

UNIVERZITET CRNE GORE

PRIRODNO-MATEMATIČKI FAKULTET

RAČUNARSTVO I INFORMACIONE TEHNOLOGIJE

PODGORICA

**Seminarski rad**

**SORT ALGORITAM**

Predmet: Paralelni algoritmi

|  |  |
| --- | --- |
| Profesor:  dr Igor Jovančević | Studenti:  Milica Bjelica  Krsto Kustudić, 9/24 |
| Asistent:  mr Nikola Pižurica |  |

**SADRŽAJ:**

[1. UVOD 1](#_Toc194615516)

[2. BUBBLE SORT ALGORITAM 2](#_Toc194615523)

[2.1 Implementacija algoritma 2](#_Toc194615518)

[2.2 Složenost algoritma 4](#_Toc194615519)

[2.2.1 Vremenska složenost(Time complexity) 4](#_Toc194615520)

[2.2.2 Prostorna složenost (space complexity) 5](#_Toc194615521)

[2.3 Paralelizacija algoritma 6](#_Toc194615522)

[3. SELECTION SORT ALGORITAM 9](#_Toc194615523)

[3.1 Implementacija algoritma 9](#_Toc194615524)

[3.2 Složenost algoritma 12](#_Toc194615525)

[3.2.1 Vremenska složenost (Time complexity) 12](#_Toc194615526)

[3.2.2 Prostorna složenost (Space complexity) 12](#_Toc194615527)

[3.3 Paralelizacija algoritma 13](#_Toc194615528)

[4. INSERTION SORT ALGORITAM 14](#_Toc194615529)

[4.1 Implementacija algoritma 14](#_Toc194615530)

[4.2 Složenost algoritma 17](#_Toc194615531)

[4.2.1 Vremenska složenost (Time complexity) 17](#_Toc194615532)

[4.2.2 Prostorna složenost (Space complexity) 17](#_Toc194615533)

[4.3 Paralelizacija algoritma 18](#_Toc194615534)

[5. MERGE SORT ALGORITAM 19](#_Toc194615535)

[5.1 Implementacija algoritma 19](#_Toc194615536)

[5.2 Složenost algoritma 23](#_Toc194615537)

[5.2.1 Vremenska složenost (Time Complexity) 23](#_Toc194615538)

[5.2.2 Prostorna složenost (Space Complexity) 23](#_Toc194615539)

[5.3 Paralelizacija algoritma 24](#_Toc194615540)

[6. QUICK SORT ALGORITAM 26](#_Toc194615541)

[6.1 Implementacija algoritma 26](#_Toc194615542)

[6.2 Složenost algoritma 31](#_Toc194615543)

[6.2.1 Vremenska složenost (Time Complexity) 31](#_Toc194615544)

[6.2.2 Prostorna složenost 32](#_Toc194615545)

[6.3 Paralelizacija algoritma 32](#_Toc194615546)

[7. HEAP SORT ALGORITAM 34](#_Toc194615547)

[7.1 Implementacija algoritma 35](#_Toc194615548)

[7.2 Složenost algoritma 39](#_Toc194615549)

[7.2.1 Vremenska složenost (Time Complexity) 39](#_Toc194615550)

[7.2.2 Prostorna složenost 39](#_Toc194615551)

[7.3 Paralelizacija algoritma 39](#_Toc194615552)

[8. ZAKLJUČAK 40](#_Toc194615547)

[LITERATURA 41](#_Toc194615547)

# UVOd

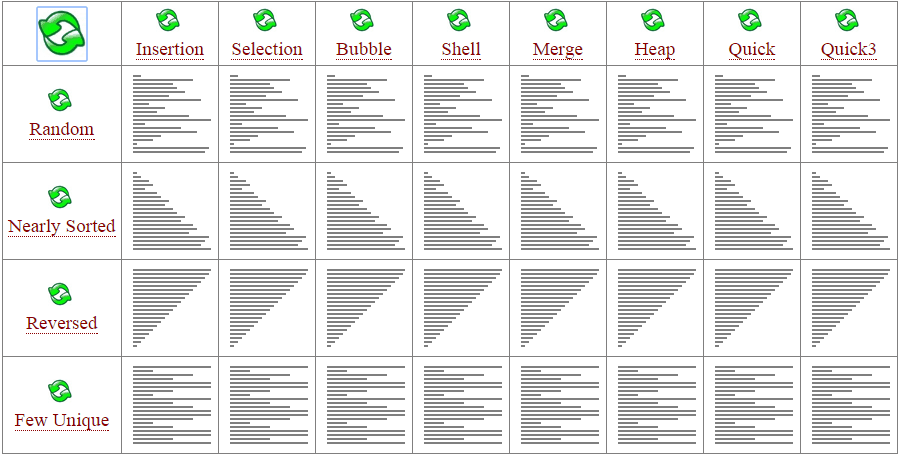
Sortiranje, poznato i kao ređanje, jedan je od najvažnijih zadataka u oblasti računarstva. U osnovi, sortiranje podrazumijeva linearno uređivanje niza elemenata po određenom kriterijumu, poput veličine brojeva, leksikografskog reda teksta, dužine nizova ili vrijednosti specifičnih polja u složenim strukturama podataka. Kao temeljna operacija, sortiranje je prisutno u brojnim primjenama koje svakodnevno koristimo, uključujući telefonske imenike, rječnike, kataloge i razne baze podataka. Uređeni podaci omogućavaju znatno bržu i jednostavniju pretragu, čime se povećava efikasnost izvršavanja različitih zadataka.

Sortirani nizovi olakšavaju izvođenje brojnih operacija, naročito pretrage, gdje binarna pretraga značajno smanjuje složenost u poređenju sa linearnom pretragom nesortiranih podataka. Ova činjenica jasno ističe važnost sortiranja za optimizaciju računarskih procesa i performansi sistema.

Postoji više različitih algoritama za sortiranje, a njihov izbor često zavisi od specifičnih potreba i zahtjeva primjene. Najjednostavniji algoritmi, poput **Bubble sort**, **Selection sort** i **Insertion sort** lako se implementiraju, ali imaju relativno slabije performanse, s vremenskom složenošću O(n²). S druge strane, složeniji algoritmi kao što su **Merge sort**, **Quick sort** i **Heap sort** omogućavaju bolje performanse, s prosječnom vremenskom složenošću O(n log n).

Prilikom analize efikasnosti algoritama za sortiranje obično se fokus stavlja na dvije osnovne operacije koje najviše utiču na performanse: poređenje elemenata i razmjenu elemenata niza. Kod nizova koji sadrže složenije tipove podataka, upravo ove operacije dominiraju ukupnom složenošću algoritma.

U ovom seminarskom radu detaljno će biti razmotren sort algoritam u kontekstu paralelnog računarstva, s posebnim osvrtom na prednosti, nedostatke i mogućnosti optimizacije kroz paralelizaciju procesa sortiranja. Kroz teorijski prikaz i praktične primjere implementacije, biće prikazano koliko paralelna implementacija može unaprijediti performanse sort algoritama, kao i pod kojim uslovima je njena primjena najefikasnija.[[1]](#footnote-1)



1. Ilustracija sort algoritama

# Bubble SORT ALGORITAM

Bubble sort (sortiranje mjehurića) jedan je od osnovnih algoritama za sortiranje niza elemenata. Princip rada ovog algoritma zasniva se na uzastopnom poređenju susjednih elemenata u nizu i njihovom zamjenjivanju ukoliko nisu u odgovarajućem redosljedu.

Tokom svake iteracije, algoritam kreće od početka niza i poredi elemente jedan za drugim. Ako su dva susjedna elementa u pogrešnom redosljedu (na primjer, lijevi element je veći od desnog pri rastućem sortiranju), algoritam ih zamjenjuje. Na ovaj način, najveći ili najmanji element (u zavisnosti od načina sortiranja) „ispliva” na kraj niza, baš kao što se mjehurić vazduha podiže na površinu vode, po čemu je ovaj algoritam i dobio ime.

Nakon prve iteracije, sigurno je da je posljednji element na pravilnoj poziciji, pa algoritam ponavlja isti postupak, ali ovaj put ignorišući posljednji element. Svakom narednom iteracijom se područje pretrage smanjuje za jedan element, sve dok čitav niz ne bude sortiran.

Bubble sort algoritam spada u jednostavnije algoritme i lako se implementira, ali njegova najveća mana je slaba efikasnost, posebno na velikim skupovima podataka, zbog vremenske složenosti O(n²). Iz tog razloga se bubble sort uglavnom koristi za edukacijske svrhe ili male skupove podataka gdje efikasnost nije presudna.

## Implementacija algoritma

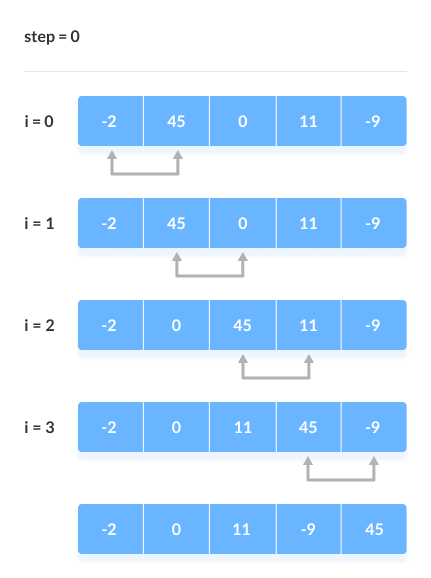
Pretpostavimo da pokušavamo sortirati niz tj. poređati elemente u rastućem poretku.

Prva iteracija (uporedi i zamijeni)

U prvoj iteraciji krećemo od samog početka niza (od prvog elementa) i vršimo sljedeće korake:

* Korak 1: Upoređujemo prvi element (na indeksu 0) sa susjednim drugim elementom (na indeksu 1). Ako je prvi element veći od drugog, to znači da nisu u pravilnom (uzlaznom) redosljedu, pa vršimo njihovu zamjenu. Ukoliko je prvi element manji ili jednak drugom, ništa ne mijenjamo i prelazimo dalje.
* Korak 2: Zatim prelazimo na drugi element (indeks 1) i upoređujemo ga sa susjednim, trećim elementom (indeks 2). Ponovo, ukoliko drugi element nije u pravilnom redosljedu u odnosu na treći (veći je od njega), mijenjaju mjesta. Ako jesu pravilno raspoređeni, nastavljamo dalje.
* Korak 3: Navedeni postupak poređenja i zamjene susjednih elemenata nastavlja se sve do pretposljednjeg elementa, koji se poredi sa posljednjim elementom u nizu.

Na kraju prve iteracije najveći element sigurno „ispliva“ na kraj niza i zauzima svoju tačnu poziciju. Svakom narednom iteracijom postupak ponavljamo, ali svaki put sa jednim elementom manje, jer posljednji elementi niza postaju sortirani.



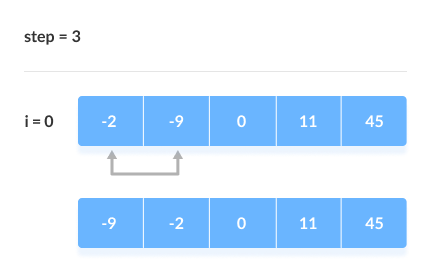
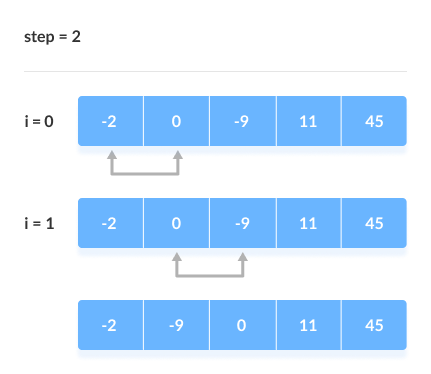
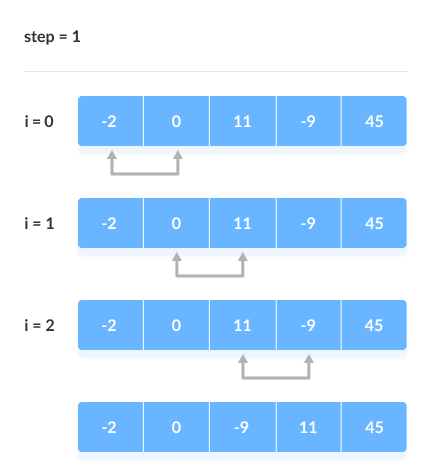
2. Ilustrovani prikaz prve iteracije

Druga iteracija

Nakon što završimo prvu iteraciju, postupak poređenja i zamjene elemenata se ponavlja za preostale, još uvijek nesortirane elemente niza.

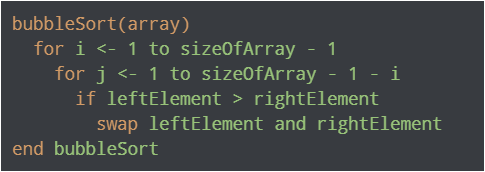
U svakoj novoj iteraciji ponovo krećemo od prvog elementa, ali sada ignorišući elemente koji su već prethodno sortirani (koji se nalaze na kraju niza). Svaka nova iteracija će na sličan način kao i prethodna dovesti najveći element iz nesortiranog dijela niza na njegovo ispravno mjesto na kraju tog dijela.

Dakle, poslije svake završene iteracije, dio niza koji je sortiran raste, a dio koji ostaje nesortiran se smanjuje za jedan element. Postupak se nastavlja sve dok svi elementi niza ne budu pravilno sortirani po željenom kriterijumu.



3. Ilustracija preostalih iteracija

Pseudo-kod na slici 4 opisuje Bubble sort algoritam. Algoritam radi na sljedeći način: prolazi kroz niz u više iteracija pomoću dvije ugnežđene petlje. Spoljašnja petlja određuje broj iteracija, a unutrašnja petlja u svakoj iteraciji poredi susjedne elemente, od prvog do posljednjeg nesortiranog elementa u nizu. Ako je lijevi element veći od desnog (pretpostavljajući uzlazno sortiranje), vrši se njihova zamjena. Nakon svake potpune iteracije spoljašnje petlje, najveći element koji nije bio sortiran dolazi na svoje pravo mjesto, odnosno „ispliva“ na kraj nesortiranog dijela niza. Ovaj proces se nastavlja sve dok se cijeli niz potpuno ne sortira.



4. Pseudo-kod za Bubble sort algoritam

U slučaju opadajućeg sortiranja (od najvećeg ka najmanjem elementu), pseudo-kod Bubble sort algoritma funkcioniše na isti način, ali sa obrnutim kriterijumom za poređenje elemenata. Ako je lijevi element manji od desnog, tada se vrši njihova zamjena mjesta. Na ovaj način, u svakoj iteraciji algoritma najmanji element „tone“ prema kraju niza. Na kraju svake iteracije najmanji element među nesortiranim elementima zauzima svoje pravo mjesto na kraju nesortiranog dijela niza. Postupak se ponavlja sve dok cijeli niz ne bude sortiran u opadajućem redosljedu.

## Složenost algoritma

### Vremenska složenost(Time complexity)

Bubble sort algoritam ima različitu složenost u zavisnosti od stanja ulaznog niza:

* **Najbolji slučaj: O(n)**

Ovaj slučaj se javlja kada je niz već sortiran, pa algoritam prođe kroz niz samo jednom bez potrebe za zamjenama, te je složenost linearna.

* **Najgori slučaj: O(n²)**

Najgori slučaj nastupa kada je niz u potpunosti obrnutog redoslijeda od željenog (na primjer, niz treba da bude uzlazno sortiran, a elementi su u silaznom redoslijedu). Tada svaki element mora „proći“ kroz cijeli niz, te su potrebna maksimalna poređenja i zamjene, što rezultira kvadratnom složenošću O(n²).

* **Prosječni slučaj: O(n²)**

Prosječni slučaj nastaje kada su elementi niza slučajno raspoređeni, odnosno nisu ni idealno sortirani, niti idealno obrnuto sortirani. U prosjeku, broj poređenja i zamjena i dalje vodi do kvadratne složenosti O(n²).

