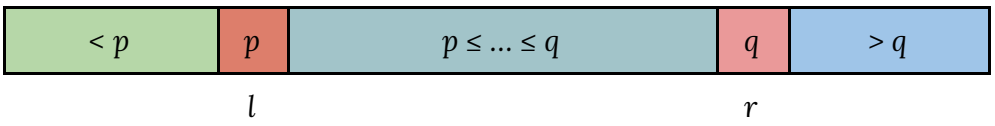
– če je trenutni element manjši od levega pivota (vrstica 7), $a[k] < p$, ga zamenjamo z l-tim elementom ter povečamo indeksa l in k (s tem srednji del prestavimo za 1 v desno),

– če je trenutni element večji od desnega pivota (vrstica 8), $a[k] > q$, ga zamenjamo z r-tim elementom in zmanjšamo r (zgornji del se razširi za 1 v levo),

– sicer povečamo k (vrstica 9).

k-ti element želimo s čim manj potezami spraviti na pravo mesto.

Na koncu zanke je potrebno levi in desni pivot premakniti na ustrezno mesto, kot je opisano v vrsticah 10 in 11. Končno stanje osnovne različice prikazuje slika 2.16.



Slika 2.16: Končno stanje tabele

Nato s 3 rekurzivnimi klici funkcije uredimo spodnji, srednji in zgornji del.

Implementacija osnovne različice iz prejšnjega odstavka [10] je opisana v algoritmu 2.9.

```
1 fun QuicksortYaroSimple(a, left, right)
2     if right <= left then return
3     if a[left] > a[right] then swap(a, left, right)
```

```
4       p = a[left], q = a[right]
5       l = left + 1, k = l, r = right - 1
6       while k <= r do
7           if a[k] < p then swap(a, l++, k++)
8           else if a[k] > q then swap(a, k, r--)
9           else k++
10      swap(a, left, --l)
11      swap(a, ++r, right)
12      QuicksortYaroSimple(a, left, l - 1)
13      QuicksortYaroSimple(a, l + 1, r - 1)
14      QuicksortYaroSimple(a, r + 1, right)
```

Algoritem 2.9: Hitro urejanje Jaroslavskega, osnovna različica

2.8.2 Izboljšana različica

Prva pohitritev izhaja iz primera, ko je k-ti element večji od desnega pivota q in bi hoteli k-ti element premakniti v zgornji del oz. zamenjati k-ti in r-ti element. V primeru, da je hkrati r-ti element večji od k-tega, lahko v zanki zmanjšujemo r in k-ti element zamenjamo s tistim, ki bo manjši od r-tega (vrstica 9):

```
while a[r] > q and k < r do r--
```

Po koncu te zanke je r-ti element res manjši od q, zato ga zamenjamo s k-tim (vrstica 10):

```
swap(a, k, r--)
```

Druga pohitritev pa temelji na želji, da bi indeks k inkrementirali na vsaki iteraciji zunanje while zanke. Tega trenutno ne moremo, ko ne velja, da $a[k] < p$ in $a[k] > q$ oz. v else if znotraj zanke. Po zamenjavi je sicer k-ti element res manjši od q, $a[k] < q$, vendar ne vemo, ali je večji ali manjši od p. Zato z dodatno zanko preverimo še ta pogoj:

```
if a[k] < p then swap(a, l++, k)
```

Po tem lahko na koncu zanke brezpogojno povečamo k.

Algoritem z obema pohitritvama [10] je opisan v algoritmu 2.10.

```
1  fun QuicksortYaro(a, left, right)
2      if (right <= left) then return
3      if a[left] > a[right] then swap(a, left, right)
4      p = a[left], q = a[right]
5      l = left + 1, k = l, r = right - 1
6      while (k <= r) do
7          if a[k] < p then swap(a, l++, k)
8          else if a[k] > q) then
9              while a[r] > q and k < r do r--
10             swap(a, k, r--)
11             if a[k] < p then swap(a, l++, k)
12         k++
13     swap(a, --l, left)
14     swap(a, ++r, right)
15     QuicksortYaro(a, left, l - 1)
16     QuicksortYaro(a, l + 1, r - 1)
17     QuicksortYaro(a, r + 1, right);
```

Algoritem 2.10: Hitro urejanje Jaroslavskega

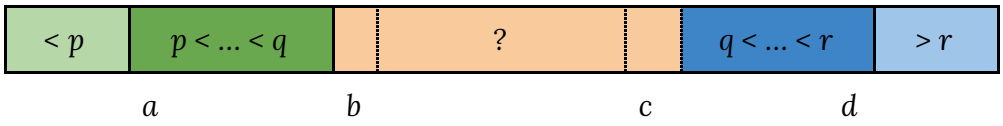
Dvopivotni Quicksort ima $2n\ln(n)$ primerjav, povprečno št. zamenjav pa $0,8n\ln(n)$, medtem ko ima klasični Quicksort $2n\ln(n)$ primerjav in $n\ln(n)$ zamenjav [16].

2.9 Tropivotno porazdeljevanje

Shrinu Kushagra, Alejandro López-Ortiz, J. Ian Munro in Aurick Qiao so leta 2013 objavili članek, ki opisuje nov tropivotni Quicksort.

