Gimnazija Vič

Jure Slak, 3. letnik

Iskanje optimalnega splošnega kompozitnega sortirnega algoritma

raziskovalna naloga na področju računalništva

mentor: Klemen Bajec, prof. mat.
somentor: Gašper Ažman, dipl. mat. (UN)

Ljubljana, 2011

Povzetek

Namen raziskovalne naloge je poiskati optimalen sortirni algoritem. V prvem delu raziskovalne naloge je opisana teorija urejanja, različni sortirni algoritmi in njihovo pričakovano obnašanje na dejanskih podatkih. Postavljeni sta tudi dve hipotezi glede učinkovitosti sortirnih algoritmov. V nadaljevanju je opisana ideja, na kateri temelji kompozitni sortirni algoritem in konfiguracija, s katero nadzorujemo njegovo delovanje. Pomemben del je teoretična izpeljava algoritma, ki s primerjanjem učinkovitosti posameznih sortirnih algoritmov poišče optimalno konfiguracijo za kompozitni sortirni algoritem. Podan je tudi opis implementacije tega algoritma, ki mu sledi predstavitev rezultatov. Rezultati prikazujejo primerjavo učinkovitosti posameznih sortirnih algoritmov na različnih vrstah podatkov. Dobljene rezultate razložimo z vidika teoretičnih osnov in na podlagi naših ugotovitev ovrednotimo hipotezi. Rezultat raziskovalne naloge je učinkovit in izjemno prilagodljiv kompozitni sortirni algoritem, ki se lahko prilagodi tako zmogljivosti strojne opreme kot podatkom, ki jih ureja.

Abstract

The purpose of the research paper is to find an optimal sorting algorithm. In the first part of the paper the theory of sorting, various sorting algorithms and their expected behaviour on actual data are described. Two hypotheses regarding efficiency of sorting algorithms are also set. The next part describes the idea on which the composite sorting algorithm is based and how it is configured. An important part is the theoretical derivation of the algorithm which finds the optimal configuration for the composite sorting algorithm by comparing efficiency of each sorting algorithm. The description of the implementation of this algorithm is also given and followed by presentation of results. Results illustrate the comparison of efficiency for each sorting algorithm on different types of data. The obtained results are then explained in terms of theoretical basics and both hypotheses are evaluated on the basis of our findings. The result of the research paper is an efficient and extremely adaptive composite sorting algorithm, which can adapt to the capabilities of hardware as well to the data being sorted.

Ključne besede: kompozitni sortirni algoritem, prilagodljivo urejanje, optimizacija urejanja

Keywords: composite sorting algorithm, adaptive sorting, sorting optimization

Zahvale

Zahvaljujem se svojemu šolskemu mentorju Klemnu Bajcu za vso pomoč in podporo, ki mi jo je nudil ob izdelavi te seminarske naloge. Posebna zahvala gre tudi zunanjemu mentorju Gašperju Ažmanu za vzpodbudo, vse koristne napotke pri nastajanju naloge, strokovno pomoč in veliko prizadevnost od začetka do konca. Zahvaljujem se tudi koordinatorici raziskovalne dejavnosti Majdi Šajn Stjepić za organizacijske napotke in knjižničarki Lidiji Vidmar za napotke pri navajanju virov. Hvala tudi Mateji Ban za temeljito lektoriranje naloge.

Kazalo

Kazalo slik

Kazalo tabel

Kazalo algoritmov

1 Uvod

Problem urejanja me je zanimal, odkar sem se naučil osnov programiranja. Verjetno zato, ker je bil program, ki je urejal koledar na moji spletni strani pravzaprav prvi uporaben program, ki sem ga napisal. Preden je bil koledar avtomatsko sortiran sem moral vedno paziti, da bom novico ročno vstavil na pravo mesto. S tem programom, ki je bil sicer dolg le nekaj vrstic, saj je klical vgrajeno funkcijo *sort* v PHP-ju, sem si precej olajšal dodajanje novic in dogodkov. Tako sem iz zanimanja do urejanja spoznal veliko različnih sortirnih metod.

Urejanje ali sortiranje na splošno pomeni preurejanje dane množice objektov v nek določen vrstni red. Namen urejanja je ponavadi olajšati kasnejše iskanje pripadnikov take urejene množice. Kot táko je v vsesplošni uporabi in ima velik pomen. Objekti so urejeni v telefonskih imenikih, davčnih registrih, stvarnih kazalih, knjižnicah, slovarjih, skladiščih in skoraj povsod tam, kjer moramo shranjene predmete iskati in dosegati. Že majhne otroke učimo, da stvari spravljajo "v red", in tako z neko vrsto urejanja prihajajo v stik daleč preden se naučijo kaj aritmetike.

1.1 Teoretični uvod

Urejanje je torej pogosto in pomembno opravilo zaradi česar se je je skozi čas razvilo veliko različnih postopkov za čimbolj učinkovito urejanje. Eden izmed razlogov za veliko število postopkov je tudi ta, da ima prav vsak izmed njih neko prednost pred ostalimi in je zaradi tega v določenih primerih primernejši. V nadaljevanju bom opisal nekaj najbolj znanih, vendar jih obstaja še mnogo več.

1.1.1 Definicija algoritma

Definicija 1. Algoritem je končno zaporedje ukazov, ki, če jih ubogamo, opravijo neko nalogo. Zanj velja:

- **Ima podatke.** Množica podatkov je lahko tudi prazna.
- **Vrne rezultat.** Vrne vsaj eno izračunano vrednost ali kaj stori, kot na primer napiše besedilo v datoteko ali na zaslon.
- **Je natančno določen.** Vsak ukaz mora nedvoumno povedati kaj storiti.
- **Se vedno konča.** Končati se mora pri vseh možnih naborih vhodnih podatkov.
- **Mogoče ga je opraviti.** V načelu ga je mogoče izpeljati "peš", s papirjem in svinčnikom.

(povz. po Kozak, 1986, str. 11)

Definicija 2. Algoritem je **kompoziten** oziroma **sestavljen**, če je sestavljen iz več drugih algoritmov.

1.1.2 Definicija urejanja

Naj bo dan končen vektor elementov $\vec{a}$ z elementi iz A, linearno urejena množica ključev $(K, \leq)$ in funkcija $f\colon A \to K, f\colon a \mapsto k$,[1] ki vsakemu $a \in A$ priredi njegov ključ $k \in K$. Tedaj lahko definiramo relacijo linearne urejenosti $\leq$ med elementi A.

$$a \leq a' \overset{\text{def}}{\iff} f(a) \leq f(a') \qquad \forall\, a, a' \in A$$

Urejanje pomeni permutiranje elementov vektorja $\vec{a}$ tako, da velja:

$$a_1 \leq a_2 \leq \cdots \leq a_n.$$

(povz. po Knuth, 1998, str. 4)

Za potrebe psevdokode definirajmo še tip *item*, ki naj ponazarja elemente A-ja. Naj bo k ključ elementa tipa *cmp_t*, na katerem je definirana relacija linearne urejenosti. Tip *cmp_t* predstavlja množico linearno urejenih ključev, v definiciji urejanja imenovano K.

```
struct item {
    cmp_t k;
    /* ostale komponente */
};
```

Ostale komponente predstavljajo s stališča urejanja nepomembne podatke o elementih v polju. S stališča sortirnih algoritmov je ključ torej *edini* pomemben podatek, zato ostalih komponent ni treba podrobno definirati. (povz. po Wirth, 1985, str. 107)

Definicija 3. Sortirni algoritem je algoritem, ki prejme neko podatkovno strukturo s podatki in neko relacijo linearne urejenosti med njimi ter jih uredi po vrstnem redu, ki ga določa ta relacija.

Definicija 4. Za sortirni algoritem pravimo, da je **stabilen**, če ostane po končanem urejanju relativni vrstni red elementov z istim ključem nespremenjen. Stabilnost urejanja je pogosto zaželena, če so elementi že urejeni po neki sekundarni $\leq$ relaciji, torej glede lastnosti, ki je (primarni) ključ ne izraža.

Definicija 5. O sortirni metodi pravimo, da se dogaja **na mestu**, če se med izvajanjem podatki, ki jih urejamo, ne kopirajo v dodaten pomnilnik.

Definicija 6. Sortirni algoritem je **primerjalen**, če ureja elemente tako, da jih primerja med seboj.

Vsi nadaljnji sortirni algoritmi bodo definirani kot procedure s tremi parametri, *a*, *prvi*, *zadnji*, ki predstavljajo:

- *a*: polje, ki ga urejamo. V splošnem zahtevamo, da je to katerakoli podatkovna struktura, ki podpira neko vrsto naključnega dostopa. Le-tega uporabljamo, kadar uporabljamo operator $[\cdot]$. Zaradi časovne analize algoritmov zahtevamo, da operator $[\cdot]$ deluje v konstantnem času. Indeksiranje elementov polja se začne z 0. V nadaljnjem opisovanju bomo uporabljali besedo **polje** za katerokoli podatkovno strukturo, ki ustreza zgornjim zahtevam.

[1] f je ponavadi trivialen, ker je $f(a)$ ponavadi kar del a-ja in nam ga ni treba računati, lahko pa to ni res.

- *prvi*: indeks prvega elementa polja, ki ga želimo urediti.
- *zadnji*: indeks elementa, ki je *za* zadnjim elementom polja, ki ga urejamo. Lahko ni veljaven indeks v a.

Vse procedure bodo spremenile polje a, nobena ne bo ničesar vrnila.

Pri definiciji časovne in prostorske zahtevnosti algoritma bo za prostorsko zahtevnost navedena zgolj dodatna poraba pomnilnika, saj vsi algoritmi porabijo $\mathcal{O}(n)$ pomnilnika za shranjevanje celotnega polja, kjer n predstavlja število elementov v polju ali razliko $zadnji - prvi$.

1.1.3 Rekurzija

Rekurzija je proces ponavljanja elementov na nek samopodoben način. V matematiki in računalništvu se nanaša na način definicije funkcij. Rekurzivna funkcija je tista, ki se v svoji definiciji sklicuje sama nase. Če želimo, da je taka definicija sploh smiselna, moramo podati vsaj en osnoven nerekurziven primer in še množico pravil, ki vse ostale primere prevedejo na osnovni primer.

Poglejmo si primer definicije fakultete.[2] Fakulteta naravnega števila n je v matematiki funkcija, ki določa zmnožek vseh celih števil, manjših ali enakih n. Označi se jo z $n!$.

$$n! = \begin{cases} 1\ ; & \text{če } n = 1 \\ n \cdot (n-1)!\ ; & \text{sicer} \end{cases}$$

Pri fakulteti imamo definiran en osnovni primer ($1! = 1$), vsi ostali primeri pa se prej ali slej pretvorijo v ta primer. Poglejmo si primer števila 4.

1. Ker 4 ni enako 1, gledamo drugo pravilo za izračun fakultete: $4!$ je torej $4 \cdot (4-1)!$
2. V izračunu imamo še vedno fakulteto, torej pogledamo kako se izračuna $3!$, da lahko izračunamo $4 \cdot (3!)$.
3. 3 ni enako 1, torej gledamo drugo pravilo, $3! = 3 \cdot (2!)$. Iz tega torej sledi $4! = 4 \cdot (3 \cdot (2!))$. V računu nastopa $2!$, izračunamo jo po drugem pravilu.
4. $2!$ izračunamo kot $2 \cdot 1!$, $4! = 4 \cdot (3 \cdot (2 \cdot (1!)))$. V računu še vedno nastopa fakulteta, torej jo izračunamo.
5. $1!$ je po prvem pravilu enaka 1. $4!$ smo prevedli v $4 \cdot (3 \cdot (2 \cdot (1)))$. Zdaj lahko brez problemov izračunamo $4!$.
6. Zmnožimo in dobimo rezultat $4! = 24$.

1.1.4 Psevdokoda

Psevdokoda je neformalen zapis nekega algoritma, ki uporablja strukture iz programskih jezikov in je pomensko pravilen, namenjen predvsem lažjemu razumevanju algoritma.

---

[2] Bolj običajno se fakulteto definira kot: $n! = \prod_{k=1}^{n} k = 1 \cdot 2 \cdot 3 \cdot \ \cdots \ \cdot n$.

Pri opisu algoritmov v psevdokodi bom uporabljal prireditve (označene s $\leftarrow$), *while* zanke, *for* zanke, vejitve (*if* stavke), klice funkcij, osnovne matematične $(+, -, \cdot, /)$ in logične operatorje $(=, \neq, \wedge, \vee, \neg, <, \leq, >, \geq)$, oklepaje (()) in simbol za zaokroževanje navzdol ($\lfloor\rfloor$). Za polja $a[i]$ pomeni element polja na indeksu i, polja se začnejo z indeksom 0. Predpostavil bom tudi, da že obstaja funkcija SWAP, ki zamenja vrednosti dveh spremenljivk. Uporabljal bom tudi Boolovi vrednosti (`true` in `false`).

1.1.5 Bisekcija

Bisekcija je postopek za iskanje ničel zveznih funkcij. Denimo, da poznamo tak interval $[a, b]$, da je zvezna funkcija $f\colon \mathbb{R} \to \mathbb{R}$ v krajiščih različno predznačena. Potem iz zveznosti sledi, da ima f na intervalu (a, b) vsaj eno ničlo. Če vzamemo sredinsko točko $s = \frac{a+b}{2}$, potem bo, razen, če je $f(s) = 0$, kar pomeni, da smo imeli srečo in zadeli ničlo, na enem izmed intervalov $[a, s]$ ali $[s, b]$ funkcija v krajiščih spet različno predznačena in to vzamemo za nov interval $[a, b]$. Postopek rekurzivno ponavljamo in v vsakem koraku nadaljujemo z razpolovljenim intervalom, ki zagotovo vsebuje vsaj eno ničlo. Ko je interval dovolj majhen (manjši od želene vrednosti ϵ), končamo in vrnemo točko s sredine intervala kot približek za ničlo funkcije f. Algoritem bisekcije je zapisan v psevdokodi kot algoritem 1. (povz. po Plestenjak, 2010, str. 36)

V našem primeru bomo iskali presečišče funkcij povprečnega časa izvajanja v odvisnosti od dolžine polja, $T_1(n)$ in $T_2(n)$ dveh sortirnih algoritmov. Iskali bomo torej tisti n, pri katerem moramo zamenjati sortirni algoritem. Tako za funkcijo f lahko vzamemo kar razliko v času urejanja med tema algoritmoma. Ko sta razliki nasprotno predznačeni, smo dobili želeni interval $[a, b]$. Naš ϵ bo enak 1, saj so dolžine polj lahko le cela števila in je ena najmanjša dolžina, pri kateri se predznaka še lahko razlikujeta. Možno je sicer, da imata algoritma na intervalu $[a, b]$ več kot eno ničlo. Vemo le, da je ničel liho mnogo, bisekcija pa lahko najde katerokoli izmed njih. To za našo želeno natančnost presečišča ne bo predstavljalo večjega problema, katerakoli ničla iz intervala bo dovolj dobra.

Algoritem 1 Bisekcija

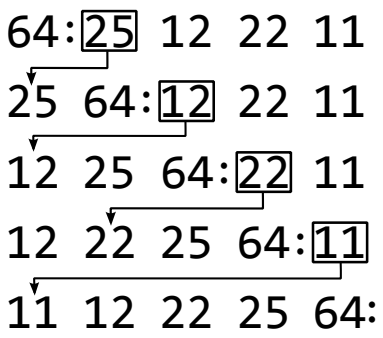
```
1: while |b − a| < ε do
2:     s ← (a+b)/2
3:     if PREDZNAK(f(a)) = PREDZNAK(f(b)) then
4:         a ← s
5:     else
6:         b ← s
7:     end if
8: end while
```

1.1.6 Urejanje z navadnim vstavljanjem

Urejanje z navadnim vstavljanjem (*angl.* insertion sort) je metoda, podobna tisti, ki jo na široko uporabljajo igralci kart. Je primerjalni sortirni algoritem.

Zaporedje $a_1, a_2, \ldots, a_n$ razdelimo na že urejeni del $a_1, a_2, \ldots, a_{i-1}$ in še neurejeni del $a_i, \ldots a_n$. Na vsakem koraku, začenši z $i = 2$ in prirastkom ena, vstavimo

i-ti element začetnega zaporedja na ustrezno mesto v končnem zaporedju. Ko je i enak 2, je prvi del trivialno urejen, saj vsebuje le en element. Postopek urejanja z vstavljanjem je prikazan na sliki 1. Algoritem navadnega vstavljanja je v psevdokodi napisan kot algoritem 2.



Slika 1: Grafična predstavitev urejanja z navadnim vstavljanjem.