U detaljnoj analizi, broj poređenja smanjuje se za jedan element svakom iteracijom algoritma, na sljedeći način: Prvi ciklus ima (n - 1) poređenja, drugi ciklus (n - 2) poređenja, treći ciklus (n - 3) poređenja i tako dalje, sve do posljednjeg ciklusa koji ima 1 poređenje.

Ukupni broj poređenja se dobija zbirno kao:

(n-1) + (n-2) + (n-3) +.....+ 1 = n(n-1)/2

Ovaj izraz ima kvadratnu zavisnost od veličine ulaznog niza, što potvrđuje vremensku složenost O(n²).

### Prostorna složenost (space complexity)

Bubble sort algoritam ima prostornu složenost O(1), što znači da zahtijeva konstantnu količinu dodatnog prostora u memoriji, bez obzira na veličinu niza. Ovo je zbog toga što se za razmjenu elemenata koristi samo jedna dodatna privremena promjenljiva. Kod optimizovane verzije Bubble sorta (kada uvedemo dodatnu promjenljivu za provjeru je li niz već sortiran) koriste se ukupno dvije dodatne promjenljive, što opet predstavlja konstantnu složenost (O(2), što se označava kao O(1)).

Bubble sort algoritam je stabilan, što znači da elementi s istim vrijednostima (npr. dva ista broja ili elementi koji se smatraju jednakim po kriterijumu sortiranja) nakon sortiranja zadržavaju originalni redoslijed koji su imali u ulaznom nizu.

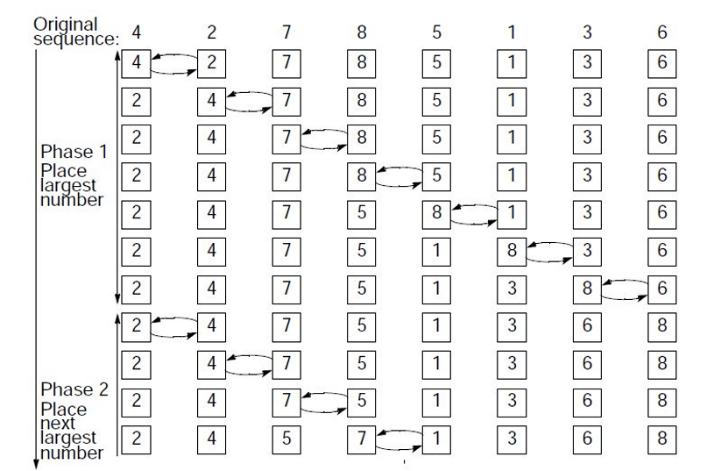
Bubble sort se koristi uglavnom u sljedećim situacijama:

* Kad složenost nije bitna (npr. mali nizovi gdje nije problem što je sporiji).
* Kad je važnija jednostavnost koda i lakoća implementacije, npr. u edukativnim situacijama gdje se uči osnovni koncept algoritama i programiranja.

U praksi se rijetko koristi na velikim skupovima podataka zbog svoje neefikasnosti (spore performanse).[[2]](#footnote-2)

## Paralelizacija algoritma

Bubble sort, u obliku kakav je napisan na slici 4, predstavlja čisto sekvencijalni algoritam. Svaki korak u unutrašnjoj petlji se izvršava prije narednog, a cijela unutrašnja petlja mora biti završena prije nego što počne sljedeća iteracija spoljašnje petlje.

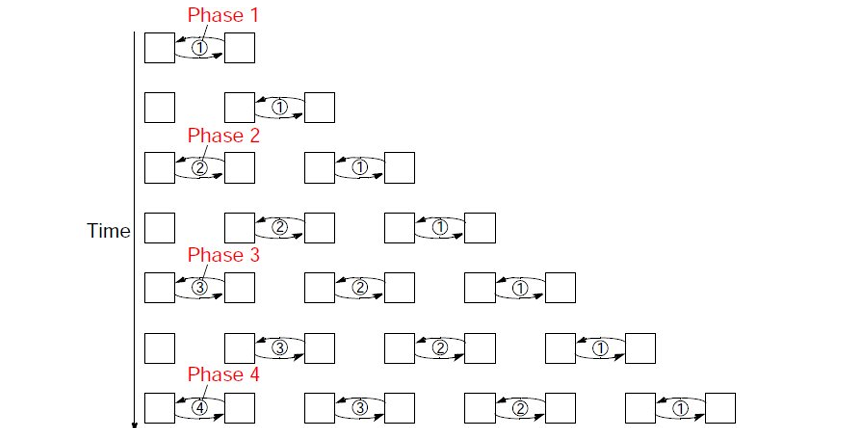


5. Koraci u Bubble sort algoritmu

Mnogi paralelne implementacije sekvencijalnih algoritama sortiranja, su sinhroni algoritmi, što znači da grupa akcija koje se izvode nad brojevima mora biti završena prije nego što započne sljedeća grupa akcija. Takav je i Bubble sort algoritam jer svaka operacija zavisi od prethodne ali to ne znači da se algoritam ne može prilagoditi paralelnom izvođenju. Ključna ideja je da se određene operacije ne moraju čekati da se prethodne potpuno završe, već se mogu izvršavati paralelno, pod uslovom da ne dolazi do konflikta između istovremenih zamjena.

Naime, zamjene u Bubble sort algoritmu (operacije poređenja i eventualne razmjene elemenata) mogu se organizovati tako da formiraju cjevasti tok izvršavanja (pipeline). U ovoj strukturi, svaki broj "putuje" kroz niz prema svojoj poziciji, ali sa vremenskim pomakom u odnosu na prethodne brojeve, analogno slično kao što vagoni voza ulaze jedan za drugim u tunel, ne sudarajući se međusobno.

Slika 6 ilustruje upravo ovu ideju. Prva zamjena (npr. pomjeranje najmanjeg broja ka početku) započinje odmah. Već u narednom vremenskom koraku, druga zamjena može da krene, ali ne odmah iza prve, već sa malim pomakom, kako ne bi došlo do „preklapanja“ sa prethodnom operacijom. Taj pomak se obezbjeđuje tako što je nova operacija pomjerena za jedno mjesto udesno, odnosno razdvojena jednim procesom. Na isti način, treća zamjena može da počne nakon druge, i tako redom. Na taj način se formira lanac istovremenih operacija koje teku nezavisno, ali sinhronizovano omogućavajući višestruke pomake brojeva kroz niz bez međusobnog ometanja.



6. Preklapanje koraka sortiranja u cjevovodu

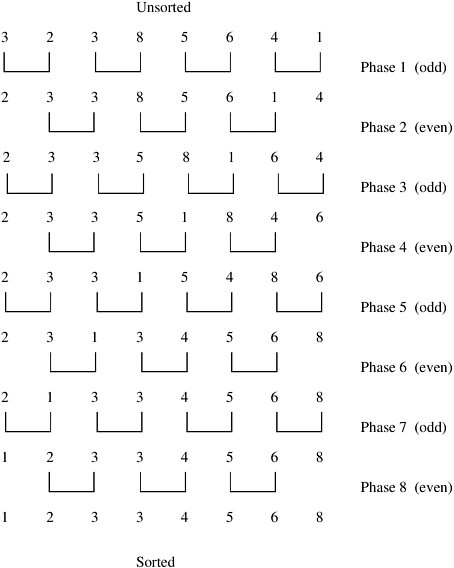
Ovaj pristup značajno poboljšava efikasnost izvođenja bubble sort algoritma u paralelnim sistemima, jer omogućava preklapanje koraka sortiranja i bolju iskorišćenost dostupnih procesora.

Na ovoj ideji zasniva se posebna verzija Bubble sort algoritma koja se naziva **NEPARNO-PARNO SORTIRANJE (ODD-EVEN TRANSPOSITION SORT).**

Algoritam neparno-parnog sortiranja (odd-even transposition sort) sortira niz od n elemenata kroz ukupno n faza. Algoritam se zasniva na naizmjeničnom izvršavanju dvije vrste faza: neparne i parne.

Tokom neparne faze, porede se elementi sa neparnim indeksima sa svojim desnim susjedima, i ako su u pogrešnom redoslijedu, vrši se njihova zamjena.

Tokom parne faze, porede se elementi sa parnim indeksima sa desnim susjedima, uz identičan princip zamjene ako nisu pravilno raspoređeni.



7. Ilustracija algoritma neparno-parnog sortiranja (odd-even transposition sort)

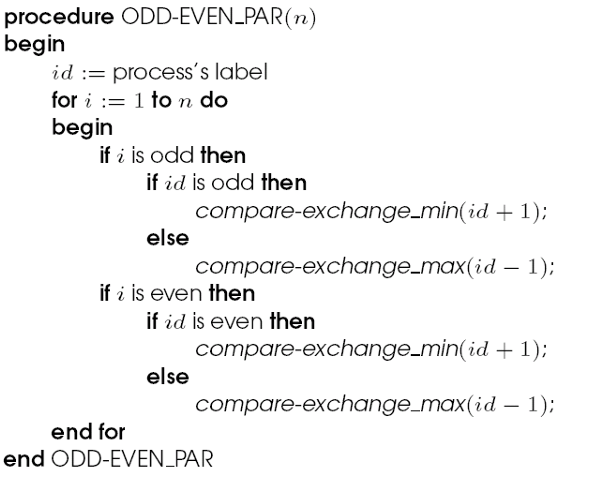
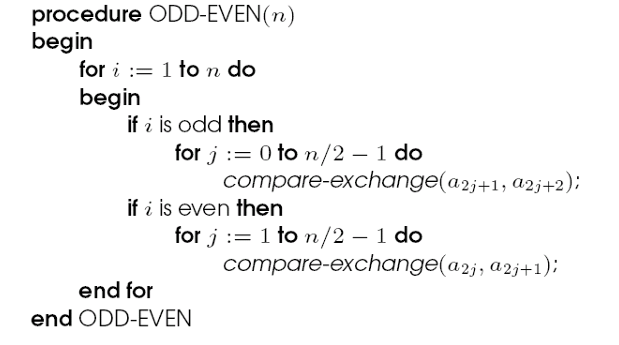
Nakon što se izvrši svih n faza, niz će biti potpuno sortiran. Budući da svaka faza zahtijeva O(n) operacija poređenja i zamjene, a ukupan broj faza je n, ukupna sekvencijalna vremenska složenost algoritma iznosi O(n²). Odd-even transposition sort nema posebnu prednost nad običnim bubble sort-om kada se koristi u sekvencijalnom programiranju, i zato se u tom kontekstu rijetko spominje.[[3]](#footnote-3)

Međutim paralelna implementacija algoritma neparno-parnog sortiranja (odd-even transposition sort) ima manju vremensku složenost. Tokom svake faze algoritma, operacije poređenja i zamjene na parovima elemenata mogu se izvoditi istovremeno, što omogućava efikasnu raspodjelu posla među procesima.

Razmotrimo slučaj gdje se svaki element nalazi u zasebnom procesu, odnosno gdje je broj procesa jednak broju elemenata n koje treba sortirati. Pretpostavimo da su procesi organizovani kao jednodimenzionalni niz, i da se element aᵢ nalazi na procesu Pᵢ, za i = 1, 2, ..., n.

U toku neparne faze, svaki proces sa neparnim indeksom (P₁, P₃, P₅, ...) poredi i, ako je potrebno, zamjenjuje svoj element sa elementom desnog susjeda. Tokom parne faze, isti postupak sprovode procesi sa parnim indeksom (P₂, P₄, P₆, ...), opet sa svojim desnim susjedima.

U svakoj fazi, ili parni ili neparni procesi istovremeno izvršavaju korak poređenja i zamjene sa svojim susjedima. Ovakva razmjena može se izvršiti u vremenu O(1) (konstantno vrijeme). Pošto se ukupno izvodi n faza, ukupno paralelno vrijeme izvršavanja ovog algoritma je O(n).



8. Pseudo kod za sekvencijalnu(lijevo) i paralelnu(desno) implementaciju algoritama neparno-parnog sortiranja(odd-even transposition sort)

U sekvencijalnoj verziji jedan proces izvršava čitav algoritam, prolazeći kroz sve faze n puta, i u svakoj fazi iterativno poredi i eventualno zamjenjuje parove elemenata. Operacije se obavljaju redom, unutar for petlji, bez paralelizma.

S druge strane, paralelna verzija pretpostavlja da svaki element obrađuje poseban proces. Svaki proces ima svoj identifikator (id) i u svakoj od n faza samostalno odlučuje s kojim susjedom (lijevim ili desnim) će obaviti razmjenu, u zavisnosti od toga da li je faza i njegov id parni ili neparni. Sve operacije poređenja i zamjene u jednoj fazi se izvršavaju istovremeno, što omogućava ubrzanje putem paralelizacije.[[4]](#footnote-4)

Prostorna složenost sekvencijalnog i paralelnog odd-even transposition algoritma prikazanih u pseudokodovima je linearna, odnosno O(n). Oba algoritma zahtijevaju ulazni niz veličine n elemenata na kojem vrše sortiranje, ali ne koriste dodatne pomoćne strukture koje značajno rastu s veličinom ulaza. Potrebne dodatne varijable, kao što su indeksi petlji ili oznake procesa, zahtijevaju konstantan dodatni prostor, što znači da ne povećavaju ukupnu složenost. Dakle, ukupno gledano, oba algoritma su memorijski efikasna jer koriste minimalnu dodatnu memoriju, te njihova ukupna prostorna složenost ostaje O (n).

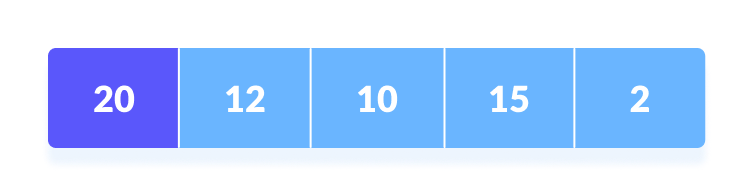
# SELECTION SORT ALGORITAM

Selection sort ili sortiranje odabirom je algoritam sortiranja koji radi tako što u svakoj iteraciji pretražuje nesortirani dio niza kako bi pronašao najmanji (u slučaju uzlaznog sortiranja tj. sortiranja u rastućem poretku) ili najveći (u slučaju silaznog sortiranja tj. sortiranja u opadajućem poretku) element. Kada pronađe taj element, algoritam ga zamjenjuje sa prvim elementom nesortiranog dijela niza. Time, nakon svake završene iteracije, niz je podijeljen na dva dijela – lijevi dio koji je sortiran, i desni dio koji je nesortiran. Sa svakom novom iteracijom, sortiran dio raste za jedan element, a nesortiran dio se smanjuje.

Ovaj postupak se ponavlja sve dok niz u potpunosti ne bude sortiran. Glavna ideja algoritma ogleda se upravo u „odabiru“ najmanjeg ili najvećeg elementa preostalog dijela niza i njegovom smještanju na početak tog dijela. Selection sort je poznat po jednostavnoj implementaciji, ali mu je vremenska složenost O(n²), što ga čini neefikasnim za sortiranje većih nizova. Zbog toga se uglavnom koristi u obrazovne svrhe ili kod manjih skupova podataka gdje efikasnost nije od presudnog značaja.

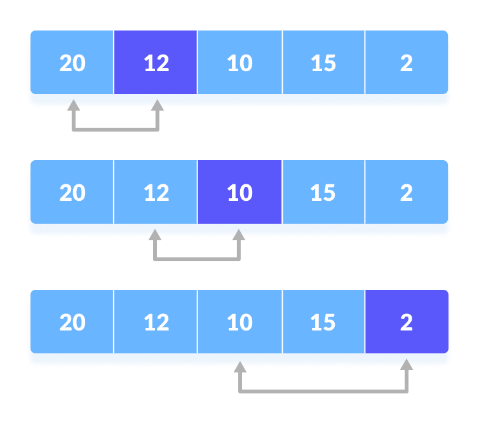
## Implementacija algoritma

Na početku svake iteracije, algoritam uzima prvi element nesortiranog dijela niza i postavlja ga kao trenutni minimum. U prvoj iteraciji, to je prvi element cijelog niza, dok u narednim iteracijama indeks ovog početnog elementa napreduje ka desno, kako se dio sortiranog niza širi.



