Algoritmi i rešavanje problema

Upotreba računara za rešavanje nekog problema najčešće zahteva programiranje, odnosno formulisanje plana za precizno definisane akcije, koje računar treba da obavi da bi se dobilo rešenje problema. Takav plan je algoritam.

Algoritmi su planovi za obavljanje akcija. Algoritmi nisu vezani samo za programiranje. Neki opšti primeri algoritama bi mogli biti:

- objašnjenje kako doći do određene adrese
- recept za pripremu nekog jela
- vez

Ovi algoritmi zahtevaju različit nivo tehničkog znanja da bi neko uspeo da ih izvrši. Prvi algoritam se može izraziti običnim jezikom. Drugi algoritam traži da imate malo znanja iz kuvanja, dok bi treći tražio iskustvo u čitanju šema za vez.

Algoritmi se međusobno razlikuju i po nivou detalja koji su u njima dati. Programski jezici su metodi za predstavljanje algoritama, kako ljudima tako i računarima. Različiti programski jezici mogu da podrže različit nivo detalja. Jezici nižeg nivoa algoritme predstavljaju sa previše detalja, tako da ih ljudi teško mogu ispratiti. Zbog toga se koriste jezici višeg nivoa, a računar može da prevede algoritam izražen u ovakvom programskom jeziku u instrukcije koje procesor izvršava.

Računari i rešavanje problema

Programiranje je pre svega aktivnost koja se odnosi na rešavanje problema. Matematičar Džordž Polia je rešavanje problema podelio na četiri aktivnosti:

1. **Shvatanje problema**: Ovaj prvi korak može biti težak, ali je vrlo bitan. Bilo bi vrlo glupo tražiti odgovor na pitanje koje niste potpuno shvatili. Generalno, moraju se pronaći dati podaci, šta je nepoznato (željeni rezultat), kao i veze između datih podataka i onog što je nepoznato. Vrlo je važno biti siguran da su date informacije dovoljne za rešavanje problema.
2. **Smišljanje plana**: Nakon što se shvati problem, neko mora smisliti plan akcija koje će taj problem rešiti. Plan se pravi kretanjem od podataka do rezultata, u skladu sa vezama koje između njih postoje. Ako je problem trivijalan, ovaj korak ne traži mnogo razmišljanja. Generalno se mogu pomenuti sledeće tehnike:
    - Pronalaženje sličnih, rešenih problema
    - Preoblikovanje originalnog problema, tako da liči na poznati problem.
    - Skraćivanje problema na manji problem, koji se može rešiti
    - Generalizacija problema
    - Pronalaženje postojećeg rada, koji može pomoći u traženju rešenja
3. **Izvršenje plana**: Nakon što je definisan plan, njega bi se trebalo u potpunosti pridržavati. Svaki element plana treba da se proveri i primeni. Ako se zaključi da neki elementi plana nisu dobri, plan treba promeniti.
4. **Ocena**: Na kraju treba ispitati rezultat, da biste bili sigurni da je on u redu i da je problem rešen.

Kao što znamo uvek postoji više načina da se reši isti problem. Kada se različiti ljudi susretnu sa istim problemom, verovatno je da će svaki od njih problem rešiti na drugi način. Ipak, da bi osoba bila efikasna, ona mora da prihvati sistematski metod rešavanja problema. Ako se problem rešava pomoću računara, sistematičnost je krucijalna. Ako nema sistematičnosti čovek se lako može naći u situaciji da suviše kasno zaključi da je trebalo više vremena da posveti planiranju.

Na bazi metoda za rešavanje problema koji je ovde iznet može se izvesti metod za rešavanje problema u više koraka. Ovaj metod se može prilagoditi stilu svakog pojedinca. Metod ima puno zajedničkog sa životnim ciklusom softvera, odnosno njegovim fazama.

Ti koraci su:

1. Definisanje problema
2. Projektovanje rešenja
3. Fino podešavanje rešenja

4. Razvoj strategije testiranja
5. Kodiranje i testiranje programa
6. Pisanje dokumentacije
7. Održavanje programa

Definisanje problema

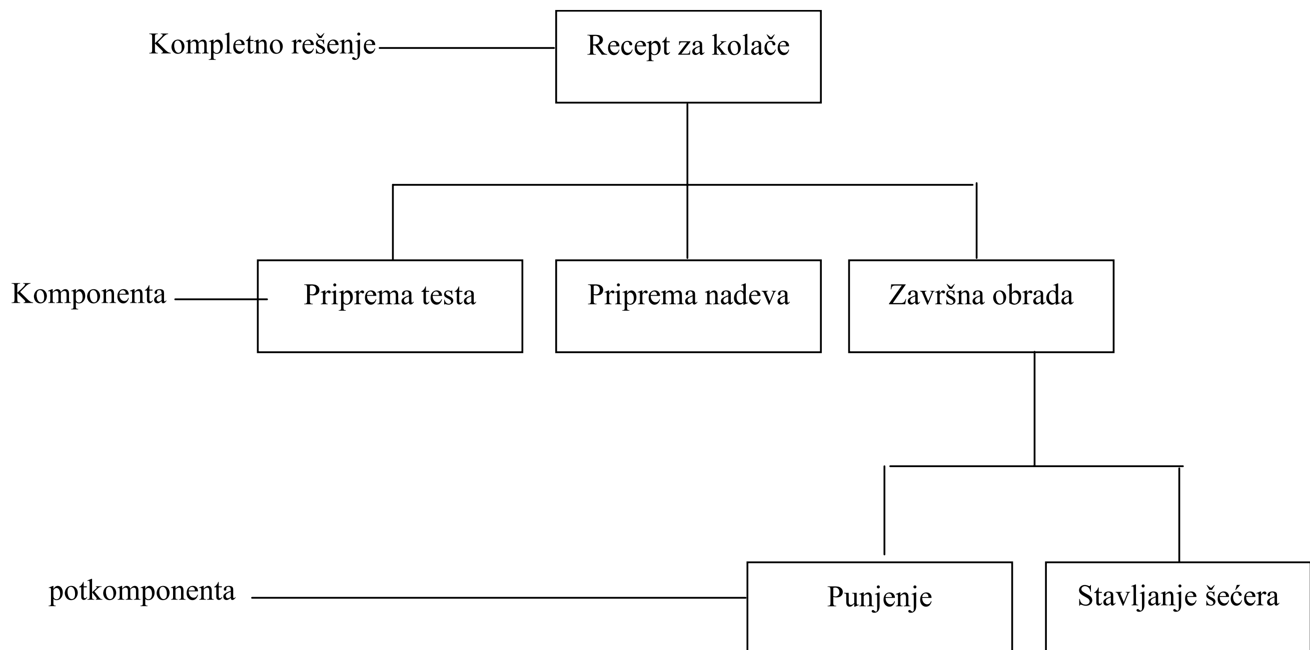
Prvi korak je da se problem potpuno shvati i razume. Inicijalni opis problema je obično maglovit. Mora se napraviti precizan opis. Onaj ko iznosi problem (korisnik) i onaj ko taj problem rešava (programer) moraju zajedno raditi da bi bili sigurn da su oboje shvatili problem. Ovo bi trebalo da vodi do kompletne specifikacije problema, uključujući preciznu definiciju ulaznih podataka (dati podaci), kao i željenog izlaza (rezultata).

Projektovanje rešenja

U ovom koraku treba definisati okvir rešenja. Polazi se od originalnog problema, koji se deli na manje potprobleme. Potprobleme je lakše rešiti pošto su manji od originalnog problema. Njihova rešenja će biti komponente krajnjeg rešenja. Isti metod "podeli pa vladaj" se primenjuje i na potprobleme, sve dok se ceo problem ne pretvori u plan sa precizno definisanim koracima.

Objasnimo ovo detaljnije na primeru pripreme kolača. Ovo nije računarski problem, ali je dobar primer, da se shvati princip. Recimo da domaćica pravi kolače. Kolači su od testa, koje se puni nekim nadevom i na kraju se sve to preliva šećerom u prahu. Prvo se pripremi testo, koje se zatim oblikuje prema želji. Posle toga se testo peče 20 minuta. Zatim se stavlja nadev, koji je prethodno pripremljen i na kraju se sve to pospe šećerom u prahu.

Ceo problem se može podeliti na tri potproblema. Pravljenje testa, priprema nadeva i završno pravljenje kolača. Svaki od potproblema se može dalje deliti, dok ne to ne postanu ekstremno mali potproblemi. U toku ovog koraka nastaje dijagram koji ilustruje strukturu rešenja. Ovaj dijagram se naziva strukturnim dijagramom, pošto pokazuje strukturne komponente rešenja, kao i veze između komponenti.



Fino podešavanje rešenja

Rešenje koje se dobija u prethodnom koraku je na visokom nivou. Tu je krajnji zadatak razlaganje na nekoliko podzadataka, ali nema indikacija kako bi se ti podzadaci izvršili. Opšte rešenje se mora dalje razlagati, odnosno moraju se dodati detalji.

Svaki pravougaonik sa strukturnog dijagrama se mora vezati uz precizan opis metoda za izvršenje tog zadatka. Precizan opis metoda podrazumeva niz dobro definisanih koraka, koji kada se izvrše završavaju taj zadatak. Taj niz koraka se naziva

algoritmom. Algoritmi se obično daju u pseudokodu. Pseudokod je poseban jezik koji se koristi za opis onog što se radi sa podacima. Ovaj jezik je nezavisan od konkretnog programskog jezika.

Recimo da bi se zadatak pripreme nadeva mogao predstaviti na sledeći način:

```
Stavljanje šlaga u mikser
Stavljanje ostalih sastojaka
Izbor režima rada miskera
Pokretanje miksera
Zaustavljanje miksera
Vađenje nadeva iz miksera
```

Ukoliko neki od koraka nije dovoljno jasan postupak se može ponovo primeniti, odnosno može se vršiti dalje raščlanjivanje, sve dok se ne dobije rešenje koje je potpuno jasno i upotrebljivo.

Razvoj strategije testiranja

U prethodnom koraku je napravljen algoritam koji se koristi za pisanje programa koji odgovara rešenju. Ovaj program će se izvršavati na računaru i od njega se očekuje da proizvede ispravno rešenje. Vrlo je bitno proveriti taj program u više različitih situacija, sa različitim kombinacijama ulaznih podataka, da bismo bili sigurni da su rešenja u redu.

Svaki test predstavlja različitu kombinaciju ulaznih podataka. Program se testira tako što se napravi niz slučajeva testiranja, koji pokrivaju ne samo uobičajene ulazne podatke, već i ekstremne vrednosti. Pored toga se koriste i specijalne kombinacije ulaznih veličina, da bi se testirali konkretni aspekti programa.