Urejanje z vstavljanjem je prikazano na petih naključno razporejenih številih. Urejeni del je od neurejenega ločen z dvopičjem, puščice nakazujejo mesto, na katerega smo vstavili trenutni element.

Algoritem 2 Urejanje z vstavljanjem

```
1:  function INSERTIONSORT(a, prvi, zadnji)
2:      for i ← prvi + 1 to zadnji do
3:          x ← a[i]
4:          j ← i − 1
5:          while (x < a[j]) ∧ (j ≥ 0) do
6:              a[j + 1] ← a[j]
7:              j ← j − 1
8:          end while
9:          a[j + 1] ← x
10:     end for
11: end function
```

Med postopkom iskanja pravega mesta je dobro med izvajanji primerjanj sproti premikati elemente. Z drugimi besedami pustiti x, da se "pogrezne" tako, da x primerjamo z naslednjim elementom a_j in ga bodisi vstavimo, če je ključ a_j manjši ali enak x, bodisi a_j premaknemo na desno ter nadaljujemo proti levi, če še nismo pri levem robu polja. Postopek "pogrezanja" lahko ustavita ta dva ločena pogoja, kot je prikazano v *while* zanki v algoritmu 2 na vrstici 5. Ko vstavimo še zadnji element v že urejeno zaporedje, smo z urejanjem zaključili.

Časovna in prostorska zahtevnost
Časovna zahtevnost je v

- najboljšem primeru $\mathcal{O}(n)$
- povprečnem primeru $\mathcal{O}(n^2)$
- najslabšem primeru $\mathcal{O}(n^2)$

Prostorska zahtevnost je v vsakem primeru $\mathcal{O}(1)$, saj se vse premene dogajajo na mestu. (povz. po Wirth, 1985, str. 109)

Spodnja časovna meja ustreza primeru, ko so elementi na začetku že urejeni, zgornja pa primeru, ko so elementi na začetku nasprotno urejeni. Podani algoritem opisuje tudi stabilen postopek urejanja, kajti medsebojni vrstni red elementov z enakimi ključi ostane nespremenjen.

Urejanje z navadnim vstavljanjem je zelo učinkovito na majhnih poljih. Učinkovito je tudi na že skoraj urejenih poljih, saj se s tem približujemo obnašanju v najboljšem primeru. Zaradi kvadratne časovne zahtevnosti je urejanje z navadnim vstavljanjem zelo neučinkovito na dolgih poljih, če ta niso že skoraj urejena.

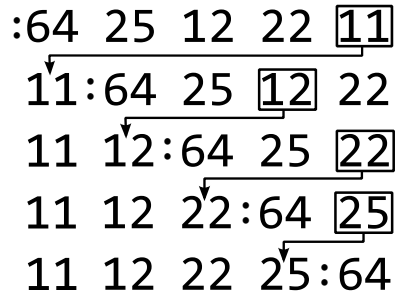
1.1.7 Urejanje z izbiranjem

Urejanje z izbiranjem (*angl.* selection sort) je eden najenostavnejših primerjalnih sortirnih algoritmov.

Algoritem je sledeč:

1. najdi element z najmanjšim ključem
2. zamenjaj ga s prvim elementom
3. ponovi zgornja koraka za ostanek polja.

Polje je med urejanjem razdeljeno na dva dela: podpolje že urejenih elementov, ki se nahaja na začetku in podpolje elementov, ki jih je še potrebno urejati in zasedajo preostali del polja. Postopek urejanja z izbiranjem je prikazan na sliki 2.



Slika 2: Grafična predstavitev urejanja z izbiranjem.

Najmanjši elementi v drugem delu polja so obkroženi. Puščica ponazarja, kako jih vstavimo na konec prvega dela. Dela sta ločena z dvopičjem.

V psevdokodi je urejanje z izbiranjem zapisano kot algoritem 3.

Časovna in prostorska zahtevnost

Časovna zahtevnost je v vsakem primeru enaka in sicer $\mathcal{O}(n^2)$.

Prostorska zahtevnost je v najboljšem, povprečnem in najslabšem primeru $\mathcal{O}(1)$, saj algoritem opravlja vse premene na mestu. (povz. po Wirth, 1985, str. 113)

Ker ima urejanje z izbiranjem v vsakem primeru kvadratno časovno zahtevnost, je neprimerno za daljša polja. Urejanje z izbiranjem tudi ni odvisno od prvotnega vrstnega reda, saj mora za iskanje najmanjšega elementa vedno pregledati celoten ostanek polja. Njegova prednost je v tem, da stori zelo malo zamenjav elementov, kar je še posebej primerno takrat, ko je premikanje elementov drago (imamo velike elemente).

Algoritem 3 Urejanje z izbiranjem

```
1: function SELECTIONSORT(a, prvi, zadnji)
2:     for i ← prvi to zadnji do
3:         x ← i
4:         for j ← i + 1 to zadnji do
5:             if a[j] < a[x] then
6:                 x ← j
7:             end if
8:         end for
9:         SWAP(a[x], a[i])
10:    end for
11: end function
```

1.1.8 Hitro urejanje

Hitro urejanje ali urejanje s porazdelitvami[3](*angl.* quicksort) je eden izmed najučinkovitejših sortirnih algoritmov, ki jih poznamo, razvil pa ga je C. A. R. Hoare. Je primerjalni sortirni algoritem, ki je zgrajen na principu deli in vladaj.

Sortiranje s porazdelitvami temelji na dejstvu, da moramo premene opravljati na večje razdalje, da bi bile učinkovitejše. Recimo, da imamo n nasprotno urejenih elementov. V tem primeru jih lahko urejamo z le $^n/_2$ premenami tako, da najprej premenjamo prvega in zadnjega in se postopoma pomikamo proti sredini. Seveda je to možno le, če vemo, da so elementi natanko nasprotno urejeni.

Prejšnji primer nas napelje na naslednji algoritem: izberemo poljuben element (recimo mu pivot in ga označimo z x), nato začnemo polje a pregledovati z leve, dokler ne najdemo elementa $a_i > x$ in nato z desne dokler ne najdemo elementa $a_i < x$. Elementa sedaj medsebojno zamenjamo in nadaljujemo s pregledovanjem in premenami, dokler se ne srečamo nekje na sredi polja. Polje je sedaj razdeljeno na levi del s ključi manjšimi od x in na desni del s ključi večjimi od x. Pivot nato umestimo med oba dela, kar je tudi njegovo končno mesto, in metodo ponovimo na delih levo in desno od pivota, dokler ne pridemo do že urejenih podpolj, torej tistih z dolžino manjšo ali enako 1.

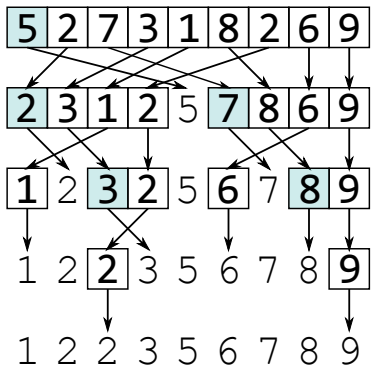
Iz prejšnjega algoritma ugotovimo, da je izbira pivota zelo pomembna. Pogosto je, da za pivot vzamemo kar zadnji element, vendar to sproži ravno najslabšo možnost izvajanja programa, če so elementi že urejeni. Zato sem v svoji implementaciji za pivot vzel mediano prvega, srednjega in zadnjega elementa oziroma t.i. mediano treh. Pogosta izbira je tudi naključni pivot. Postopek urejanja s porazdelitvami je prikazan na sliki 3. V psevdokodi je algoritem hitrega urejanja zapisan kot algoritem 4.

Časovna in prostorska zahtevnost

Časovna zahtevnost je v

- najboljšem primeru $\mathcal{O}(n \log_2 n)$
- povprečnem primeru $\mathcal{O}(n \log_2 n)$
- najslabšem primeru $\mathcal{O}(n^2)$

[3]Vir: IJS-jev slovar računalniških izrazov `http://www.ijs.si/cgi-bin/rac-slovar`.



Slika 3: Grafična predstavitev hitrega urejanja.

Hitro urejanje je prikazano na primeru devetih naključno razporejenih števil. Izbrani pivoti so obarvani. Vsaka vrstica predstavlja svojo globino rekurzije.

Algoritem 4 Hitro urejanje

```
1:  function QUICKSORT(a, prvi, zadnji)
2:      if zadnji − prvi < 2 then return
3:      end if
4:      s ← prvi
5:      e ← zadnji − 1
6:      m ← ⌊(s + e)/2⌋
7:      if (a[m] < a[s] < a[e]) ∨ (a[e] < a[s] < a[m]) then
8:          SWAP(a[s], a[e])                          ▷ a[s] je mediana treh
9:      else if (a[e] < a[m] < a[s]) ∨ (a[s] < a[m] < a[e]) then
10:         SWAP(a[m], a[e])                          ▷ a[m] je mediana treh
11:     end if
12:     pivot ← a[e]
13:     e ← e − 1
14:     while s ≤ e do
15:         while (s ≤ e) ∧ (a[s] < pivot) do   ▷ prvi element z leve, večji od pivota
16:             s ← s + 1
17:         end while
18:         while (s ≤ e) ∧ (a[e] ≥ pivot) do   ▷ prvi z desne, ki je manjši od pivota
19:             e ← e − 1
20:         end while
21:         if s < e then
22:             SWAP(a[s], a[e])
23:         end if
24:     end while
25:     a[zadnji − 1] ← a[s]
26:     a[s] ← pivot
27:     QUICKSORT(a, prvi, s)
28:     QUICKSORT(a, s + 1, zadnji)
29: end function
```

Prostorska zahtevnost je v najboljšem in povprečnem primeru $\mathcal{O}(\log_2 n)$, saj algoritem opravlja premene na mestu, vsak rekurziven klic pa zahteva $\mathcal{O}(1)$ prostora.

V najslabšem primeru je rekurzivnih klicev n, zato zahteva algoritem $\mathcal{O}(n)$ prostora. (povz. po Wirth, 1985, str. 128)

Hitro urejanje je eden najhitrejših sortirnih algoritmov, kar jih poznamo. Svojo hitrost dolguje arhitekturi današnjih procesorjev, ki imajo malo registrov in precej notranjega pomnilnika, saj lahko pivot ponavadi shranimo v register, kar prihrani veliko poizvedb do pomnilnika. Dober je na enakomerno porazdeljenih podatkih, a kljub pazljivemu izbiranju pivotov obstaja možnost neželenega najslabšega primera, čeprav je zelo neverjetna. Znana izboljšava hitrega urejanja je tudi, da krajših polj ne uredimo spet s hitrim urejanjem, temveč s kakšnim hitrejšim algoritmom za kratka polja, na primer z urejanjem z vstavljanjem.

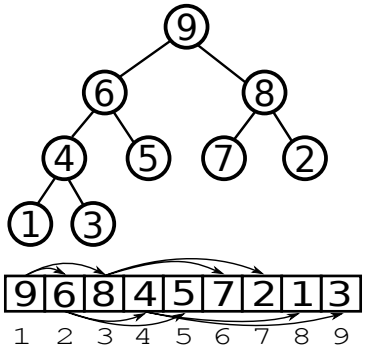
1.1.9 Urejanje s kopico

Urejanje s kopico (*angl.* heapsort) je sortirni algoritem, ki si pri urejanju pomaga s kopico.

Maksimalna kopica je dvojiško drevo, pri katerem velja, da je ključ očeta vedno večji od ključa otroka. Privzemimo, da je levo poravnano. Maksimalno kopico lahko predstavimo v polju, kjer je koren kopice na mestu 1, in velja, da sta otroka elementa na mestu i na mestih $2i$ in $2i + 1$. Maksimalno kopico, prikazano na sliki 4, tako predstavimo kot tako zaporedje elementov $a_i, a_{i+1}, \ldots, a_n$, da za njih velja:

$$a_i \geq a_{2i}$$
$$a_i \geq a_{2i+1}$$

pri vseh $i = 1 \ldots \lfloor {}^{n}/_{2} \rfloor$. (povz. po Kozak, 1986, str. 104)

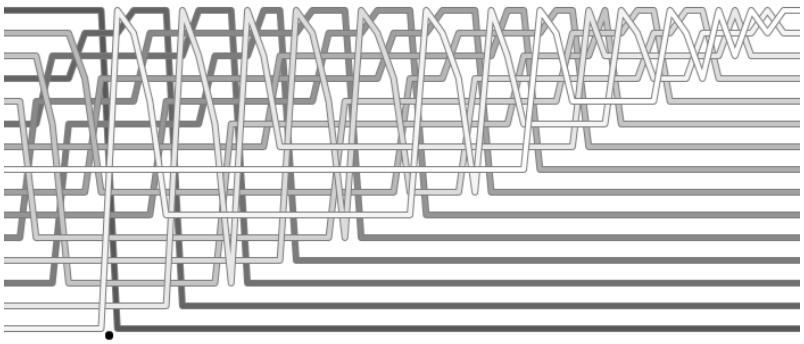


Slika 4: Kopica.

Predstavitev maksimalne kopice kot dvojiškega drevesa (zgoraj) in predstavitev maksimalne kopice v polju (spodaj). Vidimo, da je invarianta kopice vzpostavljena v obeh primerih. Pri predstavitvi kopice v polju so pod poljem prikazani tudi indeksi posameznih elementov.

Urejanje s kopico podatke najprej preuredi v maksimalno kopico. To naredimo tako, da za vse elemente, ki imajo otroke, ponovimo naslednji postopek: če je kateri izmed otrok večji od očeta, potem očeta zamenjamo z večjim izmed obeh otrok ter ponovimo postopek dokler ni zadoščeno invarianti kopice. Nato odstranimo koren kopice – največji element – in ga zamenjamo z zadnjim elementom kopice. Potem ponovno vzpostavimo invarianto kopice na preostanku elementov in zopet odstranimo korenski element ter ga zamenjamo s tistim na predzadnjem mestu. To

ponavljamo, dokler ni kopica dolga le en element. Urejanje s kopico je prikazano na sliki 5. V psevdokodi je algoritem urejanja s kopico prikazan kot algoritem 5.



Slika 5: Grafična predstavitev urejanja s kopico.

Vsaka črta predstavlja svoj element; temnejša kot je črta, večji je element. Do pike na dnu slike poteka urejanje podatkov v kopico, nato pa odstranjevanje največjega elementa in ponovna vzpostavitev kopice vse do konca. Slika je bila ustvarjena s programsko kodo, dostopno na spletnem naslovu: `http://corte.si/posts/code/visualisingsorting/index.html`.

Časovna in prostorska zahtevnost

Časovna zahtevnost je v najboljšem, povprečnem in najslabšem primeru $\mathcal{O}(n \log_2 n)$.

Prostorska zahtevnost je v vsakem primeru $\mathcal{O}(1)$, saj je kopica predstavljena kar v polju, ki ga urejamo, in ker se vse premene dogajajo na mestu. (povz. po Wirth, 1985, str. 123)

Čas izvajanja urejanja s kopico je neodvisen od morebitne urejenosti podatkov v polju, kot lahko razberemo iz časovne zahtevnosti. Zaradi dokaj zapletenega postopka urejanja na manjših poljih ni tako hiter, je pa čedalje hitrejši, ko se dolžina polja povečuje.

1.1.10 Urejanje z zlivanjem

Urejanje z zlivanjem (*angl.* merge sort) je primerjalni sortirni algoritem, zgrajen na principu deli in vladaj. Razvil ga je John von Neumann leta 1945. Urejanje z zlivanjem uporablja idejo, da je v urejeno polje hitreje združiti dve že urejeni polji, kot pa dve še neurejeni. Algoritem urejanja z zlivanjem ureja tako:

1. če je polje dolgo 0 ali 1, je že urejeno, če ne:
2. razdeli polje v dve približno enako dolgi podpolji
3. urejaj podpolji
4. združi podpolji nazaj v celotno urejeno polje.

Algoritem je grafično prikazan na sliki 6 na primeru sedmih naključno izbranih števil. V psevdokodi je urejanje z zlivanjem zapisano kot algoritem 6.