Algoritem, kot že ime pove, uporablja tri pivote: p, q in r, za katere velja $p < q < r$. V vsaki iteraciji algoritem particionira tabelo v štiri podtabele

(kot prikazano na sliki 2.17) in vsako izmed njih rekurzivno uredi. Na prvi pogled izgleda, kot da algoritem izvaja dva nivoja običajnega enopivotnega hitrega urejanja v enem koraku porazdeljevanja. Toda sredinski pivot q vsebuje več informacij, saj predstavlja mediano med tremi pivoti. To je enako običajnemu algoritmu hitrega urejanja, ki na alternirajočih nivojih kot pivot izbere mediano treh. S tega vidika bi lahko pričakovali, da bo algoritem po hitrosti med klasičnim algoritmom hitrega urejanja in hitrim urejanjem, ki kot pivot uporablja mediano treh števil. V praksi pa se izkaže, da je ta algoritem veliko hitrejši [9].
V psevdokodi tropivotnega porazdeljevanja bomo tabelo označili z A, saj a označuje indeks. Predpostavimo, da so vsi elementi različni. Algoritem za particioniranje uporablja štiri indekse: a, b, c in d, ki ohranjajo zančno invarianto prikazano na spodnji shemi:



Slika 2.17: Splošno stanje tabele pri tropivotnem porazdeljevanju

Indeksa a in b inicializiramo tako, da kažeta na prvi element tabele, c in d pa na zadnji element. Algoritem premika b in c enega proti drugemu in premika elemente na $a[b]$ in $a[c]$ v pravilno podtabelo, zanka pa se konča, ko se b in c križata ($b > c$). V primeru, da je $a[b] < q$, če velja $a[b] < p$, zamenjamo $a[b]$ in $a[a]$ ter inkrementiramo a in b, sicer pa ne naredimo ničesar in samo inkrementiramo b (saj je $a[b]$ na pravem indeksu). Simetrično velja za primer $a[c] > q$. Če je $a[b] > q$ in $a[c] < q$, potem oba elementa zamenjamo po enem izmed štirih primerov in po potrebi povečamo oz. zmanjšamo a, b, c in d:

– Če je $a[b] > r$:

  (a) če je $a[c] < p$, potem zamenjamo $a[b]$ in $a[a]$ ter $a[a]$ in $a[c]$ ter inkrementiramo a,

  (b) sicer zamenjamo $a[b]$ in $a[c]$.

  - V obeh primerih na koncu zamenjamo $a[c]$ in $a[d]$, inkrementiramo b in dekrementiramo c in d.

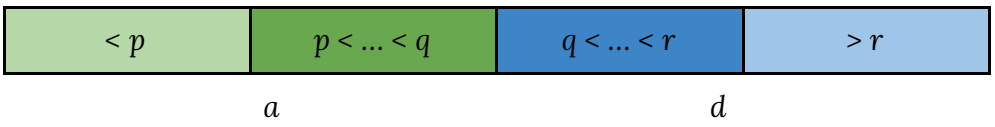
- sicer ($a[b] < r$):

  (a) če je $a[c] < p$, potem zamenjamo $a[b]$ in $a[a]$ ter $a[a]$ in $a[c]$ ter inkrementiramo a,

  (b) sicer zamenjamo $a[b]$ in $a[c]$.

  - V obeh primerih na koncu inkrementiramo b in dekrementiramo c.

Končno stanje tabele po porazdeljevanju je prikazano na sliki 2.18.



Slika 2.18: Končno stanje tabele pri tropivotnem porazdeljevanju

Psevdokoda algoritma za particioniranje [9] je opisana v algoritmu 2.11.

```
1  fun Partition3Pivot(A, left, right)
2      a = left + 2, b = left + 2
3      c = right - 1, d = right - 1
4      p = A[left], q = A[left + 1], r = A[right]
5      while b <= c do
6          while A[b] < q and b <= c do
7              if A[b] < p then
8                  swap(A, a, b)
9                  a = a + 1
10             b = b + 1
11         while A[c] > q and b <= c do
12             if A[c] > r then
```

```
13                  swap(A, c, d)
14                  d = d - 1
15              c = c - 1
16          if b <= c then
17              if A[b] > r then
18                  if A[c] < p then
19                      swap(A, b, a), swap(A, a, c)
20                      a = a + 1
21                  else
22                      swap(A, b, c)
23                  swap(A, c, d)
24                  b = b + 1, c = c - 1, d = d - 1
25              else
26                  if A[c] < p then
27                      swap(A, b, a), swap(A, a, c)
28                      a = a + 1
29                  else
30                      swap(A, b, c)
31                  b = b + 1, c = c - 1
32      a = a - 1, b = b - 1, c = c + 1, d = d + 1
33      swap(A, left + 1, a), swap(A, a, b)
34      a = a - 1
35      swap(A, left, a), swap(A, right, d)
```

Algoritem 2.11: Tropivotno hitro urejanje

Poglavje 3

Eksperimentalna primerjava

Algoritme smo implementirali v programskem jeziku C. Poganjali smo jih na računalniku s procesorjem Intel(R) Core(TM) i7-7700HQ, s frekvenco 2.80 GHz, ki ima 4 jedra (8 niti), 6 MB predpomnilnika in 16 GB DDR4 pomnilnika s frekvenco 1200 MHz. Za prevajanje kode smo uporabili gcc z argumentom za optimizacijo -O3.

3.1 Implementacija algoritmov

Pri izvajanju algoritmov smo zraven velikosti (in drugih parametrov) vhodnih podatkov beležili:

- število primerjav s pivotnim elementom
- število premikov
- število rekurzivnih klicev
- št. ponovitev pri merjenju števila primerjav, premikov in rekurzivnih klicev
- čas izvajanja
- št. ponovitev pri merjenju časa izvajanja

Vse testne primere smo izvedli dvakrat, saj štetje premikov, primerjav in rekurzivnih klicev poveča čas izvajanja. Prvič smo beležili število premikov, število primerjav s pivotnim elementom, število rekurzivnih klicev ter število ponovitev (različnih testnih primerov), ki nam jih je uspelo izvesti znotraj določenega časovnega intervala. Drugič pa smo beležili čas ter prav tako število ponovitev, ki nam jih je uspelo izvesti.

Implementirane algoritme in generator testnih primerov najdemo na mojem Github repozitoriju [3].

3.2 Generator testnih primerov

Algoritme smo poganjali nad testnimi primeri različnih družin in velikostnih razredov. Po pregledu področja smo se odločili, da bomo razširili Bentley-McIlroy zbirko testnih primerov [6], ki jo je med drugimi uporabil tudi Jaroslavski.

