Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jernej Erker

Izvedbe hitrega urejanja za CPE in GPE

diplomsko delo

univerzitetni študij računalništva in informatike

Mentor: doc. dr. Tomaž Dobravec

Ljubljana, 2013

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01926/2013
Datum: 05.04.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **JERNEJ ERKER**

Naslov: **IZVEDBE HITREGA UREJANJA ZA CPE IN GPE**
QUICKSORT ALGORITHMS FOR THE CPU AND GPU

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Algoritem za hitro urejanje podatkov (splošno znan kot QuickSort) praznuje častitljivih 50 let obstoja. Kljub temu velja za enega izmed najhitrejših algoritmov na področju notranjega urejanja podatkov. Različne izvedbe tega algoritma se uporabljajo v standardnih knjižnicah mnogih programskih jezikov.
V diplomskem delu preglejte in opišite izvedbe algoritma za hitro urejanje. Osredotočite se na fazo porazdeljevanja podatkov in predstavite različne možnosti glede števila izhodnih tabel in pivotov ter podajte analize časovnih zahtevnosti. Predstavite tudi paralelno izvedbo algoritma, ki je primerna za izvajanje na GPE. Predstavljene algoritme implementirajte ter jih s pomočjo različnih testov primerjajte med seboj.

Mentor:

doc. dr. Tomaž Dobravec

Fakulteta za računalništvo in informatiko
2
UNIVERZA V LJUBLJANI

Dekan:

prof. dr. Nikolaj Zimic

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Jernej Erker, z vpisno številko **63050057**, sem avtor diplomskega dela z naslovom:

Izvedbe hitrega urejanja za CPE in GPE

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Tomaža Dobravca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 26. junija 2013 Podpis avtorja:

Zahvaljujem se mentorju doc. dr. Tomažu Dobravcu za potrpežljivost in pomoč pri izdelavi diplomske naloge.
Zahvaljujem se tudi moji družini in dekletu za vso izkazano podporo.

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Algoritmi za urejanje **5**

2.1 Osnovni pojmi 5

2.2 Osnovni algoritmi 8

2.3 Izboljšani algoritmi 9

3 Hitro urejanje (Quicksort) **11**

3.1 Hitro urejanje s križanjem kazalcev 13

3.2 Eno-zančno hitro urejanje 21

3.3 Tro-smerno hitro urejanje 25

3.4 Dvo-pivotno hitro urejanje 32

3.5 Bsort 43

4 Hitro urejanje na GPE **49**

4.1 CUDA 49

4.2 Osnovna implementacija 50

4.3 GPU-Quicksort 54

5 Primerjava algoritmov in porazdelitev **57**

5.1 Okolje za testiranje 57

5.2 Primerjava 58

6 Zaključek **65**

Seznam slik **69**

Literatura **71**

Seznam uporabljenih kratic in simbolov

CPE - centralna procesna enota
GPE - grafična procesna enota
CUDA - Compute Unified Device Architecture

Algoritmi

2pivot - dvo-pivotno hitro urejanje
BM - Bentley-McIlroyev algoritem za hitro urejanje
java7 - algoritem za urejanje v Javi 7, ki temelji na dvo-pivotnem urejanju
qpu-qs - GPU-Quicksort, vzporedni algoritem za hitro urejanje na GPE

Povzetek

Algoritem za hitro urejanje, bolj poznan kot *Quicksort*, je na področju urejanja podatkov prisoten že več kot pol stoletja. Kljub temu je še vedno zelo popularen in hiter, različne izvedbe algoritma pa so vsebovane v standardnih knjižnicah mnogih programskih jezikov. Je dokaj enostaven za razumevanje, dobro deluje v večini primerov in ne porabi veliko virov. Osnovan je na metodi *deli in vladaj*, kar pomeni, da večji problem rešimo tako, da ga najprej razdelimo na več manjših podobnih problemov, jih rešimo in iz rešitev sestavimo rešitev originalnega problema. Algoritem je bil skozi leta predmet mnogih poizkusov izboljšav, število različnih implementacij je zato veliko. Glavne razlike med njimi so v načinu porazdeljevanja zaporedij.

Podjetje NVIDIA je leta 2006 predstavilo paralelno arhitekturo CUDA, ki omogoča splošno namensko računanje na grafičnih procesorjih. Prednost izvajanja algoritmov na GPE je uporaba masovnega paralelizma, ki pa ga ni enostavno učinkovito izkoristiti za prav vsak problem.

Cilj tega dela je predstaviti različne načine porazdeljevanja zaporedij in pokazati, kakšne posledice ima pri tem uporaba različnega števila pivotov in razdeljevanja zaporedja na različna števila podzaporedij. Podali bomo analize časovnih zahtevnosti vseh primerov in opisali algoritem za hitro urejanje, pripravljen za izvajanje na GPE. Predstavljene algoritme bomo na koncu med seboj primerjali in rezultate interpretirali.

Ključne besede:

hitro, vzporedno, urejanje, algoritmi, CUDA

Abstract

Quicksort algorithm was discovered in 1960 and is present for more than half a century, nevertheless it is still very popular and fast. Different versions of the algorithm are being used in standard libraries of a number of programming languages. The algorithm is fairly simple to understand, robust in the sense that it is efficient in most cases and it does not need a lot of resources. It is based on the *Divide and Conquer* principle which means that we solve the problem by recursively breaking it down into two or more sub-problems, which we solve and combine the solutions to form a solution to the original problem. There were many attempts of improving the algorithm, so there are a lot of different implementations. The main difference between them is the manner in which they partition the elements of the array.

NVIDIA Corporation presented the parallel computing platform CUDA in 2006. It makes GPUs accessible for computation like CPUs. Main advantage of executing algorithms on the GPU is the use of massive parallelism. Actually making good use of this parallelism is another thing altogether.

The purpose of this thesis is to present different partitioning methods and introduce the consequences of using a different number of pivots or partitioning the array into a different number of sub-arrays. We will review analyses of the time complexities of all the cases and describe the quicksort algorithm, prepared for execution on the GPU. In the end we will compare presented algorithms and state our interpretations of the results.

Key words:

quicksort, parallel, sorting, algorithms, CUDA

Poglavje 1

Uvod

Urejanje je proces, ki zaporedje objektov postavi v neko logično ureditev. Ponavadi ga uporabljamo zato, ker je iskanje elementov po taki ureditvi veliko lažje in hitrejše. Kot primer lahko vzamemo mesečni izpisek opravljenih bančnih transakcij, ki je urejen po datumu. V takšno stanje ga je verjetno spravil eden od mnogih algoritmov za urejanje, ki so danes prisotni praktično v vsakem računalniškem sistemu. Urejanje je pogosto prvi korak organizacije podatkov in igra veliko vlogo tako v vsakdanjem življenju kot tudi v npr. astrofiziki, lingvistiki, genetiki, napovedovanju vremena in številnih ostalih področjih [1].

Naloge urejanja se razlikujejo predvsem po *definiciji relacije urejenosti* ter po *velikosti zbirke podatkov*, posamezni algoritmi pa po svoji *zapletenosti*.

Relacija urejenosti je lahko definirana na različne načine, v večini pa jo zapišemo s pomočjo običajnih relacij $<$, $=$ in $>$, ki morajo biti definirane za številske tipe in tudi za znakovna zaporedja. Delu podatka, na podlagi katerega izvajamo primerjanje, ponavadi pravimo ključ. Cilj urejanja je takšna ureditev zaporedja, kjer je ključ vsakega podatka manjši (ali enak) od ključev elementov z višjimi indeksi in hkrati večji (ali enak) od ključev elementov z nižjimi indeksi [2].

Glede na število podatkov razlikujemo *notranje* in *zunanje* urejanje. V prvem primeru je možno hraniti vse podatke v notranjem pomnilniku ra-

čunalnika, v drugem pa je potrebno uporabiti zunanje pomnilnike, recimo datoteke.

Algoritme za posamezne naloge urejanja lahko razvrstimo v dve skupini. *Navadni algoritmi* za urejanje so po svoji zasnovi preprosti, vendar porabijo razmeroma veliko časa. *Izboljšani algoritmi* so bolj zapleteni, vendar prihranijo pri času urejanja. Glavno merilo za učinkovitost algoritma je lahko število osnovnih operacij (primerjav, zamenjav), število rekurzivnih klicev, lahko pa tudi npr. število dostopov do zunanjega pomnilnika, ki jih algoritem opravi med urejanjem različnih zaporedij.

Eden izmed izboljšanih algoritmov, ki si ga bomo ogledali bolj podrobno, je *algoritem za hitro urejanje oz. urejanje s porazdelitvami* (ang. *Quicksort*). Leta 1960 ga je odkril Tony Hoare [7], sodi pa med bolj popularne algoritme na sploh. Razlogov za to je več - ni ga težko implementirati, je robusten, kar pomeni, da dobro deluje v večini različnih primerov, in pa, porabi manj virov (časa in prostora) kot drugi algoritmi za urejanje. Temelji na metodi snovanja algoritmov *deli in vladaj* (ang. *Divide and Conquer*). Večji problem rešimo tako, da ga razdelimo na več manjših podproblemov, te rekurzivno rešimo, nato pa iz rešitev sestavimo rešitev originalnega problema. Klasična različica algoritma za hitro urejanje si najprej izbere poseben element, imenovan *pivot*. Vhodno zaporedje nato razdeli na dve manjši zaporedji tako, da pristanejo v prvem, levem zaporedju vsi elementi, ki so manjši od pivota, v drugem, desnem zaporedju pa vsi večji. Elementi, ki so enaki pivotu, so lahko v poljubnem delu. Obe podzaporedji nato rekurzivno uredi in sestavi originalno rešitev, kar je zaradi načina porazdelitve trivialno.

Implementacijo algoritma za hitro urejanje najdemo skoraj v vseh standardnih knjižnicah programskih jezikov, saj je v primerjavi z drugimi algoritmi v praksi zelo hiter. Zanj obstaja veliko število različnih implementacij, ki prej opisano osnovno rešitev še dodelajo in na mnogih področjih izboljšajo. Med ta področja sodi izbira pivotnega elementa, ki je lahko enostavno prvi, zadnji ali sredinski element tabele, lahko pa je tudi naključen ali mediana več elementov. Izboljšan algoritem lahko pri manjših podzaporedjih zamenja

način urejanja npr. na urejanje z vstavljanjem, ki je pri kratkih zaporedjih bolj učinkovito in hitrejše. Glavne razlike in izboljšave pa se pokažejo pri načinu porazdeljevanja. Zanimajo nas predvsem porazdelitve, ki se izvedejo neposredno na vhodnem zaporedju, brez zahteve po dodatnem prostoru (ang. *in-place* porazdelitve). Klasična porazdelitev uporablja en pivot in zaporedje razdeli na dva dela, seveda pa je pivotov ali delov zaporedij lahko tudi več.

Zaradi snovanja algoritma na metodi "deli in vladaj"lahko hitro urejanje prevedemo v vzporedno obliko in ga tako pripravimo na izvajanje na več procesorjih hkrati, paralelno. Proces porazdeljevanja je sicer težje paralelizirati, posamezna podzaporedja pa so med seboj neodvisna in se lahko urejajo istočasno.

Današnje grafične kartice posedujejo zelo zmogljive več-jedrne procesorje, katerih računska moč pa je večinoma neizkoriščena, z izjemo video iger. Glede na to, da so ti procesorji specializirani za računsko zahtevne paralelne operacije, jih lahko uporabimo za reševanje problemov, za katere se da implementirati učinkovite vzporedne algoritme. Podjetje NVIDIA je ustvarilo arhitekturo CUDA, ki razvijalcem omogoča prav to, da izkoristijo moč grafičnih procesorjev za reševanje splošnejših problemov, ne več samo takih, povezanih z grafiko.

V tem delu bomo najprej na hitro pregledali splošne algoritme za urejanje, nato pa se osredotočili na različne implementacije algoritma za hitro urejanje, jih podrobneje obdelali, opisali in analizirali njihove časovne zahtevnosti. Predstavili bomo arhitekturo CUDA in izvedbe vzporednih algoritmov, prilagojene za izvajanje na GPE. Na koncu bomo naredili primerjavo med klasičnimi zaporednimi implementacijami algoritmov v jeziku C in vzporednimi implementacijami algoritmov na arhitekturi CUDA.

Poglavje 2

Algoritmi za urejanje

2.1 Osnovni pojmi

2.1.1 Poraba časa

Preden se lotimo samih algoritmov, si poglejmo, kako bomo ocenjevali njihovo časovno porabo. Celoten sestavek je povzet po [2].

Pri ocenah porabe virov bomo izhajali iz določenih osnovnih lastnosti računalnika, na katerem se algoritem izvaja. Recimo, da sta S_1 in S_2 dve zaporedji stavkov v programu in da $T(S_i)$ pri $1 \leq i \leq 2$ označuje čas, ki ga program porabi na zaporedju S_i. Potem je osnovna lastnost, iz katere izhajamo:

$$T(S_1; S_2) = T(S_1) + T(S_2). \tag{2.1}$$

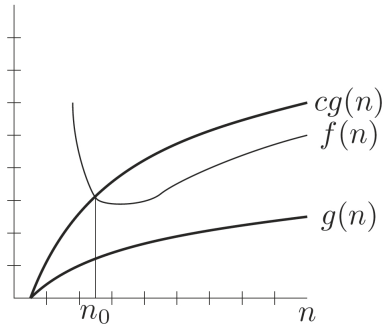
Zavedati se moramo, da je to poenostavljena enačba, ki se omejuje na računalniški model preprostega *zaporednega procesorja*, ki ukaze izvaja enega za drugim in kjer se vsak program konča, preden se začne naslednji. Danes seveda obstajajo več-procesorski računalniški sistemi, za katere relacija (2.1) ne velja, vendar pa še vedno predstavlja osnovo za vsako analizo porabe časa [2].

Asimptotična rast funkcij

Pogosto želimo izpostaviti, da je neka funkcija $f(n)$ po *velikostnem redu asimptotično omejena (navzgor, navzdol) ali pa enaka g(n).* Bolj natančno lahko asimptotično *zgornjo mejo* zapišemo kot množico funkcij $\mathcal{O}(g(n))$:

$$\mathcal{O}(g(n)) = \{f(n) \mid \exists c, n_0 > 0 \ \forall n \geq n_0 \ [f(n) \leq cg(n)]\}$$

Ta zapis je enakovreden trditvi $f(n) \in \mathcal{O}(g(n))$, vendar se ponavadi namesto tega napiše kar $f(n) = \mathcal{O}(g(n))$, kar pa zna včasih biti nekoliko zavajajoče. Za boljše razumevanje si lahko pogledamo sliko 2.1.



Slika 2.1: Asimptotična zgornja meja

Podobno lahko definiramo asimptotično *spodnjo mejo* kot množico funkcij $\Omega(g(n))$:

$$\Omega(g(n)) = \{f(n) \mid \exists c, n_0 > 0 \ \forall n \geq n_0 \ [f(n) \geq cg(n)]\}$$

V primeru, ko velja hkrati $f(n) = \mathcal{O}(g(n))$ in $f(n) = \Omega(g(n))$, pa rečemo, da je $f(n)$ po velikostnem redu enaka $g(n)$ in zapišemo $f(n) = \Theta(g(n))$.

Čas

Kot pravi lastnost (2.1), pri ocenjevanju časa, ki ga program porabi, seštejemo čase za posamezne dele programa, npr. posamezne stavke. Porabo časa posameznega stavka ponavadi poenostavimo in privzamemo, da vsi stavki porabijo enak čas, ali pa, da nekateri stavki trajajo 1 enoto, drugi pa 0, tako da slednjih sploh ne upoštevamo.

Za primer lahko vzamemo zanko *FOR i = 1 TO n DO* S_i. Takšne in drugačne zanke so med najpomembnejšimi elementi programov. Pri ocenjevanju časa ima torej pomembno vlogo izračun vrednosti vrst $\sum_{i=1}^{n} S_i$. Za npr. zelo značilno aritmetično vrsto:

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n$$

lahko ugotovimo oz. dokažemo z indukcijo po n, da velja

$$\sum_{i=1}^{n} i = \tfrac{1}{2}n(n+1) = \Theta(n^2).$$

Rekurenčne relacije

Veliko algoritmov ima rekurzivno definicijo. Za ocenjevanje časa takih algoritmov uporabljamo rekurenčne relacije.

Recimo, da imamo nek problem, pri katerem bomo velikost vhodnih podatkov označili s parametrom n. Algoritem vhodne podatke najprej transformira v a sorodnih podproblemov velikosti $\frac{n}{c}$, nato pa reši vseh a podproblemov in dobljene rešitve združi v rešitev originalnega problema. Če je za postopek razdelitve na podprobleme in za združevanje rešitev potreben čas velikostnega razreda n^r (r je neka konstanta), potem za celoten čas, ki ga algoritem porabi, dobimo relacijo

$$T(n) = \begin{cases} b, & n = 1 \\ aT(\frac{n}{c}) + bn^r, & n > 1 \end{cases}$$

kjer je b večji izmed časov, ki jih algoritem porabi ali za primer $n = 1$ ali pa za razcep problema velikosti $n > 1$ na podprobleme in za kasnejšo združitev rezultatov. Ocena za $T(n)$ je tako:

$$T(n) = \begin{cases} \Theta(n^r), & a < c^r \\ \Theta(n^r log(n)), & a = c^r \\ \Theta(n^{\log_c a}), & a > c^r \end{cases}$$

kjer so a, b, c in r nenegativne konstante in je $c > 0$ in $a \leq c$.

2.2 Osnovni algoritmi

Navedeni osnovni algoritmi so enostavni za razumevanje, njihova najslabša in *tudi povprečna* časovna zahtevnost je reda $\mathcal{O}(n^2)$.

Navadno vstavljanje

Navadno vstavljanje (ang. *insertion sort*) je algoritem, ki ga lahko uporabimo za urejanje igralnih kart - vsako karto posebej vstavimo na primerno mesto med že urejene karte. Algoritem torej hodi po seznamu in pregleda vsak element ter ga postavi na pravilno mesto, seveda pa si mora tam prej pripraviti prostor tako, da večje elemente prestavi v desno. Ko pregleda celoten seznam, je le-ta popolnoma urejen. Odličen je torej za že urejena ali pa vsaj delno urejena zaporedja, saj v najboljšem primeru doseže linearen čas, ker mu ni treba opraviti nobene zamenjave.

Navadno izbiranje

Algoritem za navadno izbiranje (ang. *selection sort*) je eden od najpreprostejših algoritmov za urejanje. Najprej v zaporedju poišče najmanjši element in ga zamenja s prvim elementom. Potem poišče drugi najmanjši element in ga zamenja z drugim elementom ter nadaljuje po tem principu, dokler celotno zaporedje ni urejeno. Iskanje najmanjšega elementa v seznamu ne da nobene koristne informacije za pregled seznama v naslednjem koraku. Zato ta algoritem porabi približno enako časa za urejanje že urejenih zaporedij, zaporedij s samimi enakimi ključi ali naključno urejenih zaporedij. Po drugi strani pa opravi točno n zamenjav, kar lahko pride prav nekje, kjer je dostop do zaporedja oz. pisanje vanj draga operacija.