Pogledajmo na primer, program za plate. Test slučajevi bi trebalo uključuju vrednosti koje pokrivaju ceo opseg plata, uključujući i krajnje vrednosti, maksimalan broj radnih sati itd. Tipični test bi mogao da definiše nadnicu za jedan sat, radne sate i očekivane rezultate.

```
nadnica = $10.00 radniSati = 35 rezultat = $350.00
```

Slučajeve testiranja je najbolje napraviti pre pisanja programa, pošto se oni mogu koristiti i za proveru algoritma, a i zato što pritisak za što brži završetak programa može dovesti do gubitka objektivnosti, ako se testira u letu.

Kodiranje i testiranje programa

Algoritmi napisani u pseudokodu ne mogu da se izvrše direktno na računaru. Taj pseudokod mora prvo da se prevede u programski jezik. To je proces kodiranja.

Nakon kodiranja algoritma, program mora da se testira. To se radi na osnovu strategije testiranja, koja je napravljena u prethodnom koraku. Program mora da se izvrši i za svaki slučaj testiranja mora da se proveri da li rezultat odgovara onome što je predviđeno. Ako ima razlike rezultat treba analizirati, da bi se pronašla greška. Greška može biti na jednom od četiri mesta:

- **Kodiranje**: Greška je nastala u prevođenju algoritma u programski jezik.
- **Algoritam**: U algoritmu je postojala greška koja nije primećena.
- **Dizajn programa**: Možda je dizajn programa loš, pa je to dovelo do greške.
- **Izračunavanje rezultata testiranja koji se očekuju**: Možda je rezultat testiranja loše proračunat.

Nakon što se pronađe greška, prave se revizije. Ovo se ponavlja sve dok program ne da očekivane rezultate za sve slučajeve testiranja. Proces kodiranja i testiranja se jednim imenom naziva implementacijom.

Pisanje dokumentacije

Pisanje dokumentacije počinje sa prvim korakom u razvoju programa i nastavlja se tokom životnog ciklusa programa. Dokumentacija mora da sadrži sledeće:

- Objašnjenje svih koraka i metoda.
- Odluke vezane za dizajn programa.
- Probleme koji su se javili tokom pisanja i testiranja programa.

Završna dokumentacija mora da sadrži i odštampanu kopiju programa, tzv. listing. Taj listing treba da bude odštampan sa odgovarajućim zaglavljem, tako da se lakše prati. U programima treba da postoje komentari. Ti komentari su kasnije naročito korisni, kada dođe do promene programa.

Dokumentacija mora da sadrži i određene instrukcije za korisnika. Ako programe koriste ljudi koji nisu bliski sa računarima, mora postojati korisničko upustvo, koje bez tehničkih izraza, objašnjava kako se program koristi, kako se pripremaju ulazni podaci i kako se interpretiraju rezultati.

Održavanje programa

Održavanje programa nije direktno deo procesa implementacije. Tu spadaju sve aktivnosti koje se dešavaju nakon što program počne da se koristi. Termin održavanje, kada se primeni na programe, ima različito značenje od uobičajenog. Kuće, automobili ili drugi objekti imaju održavanje zato što se vremenom troše. Programi, međutim, nemaju fizičke osobine, već samo logičke, tako da se ne troše usled upotrebe.

Može se desiti da program izvršite 100000 puta i da on radi dobro, a da se u sledećem prolazu desi greška. To može dovesti u zabludu da program nije dobar, ali se u suštini desilo da je program naišao na neke okolnosti koje nisu bile predviđene. Drugim rečima, to znači da program nije bio testiran u tim okolnostima i da od početka nije radio pod tim uslovima. U ovom smislu održavanje programa se uglavnom odnosi na kompletiranje procesa implementacije.

Održavanje programa uključuje i poboljšanja za koja se, tokom upotrebe programa, zaključi da su potrebna. Održavanje programa uključuje:

- eliminisanje grešaka koje su naknadno otkrivene.
- modifkaciju tekućeg programa
- dodavanje novih osobina u program
- ažuriranje dokumentacije

Vrlo često se dešava da cena održavanja programa prevaziđe inicijalnu cenu koja je plaćena za prvobitni razvoj.

Algoritmi

Definicija: Algoritam je precizan plan za obavljanje niza akcija, koje vode određenom cilju.

Algoritmi mogu biti, na primer:

- priprema doručka
- izračunavanje srednje vrednosti
- promena gume na automobilu
- igranje jamba

Obratite pažnju na to da svaki od prethodno navedenih algoritama definiše dve stvari:

1. Akciju, koja se zadaje glagolima, kao što su "priprema", promena, igranje
2. Podatke na kojima se akcija vrši. To su imenice, kao što su doručak, guma na automobilu i sl.

Iz ovih primera treba da shvatite da se algoritmi ne koriste samo kod računara, već postoje u svakodnevnom životu.

Kriterijumi za algoritme

Algoritmi treba da zadovolje sledeća četiri kriterijuma:

Kompletnost: Da bi algoritam bio kompletan, sve njegove akcije moraju biti egzaktno definisane. Pogledajmo na primer, algoritam koji opisuje kako stići do nekog restorana.

1. Krenite glavnom ulicom i na raskrsnici sa ulicom Kralja Aleksandra skrenite levo.
2. Idite pravo još jedan kilometar i bićete ispred restorana.

Da li je ovo algoritam? Nije, pošto uputstva nisu precizna. Na koju stranu glavne ulice treba krenuti. Ako već poznajete tu oblast onda možda i imate predstavu na koju stranu treba krenuti, ali ipak uputstva nisu precizna.