Generatorje smo parametrizirali z naslednjimi parametri:

- n: dolžina tabele

- m: modul

- p_1: dodatni parameter za prvi korak

- p_{21}, p_{22}: dodatna parametra za drugi korak

- p_{31}, p_{32}: dodatna parametra za tretji korak

Testne primere smo generirali v treh korakih. Prvi korak zgenerira tabelo n števil. Izbiramo lahko med tremi načini generiranja:

- *Rand(n, m)*: zgenerira n naključnih števil med 0 in $m - 1$,

- *Saw(n, m, p_1)*: zgenerira n števil po formuli $a[i] = (i \cdot p_1) \mod m$

- *Shuffle(n, m)*[1]: Na začetku inicializiramo $j = 0$, $k = 1$. Z iteracijsko spremenljivko i iteriramo od 0 do $n - 1$ in računamo naključna števila med 0 in $m - 1$, kot je opisano v algoritmu 3.1. Če je število enako 0, povečamo k za 2 in vrednost k shranimo v $a[i]$. Če pa je število večje od 0, pa povečamo j za 2 in vrednost j shranimo v $a[i]$.

```
1 fun shuffle(a, n, m)
2     for (i = 0, j = 0, k = 1; i < n; i++)
3         a[i] = rand() % m ? (j += 2) : (k += 2)
```

Algoritem 3.1: Shuffle, eden izmed algoritmov prvega koraka

V drugem koraku lahko vnesemo dodaten šum ali pa postavimo spodnjo ali zgornjo mejo za števila v tabeli.

- *Id*: ohrani tabelo nespremenjeno
- *Plateau(p_{21}, p_{22})*: vsa števila, večja od zgornje meje p_{22}, zmanjša na p_{22}, vsa števila, večja od spodnje meje p_{21}, poveča na p_{21}
- *Dither(p_{21})*: prišteje $i \mod p_{21}$ k $a[i]$: $a[i] = a[i] + (i \mod p_{21})$

V tretjem koraku lahko dodatno permutiramo obstoječo tabelo z izbiro enega izmed naslednjih algoritmov:

- *Id*: ohrani tabelo nespremenjeno
- *Sort*: uredi tabelo
- *Reverse(p_{31}, p_{32})*: elemente med indeksoma $p_{31} \cdot n$ in (vključno) $p_{32} \cdot n$ postavi v obratni vrstni red
- *RandPerm*: naredi naključno permutacijo tabele

[1]Kljub imenu ne gre za permutacijo, tak način generiranja sta uporabila Bentley in McIlroy v članku [6] in ga poimenovala shuffle.

- *Swap*(p_{31}): naredi $p_{31} \cdot n$ permutacij dveh naključnih elementov

Parametra p_{31} in p_{32} sta elementa množice $[0, 1]$. S tem lahko predstavimo indekse od 0 do n. V primeru Reverse ni smiselno imeti večje domene, saj drugi indeksi ne obstajajo. V primeru Swap pa ni smiselno, da naredimo več kot n zamenjav.

3.2.1 Naključna permutacija

Fisher-Yates prerazporeditev

Fisher-Yates prerazporeditev (angl. Fisher–Yates shuffle), včasih tudi Knuthova prerazporeditev (angl. Knuth's shuffle), je algoritem za generiranje naključne permutacije končnega zaporedja. Algoritem sta prva opisala Ronald Fisher in Frank Yates leta 1938 v svoji knjigi *Statistical tables for biological, agricultural and medical research*[1]. Pomemben je zato, ker vrne nepristransko permutacijo, kjer je vsaka permutacija enako verjetna. Za generiranje naključne permutacije števil od 1 do n sledimo naslednjim korakom:

1. Zapišemo števila od 1 do n.

2. Izberemo naključno število k med 1 in (vključno) številom števil, ki jih še nismo obdelali.

3. Štejemo od spodaj navzgor in k-to število, ki še ga nismo obdelali, prečrtamo in ga zapišemo na ločen seznam.

4. Ponavljamo 2. korak, dokler vsa števila niso obdelana (prečrtana).

5. Zaporedje na ločenem seznamu iz 3. koraka je naša naključna permutacija.

Ob predpostavki, da so števila izbrana v 2. koraku resnično naključna in nepristranska, bo nepristranska tudi permutacija.

Sodobna različica algoritma

Sodobno različico algoritma je predstavil Richard Durstenfeld, leta 1964, popularizíral pa jo je Donald E. Knuth v svoji knjigi *The Art of Computer Programming* kot "Algorithm P (Shuffling") [1]. Medtem ko Fisher-Yates algoritem porabi veliko časa za štetje neobdelanih števil v 3. koraku, Durstenfeld predlaga rešitev, kjer obdelan element premaknemo na konec seznama tako, da ga zamenjamo z zadnjim neobdelanim elementom. Ta izboljšava zmanjša časovno zahtevnost iz kvadratne časovne zahtevnosti $O(n^2)$ na linearno $O(n)$.

```
1 fun DurstenfeldShuffle(a, n)
2     for i from n − 1 downto 1
3         j = random(0, i) // 0 ≤ j ≤ i
4         swap(a, i, j)
```

Algoritem 3.2: Durstenfeldov algoritem za generiranje naključne permutacije zaporedja

3.3 Testni primeri

Algoritme smo pognali nad testnimi scenariji *naključno*, *duplikati* in *skoraj urejeno*, ki smo jih generirali z našim generatorjem testnih primerov na naslednji način:

Naključno Testni scenarij vključuje naključne permutacije tabele z elementi od 0 do (vključno) $n-1$. Generiramo ga s parametri (Saw(n, n, 1), Id, RandPerm).

Duplikati Vključuje naključno permutacijo števil, ki jih sestavlja 85 % enakih števil. Najprej zgeneriramo n števil od 0 do $n-1$, jih omejimo z zgornjo mejo $(1-0,85)\cdot n$ ter tabelo naključno permutiramo. Generiramo ga s parametri (Saw(n, n, 1), Plateau(0, $0,15\cdot n$), RandPerm).

Skoraj urejeno Sestavljen je iz tabele z n elementi od 0 do $n-1$, kjer opravimo $10\% \cdot n$ naključnih zamenjav. Generiramo ga s parametri (Saw(n, n, 1), Id, Swap($0, 1 * n$)).