Algoritem 5 Urejanje s kopico

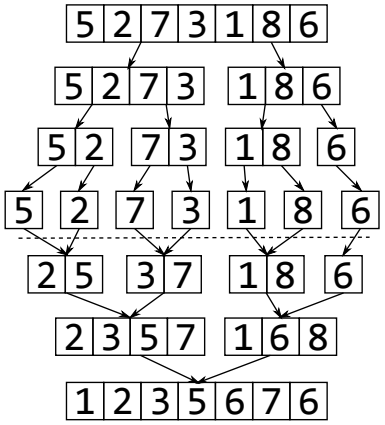
```
1:  function HEAPSORT(a, prvi, zadnji)
2:      NAKOPIČI(a, prvi, zadnji)
3:      while prvi < zadnji do
4:          SWAP(a[prvi], a[zadnji]);
5:          zadnji ← zadnji − 1
6:          POGREZNI(a, prvi, prvi, zadnji)
7:      end while
8:  end function
9:
10: function NAKOPIČI(a, prvi, zadnji)
11:     zacetek ← ⌊(prvi + zadnji)/2⌋ − 1
12:     while zacetek ≥ prvi do          ▷ pogreznemo vse elemente, ki imajo otroke
13:         odmik ← koren − prvi
14:         POGREZNI(a, prvi, zacetek, zadnji)
15:         zadnji ← zadnji − 1
16:     end while
17: end function
18:
19: function POGREZNI(a, prvi, koren, zadnji)
20:     odmik ← koren − prvi
21:     while koren + odmik < zadnji do      ▷ dokler ima koren vsaj enega otroka
22:         otrok ← koren + odmik + 1
23:         zamenjava ← koren          ▷ zapomnimo si, s čim naj zamenjamo koren
24:         if a[koren] < a[otrok] then                ▷ če je otrok večji od korena
25:             zamenjava = otrok          ▷ potem moramo zamenjati prvega otroka
26:         end if          ▷ če drugi otrok obstaja in je večji od trenutne zamenjave
27:         if (otrok + 1) ≤ (zadnji ∧ zamenjava < a[otrok + 1]) then
28:             zamenjava = otrok + 1                  ▷ potem zamenjamo njega
29:         end if
30:         if zamenjava ≠ koren then
31:             SWAP(a[zam], a[koren])
32:             koren ← zamenjava
33:             odmik ← koren − prvi
34:         else
35:             return
36:         end if
37:     end while
38: end function
```

Časovna in prostorska zahtevnost

Časovna zahtevnost je v najboljšem, povprečnem in najslabšem primeru $\mathcal{O}(n \log_2 n)$. Časovna zahtevnost urejanja z zlivanjem ni odvisna od morebitne že delne urejenosti ali neurejenosti podatkov v polju.

Prostorska zahtevnost je v vsakem primeru $\mathcal{O}(n)$.

(povz. po Knuth, 1998, str. 158)



Slika 6: Grafična predstavitev urejanja z zlivanjem.

Vsaka vrstica števil predstavlja svojo globino rekurzije, vse dokler niso polja dolga le po en element. Nato se začne izvajati četrta točka algoritma, ponazorjena z vrsticami nižje od črtkane črte, ki združi podpolja v urejeno polje.

V praksi je urejanje z zlivanjem najbolj učinkovito na sekvenčnih medijih, kot na primer tračnih enotah, ali na podatkovnih strukturah, ki nimajo hitrega neposrednega dostopa do nekega elementa z njegovim indeksom, kot na primer povezani seznami.

1.2 Opredelitev problema in cilji

Problem naloge je poiskati najučinkovitejši sortirni algoritem. Tak algoritem mora biti prilagodljiv, saj mora biti čim hitrejši na različnih računalnikih in različnih tipih podatkov. Prav to je problem sortirnih algoritmov, ki so trenutno dostopni v vgrajenih knjižnicah za urejanje, kot na primer *algorithm* v C++. Algoritmi, implementirani v knjižnicah, so zelo optimizirani za že vgrajene tipe in so napisani tako, da so hitri na večini računalnikov. To pomeni, da vgrajeni algoritem odlično ureja vgrajene tipe na povprečnem računalniku, ko pa se oddaljujemo od povprečja ali pa algoritmu damo urejati netipične podatke, se algoritem oddalji od optimalnosti. V tej nalogi želimo poiskati tak sortirni algoritem, ki se lahko prilagaja računalniku in podatkom ter eksperimentalno ugotovi, kaj je pri trenutnih pogojih optimalen sortirni algoritem. Zaželeno je tudi, da je enostaven za uporabo, vendar ima uporabnik kljub temu nad njim dovolj nadzora.

1.3 Hipoteze

1. Učinkovitost sortirnega algoritma je zelo odvisna od lastnosti elementov, ki jih urejamo.

2. Kompoziten sortirni algoritem se obnese boljše na različnih tipih podatkov kot vsak posamezen algoritem.

Algoritem 6 Urejanje z zlivanjem

```
1:  function MERGESORT(a, prvi, zadnji)
2:      if zadnji − prvi < 2 then return
3:      end if
4:      srednji ← ⌊(prvi + zadnji)/2⌋
5:      MERGESORT(a, prvi, srednji)
6:      MERGESORT(a, srednji + 1, zadnji)
7:      MERGE(a, prvi, srednji, zadnji)
8:  end function
9:
10: function MERGE(a, prvi, srednji, zadnji)
11:     i ← prvi
12:     j ← srednji
13:     t ← prvi
14:     začasen ←KOPIRAJ(a, prvi, zadnji)        ▷ kopiramo polje a v pomnilnik
15:     while i < srednji ∧ j < zadnji do
16:         if a[i] < a[j] then
17:             a[t] ← začasen[i]
18:             i ← i + 1
19:         else
20:             a[t] ← začasen[j]
21:             j ← j + 1
22:         end if
23:         t ← t + 1
24:     end while
25:     while i < srednji do
26:         a[t] ← začasen[i]
27:         t ← t + 1
28:     end while
29:     while j < srednji do
30:         a[t] ← začasen[j]
31:         t ← t + 1
32:     end while
33: end function
```

2 Metode dela

Za preverjanje hipotez sem si izbral eksperiment. Hipotezi je z njim najlažje potrditi ali ovreči, drugih virov na to temo pa ni v izobilju. Pri pripravi eksperimenta sem potreboval tudi nekaj sekundarnih virov. Za izvedbo eksperimenta sem implementiral vse v teoretičnem delu opisane sortirne algoritme. Za implementacijo sem si izbral programski jezik C++, predvsem zato ker se program prevede v strojno kodo, nad katero ima programer veliko nadzora. Zmožen je optimizacije na najnižji ravni, medtem ko je pisanje kode kljub temu nezahtevno opravilo kot pri drugih programskih jezikih tretje generacije.

2.1 Implementacija sortirnih algoritmov

Vsak izmed sortirnih algoritmov je implementiran kot funkcija z dvema parametroma. Vsaka funkcija sprejme dva iteratorja z naključnim dostopom (*Random Access Iterator* ali kar *RAI*), ki kažeta na prvi element dela polja, ki ga urejamo, in na prvi element za koncem tega dela. Slednji je lahko neveljaven v polju (kaže preko njegovega konca). Tako obliko funkcij sem si izbral predvsem zato, ker je v skladu z zapisom, ki ga uporablja C++-ova knjižnica *algorithm*. Drugi razlog je, da s tem zapisom dosežemo minimalno število parametrov pri želeni funkcionalnosti. Če bi želeli le en parameter, bi bilo to izvedljivo, vendar bi se morali odpovedati zmožnosti, da lahko sedaj uredimo le želeni del polja. V primeru zgolj enega parametra to ne bi bilo možno, vedno bi se urejalo celotno polje.

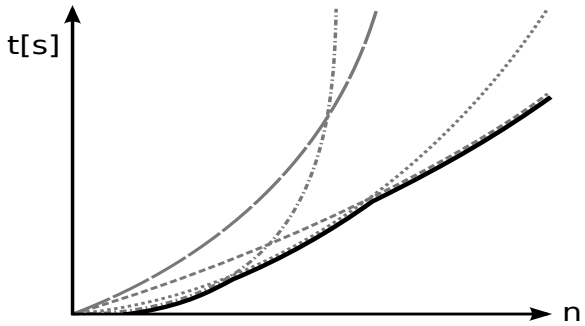
Vsak sortirni algoritem, ki sem ga implementiral, ureja zgolj z uporabo operatorja $<$ ("je manjše"), ki je usklajen z relacijo linearne urejenosti $\leq$ iz uvoda. Vsak algoritem je definiran kot funkcija tipa *void*, kar pomeni, da ne vrne ničesar, in torej spremeni vrstni red elementov polja, ki ji ga podamo.

2.2 Implementacija kompozitnega sortirnega algoritma

Ker so sortirni algoritmi vsak zase dobri na zgolj nekaterih dolžinah polj in se pri tej lastnosti razlikujejo, nas to privede do zamisli, da bi pri dani dolžini n uporabili tisti sortirni algoritem, ki je pri tem n najboljši. Ta zamisel je prikazana na sliki 7. Torej moramo vedeti, pri katerem n je potrebno uporabiti kateri sortirni algoritem, kar pa si bomo morali zapomniti v konfiguracijo.

Konfiguracija pove, kateri sortirni algoritem naj se uporablja pri katerem n. Lahko se jo predstavi kot niz znakov z zelo preprostimi pravili. To je zaželeno predvsem zato, ker je konfiguracija namenjena temu, da je trajna in berljiva ter da si jo uporabnik shrani. Ker je predstavljena kot niz znakov, jo lahko tudi zapiše v datoteko in jo po želji tudi spremeni.

Kompozitni sortirni algoritem torej vedno uporabi tisti sortirni algoritem, ki je za trenutni n določen v konfiguraciji. Tako spremenjen algoritem za urejanje bi v idealnem primeru torej vhodno polje vedno uredil s čimbolj optimalnim sortirnim algoritmom. Pomaga nam tudi dejstvo, da imajo nekateri algoritmi za urejanje zaradi principa deli in vladaj še to lepo lastnost, da polja delijo na manjša podpolja. Ta manjša polja lahko spet uredimo z optimalnih algoritmom za njihovo dolžino. To nam prinese možnost veliko hitrejšega urejanja polj, saj se lahko odločimo, kateri sortirni algoritem bomo uporabili, ne le za vhodno polje, temveč tudi za nekatera



Slika 7: Ideja kompozitnega sortirnega algoritma.

Sive krivulje prikazujejo algoritme, ki niso kompozitni, črna pa kompozitni algoritem. Krivulja kompozitnega algoritma ustreza spodnji meji ostalih krivulj, torej se vedno uporablja najhitrejši sortirni algoritem pri tistem n.

podpolja, ki jih ustvarijo sortirni algoritmi, kot na primer hitro urejanje ali urejanje z zlivanjem.

Kompozitni sortirni algoritem mora torej pri svoji implementaciji poleg dveh iteratorjev z naključnim dostopom sprejeti tudi konfiguracijo. Procedura *sort*, ki predstavlja kompozitni sortirni algoritem, deluje enako kot algoritem 7.

Algoritem 7 Kompozitni sortirni algoritem

```
1: function SORT(prvi, zadnji, conf)
2:     n ← zadnji − prvi
3:     algoritem ← NAJDI_PRAVI_ALGORITEM(n,conf)
4:     Sortiraj z algoritmom.
5: end function
```

Če v *algoritmu*, ki ga kličemo pri algoritmu 7 na vrstici 4, pride do rekurzivnega klica, naj kliče funkcijo *sort* namesto samega sebe.

V skladu s tem je bilo treba popraviti ostale sortirne algoritme, da namesto rekurzivnega klica samega sebe kličejo proceduro *sort* in tako pustijo, da se manjša polja morda uredijo z drugim sortirnim algoritmom ter tako prispevajo k bolj učinkovitemu urejanju celotnega polja. Da pa sortirni algoritmi sploh lahko kličejo glavno proceduro *sort*, morajo kot parameter prejeti tudi konfiguracijo, ki jo na koncu podajo nazaj proceduri *sort*.

Funkcija NAJDI_PRAVI_ALGORITEM, ki jo kliče procedura SORT, se mora ob vsakem klicu odločiti, kateri algoritem naj uporabi. Po celi konfiguraciji mora torej poiskati, kateri sortirni algoritem naj uporabi, kar običajno terja časovno zahtevnost $\mathcal{O}(n)$, kjer je n število vnosov v konfiguracijo. Ker pa mora biti podana konfiguracija smiselna, mora biti urejena naraščajoče. Torej lahko želeni dolžini priredimo sortirni algoritem v logaritemskem času, in ne v linearnem, z dvojiškim iskanjem po urejenem polju, kot opisano v (Knuth, 1998, str. 409.). To je velika izboljšava še posebej za dolge konfiguracije, čeprav je lahko pri krajših dvojiško iskanje celó malo počasnejše od linearnega.

Da je konfiguracija smiselna, mora biti sestavljena iz zaporedij oblike

$$: sort - - > število; \quad .$$

sort pove, kateri sortirni algoritem naj se uporablja na katerih poljih. Črka, ki predstavlja posamezen algoritem, je prva črka njegovega angleškega imena. ($i-$insertion, $s-$selection, $q-$quick, $h-$heap, $m-$merge)

število predstavlja katerokoli možno dolžino polja, ki ga urejamo. Torej je lahko katerokoli nenegativno število. Dovoljeno je tudi število -1, ki predstavlja neskončno (∞).

Zadostiti mora tudi naslednjima pogojema:

1. števila morajo biti urejena naraščajoče in
2. zadnje in samo zadnje število mora biti -1.

S tema dvema pogojema dosežemo, da se za vse možne dolžine polj enolično določi, kateri "sort" naj se uporablja, kar je tudi namen konfiguracije.

V nadaljevanju naloge bom pri vseh primerih konfiguracij zaradi večje preglednosti namesto "$- - >$" pisal "$\longrightarrow$".

Primer smiselne konfiguracije:

$$: i \longrightarrow 20; : h \longrightarrow 45; : i \longrightarrow 123; : q \longrightarrow 7925; : m \longrightarrow -1;$$

Konfiguracija upošteva vsa slovnična pravila. Števila so urejena naraščajoče in samo zadnje število je -1. Ta konfiguracija je torej pravilna in smiselna. Konfiguracija pove, da naj se polja, katerih dolžine so iz intervala $[0, 20]$ ureja z vstavljanjem, polja z dolžinami iz intervala $(20, 45]$ naj se ureja s kopico, polja, ki imajo dolžine na intervalu $(45, 123]$ zopet z vstavljanjem, polja z dolžinami na intervalu $(123, 7925]$ s porazdelitvami, polja, katerih dolžine pa so iz intervala $(7925, \infty)$ oziroma tista, katerih dolžine so večje od 7925, pa naj se ureja z zlivanjem. Tako je za prav vse možne dolžine enolično določeno, kateri sortirni algoritem naj bo uporabljen.

Oglejmo si še nekaj realnih primerov, dobljenih za različne tipe podatkov.

$$: i \longrightarrow 142; : q \longrightarrow -1;$$

$$: m \longrightarrow -1;$$

Tudi ta dva primera sta smiselna. V prvem vidimo, da naj se uporablja urejanje z vstavljanjem, če je dolžina polja manjša ali enaka 142, sicer naj se uporablja hitro urejanje. Pri drugem primeru, ki je tudi eden izmed najkrajših možnih primerov konfiguracije, se uporablja samo urejanje z zlivanjem.

Definirajmo še množico **osnovnih konfiguracij**. To je množica konfiguracij, pri katerih sortirni algoritem ni kompoziten, torej

$$\{" : i \longrightarrow -1;", " : s \longrightarrow -1;", " : q \longrightarrow -1;", " : h \longrightarrow -1", " : m \longrightarrow -1;"\}.$$

Ker se procedura *sort* uporablja kot knjižnica, potebujemo tudi pretvornik konfiguracije v niz znakov in obratno, zaradi česar je potrebno formalno definirati slovnico konfiguracije, ki je (v notaciji BNF)[4] sledeča:

$$
\begin{aligned}
\text{<sort>} &::= \text{“}i\text{”} \mid \text{“}s\text{”} \mid \text{“}q\text{”} \mid \text{“}h\text{”} \mid \text{“}m\text{”} \\
\text{<število>} &::= \in \mathbb{N} \mid \text{“} -1\text{”} \\
\text{<vnos>} &::= \text{“ :”} \ \text{<sort>} \ \text{“}- - >\text{”} \ \text{<število>} \ \text{“;”} \\
\text{<konfiguracija>} &::= \ \text{<vnos>} \mid \text{<konfiguracija><vnos>}
\end{aligned}
$$

2.3 Iskanje optimalne konfiguracije

Za hitrost procedure *sort* je bistvena konfiguracija, zato je iskanje optimalnega kompozitnega sortirnega algoritma pravzaprav iskanje optimalne konfiguracije zanj. Ta razdelek je namenjen prav temu.