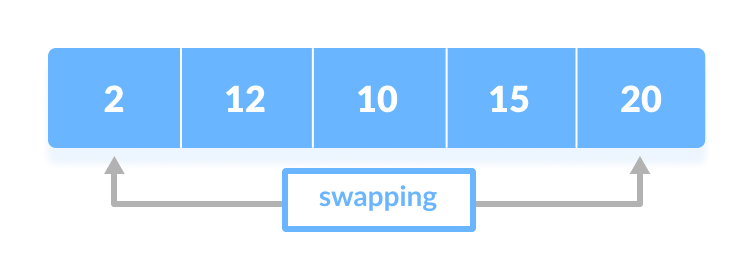
9.Postavljanje početnog elementa niza kao trenoutni minimum

Nakon odabira početnog minimuma, algoritam poredi ovaj element sa svim ostalim elementima u nesortiranom dijelu niza. Ako se prilikom poređenja naiđe na element koji je manji od trenutnog minimuma, onda se pronađeni element označava kao novi minimum. Algoritam zatim nastavlja poređenja, prelazeći redom preko svakog narednog elementa sve do posljednjeg elementa u nesortiranom dijelu niza. Na kraju ovog koraka, pronađen je pravi minimalni element nesortiranog dijela niza.



10. Traženje stvarnog minimuma

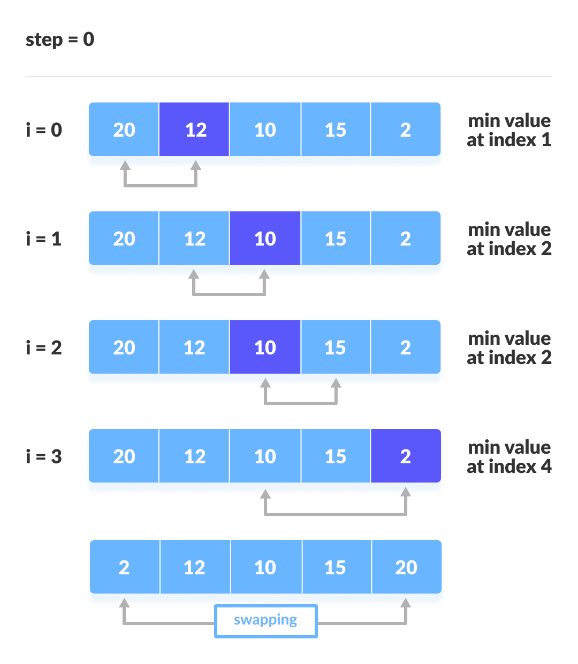
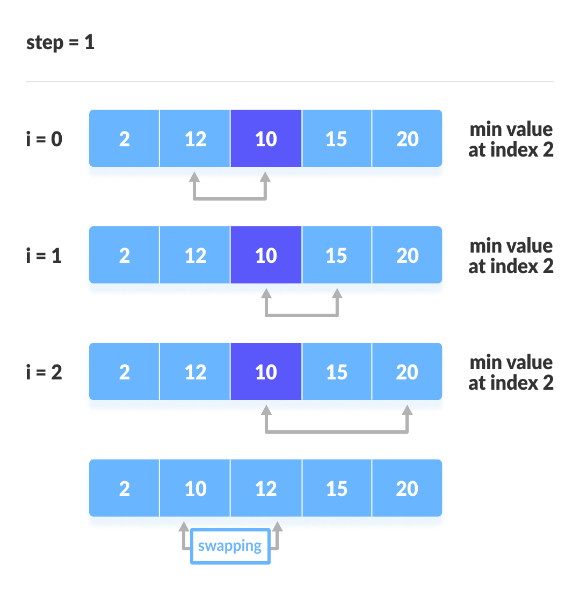
Kad se završava poređenje svih elemenata u nesortiranom dijelu, algoritam je pronašao element sa najmanjom vrijednošću. Taj minimalni element zatim mijenja mjesta sa prvim elementom nesortiranog dijela niza (ako već nije na tom mjestu). Ovom zamjenom minimalni element zauzima svoju tačnu poziciju na početku nesortiranog dijela niza. Nakon ovog koraka, niz je podijeljen na dva dijela: sortirani dio (lijevo) i nesortirani dio (desno).



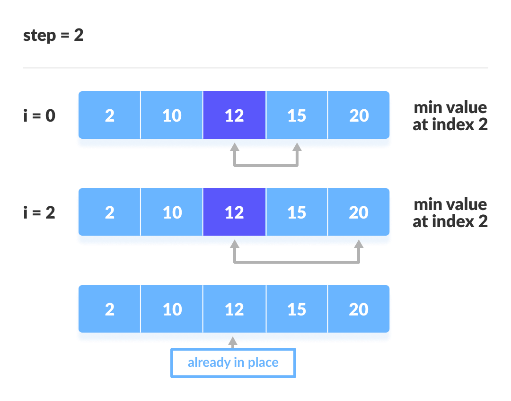
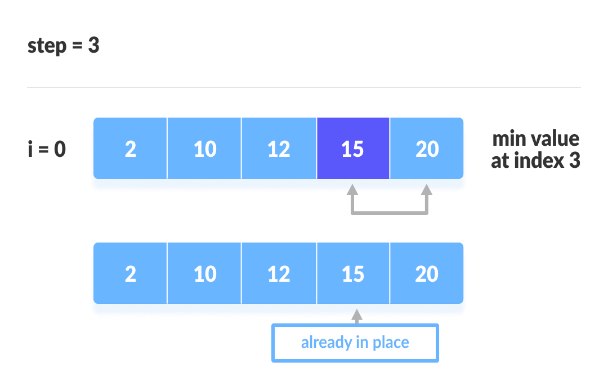
11.Minimum se postavlja na početak nesortiranog dijela niza

U sljedećoj iteraciji algoritam pomjera početnu poziciju (indeks) za jedan korak udesno, počevši od prvog elementa preostalog nesortiranog dijela niza, te ponovo prolazi kroz prethodna tri koraka: ponovo odabira prvi element nesortiranog dijela kao privremeni minimum, traži stvarni minimum u ostatku niza i postavlja pronađeni minimalni element na početak nesortiranog dijela.

Iteracije se ponavljaju sve dok algoritam ne prođe kroz cijeli niz. Na kraju rada algoritma, svi elementi se nalaze na svojim odgovarajućim pozicijama, a niz je u potpunosti sortiran u rastućem redosljedu. Ovakvim pristupom Selection sort algoritam postepeno „gradi“ sortiran niz od najmanjih ka najvećim elementima, uvijek pronalazeći i izdvajajući najmanji preostali element u svakom koraku.

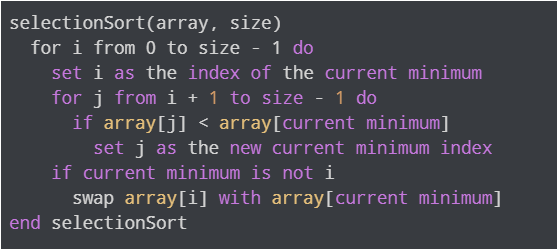
 

12. Prva i druga iteracija

13. Treća i četvrta iteracija

Na slici 14prikazan je pseudo-kod Selection sort algoritma koji prima ulazni niz (*array*) i njegovu veličinu (*size*). Algoritam prolazi kroz niz u petlji (*for i*) od indeksa 0 do size - 1. Na početku svake iteracije spoljašnje petlje, indeks prvog nesortiranog elementa (*trenutno i*) postavlja se kao početni minimum. Zatim, pomoću unutrašnje petlje (*for j*), algoritam kreće od sljedećeg elementa (*indeks i + 1*) i poredi ga sa trenutnim minimumom (*array[current minimum]*). Ako tokom prolaska kroz niz pronađe element koji je manji od trenutnog minimuma, taj novi element označava kao novi minimum (*ažurira indeks minimuma na j*). Kada se unutrašnja petlja završi, provjerava se da li se pronađeni minimalni element nalazi na indeksu različitom od početne pozicije (*i*). Ako je tako, algoritam zamjenjuje mjesta trenutnog minimuma sa elementom na početku nesortiranog dijela (*array[i]*). Ukoliko je minimum već na ispravnom mjestu, nema potrebe za zamjenom. Ovaj proces se ponavlja sve dok algoritam ne prođe kroz sve elemente niza, čime se osigurava da svaki element dođe na svoje ispravno mjesto, a niz na kraju ostaje sortiran.



14. Pseudo-kod za Selection sort algoritam

Ako želimo da Selection sort algoritam sortira niz u opadajućem poretku (od najvećeg ka najmanjem), onda se radi veoma sličan postupak kao za rastući poredak. Jedina razlika je to da se u svakoj iteraciji traži najveći element u nesortiranom dijelu niza umjesto najmanji kao što je slučaj kod sortiranja u rastućem poredku.

## Složenost algoritma

### Vremenska složenost (Time complexity)

Vremenska složenost Selection sort algoritma u svim slučajevima (najboljem, najgorem i prosječnom) jeste O(n²). U svakom slučaju, Selection sort mora da prođe kroz cijeli niz, tražeći minimum u svakom prolazu kroz nesortirani dio. To znači da se uvijek mora obaviti isti broj poređenja, bez obzira na stanje ulaznog niza (bilo da je niz već sortiran, potpuno nesortiran ili djelimično sortiran).

Broj poređenja za svaku iteraciju izgleda ovako: U prvom prolazu algoritam izvrši (n - 1) poređenja, u drugom prolazu (n - 2) poređenja, u trećem prolazu (n - 3) poređenja i tako dalje, sve dok posljednji prolaz ne završi s jednim poređenjem.

Kad zbrojimo sva poređenja dobijemo sljedeći izraz:

(n - 1) + (n - 2) + (n - 3) + ..... + 1 = n(n - 1) / 2

Ovaj izraz je približno jednak n2/2 , što pokazuje kvadratnu složenost. U notaciji velikog O, konstante zanemarujemo, te ostaje O(n²).

### Prostorna složenost (Space complexity)

Selection sort ima prostornu složenost O(1), što znači da algoritam koristi minimalnu količinu dodatne memorije za svoj rad. Konkretno, koristi se samo jedna dodatna varijabla koja služi za pamćenje indeksa trenutno najmanjeg elementa u nesortiranom dijelu niza. Bez obzira na veličinu ulaznog niza, količina dodatne memorije koju koristi Selection sort ostaje konstantna (uvijek ista).

Algoritam Selection sort nije stabilan. To znači da ukoliko postoje elementi jednake vrijednosti, njihov redosljed u konačnom sortiranom nizu ne mora ostati isti kao u originalnom nizu. Drugim riječima, Selection sort može promijeniti relativni redosljed istovrijednih elemenata.

Selection sort algoritam se najčešće koristi u situacijama gdje:

* Treba sortirati male skupove podataka:

Zbog jednostavnosti implementacije, Selection sort je pogodan za male nizove ili popise, gdje njegova kvadratna složenost O(n²) ne predstavlja problem u smislu efikasnosti.

* Trošak zamjene elemenata nije bitan:

Selection sort vrši relativno mali broj zamjena elemenata (tačno jednu zamjenu po prolazu kroz niz). Ako je zamjena elemenata skupa operacija, Selection sort može biti bolji izbor od algoritama poput Bubble sorta, koji imaju više zamjena elemenata.

* Provjera svih elemenata je obavezna:

Ukoliko je potrebno pregledati svaki element niza, Selection sort uvijek prolazi kroz sve elemente, što osigurava detaljnu provjeru.

* Troškovi pisanja u memoriju su značajni (kao u flash memoriji):

Selection sort algoritam radi mali broj pisanja u memoriju (tačno O(n)), jer u svakoj iteraciji izvrši maksimalno jednu zamjenu. To ga čini pogodnim za korišćenje u uređajima koji imaju ograničen broj ciklusa pisanja u memoriju (poput flash memorije), u poređenju sa Bubble sort algoritmom koji pravi veći broj zamjena i ima O(n²) pisanja u memoriju.[[5]](#footnote-5)

## Paralelizacija algoritma

Iako Selection sort ima jednostavnu strukturu, njegova paralelizacija je ograničena zbog sekvencijalne zavisnosti među iteracijama. Naime, svaka nova iteracija počinje tek kada se prethodna završi, jer se minimum uvijek traži u preostalom, nesortiranom dijelu niza.

Jedini dio algoritma koji se može djelimično paralelizovati jeste traženje minimuma. Ova operacija se može ubrzati korišćenjem paralelne redukcije, gdje svaki proces traži lokalni minimum u svom segmentu, a zatim se lokalni minimumi upoređuju da bi se pronašao globalni minimum. Ipak, ostatak algoritma ostaje sekvencijalan, jer se sortiranje nastavlja tek kada se prethodni minimum postavi na svoje mjesto.

Zbog ovih zavisnosti i potrebe za čestom sinhronizacijom, paralelna verzija Selection sorta nije efikasna i nije troškovno optimalna. S obzirom da se koristi n procesa, ukupni trošak algoritma (vrijeme × broj procesa) iznosi O(n²) — što je identično kao i kod sekvencijalne verzije, pa se troškovna optimalnost ne postiže.

Zbog navedenih ograničenja, Selection sort se **rijetko koristi u paralelnim sistemima**, a značajno efikasniji izbor predstavljaju algoritmi poput **parallel merge sorta** i **odd-even transposition sorta**, koji omogućavaju veći stepen istovremene obrade i bolju skalabilnost.

# INSERTION SORT ALGORITAM

Insertion sort, ili sortiranje umetanjem, jednostavan je algoritam za sortiranje niza elemenata koji radi tako što jedan po jedan uzima elemente iz nesortiranog dijela niza i postavlja ih na njihovo pravo mjesto unutar već sortiranog dijela niza.

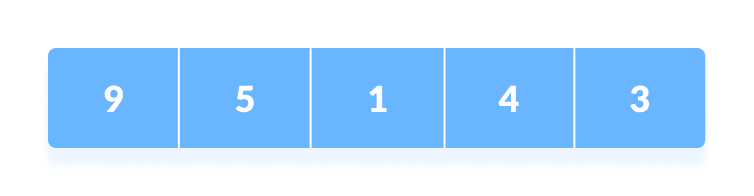
U početku, algoritam pretpostavlja da je prvi element niza već sortiran. Nakon toga, uzima se sljedeći element iz nesortiranog dijela niza i poredi se redom s elementima u sortiranom dijelu, s desna na lijevo. Ako je ovaj odabrani element manji (u slučaju uzlaznog sortiranja) od elementa u sortiranom dijelu, elementi se pomjeraju za jedno mjesto udesno kako bi napravili mjesta za ubacivanje novog elementa na njegovo odgovarajuće mjesto. Proces se nastavlja sve dok se ne pronađe prava pozicija za novi element. Ovaj postupak se ponavlja za sve preostale elemente u nesortiranom dijelu niza, sve dok čitav niz ne postane sortiran.

Princip rada ovog algoritma je veoma sličan načinu na koji ljudi ručno sortiraju karte tokom kartaške igre. Prva uzeta karta predstavlja osnov za slaganje ostalih karata. Ako je sledeća uzeta karta veća od karte u ruci, stavlja se s desne strane, inače s lijeve strane. Na isti način, ostale nerazvrstane karte se uzimaju i stavljaju na svoje mjesto.

Dakle, sortiranje umetanjem korak po korak gradi sortirani niz, uvijek osiguravajući da je dio niza koji je obrađen uredno sortiran. Prednosti ovog algoritma su jednostavna implementacija i efikasnost za manje ili djelimično sortirane nizove. Međutim, njegova vremenska složenost u najgorem slučaju je O(n²), što ga čini manje efikasnim za velike nizove.