Te scenarije smo generirali za $n = 10000, 20000, ..., 100000, 200000, ..., 1000000$. Za vsak zgoraj opisan testni scenarij in dolžino pa smo zgenerirali 100 naključnih testnih primerov.

3.4 Rezultati

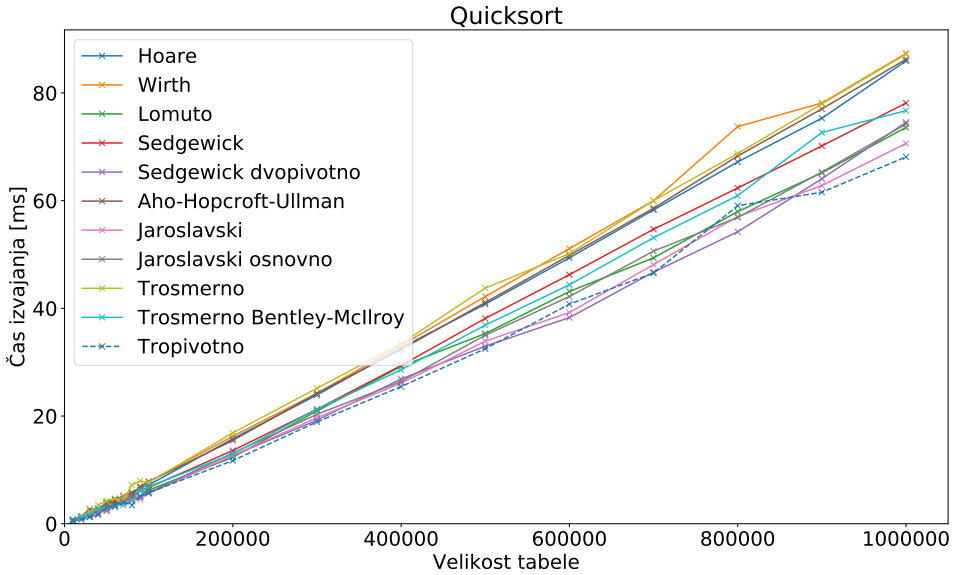
3.4.1 Naključno

Na sliki 3.1 je prikazan čas izvajanja posameznih algoritmov v milisekundah. Najhitrejši algoritem hitrega urejanja je tropivotno hitro urejanje. Nekoliko počasnejše je hitro urejanje Jaroslavskega. Najpočasnejši je bil Wirthov algoritem.

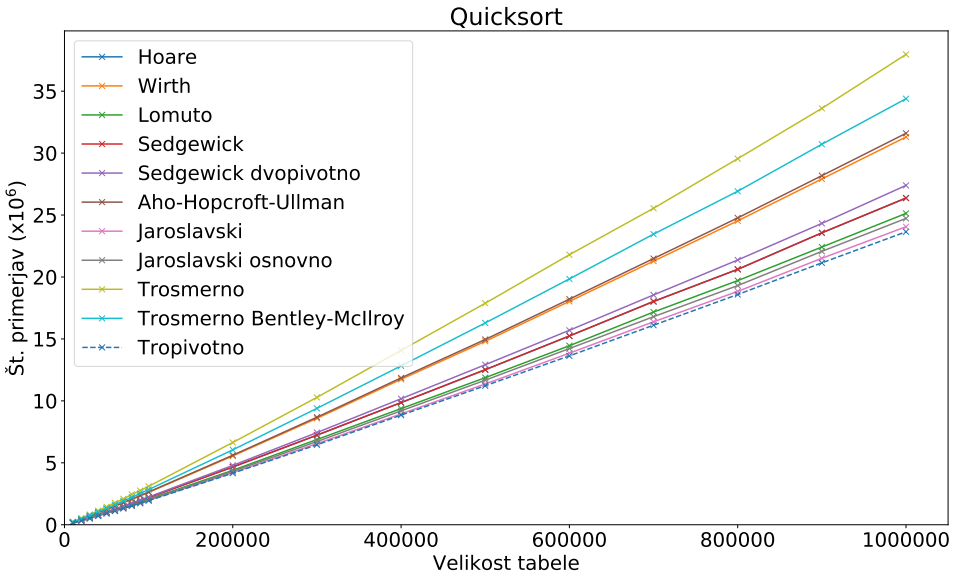
Slika 3.2 prikazuje število primerjav s pivotnim elementom. Najmanj jih ima tropivotno porazdeljevanje, največ pa trosmerno.

Na sliki 3.3 je prikazano število premikov. Najmanj jih ima trosmerno Bentley-McIlroy, tik za njim sledi Wirthovo porazdeljevanje. Največ premikov ima navadno trosmerno porezdeljevanje, ki smo ga z grafa odrezali zato, da je bolje vidna razlika med preostalimi algoritmi.

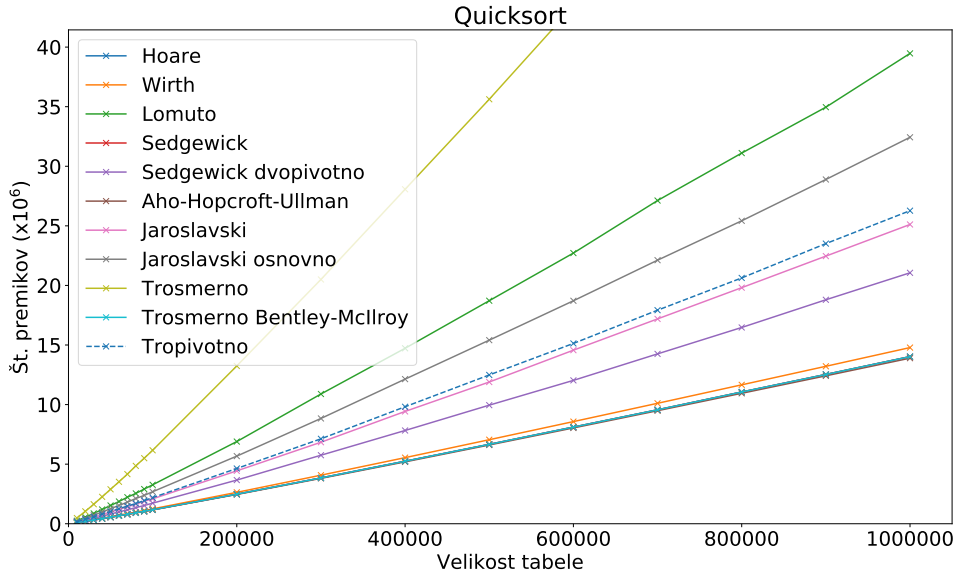
S slike 3.4 je razvidno, da tropivotno porazdeljevanje opravi najmanjše število rekurzivnih klicev.