Glavna ideja je, da dva kompozitna sortirna algoritma lahko primerjamo po času izvajanja. Tisti, ki porabi manj časa, je boljši pri dolžini polja, ki sta ga urejala. S tem smo za algoritma a in b definirali relacijo a je n-boljši od b:

$$a \leq_n b \overset{\text{def}}{\iff} a \text{ je v povprečju na poljih dolžine } n \text{ boljši od } b.$$

Tako lahko problem rešimo kar s požrešno metodo: pri posamezni dolžini ugotovimo, kateri algoritem je najboljši, in to vnesemo v konfiguracijo. Glavni podatek, ki nam ga poda uporabnik, je meja učenja M. To je največja dolžina, za katero bo algoritem še iskal optimalno konfiguracijo. Od te dolžine naprej konfiguracija ni več nujno optimalna. Seveda v konfiguracijo pišemo le, če pride do spremembe na vodilnem mestu. Algoritem za iskanje najboljše konfiguracije je zapisan v psevdokodi kot algoritem 8.

Optimalno konfiguracijo bomo iskali takole:
za vsak n od začetnega do M:

1.) ustvarimo kandidate

2.) primerjamo kandidate

3.) najboljšega zapišemo v konfiguracijo

4.) nadaljujemo z naslednjim n.

Zgornji štirje koraki so v algoritmu 8 zapisani kot štiri zaporedne vrstice v *while* zanki na vrsticah od 3 do 8.

Ustvarjanje kandidatov poteka po sledečem algoritmu: za vsak sortirni algoritem s izvedemo sledeči korak: kandidat je trenutni najboljši, ki pa namesto do ∞ z zadnjim sortirnim algoritmom ureja le do prejšnjega n, potem pa naprej s s. Tako ustvarimo množico vseh možnih kandidatov za trenutno najboljši sortirni algoritem. To množico lahko opišemo tudi kot množico konfiguracij, ki nastane tako, da trenutni

[4] Backus–Naur Form, standardna notacija za kontekstno-neodvisne slovnice.

najboljši konfiguraciji dodamo vsako izmed osnovnih konfiguracij. V algoritmu 8 se zgornji postopek izvede ob klicu funkcije USTVARI_KANDIDATE.

Primer: recimo, da je trenutni najboljši algoritem

$$\text{“}: s \longrightarrow 10; : i \longrightarrow 47; : q \longrightarrow -1;\text{”}$$

in trenutni n 132. Vsi kandidati so pri danem najboljšem algoritmu torej:

$$\begin{aligned}
&\text{“}: s \longrightarrow 10; : i \longrightarrow 47; : q \longrightarrow 131; : i \longrightarrow -1;\text{”},\\
&\text{“}: s \longrightarrow 10; : i \longrightarrow 47; : q \longrightarrow 131; : s \longrightarrow -1;\text{”},\\
&\text{“}: s \longrightarrow 10; : i \longrightarrow 47; : q \longrightarrow 131; : q \longrightarrow -1;\text{”},\\
&\text{“}: s \longrightarrow 10; : i \longrightarrow 47; : q \longrightarrow 131; : h \longrightarrow -1;\text{”},\\
&\text{“}: s \longrightarrow 10; : i \longrightarrow 47; : q \longrightarrow 131; : m \longrightarrow -1;\text{”}.
\end{aligned}$$

Algoritem 8 Iskanje optimalne konfiguracije

```
1: naj ←""
2: n ← izbrana začetna vrednost
3: while n < M do
4:     kandidati ←USTVARI_KANDIDATE(naj)
5:     nova ←IZBERI_NAJBOLJŠO_PRI_N(n, kandidati, cmp)
6:     naj ← naj+": nova.ZADNJI_ALGO ⟶ n;"
7:     n ← n + 1
8: end while
9: NASTAVI_ZADNJO_MEJO(naj, −1)
10: return naj
```

Ker je prejšnji postopek za iskanje optimalne konfiguracije precej dolgotrajen, še posebej pri velikih n, ga lahko pospešimo. Določene n izpustimo in to tako, da jih pri krajših poljih izpustimo manj, pri daljših, ki se urejajo dalj časa, pa več. Tako moramo uvesti še dve spremenljivki: *razmik* in *pospešek*. *Razmik* pove trenutno število n-jev, ki jih bomo preskočili, *pospešek* pa pove, za koliko naj se *razmik* poveča pri vsaki iteraciji. Primer: če začnemo z $n = 4$ in *razmikom* 1 ter *pospeškom* 2, potem bomo preverjali najboljši algoritem pri vrednostih $n = 4, 5, 8, 13, 20, 29, 40, 53 \ldots$

S preskakovanjem nekaterih n-jev pridobimo na hitrosti algoritma, vendar izgubimo na natančnosti. Zdaj ob spremembi najboljšega algoritma ne vemo več, pri točno katerem n je potrebno zamenjati na nov algoritem. Ob takem primeru pa lahko natančneje poiščemo točko presečišča z bisekcijo. Seveda moramo pri tem postopku predpostaviti, da imata kompozitna sortirna algoritma na intervalu med prejšnjim in trenutnim n zgolj eno presečišče. Algoritem za iskanje presečišča med dvema kompozitnima sortirnima algoritmoma je zapisan v psevdokodi kot algoritem 9 in opisan v sledečih korakih:

1. **Najdemo začetno stanje.** Na vrsticah 2 in 3 algoritma 9 ugotovimo, kateri sortirni algoritem je boljši pri m in M, torej na mejah območja, na katerem se nahaja presečišče. Rezultat primerjave pri m in M je logična vrednost, ki jo shranimo v spremenljivki *spodaj* za m in *zgoraj* za M.

2. **Preverimo obstoj presečišča.** Če sta spodaj in zgoraj enaka, presečišča ni na intervalu $[m, M]$ in zato vrnemo ali zgornjo ali spodnjo mejo, tisto, ki je bližje pravemu presečišču. To je naloga *if* stavka na vrsticah 4 do 6.

3. **Preverimo, če smo že našli presečišče.** Če je razlika med m in M manjša od 1, potem je m enak M in je tam tudi presečišče. Našli smo rešitev in vrnemo vrednost m. Prejšnji pogoj je pravzaprav pogoj *while* zanke na vrstici 7

4. **Najdemo podinterval s presečiščem.** Preverimo, kateri algoritem je boljši pri srednji vrednosti $\frac{m+M}{2}$ (vrstica 8). Če je enak kot pri spodnji, je presečišče na intervalu $\left[\frac{m+M}{2}, M\right]$. Če pa je enak kot pri zgornji, pa je presečišče na intervalu $\left[m, \frac{m+M}{2}\right]$. Zgornjo ali spodnjo mejo nato spremenimo tako, da ustreza želenemu intervalu. To nalogo opravlja *if* stavek na vrsticah od 9 do 13.

5. **Ponovimo na želeni polovici.** Ponovimo korake od 3 naprej. To ponavljanje je predstavljeno z *while* zanko na vrsticah 7–14.

Algoritem 9 Presečišče dveh kompozitnih sortirnih algoritmov

```
 1: function PRESEČIŠČE(m, M, alfa, beta, cmp)
 2:     spodaj ← alfa ≤_m beta
 3:     zgoraj ← alfa ≤_M beta
 4:     if spodaj = zgoraj = false then return m          ▷ presečišče ni na
 5:     else if spodaj = zgoraj = true then return M      ▷ intervalu [m, M]
 6:     end if
 7:     while M − m > 0 do
 8:         sredina ← alfa ≤_⌊(m+M)/2⌋ beta
 9:         if sredina = spodaj then
10:             m ← ⌊(m+M)/2⌋
11:         else
12:             M ← ⌊(m+M)/2⌋
13:         end if
14:     end while
15:     return m
16: end function
```

2.4 Implementacija algoritma za iskanje optimalne konfiguracije

Ta razdelek je namenjen predvsem predstavitvi implementacije algoritma opisanega v razdelku 2.3. Poglavje je namenjeno tudi opisu uporabe knjižnice, ki implementira algoritem za iskanje optimalne konfiguracije in ga lahko bralec, ki ga implementacija ne zanima, brez večje škode za nadaljnje razumevanje preskoči.

Funkcija, ki implementira algoritem za iskanje optimalne konfiguracije, se imenuje *learn* in je definirana tako:

```
conf_t learn(const size_t M,
             const Compare<container_t>& cmp);
```

- Parameter M predstavlja mejo učenja. Parameter je konstanten, funkcija ga ne sme spreminjati. M ima tip *size_t*, kar je najmanj 32 bitno nenegativno celo število, torej predstavlja ravno tista števila, ki so veljavne dolžine polj.

- Parameter *cmp* predstavlja objekt tipa *Compare*, ki pomaga pri primerjanju različnih sortirnih metod. Parameter je konstanten, funkcija ga ne spreminja. Parameter je podan kot referenca (označeno z &), kar pomeni, da se ob klicu funkcije ne bo kopirala njegova vrednost, ampak se bo podal točno ta objekt. To je precej pomembno, saj je ta parameter lahko precej velik in je zaradi tega kopiranje neprimerno.

Objekt *Compare* je definiran tako:

```
Compare(const size_t iterations,
        const size_t limit,
        const double acceleration,
        const container_t& data);
```

- Parameter *iterations* pomeni, koliko iteracij posamezne sortirne metode bo naredil program, preden zabeleži njen rezultat. Večje kot je to število, bolj statistično zanesljive so meritve.

- Parameter *limit* pomeni število elementov, ki jih lahko hkrati hranimo v pomnilniku. Večje kot je to število, hitrejši bo program in bolj natančne bodo meritve. Biti mora najmanj M. Program bo v pomnilniku hranil največ $iterations \cdot M$ elementov, tudi če je dovoljen limit večji od tega.

- Parameter *acceleration* pomeni pospešek povečevanja dolžine polj, ki jih bo program urejal. Razmik med dolžinami se torej vsako iteracijo poveča za *acceleration*. Nujno mora biti pozitiven ali enak 0. Če je enak nič, potem se bo algoritem obnašal kot osnovni algoritem za iskanje optimalne konfiguracije (algoritem 8). Začetna vrednost razmika med elementi je 1, prva dolžina, pri kateri pa bo algoritem iskal najboljšo konfiguracijo pa bo 4. Ti dve vrednosti sem izbral, ker sta ob testiranjih dajali najboljše rezultate.

- Parameter *data* predstavlja podatke, iz katerih bo program jemal naključne elemente za polja, ki jih bo urejal. Polja bodo enakega tipa kot *data*, saj ima lahko tip podatkovne strukture velik vpliv na hitrost algoritmov. Podatki naj predstavljajo čim bolj tipične podatke, ki bodo kasneje urejeni, saj bodo tako rezultati boljši. Ta parameter je podan kot referenca, s čimer preprečimo kopiranje tega običajno velikega parametra v pomnilniku.

Vsi parametri so podani kot konstantni, saj se znotraj funkcije ne smejo spreminjati.

2.5 Merjenje časa izvajanja posameznih algoritmov

Merjenje časa izvajanja posameznih sortirnih algoritmov je pravzaprav edini način za primerjanje njihove učinkovitosti, zato se mu velja podrobneje posvetiti.

Parametri, ki nam jih poda uporabnik, so definirani v razdelku 2.3. Za merjenje časa sta pomembna le dva, to sta *iterations* in *limit*. Parameter, ki ga še potrebujemo, pa je dolžina polja, ki ga bo sortirni algoritem urejal, imenujmo ga n.

Naiven algoritem za merjenje je naslednji: Naredimo polje dolžine n in si zapomnimo trenuten čas. Premešamo polje in ga uredimo z želenim algoritmom. Ponovimo zadnja dva koraka tolikokrat, da je število ponovitev enako številu, ki ga je podal uporabnik. Nato si spet zapomnimo čas. Kvocient med razliko in številom iteracij je naš povprečen čas.

Glavna napaka zgornjega algoritma je v tem, da v celoten čas šteje tudi tisti čas, ko mešamo polje, ne le tistega, ko se polje ureja. To bi lahko odpravili tako, da bi pred vsakim urejanjem in po njem zabeležili čas, čase nato sešteli in jih delili z številom iteracij, vendar to ne bi bilo dovolj natančno, saj merjenje tako kratkih časovnih intervalov, še posebej ko se urejajo kratka polja, enostavno ni dovolj natančno, da bi bilo statistično zanesljivo. Zaželeno je torej, da se meri zgolj čas, ki ga algoritem porabi za urejanje polja, in večja natančnost meritve, kar narekuje večkratno urejanje. To storimo tako, da uredimo več polj zaporedoma in izmerimo skupni čas urejanja, iz katerega nato izračunamo povprečno vrednost. Ker želimo, da merimo zgolj čas, ko algoritem ureja, moramo pred tem premešana polja shraniti v pomnilnik. Tu nastopi parameter *limit*, ki pove, koliko elementov smemo imeti naenkrat v pomnilniku. Večji kot je limit, več meritev se bo izvedlo naenkrat in rezultat bo bolj natančen.

Čas izvajanja sem meril z funkcijo *clock_gettime*, ki ima od vseh znanih funkcij največjo natančnost (meri do nanosekunde natančno), kar je dovolj za naše potrebe.

Algoritem, ki meri čas posameznega algoritma, je zapisan je v psevdokodi kot algoritem 10 in opisan v sledečih korakih:

1. **Inicializacija.** Definirajmo spremenljivko *čas* in *napredek* ter ju nastavimo na 0 (vrstica 1, algoritem 10).

2. **Ugotovimo, koliko polj si lahko pripravimo.** Ker lahko polja, ki si jih bomo pripravili za testiranje, zavzemajo precej veliko količino pomnilnika, je njihovo število odvisno od limita, ki nam ga je podal uporabnik. Pripravimo si največ $k = \lfloor limit/n \rfloor$ polj, kjer je n dolžina polj. Če pripravimo samo k polj, zagotovo ne bomo prebili limita uporabnika. Če pa je razlika med napredkom in želenim številom iteracij še manjša od k, potem si moramo pripraviti le toliko polj, kot je še želenih iteracij. To naredi vrstica 3, kjer MIN predstavlja funkcijo, ki vrne manjšega od dveh parametrov.

3. **Pripravimo si želeno število polj.** To naredi na vrstici 4 klicana funkcija NAREDI_NAKLJUČNA_POLJA. Njeno obnašanje je sledeče: naredi vektor polj, ki so enakega tipa kot tisto, ki nam ga je podal uporabnik kot parameter *data*. Nato polja napolni z naključnimi elementi iz podanega polja s podatki. Z izbiro naključnih elementov in ne naključnih rezin preprečimo, da bi morda dobili drugačne nezaželene rezultate. To bi se lahko zgodilo, če bi uporabnik na primer pozabil premešati polje, preden ga poda, in bi se, če bi jemali zgolj rezine, vedno urejala že urejena polja, kar bi dalo precej drugačne rezultate kot sicer.

4. **Izmerimo čas urejanja.** Zapomnimo si trenuten čas (vrstica 5), nato uredimo vsa predpripravljena polja z želenim sortirnim algoritmom (vrstice 6–8) in si nato spet zapomnimo čas (vrstica 9).

5. **Povečamo skupni čas urejanja in napredek.** Razliko v času prištejemo spremenljivki *čas* in povečamo napredek za k.

6. **Ponavljamo do želenega števila iteracij.** Če je napredek večji ali enak želenemu številu ponovitev, končajmo. Drugače ponovimo vse korake od 2 naprej. To ponavljanje je prikazano v *repeat* zanki med vrsticama 2 in 12.

7. **Izračunamo povprečje.** Delimo *čas* s številom iteracij, ki so bile izvedene (tako dobimo čas izvajanja za urejanje enega samega polja) in to je iskana vrednost, ki jo vrnemo na vrstici 13.

Algoritem 10 Merjenje časa

```
1:  čas ← napredek ← 0
2:  repeat
3:      koliko ←MIN(⌊limit/n⌋, iteracije − napredek)
4:      polja ←NAREDI_NALKJUČNA_POLJA(koliko, data)
5:      start ←DOBI_ČAS()
6:      for i ← 0 to koliko do
7:          uredimo polje polja[i] z želenih algoritmom
8:      end for
9:      stop ←DOBI_ČAS()
10:     čas ← čas + stop − start
11:     napredek ← napredek + koliko
12: until napredek ≥ iteracije
13: return čas/iteracije
```

Tako merimo zgolj čas, ko algoritmi urejajo, hkrati pa se držimo limita v pomnilniku, ki nam ga je določil uporabnik. Časa ustvarjanja polj tako ne upoštevamo k rezultatu, kot je tudi prav.

3 Rezultati

Naš kompozitni sortirni algoritem smo primerjali z vsakim posameznim algoritmom izmed tistih, ki so bili opisani v teoretičnem uvodu v razdelku 1.1. C++ algoritem je algoritem, ki je že vgrajen v C++, v knjižnici *algorithm*. Ta algoritem nam je predstavljal oceno učinkovitosti kompozitnega algoritma. Imena sortirnih algoritmov so taka kot v teoretičnem uvodu, kompozitni sortirni algoritem se v tabelah zaradi krajšega zapisa imenuje kar urejanje s konfiguracijo. Vse algoritme sem testiral na štirih različnih tipih podatkov: *int*, *string*, *huge* in *slow*, ki so opisani v nadaljevanju. Podatkovna struktura, ki sem jo urejal, je bila vektor,[5] torej podatkovna struktrura, ki ustreza vsem zahtevanim kriterijem.