## Implementacija algoritma

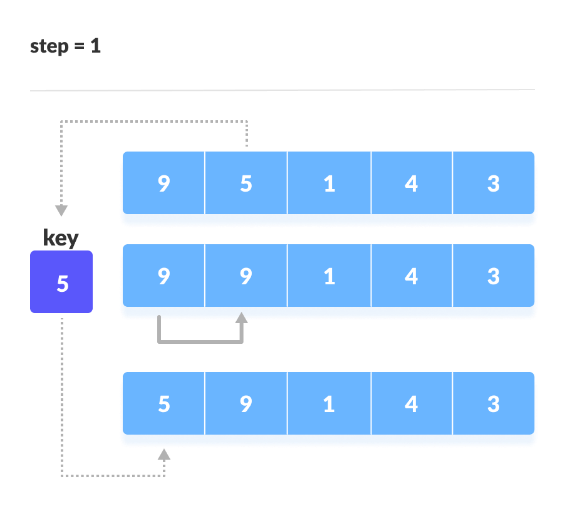
Pretpostavimo da trebamo sortirati sljedeći niz.



15.Početni niz koji treba sortirati

Prvi korak

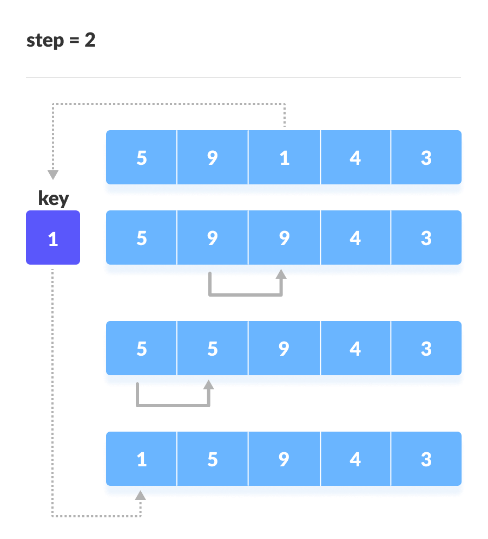
Pretpostavljamo da je prvi element (9) već sortiran. Sada prelazimo na drugi element (5) i privremeno ga izdvajamo kao ključ (key). Zatim uspoređujemo ovaj ključ (5) s prvim (sortiranim) elementom (9). Budući da je ključ (5) manji od prvog elementa (9), pomjeramo 9 jedno mjesto udesno i stavljamo ključ (5) na prvu poziciju u nizu. Nakon završetka ovog koraka niz je [5, 9, 1, 4, 3]. Dakle, ako je prvi element veći od ključa, ključ se postavlja ispred prvog elementa.



16. Ilustracija prvog koraka

Drugi korak

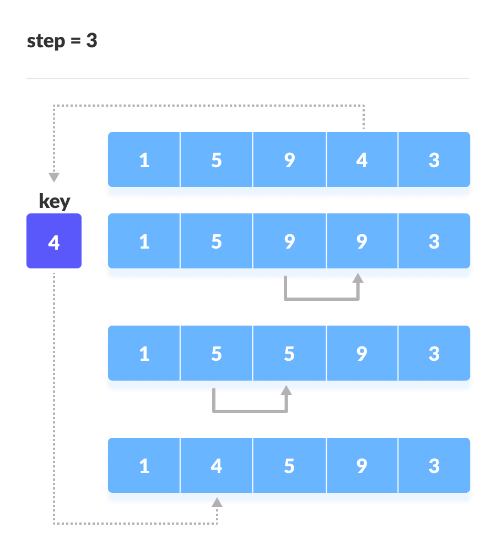
Sad imamo dva elementa ([5, 9]) koji su već sortirani. Sljedeći nesortirani element (treći po redu) je broj 1 koji opet izdvajamo kao ključ (key). Poredeći ključ (1) s elementima u sortiranom dijelu niza (9, zatim 5), uočavamo da je broj 1 manji od oba elementa. Pomjeramo oba ova elementa (9 i 5) za jedno mjesto udesno kako bismo napravili mjesta za naš ključ (1). Postavljamo ključ (1) na početak niza jer je najmanji. Nakon završetka ovog koraka niz je [1, 5, 9, 4, 3].



17. Ilustracija drugog koraka

Treći korak

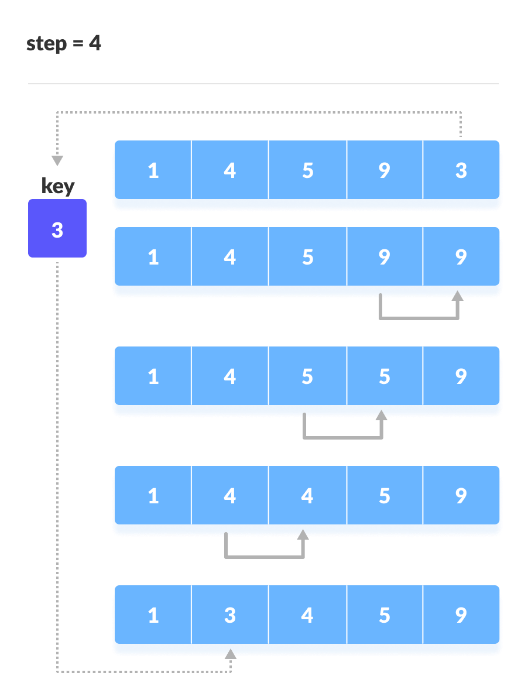
U ovom trenutku prva tri elementa ([1, 5, 9]) su već sortirana. Sljedeći nesortirani element (četvrti po redu) je broj 4 koji uzimamo kao ključ. Ključ (4) uspoređujemo s prethodno sortiranim elementima (9, 5, 1), krećući od posljednjeg sortiranog elementa prema početku. Element 9 je veći od ključa 4 pa ga pomjeramo jedno mjesto udesno. Zatim uspoređujemo ključ 4 s elementom 5 koji je također veći od ključa, pa i njega pomjeramo jedno mjesto udesno. Element 1 je manji od ključa 4, pa ne pravimo dalja pomjeranja. Ključ (4) sada stavljamo na poziciju odmah iza elementa 1. Nakon završetka ovog koraka niz je [1, 4, 5, 9, 3].



18. Ilustracija trećeg koraka

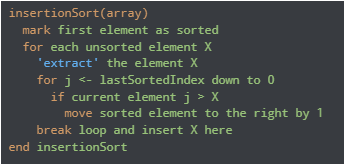
Na isti način nastavljamo iteracije za svaki naredni nesortirani element (npr. broj 3), poredeći ga redom s elementima iz sortiranog dijela i umećući ga na pravo mjesto, dok cijeli niz ne bude potpuno sortiran.

Ovim postupkom, Insertion sort postepeno proširuje sortirani dio niza sve dok cijeli niz ne bude pravilno uređen.



19.Ilustracija četvrtog koraka

Na slici 20 je prikazan pseudo-kod algoritma Insertion sort. Prvo se označava prvi element u nizu (*array*) kao već sortiran. Nakon toga, za svaki sljedeći nesortirani element (označen kao *X*) iz niza izvršava se proces „umetanja“. Element X se uzima i poredi s elementima koji su već sortirani. Ovaj proces poređenja i pomjeranja se vrši pomoću petlje koja kreće od indeksa posljednjeg sortiranog elementa (*lastSortedIndex*) prema indeksu 0. Ako je trenutni element na poziciji j veći od elementa X, trenutni element (*array[j]*) se pomjera za jedno mjesto udesno. Kada algoritam naiđe na element koji nije veći od X, prekida petlju, te se element X postavlja (umeće) na trenutno slobodnu poziciju. Postupak se ponavlja za sve preostale nesortirane elemente sve dok cijeli niz ne postane sortiran.



20.Pseudo-kod za Insertion sort algoritam

## Složenost algoritma

### Vremenska složenost (Time complexity)

Insertion sort ima različite vremenske složenosti u zavisnosti od redoslijeda elemenata u nizu:

* **Najgori slučaj: O(n²)**

Najgori slučaj se događa kada elementi niza imaju obrnut redoslijed u odnosu na željeni (na primjer, niz je silazno sortiran, a mi ga želimo uzlazno sortirati). U ovom slučaju, svaki element iz nesortiranog dijela mora se porediti sa svim elementima već sortiranog dijela. U prvoj iteraciji, pravi se maksimalno 1 poređenje, u drugoj iteraciji maksimalno 2 poređenja, u trećoj maksimalno 3 poređenja ..., u posljednjoj iteraciji maksimalno (n-1) poređenja. Kada saberemo broj poređenja kroz sve iteracije, dobijamo izraz:

​ 1+2+3+⋯+(n−1)=n(n−1)​/2

Ovaj izraz je približno jednak n2/2, što znači da je vremenska složenost u najgorem slučaju O(n²).

* **Najbolji slučaj: O(n)**

Najbolji slučaj se događa kada je niz već sortiran. U ovom slučaju, svaki sljedeći element koji algoritam provjerava je već na svom odgovarajućem mjestu. Zbog toga se, u najboljem slučaju, unutrašnja petlja (koja poredi elemente) nikada ne izvršava, već algoritam prolazi kroz niz samo jednom. Samim tim, složenost je linearna, odnosno O(n).

* **Prosječni slučaj: O(n²)**

Prosječni slučaj nastaje kad su elementi nasumično raspoređeni. U praksi ovo je najčešći scenarij. U ovom slučaju, broj usporedbi je u prosjeku sličan kao i u najgorem slučaju, što rezultira prosječnom složenošću od O(n²).

### Prostorna složenost (Space complexity)

Insertion sort ima prostornu složenost O(1), što znači da koristi konstantnu količinu dodatnog prostora. Konkretno, ovaj algoritam koristi samo jednu dodatnu varijablu (*key*) za privremeno čuvanje trenutnog elementa kojeg treba umetnuti na odgovarajuće mjesto.

Insertion sort je stabilan algoritam, što znači da elementi s istim vrijednostima nakon sortiranja ostaju u istom redoslijedu kao što su bili u originalnom nizu. Stabilnost je važna kod sortiranja složenijih struktura podataka, gdje je redoslijed jednako vrijednih elemenata bitan.

Insertion sort je jednostavan algoritam, efikasan za male ili već djelimično sortirane nizove. Međutim, zbog vremenske složenosti O(n²) u prosječnom i najgorem slučaju, nije pogodan za sortiranje velikih nizova.[[6]](#footnote-6)

## Paralelizacija algoritma

Iako je Insertion sort jednostavan i efikasan za male ili djelimično sortirane nizove, njegova paralelizacija predstavlja značajan izazov zbog sekvencijalne zavisnosti između koraka algoritma. U osnovi, Insertion sort funkcioniše tako što za svaki novi element iz nesortiranog dijela niza traži odgovarajuću poziciju u već sortiranom dijelu, pomjerajući elemente udesno dok ne pronađe pravo mjesto. Ovaj proces zahtijeva da sortirani dio niza bude već ispravno formiran prije nego što se obradi sljedeći element. Zbog toga, svaka iteracija algoritma zavisi od rezultata prethodne, što ga čini sekvencijalnim po svojoj prirodi. Na primjer, treći element se ne može pravilno smjestiti dok prethodna dva nisu već sortirana. Ova zavisnost onemogućava klasičnu paralelizaciju u kojoj bi više elemenata moglo simultano da „traže“ svoje mjesto, jer bi to moglo dovesti do narušavanja reda i neispravnog sortiranja.

Ipak, određeni pokušaji ograničene paralelizacije postoje, i najčešće su teorijskog ili obrazovnog karaktera. Jedna od varijanti jeste pipelining pristup, gdje se različite faze umetanja izvode u kaskadi kroz više procesa. U tom modelu, svaki proces „prima“ element, obrađuje ga djelimično i prosleđuje dalje. Međutim, ovakav pristup ne ostvaruje stvarni paralelizam u smislu vremenskog ubrzanja, već više simulira sekvencijalni tok kroz niz procesa. Drugi pristup pokušava da podijeli niz na više manjih podnizova, sortira ih paralelno pomoću insertion sorta, i zatim izvrši spajanje (merge). Ipak, ovakav pristup praktično uvodi potpuno novi algoritam, koji više nije klasičan Insertion sort, već hibrid sa metodama nalik merge sortu.

Zbog svega navedenog, može se zaključiti da je Insertion sort izuzetno teško paralelizovati na efikasan način, jer su njegove osnovne operacije međuzavisne. Čak i kada bi se dio algoritma paralelizovao, dobici u performansama bi bili minimalni, dok bi složenost implementacije i komunikacije među procesima porasla.

U modelu u kojem se koristi jedan proces po procesoru, svaki proces bi imao ograničenu i nesamostalnu ulogu, jer ne bi mogao nezavisno da izvrši umetanje svog elementa bez znanja o stanju prethodnih elemenata u nizu.

Dakle, zbog svoje sekvencijalne strukture i zavisnosti među koracima, Insertion sort se ne smatra pogodnim za paralelnu obradu. Njegova primjena u paralelnim algoritmima je rijetka i više eksperimentalna nego praktična

# Merge sort algoritam

Merge sort je efikasan algoritam za sortiranje koji se zasniva na strategiji *Podijeli i osvoji* (Divide and Conquer). Njegova osnovna ideja je da se veliki problem sortiranja podijeli na više manjih i jednostavnijih podproblema, koji se zatim rješavaju nezavisno, nakon čega se njihova rješenja kombinuju u jedno konačno i potpuno rješenje. Algoritam se sastoji iz tri glavna koraka: podjela (divide), rješavanje (conquer) i spajanje (combine).

U fazi podjele, niz koji se sortira rekurzivno se dijeli na dvije približno jednake polovine. Ukoliko niz sadrži više od jednog elementa, izračunava se srednji indeks q između lijevog i desnog kraja niza, i niz se razdvaja na dva dijela – lijevu i desnu polovinu. Ovaj proces se ponavlja sve dok svi dobijeni podnizovi ne sadrže samo po jedan element. Takvi nizovi se smatraju trivijalno sortiranim, jer jedan element ne može biti u pogrešnom redosledu.

Kada su svi nizovi svedeni na pojedinačne elemente, započinje faza rješavanja, tj. spajanja podnizova. U ovoj fazi, dva po dva podniza koja su već sortirana kombinuju se u veće sortirane cjeline. Tokom spajanja se porede početni elementi oba podniza, i manji od njih se ubacuje u novi niz. Proces se nastavlja sve dok se svi elementi ne iskoriste, pri čemu se preostali elementi jednog od podnizova jednostavno dodaju na kraj. Spajanje se ponavlja sve dok se svi nivoi rekurzije ne završe i ne dobije se konačan, sortirani niz.

Merge Sort je stabilan algoritam, što znači da čuva originalni redosljed elemenata sa jednakom vrijednošću. Međutim, pošto koristi dodatne pomoćne nizove tokom procesa spajanja, nije in-place algoritam. Njegova vremenska složenost je O(n log n) u svim slučajevima (najbolji, prosječan i najgori), što ga čini pogodnim za obradu velikih količina podataka.

## Implementacija algoritma

Pretpostavimo da trebamo da sortiramo sljedeći niz:



21. Početni niz koji treba sortirati

Prvi korak – Divide (podijeli)

Prva operacija algoritma je podjela niza na dvije jednake polovine:

Lijevi podniz: (6, 5, 12) i desni podniz: (10, 9, 1).

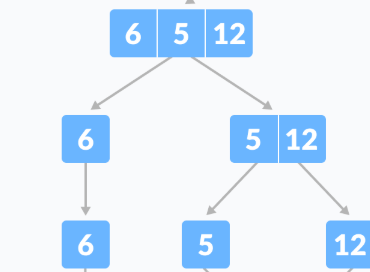


22. Ilustrovani prikaz niza nakon prvog koraka

Drugi korak

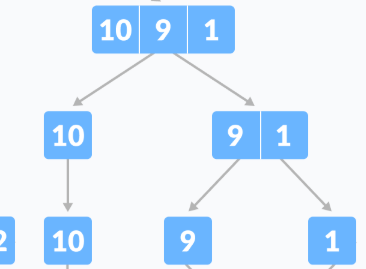
U ovom koraku se lijevi podniz rekurzivno dijeli sve dok se ne dobije sortirani niz dužine jedan. Postupak je sljedeći. Niz (6, 5, 12) se dijeli na 2 podniza: (6) i (5, 12). Zatim (5, 12) ide u još jednu podjelu: (5) i (12).