Nedvosmislenost: Skup instrukcija će biti nedvosmislen ako postoji samo jedan mogući način da se one interpretiraju. Na primer, čak i ako prihvatimo da su instrukcije za put do restorana dovoljno precizne, ipak postoji dvosmislenost o tome u kom smeru krenuti. Mi ne smemo da pretpostavljamo neko lokalno znanje, jer, ako poznajemo taj kraj, onda i znamo kako da dođemo do restorana. Dvosmislenost se može rešiti probom. Možemo da probamo da idemo na jednu stranu, pa ako pronađemo restoran onda i to treba dodati u algoritam.

Određenost: Treća osobina kaže da ako se prate instrukcije, onda je izvesno da će se postići željeni rezultat. Pogledajmo sledeći algoritam, koji pokazuje kako postati milioner:

1. Uzmite sav svoj novac iz banke i promenite ga u metalne novčiće.
2. Idite u kockarnicu.
3. Igrajte na mašinama sve dok ne dobijete milion dinara.

Jasno je da ovaj algoritam neće uvek dovesti do željenog rezultata, da postanete milioner. Mnogo je verovatnije da ćete izgubiti sav novac. Poenta je da ne možemo biti sigurni kakav će biti rezultat. Na taj način skup instrukcija nije određen, pa samim tim ni instrukcije ne predstavljaju algoritam.

Konačnost:Ova osobina kaže da se instrukcije moraju završiti nakon određenog broja koraka."Algoritam" kojim se postaje milioner i ovde ne zadovoljava. Može se desiti da nikad ne dobijete milion dinara, ali ni da izgubite sav novac. Drugim rečima, ako sledite instrukcije može se desiti da završite igrajući na automatima zauvek.

Konačnost se ne odnosi samo na završetak nakon određenog broja koraka, već i na to da instrukcije treba da koriste konačan broj promenljivih da se dobije željeni rezultat.

Osobine koje bi algoritmi trebalo da imaju

I kada algoritam zadovolji kriterijume koje smo pomenuli (preciznost, nedvosmislenost, određenost, konačnost), može se desiti da nije pogodan da se koristi kod rešenja problema putem računara. Poželjni atributi, koje jedan algoritam treba da ima su:

Opštost: Algoritam treba da rešava klasu problema, a ne samo jedan problem. Na primer, algoritam koji traži srednju vrednost od četiri zadate vrednosti, generalno nije tako koristan kao ona koji izračunava srednju vrednost za proizvoljan broj vrednosti.

Dobra struktura: Ovaj atribut se odnosi na konstrukciju algoritma. Algoritam sa dobrom strukturom treba da se pravi preko dobrih blokova, koji omogućavaju da se algoritam:

- lako objasni
- lako razume
- lako testira
- lako menja

Blokovi od kojih je algoritam sastavljen treba da budu tako napravljeni da se mogu lako zameniti boljom verzijom, bez nekih velikih promena.

Efikasnost: Brzina izvršenja algoritma je često važna osobina, isto kao i njegova veličina i kompaktnost. Inicijalno ovi atributi nisu toliko bitni. Prvo se mora napraviti algoritam sa dobrom strukturom koji nas vodi do željenog rešenja, pod svim uslovima. Tek nakon toga možemo poboljšavati efikasnost algoritma.

Lakoća upotrebe: Ova osobina se odnosi na udobnost i lakoću sa kojom korisnici algoritam mogu da primene na svoje podatke. Ponekad ono što omogućava da korisnik lakše razume algoritam, vodi ka teškoćama za programera (i obrnuto).

Predstavljanje algoritama

Da bi se iskoristili na pravi način, algoritmi se moraju predstaviti tako da ih razumemo. Postoji više načina za predstavljanje algoritama.

Verbalno predstavljanje: Kod ovakvog predstavljanja se algoritmi izražavaju u nekom od jezika koji ljudi koriste (svakodnevni jezik). Pogledajmo na primer, algoritam koji opisuje kako se izračunava nedeljna plata nekog službenika.

```
Ako službenik ima manje (ili jednako) od 40 radnih sati u toku nedelje,
onda se njegova plata računa kao broja radnih sati puta cena po satu
```

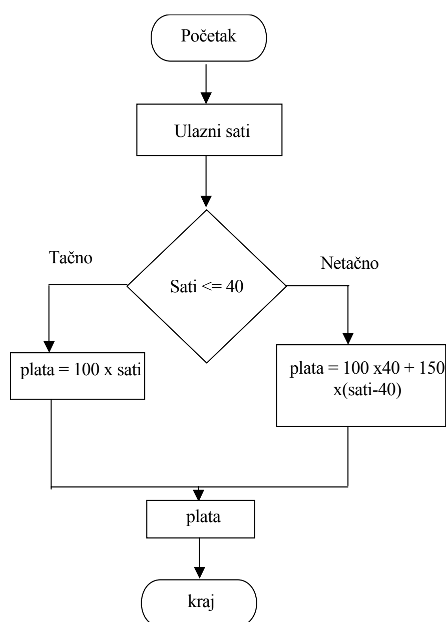
```
(100 din). Ako je radio više od 40 sati, onda se za svaki sat preko te
brojke, cena rada povećava i iznosi 150 din.
```

Ako je broj radnih sati, na primer, 25, onda se plata računa množenjem broja radnih sati (25) i cene rada (100 din.) što daje 2500. Ako je broj radnih sati 50, onda je plata sastavljena iz dva dela. Jedan deo je plata za uobičajenih 40 radnih sati i iznosi (40 x 100) 4000 din. Radni sati preko 40 se plaćaju posebno i ta suma je u ovom slučaju (50 - 40) x 150 = 1500 din. Ukupna suma je zbir ove dve sume i iznosi 5500 din. To je znači finalni izlaz algoritma.