Pred testiranjem sem poiskal optimalno konfiguracijo za vsakega od tipov. Zgornja meja učenja M je bila 10.000, za vsako testirano dolžino je program naredil 10 iteracij, v pomnilniku pa je bilo dovoljenih največ 100.000 elementov. Razmik med elementi se je pri vsaki iteraciji povečal za 20. Limit je bil vedno dovolj velik, da je bilo možno v pomnilniku ustvariti vsa polja za testiranje. Za vsak tip sem konfiguracijo poiskal večkrat, vsakič so si bile med seboj zelo podobne, le mejne vrednosti posameznih algoritmov so se rahlo spreminjale. Izbral sem tisto, ki je bila najbolj povprečna in je izgledala najbolj smiselno.

Testiranje je bilo izvedeno na računalniku s procesorjem Intel® Core™ 2 Duo CPU T5870 @ 2.00GHz, 2 GB pomnilnika, na sistemu Kubuntu 10.10 z verzijo Linuxovega jedra 2.6.35-25-generic. Program je bil preveden s prevajalnikom `gcc` verzije 4.4.5 parametri `-O3`, `-Wall`, `-Weffc++`, `-pedantic` in `-std=gnu++0x`. Parameter `-O3` pomeni najvišjo stopnjo avtomatske optimizacije. Parameter `-std=gnu++0x` pa pove, kateri dialekt jezika je uporabljen, v našem primeru je to osnutek novega standarda za C++. Ostali parametri niso tako pomembni, njihovi pomeni so pojasnjeni v strani z navodili za uporabo za program `gcc`.[6]

Rezultati bodo vedno predstavljeni s tabelo, ki bo prikazovala čase urejanja posameznih algoritmov pri določenem številu elementov. Ponavadi je to število 10.000, saj je bil pri tem številu elementov razpored sortirnih algoritmov po učinkovitosti že močno določen in se tudi pri večanju števila ne bi spreminjal. Drugi razlog je tudi, da urejanje polj večih dolžin traja tudi dalj časa, s tem pa bi bilo pridobivanje drugih rezultatov za dalj časa prekinjeno, še posebej dolgo časa traja izdelovanje grafov. Učinkovitost sortirnih algoritmov bo predstavljena tudi z grafi, na katerih bodo krivulje vseh sortirnih algoritmov. Najpomembnejše pa bodo vrednosti v tabeli, ki bo prikazovala vsoto časov sortirnih algoritmov za vse dolžine za določen tip. To je pravzaprav približek ploščine med krivuljo sortirnega algoritma in abscisno osjo na določenem intervalu. Tako lahko vsak sortirni algoritem ocenimo, kako dober je na nekem tipu — manjša kot je ploščina, boljši je. Ker je to najbolj splošna ocena vsakega algoritma, ji velja dati posebno težo.

[5] Del STL (*angl.* standard template library) v C++, implementirana v knjižnici *vector*.

[6] *Angl.* man page. Stran z navodili je dostopna preko ukaza `man gcc`.

3.1 Tip *int*

Tip *int* predstavlja 32 bitno predznačeno celo število. Zavzame malo prostora v pomnilniku. Primerjanje dveh objektov tega tipa je hitro v primerjavi z drugimi tipi.

Najdena optimalna konfiguracija za kompozitni sortirni algoritem za tip *int* je sledeča:

$$: s \longrightarrow 5; : i \longrightarrow 67; : q \longrightarrow -1; \quad .$$

Tabela 1: Rezultati za tip *int*.

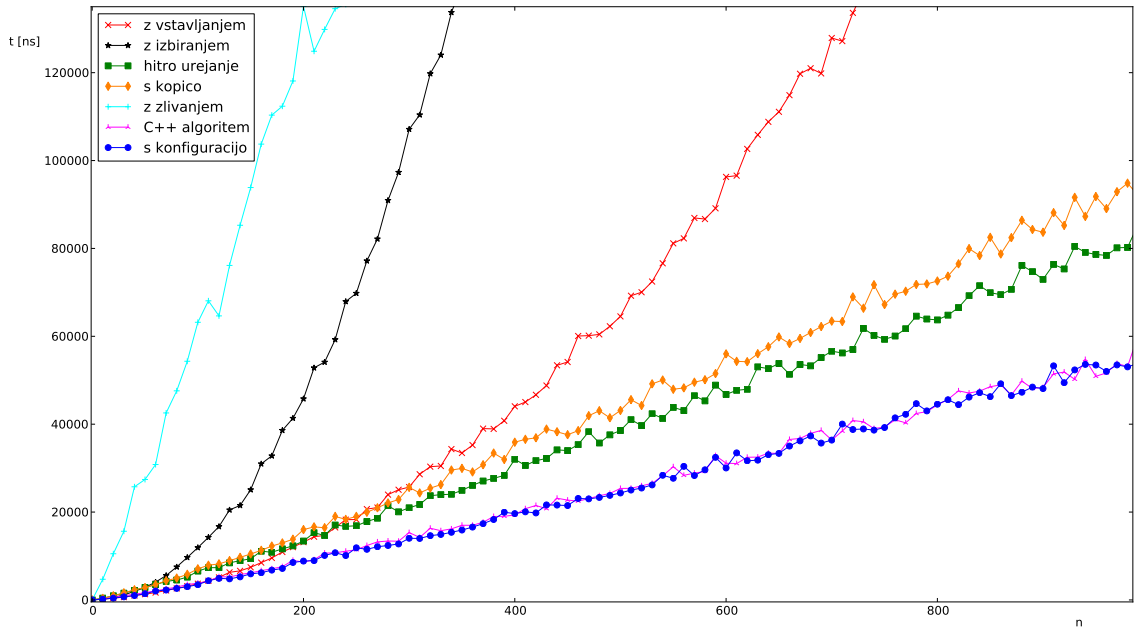
Bralec naj bo pozoren na časovne enote v stolpcih, saj se spreminjajo zaradi krajšega zapisa in natančnosti.

št. elementov **št. iteracij**	**10 el.** **100.000 it.**	**100 el.** **10.000 it.**	**1.000 el.** **1.000 it.**	**10.000 el.** **100 it.**	**100.000 el.** **10 it.**
urejanje	**čas [ns]**	**čas [μs]**	**čas [μs]**	**čas[ms]**	**čas[ms]**
z vstavljanjem	153,71	3,87536	259,74	23.651,79	2.413,15
z izbiranjem	162,25	11,52539	1.173,38	120.852,69	12.686,12
hitro urejanje	418,77	6,06326	80,81	1.053,24	12,31
s kopico	356,11	6,63458	93,08	1.341,48	16,66
z zlivanjem	5.142,34	59,73560	582,05	7.167,58	75,58
C++ algoritem	185,49	4,24863	59,83	698,21	8,82
s konfiguracijo	169,79	3,61435	58,69	749,76	9,35

Tabela 2: Skupen čas urejanja za tip *int*.

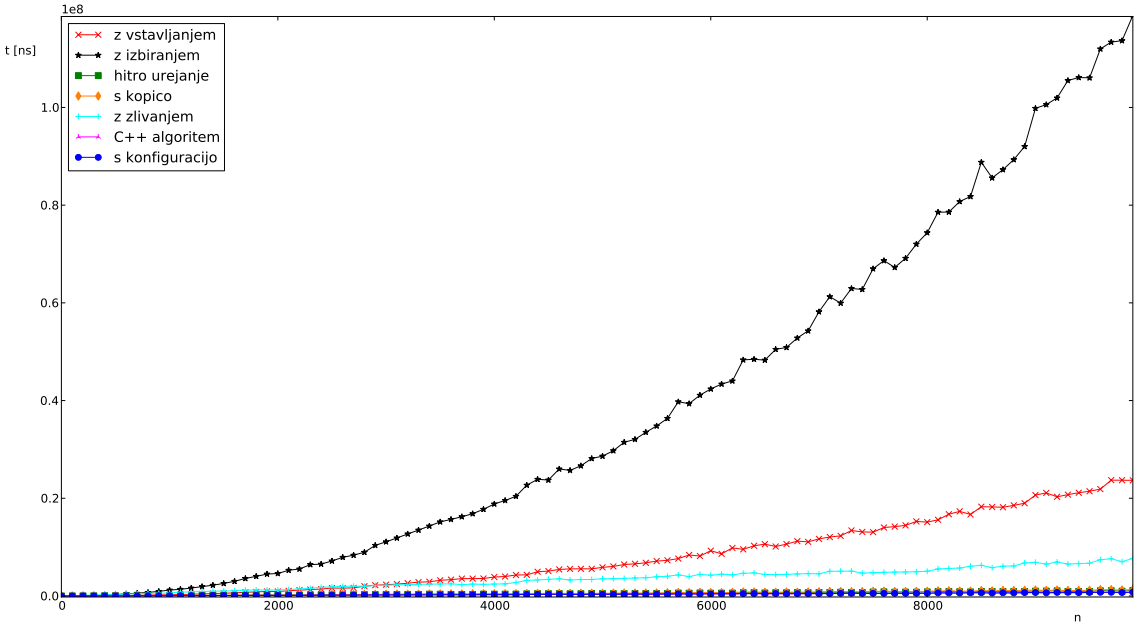
Skupne čase urejanja smo dobili tako, da za posamezen algoritem seštejemo ordinate vseh točk na sliki 9, ki prikazujejo čase tega sortirnega algoritma. Vrednosti so zaradi preglednosti urejene naraščajoče.

sortirni algoritem	**čas[ns], $1 \leq n \leq 10.000$**
vgrajeni algoritem v C++	33.938.913
urejanje s konfiguracijo	35.476.846
hitro urejanje	50.638.687
urejanje s kopico	59.057.953
urejanje z zlivanjem	345.880.886
urejanje z vstavljanjem	797.124.676
urejanje z izbiranjem	3.874.923.520



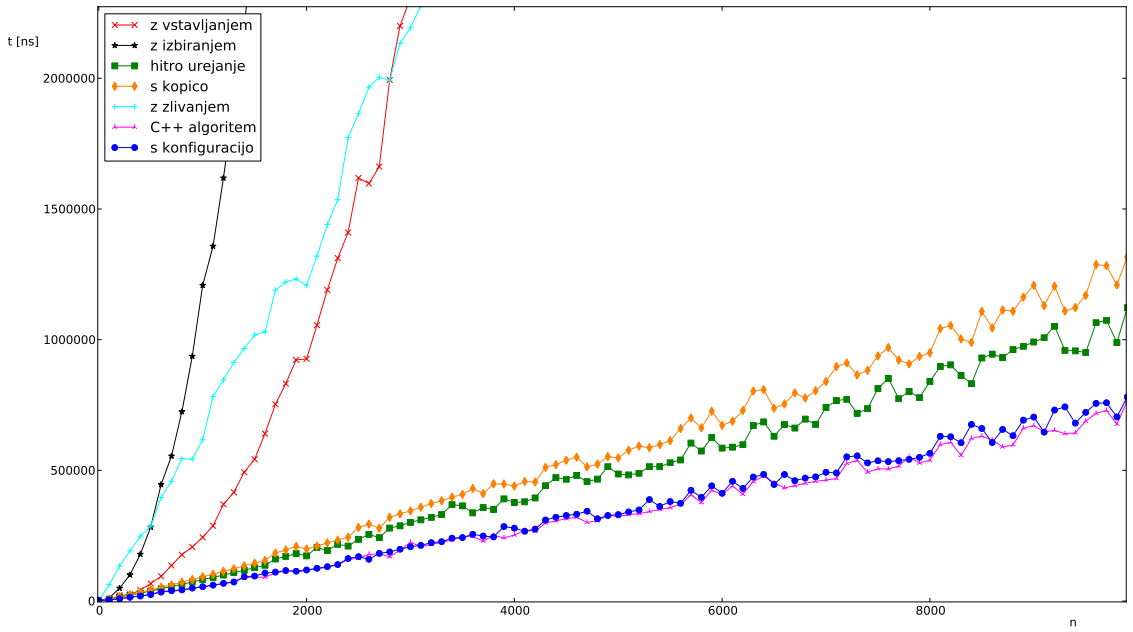
Slika 8: Rezultati za tip *int*, 1.000 elementov.

Graf prikazuje čase urejanja različnih sortirnih algoritmov v odvisnosti od dolžine polja. Polja so dolga od 1 do 1.000 elementov. Za vsako dolžino polja je program naredil 1.000 iteracij. Prikazan je le del grafa, da se vidi primerjava med učinkovitejšimi urejanji.



Slika 9: Rezultati za tip *int*, 10.000 elementov.

Graf prikazuje čase urejanja različnih sortirnih algoritmov v odvisnosti od dolžine polja, vse od dolžine 1 do 10.000. Program je za vsako dolžino polja naredil 100 iteracij. Samo spodnji del grafa je prikazan na sliki 10.



Slika 10: Rezulati za tip *int*, 10.000 elementov.

Na sliki je prikazan spodnji del grafa na sliki 9 za boljšo primerjavo med učinkovitejšimi algoritmi.

3.2 Tip *string*

Tip *string* predstavlja niz znakov poljubne dolžine. Nizi, ki sem jih urejal, so bili dolgi od enega do dvanajst znakov in so v pomnilniku zavzeli sorazmerno malo prostora, vendar več kot *int*. Primerjava dveh nizov znakov lahko traja sorazmerno dolgo, saj se nizi primerjajo leksikografsko, najprej po prvi črki, nato po drugi in tako vse do zadnje. Primerjanje črke z drugo traja tako dolgo kot primerjanje dveh objektov tipa *int*. Primerjava dveh nizov znakov tako traja najmanj tako dolgo kot primerjava dveh števil, lahko pa tudi veliko dlje, odvisno od števila enakih zaporednih črk med nizoma.

Najdena optimalna konfiguracija za tip *string* je sledeča:

$$: s \longrightarrow 5; : q \longrightarrow 3454; : m \longrightarrow 3844; : q \longrightarrow 4948;$$
$$: h \longrightarrow 6154; : m \longrightarrow 6530; : h \longrightarrow -1; \quad .$$

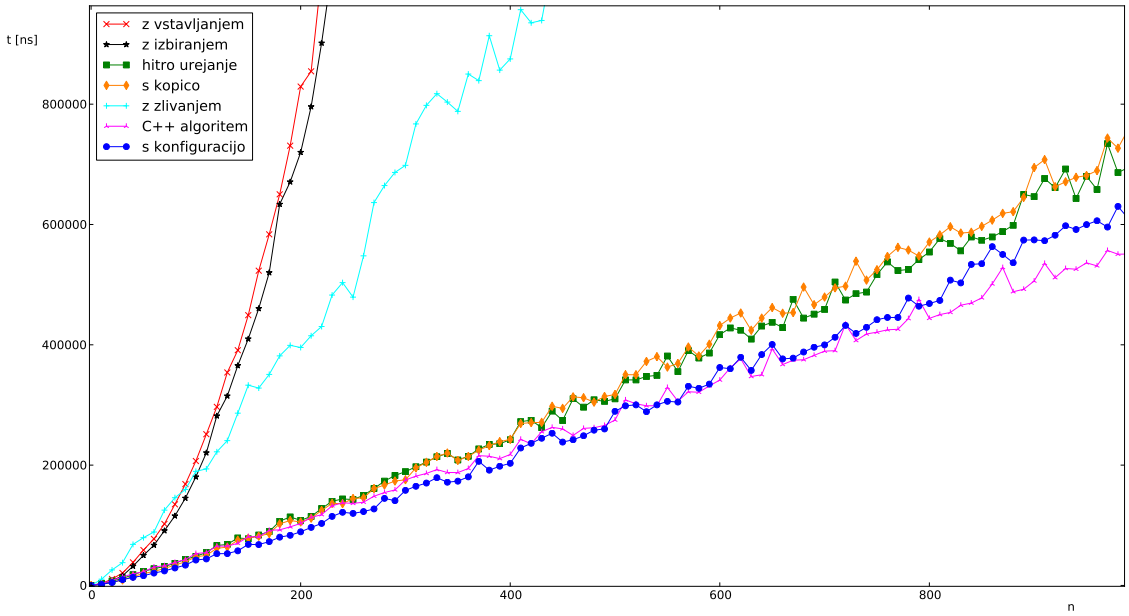
Tabela 3: Rezultati za tip *string*.