Navadne zamenjave

Pri navadnih zamenjavah oz. mehurčnem urejanju (ang. *bubble sort*) algoritem začne na začetku zaporedja in primerja prva dva elementa. Če je prvi večji od drugega, ju zamenja, drugače gre naprej. Tako nadaljuje do konca zaporedja, vendar pa to še ni urejeno v celoti. Celotno zanko ponavlja toliko

časa, dokler enkrat ne preide celega zaporedja in mu vmes ni potrebno opraviti nobene zamenjave. Edina prednost algoritma je v urejanju delno ali že v celoti urejenih zaporedij, pa še tam se odreže slabše kot navadno vstavljanje, tako da se v praksi le redko uporablja.

2.3 Izboljšani algoritmi

Shellovo urejanje (izboljšano vstavljanje)
Shellov algoritem (ang. *Shell sort*) za urejanje ima danes bolj kot ne zgodovinski pomen, ker predstavlja prvi algoritem za urejanje, ki je porabil manj časa kot $\mathcal{O}(n^2)$ [2]. Algoritem deluje tako, da vhodno zaporedje razdeli na več porazdeljenih podzaporedij, nato pa vsakega uredi s pomočjo algoritma za navadno vstavljanje. Potem postopek ponovi, vendar na manjšem številu daljših podzaporedij in tako nadaljuje, dokler ne pride do urejanja celotnega zaporedja. Algoritem izkorišča dve prednosti navadnega vstavljanja, in sicer hitro urejanje kratkih zaporedij in hitro urejanje delno urejenih zaporedij. Sprva so zaporedja krajša, pozneje pa vedno daljša, vendar delno že urejena. Ob skrbno izbranem številu podzaporedij v vsakem koraku je časovna zahtevnost algoritma $\mathcal{O}(n^{1.2})$ [2].

Urejanje s kopico (izboljšano izbiranje)
Urejanje s kopico (ang. *heap sort*) izkorišča dejstvo, da se pri navadnem izbiranju že opravljeno delo venomer ponavlja. Isti postopek iskanja najmanjšega elementa se ponavlja iz iteracije v iteracijo, ne glede na to, da bi se dalo informacije iz enega obhoda koristno uporabiti tudi kasneje. Točno to počne urejanje s kopico - ohranja informacijo iz predhodnih iskanj najmanjših elementov v posebni podatkovni strukturi, ki ji pravimo kopica. Kopica *je urejeno, levo poravnano, uravnoteženo dvojiško drevo*, njegova vozlišča pa hranijo informacijo o ključih. Drevo je urejeno, kar pomeni, da je ključ vozlišča v večji (oz. manjši) od ključev vseh vozlišč w, ki pripadajo poddrevesu s korenom v. Ko so podatki shranjeni v kopici, iskanje najmanjšega elementa

traja samo še $\mathcal{O}(\log n)$, kar je precej hitreje kot iskanje po celotnem seznamu, ki vzame $\mathcal{O}(n)$ časa.

Poglavje 3

Hitro urejanje (Quicksort)

Algoritem za hitro urejanje je eden izmed najbolj učinkovitih algoritmov za notranje urejanje. Njegove prednosti so v tem, da urejanje poteka neposredno na vhodnem zaporedju, tako da porabi le malo dodatnega pomnilnika, ter da je njegova časovna zahtevnost v povprečju sorazmerna z $n \log n$, kjer je n dolžina zaporedja. Izmed prej omenjenih algoritmov ti dve lastnosti poseduje samo urejanje s kopico, čigar časovna zahtevnost je prav tako reda $n \log n$, vseeno pa je v praksi počasnejše. Pri implementaciji algoritma za hitro urejanje je potrebno biti izredno previden, saj je zelo občutljiv in se kaj hitro lahko zgodi, da ima še tako majhna sprememba katastrofalne posledice [1].

Od nastanka algoritma leta 1960 je bil le-ta predmet mnogih raziskav in modifikacij, ki so poizkušale izboljšati predvsem njegovo $\mathcal{O}(n^2)$ časovno zahtevnost iz najslabšega primera. Izboljšave lahko v splošnem razdelimo na tri področja:

- izbira pivotnega elementa,
- vpeljava drugega algoritma za urejanje krajših zaporedij,
- različni načini porazdeljevanja.

Še eno področje izboljšav - prilagodljivo urejanje zaporedij, ki so že (delno) urejena - je predlagal Roger L. Wainwright [15].

Izbira pivota je lahko čisto enostavna, lahko npr. izberemo skrajno levi, sredinski ali desni element zaporedja. Vendar pa hitro vidimo, da pride do

težav, če recimo v že urejenem zaporedju za pivot izberemo skrajno levi element - takrat se algoritem za hitro urejanje izrodi v navadno urejanje z izbiranjem. Da se najslabšemu primeru v kar največji meri izognemo, lahko za pivot izberemo naključni element zaporedja [8]. Vendar operacija računanja naključnega elementa ni zastonj, poleg tega pa program upočasnimo za vsa zaporedja, samo zato, da se izognemo nekaj anomalijam [9]. Pivot lahko izberemo tudi kot mediano več elementov, npr. levega, sredinskega in desnega [10], ali pa kot mediano celotnega zaporedja, vendar je v praksi za to potrebnega preveč dela.

Osnovni algoritem se izkaže za neučinkovitega pri urejanju kratkih zaporedij. To ima še posebno negativne posledice, saj mora program zaradi svoje rekurzivne narave vedno urediti veliko število kratkih podzaporedij. Prav zato je Hoare predlagal, da se na zaporedju, krajšemu od neke meje m, uporabi bolj učinkovita metoda urejanja [8]. Običajno je to *navadno urejanje z vstavljanjem*. V [9] pa je pokazano, da obstaja še boljši način, in sicer, da kratka podzaporedja med porazdeljevanjem preprosto ignoriramo. Na koncu dobimo seznam, kjer so na pravih mestih elementi, ki so bili uporabljeni kot pivoti, vmes pa so neurejena podzaporedja dolžine m ali manj. Potrebno je samo še pognati algoritem navadnega urejanja z vstavljanjem nad celotnim zaporedjem in le-to bo precej učinkovito urejeno. Analize pokažejo, da ta končni korak porabi le malo več časa, kot bi ga porabila urejanja vseh kratkih zaporedij, vendar pa s tem v povprečju odpravimo $\frac{3}{4}$ vseh zapisovanj na sklad, saj vsa zaporedja z m ali manj elementi med porazdeljevanjem ignoriramo in za njih alternativne metode urejanja sploh ne kličemo. Najboljša vrednost za m naj bi bila med 6 in 15 [9].

Glavne razlike med algoritmi za hitro urejanje se pokažejo pri načinu porazdeljevanja. Pri klasičnem načinu uporabljamo en sam pivot in zaporedje porazdelimo na dva dela - v enega gredo elementi, manjši od pivota, v drugega pa večji. Lahko ga razdelimo tudi na tri dele, kar nam pride prav v primeru, da je v zaporedju zelo malo različnih elementov. Algoritmu, ki v tretji del porazdeli pivotu enake elemente, pravimo tro-smerno hitro urejanje.

Namesto enega pivota lahko uporabimo tudi dva in zaporedje spet razdelimo na tri dele. V prvem so elementi, manjši od prvega pivota, v drugem delu so elementi, večji od drugega pivota, v tretjem delu pa ostanejo vmesni elementi. Tega načina porazdeljevanja se poslužuje dvo-pivotno hitro urejanje [12].

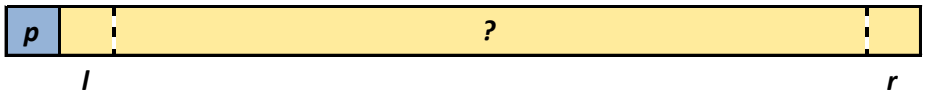
V zadnji kategoriji imamo opravka z algoritmi, ki poizkušajo izboljšati učinkovitost hitrega urejanja v najslabšem možnem primeru, torej primeru, ko je zaporedje že v celoti ali vsaj delno urejeno. Dva zelo podobna primera takih algoritmov sta *Bsort* in *Qsorte* [14, 15].

3.1 Hitro urejanje s križanjem kazalcev

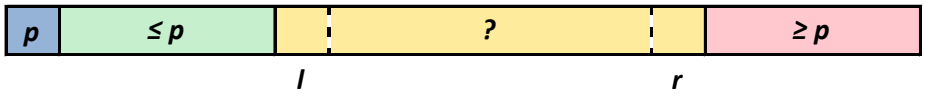
Najprej si poglejmo prvotno verzijo algoritma, ki jo je razvil Hoare.

Porazdelitev

Na začetku se neobdelani del razteza čez celotno tabelo, za pivotni element pa se porabi prvi element zaporedja. Stvari torej izgledajo takole:



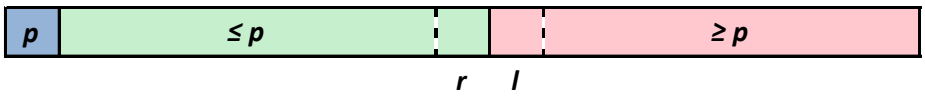
Med porazdeljevanjem je tabela sestavljena iz štirih delov: pivota, *spodnjega dela*, *neobdelanega dela* in *zgornjega dela*. Spodnji del vsebuje elemente, ki so manjši od pivota, zgornji del pa tiste, ki so od pivota večji. Levi indeks l postavlja mejo med spodnjim in neobdelanim delom, desni indeks r pa med neobdelanim in zgornjim delom.



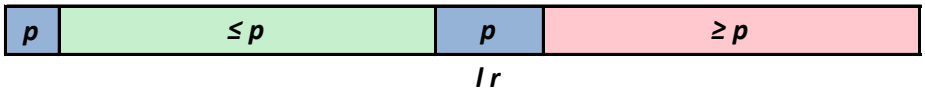
V vsakem koraku porazdeljevanja se levi kazalec pomika v desno toliko časa, dokler ne naleti na element, ki je od pivota večji oz. mu je enak, vse manjše pa preskoči. Podobno je z desnim kazalcem, ki se pomika v levo smer,

dokler ne pride do elementa, ki je od pivota manjši oz. mu je enak. Sedaj je torej l-ti element večji ali enak pivotu, r-ti element pa manjši ali enak pivotu, kar pomeni, da sta ravno v napačnih delih zaporedja, zato ju zamenjamo. Porazdeljevanje se konča, ko se leva in desna meja prekrižata, iz česar izvira tudi ime te tehnike.

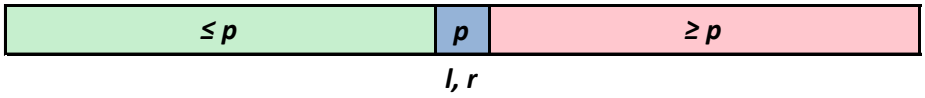
Na koncu sta možnosti dve. Meji se lahko dejansko prekrižata, torej $l > r$:



Lahko pa se kazalca ustavita na istem elementu, kar pomeni, da je leta v tem primeru enak pivotu. Seveda, če je leva meja prišla do elementa, večjega ali enakega pivotu, in če je desna meja prišla do elementa, manjšega ali enakega pivotu, ter če sta se obe meji ustavili na istem elementu, potem ta element mora biti enak pivotu.

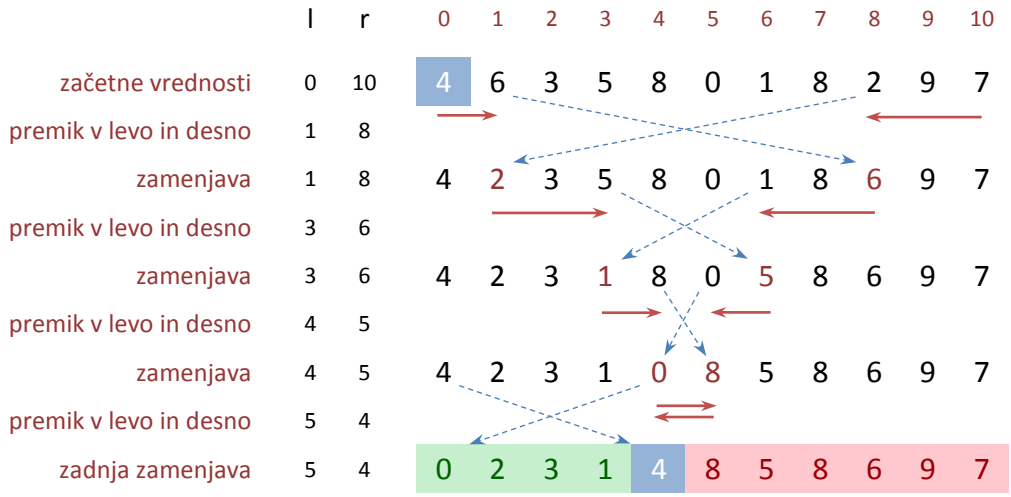


Ob zaključku porazdeljevanja je potrebno prestaviti še pivot iz skrajno levega položaja na ustrezno mesto. Zamenjamo ga z r-tim elementom in dobimo končno sliko:



Proces porazdeljevanja je s tem končan, vhodno zaporedje pa seveda še ni do konca urejeno, je le razdeljeno na tri dele, in sicer na spodnji del, pivot in zgornji del. Za popolno ureditev je potrebno še rekurzivno urediti spodnji in zgornji del.

Preprost primer enega koraka porazdeljevanja (vsebina tabele pred in po vsaki zamenjavi) si lahko pogledamo na sliki 3.1.



Slika 3.1: Primer porazdeljevanja s križanjem kazalcev

Algoritem

Celoten algoritem hitrega urejanja s križanjem kazalcev prikazuje izvorna koda 3.1.

Urejanje se začne s klicem funkcije `quicksort`, ki zaporedje najprej porazdeli, nato pa rekurzivno uredi oba dela. Glavni del urejanja se izvede v funkciji `partition`. Zunanja zanka se izvaja neprestano do preklica, torej dokler se leva in desna meja (l in r) ne prekrižata. Pogoj za križanje se preverja v vrstici 13. Notranji zanki premikata meji vsako v svojo stran, dokler ne naletita na elementa, ki se nahajata v napačnem delu. Takšna dva elementa se zamenjata in proces se ponovi. V bistvu zanki samo povečujeta števec in primerjata element tabele z neko fiksno vrednostjo. Prav to naredi algoritem tako hiter, težko si je namreč zamisliti bolj preprosto zanko. V prvi notranji zanki se še dodatno popazi na to, da leva meja ne preide izven tabele. Podobno preverjanje ni potrebno v drugi notranji zanki, saj je skrajno levi element tabele pivot, ki deluje kot čuvaj [17].

Izvorna koda 3.1: Hitro urejanje s križanjem kazalcev

```
1  void quicksort(Item a[], int left, int right) {
2    if (right <= left) return;
3    int p = partition(a, left, right);
4    quicksort(a, left, p-1);
5    quicksort(a, p+1, right);
6  }
7
8  int partition(Item a[], int left, int right) {
9    int l = left, r = right + 1; Item p = a[left];
10   while(1) {
11     while (less(a[++l], p)) if(l == right) break;
12     while (less(p, a[--r])) ;
13     if (l >= r) break;
14     exch(a, l, r);
15   }
16   exch(a, r, left);
17   return r;
18 }
```

Skozi leta se je razvilo veliko število različnih implementacij porazdeljevanja s križanjem kazalcev. Na primer, za pivot lahko vzamemo tudi skrajno desni element tabele. Seveda bi bilo v tem primeru potrebno zagotoviti, da desna meja ne bi prešla izven tabele, tako da bi preverjanje iz prve notranje zanke prestavili v drugo. Lahko pa pred začetkom urejanja poiščemo najmanjši (največji) element tabele ter ga postavimo na skrajno levo (desno) - tako se preverjanju pogoja popolnoma izognemo.

Analiza

Algoritem v najboljšem primeru razdeli zaporedje točno na pol. Število primerjav v takem primeru zadosti splošni rekurenčni enačbi pristopa "deli in vladaj", tj. $C_n = 2C_{n/2} + n$. Izraz $2C_{n/2}$ pokrije ceno urejanja obeh podzaporedij, n pa je cena porazdeljevanja originalnega zaporedja, kjer gremo čez vseh n elementov in jih po potrebi premaknemo v drug del zaporedja [1].

Naj bo C_n število primerjav, potrebnih za urejanje zaporedja dolžine n. Imamo torej $C_0 = C_1 = 0$, za $n > 0$ pa lahko zapišemo rekurenčno relacijo:

$$C_n = C_{\lfloor n/2 \rfloor} + C_{\lceil n/2 \rceil} + n$$

Prvi izraz na desni je število primerjav za urejanje spodnjega dela zaporedja, drugi izraz je število primerjav za urejanje zgornjega dela zaporedja, tretji izraz pa je število primerjav med porazdeljevanjem. Do točne rešitve enačbe pridemo, ko je n potenca števila 2, npr. $n = 2^k$. Ker je $\lfloor n/2 \rfloor = \lceil n/2 \rceil = 2^{k-1}$, lahko enačbo preoblikujemo v

$$C_{2^k} = 2C_{2^{k-1}} + 2^k$$

Obe strani delimo z 2^k

$$C_{2^k}/2^k = C_{2^{k-1}}/2^{k-1} + 1$$

Če uporabimo isto enačbo nad prvim izrazom na desni strani, dobimo

$$C_{2^k}/2^k = C_{2^{k-2}}/2^{k-2} + 1 + 1$$

Prejšnji korak ponovimo še $k - 1$ krat

$$C_{2^k}/2^k = C_{2^0}/2^0 + k$$

Obe strani pomnožimo z 2^k in dobimo rešitev

$$C_n = C_{2^k} = k2^k = n \log n$$

Kljub temu, da stvari niso vedno tako lepe, je vseeno res, da porazdeljevanje *v povprečju* razdeli zaporedje na pol. Poglejmo si torej, kako se algoritem obnaša v povprečju.

Analiza povprečnega primera je povzeta po [5]. Lotili se je bomo tako, da bomo najprej pogledali, kaj se dogaja v prvem koraku porazdeljevanja. Po križanju kazalcev imamo zaporedje razdeljeno na spodnji in zgornji del. Na tem mestu je potrebno izpostaviti, da obe podzaporedji ostaneta naključno

urejeni, če je originalno zaporedje bilo v naključnem vrstnem redu, saj relativna urejenost elementov nima nobenega vpliva na porazdeljevanje. To pa pomeni, da lahko na podlagi prvega koraka porazdeljevanja prispevke naknadnih porazdeljevanj določimo kar z indukcijo po n.

Zavoljo lažje analize bomo privzeli, da so vsi elementi med seboj različni oz. da so preprosto kar števila $\{1, 2, \ldots, n\}$ v nekem vrstnem redu. Četudi bi zaporedje vsebovalo enake elemente, to ne bi imelo posebno velikega vpliva na porabo časa, le natančna analiza bi bila veliko bolj zapletena [1].