Predstavljanje pomoću dijagrama toka: Kod ovog načina se algoritam predstavlja grafički preko simbola, koji su spojeni strelicama. Ovde se koriste sledeći simboli:

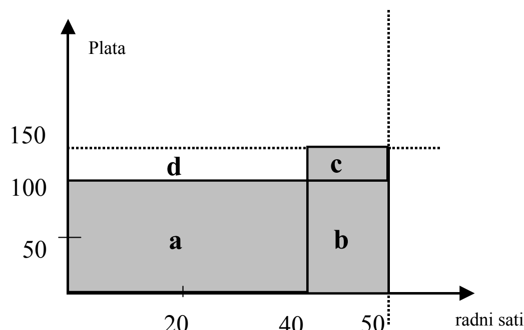
- Zaobljeni pravougaonici označavaju početak i kraj
- Obični pravougaonici označavaju akcije
- Rombovi označavaju odluke koje se donose. Oni sadrže i uslove koji određuju kojim putem se tok nastavlja.

Grafička forma omogućava ljudima da lakše prate redosled akcija, odnosno tok kontrole. Sledeća slika prikazuje algoritam koji smo prethodno verbalno opisali. Obratite pažnju na to da je plaćanje osnovne sume za 40 radnih sati prikazano kao 40x100, a ne kao suma 400. Ako bismo prikazali broj 400, onda bismo sakrili dve komponente dijagrama. U tom slučaju bi se 400 pojavio kao neki magični broj, a algoritam bi teži za razumevanje.



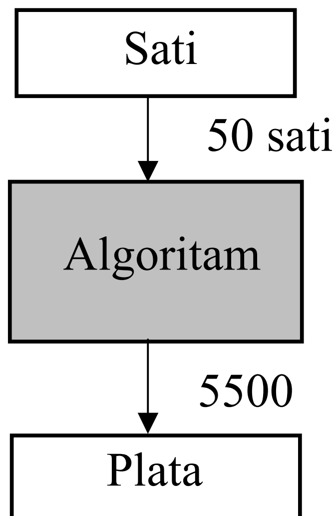
Grafikoni: Ovde je u pitanju predstavljanje algoritama preko dijagrama. Ono ljudima može pomoći da bolje shvate izvršenje algoritma. Grafikon na sledećoj slici pokazuje kako cena radnog sata zavisi od broja radnih sati. Ukupna suma koja se plaća za neki broj radnih sati je osenčeni deo. Na primer, ako je broj radnih sati 50, ukupna plata odgovara osenčenom delu koji se sastoji od tri mala pravougaonika označena sa a, b i c. Ukupna plata je onda:

PLATA = 100 x 40 + 100 x 10 + 50 x 10 = 5 500.



Dijagrami toka podataka: Ovi dijagrami predstavljaju algoritme kao "crne kutije", tako da ne možete da vidite šta se unutra dešava. Prikazani su samo ulazni i izlazni podaci, kao što se može videti na sledećoj slici. U ovom slučaju podatak (50 sati) ulazi u pravougaonik za izračunavanje plate, iz kojeg na dnu izlazi izračunata plata (5500). Ovi dijgrami sakrivaju detalje algoritma, ali su korisni kada treba opisati interakciju i tok podataka između algoritama.

Dijagrami toka podataka pokazuju šta se dešava, dok dijagrami kontrole toka pokazuju kako se to dešava.



Pseudokod: Pseudokod predstavlja mešavinu matematike, logike i prirodnog jezika. Algoritam iz prethodnih primera bi se mogao izraziti ovako:

```
Input brojRadnihSati
If brojRadnihSati > 40
   Set Plata to Nadnica x 40 + 1.5 x Nadnica x (brojRadnihSati - 40)
Else
   Set Plata to Nadnica x 40
```

Analiza algoritama

Ako rešavanje nekog problema poverite različitim ljudima, verovatno ćete dobiti onoliko različitih rešenja koliko ljudi to bude radilo. Ako se to prenese na oblast programiranja, verovatno će svako imati svoj algoritam. Postavlja se pitanje kako izabrati pravi algoritam. Koje su to mere koje ukazuju da je jedan algoritam bolji od drugog.

Ako su u pitanju algoritmi koji su poznati u literaturi, onda se u literaturi mogu naći i podaci o tome koji je od tih algoritama najbolji. Sa druge strane, ako su u pitanju novi problemi, odnosno novi algoritmi, treba na umu imati neke osnovne napomene.

Problemi koji se rešavaju obično imaju svoju "prirodnu veličinu". To je najčešće količina podataka koji treba da se obrade. Ova veličina se obično označava sa N. Kod analize algoritama bismo voleli da znamo resurse koji su korišćeni (najčešće vreme koje je proteklo) u funkciji od N.

Često smo kod analize zainteresovani za srednje vreme koje bi proteklo dok program analizira podatke. To srednje vreme je sredina vremena potrebnog za obradu tipičnih ulaznih podataka i vremena potrebnog za obradu najgore kombinacije ulaznih podataka.