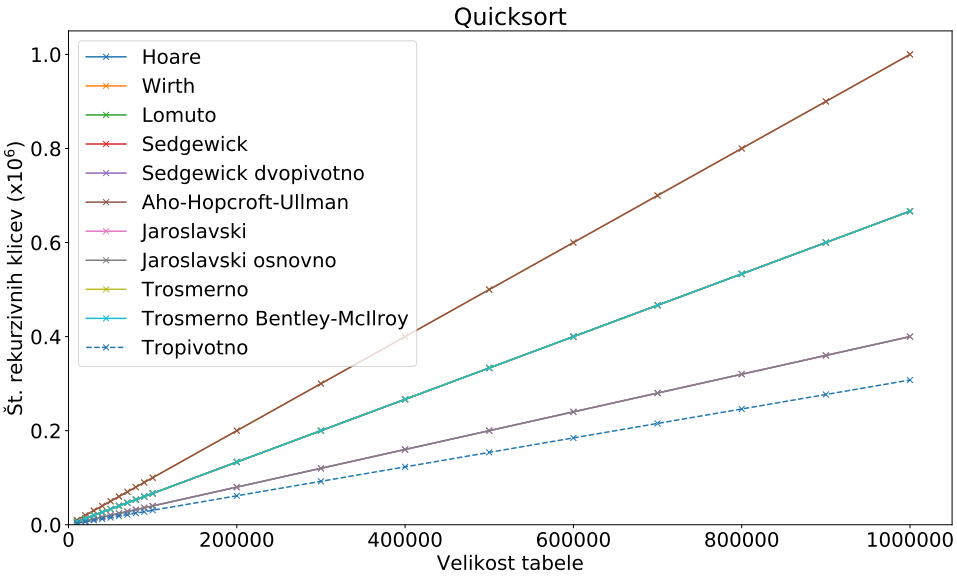
Slika 3.1: Testni scenarij naključno, čas izvajanja



Slika 3.2: Testni scenarij naključno, število primerjav



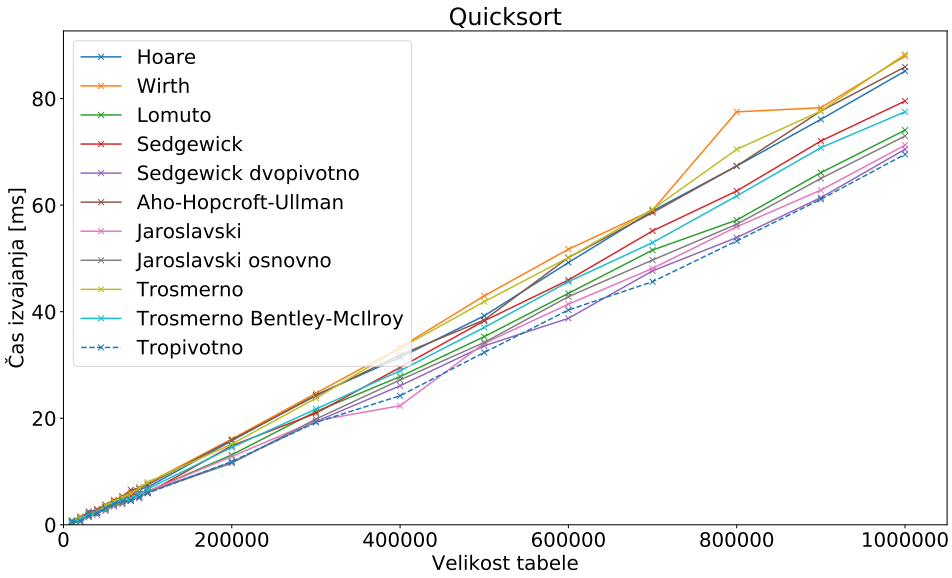
Slika 3.3: Testni scenarij naključno, število premikov



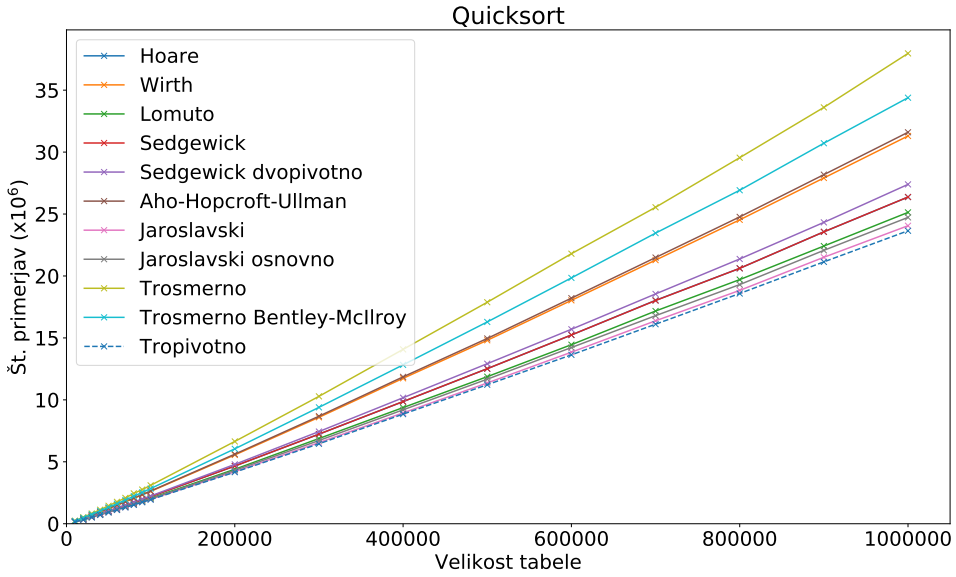
Slika 3.4: Testni scenarij naključno, število rekurzivnih klicev