Sedaj pa recimo, da je s vrednost prvega elementa K_1, ter da je točno t elementov izmed $K_1, \ldots, K_s$ večjih od s. Če je $s = 1$, je enostavno videti, da v prvem koraku porazdeljevanja dobimo vrednosti $A = 1$, $B = 0$ in $C = n+1$, kjer je:

- A = število porazdeljevanj,
- B = število zamenjav,
- C = število primerjav med porazdeljevanjem

V primeru, ko je $s > 1$, pa lahko z malo razmisleka pridemo do vrednosti

$$A = 1,\ B = t,\ C = n + 1,\ za\ 1 \leq s \leq n.$$

Za izračun celotnega trajanja je potrebno tem vrednostim dodati še prispevke iz porazdeljevanj, ki bodo naknadno uredila podzaporedja dolžin $s-1$ in $n-s$.

Sedaj lahko zapišemo formule za A, B in C, predpostavljamo pa, da je originalno zaporedje v naključnem vrstnem redu. Poglejmo si povprečno število primerjav C_n, ki se opravijo med porazdeljevanjem. Pri analizi bomo upoštevali tudi parameter m, ki pomeni največje število elementov podzaporedja, ki ga še urejamo s porazdeljevanjem. Če je torej zaporedje krajše od m, ga uredimo npr. s preprostejšim urejanjem z vstavljanjem, saj je izkoristek tako večji. Ko je $n \leq m$, je $C_n = 0$. V nasprotnem primeru pa, ker se

vsaka vrednost s pojavi z verjetnostjo $\frac{1}{n}$, pridemo do izraza

$$\begin{aligned} C_n &= \tfrac{1}{n}\sum_{s=1}^{n}(n+1+C_{s-1}+C_{n-s}) \\ &= n+1+\tfrac{2}{n}\sum_{0\le k<n} C_k,\ za\ n>m. \end{aligned} \tag{3.1}$$

Podobne enačbe veljajo tudi za A_n in B_n.

Obstaja enostaven način kako rešiti rekurenčne enačbe oblike

$$x_v = f_v + \tfrac{2}{v}\sum_{0\le k<v} x_k,\ za\ v \ge w. \tag{3.2}$$

Najprej se je potrebno znebiti vsote. Enačbo (3.2) pomnožimo z v in odštejemo od iste enačbe za $v+1$

$$\begin{aligned} (v+1)x_{v+1} &= (v+1)f_{v+1} + 2\sum_{0\le k\le v} x_k \\ vx_v &= vf_v + 2\sum_{0\le k<v} x_k \end{aligned}$$

Tako dobimo

$$(v+1)x_{v+1} - vx_v = g_v + 2x_v,\ kjer\ g_v = (v+1)f_{v+1} - vf_v$$

Sedaj lahko rekurenčno enačbo zapišemo v precej bolj enostavni obliki

$$(v+1)x_{v+1} = (v+2)x_v + g_v,\ za\ v \ge w.$$

Vsaka rekurenčna enačba oblike $a_v x_{v+1} = b_v x_v + g_v$ se da skrčiti na enostavnejšo vsoto, če obe strani pomnožimo z $a_0 a_1 \ldots a_{v-1}/b_0 b_1 \ldots b_v$. V našem primeru je ta faktor $v!/(v+2)! = 1/(v+1)(v+2)$, tako da kot posledico enačbe (3.2) dobimo

$$\frac{x_{v+1}}{v+2} = \frac{x_v}{v+1} + \frac{(v+1)f_{v+1} - vf_v}{(v+1)(v+2)},\ za\ v \ge w.$$

Če sedaj nastavimo $f_v = v + 1$ in uporabimo k-to harmonično število $\mathcal{H}_k = \sum_{i=1}^{k} \frac{1}{i}$, dobimo

$$\begin{aligned} x_v/(v+1) &= 2/(v+1) + 2/v + \ldots + 2/(w+2) + x_w/(w+1) \\ &= 2(\mathcal{H}_{v+1} - \mathcal{H}_{w+1}) + x_w/(w+1),\ za\ v \geq w. \end{aligned}$$

Rešitev enačbe (3.1) dobimo, če nastavimo $w = m + 1$ in $x_v = 0$ za $v \leq m$

$$\begin{aligned} C_n &= (n+1)(2\mathcal{H}_{n+1} - 2\mathcal{H}_{m+2} + 1) \\ &\approx 2(n+1)\ln(\tfrac{n+1}{m+2}),\ za\ n > m. \end{aligned} \tag{3.3}$$

Ostale vrednosti lahko izračunamo po istem kopitu. Ko je $n > m$, imamo:

$$\begin{aligned} A_n &= 2(n+1)(m+2) - 1 \\ B_n &= \tfrac{1}{6}(n+1)\left(2\mathcal{H}_{n+1} - 2\mathcal{H}_{m+2} + 1 - \tfrac{6}{m+2}\right) + \tfrac{1}{2} \end{aligned} \tag{3.4}$$

Recimo, da nad kratkimi zaporedji ne bomo uporabili nobenega drugega algoritma za urejanje, torej da je $m = 1$. Potem bi lahko zapisali

$$C_n \approx 2(n+1)\big(1/3 + 1/4 + \ldots + 1/(n+1)\big)$$

Vrednost v oklepaju je diskretni približek področja pod krivuljo $2/x$ od 3 do $n + 1$. Z integriranjem dobimo $C_n \approx 2n \ln n \approx 1.386 n \log_2 n$, kar pomeni da je povprečno število primerjav le 39 procentov večje kot v najboljšem primeru. Po podobnem principu lahko izračunamo, da algoritem opravi približno šestkrat manj zamenjav kot primerjav [5].

Navzlic mnogim prednostim ima algoritem ogromno pomanjkljivost. V primeru, da se število elementov spodnjega in zgornjega dela zelo razlikuje, se algoritem izkaže za skrajno neučinkovitega. Če je zaporedje že urejeno, algoritem v vsakem koraku obdela le po en element. Število primerjav v najslabšem primeru je torej

$$n + (n-1) + (n-2) + \ldots + 2 + 1 = (n+1)n/2.$$

Situaciji, kjer je časovna zahtevnost algoritma $\mathcal{O}(n^2/2)$, se lahko izognemo tako, da zaporedje pred urejanjem naključno premešamo [1].

Hoare je v svojem originalnem članku predlagal dva boljša načina izbire pivotnega elementa. Prvi način je, kot smo že povedali, da za pivot izberemo naključni element. Glede na prej izračunane enačbe, moramo naključno število v povprečju izračunati samo $2(n+1)/(m+2)-1$ krat, tako da dodatnega časa ne porabimo veliko, vendar pa s tem dobimo precej dobro zaščito pred najslabšim primerom [5].

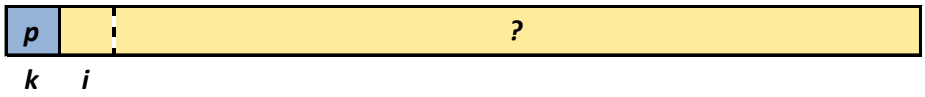
Drugi predlog je bil, da se pivot izbere kot mediana nekega majhnega vzorca. Tega principa se je poslužil R. C. Singleton [10], ki je za pivot vzel mediano prvega, srednjega in zadnjega elementa. S tem se je število primerjav zmanjšalo iz $1.386n \log n$ na približno $1.188n \log n$, število zamenjav pa povišalo iz $C_n/6$ na $C_n/5$. Celoten čas algoritma se je tako zmanjšal za približno 8%. Najslabši možni primer je sicer še vedno reda n^2, verjetnost, da do tega pride, pa je zelo majhna.

3.2 Eno-zančno hitro urejanje

Porazdelitev s samo enim obhodom zanke je predlagal Nico Lomuto, algoritem pa je po njegovi zamisli implementiral Jon Bentley [6].

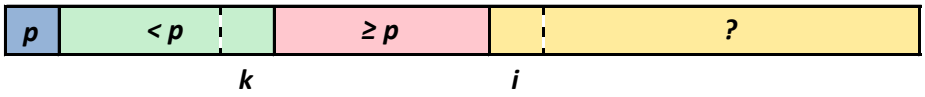
Porazdelitev

Začetek je podoben kot pri prejšnji porazdelitvi. Za pivot se vzame prvi element tabele, preostali del zaporedja je zaenkrat neobdelan.

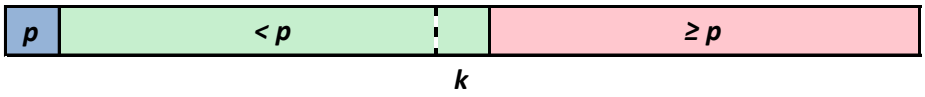


Neobdelani del se med porazdeljevanjem razdeli na *spodnji del*, ki vsebuje elemente, manjše od pivota, na *zgornji del*, ki vsebuje elemente, večje od pivota, ter na del s preostalimi neobdelanimi elementi. Kazalec k kaže na zadnji element spodnjega dela in tako ponazarja mejo med spodnjim in zgornjim delom. Kazalec i pa je v bistvu števec, ki se sprehaja preko tabele,

predstavlja torej mejo med zgornjim in neobdelanim delom in kaže na prvi še neobdelani element.



Porazdeljevanje je sprehajanje z indeksom i od prvega elementa, ki sledi pivotu, do konca tabele. Pri tem se vsak element primerja s pivotom, da se razloči, v kateri del spada. Če je večji od pivota, je potrebno samo povečati i, saj je že v zgornjem delu tabele. Če pa je manjši od pivota, ga je potrebno prestaviti v spodnji del. To se naredi tako, da se najprej poveča kazalec k - ta sedaj kaže na prvi element zgornjega dela. Nato je potrebno samo še zamenjati k-ti in i-ti element ter povečati i. Tako se i-ti element prestavi na zadnje mesto v spodnjem delu, kot stranski učinek pa se prvi element zgornjega dela prestavi na zadnje mesto zgornjega dela. Zgleda, kot da zgornji del naredi rotacijo v levo. Ob prehodu čez celotno tabelo stvari stojijo takole:



Na koncu je potrebno le še zamenjati pivot z zadnjim elementom spodnjega dela, torej s k-tim elementom.

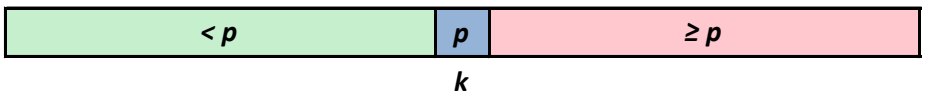
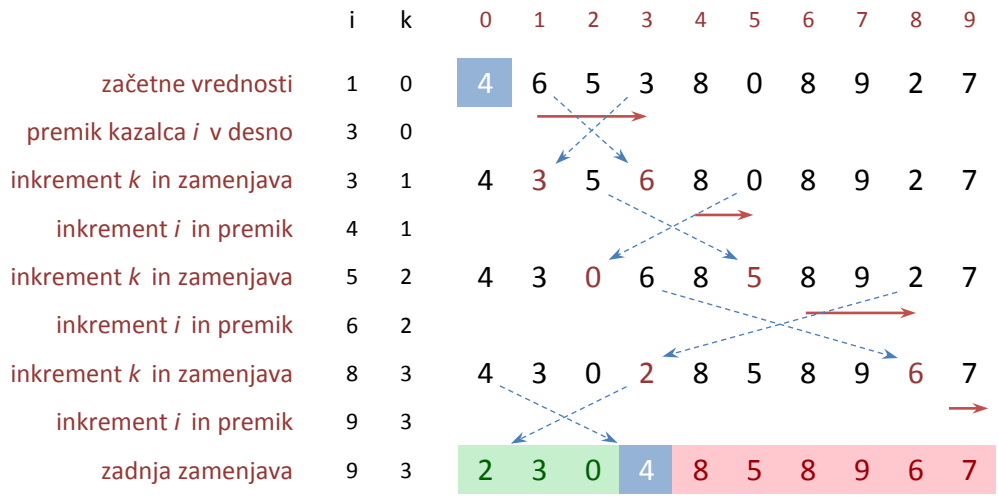


Tabela seveda po enem koraku porazdeljevanja še ni v celoti urejena, je le razdeljena na tri dele, torej na spodnji del, pivot ter zgornji del. Algoritem mora za popolno ureditev še rekurzivno urediti spodnji in zgornji del.

Preprost primer enega koraka porazdeljevanja lahko vidimo na sliki 3.2.



Slika 3.2: Primer eno-zančnega porazdeljevanja

Algoritem

Celoten algoritem eno-zančnega hitrega urejanja prikazuje izvorna koda 3.2. Glavna značilnost te porazdelitve je enostavna implementacija z eno samo zanko `for`. Funkcija `quicksort` se ponavlja iz algoritma v algoritem, metode porazdeljevanja pa se med seboj razlikujejo. Ključni del se dogaja v vrsticah 10 in 11, kjer se ugotovi, kateremu delu pripada trenutni element. Če je le-ta večji od pivota, je na pravem mestu in se v jedru zanke sploh ne zgodi nič. Če pa je trenutni element manjši od pivota, ga je potrebno prestaviti v levi, spodnji del, vendar tam ni prostora. Zato povečamo kazalec k in opravimo zamenjavo ter tako postavimo oba elementa na pravo mesto. V 12. vrstici se zamenja še pivot.

Analiza

Lomutova porazdelitev z eno zanko je izmed vseh opisanih najbolj enostavna. Tudi Jon Bentley se je pritoževal nad zapletenostjo originalne rešitve in bil navdušen nad enostavnostjo Lomutove [6].

Število primerjav v enem koraku porazdeljevanja je vedno točno $n - 1$, kjer je n dolžina trenutnega podzaporedja. To pomeni, da se opravi ena

Izvorna koda 3.2: Eno-zančno hitro urejanje

```
1  void quicksort(Item a[], int left, int right) {
2    if (right <= left) return;
3    int p = partition(a, left, right);
4    quicksort(a, left, p-1);
5    quicksort(a, p+1, right);
6  }
7
8  int partition(Item a[], int left, int right) {
9    int i, k = left; Item p = a[left];
10   for(i = left + 1; i <= right; i++)
11     if(less(a[i], p)) exch(a, i, ++k);
12   exch(a, left, k);
13   return k;
14 }
```

primerjava za vsak element (razen pivota), kar je drugače kot pri porazdeljevanju s križanjem kazalcev. Tam se opravi n oz. $n+1$ primerjav, odvisno od tega, kako daleč prodre desna meja v levi del.

Število zamenjav pri enem koraku eno-zančnega porazdeljevanja znaša $\frac{n}{2} - \frac{1}{2}$, kar je trikrat toliko kot pri porazdeljevanju s križanjem kazalcev [19]. Zaradi lažje analize privzemimo, da urejamo naključne permutacije elementov $\{1, \ldots, n\}$. Sprehajalni indeks i preišče celotno zaporedje in opravi zamenjavo vsakič, ko najde element, ki je manjši od pivota. V zaporedju $\{1, \ldots, n\}$ je točno $p-1$ elementov manjših od pivota p, torej opravimo $p-1$ zamenjav. Celotno pričakovano vrednost števila zamenjav dobimo s povprečjem preko vseh možnih pivotov [19]. Verjetnost elementa, da bo izbran za pivot, je $\frac{1}{n}$, povprečno število zamenjav je torej:

$$\frac{1}{n}\sum_{p=1}^{n}(p-1) = \frac{n}{2} - \frac{1}{2}$$

Recimo, da bi za pivot namesto skrajno levega elementa izbrali skrajno desnega. Potem bi eno-zančno porazdeljevanje pri urejanju že urejenega zaporedja naredilo veliko nepotrebnih zamenjav, medtem ko porazdeljevanje s

križanjem kazalcev ne bi naredilo nobene. Pri urejanju zaporedja s samimi enakimi elementi pa bi originalna rešitev sicer zamenjala vsaka dva para elementov, vendar bi se sprehajalna indeksa srečala na polovici zaporedja in zahtevnost bi vseeno ostala reda $n \log n$. Pri Lomutovi porazdelitvi bi se ravno tako zamenjal vsak element, kar pa je huje - ob koncu koraka bi vsakič dobili eno prazno podzaporedje, drugo pa bi vsebovalo vse ostale elemente. Zahtevnost algoritma bi bila v tem primeru najslabša možna, torej reda n^2.

3.3 Tro-smerno hitro urejanje

Do sedaj smo predpostavili, da so vsi elementi zaporedja med seboj različni, kar pa se dejansko le redko zgodi (npr. pri urejanju elementov množice). V praksi imamo pogosto opravka z zaporedji, ki vsebujejo veliko število podvojenih elementov. V podjetju bi na primer želeli urediti svoje zaposlene po letnici rojstva, ali pa jih mogoče ločiti glede na spol. Na splošno sicer zadovoljivo obnašanje algoritmov za hitro urejanje se da v takšnih primerih še precej izboljšati. S podzaporedji, sestavljenimi iz samih enakih elementov, se načeloma ni treba več ukvarjati, pa vendar jih do sedaj omenjeni algoritmi vseeno razbijajo na manjše in manjše kose.

V sledečem sestavku si bomo pogledali porazdelitve, ki zaporedja ne razdelijo več samo na dele s strogo večjimi ali manjšimi elementi, temveč posebej obravnavajo tudi elemente, ki so pivotu enaki. Takim porazdelitvam rečemo tro-smerne porazdelitve, z njimi pa so se veliko ukvarjali Dijkstra [4], Sedgewick [1] ter Bentley in McIlroy [11].

3.3.1 Osnovna porazdelitev

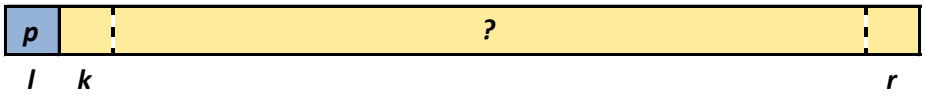
Enostavnejša različica tro-smernega porazdeljevanja je ekvivalentna problemu *Nizozemske narodne zastave (ang. Dutch national flag)* [18], ki ga je predlagal Dijkstra [4]. Nizozemska zastava sestoji iz rdeče, bele in modre barve. Recimo, da imamo v vrsti naključno postavljene žoge teh treh barv. Rešitev problema je takšna razvrstitev žog, pri kateri so vse žoge enake barve

skupaj, skupine žog pa si sledijo v pravilnem vrstnem redu.

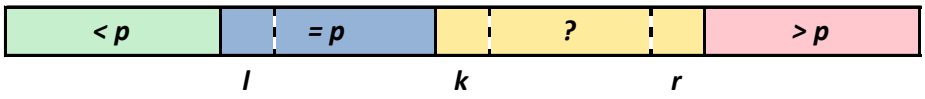
Dijkstrova rešitev problema, prenesena na problem porazdeljevanja, je enostavna. Temelji na enem samem prehodu zanke, kjer se vsak element primerja s pivotom in postavi v ustreznega izmed treh delov.

Porazdelitev

Zaporedje se razdeli na štiri kose: *spodnji del*, v katerem so elementi, manjši od pivota, *srednji del*, kjer so elementi, enaki pivotu, *neobdelani del* ter *zgornji del* z elementi, ki so od pivota večji. Začetno stanje je sledeče:



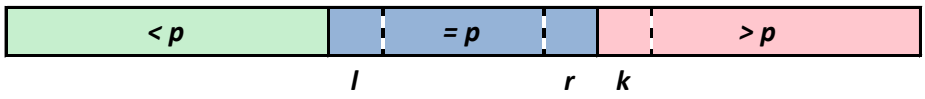
Tekom porazdeljevanja indeksa l in r omejujeta spodnji in zgornji del, l kaže na prvi element srednjega dela, r pa na zadnji element neobdelanega dela. Kazalec k kaže na prvi še neobdelani element in je torej meja med srednjim in neobdelanim delom.