Već smo pomenuli da većina algoritama ima primarni parametar N, obično broj podataka koji se obrađuju, koji najznačajnije utiče na vreme izvršenja algoritma. Taj parametar može biti stepen polinoma, veličina datoteke koja se sortira ili pretražuje, broj čvorova u grafu itd. Većina osnovnih algoritama ima vreme izvršenja proporcionalno jednoj od sledećih funkcija:

- 1 - Većina instrukcija se u većini programa izvršava jednom ili samo nekoliko puta. Ako su sve instrukcije u programu takve, onda se kaže da je vreme izvršenja konstantno. Očigledno je da je to situacija kojoj treba težiti kod kreiranja algoritma.
- log N - Ako je vreme izvršenja programa logaritamska funkcija, onda program postaje pomalo sporiji, kako N raste. Ovakvo vreme izvršenja se obično javlja u programima kod kojih se veliki problem rešava razlaganjem na

manje, tako što se veličina deli nekim konstantnim brojem. Ta konstanta se može smatrati manjom od neke ogromne brojke. Osnova algoritma utiče na konstantu, ali ne toliko puno. Ako je N hiljadu logN je 3 (ako je osnova 10), a ako se N poveća na 1 000 000 logN se duplira. Kad god se N poveća na N^2 logN se duplira. Kad god se N duplira, logN se poveća za neku konstantu.

- N - Ovo se dešava kada je vreme izvršenja programa linearno. Ovo je generalno slučaj kada se određeno, kratko vreme, troši na obradu svakog ulaznog elementa. Ako je N milion onda je takvo i vreme izvršenja. Kad god se N duplira isto se dešava i sa vremenom izvršenja. Ovo je optimalno rešenje za algoritam koji treba da obradi N ulaza (ili proizvede N izlaza).
- N logN Ovakvo rešenje se javlja kod algoritama koji problem rešavaju podelom na manje probleme, pri čemu se ti potproblemi rešavaju nezavisno, a na kraju se ti rezultati kombinuju u jedan. Ovakvi algoritmi se označavaju sa "N log N". Ako je N milion, onda je N log N možda 20 miliona. Kada se N duplira vreme izvršenja se poveća više od dva puta (ali ne mnogo više).
- N^2 - Ako je vreme izvršenja algoritma kvadratno, onda je on pogodan samo za rešavanje relativno malih problema. Kvadratno vreme izvršenja se javlja kod algoritama koji obrađuju sve parove ulaznih podataka (možda u dvostruko ugnežđenoj petlji). Ako je N hiljadu, onda je vreme izvršenja milion. Kad god se N duplira vreme izvršenja se učetvorostruči.
- N^3 - Ovaj algoritam je sličan sa prethodnim. Razlika je u tome da se obrađuju trojke podataka (trostruko ugnežđenoj petlji). Vreme izvršenja je kubno. Ovaj algoritam je takođe praktičan samo za male probleme. Ako je N hiljadu vreme izvršenja je milijarda. Kad god se N duplira, vreme izvršenja se poveća osam puta.
- 2^N - Ponekad se koriste i algoritmi sa eksponencijalnim vremenom izvršenja. Ako je N dvadeset, vreme izvršenja je milion. Kad god se N poveća dva puta, vreme izvršenja poraste na kvadrat.

Aritmetički algoritmi

Algoritmi za obavljanje elementarnih matematičkih operacija, kao što su sabiranje, množenje i deljenje imaju dugačku istoriju i datiraju još od arapskog matematičara al-Khowdirzmija, pa se čak ide i dalje u prošlost, do starih grka i vavilonaca.

Računari imaju ugrađene mogućnosti za aritmetičke operacije nad celim i realnim brojevima, tako da tu i nema svrhe posezati za algoritmima. Algoritmi dolaze u obzir kada se ove operacije obavljaju nad složenijim matematičkim objektima, kao što su polinomi ili matrice.

Polinomi

Pretpostavimo da želimo da napišemo program koji sabira dva polinoma, odnosno želimo da se obavi izračunavanje tipa:

$$(1+ 2x - 3x^3) + (2 -x) = 3 + x - 3x^3$$

Uopšteno se može reći da želimo da napravimo program koji će biti u stanju da izračuna r(x) = p(x) + q(x), gde su p i q polinomi sa N koeficijenata. Sledeći program je jednostavna implementacija sabiranja polinoma:

```
public double[] sabiranje(double[] p, double[] q){
    double[] r;
    int duzina = p.length;
    r = new double[duzina];
    for(int i = 0; i < duzina; i++){
        r[i] = p[i] + q[i];
    }
    return r;
}
```

U ovom programu se polinom $p(x) = p_0 + p_1x + ... + p_{n-1}x^{N-1}$ predstavlja preko niza p, sa N elemenata, gde je $p[j] = p_j$, itd. Polinom N-1 -og stepena se definiše sa N koeficijenata. Ulaz u metod su nizovi p i q. Izlaz je niz r, koji predstavlja zbir ovih polinoma.

Program za sabiranje polinoma je trivijalan, kad se jednom izabere način predstavljanja polinoma. Same operacije se posle lako kodiraju.

Ovaj program i algoritam je lako preurediti tako da se ostvari množenje polinoma. Dovoljno je promeniti petlju koja obavlja sabiranje:

```
    public double[] mnozenje(double[] p, double[] q){
    double[] r;
    int duzina = p.length;
    r = new double[2*duzina - 1];
    for(int i = 0; i < duzina; i++){
        for(int j = 0; j < duzina; j++){
            r[i+j] = r[i+j] + p[i] * q[j];
        }
    }
    return r;
}
```

Naravno da se mora promeniti i deklaracija za r, tako da ovaj niz može da primi dva puta više koeficijenata. Svaki od N koeficijenata iz p se množi svakim od N koeficijenata iz q. Jasno je da je ovo kvadratni algoritam.