3.4.2 Duplikati

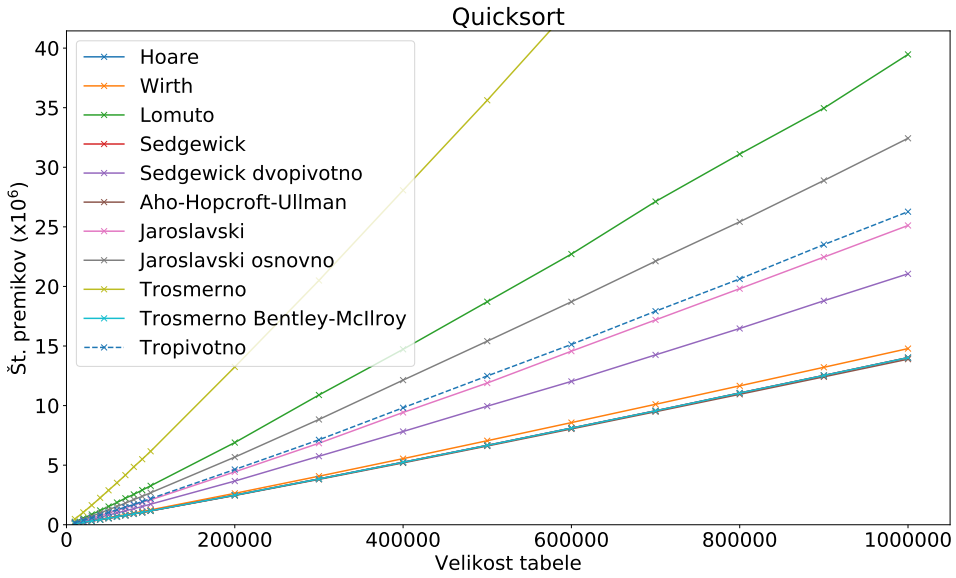
S slike 3.5 je razvidno, da je najhitrejše tropivotno hitro urejanje, ki ima najmanjše število primerjav (slika 3.6), po številu premikov pa je v sredini. Najdaljši čas izvajanja ima trosmerna različica, ki ima tudi največ primerjav. Trosmerno porazdeljevanje ima ponovno največje število premikov (slika 3.7), zato smo ta graf odrezali za boljši pregled ostalih algoritmov. Najmanjše število rekurzivnih klicev imata v tem primeru obe trosmerni hitri urejanji, kar vidimo na sliki 3.8.



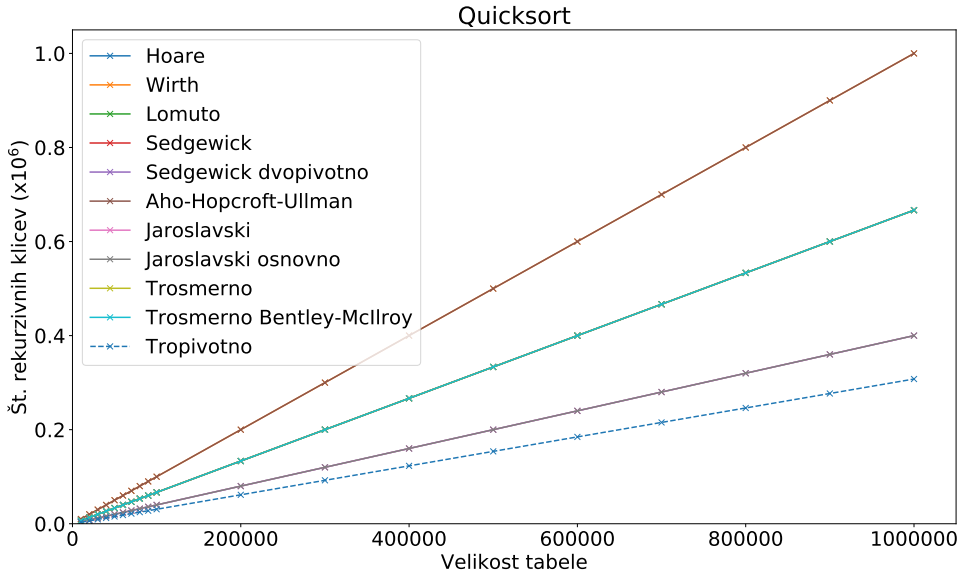
Slika 3.5: Testni scenarij duplikati, čas izvajanja



Slika 3.6: Testni scenarij duplikati, število primerjav



Slika 3.7: Testni scenarij duplikati, število premikov



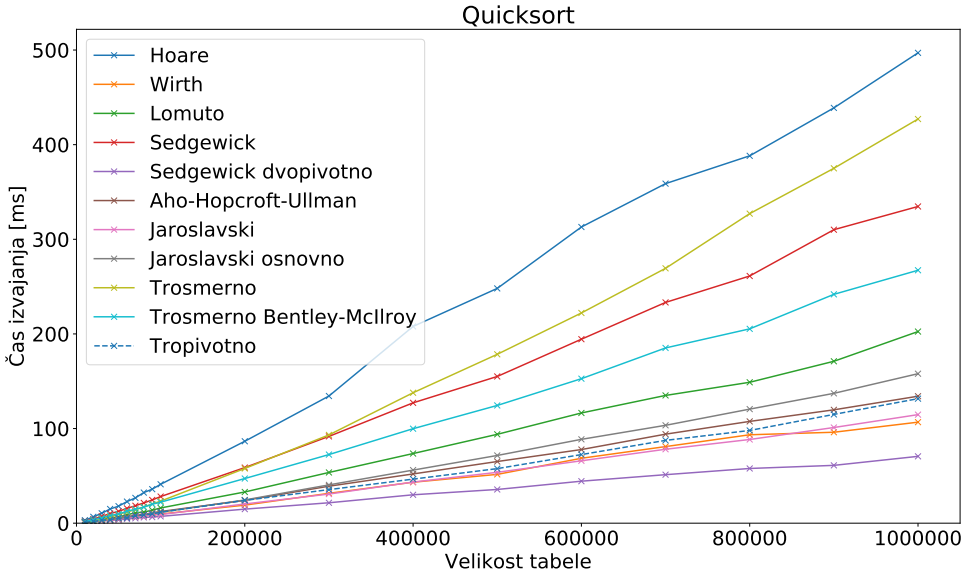
Slika 3.8: Testni scenarij duplikati, število rekurzivnih klicev