št. elementov **št. iteracij**	**10 el.** **100.000 it.**	**100 el.** **10.000 it.**	**1.000 el.** **1.000 it.**	**10.000 el.** **100 it.**
urejanje	**čas [μs]**	**čas [μs]**	**čas [μs]**	**čas[μs]**
z vstavljanjem	2,24	160,40	18.925,12	2.077.954,54
z izbiranjem	1,53	151,75	18.456,58	2.146.882,90
hitro urejanje	2,44	44,26	704,11	15.663,25
s kopico	1,71	40,82	736,21	10.516,13
z zlivanjem	8,66	168,17	2.358,61	38.040,68
C++ algoritem	2,81	42,44	573,08	6.094,49
s konfiguracijo	1,55	35,78	644,23	10.576,94

Tabela 4: Skupen čas urejanja za tip *string*.

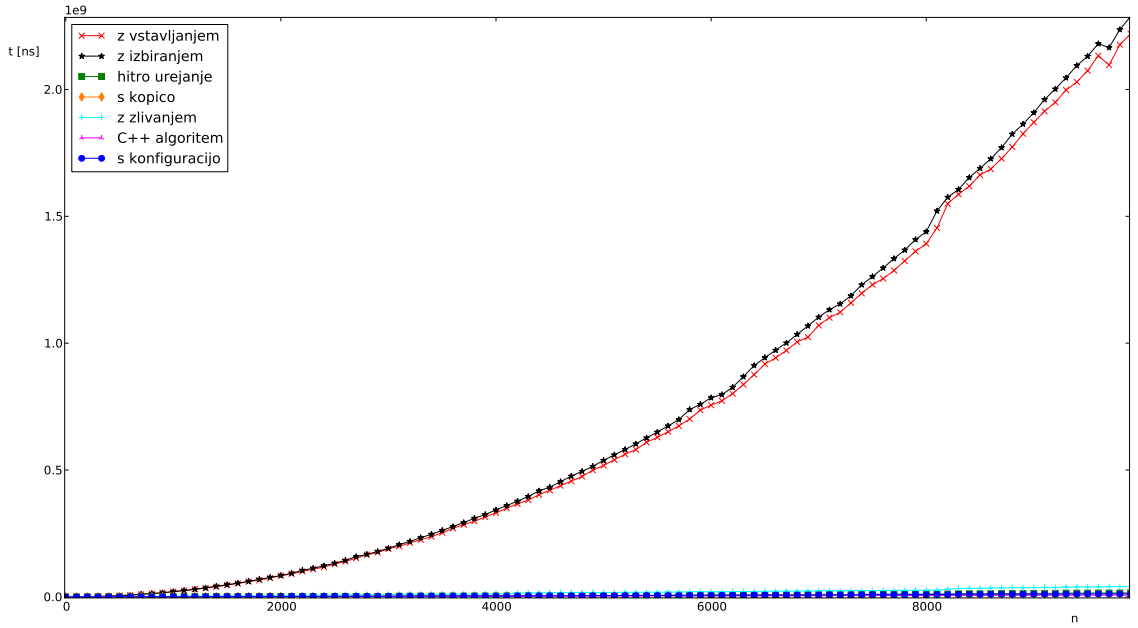
Skupne čase urejanja smo dobili tako, da za posamezen algoritem seštejemo ordinate vseh točk na sliki 12, ki prikazujejo čase tega sortirnega algoritma. Vrednosti so zaradi preglednosti urejene naraščajoče.

sortirni algoritem	**čas[ns], $1 \leq n \leq 10.000$**
vgrajeni algoritem v C++	331.908.132
urejanje s konfiguracijo	518.292.327
urejanje s kopico	540.306.780
hitro urejanje	636.750.157
urejanje z zlivanjem	1.752.806.479
urejanje z vstavljanjem	72.328.942.517
urejanje z izbiranjem	74.442.461.661



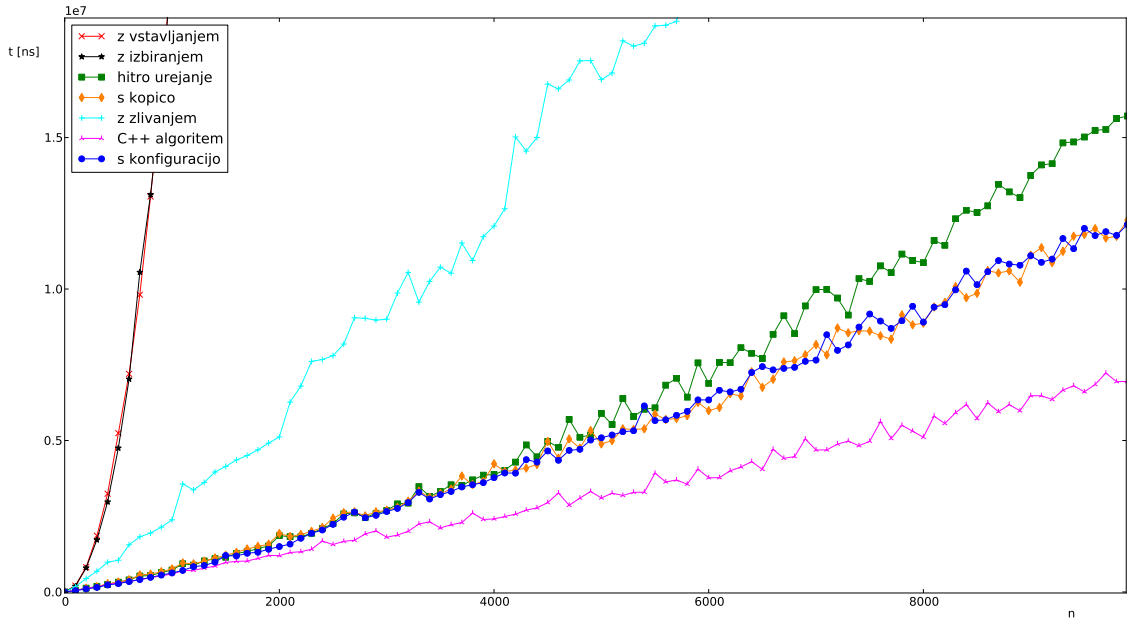
Slika 11: Rezultati za tip *string*, 1.000 elementov.

Graf prikazuje čase urejanja različnih sortirnih algoritmov za tip *string* v odvisnosti od dolžine polja, vse od dolžine 1 do 1.000. Program je za vsako dolžino polja naredil 100 iteracij. Približan je tako, da kaže zgolj popolne grafe učinkovitejših algoritmov za boljšo primerjavo.



Slika 12: Rezultati za tip *string*, 10.000 elementov.

Graf prikazuje čase urejanja različnih sortirnih algoritmov za tip *string* v odvisnosti od dolžine polja, vse od dolžine 1 do 10.000. Program je za vsako dolžino polja naredil 10 iteracij. Spodnji del tega grafa je prikazan na sliki 13.



Slika 13: Rezultati za tip *string*, 10.000 elementov.

Slika prikazuje le spodnji del grafa na sliki 12 za boljšo primerjavo med učinkovitejšimi algoritmi.

3.3 Tip *huge*

Tip *huge* predstavlja uporabniški tip, katerega glavna lastnost je, da zavzame veliko pomnilnika. Primerjava dveh objektov tega tipa traja tako dolgo kot primerjava dveh števil. Posamezen objekt zavzame v pomnilniku veliko več prostora kot povprečen *string* in zato tudi njegovo kopiranje traja dlje.

Najdena optimalna konfiguracija za kompozitni sortirni algoritem za tip *huge* je sledeča:

$$: i \longrightarrow 12; : s \longrightarrow 68; : q \longrightarrow 100; : s \longrightarrow 148; : q \longrightarrow 189; : s \longrightarrow 287; : q \longrightarrow -1; \quad .$$

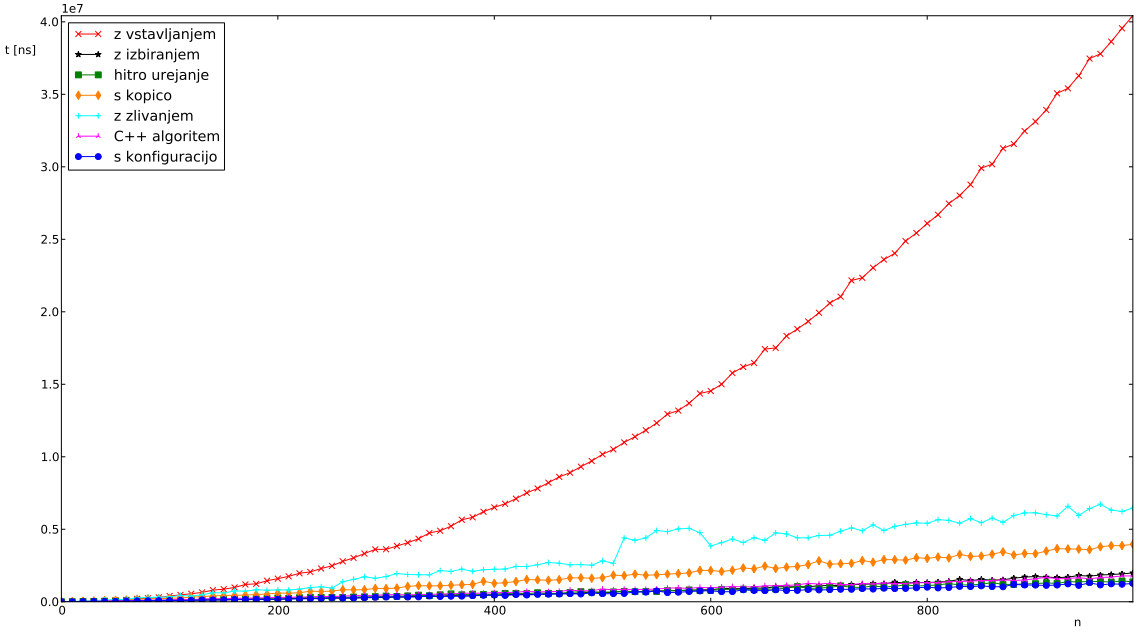
Tabela 5: Rezultati za tip *huge*.

št. elementov **št. iteracij**	**10 el.** **10.000 it.**	**100 el.** **1.000 it.**	**1.000 el.** **100 it.**	**10.000 el.** **10 it.**
urejanje	**čas [μs]**	**čas [μs]**	**čas [μs]**	**čas[μs]**
z vstavljanjem	5,32	306,65	41.567,63	13.789.989,83
z izbiranjem	5,57	89,48	2.034,52	177.534,11
hitro urejanje	6,21	102,77	1.386,03	23.740,07
s kopico	9,09	233,28	3.845,66	64.870,91
z zlivanjem	12,26	364,26	6.311,54	212.215,60
C++ algoritem	6,53	123,15	1.898,45	27.993,51
s konfiguracijo	5,78	77,41	1.215,61	21.840,79

Tabela 6: Skupen čas urejanja za tip *huge*.

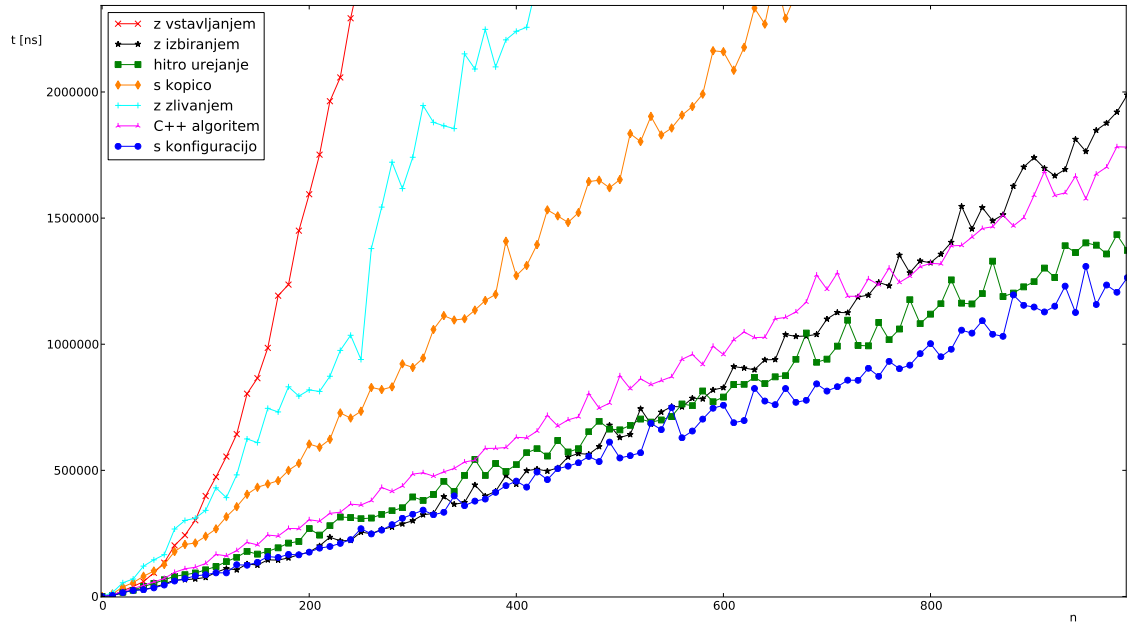
Skupen čas urejanja je pravzaprav vsota časov urejanja posameznega algoritma, ki so prikazani na grafu na sliki 14. Vrednosti so zaradi preglednosti urejene naraščajoče.

sortirni algoritem	**čas[ns], $1 \leq n \leq 1.000$**
urejanje s konfiguracijo	58.426.372,19
hitro urejanje	67.858.490,70
urejanje z izbiranjem	75.267.185,97
vgrajeni algoritem v C++	81.821.545,08
urejanje s kopico	177.185.498,27
urejanje z zlivanjem	320.115.702,80
urejanje z vstavljanjem	1.341.428.566,28



Slika 14: Rezultati za tip *huge* za polja z dolžino manjšo od 1.000.

Graf prikazuje čase urejanja različnih sortirnih algoritmov za tip *huge* v odvisnosti od dolžine polja, vse od dolžine 1 do 1.000. Za vsako dolžino polja je program naredil 100 iteracij. Samo spodnji del grafa je prikazan na sliki 15.



Slika 15: Rezultati za tip *huge*, 1.000 elementov.

Slika prikazuje samo spodnji del grafa na sliki 14 za boljšo primerjavo med učinkovitejšimi algoritmi.

3.4 Tip *slow*

Tip *slow* predstavlja uporabniški tip, katerega glavna lastnost je, da primerjanje dveh objektov traja zelo dolgo. Primerjava dveh objektov tako traja veliko več časa kot primerjava dveh povprečnih nizov znakov. V pomnilniku zavzame zelo malo prostora, približno toliko kot *int*.

Najdena optimalna konfiguracija za kompozitni sortirni algoritem za tip *slow* je sledeča:

$$: i \longrightarrow 7; : m \longrightarrow -1; \quad .$$

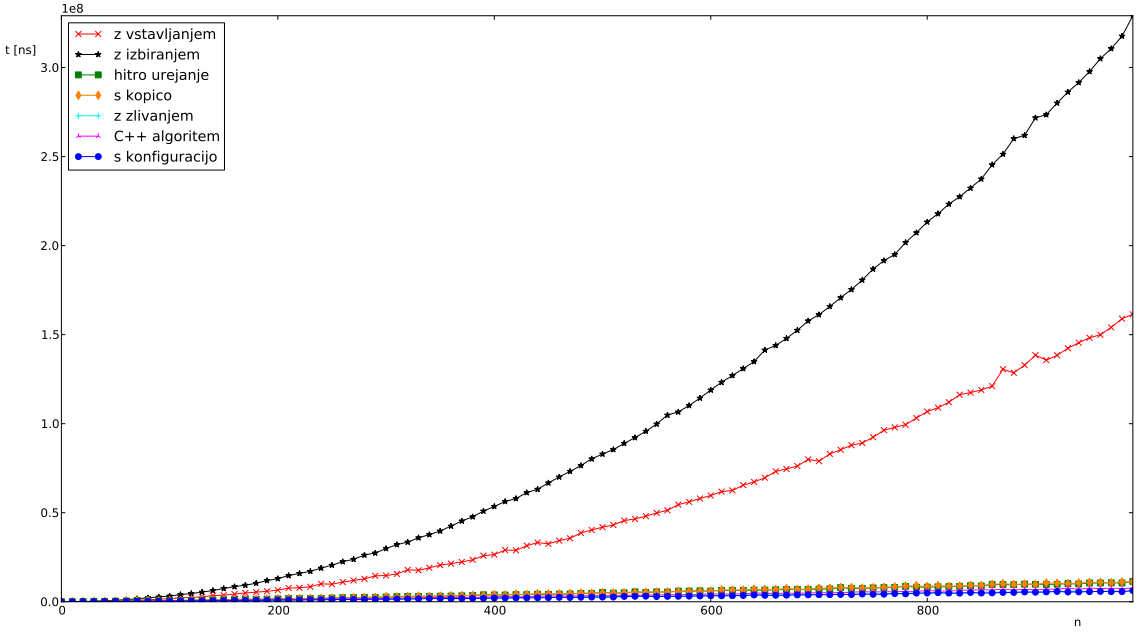
Tabela 7: Rezultati za tip *slow*.