Matrice

Algoritmi za obavljanje osnovnih aritmetičkih operacija nad matricama su slični sa onima za polinome, mada malo komplikovaniji. Pretpostavimo da želimo da nađemo sumu dve matrice:

$$\begin{pmatrix} 1 & 3 & -4 \\ 1 & 1 & -2 \\ -1 & -2 & 5 \end{pmatrix} + \begin{pmatrix} 8 & 3 & 0 \\ 3 & 10 & 2 \\ 0 & 2 & 6 \end{pmatrix} = \begin{pmatrix} 9 & 6 & -4 \\ 4 & 11 & 0 \\ -1 & 0 & 11 \end{pmatrix}.$$

Program izgleda ovako:

```
public int[][]sabiranjeMatrica(int[][] p, int[][] q){
    int[][] r = new int[p.length][p.length];
    for(int i = 0; i < p.length; i++){
        for(int j = 0; j < p[i].length; j++){
            r[i][j] = p[i][j] + q[i][j];
        }
    }
    return r;
}
```

Množenje matrica je komplikovanija operacija. U ovom slučaju je potrebno sabrati proizvode odgovarajućih elemenata. To znači da se element sa indeksom ij dobija sabiranjem proizvoda i-te vrste matrice p sa j-om kolonom matrice q, odnosno p[i, 1]*q[1, j]+p[i, 2]*q[2, j]+... p[i, N-l]*q[N-1, j]. Program je:

```
public int[][]mnozenjeMatrica(int[][] p, int[][] q){
    int[][] r = new int[p.length][q[0].length];
    for(int i = 0; i < p.length; i++){
        for(int j = 0; j < p[i].length; j++){
            int t = 0;
            for(int k = 0; k < p.length; k++){
               t = t + p[i][k]*q[k][j];
            }
            r[i][j] = t;
        }
    }
    return r;
}
```

Svaki od N^2 elemenata u rezultujućoj matrici se dobija preko N množenja, tako da je za množenje dveju matrica dimenzija N potrebno oko N^3 operacija. Ovo nije baš pravi kubni algoritam, pošto je broj podataka N^2.

Sortiranje

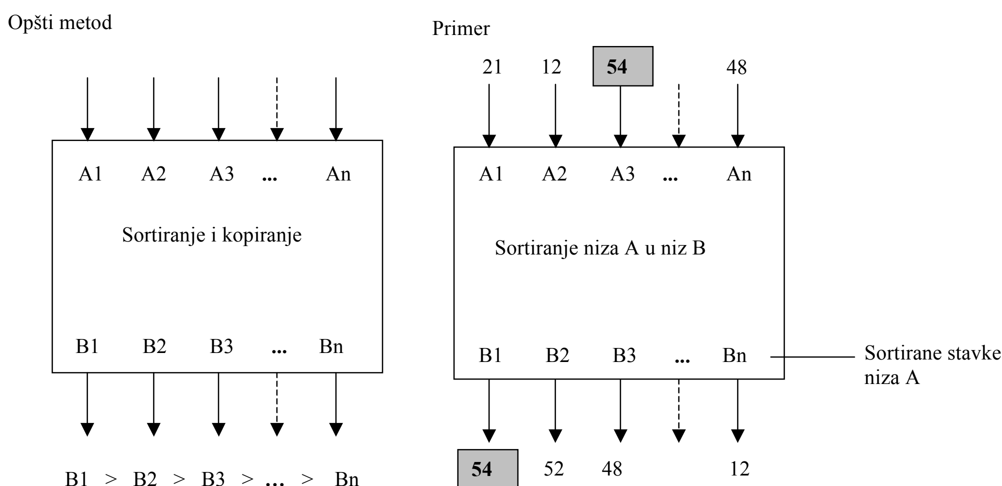
Sortiranje je proces koji se izuzetno često javlja u računarskim programima. Sortiranje je proces uređivanja stavki koje čine neki skup, prema nekom kriterijumu (numeričkom, po abecedi). Stavke koje se sortiraju mogu biti male, poput brojeva, slova ili nizova karaktera, ali i veliki zapisi, poput zapisa iz bibliotečkih kataloga, kada svaki zapis sadrži mnoštvo informacija o knjizi. Stavke se obično sortiraju prema rastućim ili opadajućim vrednstima nekog polja, koje se naziva ključem za sortiranje. Ovo polje može biti numeričko ili alfabetsko.

Obično se stavke koje se sortiraju nalaze u nizu.

Proces sortiranja se može odvijati sa tri različite strategije. Strategije ćemo objasniti na primeru brojeva koji se sortiraju po opadajućem nizu.