Sada imamo nizove: (6), (5) i (12).  
Svaki sadrži jedan element, što znači da su to bazni slučajevi algoritma(sortirani po definiciji).



23. Ilustrovani prikaz lijevog podniza nakon koraka broj 2

Isti postupak se ponavlja i za desni podniz (10, 9, 1). Prvo imamo podnizove (10) i (9, 1). Zatim se rekurzivno postupak ponavlja za niz (9, 1), te na kraju ovoga koraka imamo 3 niza sa po jednim elementom: (10), (9), i (1).



24. Ilustrovani prikaz desnog podniza nakon koraka broj 2

Treći korak – spajanje

Sada počinje faza kombinovanja, gdje se sortirani nizovi sa po jednim elementom spajaju u veće sortirane podnizove.

* Lijeva grana spajanja:

1. Prvo se spajaju (5) i (12):
   1. 5 je manji od 12 → rezultat: (5, 12)
2. Zatim se (6) i (5, 12) spajaju:
   1. 6 i 5 → 5 ide prvi
   2. 6 i 12 → 6 ide drugi
   3. ostaje 12 → ide treći  
       → rezultat: (**5, 6, 12)**

Niz (5, 6, 12) predstavlja konačno sortirani rezultat lijeve polovine početnog niza.

* Desna grana spajanja:

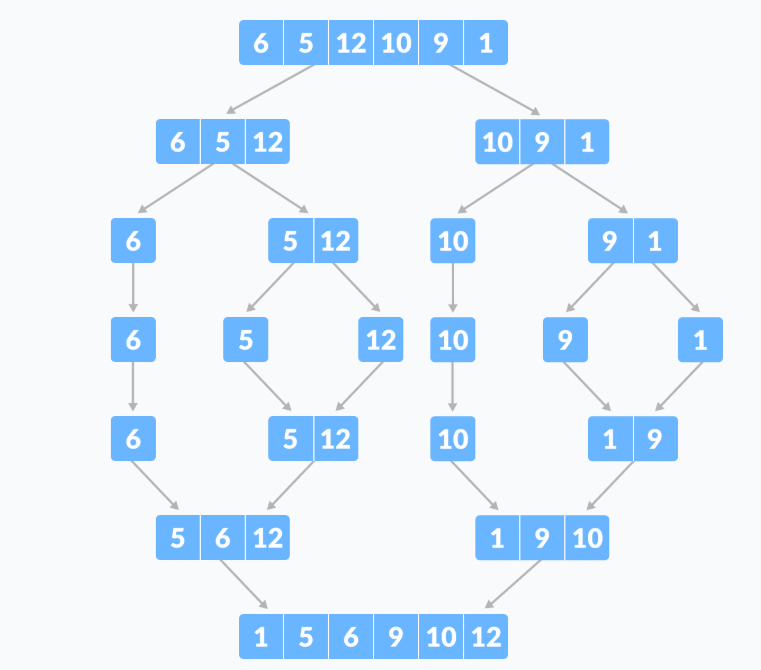
1. Prvo se spajaju (9) i (1):
   1. 1 je manji od 9 → rezultat: (1, 9)
2. Zatim se (10) i (1, 9) spajaju:
   1. 10 i 1 → 1 ide prvi
   2. 10 i 9 → 9 ide drugi
   3. ostaje 10 → ide treći  
       → rezultat: (**1, 9, 10)**

Ovo je konačno sortirani rezultat desne polovine početnog niza.

Na kraju se ove dvije sortirane polovine spajaju u konačan, sortiran niz. Spajanje ide ovako:

1. 5 i 1 → 1 ide prvi
2. 5 i 9 → 5 ide drugi
3. 6 i 9 → 6 ide treći
4. 12 i 9 → 9 ide četvrti
5. 12 i 10 → 10 ide peti
6. ostaje 12 → ide šesti

Konačni rezultat sortiranja je niz: (1, 5, 6, 9, 10, 12).

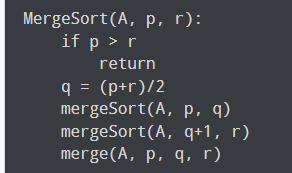


25. Ilustracija kompletnog merge sort algoritma

Prikazana funkcija MergeSort(A, p, r) na slici 26 predstavlja rekurzivnu implementaciju Merge sort algoritma. Ona prima tri parametra: niz A koji treba sortirati, te indekse p i r koji označavaju početak i kraj dijela niza koji se sortira. Prvi korak u funkciji je provjera da li je lijeva granica veća od desne (if p > r). Ovo predstavlja bazni slučaj rekurzije – kada je podniz prazan ili nepravilno definisan, funkcija se ne izvršava dalje i odmah se vraća. U praksi, bazni slučaj se često postavlja kao p >= r, što obuhvata i situaciju kada podniz ima tačno jedan element, jer se takav niz smatra već sortiranim.

Ako bazni slučaj nije zadovoljen, funkcija računa sredinu podniza kao q = (p + r) / 2. Ova vrijednost predstavlja indeks koji dijeli trenutni podniz na dvije polovine: lijevu od p do q i desnu od q + 1 do r. Sljedeći koraci su dva rekurzivna poziva: mergeSort(A, p, q) za sortiranje lijeve polovine i mergeSort(A, q + 1, r) za sortiranje desne polovine. Svaki od ovih poziva ponaša se na isti način – dijeli podniz na još manje dijelove dok se ne dođe do nizova sa jednim elementom.

Kada su oba podniza sortirana, funkcija merge(A, p, q, r) se poziva kako bi se ta dva dijela spojila u jedan veći sortirani podniz. Funkcija merge ima ključnu ulogu u algoritmu – uzima dva sortirana podniza (A[p..q] i A[q+1..r]) i kombinuje ih u jedan novi niz A[p..r], zadržavajući uzlazni redosljed elemenata. Na taj način se nizovi postepeno "grade" nazad, od najmanjih elemenata ka cijelini, sve dok se ne dobije potpuno sortiran niz. Cjelokupni proces kombinuje efikasnost podjele sa preciznim i kontrolisanim spajanjem, što omogućava Merge sort algoritmu da postigne stabilnu i optimalnu vremensku složenost od O(n log n) bez obzira na početni raspored elemenata.



26. Implementacija Merge sort algoritma

## Složenost algoritma

### Vremenska složenost (Time Complexity)

Merge sort algoritam ima konzistentnu vremensku složenost O(n log n) u najboljem, najgorem i prosječnom slučaju. Ova osobina čini Merge sort izuzetno predvidivim i efikasnim algoritmom za sortiranje, posebno kod velikih skupova podataka.

* **Najbolji slučaj – O(n log n)**

U najboljem slučaju, niz je već djelimično ili potpuno sortiran. Ipak, Merge sort neće to iskoristiti kao što to rade neki drugi algoritmi (npr. Quick Sort), jer uvijek vrši rekurzivnu podjelu niza i spajanje, bez obzira na raspored elemenata.  
Dakle, čak i u najboljem slučaju, vrijeme izvršavanja ostaje O(n log n), jer se niz dijeli log₂n puta i svaki nivo podjele zahvata ukupno n elemenata tokom spajanja.

* **Prosječan slučaj – O(n log n)**

U prosječnom slučaju, raspored elemenata u nizu je nasumičan. Merge sort i dalje rekurzivno dijeli niz i vrši spajanje na isti način, što rezultira istim brojem operacija kao i u najboljem slučaju. Zato i prosječna složenost ostaje O(n log n).

* **Najgori slučaj – O(n log n)**

U najgorem slučaju, niz je obrnutog redosleda (potpuno nesortiran). Ipak, kao i u ostalim slučajevima, Merge sort i dalje funkcioniše identično: vrši jednaku količinu rekurzivnih podjela i spajanja. Zbog toga čak i u najgorem mogućem rasporedu, broj operacija se ne povećava značajno i složenost ostaje O(n log n).

### Prostorna složenost (Space Complexity)

Prostorna složenost Merge sort algoritma iznosi O(n). To znači da Merge sort zahtijeva dodatni memorijski prostor proporcionalan veličini ulaznog niza.

Tokom faze spajanja (merge), Merge sort koristi dodatne pomoćne nizove da bi smjestio sortirane vrijednosti iz lijevog i desnog podniza. Zatim se ti elementi prepisuju nazad u glavni niz. Pošto u najgorem slučaju može biti potrebno čuvanje n/2 elemenata iz lijevog i n/2 iz desnog podniza, ukupna dodatna memorija koja se koristi je O(n).

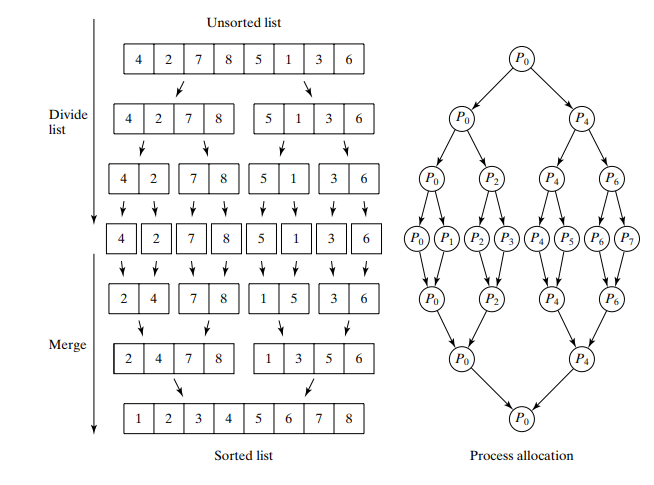
Zbog toga Merge sort nije in-place algoritam (ne sortira sve unutar samog niza, već koristi dodatni prostor za rad).

Dakle, Merge sort je stabilan, predvidiv i efikasan algoritam, ali za razliku od nekih drugih, žrtvuje memoriju da bi obezbijedio dosljedne performanse. [[7]](#footnote-7)

## Paralelizacija algoritma

Paralelna implementacija algoritma Merge sort prirodno proizilazi iz njegove specifične „podijeli-pa-vladaj” (divide-and-conquer) strukture. Prilikom paralelizacije, prvobitna lista se sukcesivno dijeli na manje dijelove, koji se potom distribuiraju na više dostupnih procesora – slika 27.1. Proces dijeljenja počinje tako što glavni procesor (na primjer, procesor P0) šalje polovinu podataka drugom procesoru (P4). U sljedećem koraku, procesor P0 šalje četvrtinu početne liste procesoru P2, a procesor P4 šalje svoju četvrtinu liste procesoru P6. Ovaj postupak dijeljenja nastavlja se sve dok svi dostupni procesori ne dobiju svoje manje dijelove liste koje treba sortirati.

Kada se podaci rasporede na procesore, svaki od njih nezavisno vrši sortiranje primljenih podlista. Nakon što svaki procesor zasebno završi lokalno sortiranje, slijedi faza spajanja (engl. merge). U ovoj fazi obrađene podliste se sukcesivno kombinuju u veće sortirane cjeline. Proces spajanja odvija se od nižih nivoa stabla ka višim nivoima, pri čemu procesori na nižim nivoima šalju svoje već sortirane podliste procesorima na višim nivoima. Postupak se nastavlja sve dok se sve podliste ponovo ne sjedine u jednu potpuno sortiranu listu, koju na kraju posjeduje glavni procesor (P0).



27.1 - primjer alg. pomocu alociranja stabla procesora

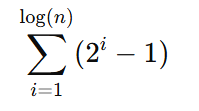
Važno je istaći da se komunikacija među procesorima tokom faza dijeljenja i spajanja odvija simetrično, ali u suprotnim smjerovima. U fazi dijeljenja, komunikacija se odvija od glavnog procesora prema nižim nivoima stabla, dok u fazi spajanja komunikacija ide obrnuto, od nižih nivoa ka glavnom procesoru. Ukupno komunikaciono vrijeme ovakvog paralelnog pristupa može se približno izraziti formulom:



27.2 - formula za računanje vremena izvršavanja alg.

gdje je 𝑝 broj procesora, *startup* vrijeme potrebno za inicijalizaciju komunikacije, a *data* vrijeme potrebno za prenos pojedinačnog podatka između procesora.

Što se tiče računanja, ono se odvija isključivo tokom faze spajanja podlista. U toj fazi svaki korak zahtijeva određeni broj operacija poređenja i prebacivanja elemenata. U najgorem slučaju, za paralelno spajanje svih podlista veličine 𝑠 u svakom pojedinačnom koraku spajanja, potrebno je *2s−1* operacija. Imajući u vidu da ukupno postoji log(n) koraka spajanja, ukupno računanje može se izraziti sljedećim zbirnim izrazom:

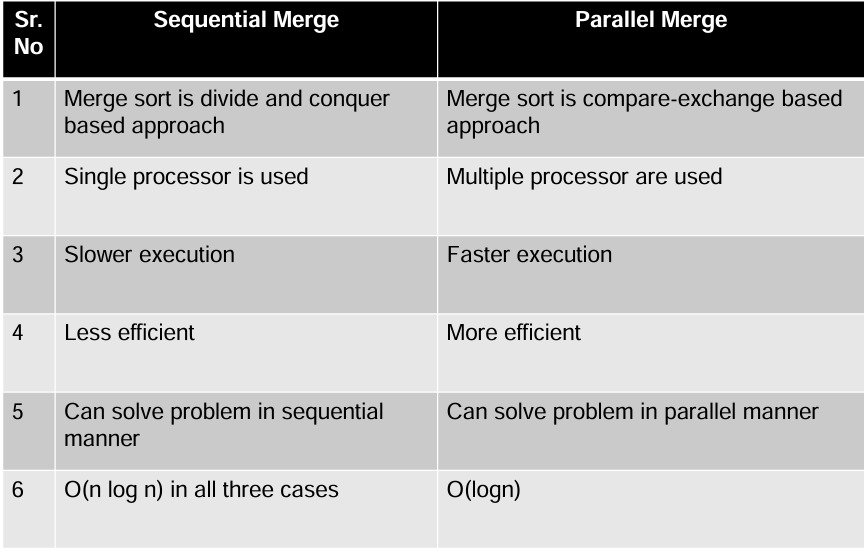


27.3 - formula za ukupno računanje vremena izvršavanja

Ovaj izraz, kada se sumira, rezultira ukupnim brojem koraka proporcionalnim veličini liste 𝑛, što daje konačnu vremensku složenost 𝑂(𝑛). Drugim riječima, kada zanemarimo komunikaciono vrijeme, računarska složenost paralelne implementacije Merge sort-a linearno zavisi od broja elemenata koje je potrebno sortirati.

Iako paralelizacija omogućava efikasnije iskorišćavanje procesorskih resursa i znatno ubrzava proces sortiranja, jedan od njenih nedostataka je neujednačeno opterećenje procesora tokom izvođenja algoritma. Na početku i pri kraju algoritma, pojedini procesori mogu biti manje angažovani, što dovodi do izvjesnog gubitka efikasnosti i neravnomjerne iskorišćenosti računarskih resursa.