Sprehajalni indeks k se pelje preko neobdelanega dela zaporedja do zgornjega dela. Na vsakem koraku se k-ti element primerja s pivotom in ima tri možnosti. Lahko je:

- manjši od pivota, torej sodi v spodnji del, zato se zamenja z l-tim elementom in kazalca l in k se povečata;
- večji od pivota, torej sodi v zgornji del, zato se zamenja z r-tim elementom, kazalec r pa se zmanjša. Kazalca k ne povečamo, saj smo na njegovo mesto dobili bivši r-ti element, ki je še neobdelan;
- enak pivotu, torej sodi v srednji del, kjer pa se v bistvu že nahaja, zato samo povečamo kazalec k.

Po končanem sprehodu čez celotno tabelo neobdelanega dela ni več, ostanejo nam samo še spodnji, srednji ter zgornji del.



Preprost primer enega koraka porazdeljevanja lahko vidimo na sliki 3.3.

Slika 3.3: Primer osnovnega tro-smernega porazdeljevanja

Algoritem

Celoten algoritem osnovnega tro-smernega hitrega urejanja prikazuje izvorna koda 3.3.

Porazdeljevanje se izvršuje med vrsticami 6 in 10. Opazimo lahko, da se prav vsak element zamenja, izjema so le tisti, ki so pivotu enaki. V primeru, ko število podvojenih elementov v zaporedju ni veliko, se zato opravi veliko več zamenjav, kot bi jih opravil kakšen od prej omenjenih algoritmov. Te težave pa prebrodi popularnejši algoritem, ki sta ga razvila Bentley in McIlroy.

Izvorna koda 3.3: Osnovno tro-smerno hitro urejanje

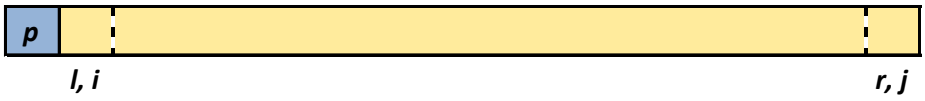
```
1  void quicksort(Item a[], int left, int right) {
2    if (right <= left) return;
3
4    int l = left, k = left + 1, r = right; Item p = a[left];
5
6    while(k <= r) {
7      if (less(a[k], p)) exch(a, l++, k++);
8      else if(less(p, a[k])) exch(a, k, r--);
9      else k++;
10   }
11
12   quicksort(a, left, l-1);
13   quicksort(a, r+1, right);
14 }
```

3.3.2 Bentley-McIlroyeva porazdelitev

V devetdesetih letih prejšnjega stoletja sta J. Bentley in D. McIlroy razvila boljšo različico tro-smernega porazdeljevanja. Dognala sta, da se v praktičnih situacijah, kjer je veliko podvojenih elementov, njun algoritem obnese veliko bolje od urejanja z zlivanjem in drugih metod.

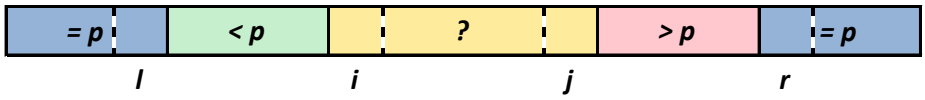
Porazdelitev

Začetno stanje je takšno kot ponavadi:



Tekom porazdeljevanja se tabela spet razdeli na spodnji, srednji in zgornji del. Za razliko od prejšnje metode pa sta srednja dela sedaj dva - vsak se kopiči na svojem robu tabele.

Indeksa l in r postavljata meji med srednjim delom ter spodnjim oz. zgornjim delom. Indeksa i in j pa sta sprehajalna indeksa in delujeta na podoben



način kot pri porazdeljevanju s križanjem kazalcev. Ta dva indeksa se torej sprehajata vsak v svojo smer, dokler ne naletita na element, ki je ravno na napačnem mestu. Taka elementa se zamenjata in proces se nadaljuje, dokler se kazalca ne prekrižata. Če se ustavita na pivotu enakem elementu, ga prestavita v levi oz. desni srednji del. Na koncu porazdeljevanja je seveda potrebno oba srednja dela prestaviti na sredino tabele. Elemente levega srednjega dela zato zamenjamo s tistimi elementi spodnjega dela, ki so naravnani bolj proti desni strani. Podobno elemente desnega srednjega dela zamenjamo z bolj levo naravnanimi elementi zgornjega dela.

Preprost primer enega koraka porazdeljevanja lahko vidimo na sliki 3.4.

Slika 3.4: Primer Bentley-McIlroyevega tro-smernega porazdeljevanja

Algoritem

Celoten algoritem Bentley-McIlroyevega tro-smernega hitrega urejanja prikazuje izvorna koda 3.4.

Izvorna koda 3.4: Bentley-McIlroyevo tro-smerno hitro urejanje

```
1  void quicksort(Item a[], int left, int right) {
2    if (right <= left) return;
3
4    int k, l = left, i = left, j = right + 1, r = j;
5    Item p = a[left];
6
7    while(1) {
8      while (less(a[++i], p)) if(i == right) break;
9      while (less(p, a[--j])) ;
10     if (i >= j) break;
11     exch(a, i, j);
12     if(equal(a[i], p)) exch(a, ++l, i);
13     if(equal(a[j], p)) exch(a, --r, j);
14   }
15   i = j + 1;
16   for(k = left; k <= l; k++) exch(a, k, j--);
17   for(k = right; k >= r; k--) exch(a, k, i++);
18
19   quicksort(a, left, j);
20   quicksort(a, i, right);
21 }
```

Sam algoritem je zelo podoben algoritmu hitrega urejanja s križanjem kazalcev, ki smo ga opisali v razdelku 3.1. V vrsticah od 8 do 11 se praktično ne razlikujeta, dodano je še preverjanje enakosti elementa pivotu v vrsticah 12 in 13. Po končanem porazdeljevanju se v vrsticah 16 in 17 srednji del premakne iz robov tabele na sredino.

Največja prednost te implementacije v primerjavi s prejšnjo je način obravnavanja elementov, enakih pivotu. Prejšnja verzija se najbolje počuti pri urejanju majhnega števila različnih elementov. Večje število pivotu ena-

kih elementov torej pomeni boljšo zmogljivost, saj je za vsak element, ki je od pivota različen, potrebno narediti dodatno zamenjavo. Bentley-McIlroyeva verzija pa naredi dodatne zamenjave samo za pivotu enake elemente. Če takih ni, se obnaša podobno kot porazdeljevanje s križanjem kazalcev, kar ni slabo. Če je pivotu enakih elementov veliko, sicer zapravlja čas s prestavljanjem elementov na rob in na koncu nazaj na sredo tabele. Vendar pa izgubljeno hitro nadoknadi, saj je število rekurzivnih klicev veliko manjše na račun srednjega dela, s katerim se ni potrebno več ukvarjati.

Analiza

Povedali smo že, da izbira pivotnega elementa precej vpliva na obnašanje algoritma. Več kot vložimo truda v iskanje dejanskega srednjega elementa, bolj se časovna zahtevnost približa redu $n \log n$. Avtorja sta v končni verziji algoritma pivot izbrala tako, da sta izračunala "psevdo-mediano". Proceduro je zasnoval John Tukey, deluje pa tako, da vzame tri vzorce, vsakega iz treh elementov, iz njih izračuna mediane in kot rezultat vrne mediano vseh treh median. Veliko boljši približek za srednji element tako dobimo za ceno samo dvanajstih dodatnih primerjav. Res pa je, da je to za kratka zaporedja veliko, zato njuna končna koda za pivot kratkih zaporedij vzame srednji element, pivot srednje dolgih zaporedij izbere kot mediano prvega, srednjega in zadnjega elementa, v dolgih zaporedjih pa izračuna psevdo-mediano devetih enakomerno razporejenih elementov. Končna verzija algoritma se poslužuje tudi izboljšave, ki jo je predlagal Robert Sedgewick - kratka zaporedja se urejajo s pomočjo urejanja z vstavljanjem.

Bentley in McIlroy sta za potrebe analize pravilnosti in učinkovitosti algoritma ustvarila program, ki je samodejno generiral neugodna zaporedja, vključno z vsemi nadležnimi primeri, ki sta jih našla v literaturi. Nato je nad njimi pognal algoritem za urejanje in se pritožil, če je pri tem porabil preveliko število primerjav. Vsak test je torej štel primerjave in zabeležil primere, kjer je številka presegla $An \log n$, A pa je tipično bil 1.2. Če je bilo število primerjav ogromno ($A = 10$), se je urejanje prekinilo. Prav s takšnim

testiranjem sta avtorja našla veliko hroščev v prejšnjih implementacijah algoritma. Za razliko od slednjih se je njun novi algoritem izkazal za precej robustnega. Opozorilni prag $1.2n \log n$ je preseglo le manj kot 2 procenta vseh testnih primerov, številka pa ni nikoli presegla $1.5n \log n$ [11].

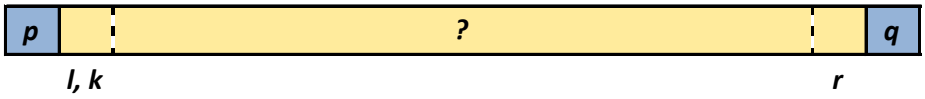
3.4 Dvo-pivotno hitro urejanje

Dolgo je veljalo, da se uporaba dveh ali več pivotov pri porazdeljevanju v praksi ne obnese dobro. Z dvo-pivotnimi porazdelitvami je začel že Robert Sedgewick leta 1975, vendar je njegova metoda naredila večje število primerjav in zamenjav kot običajni algoritem. Pascal Hennequin je leta 1991 ugotovil, da je njegova verzija porazdeljevanja z r pivoti pri $r = 2$ podobno učinkovita kot original, če pa je r večji, so prihranki zelo majhni, porazdeljevanje pa veliko bolj zapleteno [17].

Prepričanje o pomanjkljivostih in slabostih dvo- ali več pivotnega porazdeljevanja je zaradi vsega tega veljalo do leta 2009, ko je Vladimir Yaroslavskiy na forumu objavil svojo izboljšano verzijo. Njegov algoritem je hiter in enostaven, postal pa je tudi privzet način urejanja v Javi 7.

Porazdelitev

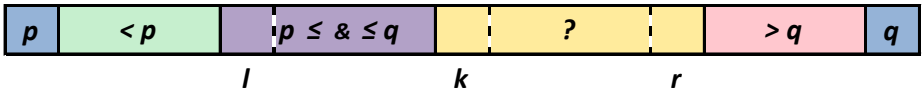
Začetno stanje je sledeče:



Zaporedje se tekom porazdeljevanja razdeli na šest delov:

- levi pivot p,
- *spodnji del* z elementi, manjšimi od p,
- *srednji del* z elementi med p in q,
- del z neobdelanimi elementi,
- *zgornji del* z elementi, večjimi od q ter

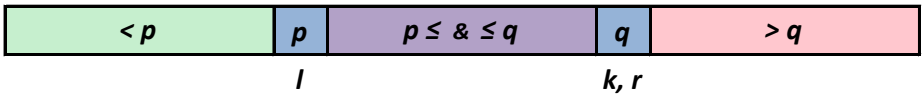
- desni pivot q



Indeks l je meja med spodnjim in srednjim delom, r pa med neobdelanim in zgornjim delom. Indeks k se sprehaja preko neobdelanega dela tabele in vsak element primerja z obema pivotoma. Možnosti so naslednje:

- trenutni element je manjši od pivota p - potrebno ga je zamenjati z l-tim elementom in povečati oba indeksa l in k;
- trenutni element je večji od desnega pivota q - potrebno ga je zamenjati z r-tim elementom in zmanjšati r;
- vrednost trenutnega elementa je med obema pivotoma p in q - samo povečamo k.

Ko preidemo čez celotno tabelo, je potrebno na prava mesta postaviti še oba pivota, zato levega zamenjamo z zadnjim elementom spodnjega dela, desnega pa s prvim elementom zgornjega dela.



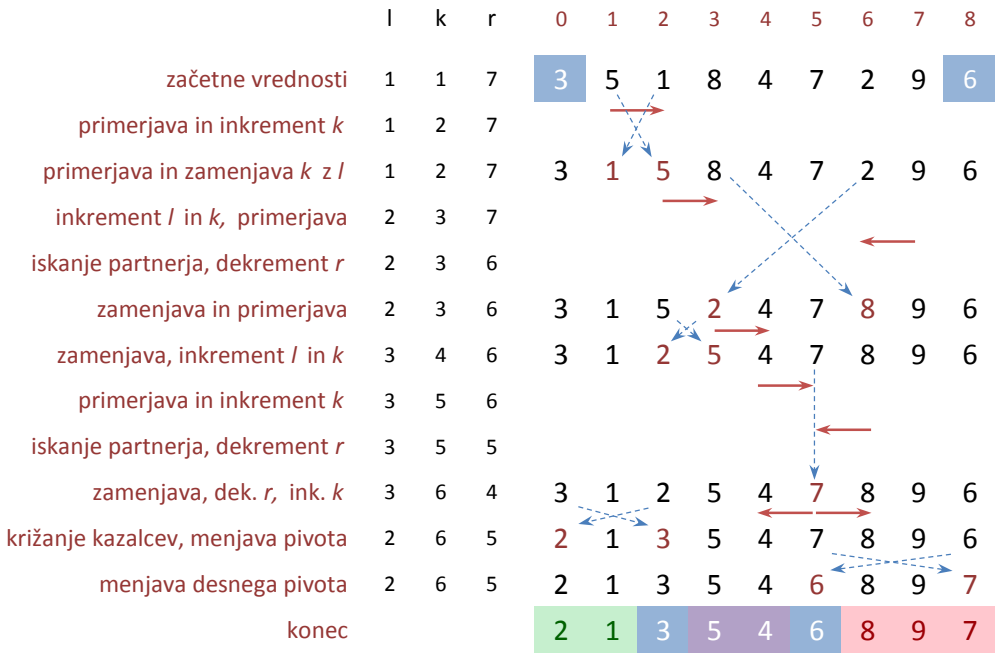
Za dokončno ureditev zaporedja so tokrat seveda potrebni trije rekurzivni klici, za vsak del eden.

Preprost primer enega koraka porazdeljevanja lahko vidimo na sliki 3.5.

Algoritem

Celoten algoritem dvo-pivotnega hitrega urejanja prikazuje izvorna koda 3.5.

Pred začetkom porazdeljevanja je potrebno določiti pivotne elemente. Pričakuje se, da je levi pivot p manjši od desnega q, kar se uredi v 4. vrstici.



Slika 3.5: Primer dvo-pivotnega porazdeljevanja

Po končani zanki je potrebno prestaviti še oba pivota, kar se zgodi v vrsticah 19 in 20.

Porazdeljevanje, kot smo ga opisali v prejšnjem razdelku, še ni maksimalno dodelano, zato ga je avtor algoritma še malce izboljšal. Recimo, da imamo primer, ko je trenutni element k večji od desnega pivota q - takrat se morata zamenjati k-ti in r-ti element. Vendar pa je lahko ta r-ti element tudi sam večji od pivota q, torej je že na pravem mestu. Zato v 12. vrstici pred zamenjavo najprej preskočimo elemente, ki so večji od q. Tako zagotovo ne bomo naredili odvečne zamenjave.

Drugi trik pa temelji na želji, da bi sprehajalni indeks k povečali v vsaki iteraciji zunanje zanke `while`. Pri tem nas ovira samo iz prejšnje točke zamenjan element, ki je po novem na mestu k. Vemo sicer, da je manjši od q, ne vemo pa, ali je manjši ali večji od p. Zato v 14. vrstici dodatno preverimo še ta pogoj in s tem k-ti element dokončno obdelamo [17].

Izvorna koda 3.5: Dvo-pivotno hitro urejanje

```
1  void quicksort(Item a[], int left, int right) {
2    if (right <= left) return;
3
4    if(a[left] > a[right]) exch(a, left, right);
5    Item p = a[left], q = a[right];
6
7    int l = left + 1, k = l, r = right - 1;
8
9    while(k <= r) {
10     if (less(a[k], p)) exch(a, l++, k);
11     else if(less(q, a[k])) {
12       while(less(q, a[r]) && k < r) r--;
13       exch(a, k, r--);
14       if(less(a[k], p)) exch(a, l++, k);
15     }
16     k++;
17   }
18
19   exch(a, --l, left);
20   exch(a, ++r, right);
21
22   quicksort(a, left, l-1);
23   quicksort(a, l+1, r-1);
24   quicksort(a, r+1, right);
25 }
```

Analiza

Celotna analiza je povzeta po [13]. Med porazdeljevanjem moramo za vsak element $x \notin \{p, q\}$ ugotoviti, ali velja $x < p$, $p < x < q$ ali $q < x$. V ta namen moramo x primerjati s p *in/ali* q. Recimo, da x najprej primerjamo s p - potem je za vse možne vrednosti x, p in q v povprečju $\frac{1}{3}$ možnosti, da je $x < p$. V tem primeru ne potrebujemo nobene dodatne primerjave, drugače pa moramo x primerjati še z q. Pričakovano število primerjav za en element je tako $\frac{1}{3} \cdot 1 + \frac{2}{3} \cdot 2 = \frac{5}{3}$. V enem koraku porazdeljevanja n elementov, vključno

s pivotoma p in q, je pričakovano število primerjav torej $\frac{5}{3} \cdot (n-2)$.

V naključnem permutacijskem modelu nam poznavanje elementa $y \neq x$ ne pove nič o razmerju med x-om in pivotoma. Mislili bi si torej, da vsaka metoda porazdeljevanja potrebuje vsaj $\frac{5}{3} \cdot (n-2)$ primerjav, vendar temu ni tako. Razlog tiči v prej omenjeni predpostavki o neodvisnosti elementov, ki pa drži samo za algoritme, pri katerih se primerjanje opravi na točno eni sami lokaciji v kodi. Naš algoritem vsebuje več ukazov za primerjanje na različnih odsekih v kodi, katere dejansko doseže, pa je *odvisno* od pivotov p in q. Seveda, saj je logično, da bo število elementov, ki bodo pristali v spodnjem, srednjem ali zgornjem delu direktno odvisno od obeh pivotov. Če se pogosto izvede primerjava, če je p velik, potem se splača najprej preveriti $x < p$ - primerjava se zgodi bolj pogosto kot v povprečju samo, če je verjetnost, da je $x < p$ večja kot v povprečju. Tako se lahko zgodi, da število pričakovanih primerjav za ta element pade pod spodnjo mejo $\frac{5}{3}$. To pa je tudi glavna prednost tega algoritma: vedno se najprej obdela "boljša"primerjava, torej tista, ki zahteva manjše število naknadnih korakov.