3.4.3 Skoraj urejeno

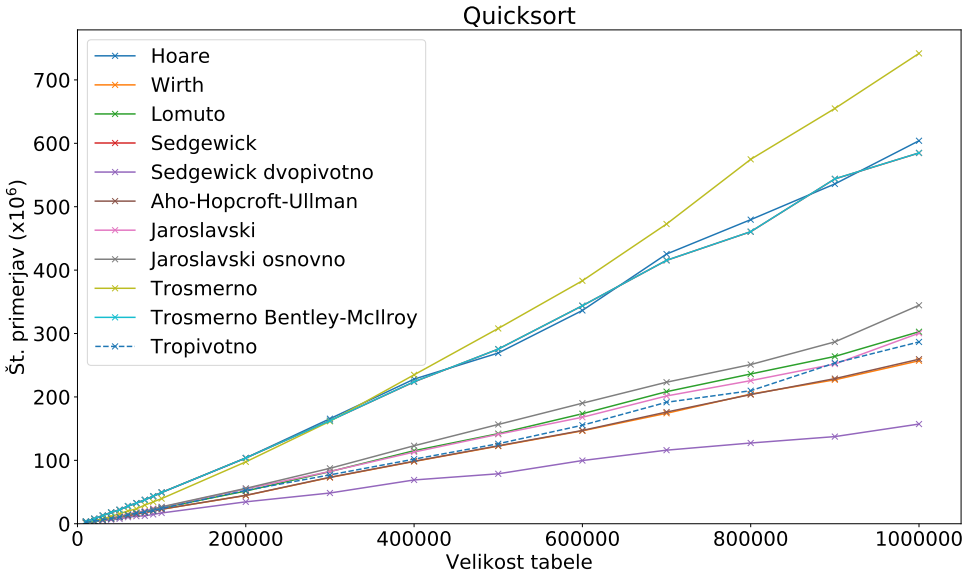
Na sliki 3.9 vidimo, da je v primeru skoraj urejenih podatkov najhitrejše Sedgewickovo dvopivotno hitro urejanje. Razlog tiči v tem, da tropivotno porazdeljevanje naredi veliko več premikov kot Sedgewickovo dvopivotno.

Na sliki 3.10 vidimo, da ima najmanjše število primerjav prav tako Sedgewickovo dvopivotno hitro urejanje, ki mu sledijo Wirthovo, Aho-Hopcroft-Ullman in tropivotno. Na sliki 3.11 opazimo, da ima namanj premikov trosmerno Bentley-McIlroy hitro urejanje.

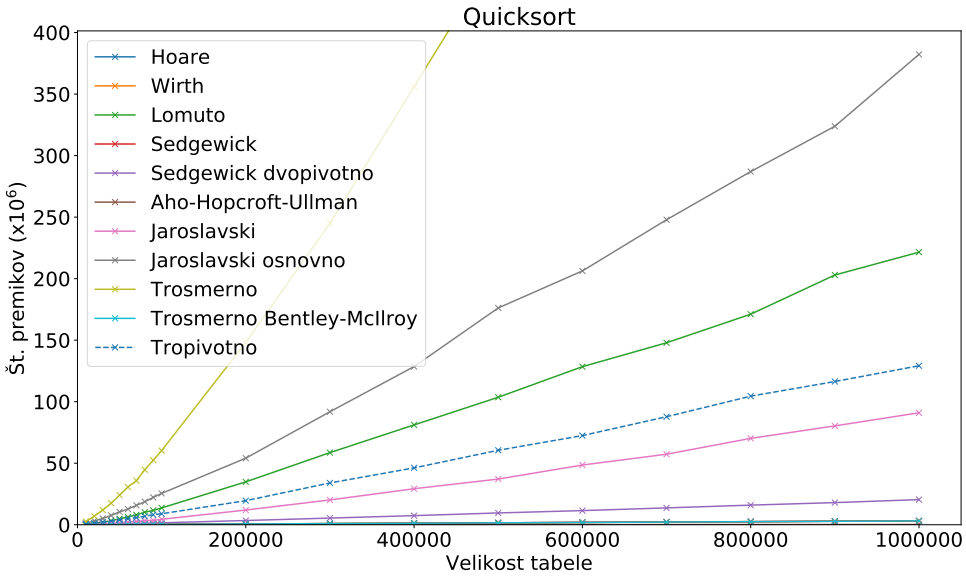
Najmanjše število rekurzivnih klicev pa ima spet tropivotno hitro urejanje, kar vidimo na sliki 3.12.