Na slici 27.4 prikazane su prednosti i mane sekvencijalnog i paralelnog pristupa Merge sort algoritmu, respektivno.[[8]](#footnote-8)



27.4 - sekvencijalni i paralelni Merge sort

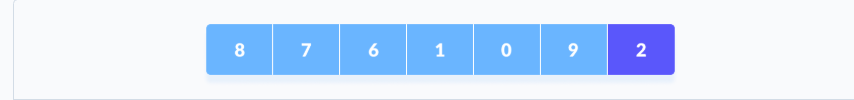
# Quick sort algoritam

Quick sort je jedan od najbržih i najefikasnijih algoritama za sortiranje, koji se temelji na paradigmi *"Podijeli i osvoji"* (Divide and Conquer). Suština algoritma je u tome da se niz dijeli na manje podnizove pomoću specijalno odabranog elementa koji se naziva pivot. Pivot je ključni element jer određuje granicu između elemenata koji su manji od njega i onih koji su veći. Tokom procesa podjele, pivot se pozicionira tako da su svi elementi manji od pivota smješteni lijevo, a svi elementi veći od pivota desno u odnosu na njegovu poziciju. Nakon toga se isti postupak rekurzivno primjenjuje na lijevu i desnu polovinu niza – svaki od tih podnizova se ponovo dijeli oko svog pivota, sve dok svi podnizovi ne budu svedeni na nizove sa jednim elementom. U tom trenutku se svi elementi nalaze na tačnim pozicijama, i niz se smatra sortiranom strukturom. Iako se elementi tehnički ne „spajaju“ kao kod Merge Sorta, krajnji rezultat predstavlja potpuno sortirani niz. Quick Sort je poznat po svojoj izuzetnoj brzini u praksi, jer sortira *in-place* (bez dodatnog prostora) i ima prosječnu vremensku složenost O(n log n), što ga čini idealnim za velike podatke i primjene gdje je memorijska efikasnost važna.

## Implementacija algoritma

Prvi korak

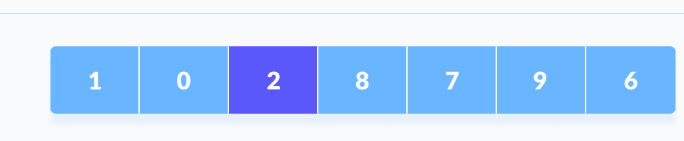
Prvi korak u radu Quick Sort algoritma je odabir pivot elementa, koji predstavlja centralnu tačku oko koje će se niz reorganizovati. U konkretnoj implementaciji prikazanoj na slici 28, kao pivot se bira posljednji (desni) element niza, što je česta i jednostavna praksa. Za dati niz [8, 7, 6, 1, 0, 9, 2], pivot je broj 2. Uloga pivota je ključna jer se svi elementi u nizu porede s njim i niz se reorganizuje tako da svi elementi manji od pivota budu sa lijeve strane, a svi veći ili jednaki sa desne strane. Iako se u ovom koraku pivot još uvijek ne pomjera na konačnu poziciju, on služi kao orijentir za dalji raspored elemenata. Kroz naredne korake algoritma, pivot će biti pravilno postavljen između podnizova manjih i većih elemenata, čime će se podjela niza izvršiti na pravilan način i omogućiti dalji rekurzivni rad algoritma.



28. Odabir pivot elementa

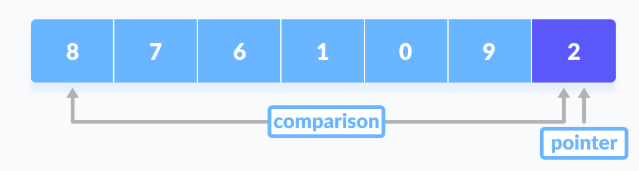
Drugi korak

Nakon što se u Quick sort algoritmu odabere pivot element – što je u ovom slučaju posljednji element niza, broj 2 – prelazi se na ključnu fazu algoritma: reorganizaciju elemenata niza u odnosu na pivot (slika 24).



29. Reorganizacija elemenata u odnosu na pivot

Cilj ove faze je da svi elementi manji od pivota budu premješteni na lijevu stranu, a svi veći ili jednaki na desnu, čime se pivot pravilno pozicionira za dalji rekurzivni rad. Proces započinje tako što se jedan pokazivač (pointer) fiksira na pivot, dok se drugi koristi za poređenje elemenata od početka niza (slika 30).



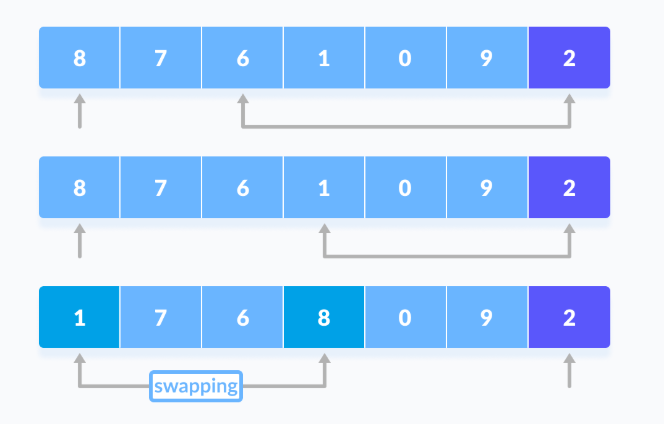
30. Poređenje elemenata sa pivotom

U trenutku kada algoritam naiđe na element veći od pivota, zabilježi se pozicija tog elementa kao potencijalna za zamjenu (slika 31).



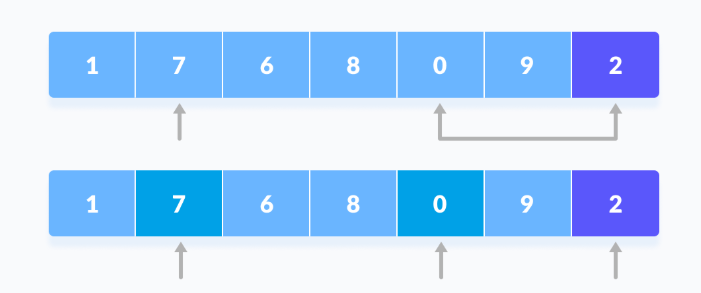
31. Označavanje pozicije elementa

Kada se kasnije naiđe na element koji je manji od pivot elementa, vrši se zamjena s prethodno označenim većim, što se može vidjeti na primjeru zamjene brojeva 1 i 8, čime 1 prelazi lijevo, a 8 se pomjera desno (slika 32).



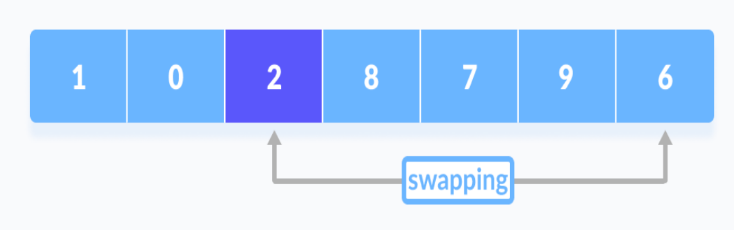
32. Proces poređenja i zamjene elemenata

Ovaj postupak se ponavlja – svaki novi manji element se zamjenjuje sa narednim većim elementom pronađenim desno, kao što je slučaj sa brojem 0, koji zamjenjuje broj 7 (slika 33).



33. Primjer zamjene dva elementa

Proces reorganizacije traje sve dok se ne obradi pretposljednji element niza. Kada je poređenje završeno, pivot element (2) se zamjenjuje sa prvim većim elementom iz desnog segmenta, čime zauzima svoju konačnu poziciju u sredini, između manjih i većih elemenata (slika 34). Na taj način, niz je podijeljen na dva dijela – lijevi dio koji sadrži elemente manje od pivota i desni dio koji sadrži elemente veće ili jednake – čime je postavljena osnova za naredne rekurzivne pozive koji se izvode na ta dva podniza.



34. Zamjena mjesta pivot elementu

Treći korak – podjela podnizova

Nakon što je pivot pravilno pozicioniran i niz podijeljen na dva dijela, Quick sort algoritam nastavlja rad tako što rekurzivno primjenjuje isti proces na podnizove s lijeve i desne strane pivota. Ova faza je prikazana na *slici 35*, gdje se vidi kako se pivot element ponovo bira — prvo za lijevi, a zatim za desni podniz. Svaki od tih podnizova se dalje dijeli sve dok se ne svede na podnizove koji sadrže samo jedan element. Naime, kada podniz sadrži jedan jedini element, on se smatra sortiranim po definiciji, jer više nema elemenata s kojima bi mogao biti u pogrešnom redosledu.



35. Biranje pivot elementa za desni i lijevi podniz

Na *slici 36*prikazan je proces rekurzivnog sortiranja lijevog podniza [8, 7, 6, 1, 0, 9, 2]. Nakon prvobitnog particionisanja, niz postaje [1, 0, 2, 8, 7, 9, 6], gdje se zatim lijevi segment [1, 0] dalje razdvaja i sortira sve dok se ne svede na [0, 1]. To su već sortirani elementi koji zauzimaju svoje tačne pozicije. Time je lijevi segment u potpunosti sortiran.



36. Rekurzivno sortiranje lijevog podniza

Na *slici 37* ilustruje se paralelni proces za desni podniz [8, 7, 9, 6], koji nakon više rekurzivnih poziva i pivot rotacija postupno prelazi u [6, 7, 8, 9]. U svakom koraku se bira novi pivot, a preostali elementi se reorganizuju na osnovu njega, sve dok svaki manji podniz ne postane trivijalno sortiran.

 37. Rekurzivno sortiranje desnog poniza

Na kraju, kada su i lijevi i desni segment potpuno sortirani kroz niz rekurzivnih podjela i reorganizacija, svi elementi se već nalaze na svojim tačnim pozicijama u nizu. Bitno je naglasiti da Quick sort ne vrši eksplicitno spajanje kao Merge sort, već se rezultat dobija kroz pozicioniranje pivota i rekurzivnu reorganizaciju. Na osnovu prikazanih ilustracija, može se jasno pratiti kako niz [8 7 6 1 0 9 2] postepeno postaje sortirani niz [0 1 2 6 7 8 9], čime je proces sortiranju završen.

Prikazani kod na slici 38 predstavlja implementaciju Quick sort algoritma pomoću dvije funkcije – quickSort i partition. Funkcija quickSort koristi rekurziju kako bi sortirala niz između indeksa leftmostIndex i rightmostIndex. Ako je lijeva granica manja od desne, algoritam poziva funkciju partition koja pronalazi tačnu poziciju pivot elementa, odnosno poziciju na koju on pripada u sortiranom nizu. Nakon toga, Quick Sort se rekurzivno poziva za lijevi i desni dio niza. Sama funkcija partition bira posljednji element kao pivot, i koristi pomoćnu varijablu storeIndex kako bi organizovala sve elemente manje od pivota na lijevu stranu. Tokom prolaska kroz niz, svaki element manji od pivota zamjenjuje se s elementom na trenutnoj storeIndex poziciji, a zatim se pivot postavlja odmah iza posljednjeg manjeg elementa. Funkcija na kraju vraća novu poziciju pivota, što omogućava dalju rekurzivnu podjelu niza. Na ovaj način, niz se efikasno sortira bez potrebe za dodatnim memorijskim prostorom, što čini Quick sort jednim od najbržih i najčešće korišćenih algoritama za sortiranje.



38. Implementacija Quick sort algoritma

## Složenost algoritma

### Vremenska složenost (Time Complexity)

Vremenska složenost Quick Sort algoritma zavisi od toga kako se bira pivot element prilikom svakog rekurzivnog poziva. Ona se izražava u tri osnovna slučaja: najbolji, prosječni i najgori.

* **Najbolji slučaj – O(n log n)**

Quick Sort ostvaruje najbolju efikasnost kada se pivot bira tako da ravnomjerno dijeli niz na dva jednako velika podniza. To se dešava kada je pivot tačno srednji element ili blizu sredine niza. U tom slučaju, niz se dijeli na polovinu sa svakim rekurzivnim pozivom, a ukupno se izvrši log n nivoa rekurzije. Na svakom nivou, potrebno je O(n) operacija za reorganizaciju elemenata, pa je ukupna složenost O(n log n).

* **Prosječan slučaj – O(n log n)**

Kada se pivot bira nasumično ili kada ne dijeli niz ravnomjerno, ali ni ekstremno loše, govorimo o prosječnom slučaju. Statistički gledano, ako se pivot rasporedi “razumno ravnomjerno” kroz više poziva, Quick sort i dalje pravi logaritamski broj nivoa i vrši linearan broj operacija po nivou. Zato i u prosjeku Quick Sort ima složenost O(n log n), što ga čini vrlo efikasnim za većinu ulaza.

* **Najgori slučaj – O(n²)**

Najgori slučaj nastaje kada se pivot stalno bira kao najmanji ili najveći element, što dovodi do toga da se jedan podniz sastoji od n - 1 elemenata, a drugi bude prazan. U tom slučaju se rekurzija izvršava n puta, a svaki nivo vrši n operacija, što rezultira kvadratnom složenošću O(n²). Ovo se može dogoditi, na primjer, ako je ulazni niz već sortiran, a pivot se bira uvijek kao krajnji element.

### Prostorna složenost

Quick sort je poznat po tome što je in-place algoritam, što znači da ne koristi dodatni prostor za nizove kao što to čini Merge sort. Međutim, koristi se memorija na pozivnom steku (call stack) zbog rekurzivnih poziva.

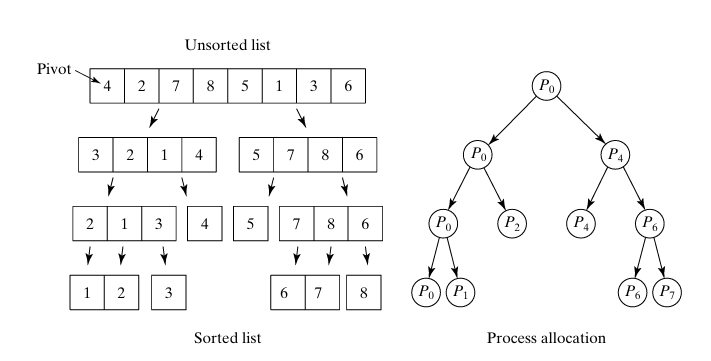
U najboljem i prosječnom slučaju, stek se koristi za oko log n nivoa rekurzije, pa je prostorna složenost O(log n).

U najgorem slučaju, ako se ne koristi optimizacija (npr. tail recursion ili randomizacija pivota), broj nivoa može biti i do n, što bi značilo složenost O(n).

U praksi, ovaj algoritam je memorijski veoma efikasan jer koristi minimalan dodatni prostor osim privremenih zamjena i pokazivača.[[9]](#footnote-9)

## Paralelizacija algoritma

Quick sort je prirodno rekurzivan algoritam, što znači da se njegovo izvođenje lako može predstaviti kao binarno stablo – pri svakom rekurzivnom pozivu lista se dijeli u dvije podliste, lijevu i desnu, na osnovu izabranog pivota. Upravo ova struktura dijeljenja pruža potencijal za paralelizaciju algoritma.

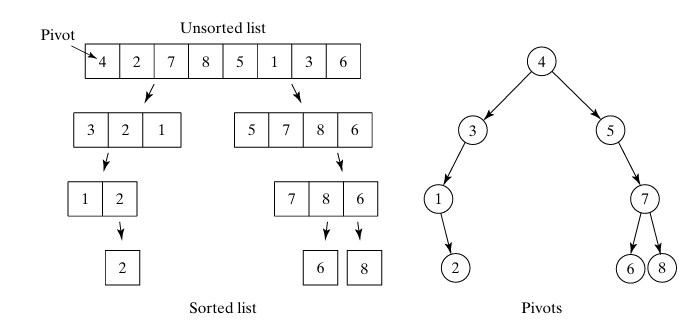


Slika 381 - Quick sort - stablo alociranja procesa

Jedan od očiglednih načina da se Quic ksort algoritam paralelizuje jeste da jedan procesor započne sortiranje, a zatim jedan od rekurzivnih poziva – najčešće desnu polovinu – delegira drugom procesoru, dok prvi procesor nastavlja sa lijevom polovinom. Ovaj pristup dovodi do formiranja stabla poziva, vrlo sličnog strukturi koju nalazimo kod Merge sort algoritma. Na slici 38.1 jasno je prikazan tok sortiranja kroz grananje, gdje se pivot element nosi uz lijevu granu sve dok ne bude konačno smješten. Ključna osobina pivot elementa jeste da, jednom kada bude pozicioniran između dvije podliste, njegova pozicija postaje konačna – više se ne uzima u obzir u narednim fazama sortiranja. U tom smislu, pivot se može smatrati "završenim poslom".