Rekurenčna enačba Privzeli bomo, da so vhodna zaporedja naključne permutacije elementov $\{1, \ldots, n\}$ in vsaka permutacija se zgodi z verjetnostjo $1/n!$. Prvi in zadnji element bosta pivota, manjši bo p, večji pa q. Vsaka primerjava ključev vključuje pivotni element iz trenutnega koraka porazdeljevanja. Naključnost se ohranja, kar pomeni, da če je originalno zaporedje naključna permutacija elementov, so naključna tudi podzaporedja, nad katerimi se rekurzivno kliče urejanje. Zaradi tega lahko, podobno kot v razdelku 3.1, postavimo enačbo za rekurenčno relacijo na podlagi prvega koraka porazdeljevanja, prispevke ostalih korakov pa izračunamo z indukcijo po n.

Pričakovana cena C_n za urejanje naključne permutacije zaporedja dolžine

n ustreza rekurenčni relaciji (za $n \geq 2$):

$$\begin{aligned} C_n &= \sum_{1\leq p<q\leq n} Verjetnost\,[pivota(p,q)] \cdot (cena\ porazdel. + rekurzija) \\ &= \sum_{1\leq p<q\leq n} \frac{2}{n(n-1)}(cena\ porazdeljevanja + C_{p-1} + C_{q-p-1} + C_{n-q}) \\ &= \mathbb{E}\ cena\ porazdeljevanja + \frac{2}{n(n-1)} \cdot 3\sum_{k=0}^{n-2}(n-k-1)C_k \qquad (3.5) \end{aligned}$$

Ker algoritem preskoči podzaporedja dolžine ≤ 1, imamo osnovna primera $C_0 = C_1 = 0$.

Rekurenčno relacijo bomo rešili za linearne pričakovane cene porazdeljevanja $a(n+1)+b$. Izkaže se, da na tak način ne moremo izračunati cen za zaporedja dolžine $n = 2$, zato k osnovnim primerom dodamo še $C_2 = d$, rekurenčno enačbo pa rešujemo za $n \geq 3$.

Najprej nastavimo $D_n := \binom{n+1}{2}C_{n+1} - \binom{n}{2}C_n$, da se znebimo faktorja v vsoti:

$$\begin{aligned} D_n &= \tbinom{n+1}{2}(a(n+2)+b) - \tbinom{n}{2}(a(n+1)+b) + \\ &\quad \tfrac{(n+1)n}{2}\tfrac{6}{(n+1)n}\sum_{k=0}^{n-1}(n-k)C_k - \tfrac{n(n-1)}{2}\tfrac{6}{n(n-1)}\sum_{k=0}^{n-2}(n-k-1)C_k \\ &= 3\tbinom{n+1}{2}a + n\cdot b + 3\sum_{k=0}^{n-1}C_k,\ za\ n \geq 3. \end{aligned}$$

Preostali del rekurenčne enačbe lahko preprosto rešimo z razlikami $E_n := D_{n+1} - D_n = 3(n+1)a + b + 3C_n$ za $n \geq 3$. Z uporabo E_n in parih nezanimivih, a pomembnih operacij, dobimo

$$(E_n - 3C_n)/\tbinom{n+2}{2} = C_{n+2} - \tfrac{2n}{n+2}C_{n+1} + \tfrac{n-3}{n+1}C_n.$$

Če uvedemo še eno spremenljivko $F_n := C_n - \frac{n-4}{n}\cdot C_{n-1}$, se hitro vidi, da velja $F_{n+2} - F_{n+1} = C_{n+2} - \frac{2n}{n+2}C_{n+1} + \frac{n-3}{n+1}C_n$, tako da lahko zapišemo

$$F_{n+2} - F_{n+1} = (E_n - 3C_n)/\tbinom{n+2}{2} = \big(3(n+1)a+b\big)/\tbinom{n+2}{2},\ za\ n \geq 3.$$

To zadnjo enačbo lahko sedaj iteriramo:

$$\begin{aligned}
F_n &= \sum_{i=5}^{n} \big(3(i-1)a+b\big)/\tbinom{i}{2} + F_4 \\
&= \sum_{i=5}^{n} \frac{3(i-1)a}{\frac{1}{2}i(i-1)} + \sum_{i=5}^{n} \frac{b}{\frac{1}{2}i(i-1)} + F_4 \\
&= 6a\sum_{i=5}^{n} \tfrac{1}{i} + 2b\sum_{i=5}^{n} (\tfrac{1}{i-1} - \tfrac{1}{i}) + F_4 \\
&= 6a(\mathcal{H}_n - \mathcal{H}_4) + 2b(\tfrac{1}{4} - \tfrac{1}{n}) + F_4, \ za\ n \geq 5.
\end{aligned}$$

Če združimo zadnjo enačbo z definicijo F_n, dobimo

$$C_n = \tfrac{n-4}{n} \cdot C_{n-1} + 6a(\mathcal{H}_n - \mathcal{H}_4) + 2b(\tfrac{1}{4} - \tfrac{1}{n}) + F_4 \tag{3.6}$$

Z množenjem z $\binom{n}{4}$ in upoštevanjem $\binom{n}{4} \cdot \frac{n-4}{n} = \binom{n-1}{4}$ dobimo rekurenčno enačbo za $G_n := \binom{n}{4} C_n$:

$$\begin{aligned}
G_n &= G_{n-1} + 6a(\mathcal{H}_n - \mathcal{H}_4)\tbinom{n}{4} + 2b(\tfrac{1}{4} - \tfrac{1}{n})\tbinom{n}{4} + F_4\tbinom{n}{4} \\
&= \sum_{i=5}^{n} \Big[6a(\mathcal{H}_i - \mathcal{H}_4)\tbinom{i}{4} + 2b(\tfrac{1}{4} - \tfrac{1}{i})\tbinom{i}{4} + F_4\tbinom{i}{4}\Big] + G_4 \\
&= \sum_{i=1}^{n} \Big[6a(\mathcal{H}_i - \mathcal{H}_4)\tbinom{i}{4} + 2b(\tfrac{1}{4} - \tfrac{1}{i})\tbinom{i}{4} + F_4\tbinom{i}{4}\Big] \underbrace{-F_4\tbinom{4}{4} + G_4}_{=0} \\
&= 6a\sum_{i=1}^{n} \mathcal{H}_i\tbinom{i}{4} + (\tfrac{1}{2}b - 6\mathcal{H}_4 a + F_4)\sum_{i=1}^{n} \tbinom{i}{4} - 2b\sum_{i=1}^{n} \tfrac{1}{i}\tbinom{i}{4} \\
&= 6a\tbinom{n+1}{5}(\mathcal{H}_{n+1} - \tfrac{1}{5}) + (\tfrac{1}{2}b - 6\mathcal{H}_4 a + F_4)\tbinom{n+1}{5} - 2b\sum_{i=1}^{n} \tfrac{1}{4}\tbinom{i-1}{3} \\
&= 6a\tbinom{n+1}{5}(\mathcal{H}_{n+1} - \tfrac{1}{5}) + (\tfrac{1}{2}b - 6\mathcal{H}_4 a + F_4)\tbinom{n+1}{5} - \tfrac{1}{2}b\tbinom{n}{4}.
\end{aligned}$$

Končno pridemo do eksplicitne formule za C_n, ki velja za $n \geq 4$:

$$C_n = G_n/\tbinom{n}{4} = \tfrac{6}{5}a \cdot (n+1)(\mathcal{H}_{n+1} - \tfrac{1}{5}) + (\tfrac{1}{10}b - \tfrac{6}{5}\mathcal{H}_4 a + \tfrac{1}{5}F_4) \cdot (n+1) - \tfrac{1}{2}b.$$

Z uporabo $F_4 = 5a + b + \frac{1}{2}d$ lahko enačbo poenostavimo v

$$C_n = \tfrac{6}{5}a \cdot (n+1)(\mathcal{H}_{n+1} - \tfrac{1}{5}) + (-\tfrac{3}{2}a + \tfrac{3}{10}b + \tfrac{1}{10}d) \cdot (n+1) - \tfrac{1}{2}b. \tag{3.7}$$

Cena prvega koraka porazdeljevanja Sedaj je treba analizirati še pričakovano število primerjav in zamenjav v prvem koraku porazdeljevanja naključne permutacije števil $\{1, \ldots, n\}$. Definirajmo množice elementov

- spodnjega dela - $S := \{1, \ldots, p-1\}$,
- srednjega dela - $M := \{p+1, \ldots, q-1\}$,
- zgornjega dela - $L := \{q+1, \ldots, n\}$.

Algoritem zaradi ohranjanja naključnosti ne more razlikovati $x \in C$ od $y \in C$, kjer $C \in \{S, M, L\}$. Zato lahko za potrebe analize cen zamenjamo vse elemente (razen pivotov) z znaki s, m ali l, ki so po vrsti elementi množic S, M ali L. Očitno je, da v vsakem primeru na koncu vsakega koraka porazdeljevanja dobimo enako besedo, in sicer $s \ldots spm \ldots mql \ldots l$. Definirajmo še množice pozicij (indeksov)

- spodnjega dela - $\mathcal{S} := \{2, \ldots, p\}$,
- srednjega dela - $\mathcal{M} := \{p+1, \ldots, q-1\}$,
- zgornjega dela - $\mathcal{L} := \{q, \ldots, n-1\}$.

Z definicijami si lahko pomagamo takole: za neko permutacijo, $c \in \{s, m, l\}$ in množico pozicij $\mathcal{P} \subset \{1 \ldots n\}$, lahko napišemo $c \,@\, \mathcal{P}$, kar bo pomenilo število elementov tipa c, ki se v permutaciji nahajajo na pozicijah v $\mathcal{P}$. V našem primeru porazdeljevanja na sliki 3.5 imamo tako npr. $\mathcal{M} = \{4, 5\}$, na teh pozicijah pa pred porazdeljevanjem najdemo elementa 8 in 4, od katerih eden pripada M, eden pa L, zato lahko napišemo $m \,@\, \mathcal{M} = 1$, $l \,@\, \mathcal{M} = 1$ ter $s \,@\, \mathcal{M} = 0$.

Glede na to, da se ukvarjamo z naključnimi permutacijami, je tudi $c \,@\, \mathcal{P}$ naključna spremenljivka. V analizi nas bo zanimala pričakovana vrednost $c \,@\, \mathcal{P}$ pod pogojem, da sta prvi in zadnji element permutacije pivota p in q oz. ravno obratno, kar bomo zapisali kot $\mathbb{E}\,[c \,@\, \mathcal{P} \mid p, q]$. Ker je število elementov tipa c - $\#c$ - odvisno samo od pivotov, ne od same permutacije, je $\#c$ povsem določena konstanta v $\mathbb{E}\,[c \,@\, \mathcal{P} \mid p, q]$. Torej je $c \,@\, \mathcal{P}$ hipergeometrično porazdeljena naključna spremenljivka: za tipe elementa c izbiramo $\#c$ pozicij izmed $n-2$ možnih pozicij z vzorčenjem brez vračanja.

$\mathbb{E}[c \circledcirc \mathcal{P} \mid p, q]$ lahko tako izrazimo kot pričakovano vrednost hipergeometrične porazdelitve: $\mathbb{E}[c \circledcirc \mathcal{P} \mid p, q] = \#c \cdot \frac{|\mathcal{P}|}{n-2}$ in pridemo do enačbe

$$\begin{aligned} \mathbb{E}[c \circledcirc \mathcal{P}] &= \sum_{1 \le p < q \le n} \mathbb{E}[c \circledcirc \mathcal{P} \mid p, q] \cdot Verjetnost[pivota(p, q)] \\ &= \frac{2}{n(n-1)} \sum_{1 \le p < q \le n} \#c \cdot \frac{|\mathcal{P}|}{n-2} \end{aligned} \tag{3.8}$$

Sedaj, ko smo seznanjeni z notacijo, se lahko lotimo dejanskega računanja pričakovanega števila primerjav in zamenjav.

Primerjave Primerjave v algoritmu se izvajajo na petih mestih, točneje v vrsticah 4, 10, 11, 12 in 14. Četrta vrstica primerja kandidata za pivota in se izvede samo enkrat. Deseta vrstica se izvede za vsako vrednost indeksa k, razen po zadnjem povečanju, ko pogoj zanke ne drži več. Podobno se vrstica 12 izvede za vsako vrednost indeksa r, razen za zadnjo. Primerjava v vrstici 11 se zgodi samo, ko pogoj v vrstici 10 ni izpolnjen. Zaradi vrstice 11 imamo torej toliko primerjav, kolikokrat k doseže vrednosti, kjer je $A[k] \ge p$. Podobno velja za vrstico 14, ki se izvede za vse vrednosti r, kjer $A[r] \le q$ (elementa na indeksih k in r sta se zamenjala eno vrstico višje in čeprav v kodi piše $A[k]$, v bistvu s q primerjamo bivši element $A[r]$).

Na koncu se pivot q prestavi na pozicijo r (vrstica 20), torej mora z upoštevanjem naših predpostavk veljati $r = q$. Indeks r tako v vrstici 12 doseže vrednosti $\mathcal{R} = \{n-1, n-2, \ldots, q\} = \mathcal{L}$. Zunanjo zanko vedno zapustimo s $k = r + 1$ ali $k = r + 2$. V obeh primerih k doseže vsaj vrednosti $\mathcal{K} = \{2, \ldots, q-1\} = \mathcal{S} \cup \mathcal{M}$. Za primer $k = r + 2$ uvedemo nov izraz $3 \cdot \frac{n-q}{n-2}$ [13].

Z vsoto vseh prispevkov dobimo $c_n^{p,q}$ - pogojno pričakovano vrednost

števila primerjav v prvem koraku porazdeljevanja naključne permutacije:

$$\begin{aligned} c_n^{p,q} &= 1 + |\mathcal{K}| + |\mathcal{R}| + \big(\mathbb{E}\,[m \,@\, \mathcal{K} \mid p,q] + \mathbb{E}\,[l \,@\, \mathcal{K} \mid p,q]\,\big) \\ &\quad + \big(\mathbb{E}\,[s \,@\, \mathcal{R} \mid p,q] + \mathbb{E}\,[m \,@\, \mathcal{R} \mid p,q]\,\big) \\ &\quad + 3 \cdot \tfrac{n-q}{n-2} \\ &= n - 1 + ((q-p-1) + (n-q))\tfrac{q-2}{n-2} \\ &\quad + ((p-1) + (q-p-1))\tfrac{n-q}{n-2} \\ &\quad + 3 \cdot \tfrac{n-q}{n-2} \\ &= n - 1 + (n-p-1)\tfrac{q-2}{n-2} + (q+1)\tfrac{n-q}{n-2}. \end{aligned}$$

Po formuli za polno matematično upanje pa dobimo pričakovano število primerjav v prvem koraku porazdeljevanja naključne permutacije $\{1, \ldots, n\}$:

$$\begin{aligned} c_n := \mathbb{E}\, c_n^{p,q} &= \tfrac{2}{n(n-1)} \sum_{p=1}^{n-1} \sum_{q=p+1}^{n} c_n^{p,q} \\ &= n - 1 + \tfrac{2}{n(n-1)(n-2)} \sum_{p=1}^{n-1} (n-p-1) \sum_{q=p+1}^{n} (q-2) \\ &\qquad + \tfrac{2}{n(n-1)(n-2)} \sum_{q=2}^{n} (n-q)(q+1) \sum_{p=1}^{q-1} 1 \\ &= n - 1 + \big(\tfrac{5}{12}(n+1) - \tfrac{4}{3}\big) + \tfrac{1}{6}(n+3) = \tfrac{19}{12}(n+1) - 3 \end{aligned} \tag{3.9}$$

Zamenjave Zamenjave v algoritmu nastopajo v vrsticah 4, 10, 13, 14, 19 in 20. Vrstici 19 in 20 se izvedeta točno enkrat. V četrti vrstici se enkrat zamenjata pivota, če je to potrebno, kar se zgodi z verjetnostjo $\frac{1}{2}$. V deseti vrstici se opravi po ena zamenjava za vsako vrednost indeksa k, kjer je $A[k] < p$. Vrstica 13 se izvede vsakič, ko je k-ti element večji od pivota q. Zamenjavo v 14. vrstici pa dosežemo za vse indekse r, kjer $A[r] < p$ (isti princip kot pri primerjavah).

Z uporabo prej definiranih $\mathcal{K}$ in $\mathcal{R}$ dobimo $s_n^{p,q}$, pogojno pričakovano

vrednost števila zamenjav:

$$\begin{aligned} s_n^{p,q} &= \tfrac{1}{2} + 1 + 1 + \mathbb{E}\,[s \mathbin{@} \mathcal{K} \mid p,q] + \mathbb{E}\,[l \mathbin{@} \mathcal{K} \mid p,q] + \mathbb{E}\,[s \mathbin{@} \mathcal{R} \mid p,q] + \tfrac{n-q}{n-2} \\ &= \tfrac{5}{2} + (p-1)\tfrac{q-2}{n-2} + (n-q)\tfrac{q-2}{n-2} + +(p-1)\tfrac{n-q}{n-2} + \tfrac{n-q}{n-2} \\ &= \tfrac{5}{2} + +(n+p-q-1)\tfrac{q-2}{n-2} + p \cdot \tfrac{n-q}{n-2}. \end{aligned}$$

Zopet povprečimo preko vseh možnih p in q in dobimo:

$$\begin{aligned} s_n := \mathbb{E}\, s_n^{p,q} &= \tfrac{5}{2} + \tfrac{2}{n(n-1)(n-2)} \sum_{q=2}^{n} (q-2) \sum_{p=1}^{q-1} (n+p-q-1) \\ &\quad + \tfrac{2}{n(n-1)(n-2)} \sum_{q=2}^{n} (n-q) \sum_{p=1}^{q-1} p \\ &= \tfrac{5}{2} + \left(\tfrac{5}{12}(n+1) - \tfrac{4}{3}\right) + \tfrac{1}{12}(n+1) = \tfrac{1}{2}(n+1) + \tfrac{7}{6}. \end{aligned} \tag{3.10}$$

Prednost Yaroslavskiyevega algoritma Pri računanju $\mathbb{E}\,[c \mathbin{@} \mathcal{P}]$ za $c = s, m, l$ in $\mathcal{P} = \mathcal{S}, \mathcal{M}, \mathcal{L}$ naletimo na zelo zanimive *asimetrične* rezultate [13]. V povprečju se *več kot polovica* vseh elementov tipa l nahaja v $\mathcal{L}$. To dejstvo pa lahko s pridom izkoristimo, in sicer tako, da če vemo, da gledamo neko pozicijo v $\mathcal{L}$, se nam veliko bolj splača element najprej primerjati s q, saj bo element večji od q s kar polovično verjetnostjo. Točno to pa se v algoritmu zgodi v vrstici 12. Pričakovano število primerjav je tako $< \frac{1}{2} \cdot 2 + \frac{1}{2} \cdot 1 = \frac{3}{2} < \frac{5}{3}$.