št. elementov **št. iteracij**	**10 el.** **10.000 it.**	**100 el.** **1.000 it.**	**1.000 el.** **100 it.**	**10.000 el.** **10 it.**
urejanje	**čas [μs]**	**čas [μs]**	**čas [μs]**	**čas[μs]**
z vstavljanjem	17,31	1.703,821	166.257,99	16.525.726,29
z izbiranjem	29,31	3.308,363	329.950,63	33.201.205,85
hitro urejanje	39,48	749,242	10.871,17	146.598,49
s kopico	24,97	681,290	11.211,39	157.949,93
z zlivanjem	18,33	400,140	6.117,13	85.727,57
C++ algoritem	20,37	506,029	7.745,60	105.082,31
s konfiguracijo	16,75	382,914	5.973,63	83.143,84

Tabela 8: Skupen čas urejanja za tip *slow*.

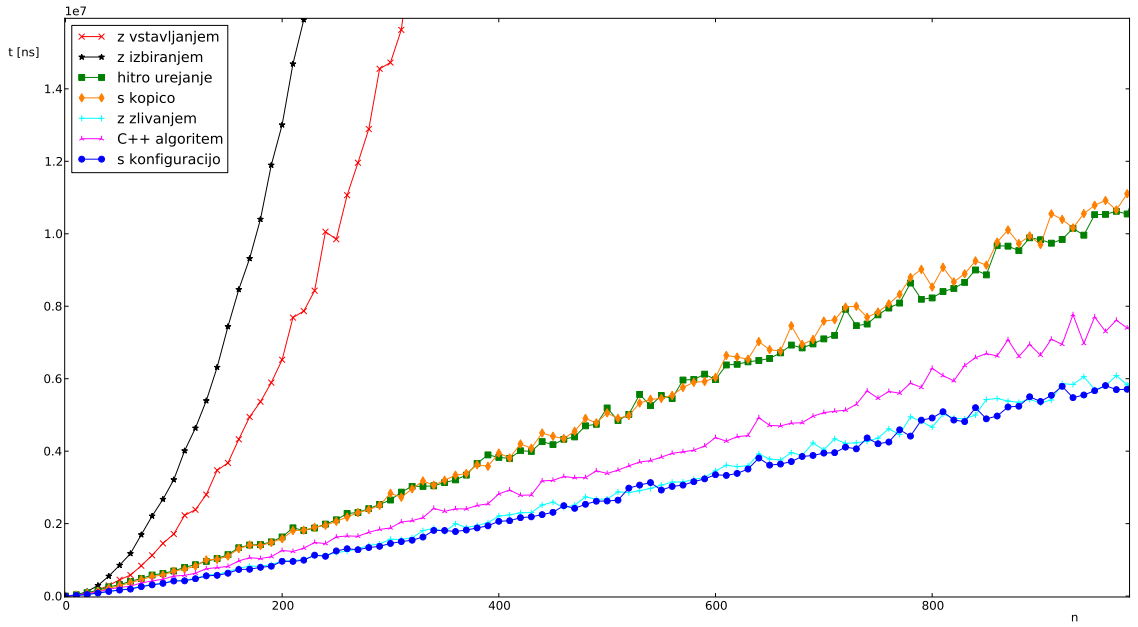
Skupen čas urejanja za vsak posamezen algoritem dobimo tako, da seštejemo čase, ki jih je porabil pri vsakem n posebej. Skupen čas urejanja pravzaprav predstavlja približek ploščine, ki jo na določenem intervalu omejita graf soritrnega algoritma in abscisna os na grafu na sliki 16. Vrednosti so zaradi preglednosti urejene naraščajoče.

sortirni algoritem	**čas[ns], $1 \leq n \leq 1.000$**
urejanje s konfiguracijo	275.974.571,31
urejanje z zlivanjem	281.640.609,23
vgrajeni algoritem v C++	353.613.310,76
hitro urejanje	501.905.163,63
urejanje s kopico	505.171.708,97
urejanje z vstavljanjem	5.427.392.234,82
urejanje z izbiranjem	10.826.003.228,79



Slika 16: Rezultati za tip *slow* za polja z dolžino manjšo od 1.000.

Graf prikazuje čase urejanja različnih sortirnih algoritmov za tip *slow* v odvisnosti od dolžine polja, vse od dolžine 1 do 1.000. Za vsako dolžino polja je program naredil 10 iteracij. Samo spodnji del grafa je prikazan na sliki 17.



Slika 17: Rezultati za tip *slow*, 1.000 elementov.

Slika prikazuje zgolj spodnji del grafa na sliki 16 za boljšo primerjavo med učinkovitejšimi algoritmi.

4 Ugotovitve in razprava

4.1 Tip *int*

Dobljena konfiguracija : $s \longrightarrow 5;: i \longrightarrow 67;: q \longrightarrow -1;$ se ujema s teoretičnim delom. Urejanje z izbiranjem in urejanje z vstavljanjem sta dobra na kratkih poljih, na daljših poljih pa je boljše hitro urejanje kot zelo hiter algoritem na naključno porazdeljenih podatkih.

Rezultati v tabeli 1 za dolžino 10 kažejo podobno sliko. Urejanje z izbiranjem in urejanje z vstavljanjem sta boljša od vseh ostalih sortirnih algoritmov, urejanje s konfiguracijo in C++-ov algoritem, ki je prav tako kompoziten, sta za spoznanje slabša. Pri urejanju s konfiguracijo je to posledica iskanja po konfiguraciji, česar urejanju z vstavljanjem ni potrebno delati. Sicer se urejanje s konfiguracijo obnaša popolnoma enako kot urejanje z vstavljanjem, saj vidimo, da se polja dolžine 10 urejajo z le-tem. Urejanje z zlivanjem je pri tej dolžini daleč najslabše, saj naredi veliko rekurzivnih klicev in kopiranja elementov, kar je pri tako kratki dolžini zelo zamudno.

Pri stotih elementih se vrstni red že spremeni. Močno nazaduje urejanje z izbiranjem zaradi kvadratne časovne zahtevnosti tudi v najboljšem primeru. Urejanje z vstavljanjem je še vedno na prvem mestu med nekompozitnimi algoritmi, medtem ko se pri urejanju s konfiguracijo že kaže njegova kompozitna narava, saj pri poljih dolžine 100 že uporablja hitro urejanje, ki polja razdeli na manjša podpolja, ki se potem, ko je njihova dolžina manjša ali enaka 67, uredijo z vstavljanjem. Tako dosežemo hitrejše urejanje, kar se pozna že pri 100 elementih. Urejanje z zlivanjem je še vedno na zadnjem mestu, saj število elementov še ni dovolj veliko, da bi se vsi rekurzivni klici in kopiranje elementov splačali. Tudi na sliki 8 se vidijo podobne ugotovitve. Urejanje z zlivanjem in urejanje z izbiranjem sta zelo počasna, sledi jima urejanje z izbiranjem, nato urejanje s kopico in tik za njim hitro urejanje. C++-ov algoritem in urejanje s konfiguracijo se praktično prekrivata, vendar je urejanje s konfiguracijo za približno $n < 600$ zagotovo boljše od C++-ovega algoritma, za n-je večje od 600 pa v večini primerov, a že vidimo, da se približuje njuno presečišče.

Pri tisočih elementih pa se že pokaže pričakovana časovna zahtevnost algoritmov. Hitro urejanje in urejanje s kopico napredujeta na prvo in drugo mesto med nekompozitnimi algoritmi. Urejanje z zlivanjem uspe prehiteti le urejanje z izbiranjem, ki je močno zasidrano na zadnjem mestu s kvadratno časovno zahtevnostjo. Urejanje z vstavljanjem zaradi svoje kvadratne časovne zahtevnosti v povprečnem primeru nazaduje, a je še vedno boljše od urejanja z zlivanjem. Kompozitna algoritma zdaj že močno pokažeta svoje prednosti, urejanje s konfiguracijo se namreč obnaša popolnoma enako kot pri 100 elementih in je še malce boljše od vgrajenega algoritma v C++.

Pri deset tisoč elementih se dokončno pokažejo pričakovane časovne zahtevnosti algoritmov. Lepo vidne so tudi na sliki 9, ko se vidijo zgolj urejanje z izbiranjem, urejanje z vstavljanjem in urejanje z zlivanjem. Urejanje z zlivanjem napreduje, čeprav je še vedno močno počasnejše kot hitro urejanje ali urejanje s kopico, medtem ko se urejanje z vstavljanjem premakne na predzadnje mesto. Oba kompozitna algoritma sta precej boljša od ostalih algoritmov, kot se vidi na sliki 10. Spremeni se tudi prvo mesto – vgrajeni algoritem v C++ prehiti urejanje s konfiguracijo. To spremembo smo že predhodno slutili, saj sta se drug drugemu čedalje bolj približevala. C++-ov algoritem je boljši na daljših poljih, ker ima boljšo implementacijo osnovnih

algoritmov in je tudi prilagojen za vgrajene tipe v C++.

Pri sto tisoč elementih se le dokončno potrdijo mesta algoritmov, algoritma s kvadratno časovno zahtevnostjo sta 32-krat in 170-krat slabša od urejanja z zlivanjem, ter kar 200-krat in kar 1.000-krat slabša od hitrega urejanja. Kompozitna algoritma sta na prvem mestu, s tem, da je urejanje s konfiguracijo na drugem mestu.

V tabeli 2 vidimo seštevek vseh časov urejanja pri vseh dolžinah za vsak posamezen algoritem. Vidimo, da je najboljši C++-ov algoritem, tik za njim pa urejanje s konfiguracijo. Oba algoritma sta primerna za urejanje polj s števili, urejanje s konfiguracijo bi bilo celo boljše, če bi gledali ploščine pod grafi samo do 1.000 elementov. Če bi gledali ploščine do 100.000 elementov, pa bi bil C++-ov algoritem še boljši kot sedaj. Algoritmu, vgrajenemu v C++, in urejanju s konfiguracijo sledita hitro urejanje in urejanje s kopico, nato pa urejanje z zlivanjem. Urejanje z vstavljanjem se je kljub svoji pričakovani kvadratni časovni zahtevnosti uvrstilo precej blizu urejanja z zlivanjem. Urejanje z izbiranjem je na zadnjem mestu in se je izkazalo skrajno neprimerno za urejanje polj s števili.

4.2 Tip *string*

Konfiguracija za ta tip je : $s \longrightarrow 5$; : $q \longrightarrow 3454$; : $m \longrightarrow 3844$; : $q \longrightarrow 4948$; : $h \longrightarrow 6154$; : $m \longrightarrow 6530$; : $h \longrightarrow -1$; in se precej razlikuje od tiste, dobljene za tip *int*. Urejanje s pričakovano kvadratno časovno zahtevnostjo se pojavi zgolj na začetku in še to samo za polja do pet elementov. Nato se vse do konca uporabljajo algoritmi s časovno zahtevnostjo $\mathcal{O}(n \log_2 n)$. Proti koncu se tudi hitro urejanje izkaže za manj učinkovito (ne pozabimo, da ima kvadratno časovno zahtevnost v najslabšem primeru) in ga premagata urejanje z zlivanjem in urejanje s kopico, ki imata vedno časovno zahtevnost $\mathcal{O}(n \log_2 n)$.

Pri desetih elementih je najboljše urejanje z izbiranjem, kot lahko vidimo v prvem stolpcu tabele 3. Uporaba tega urejanja kaže na to, da je pri nizih znakov bolj učinkovito opravljati več primerjav in manj kopiranj. Njegov tekmec z enako časovno zahtevnostjo, urejanje z vstavljanjem, je bil tu slabši, saj opravlja manj primerjanj in več kopiranj, kar se pri nizih znakov, ki so nekajkrat večji od števil, ne splača. Urejanje z zlivanjem je podobno kot pri tipu *int* najslabše, saj se veliko kopiranja in rekurzivnih klicev pri tako kratkih poljih ne splača. Zelo dober rezultat je doseglo tudi urejanje s kopico, kar kaže na večjo učinkovitost nerekurzivnih algoritmov pri kratkih poljih. Urejanje s konfiguracijo je sicer že sestavljeno, in sicer iz hitrega urejanja in urejanja z izbiranjem. Malo počasnejše je le od slednjega, zaradi iskanja po konfiguraciji in rekurzivnih klicev. V primerjavi s poljem števil traja v povprečju približno desetkrat dlje, da uredimo polje nizov, kar potrjuje, da se nizi počasneje primerjajo in kopirajo.

Pri stotih elementih obe urejanji s kvadratno časovno zahtevnostjo že nazadujeta, slabše je le urejanje z zlivanjem, ki se mu vse kopiranje v dodaten pomnilnik in rekurzivni klici še vedno ne splačajo. Urejanje s kopico kot nerezkurziven algoritem z ustreznim razmerjem med primerjavami in zamenjavami je na prvem mestu med nekompozitnimi algoritmi. Urejanje s konfiguracijo že pokaže svoje prednosti, saj prehiti vse algoritme, tudi vgrajeni algoritem v C++. Pri hitrem urejanju je število rekurzivnih klicev še vedno preveliko, da bi se splačalo. Na sliki 11 vidimo, kako je urejanje s konfiguracijo do malo manj kot 600 elementov v polju boljše od vgrajenega

algoritma v C++, potem pa se situacija obrne. Oba algoritma s kvadratno časovno zahtevnostjo precej hitro izgineta s slike, saj sta že pri pri 100 elementih v polju popolnoma nekonkurenčna ostalim. Prav tako je s svojim izdatnim kopiranjem elementov počasno tudi urejanje z zlivanjem.

Pri tisočih elementih močno napredujeta algoritem, vgrajen v C++, in hitro urejanje. Pri C++-ovemu algoritmu se vidi njegova splošna optimizirana implementacija za vgrajene tipe in za daljša polja. Hitremu urejanju se rekurzivni klici splačajo, saj je malo boljši od še vedno odličnega urejanja s kopico. Urejanje s konfiguracijo je sicer malo slabše od vgrajenega algoritma v C++, a še vedno bolje od vseh nekompozitnih algoritmov. Oba algoritma s kvadratno časovno zahtevnostjo sta močno na zadnjih dveh mestih, urejanje z izbiranjem še vedno malo boljše od urejanja z vstavljanjem kljub kvadratni časovni zahtevnosti v vsakem primeru.

Pri deset tisoč elementih je C++-ov algoritem še vedno zanesljivo na prvem mestu. Urejanje s konfiguracijo ni več kompozitno, uporablja le urejanje s kopico, zato je tudi njegov rezultat malce slabši od urejanja s kopico samega (kot vidno na sliki 13) zaradi branja konfiguracije. Za polja z dolžinami od 6.155 do 6.530 urejanje s konfiguracijo uporablja urejanje z zlivanjem, čeprav je precej slabše od ostalih algoritmov s podobno časovno zahtevnostjo. Urejanje s konfiguracijo ga kljub temu uporablja, ker izkorišča njegovo rekurzivno naravo in krajša polja uredi s primernejšim algoritmom. Urejanje z zlivanjem pa tudi krajša polja uredi z zlivanjem, kar se, kot smo videli pri desetih ali stotih elementih, ne splača in je zato precej počasno. Pri poljih z deset tisoč elementi je torej najboljši nekompozitni algoritem urejanje s kopico, sledi hitro urejanje. Tudi na zadnjem mestu se zgodi zamenjava, urejanje z izbiranjem postane slabše od urejanja z vstavljanjem. Zadnje mesto lahko pripiše svoji kvadratni časovni zahtevnosti v vseh primerih. Oba algoritma sta popolnoma neprimerna za dolga polja, saj sta kar približno 200-krat počasnejša od urejanja s kopico. To se tudi vidi na sliki 12, saj se drugih algoritmov praktično ne vidi. Tak vrsti red bi se verjetno nadaljeval tudi naprej, razlike med algoritmi bi se samo povečevale.