Zbog toga se u alternativnom prikazu, kao što je prikazano na slici 38.2, stablo može redizajnirati tako da pivot izgleda kao "zadržan", i da se konačno sortiranje posmatra kao in-order obilazak tog stabla: prvo lijeva podlista, zatim pivot, pa desna podlista.

Ipak, uprkos potencijalu za visoku paralelizaciju kroz rekurzivne grane, osnovni problem ostaje u tome što inicijalnu podjelu mora izvršiti samo jedan procesor. Bez obzira na broj dostupnih jezgara, pivot mora biti izabran i niz podijeljen prije nego što ostali procesori mogu preuzeti zadatke. Ovo stvara usko grlo (bottleneck) u performansama, jer početna sekvencijalnost ograničava ukupnu brzinu algoritma. Čak i ako je pivot idealno izabran i dijeli niz na dvije podjednake polovine, paralelizacija može početi tek nakon te prve, nezaobilazno sekvencijalne operacije. Zbog toga se Quic ksort, iako efikasan, suočava s inherentnim ograničenjima kada je riječ o potpunom iskorišćavanju paralelizacije.



Slika 38.2 - Quick sort – alociranje pivot elementa

Kada je riječ o vremenskoj složenosti, paralelizovani Quick sort zadržava istu teorijsku vremensku složenost kao i sekvencijalna verzija – u prosjeku *O(nlog⁡n)*, dok u najgorem slučaju (kada je pivot loše odabran i uvijek dijeli niz na jako nejednake dijelove) može dostići *O(n2)*. Međutim, uz dobar izbor pivota (npr. median-of-three pristup), taj najgori slučaj se značajno rijetko dešava.

U paralelnoj varijanti, koristi se paralelizacija rekurzivnih poziva, pa se uz *p* procesora ukupno vrijeme izvršavanja smanjuje, te se može postići teoretski speedup blizu *O(log⁡n)*, uz ukupnu količinu posla ostajući *O(nlog⁡n)*. Ipak, zbog sekvencijalne prirode prve podjele (kao što je već objašnjeno), kao i zbog troškova upravljanja nitima (synchronization, memory contention), ovaj speedup nije linearan i ima svoj praktični maksimum.

Među prednostima paralelnog Quick sorta ističu se njegova efikasnost kod velikih datasetova, gdje višestruki procesori ili jezgra mogu istovremeno sortirati različite dijelove niza, što značajno ubrzava ukupan proces. Takođe, zbog toga što koristi lokalnu memoriju i cache-friendly pristup, često nadmašuje Merge dort u realnim aplikacijama, posebno kada se koristi sa modernim višedretnim okruženjima i hardverom. Osim toga, jednostavno se implementira korišćenjem rekurzivnog fork-join modela.

Sa druge strane, mane uključuju kompleksniju kontrolu nad paralelizacijom – potrebno je pažljivo upravljati brojem niti kako se ne bi stvorio preveliki overhead. Prevelik broj paralelnih poziva može dovesti do zagušenja sistema, dok premali broj neće iskoristiti puni potencijal paralelizacije. Takođe, izbor lošeg pivota vodi do nebalansiranih grana, što umanjuje efikasnost raspodjele posla među procesorima. Još jedan izazov je što dijeljenje podataka među nitima može uzrokovati probleme sa cache-koherencijom i pristupom memoriji, posebno u sistemima sa Non-Uniform Memory Access (NUMA) arhitekturom.[[10]](#footnote-10)

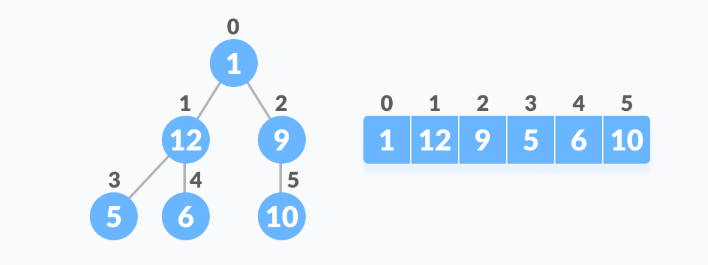
# HEAP SORT algoritam

Heap sort je efikasan i deterministički algoritam za sortiranje koji koristi strukturu podataka poznatu kao heap (hrpa), tačnije maks-heap – potpuno binarno stablo u kojem je svaki roditeljski čvor veći ili jednak od svojih potomaka. Ključna prednost ovog algoritma leži u tome što se heap može efikasno predstaviti pomoću običnog niza, bez potrebe za korišćenjem dodatnih pokazivača ili složenih struktura. Ova reprezentacija se oslanja na specifičan matematički odnos između indeksa u nizu i pozicija čvorova u binarnom stablu.

Naime, kompletno binarno stablo ima osobinu da za svaki element sa indeksom *i* u nizu: lijevo dijete se nalazi na indeksu *2i + 1*, desno dijete se nalazi na indeksu *2i + 2*, a roditelj bilo kog elementa nalazi se na indeksu *(i - 1) / 2* (u oznaci cio dio pod).

Ovaj odnos omogućava izuzetno brzu i jednostavnu navigaciju kroz strukturu stabla korišćenjem samo aritmetike nad indeksima. Zahvaljujući tome, Heap sort se može implementirati unutar jednog jedinog niza (tzv. in-place), bez potrebe za dodatnim memorijskim strukturama.

U osnovi, Heap sort funkcioniše tako što prvo pretvara ulazni niz u maks-heap – strukturu u kojoj je najveći element uvijek na vrhu (korijenu) – a zatim postepeno uklanja najveći element, premješta ga na kraj niza i ponovo prilagođava preostali niz u validan heap. Ovaj proces se ponavlja sve dok svi elementi ne budu sortirani u uzlaznom redosledu. Zahvaljujući stabilnoj vremenskoj složenosti i malim prostornim zahtjevima, Heap sort je naročito pogodan za rad sa velikim količinama podataka u sistemima ograničenih resursa.



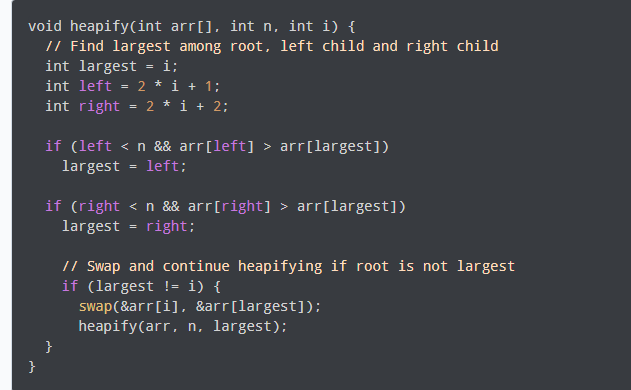
39. Ilustracija veze između niza i stabla

Na prikazanoj slici broj 39 jasno se vidi kako se elementi binarnog stabla mogu predstaviti pomoću običnog niza, prateći pravila indeksiranja karakteristična za heap strukturu. Svaki čvor u stablu ima određenu poziciju u nizu, a odnosi između roditelja i djece se računaju jednostavnom formulom: za čvor na indeksu i, njegovo lijevo dijete se nalazi na indeksu 2i + 1, a desno dijete na indeksu 2i + 2. Na primjeru, element 1 na indeksu 0 ima lijevo dijete 12 (indeks 1) i desno dijete 9 (indeks 2). Zatim, 12 na indeksu 1 ima lijevo dijete 5 (indeks 3) i desno dijete 6 (indeks 4). Ovakva organizacija omogućava jednostavno i efikasno upravljanje binarnim stablom unutar jednog niza, što je ključno za rad Heap sort algoritma.

## Implementacija algoritma

Prvi korak – izgradnja maks heap-a

Prva faza u implementaciji Heap sort algoritma jeste izgradnja maks-heapa iz proizvoljnog niza. Maks-heap je binarno stablo u kojem je svaki roditeljski čvor veći od svojih potomaka, a kako bi se postigla ta struktura, koristi se funkcija heapify (slika 39.1), koja postepeno popravlja lokalna kršenja ovog pravila.

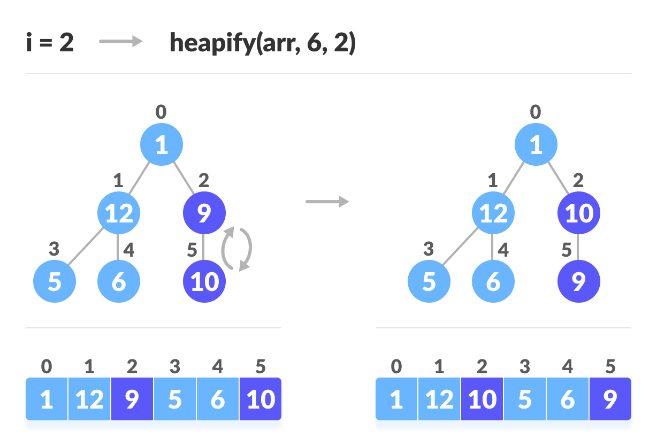
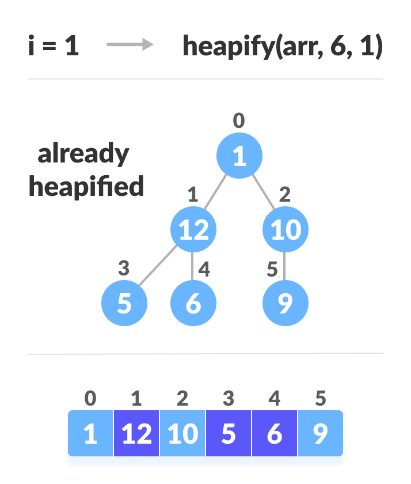


39.1. Implementacija heapify funkcije

*Funkcija heapify predstavlja ključnu komponentu Heap sort algoritma i služi za održavanje maks-heap svojstva. Ona upoređuje vrijednosti roditelja i njegove djece, i ukoliko roditelj nije najveći, vrši zamjenu sa najvećim djetetom. Zatim se rekurzivno poziva za novo pomjerenu poziciju, sve dok se ne uspostavi validna heap struktura. Ova funkcija osigurava da lokalno podstablo bude pravilno organizovano kao maks-heap.*

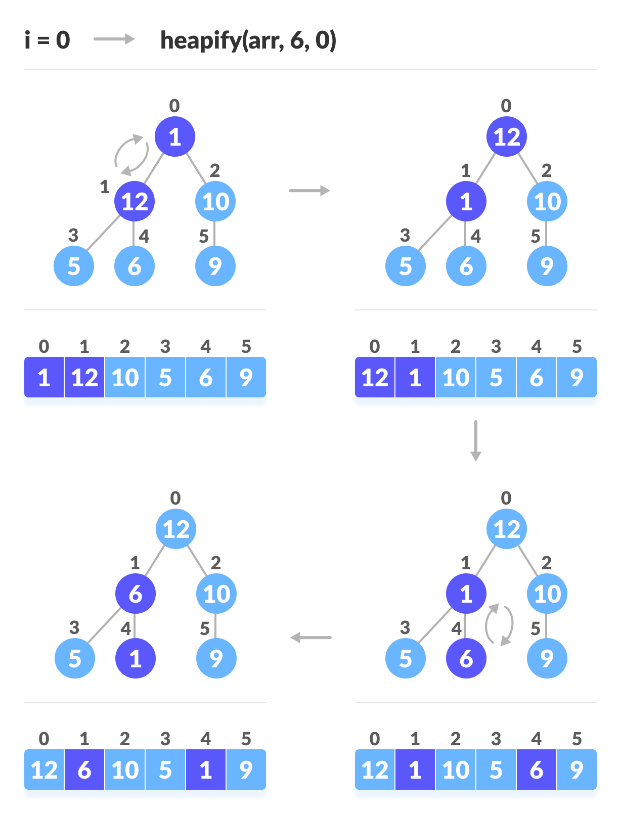
Proces izgradnje počinje od poslednjeg roditeljskog čvora, a ne od korijena. Za niz dužine n, prvi čvor koji nije list nalazi se na poziciji n/2 - 1. Svi čvorovi nakon toga su listovi i samim tim već zadovoljavaju uslov heap-a. Dakle, heapify se primjenjuje od indeksa n/2 - 1 do 0, pri čemu se koriguju lokalne greške od dna ka vrhu stabla – strategija poznata kao bottom-up pristup.

Na slici 40 (heapify na indeksu 2), čvor 9 ima desnog potomka 10, koji je veći, pa dolazi do zamjene i 10 zauzima roditeljsku poziciju. Rezultat je lokalno validan heap. Na slici broj 41 (heapify na indeksu 1), čvor 12 već zadovoljava uslov da je veći od svojih potomaka (5 i 6), pa se ništa ne mijenja.

Slike 40 i 41 – primjeri rada heapify funkcije

Na trećem koraku – slika 42 (heapify na indeksu 0), vršni čvor 1 krši pravilo jer je manji od svog lijevog djeteta 12, pa se vrši zamjena. Nakon toga se nastavlja heapify rekurzivno i dalje prema dolje, sve dok svi čvorovi ne budu u skladu s pravilom maks-heapa. U završnom stanju, niz [1, 12, 10, 5, 6, 9] postaje [12, 6, 10, 5, 1, 9], gdje je 12 sada na vrhu, a svi ostali elementi organizovani tako da zadovoljavaju maks-heap strukturu.



42. Primjer heapify funkcije i rekurzije

Ova faza je od presudnog značaja jer postavlja osnovu za ostatak Heap sort algoritma: tek nakon formiranja validnog maks-heapa može započeti proces uklanjanja najvećih elemenata i njihovog postavljanja na kraj niza, što vodi ka potpunom sortiranju.

Drugi korak – rad Heap sort algoritma

Nakon što se niz transformiše u maks-heap, svaki naredni korak u Heap sort algoritmu uključuje izvlačenje najvećeg elementa iz heap-a, njegovo premještanje na kraj niza, i obnavljanje strukture heap-a pomoću funkcije heapify. Pošto heap zadovoljava Max-heap svojstvo, najveći element se uvijek nalazi u korijenu stabla, tj. na indeksu 0.

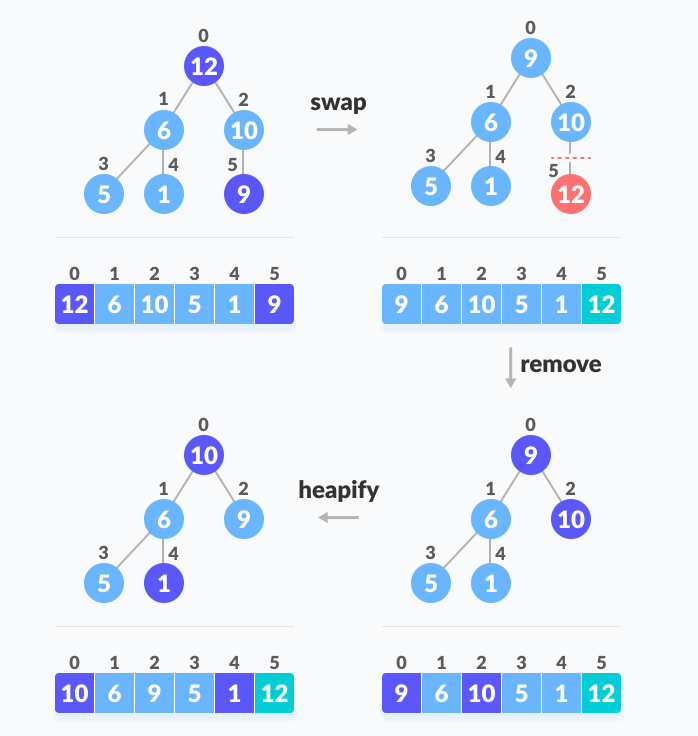
Proces se sastoji iz tri ključna koraka koji se neprestano ponavljaju:

1. Swap (zamjena): Najveći element (na vrhu) se zamjenjuje sa posljednjim elementom u nizu. Na taj način najveći element dolazi na svoje konačno mjesto u sortiranom nizu, na samom kraju.
2. Remove (uklanjanje): Nakon zamjene, veličina heap-a se smanjuje za 1, jer je najveći element već sortiran i više nije dio heap-a.
3. Heapify: Korijenski element (koji je zamijenjen manjim) možda više ne zadovoljava heap svojstvo, pa se poziva funkcija heapify da se obnovi Max-heap struktura u preostalom dijelu niza.