S pomočjo rešitve rekurenčne relacije in pričakovanega števila primerjav in zamenjav prvega koraka porazdeljevanja ter približka n-tega harmoničnega števila $\mathcal{H} \approx \ln n + 0.577216\ldots + \mathcal{O}(n^{-1})$ dobimo končno rešitev [13]. Časovna zahtevnost primerjav tako znaša

$$\tfrac{19}{10}(n+1)\mathcal{H}_{n+1} - \tfrac{711}{200}(n+1) + \tfrac{3}{2} \approx 1.9 n \ln n \approx 1.317 n \log n, \tag{3.11}$$

časovna zahtevnost zamenjav pa

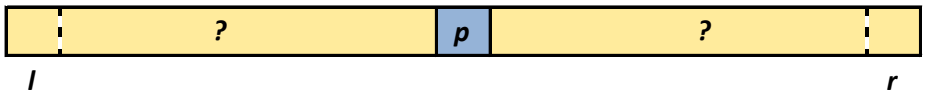
$$\tfrac{3}{5}(n+1)\mathcal{H}_{n+1} - \tfrac{27}{100}(n+1) - \tfrac{7}{12} \approx 0.6 n \ln n \approx 0.42 n \log n. \tag{3.12}$$

3.5 Bsort

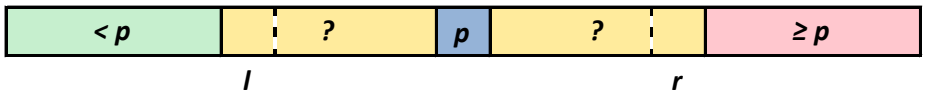
Bsort je različica hitrega urejanja, ki poskuša klasično obliko združiti s tehniko, ki jo uporablja mehurčno urejanje oz. urejanje z navadnimi zamenjavami. Algoritem teži k izboljšavi najslabšega možnega primera $\mathcal{O}(n^2)$ in deluje najbolje nad zaporedji, ki so (delno) urejena ali pa so urejena ravno v napačnem vrstnem redu.

Porazdelitev

Pri Bsortu se za pivot v vsakem koraku vzame srednji element tabele.



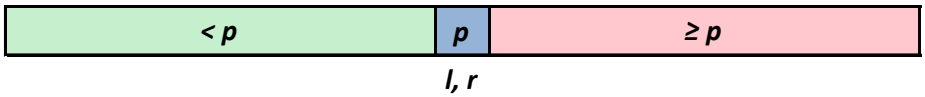
Porazdeljevanje nato poteka podobno kot pri porazdeljevanju s križanjem kazalcev, ki smo ga opisali v razdelku 3.1. Zaporedje se razdeli na spodnji, neobdelan ter zgornji del.



Kazalec l potuje z levega konca tabele proti desnem in pri tem preskakuje elemente, ki so od pivota manjši. Pri tem vsaka dva sosednja elementa primerja ter ju zamenja, v primeru da sta ravno v napačnem vrstnem redu. Tako se zagotovi, da je med gradnjo spodnjega dela najbolj desni element vedno največji. Podobno gre kazalec r iz desne proti levi ter preskakuje elemente, ki so večji ali enaki pivotu, po potrebi pa še zamenja sosednja elementa, če nista v pravem vrstnem redu. Tako je najbolj levi element zgornjega dela vedno najmanjši po vrednosti. Ko se kazalca ustavita, se l-ti in r-ti element nahajata v ravno napačnih delih zaporedja, zato se zamenjata.

Porazdeljevanje je končano, ko se kazalca prekrižata. Če med grajenjem obeh delov ni bila opravljena nobena zamenjava sosednjih elementov, potem so le-ti že v pravem vrstnem redu in nadaljnje porazdeljevanje ter urejanje

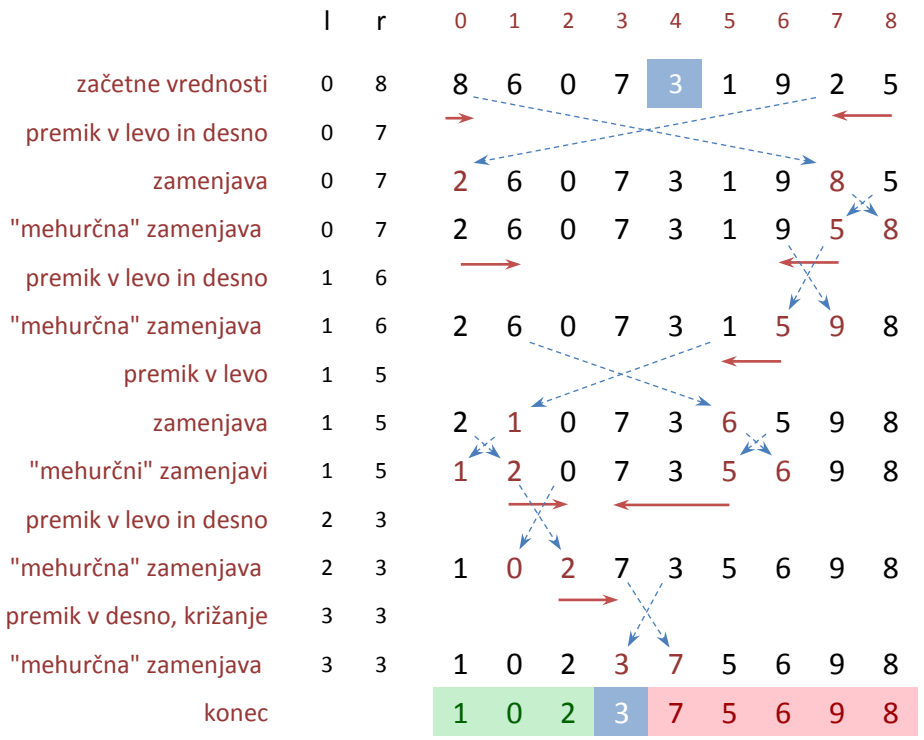
ni potrebno. Poleg tega vemo, da je po končanem porazdeljevanju podzaporedje, ki vsebuje samo dva elementa, zaradi načina konstrukcije spodnjega in zgornjega dela v vsakem primeru že urejeno. Če podzaporedje vsebuje tri elemente, ga je trivialno urediti z eno primerjavo in, če je potrebno, eno zamenjavo. Nadaljnje porazdeljevanje torej ni potrebno, če podzaporedje vsebuje manj kot štiri elemente. V nasprotnem primeru se rekurzivno uredita še spodnji in zgornji del. Pri tem pa nista vključena skrajno levi element zgornjega dela, to je v bistvu pivotni element, in njegov levi sosed, skrajno desni element spodnjega dela, ki je tudi že na pravem mestu [14].



Preprost primer enega koraka porazdeljevanja lahko vidimo na sliki 3.6.

Algoritem

Algoritem deluje po podobnem principu kot hitro urejanje s križanjem kazalcev, vmes pa vključuje še tehnike, poznane iz urejanja z navadnimi zamenjavami. Imamo torej sprehajalna kazalca l in r, ki se sprehajata vsak v svojo smer (vrstice od 7 do 18). Pri tem preskakujeta elemente, ki se že nahajajo v pravem delu, ustavita pa se na tistih, ki jih je potrebno zamenjati - to se zgodi v vrstici 20. Do tukaj je vse podobno kot pri križanju kazalcev. Razlika je v tem, da se med sprehodom čez tabelo primerjata vsaka dva sosednja elementa. V primeru, da je desni element manjši od levega, torej da nista urejena po velikosti, se zamenjata (vrstici 9 in 15). S tem se doseže to, da je v vsakem trenutku skrajno levi element zgornjega dela najmanjši, ter da je skrajno desni element spodnjega dela največji. Ko se kazalca srečata, torej ko je $i == j$, se naredijo še zadnje primerjave in zamenjave, ki zagotavljajo, da je prvi element desnega podzaporedja najmanjši ter da je zadnji element levega podzaporedja največji.



Slika 3.6: Primer porazdeljevanja algoritma Bsort

Po končanem porazdeljevanju je potrebno obravnavati obe podzaporedji (vrstice od 36 do 50). V primeru, da podzaporedje vsebuje manj kot tri elemente, ni potrebno storiti nič več, saj sta elementa zaradi sprotnega mehurčnega urejanja že v pravem vrstnem redu. Če podzaporedje vsebuje tri elemente, ga je trivialno urediti z eno samo primerjavo in zamenjavo, saj vemo, da je skrajno levi element najmanjši oz. da je skrajno desni element največji. Podzaporedje je prav tako že urejeno, če med sprehodom preko tabele ni bila opravljena nobena mehurčna zamenjava. V ta namen sta se v tretji vrstici nastavili dve zastavici, za vsakega od delov ena. Sprožili sta se takrat, ko sosednja elementa nista bila urejena in je bila opravljena zamenjava. Nadaljnje porazdeljevanje je torej potrebno le v primeru, da ima zaporedje več kot 3 elemente ter da je bila zastavica med prehodom tabele sprožena. Tedaj nam preostane le še rekurzivna ureditev spodnjega oz. zgor-

njega dela, izostaneta le pivot in njegov levi sosed, saj za njiju vemo, da sta že na pravem mestu.

Celoten algoritem Bsort-a prikazuje izvorna koda 3.6.

Izvorna koda 3.6: Bsort

```
1  void bsort(Item a[], int left, int right, int mid_key) {
2    if(right <= left) return;
3    int left_flag = 0, right_flag = 0, flag = 1;
4    int l = left, r = right; Item p = a[mid_key];
5
6    while(flag) {
7      while(less(a[l], p) && l != r) {
8        if(l != left && less(a[l], a[l-1])) {
9          exch(a, l-1, l); left_flag = 1;
10       }
11       l++;
12     }
13     while(lessorequal(p, a[r]) && l != r) {
14       if(r != right && less(a[r+1], a[r])) {
15         exch(a, r+1, r); right_flag = 1;
16       }
17       r--;
18     }
19
20     if(l != r) exch(a, l, r);
21     else {  //ce sta kazalca prekrizana
22       if(lessorequal(p, a[r])) {
23         if(less(a[r+1], a[r]) && r < right) {
24           exch(a, r+1, r); right_flag = 1;
25         }
26       }
27       else {
28         if(less(a[l], a[l-1])) {
29           exch(a, l-1, l); left_flag = 1;
30         }
31         if(less(a[l-1], a[l-2]) && l > 1) exch(a, l-1, l-2);
32       }
```

```
33        flag = 0;
34      }
35    } // konec while(flag)
36
37    int size = l-left;
38    if(size > 2 && left_flag) {
39      if(size == 3) {
40        if(less(a[left+1], a[left])) exch(a, left+1, left);
41      }
42      else bsort(a, left, l-2, floor((left+l-2)/2));
43    }
44
45    size = right-r+1;
46    if(size > 2 && right_flag) {
47      if(size == 3) {
48        if(less(a[r+2], a[r+1])) exch(a, r+1, r+2);
49      }
50      else bsort(a, r+1, right, floor((r+1+right)/2));
51    }
52  }
```

Analiza

Avtor algoritma, Roger L. Wainwright, je v svojem članku [14] brez kakršnekoli teoretične analize trdil, da Bsort za vsako vhodno zaporedje elementov potrebuje največ $\mathcal{O}(n \log n)$ primerjav, za (delno) že urejena zaporedja pa porabi samo $\mathcal{O}(n)$ primerjav. V tem primeru algoritem sploh ne bi poznal najslabšega možnega scenarija reda n^2.

Skeptični bralci članka so na revijo hitro naslovili kar nekaj jeznih ugovorov in dokaj preprostih primerov, kjer algoritem porabi $\mathcal{O}(n^2)$ časa, npr. dvakrat ponovljeno urejeno zaporedje $\{1 \ldots n\, 1 \ldots n\}$. Avtorja so opozorili tudi na napako v izvorni kodi algoritma, kjer je sprehajalni indeks na nekem mestu lahko ušel izven zaporedja. Avtor je kasneje pojasnil, da je algoritem bil specializiran za (delno) urejena zaporedja, kjer dejansko ni prišlo do najslabšega možnega primera in posledično časovne zahtevnosti reda n^2.

Tudi trditev, da algoritem za vsako porazdelitev elementov potrebuje največ $\mathcal{O}(n \log n)$ primerjav, ni bila točna - avtor se je naknadno opravičil, da so bile tukaj mišljene vse porazdelitve *iz njihove testne množice*, ne pa vse porazdelitve na sploh.

Tako Sedgewick kot Bentley sta opozarjala na občutljivost algoritma za hitro urejanje v smislu, da je potrebno biti pri implementaciji res zelo pazljiv. V literaturi je namreč dokumentiranih veliko poizkusov izboljšav algoritma, ki so vodili do kvadratne časovne zahtevnosti. Tudi za Bsort bi lahko rekli, da je eden izmed "zgrešenih" primerov. Pokazalo se je, da je lepša in enostavnejša implementacija ponavadi v praksi hitrejša in učinkovitejša.

Poglavje 4

Hitro urejanje na GPE

Algoritem za hitro urejanje je zaradi njegove narave možno preurediti v vzporedno obliko, primerno za izvajanje na arhitekturi CUDA. Posamezna porazdeljevanja zaporedij je sicer težje paralelizirati, ko pa se zaporedje enkrat razdeli na več podzaporedij, lahko nadaljnja urejanja potekajo istočasno, saj so podzaporedja med seboj neodvisna.

4.1 CUDA

CUDA je paralelna arhitektura oz. računski pogon, ki omogoča večjo računsko učinkovitost s tem, da izkoristi moč grafičnih procesorskih enot (GPE). Razvilo jo je podjetje NVIDIA leta 2006, od takrat pa je prisotna v mnogih področjih, kot so astronomija, biologija, kemija, fizika, podatkovno rudarjenje itd. Z uporabo običajnih visoko-nivojskih programskih jezikov in razširitev CUDE lahko aplikacije svoja opravila poganjajo na obeh procesnih enotah. Zaporedna opravila se izvedejo na CPE, opravila, ki jih je mogoče učinkovito paralelizirati, pa se pošlje na GPE. GPE ima arhitekturo, prilagojeno za izvajanje velikega števila niti hkrati, za razliko od CPE, kjer je število niti manjše (odvisno od števila jeder). Pri GPE je število tranzistorjev, namenjenih dejanskemu procesiranju podatkov, veliko večje od števila tranzistorjev v pomnilniškem in kontrolnem delu procesorja [20].

Izvajanje programa na GPE lahko opišemo v štirih korakih:

1. vhodni podatki se prepišejo iz glavnega pomnilnika v pomnilnik GPE
2. CPE poda GPE navodila za izvajanje
3. GPE paralelno izvede opravila
4. rezultat se prepiše iz pomnilnika GPE nazaj v glavni pomnilnik.

Programska koda, ki uporablja CUDO, sestoji iz običajne zaporedne kode in iz ščepcev (ang. *kernel*). Zaporedna koda se izvaja na CPE in med drugim vsebuje ukaze za prenos podatkov iz glavnega pomnilnika na pomnilnik GPE in nazaj, ter za klice ščepcev. Ščepci se izvajajo na GPE, naenkrat se lahko izvaja samo en, znotraj njega teče veliko število niti, ki so razporejene v bloke. Bloki so med seboj popolnoma neodvisni, lahko se izvajajo hkrati ali zaporedno. En blok lahko vsebuje največ 1024 niti, te pa med seboj komunicirajo preko skupnega pomnilnika.

4.2 Osnovna implementacija

Naloge se lotimo tako, da najprej pripravimo podatke za urejanje in jih prepišemo na CUDA omogočeno napravo. Nato v zanki kličemo ščepec na GPE, ki nam zaporedje porazdeli, na sklad pa potisne mejne vrednosti, ki določajo nova podzaporedja za urejanje. Zanka se ponavlja, dokler na skladu še kakšno podzaporedje čaka na urejanje. Za urejanje vsakega podzaporedja pokličemo nov ščepec, ki se izvaja v svojem toku (ang. *stream*). Tok v CUDI je množica ukazov, ki se na napravi izvede v določenem zaporedju, ukazi iz različnih tokov pa se lahko prekrivajo oz. se izvajajo istočasno.

Algoritem

Naša osnovna implementacija algoritma za hitro urejanje na CUDI se začne z vhodno funkcijo. Najprej v pomnilniku dodelimo ustrezno velik prostor za vse podatke, nato jim določimo dejanske vrednosti. Ukaz `cudaMemcpy` začne s prepisovanjem podatkov na napravo. Narejen je tako, da blokira celoten sistem, dokler prenos ni končan. Nato poženemo dejansko urejanje in

počakamo napravo, da konča z delom. Potem podatke prepišemo iz naprave in jih poljubno uporabimo. Za konec še sprostimo zaseden prostor v napravi in v glavnem pomnilniku.

Psevdokodo vhodne točke algoritma prikazuje izvorna koda 4.1.

Izvorna koda 4.1: Psevdokoda vhodne točke v program

```
1  int main(int argc, char **argv) {
2    size_t size = NUM_ITEMS * sizeof(int);
3
4    memory_allocation(size); // dodeli pomnilnik
5
6    init_data(); // inicializiraj vhodne podatke
7
8    // prepisi podatke na napravo
9    cudaMemcpy(d_values,h_values,size,cudaMemcpyHostToDevice);
10
11   launch_quicksort(d_values); // zazeni quicksort
12
13   // prepisi podatke iz naprave
14   cudaMemcpy(h_values,d_values,size,cudaMemcpyDeviceToHost);
15
16   // uporabi podatke...
17
18   // sprosti pomnilnik
19   free(h_values); cudaFree(d_values);
20 }
```

Metoda `launch_quicksort`, ki smo jo klicali v 11. vrstici algoritma, se izvaja še na CPE. Znotraj te metode pa najprej prvič pokličemo ščepec `quicksort`, ki se izvede na GPE in nam na sklad potisne mejne indekse podzaporedij, ki smo jih dobili ob porazdeljevanju originalnega zaporedja. Nato ponavljamo zanko, kjer vsakič najprej počakamo, da se vsa urejanja prejšnje stopnje izvršijo. Potem prepišemo sklad iz naprave ter obdelamo vsa podzaporedja, ki so tam zabeležena. Za vsako podzaporedje kreiramo

nov tok, v katerem se bo izvajal ščepec. Ko je sklad prazen, smo z urejanjem končali.

Psevdokodo začetne metode urejanja prikazuje izvorna koda 4.2.

Izvorna koda 4.2: Psevdokoda začetne metode urejanja

```
__host__ void launch_quicksort(int *values) {
  // zazeni kernel, da se sklad napolni
  quicksort<<<1,1>>>(values, 0, NUM_ITEMS-1);

  // dokler je se kaj za urediti, ponavljaj
  while(1) {
    // pocakaj, da se vsa urejanja koncajo
    cudaDeviceSynchronize();

    // prepisi sklad iz naprave
    Stack stack_copy = copy_from_device(stack);

    // ce je sklad prazen, potem je konec urejanja
    if(stack_copy.size() == 0) break;

    // dokler sklad ni prazen, vzemi s sklada in pozeni novo
       urejanje v novem toku
    while(stack_copy.size() > 0) {
      StackElement el = stack_copy.pop();
      cudaStream_t s;
      cudaStreamCreate(&s);
      quicksort<<<1,1,0,s>>>(values, el.left, el.right);
    }
  }
}
```

Sedaj pride na vrsto še ščepec, v katerem se zaporedja dejansko porazdelijo, na sklad pa se dodajo nova podzaporedja, katera je potrebno še naknadno urediti.