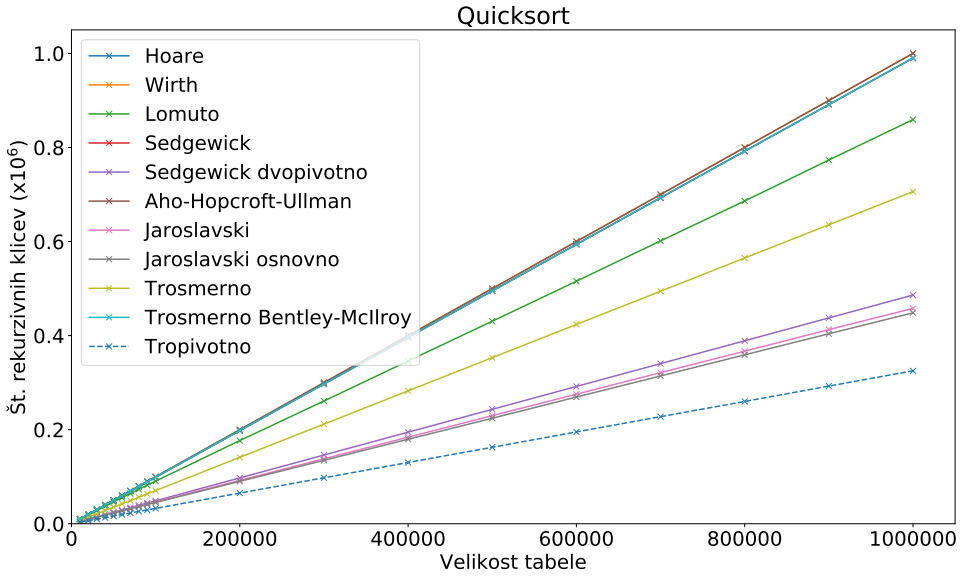
Slika 3.9: Testni scenarij skoraj urejeno, čas izvajanja



Slika 3.10: Testni scenarij skoraj urejeno, število primerjav



Slika 3.11: Testni scenarij skoraj urejeno, število premikov



Slika 3.12: Testni scenarij skoraj urejeno, število rekurzivnih klicev

Poglavje 4

Sklepne ugotovitve

V diplomskem delu smo preučili in implementirali različne načine porazdeljevanja pri hitrem urejanju. Pri implementaciji algoritmov ter generiranju testnih primerov smo uporabili programski jezik C, za poganjanje programov nad različnimi vhodnimi primeri pa bash. Za generiranje testnih primerov smo naredili svoj model, ki pokrije vse testne scenarije, ki sta jih predstavila Bentley in McIlroy, ter doda veliko novih scenarijev.

Večpivotno urejanje je dolgo časa veljalo za počasno, vse dokler Jaroslavski ni našel hitrejšega načina dvopivotnega porazdeljevanja, saj nihče ni raziskoval te smeri. Če ne bi bilo Jaroslavskega, verjetno ne bi bilo tudi tropivotnega hitrega urejanja, saj njegove ideje temeljijo na dvopivotnem porazdeljevanju.

Z analizo časov izvajanja smo ugotovili, da je najhitrejše tropivotno hitro urejanje to, ki so ga odkrili Kushagra, López-Ortiz, Munro in Qiao. Sedgewickovo porazdeljevanje, ki je včasih veljalo za eno izmed najhitrejših, se zdaj nahaja na sredini med vsemi algoritmi. Kljub temu da izvede večje število primerjav in premikov kot ostali algoritmi, je tropivotno hitro urejanje hitrejše. To pripisujemo povečevanju predpomnilnika v sodobni strojni opremi, saj učinki predpomnjenja postajajo vedno bolj izraziti [9]. Iz tega lahko sklepamo, da se čas izvajanja algoritmov spreminja z leti. Stari rezultati na sodobni opremi morda niso več enaki, trenutni rezultati pa se lahko

v prihodnosti spremenijo.

Iskanje najhitrejšega algoritma je tako živ proces, zato je pomembno, da imamo centralni repozitorij algoritmov ter testnih primerov, kamor lahko dodajamo nove algoritme in primerjamo rezultate.

Literatura

[1] Fisher–yates shuffle. Dosegljivo: `https://en.wikipedia.org/wiki/Fisher-Yates_shuffle`. [Dostopano: 27. 2. 2018].

[2] Quicksort. Dosegljivo: `https://en.wikipedia.org/wiki/Quicksort#Lomuto_partition_scheme`. [Dostopano: 5. 11. 2017].

[3] Quicksorts, Github repozitorij s kodo. Dosegljivo: `https://github.com/uroshekic/Quicksorts`. [Dostopano: 7. 3. 2018].

[4] Ullman Aho, Hopcroft. *The Design and Analysis of Computer Algorithms*. 1974.

[5] Jon Bentley. *Programming Pearls*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[6] Jon L. Bentley and M. Douglas McIlroy. Engineering a sort function. *Softw. Pract. Exper.*, 23(11):1249–1265, November 1993.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[8] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[9] Shrinu Kushagra, Alejandro López-Ortiz, J. Ian Munro, and Aurick Qiao. Multi-pivot quicksort: Theory and experiments. In *Proceedings*

of the Meeting on Algorithm Engineering & Expermiments, pages 47–60, Philadelphia, PA, USA, 2014. Society for Industrial and Applied Mathematics.

[10] Jurij Mihelič. Dvo-pivotni quicksort. Dosegljivo: `http://lalg.fri.uni-lj.si/jurij/blog/dvo-pivotni-quicksort/`, 2013. [Dostopano: 15. 12. 2017].

[11] Jurij Mihelič. Tro-smerni quicksort. Dosegljivo: `http://lalg.fri.uni-lj.si/jurij/blog/tro-smerni-quicksort/`, 2013. [Dostopano: 15. 12. 2017].

[12] Robert Sedgewick. Implementing quicksort programs. *Commun. ACM*, 21(10):847–857, October 1978.

[13] Len Shustek. Interview: An interview with C.A.R. Hoare. *Commun. ACM*, 52(3):38–41, March 2009. Interviewee-Hoare, C.A.R.

[14] Sebastian Wild and Markus E. Nebel. Average case analysis of Java 7's dual pivot quicksort. *CoRR*, abs/1310.7409, 2013.

[15] Niklaus Wirth. *Algorithms and Data Structures*. Prentice Hall, 1985.

[16] Vladimir Yaroslavskiy. Dual-pivot quicksort algorithm. Dosegljivo: `http://codeblab.com/wp-content/uploads/2009/09/DualPivotQuicksort.pdf`, 2009. [Dostopano: 10. 11. 2017].