Ova tri koraka se ponavljaju sve dok svi elementi ne budu sortirani – tj. dok heap ne postane prazan.

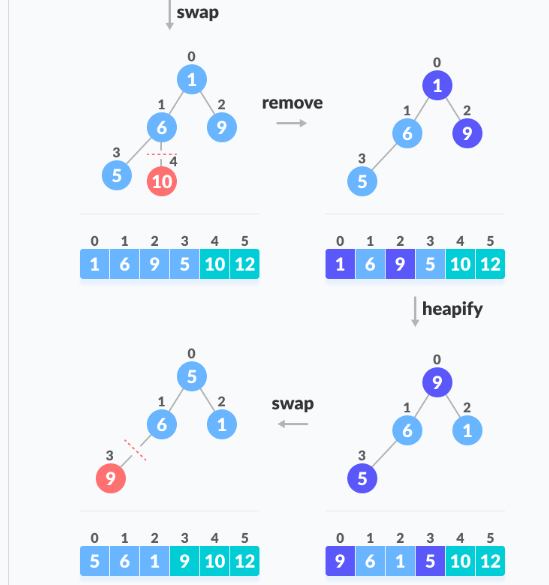
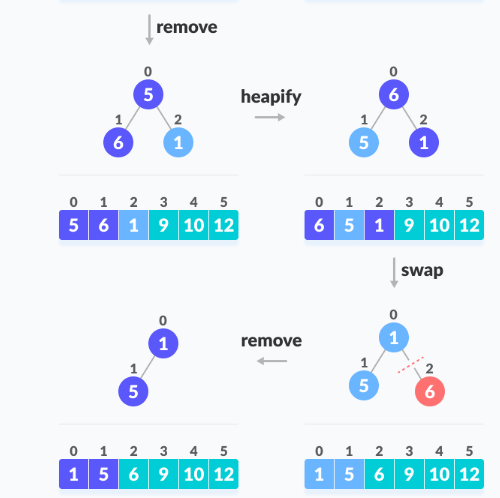
*Slijedi detaljno objašnjenje ovih koraka, uz ilustracije.*

Na prikazanoj slici 43 detaljno je predstavljen drugi dio rada Heap sort algoritma, tj. faza sortiranja nakon što je niz već transformisan u maks-heap. Na početku slike vidimo kompletan maks-heap, gdje je vrijednost 12 na vrhu, kao najveći element u stablu. U prvom koraku, korijenski element 12 se zamjenjuje sa posljednjim elementom 9 u nizu, nakon čega se 12 izdvaja kao sortirani i postavlja na kraj. Sada se heapify poziva za novi korijen (9), i budući da 10 kao desno dijete ima veću vrijednost, dolazi do zamjene: 10 dolazi na vrh, čime se ponovo uspostavlja maks-heap struktura.



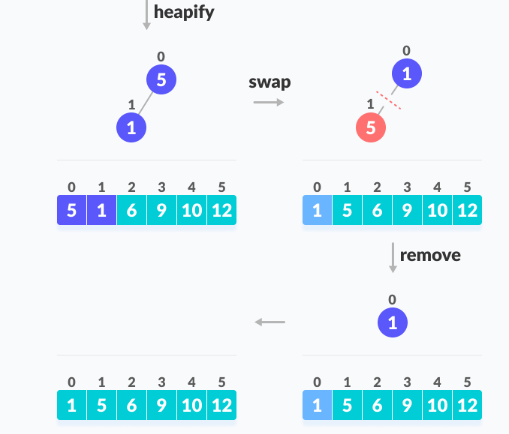
43. Primjer rada algoritma

U sljedećem koraku – slika 44, 10 se uklanja (tj. zamjenjuje sa 1), koji sada postaje novi korijen. Ponovo se poziva heapify: 1 je manji od oba potomka, pa ga prvo zamjenjuje 6, zatim ga 5 dodatno potiskuje niže u stablu, dok se struktura ponovo ne stabilizuje. Taj postupak se ponavlja rekurzivno – svaki put se korijen (najveći preostali element) premješta na kraj aktivnog dijela niza, zamjenjuje se sa posljednjim i heapify funkcija obnavlja lokalni poredak. (slika 45)

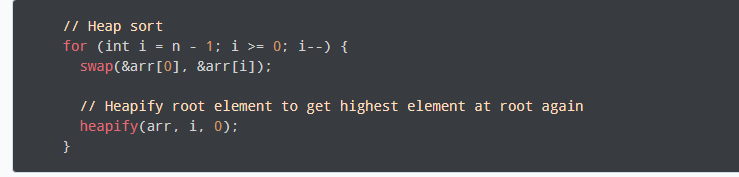
44. i 45.– nastavci primjera

Kako se heap smanjuje, tako elementi 9, 6, 5 i na kraju 1 dolaze na svoje odgovarajuće pozicije. Na svakom nivou se vizuelno prikazuje promjena u stablu i odgovarajuća izmjena u nizu ispod njega. U završnim koracima (slika 46) jasno se vidi kako se poslednji nesortirani elementi rotiraju i postavljaju u tačan redosljed – što na kraju rezultira potpuno sortiranim nizom [1, 5, 6, 9, 10, 12].



46. Kraj primjera

Date slike na jasan način prikazuju mehaniku sortiranja u Heap sort algoritmu: sve se vrti oko uklanjanja korijena i pozivanja heapify nad preostalim elementima. Istovremeno, slike vizualizuju kako se binarno stablo "ruši" odozgo dok se niz gradi od pozadi – što je suština rada ovog algoritma.



47. Implementacija Heap sort algoritma

Prikazani kod na slici 47 prikazuje završnu fazu Heap sort algoritma – samu operaciju sortiranja. U svakoj iteraciji petlje, najveći element koji se nalazi na početku niza (indeks 0) zamjenjuje se sa posljednjim nesortiranim elementom (indeks i). Nakon zamjene, heapify funkcija se poziva za korijen kako bi se ponovo uspostavilo maks-heap svojstvo u preostalom dijelu niza. Petlja se izvršava dok svi elementi ne budu pravilno sortirani, od najmanjeg (na početku) do najvećeg (na kraju niza).

## Složenost algoritma

### Vremenska složenost (Time Complexity)

Heap sort algoritam ima vremensku složenost O(n log n) u svim slučajevima — najboljem, prosječnom i najgorem. Tokom faze izgradnje maks-heapa, algoritam primjenjuje heapify funkciju na sve unutrašnje čvorove (njih je oko n/2), a svaki od njih može zahtijevati najviše log n koraka da bi se uspostavilo heap svojstvo. To ovu fazu čini složenosti O(n log n). U fazi sortiranja, najveći element (iz korijena heap-a) se premješta na kraj niza, a ostatak se ponovo heapifikuje. Budući da se heapify poziva n puta, svaka s maksimalno log n operacija, složenost i ove faze je O(n log n). Ukupna vremenska složenost ostaje O(n log n), jer se faze izvode sekvencijalno, a ne ugniježdeno.

### Prostorna složenost

Heap sort je algoritam koji se izvršava in-place, što znači da za svoje djelovanje ne zahtijeva dodatne strukture podataka, već sve promjene vrši unutar samog ulaznog niza. Zbog toga ima prostornu složenost O(1) – koristi samo konstantnu količinu dodatne memorije bez obzira na veličinu ulaznih podataka. Ovo ga čini pogodnim za sisteme sa ograničenim memorijskim resursima. Međutim, Heap sort nije stabilan algoritam, što znači da može promijeniti redosljed elemenata koji imaju istu vrijednost tokom sortiranja.[[11]](#footnote-11)

## Paralelizacija algoritma

Paralelizacija Heap sort algoritma je tehnički moguća, ali je znatno manje pogodna i efikasna u odnosu na algoritme poput Merge sort-a ili Quick sort-a, zbog njegove specifične strukture i zavisnosti između operacija. Heap sort funkcioniše tako što se od ulaznog niza prvo izgradi maksimalna hrpa (max-heap), a zatim se elementi postepeno uklanjaju sa vrha hrpe i smještaju na kraj niza. Sam proces izgradnje hrpe ima složenost *O(n),* dok operacije uklanjanja i održavanja svojstva gomile (heapify) imaju složenost *O(logn)* po elementu, što ukupno daje ukupnu složenost *O(nlogn)*.

Kada je riječ o paralelizaciji, najveća prepreka leži u zavisnostima među čvorovima u binarnom stablu hrpe – svaka heapify operacija zavisi od rezultata svojih potomaka, što znači da se ne mogu izvršavati potpuno nezavisno.

Ipak, određeni stepen paralelizacije moguć je u fazi izgradnje hrpe, i to posebno kod heapify poziva na nižim nivoima stabla (npr. listovi i njihovi neposredni roditelji), jer su ti pozivi međusobno nezavisni. Ovakav pristup poznat je kao bottom-up parallel heap construction, i omogućava da se izgradnja gomile u optimalnim uslovima smanji na vremensku složenost *O(logn)* uz korišćenje više procesorskih jezgara.

Međutim, druga faza Heap sort-a – vađenje maksimuma i reorganizacija hrpe– ostaje strogo sekvencijalna, jer svaka naredna operacija zavisi od prethodno ažurirane strukture hrpe. Zbog ovih faktora, ukupna vremenska složenost paralelizovanog Heap sort-a ostaje *O(nlogn)*, kao i kod sekvencijalne verzije, dok su stvarne performanse ograničene brojem paralelno izvršivih koraka.

Upravo zbog toga, Heap sort se u praksi rijetko koristi u paralelnim okruženjima, jer overhead sinhronizacije i zavisnosti često nadmašuju dobitke koje donosi paralelizacija. Iako je moguće paralelizovati određene dijelove Heap sort-a, njegova rekurzivna i zavisna struktura čini ga neefikasnim izborom za paralelnu obradu velikih skupova podataka, u poređenju sa algoritmima koji imaju bolju skalabilnost u paralelnim sistemima.[[12]](#footnote-12)

# ZAKLJUČAK

Sortiranje predstavlja ključnu operaciju u obradi podataka, kojom se elementi unutar skupa raspoređuju prema unaprijed definisanom redoslijedu – bilo da je riječ o uzlaznom, silaznom, abecednom ili nekom drugom kriterijumu. U savremenim računski zahtjevnim okruženjima, sekvencijalni algoritmi za sortiranje često nijesu dovoljno efikasni kada se radi o velikim količinama podataka. Upravo zbog toga se poseže za paralelnim algoritmima, koji omogućavaju kraće vrijeme izvršavanja i bolju iskorišćenost dostupnih resursa.

U prethodnim poglavljima analizirali smo nekoliko najpoznatijih i najčešće korišćenih algoritama sortiranja. Pokazalo se da se određeni algoritmi, kao što su Bubble sort, Merge sort i Quick sort, mogu uspješno paralelizovati, čime se značajno povećava njihova efikasnost. S druge strane, algoritmi poput Selection sorta, Insertion sorta i Heap sorta, zbog svoje strukturne zavisnosti i sekvencijalne prirode, ne pružaju značajne benefite u paralelnom izvođenju.

U ovom radu detaljno smo objasnili način funkcionisanja pomenutih algoritama i razmotrili potencijale i ograničenja njihove paralelizacije. Praktična implementacija paralelnih rješenja i njihova evaluacija biće predmet zasebnog projektnog zadatka, gdje će se teorijski uvidi iz ove analize konkretno primijeniti i testirati u stvarnom okruženju.

**LITERATURA**

1. „Algoritmi sortiranja“, Škola koda, pristupljeno 29. mart 2025, dostupno na: https://skolakoda.github.io/algoritmi-sortiranja.
2. „Bubble Sort Algorithm“, Programiz, pristupljeno 29. mart 2025, dostupno na: https://www.programiz.com/dsa/bubble-sort.
3. Barry Wilkinson i Michael Allen, Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, Second Edition, Prentice Hall, 2005, str. 303–311.
4. “Chapter 9: Odd-Even Transposition Sort,” Parallel and Distributed Algorithms, pristupljeno: 6. april 2025, dostupno na: http://users.atw.hu/parallelcomp/ch09lev1sec3.html
5. „Selection Sort Algorithm“, Programiz, pristupljeno 29. mart 2025, dostupno na: https://www.programiz.com/dsa/selection-sort.
6. „Insertion Sort Algorithm“, Programiz, pristupljeno 29. mart 2025, dostupno na: https://www.programiz.com/dsa/insertion-sort.
7. “Merge Sort,” Programiz, pristupljeno 31. mart 2025, dostupno na: https://www.programiz.com/dsa/merge-sort
8. Barry Wilkinson i Michael Allen, Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, Second Edition, Prentice Hall, 2005, str. 332–324.
9. “Quick sort,” Programiz, pristupljeno 01. april 2025, dostupno na: https://www.programiz.com/dsa/merge-sort
10. Barry Wilkinson i Michael Allen, Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, Second Edition, Prentice Hall, 2005, str. 334–336
11. “Heap sort,” Programiz, pristupljeno 01. april 2025, dostupno na: https://www.programiz.com/dsa/merge-sort
12. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 3rd ed., MIT Press, 2009, str. 151–163; K. Batcher, "Design of a Parallel, Hardware Priority Queue," IEEE Transactions on Computers, vol. C-29, no. 10, 1980, str. 833–840.

1. „Algoritmi sortiranja“, Škola koda, pristupljeno 29. mart 2025, dostupno na: https://skolakoda.github.io/algoritmi-sortiranja. [↑](#footnote-ref-1)
2. „Bubble Sort Algorithm“, Programiz, pristupljeno 29. mart 2025, dostupno na: https://www.programiz.com/dsa/bubble-sort. [↑](#footnote-ref-2)
3. Barry Wilkinson i Michael Allen, Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, Second Edition, Prentice Hall, 2005, str. 303–311. [↑](#footnote-ref-3)
4. “Chapter 9: Odd-Even Transposition Sort,” Parallel and Distributed Algorithms, pristupljeno: 6. april 2025, dostupno na: http://users.atw.hu/parallelcomp/ch09lev1sec3.html [↑](#footnote-ref-4)
5. „Selection Sort Algorithm“, Programiz, pristupljeno 29. mart 2025, dostupno na: https://www.programiz.com/dsa/selection-sort. [↑](#footnote-ref-5)
6. „Insertion Sort Algorithm“, Programiz, pristupljeno 29. mart 2025, dostupno na: https://www.programiz.com/dsa/insertion-sort. [↑](#footnote-ref-6)
7. “Merge Sort,” *Programiz*, pristupljeno 31. mart 2025, dostupno na: https://www.programiz.com/dsa/merge-sort [↑](#footnote-ref-7)
8. Barry Wilkinson i Michael Allen, Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, Second Edition, Prentice Hall, 2005, str. 332–324. [↑](#footnote-ref-8)
9. “Quick sort,” *Programiz*, pristupljeno 01. april 2025, dostupno na: <https://www.programiz.com/dsa/merge-sort> [↑](#footnote-ref-9)
10. Barry Wilkinson i Michael Allen, Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, Second Edition, Prentice Hall, 2005, str. 334–336. [↑](#footnote-ref-10)
11. “Heap sort,” *Programiz*, pristupljeno 01. april 2025, dostupno na: https://www.programiz.com/dsa/merge-sort [↑](#footnote-ref-11)
12. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, 2009, str. 151–163; K. Batcher, "Design of a Parallel, Hardware Priority Queue," *IEEE Transactions on Computers*, vol. C-29, no. 10, 1980, str. 833–840. [↑](#footnote-ref-12)