Psevdokodo ščepca hitrega urejanja prikazuje izvorna koda 4.3.

Izvorna koda 4.3: Psevdokoda ščepca hitrega urejanja

```
1 // Kernel funkcija
2 __global__ void quicksort(int* values, int left, int right) {
3   int nleft, nright;
4
5   // porazdeli podatke, vrni nove meje v nleft in nright
6   partition(values, left, right, &nleft, &nright);
7
8   // ce je treba urediti podzaporedje, meje porini na sklad
9   if(left < nright)
10    stack.push(left, nright);
11  if(nleft < right)
12    stack.push(nleft, right);
13 }
```

Vsak korak urejanja se lahko začne izvajati šele po tem, ko je prejšnji korak dokončan, torej vsak korak traja toliko kot njegova najpočasnejša operacija. Vsak klic ščepca, ki dejansko opravi delo, mora biti izveden s strani CPE. To pa pomeni, da je treba po vsakem koraku prenesti podatke iz naprave na gostitelja, da ta sploh ve, kaj mora še postoriti. Komunikacija med napravo in gostiteljem je lahko že sama po sebi zelo kompleksna in zahteva več kode kot samo urejanje. Zaradi enostavnosti smo vso to komunikacijo predstavili s skladom, ki pa je implementiran kot preprosta tabela mejnih vrednosti. Ob vsakem klicu ščepca smo zraven podali števec, s pomočjo katerega smo lahko garantirali, da ne bi več ščepcev v svojih tokovih pisalo po istih naslovih. Vsakokratno prepisovanje sklada iz naprave na gostitelja je dolgotrajno opravilo. Poleg tega vemo, da se mora vsaka faza končati, preden se lahko začne naslednja, kar pomeni, da vsekakor nismo prišli do optimalne rešitve.

Z najnovejšo verzijo CUDE in grafičnih kartic NVIDIA, ki imajo računsko zmožnost 3.5, je problem mogoče rešiti veliko bolj učinkovito, hitro in tudi enostavno. Razlog za to tiči v novi tehnologiji, imenovani *dinamična paralelizacija* (ang. *"dynamic parallelism"*). Le-ta omogoči grafičnemu procesorju veliko bolj avtonomno in od CPE neodvisno obnašanje. Med izvaja-

njem lahko GPE samemu sebi po potrebi dodeljuje novo delo znotraj ščepca, omogočeni so tudi rekurzivni klici. GPE bi torej lahko sam pognal urejanje obeh podzaporedij, takoj ko bi končal s porazdeljevanjem, potreba po vsakokratnem prepisovanju podatkov iz naprave na gostitelja bi odpadla, tudi sama koda bi bila lepša in lažje berljiva, predvsem pa hitrejša [21].

4.3 GPU-Quicksort

Cederman in Tsigas [16] sta se reševanja problema lotila na bolj izpopolnjen način in implementirala algoritem "GPU-Quicksort". V vsakem koraku iz zaporedja dobimo dve podzaporedji. Po nekem številu korakov bo število podzaporedij tolikšno, da bo vsak blok lahko dobil v obdelavo svoje podzaporedje. Do takrat morajo bloki med seboj sodelovati in delati na istih zaporedjih. Avtorja sta zato algoritem razdelila na dve fazi. V prvi fazi več blokov obdeluje različne kose istega zaporedja, za to pa je potrebna primerna medsebojna komunikacija in sinhronizacija. V drugi fazi vsak blok dobi svoje podzaporedje, potrebe po komunikaciji več ni, faza lahko v celoti teče samo na GPE. Avtorja sta se odločila, da bo algoritem kratka zaporedja urejal z alternativno metodo urejanja. Poleg tega njun algoritem ne izvede urejanja neposredno na vhodnem zaporedju, temveč porabi dodaten prostor. V vsakem koraku se podatki preberejo iz primarnega pomnilnika in zapišejo v pomožnega, vlogi pa se iz koraka v korak zamenjata.

Porazdelitev

Zaporedje se v prvi fazi razdeli na m enako velikih odsekov, kjer je m število blokov, ki so na voljo. Vsak blok nato dobi v obdelavo en odsek. Cilj porazdeljevanja je premakniti elemente v pomožen pomnilnik tako, da bodo od pivota manjši elementi na njegovi levi strani, elementi, večji od pivota pa na njegovi desni strani. Sedaj naletimo na problem učinkovite sinhronizacije niti - kako vsaki niti povedati, na katero mesto v pomožnem pomnilniku lahko piše. Ena od rešitev je, da vsaka nit prebere element in se z vsemi

ostalimi nitmi sporazume, kaj je prebrala. Vsaka nit nato izračuna, koliko je takih niti, ki želijo pisati na levo oz. desno od pivota in katerih *id* številka je manjša od njihove. Tako vsaka nit točno ve, koliko prostora bodo porabile niti z manjšim *id*-jem od njenega, torej ve, kam lahko varno zapiše svoj element. V prvi fazi, kjer več blokov obdeluje isto zaporedje, moramo seveda upoštevati niti iz vseh blokov. Na koncu v razmak med obe nastali zaporedji premaknemo pivotni element. Če sta podzaporedji dovolj kratki, ju uredimo z alternativno metodo. Če je nastalo že dovolj veliko število neodvisnih podzaporedij, preidemo v drugo fazo algoritma, kjer vsak blok dobi svoje podzaporedje, potrebe po komunikaciji med bloki pa več ni. Sinhronizacija med nitmi znotraj bloka je seveda še vedno potrebna in poteka na enak način kot v prvi fazi.

Poglavje 5

Primerjava algoritmov in porazdelitev

5.1 Okolje za testiranje

Strojna in programska oprema

Teste smo izvajali na sistemu z dvojedrnim procesorjem Intel Core 2 Duo E8400 s frekvenco ure 3GHz in 6MB L2 predpomnilnika. Na voljo smo imeli 4 GB osnovnega pomnilnika in grafično kartico NVIDIA GeForce 9600 GT s frekvenco ure 1625 MHz in 512 MB osnovnega pomnilnika.

Vsi opisani algoritmi so bili testirani v takšni obliki, kot so bili predstavljeni. Za večino seveda obstajajo veliko boljše implementacije z mnogimi izboljšavami, vendar bi nas te samo motile, saj je bil glavni namen tega dela primerjava različnih porazdelitev. Algoritmi so bili implementirani v jeziku C z dodatki CUDE, prevedeni pa s prevajalnikom GNU gcc verzije 3.4.4.

Generiranje vhodnih distribucij

Testni primeri so vsebovali pet različnih distribucij: naključno, Gaussovo, konstantno, že urejeno ter "porazdelitev z vedri". V slednji je bilo veliko ponavljajočih elementov - prvih 20 elementov je zavzelo vrednosti iz intervala [0, 20], v drugem "vedru" so bile vrednosti iz intervala [20, 40] in tako naprej,

po stotih elementih pa se je princip ponovil od začetka.

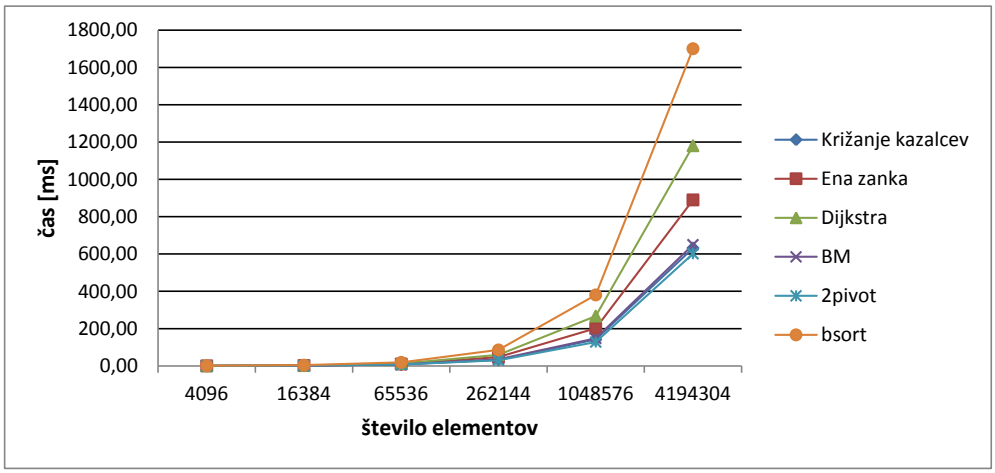
Vsak algoritem smo testirali na zaporedjih dolžine 2^k, k smo povečevali od 10 do 22, zaporedja so torej vsebovala od tisoč do 4 milijone elementov. Da bi dobili čim bolj točne podatke, smo vsak test pognali stokrat in izračunali povprečne vrednosti. Pred vsakim generiranjem novega zaporedja je bilo potrebno spremeniti "seme" naključnega generatorja, drugače bi vsa zaporedja v bistvu ostala enaka, verodostojnost podatkov pa bi tako bila manjša.

5.2 Primerjava

5.2.1 Primerjava algoritmov in porazdelitev na CPE

Naključna distribucija

Graf na sliki 5.1 prikazuje čas urejanja zaporedja naključnih elementov v od-

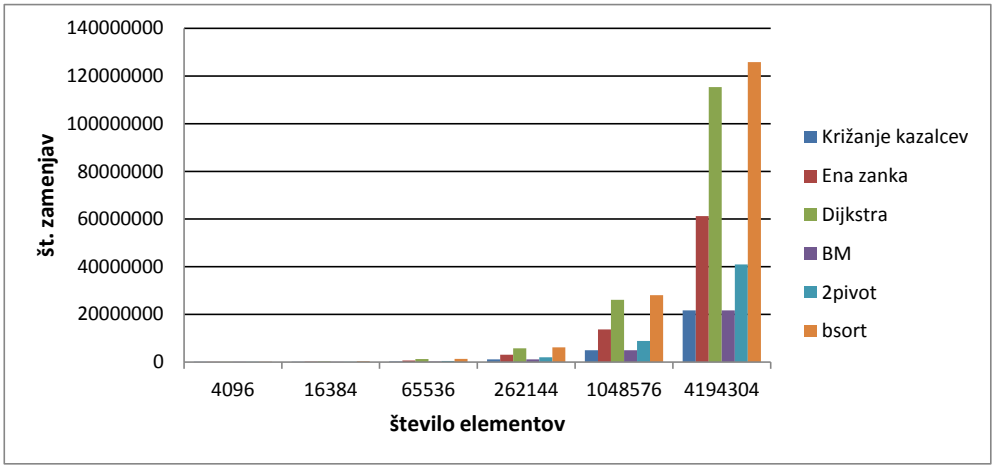


Slika 5.1: Primerjava časov urejanja zaporedja naključnih elementov

visnosti od števila elementov. Rezultati so takšni, kot smo pričakovali - malenkostno vodstvo prevzame algoritem z dvo-pivotno porazdelitvijo, sledita mu Bentley-McIlroyeva in originalna implementacija s križanjem kazalcev, na koncu se zvrstijo še enostavna Lomutova eno-zančna rešitev ter Dijkstrov algoritem in bsort. Pri krajših zaporedjih razlike še niso tako očitne,

ko pa se približamo milijon elementom, je najpočasnejši bsort skoraj trikrat počasnejši od najhitrejšega dvo-pivotnega urejanja.

Vse našteto se pokaže tudi pri številu primerjav. Algoritem 2pivot jih opravi najmanj, bsort skoraj dvakrat več kot 2pivot, ostali algoritmi pa so približno primerljivi med sabo in opravijo malo več primerjav kot 2pivot. Daleč najmanj zamenjav opravita originalna rešitev in BM, ki sta si tako ali tako zelo podobna, BM naredi dodatne zamenjave samo za pivotu enake elemente. 2pivot naredi približno dvakrat več zamenjav od njiju, kar je bilo razvidno tudi iz analize. Največ zamenjav opravita bsort in Dijkstrov algoritem. Pri bsortu se pozna, da vmes opravlja še mehurčne zamenjave, za Dijkstrov algoritem pa smo rekli, da naredi zamenjavo za vsak element, ki je od pivota različen, takih je v naključni razporeditvi elementov seveda veliko. Število zamenjav je razvidno iz grafa na sliki 5.2.



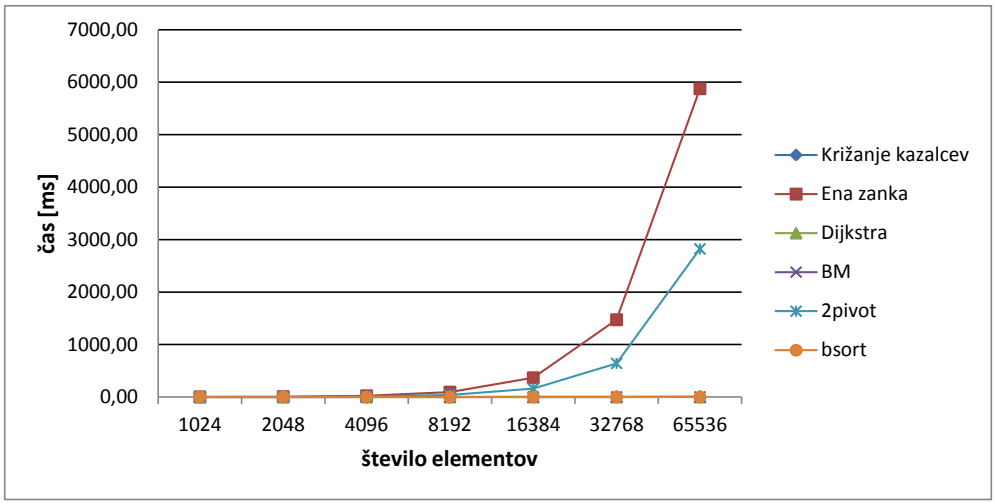
Slika 5.2: Primerjava št. zamenjav pri urejanju zaporedja naključnih elementov

Rezultati pri naključni distribuciji elementov nam dajo največ informacije o obnašanju algoritma. Ostale porazdelitve v praksi niso pogoste, a vendar si poglejmo, kako se na njih odzivajo naši algoritmi.

Pri Gaussovi porazdelitvi so rezultati praktično enaki kot pri naključni, tako da se v podrobnejšo analizo ne bomo spuščali.

Konstantna distribucija

Graf na sliki 5.3 prikazuje čas urejanja zaporedja, sestavljenega iz samih

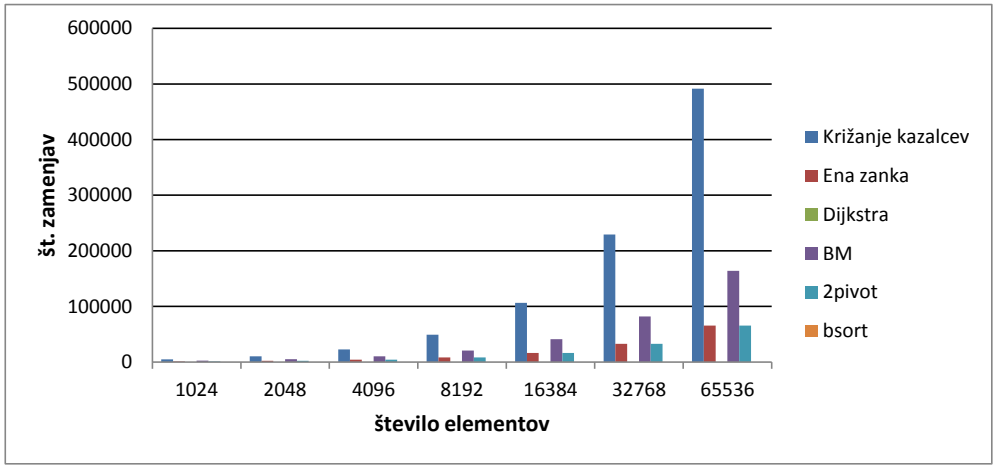


Slika 5.3: Primerjava časov urejanja zaporedja samih enakih elementov

enakih elementov. Tukaj pridejo do izraza vsi algoritmi, ki so posebej prilagojeni za takšna zaporedja. Dijkstrov algoritem se izkaže najbolje, saj dejansko samo preleti zaporedje in ne naredi nobene zamenjave, ker so vsi elementi enaki pivotu. Bsort prav tako ne opravi nobene zamenjave in tudi konča že v prvem preletu zaporedja, saj zazna, da so vsi elementi že na svojem mestu. BM sicer opravlja zamenjave po nepotrebnem, ker pivotu enake elemente prestavlja na stran, potem pa spet na sredino, vendar se vseeno dobro obnese in z urejanjem konča po enem preletu zaporedja. Originalna rešitev zamenja vsak par elementov, na koncu pa je zahtevnost algoritma kljub temu reda $n \log n$, ker se kazalca srečata na sredini zaporedja, torej se zaporedje vsakič razdeli na pol. To ne velja za Lomutov in 2pivot algoritem - v vsakem koraku se dejansko obdela samo en oz. dva elementa in tako pridemo do kvadratne zahtevnosti. Število zamenjav je razvidno iz grafa na sliki 5.4.

Distribucija z vedri

Pri distribuciji z vedri so razmerja med časi algoritmov podobna tistim pri

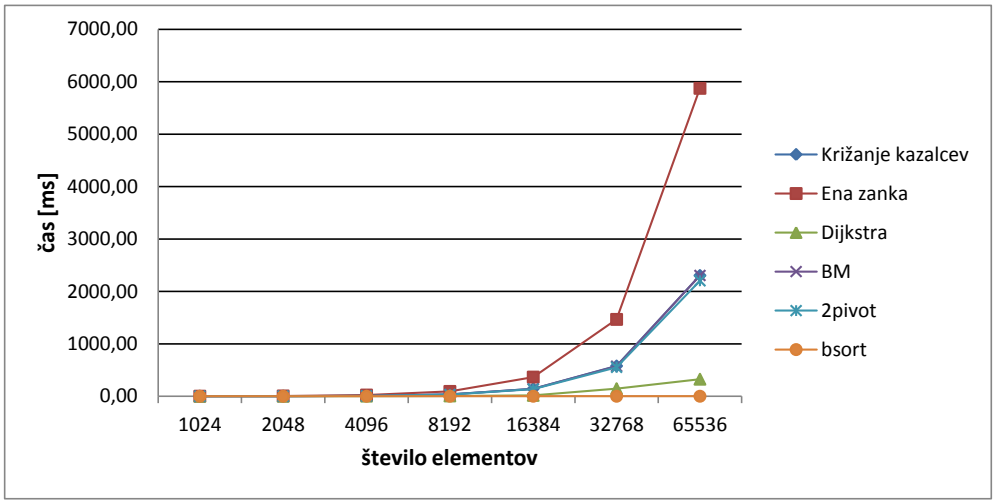


Slika 5.4: Primerjava št. zamenjav pri urejanju zaporedja samih enakih elementov

konstantni distribuciji. Najbolje se izkažeta Dijkstra in BM, saj se pri tej porazdelitvi elementi zelo ponavljajo. To pa najmanj leži Lomutovemu algoritmu, ki je v tej kategoriji najpočasnejši.