Poglejmo še splošno učinkovitost sortirnih algoritmov, prikazano v tabeli 4. Vidimo, da je daleč najboljši vgrajeni algoritem v C++, sledi mu urejanje s konfiguracijo. Podobno kot pri tipu *int*, bi bilo tudi tukaj urejanje s konfiguracijo boljše, če bi vzeli manjši interval dolžin, saj je bolj učinkovito na krajših poljih. Na daljših poljih je veliko bolj učinkovit C++-ov algoritem. Urejanju s konfiguracijo tesno sledi urejanje s kopico, ki ga urejanje s konfiguracijo tudi uporablja za velika polja. Ker velika polja prispevajo večji delež k skupnemu času, je torej razumljivo, da je urejanje s kopico tako blizu urejanju s konfiguracijo. Pravzaprav bi se njuna razlika pri večanju intervala zmanjševala. Hitro urejanje je pri poljih z nizi znakov malce manj učinkovito, veliko manj učinkovito pa je urejanje z zlivanjem. Urejanje z vstavljanjem in urejanje z izbiranjem sta na splošno zelo neprimerni za urejanje polj z nizi znakov, kar vidimo iz njunega skupnega časa, ki je kar 125 večji od časa, ki ga je porabilo urejanje s kopico.

4.3 Tip *huge*

V konfiguraciji : $i \longrightarrow 12$; : $s \longrightarrow 68$; : $q \longrightarrow 100$; : $s \longrightarrow 148$; : $q \longrightarrow 189$; : $s \longrightarrow 287$; : $q \longrightarrow -1$; vidimo, da se, kot vedno do sedaj, na začetku uporablja algoritem s kvadratno časovno zahtevnostjo. Kasneje se izmenjujeta urejanje z izbiranjem

in hitro urejanje, ki se na koncu tudi izkaže za učinkovitejše. To izmenjevanje ni posledica eksperimentalnih napak, pomeni le, da sta takrat urejanje z izbiranjem in urejanje z vstavljanjem približno enako dobra. Ker sta njuni krivulji zelo skupaj imata zelo veliko presečišč, ki se kažejo tudi v konfiguraciji in so lahko tudi posledica obnašanja hitrega urejanja, ki je odvisno od vhodnih podatkov.

Pri desetih elementih v polju sta najboljša urejanje z vstavljanjem in urejanje z izbiranjem, kot lahko vidimo v tabeli 5. Urejanje s konfiguracijo se ne obnaša kompozitno, saj ureja le z vstavljanjem. Slabše je tako od urejanja z vstavljanjem kot od urejanja z izbiranjem, saj ne more pridobivati hitrosti na kompozitnosti, temveč jo kvečjemu izgublja z nepotrebnim iskanjem po konfiguraciji. Urejanje z zlivanjem je daleč najslabše, saj se je kopiranje že v prejšnjih primerih izkazalo kot neučinkovito, kopiranje velikih elementov pa je še počasnejše.

Pri stotih elementih je urejanje s konfiguracijo že na prvem mestu. Uporablja hitro urejanje, ki je na drugem mestu med nekompozitnimi algoritmi. Že pri poljih dolžine 100 se lepo vidi, kako velikost elementov vpliva na učinkovitosti sortirnih algoritmov. Urejanje z zlivanjem je še vedno zadnje, saj se kopiranje veliki elementov ne splača kljub časovni zahtevnosti $\mathcal{O}(n \log_2 n)$. Urejanje z vstavljanjem, ki prav tako veliko kopira, je na drugem mestu. Po drugi strani je urejanje z izbiranjem, ki kopira zelo malo, primerja pa zelo veliko, kljub svoji kvadratni časovni zahtevnosti na prvem mestu med nesestavljenimi algoritmi. Dobro je tudi hitro urejanje, slabše pa je urejanje s kopico, ki elemente veliko kopira, saj jih preureja v kopico ter nato zamenja koren in spet naredi kopico, nato step zamenja koren ... C++-ov algoritem je nekje na sredini. Tudi na sliki 15 se vidi, da je to uporabniški tip, za katerega ni optimiziran. Pri tisočih elementih je najbolj opazna sprememba močno zadnje mesto urejanja z vstavljanjem (presečišče z urejanjem z zlivanjem ima na sliki 15 približno pri $n = 110$). K temu pripomore predvsem veliko kopiranja in kvadratna časovna zahtevnost v povprečnem primeru. Urejanje z zlivanjem je napredovalo na račun urejanja z vstavljanjem, vendar je še vedno na predzadnjem mestu zaradi kopiranja celotnega polja v dodaten pomnilnik. To se vidi na sliki 14, saj so zadnji trije algoritmi praktično edini vidni na sliki. Vidi se (slika 15), da urejanje s konfiguracijo v večini uporablja hitro urejanje, ki je na skupnem drugem mestu, sledi mu vgrajeni algoritem v C++, ki se mu sicer obrestuje njegova kompozitna narava, vendar ni optimiziran za uporabniške tipe. Urejanje z izbiranjem je malo nazadovalo, vendar je kljub svoji kvadratni časovni zahtevnosti še vedno boljše od urejanja s kopico. Zelo malo kopiranj elementov se tudi pri poljih dolžine tisoč še vedno zelo obrestuje.

Pri deset tisoč elementih zaradi kvadratne časovne zahtevnosti nazaduje urejanje z izbiranjem. Vrstni red ostalih se ne spremeni bistveno, poudarijo pa se razlike med algoritmi. Urejanje z vstavljanjem je na zadnjem mestu, kar 630-krat slabše od urejanja s konfiguracijo. Urejanje z zlivanjem je močno zadnje med algoritmi s pričakovano časovno zahtevnostjo $\mathcal{O}(n \log_2 n)$. Urejanje s kopico je malo boljše, najboljše pa je hitro urejanje, ki od teh očitno naredi najmanj kopiranj. Med urejanjem s kopico in hitrim urejanjem je C++-ov algoritem, urejanje s konfiguracijo pa je tudi kar močno zasidrano na prvem mestu.

Če pogledamo splošno učinkovitost sortirnih algoritmov, prikazano v tabeli 6, pri tem tipu vidimo, da je v primerjavi s prejšnjimi tipi tu precej nazadoval algoritem, vgrajen v C++. Najboljše je urejanje s konfiguracijo, s kar nekaj prednosti pred hitrim urejanjem. Hitremu urejanju presenetljivo sledi urejanje z izbiranjem, ki je bilo v prejšnjih tipih najslabše, tu pa je zaradi izredno malo premikov, ki jih naredi

med urejanjem, doseglo odlično tretje mesto. Šele četrti je vgrajeni algoritem v C++, ki se mu vidi, da ni optimiziran za uporabniške tipe. Vsi štirje zgoraj omenjeni algoritmi so kar primerni za urejanje polj z elementi tipa *huge*. Peto in šesto mesto sta zasedla urejanje s kopico in urejanje z zlivanjem, ki naredita kar veliko kopiranj elementov. Urejanje z vstavljanjem, ki prav tako naredi veliko kopiranj, poleg tega pa ima še pričakovano kvadratno časovno zahtevnost, je daleč na zadnjem mestu in zelo neprimerno za uporabo na poljih z elementi tip *huge*.

4.4 Tip *slow*

V konfiguraciji : $i \longrightarrow 7;$: $m \longrightarrow -1;$, dobljeni za tip *slow*, vidimo, da se najprej uporablja urejanje z vstavljanjem, nato pa urejanje z zlivanjem. Ker primerjanje dveh elementov tipa *slow* traja precej dolgo, so dobri tisti algoritmi, ki naredijo sorazmerno malo primerjav in več kopiranj.

V tabeli 7 vidimo, da je pri desetih elementih najboljši nekompozitni algoritem urejanje z vstavljanjem, sledi mu urejanje z zlivanjem. Situacija je ravno obratna kot pri tipu *huge*. Ti dve urejanji naredita najmanj primerjanj elementov, zato sta tudi najhitrejši kljub rekurzivni naravi urejanja z zlivanjem. Urejanje s konfiguracijo uporablja obe najboljši nekompozitni urejanji in je z malo prednosti na prvem mestu. Najslabše je presenetljivo hitro urejanje, drugo najslabše pa urejanje z izbiranjem, ki naredi ogromno primerjav in zelo malo premikov. Hitro urejanje tudi naredi precej veliko primerjav in je poleg tega še rekurzivno, kar se, kot smo videli tudi na prejšnjih primerih, ne splača na tako kratkih poljih. Urejanje s kopico je kot nerekurziven algoritem na tretjem mestu med nekompozitnimi. V širši konkurenci ga je premagal tudi vgrajeni algoritem v C++, ki se mu zapletena implementacija in optimizacija pri tako kratkih poljih ne splačata. Vidimo tudi, da urejanje polj tipa *slow* traja približno trikrat več časa kot urejanje polj elementov tipa *huge*, desetkrat več časa kot urejanje polja z nizi znakov in kar več kot stokrat dlje kot urejanje polja števil.

Pri poljih dolgih sto elementov najprej opazimo, da je urejanje z vstavljanjem nazadovalo iz drugega mesta na predzadnje. To lahko v celoti pripišemo njegovi kvadratni časovni zahtevnosti. Urejanje z zlivanjem je najboljše med nekompozitnimi algoritmi, saj naredi zelo malo primerjanj elementov in se mu splača celó kopiranje celotnega polja v dodaten pomnilnik. Urejanje z izbiranjem je močno na zadnjem mestu zaradi ogromnega števila primerjav in kvadratne časovne zahtevnosti, hitro urejanje pa je še vedno slabše od urejanja s kopico. C++-ov algoritem je še vedno na tretjem mestu s kar velikim zaostankom. Enak vrstni red se vidi tudi na slikah 16 in 17. Na prvi se vidi, da sta urejanji s pričakovano kvadratno časovno zahtevnostjo daleč najslabši, na drugi pa že zgoraj opisani vrstni red.

Pri tisoč elementih se zgodi le ena zamenjava v vrstnem redu, hitro urejanje postane učinkovitejše od urejanja s kopico, kar lahko vidimo na sliki 17 kot njuno presečišče pri približno $n = 600$. Ker imata oba enako povprečno časovno zahtevnost, hitro urejanje pa celo slabšo, je to lahko le posledica principa deli in vladaj, torej se je rekurzivnost hitremu urejanju obrestovala pri malce daljših poljih. Ostali vrstni red algoritmov ostane popolnoma enak, razlike med algoritmi se samo povečajo.

Pri poljih dolgih deset tisoč elementov vrstni red algoritmov ostane enak, povečajo se zgolj razlike med njimi. Daleč spredaj sta urejanje s konfiguracijo, ki mu tesno sledi urejanje z zlivanjem z najmanj izvedenimi primerjavami in zato najboljšim rezultatom med nekompozitnimi algoritmi. Tretji je C++ algoritem, nato hitro ure-

janje, urejanje s kopico, urejanje z vstavljanjem in na zadnje mesto prikovano urejanje z izbiranjem z ogromnim številom primerjav in kvadratno časovno zahtevnostjo v vsakem primeru. Pri večanju dolžine polja bi se verjetno vrstni red ohranjal, le razlike med algoritmi bi se povečevale. Vidimo tudi, da urejanje polja z 10.000 elementi tipa *slow* traja kar 35-krat dlje od urejanja polja z elementi tipa *int* enake dolžine, če primerjamo najučinkovitejša algoritma. Če pa primerjamo najslabša algoritma, pa urejanje polja z elementi tipa *slow* traja kar 2.600-krat dlje kot urejanje polja z elementi tipa *int*.

Če pogledamo skupen čas urejanja algoritmov v tabeli 8, je razvidno, da je na splošno najboljše urejanje s konfiguracijo. Vidi se mu, da v veliki meri uporablja urejanje z zlivanjem, ki urejanju s konfiguracijo kar tesno sledi. Urejanje z zlivanjem je zelo primerno, saj primerjava dveh objektov tipa *slow* traja precej dolgo, urejanje z zlivanjem pa naredi malo primerjav. Na naslednjem mestu je C++-ov algoritem, ki očitno ni optimiziran za uporabniške tipe. Tesno skupaj mu sledita hitro urejanje in urejanje s kopico. Urejanje z vstavljanjem se pokaže za splošno precej neučinkovito, za popolnoma neučinkovito pa se izkaže urejanje z izbiranjem, ki naredi zelo veliko primerjav, poleg tega pa ima še kvadratno časovno zahtevnost v vsakem primeru.

5 Sklep

Ugotovili smo, da je učinkovitost posameznega sortiranega algoritma odvisna od lastnosti elementov, ki ga sortiramo. To se je še posebej lepo vidi v primerjavi tipov *huge* in *slow*, ki sta namenoma zasnovana tako, da imata precej različne lastnosti. Odvisnost algoritmov od lastnosti podatkov je posledica različnega števila primerjav in zamenjav, ki jih naredijo, prav tako pa je tudi posledica njihove časovne zahtevnosti. To se je lepo pokazalo tudi v precej raznolikih konfiguracijah, ki pa so vseeno kazale grob vzorec. Pri majhnih n se je vedno izkazal za najučinkovitejšega algoritem s pričakovano kvadratno časovno zahtevnostjo, pri velikih n pa je bil vedno najprimernejši algoritem s pričakovano časovno zahtevnostjo $\mathcal{O}(n \log_2 n)$. Raznolikost dobljenih konfiguracij in različen vrstni red sortirnih algoritmov pri različnih tipih sta rezultata, zaradi katerih lahko z gotovostjo potrdimo hipotezo 1.

Kompoziten algoritem je bil v veliko primerih boljši od posameznih algoritmov. Pravzaprav je bil malo slabši le v skrajnih primerih, ko je bil sestavljen le iz enega algoritma. To se je zgodilo pri zelo kratkih poljih, ko je bil čas urejanja tako zelo majhen, da je praktično nepomembno, s katerim sortirnim algoritmom uredimo polje. Taka konfiguracija se je pojavila tudi pri tipu *string*, kjer pa je čas urejanja tako dolg, da se je iskanje po konfiguraciji izkazalo za zanemarljivo, saj razlike ni bilo moč zaznati. Tako je tudi v skrajnih primerih algoritem praktično enako dober kot algoritem, iz katerega je sestavljen. V takih primerih hipoteze 2 ne moremo potrditi, lahko pa primerjamo skupen čas urejanja posameznih algoritmov. Pri skupnem času urejanja je kompozitni algoritem vedno boljši od posameznih algoritmov in ravno skupni čas urejanja ima največjo težo, saj to uporabniku zagotavlja najučinkovitejše urejanje v povprečju. Ker se je kompozitni sortirni algoritem v tem pogledu najbolje obnesel, lahko potrdimo hipotezo 2.

6 Zaključek

Menim, da so bili doseženi vsi cilji naloge. Kompozitni sortirni algoritem, z angleškim imenom *tweaksort*, se je izkazal za odličen sortirni algoritem, ki je izjemno prilagodljiv. Prilagodi se lahko tako računalniku kot podatkom, ki jih ureja. Vsi v nalogi opisani algoritmi so implementirani v knjižnici, ki je, skupaj s primeri priložena kot priloga 1.

Knjižnica je tudi dejansko uporabna v realnih situacijah in javno dostopna na spletnem naslovu `http://gimvic.org/slak/tweaksort/` skupaj s primeri uporabe. Metoda dela se je izkazala za pravilno, eksperiment je bil izpeljan tako, da je bila podvrženost merskim napakam čim manjša. Lahko bi ga seveda izboljšali tako, da bi povečevali število iteracij in urejali tudi polja večjih dolžin. Dejansko hitrost algoritma bi lahko izboljšali z bolj optimizirano implementacijo. Poleg tega bi lahko v naš osnovni nabor sortirnih algoritmov dodali še kakšen algoritem in tako povečali raznovrstnost izbora najboljšega izmed njih. S povečevanjem števila izhodiščnih nekompozitnih algoritmov bi lahko precej izboljšali hitrost algoritma, bi pa tudi precej podaljšali čas učenja, saj bi se povečalo število kandidatov za optimalno konfiguracijo.

Neraziskana ostaja odvisnost algoritmov od podatkovne strukture, če izpustimo zahtevo po naključnem dostopu v polju. Prav tako bi lahko naprej raziskovali tudi odvisnost uspešnosti sortirnih algoritmov od predhodnega vrstnega reda podatkov v polju.

7 Viri

1. KNUTH, Donald. 1998. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley. ISBN 0-201-89685-0.

2. KOZAK, Jernej. 1986. *Podatkovne strukture in algoritmi.* Ljubljana: DMFA SRS.

3. PLESTENJAK, Bor. 2010. *Uvod v numerične metode* (v pripravi, verzija 4. marec 2010). Dostopno na spletnem naslovu: `http://www-lp.fmf.uni-lj.si/plestenjak/vaje/nafgg/Predavanja/Knjiga_NM.pdf`

4. WIRTH, Niklaus. 1985. *Računalniško programiranje, 2. del.* Ljubljana: DMFA SRS.

8 Priloge

1. Zgoščenka z knjižnicami, v katerih so implementirani vsi v nalogi opisani algoritmi.