Urejena distribucija

Po pričakovanjih se nad že urejenimi zaporedji najbolje znajde algoritem bsort, ki je za to posebej specializiran. Za ostale algoritme je to najslabši možni primer, tako da je njihova časovna zahtevnost reda n^2. Presenetljivo hiter je Dijkstrov algoritem, za katerega se izkaže, da že urejeno zaporedje zanj ne predstavlja najslabšega scenarija. Kot smo že omenili, Dijkstrov algoritem ponavadi naredi veliko več zamenjav kot kakšen drug algoritem, saj zamenja vsak od pivota različen element. Prav to pa se v primeru urejanja že urejenega zaporedja izkaže za dobro lastnost, ker s tem zaporedje nekoliko premeša. Podzaporedja, ki pri porazdeljevanju nastanejo, zato niso velikosti 0 in $n-1$ kakor pri ostalih algoritmih in tudi število rekurzivnih klicev je manjše. Rezultati so prikazani na sliki 5.5.



Slika 5.5: Primerjava časov urejanja zaporedja že urejenih elementov

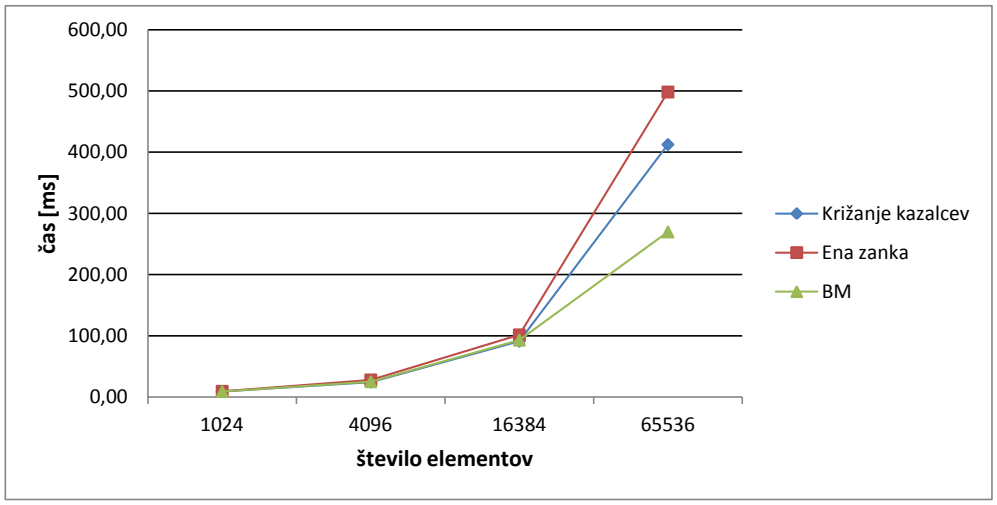
5.2.2 Primerjava porazdelitev na GPE

Kot smo povedali že v razdelku 4.2, naša implementacija hitrega urejanja na GPE ni ravno optimalna. V primerjavi s CPE algoritmi je pri urejanju zaporedja, ki vsebuje 65 tisoč naključnih elementov, naš algoritem od 35 do 50-krat počasnejši. Zaradi tega bomo, kar se tiče primerjav na GPE, med seboj primerjali samo različne porazdelitve naše implementacije algoritma.

Največjo težo pri primerjavi porazdelitev na GPE ima število "rekurzivnih"klicev. Klici seveda dejansko niso rekurzivni, saj strojna oprema in verzija CUDE, ki nam je bila na voljo, tega ni podpirala, tako da v bistvu govorimo o klicih ščepcev na GPE. Ob vsakem klicu je potrebno iz naprave na gostitelja in nazaj prepisovati podatke, kar je najdražja operacija v algoritmu.

Za boljšo predstavo podajmo en primer. Med primerjanjem porazdelitev na CPE smo zaradi lepšega izgleda kode in manjše možnosti napak za štetje primerjav napisali metodo `less`, ki je nadomestila makro. V tej metodi smo enostavno lahko povečali števec primerjav. Vendar pa skakanje v metodo in nazaj ni poceni operacija, kar se je še kako poznalo na učinkovitosti algoritma - čas urejanja se je povečal tudi za 50%.

Naši algoritmi na GPE so torej veliko počasnejši kot na CPE, razmerja med njimi pa ostajajo približno enaka. BM se je odrezal najbolje, sploh na distribucijah, kjer je bilo več ponavljajočih elementov, Lomutova enostavna eno-zančna rešitev pa je bila še skoraj dvakrat počasnejša od BM. Meritve so zbrane v grafu na sliki 5.6.



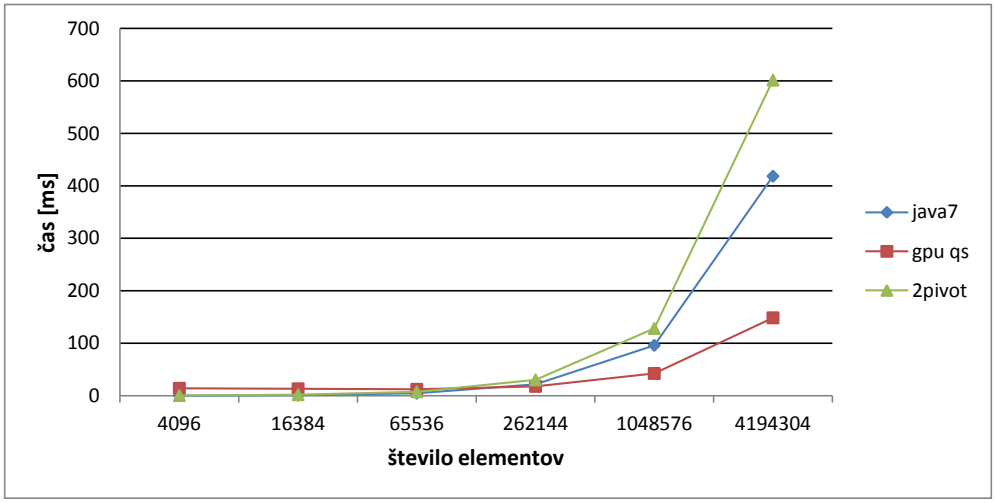
Slika 5.6: Primerjava časov urejanja zaporedja naključnih elementov na GPE

5.2.3 Primerjava med algoritmi na CPE in GPE

Za konec bomo primerjali še najhitrejši algoritem 2pivot z GPE algoritmom GPU-Quicksort. Ob stran jima bomo postavili tudi izpopolnjeno verzijo našega osnovnega dvo-pivotnega urejanja, ki je postala del standardne knjižnice Jave 7.

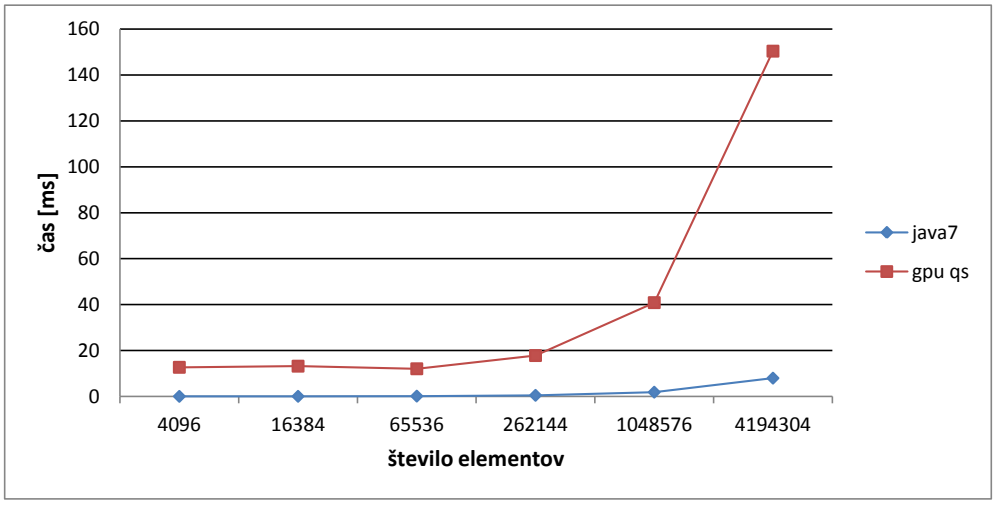
Rezultati so kar presenetljivi: osnovni 2pivot za urejanje štirih milijonov naključnih števil potrebuje 50% več časa kot java7. Le-ta pa pri istem vhodu porabi več kot dvakrat toliko časa kot gpu-qs, pri daljših zaporedjih bi bila razlika še večja. Graf na sliki 5.7 bolj nazorno prikazuje rezultate meritev.

Še ena zanimivost se pokaže pri urejanju že urejenega zaporedja. Algoritem 2pivot se v tem primeru z ostalima dvema sploh ne more primerjati, tako da ga tokrat izpustimo. Oba preostala algoritma sta zelo hitra, vendar



Slika 5.7: Primerjava časov urejanja zaporedja naključnih elementov na GPE in CPE

pa je java7 očitno zelo prilagojen za že urejena zaporedja, saj jih uredi res bliskovito hitro. Za primerjavo lahko povemo, da je algoritem za urejanje v Javi 6 že urejeno zaporedje štirih milijonov elementov uredil v 260 ms, java7 pa to stori v 8 ms. Graf na sliki 5.8 prikazuje čas urejanja že urejenega zaporedja.



Slika 5.8: Primerjava časov urejanja že urejenega zaporedja na GPE in CPE

Poglavje 6

Zaključek

Algoritem za hitro urejanje je kljub dolgi prisotnosti na računalniški sceni še vedno nepogrešljiv del knjižnice marsikaterega programskega jezika. Na lestvici najboljših in najhitrejših algoritmov za urejanje podatkov že cel čas trmasto vztraja pri vrhu. Temelji na metodi snovanja algoritmov *deli in vladaj*. Najprej si izbere poseben element, imenovan *pivot*. Vhodno zaporedje nato razdeli na dve krajši podzaporedji, tako da v prvo prestavi vse elemente, ki so manjši od pivota, v drugo pa vse elemente, ki so večji od pivota. Število pivotnih elementov in podzaporedij je odvisno od posamezne izvedbe. Podzaporedja rekurzivno uredi, nato iz dobljenih rešitev trivialno sestavi originalno rešitev.

Različne implementacije algoritma se med seboj razlikujejo predvsem po načinu porazdeljevanja zaporedij. V povprečju so vse reda $\mathcal{O}(n \log n)$ oz. v najslabšem primeru reda $\mathcal{O}(n^2)$, zato nas zanimajo deleži, ki jih k celotni časovni zahtevnosti prinesejo konstante.

Hitro urejanje s križanjem kazalcev je prvotna verzija algoritma, ki jo je razvil Hoare. Levi in desni kazalec iz robov zaporedja potujeta vsak v svojo smer in pri tem zamenjata vsaka dva elementa, ki se nahajata ravno v napačnih delih zaporedja. Povprečna časovna zahtevnost algoritma je približno $1.386n \log n$.

Lomutovo eno-zančno hitro urejanje je enostavna izvedba algoritma z

enim samim prehodom zanke. Elementi, ki so večji od pivota, ostanejo na svojem mestu, če pa je trenutni element manjši od pivota, ga je potrebno prestaviti v levo podzaporedje. V vsakem koraku porazdeljevanja se opravi natanko ena primerjava za vsak element razen pivota. Pri urejanju že urejenega zaporedja algoritem naredi veliko nepotrebnih zamenjav. Pri urejanju zaporedja s samimi enakimi elementi se prav tako zamenja vsak element, časovna zahtevnost algoritma pa se zaradi neenakomerne porazdelitve izrodi v $\mathcal{O}(n^2)$.

Dijkstrovo osnovno tro-smerno hitro urejanje tudi temelji na samo enem prehodu zanke, kjer se vsak element primerja s pivotom in prestavi v ustrezno podzaporedje. Če je element pivotu enak, potem sodi v srednji del, kjer se v bistvu že nahaja in tam ostane. V nasprotnem primeru se prestavi v spodnji oz. zgornji del. Opazimo lahko, da se zamenja vsak element, ki je od pivota različen. V primeru, ko v zaporedju ni veliko ponovljenih elementov (npr. v naključnem zaporedju), se torej algoritem ne obnese prav dobro. Presenetljivo dobro pa se algoritem obnese pri urejanju že urejenega zaporedja, saj zaradi velikega števila "nepotrebnih" zamenjav zaporedje nekoliko premeša in se s tem izogne najslabšemu primeru.

Bentley-McIlroyeva porazdelitev je v bistvu izboljšava porazdelitve s križanjem kazalcev. Njena prednost je v obravnavi elementov, ki so enaki pivotu. Med porazdeljevanjem jih prestavlja na levi in desni rob zaporedja, na koncu pa nazaj na sredino. S tem sicer zapravlja čas, vendar izgubljeno hitro nadoknadi, saj se s srednjim delom ni potrebno več ukvarjati. Če pivotu enakih elementov v zaporedju ni, se ne naredi nobena dodatna zamenjava in porazdeljevanje je podobno prvotni Hoarovi verziji. Avtorja sta z empiričnimi testi dognala, da je časovna zahtevnost njunega algoritma približno $1.2n \log n$.

Dvo-pivotno hitro urejanje uporablja dva pivota in zaporedje razdeli na 3 podzaporedja. Vsak element je lahko manjši od levega pivota, večji od desnega pivota, lahko pa sodi med njiju. Izmed vseh testiranih algoritmov se je ta algoritem pri urejanju zaporedja naključnih elementov izkazal za naj-

hitrejšega. V podrobni analizi je moč zaslediti, da obstaja velika verjetnost, da se elementi, ki so večji od desnega pivota, že nahajajo na pravih mestih, kar algoritem s pridom izkorišča. Njegova časovna zahtevnost je približno $1.317n \log n$.

Algoritem bsort je posebej prilagojen za urejanje (delno) že urejenih zaporedij. Klasično porazdeljevanje s križanjem kazalcev poskuša združiti z urejanjem z navadnimi zamenjavami. Medtem, ko kazalca potujeta vsak v svojo smer, po potrebi opravljata še mehurčne zamenjave. Če po prehodu zaporedja ni bila opravljena nobena zamenjava, je urejanje končano. Algoritem se torej dobro odreže nad že urejenimi zaporedji in zaporedji samih enakih elementov. Pri zaporedjih naključnih elementov pa se je izkazal za najslabšega izmed testiranih algoritmov.

Arhitektura CUDA omogoča reševanje splošnih problemov z uporabo grafičnih procesorjev, ki so specializirani za računsko zahtevne paralelne operacije. Naša osnovna implementacija hitrega urejanja na CUDI ni konkurenčna ostalim obstoječim rešitvam, saj porabi preveč časa za komunikacijo med CPE in GPE ter tudi ne izkorišča celotnega potenciala, ki ga ponuja CUDA. Algoritem GPU-Quicksort se pri tem izkaže veliko bolje, pri urejanju zaporedja naključnih elementov za njim zaostaja celo uradni algoritem za urejanje v Javi 7. Če k temu prištejemo še dejstvo, da se urejanje dogaja pretežno na GPE, tako da lahko na CPE vmes izvajamo druge operacije, ne bi več smelo biti dvoma pri vprašanju, ali se urejanje na GPE sploh splača. Seveda pa je vse odvisno od posameznega algoritma in njegove dovršenosti.

Slike

2.1 Asimptotična zgornja meja 6

3.1 Primer porazdeljevanja s križanjem kazalcev 15
3.2 Primer eno-zančnega porazdeljevanja 23
3.3 Primer osnovnega tro-smernega porazdeljevanja 27
3.4 Primer Bentley-McIlroyevega tro-smernega porazdeljevanja . . 29
3.5 Primer dvo-pivotnega porazdeljevanja 34
3.6 Primer porazdeljevanja algoritma Bsort 45

5.1 Primerjava časov urejanja zaporedja naključnih elementov . . 58
5.2 Primerjava št. zamenjav pri urejanju zaporedja naključnih elementov 59
5.3 Primerjava časov urejanja zaporedja samih enakih elementov . 60
5.4 Primerjava št. zamenjav pri urejanju zaporedja samih enakih elementov 61
5.5 Primerjava časov urejanja zaporedja že urejenih elementov . . 62
5.6 Primerjava časov urejanja zaporedja naključnih elementov na GPE 63
5.7 Primerjava časov urejanja zaporedja naključnih elementov na GPE in CPE 64
5.8 Primerjava časov urejanja že urejenega zaporedja na GPE in CPE 64

Literatura

[1] Robert Sedgewick and Kevin Wayne. *Algorithms*. Princeton University, 4. edition, 2011.

[2] Boštjan Vilfan. *Osnovni algoritmi*. Fakulteta za računalništvo in informatiko, 2. edition, 2002.

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

[4] Edsger W. Dijkstra. *A discipline of programming*. Prentice-Hall, 1976.

[5] Donald E. Knuth. *The art of computer programming*, volume 3. Addison Wesley, 1998.

[6] Jon Bentley. *Programming Pearls*. Addison Wesley, 2 edition, 2000.

[7] C.A.R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, vol. 4: pp. 321, 1961.

[8] C.A.R. Hoare. Quicksort. *Computer Journal*, vol. 5: pp. 10–15, 1962.

[9] Robert Sedgewick. Implementing Quicksort programs. *Communications of the ACM*, vol. 21: pp. 847–857, 1978.

[10] R.C. Singleton. An efficient algorithm for sorting with minimal storage: Algorithm 347. *Communications of the ACM*, vol. 12: pp. 186–187, 1969.

[11] Jon L. Bentley and M. Douglas McIlroy. Engineering a Sort Function. *Software: Practice and Experience*, vol. 23: pp. 1249–1265, 1993.

[12] Vladimir Yaroslavskiy. Replacement of quicksort in java.util.arrays with new dual-pivot quicksort. `http://permalink.gmane.org/gmane.comp.java.openjdk.core-libs.devel/2628`, September 2009.

[13] Sebastian Wild and Markus E. Nebel. Average Case Analysis of Java 7's Dual Pivot Quicksort. *Fachbereich Informatik, Technische Universität Kaiserslautern*, 2012.

[14] R. L. Wainwright. A class of sorting algorithms based on quicksort. *Communications of the ACM*, vol. 28: pp. 396–402, 1985.

[15] R. L. Wainwright. Quicksort algorithms with an early exit for sorted subfiles. *Communications of the ACM*, 1987.

[16] Daniel Cederman and Philippas Tsigas. A Practical Quicksort Algorithm for Graphics Processors. *Chalmers University of Technology and Göteborg University*, 2008.

[17] Jurij Mihelič. 50 let Quicksorta. `http://lalg.fri.uni-lj.si/jurij/blog/50-let-quicksorta/`, Januar 2013.

[18] Dutch national flag problem. `http://en.wikipedia.org/wiki/Dutch_national_flag_problem`.

[19] Sebastian Wild. Quicksort Partitioning: Hoare vs. Lomuto. `http://cs.stackexchange.com/questions/11458`, April 2013.

[20] CUDA Programming Guide Version 4.2. `http://docs.nvidia.com/cuda/cuda-c-programming-guide`, 2013.

[21] Stephen Jones. How Tesla K20 speeds Quicksort, a familiar comp-sci code. `http://blogs.nvidia.com/2012/09/how_tesla_k20_speeds_up_quicksort_a_familiar_comp_sci_code/`, September 2012.