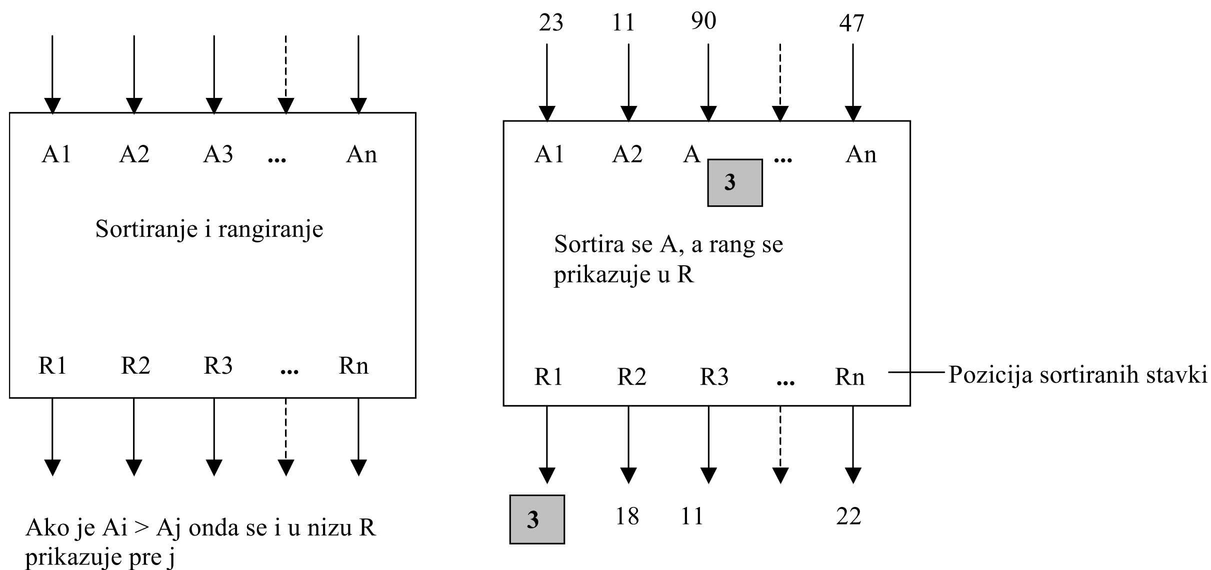
Sortiranje sa kopiranjem

Kod ovog metoda se sortiraju članovi niza A, koji se sa novim redosledom kopiraju u drugi niz B. Ovaj metod troši više memorijskog prostora, pošto je potreban još jedan niz iste veličine kao početni, u koji će se smestiti rezultat. Ovde je važno napomenuti da originalni niz posle procesa sortiranja ostaje nepromenjen.



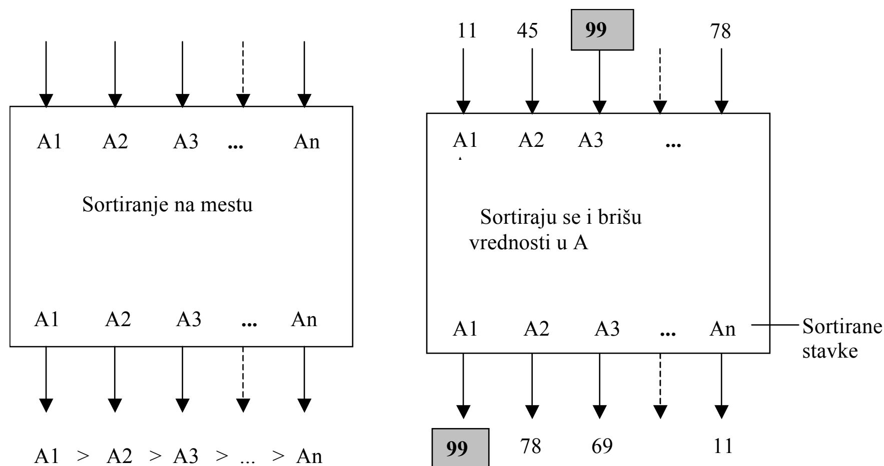
Sortiranje sa rangiranjem

Kod ovog sortiranja se generiše drugi niz R koji sadrži rang svake sortirane stavke. Rang predstavlja poziciju određene stavke u originalnom nizu. Ako pogledamo narednu sliku, videćemo da je najveća vrednost u nizu A, 90. To znači da prvi član niza R ukazuje na broj 90, tako što sadrži poziciju broja 90 u originalnom nizu A. U ovom slučaju to je 3. Kao i kod sortiranja sa kopiranjem i ovaj metod čuva originalnni niz i zahteva drugi niz. Taj drugi niz najčešće ne traži isti prostor kao originalni, pošto je u pitanju niz celih brojeva. Njegovi elementi nisu celi zapisi iz prvobitnog niza.



Sortiranje na licu mesta

Ovo sortiranje se ponekad naziva i sortiranje sa brisanjem, pošto se u ovom slučaju koristi samo jedan niz. Uređene vrednosti se posle sortiranja ponovo stavljaju u originalni niz A. Kod ovakvog sortiranja se postiže ušteda na prostoru, pošto je potreban samo jedan niz.



Metodi sortiranja

Kod svake od tri osnovne strategije soortiranja, koje smo pomenuli postoji više različitih algoritama za sortiranje. Svi ti algoritmi se mogu svrstati je četiri kategorije.

- Sortiranje sa prebrojavanjem, gde se dešava prebrojavanje i poređenje
- Sortiranje sa zamenom, gde parovi stavki menjaju mesta
- Sortiranje sa selekcijom, gde se biraju ekstremne vrednosti
- Sortiranje sa umetanjem, gde se dešava premeštanje i umetanje vrednosti

U daljem tekstu ćemo objasniti svaki od ovih metoda. Radi pojednostavljenja ćemo sortirati pozitivne cele brojeve, koji se međusobno razlikuju. Vrednosti treba da se postave u opadajući niz.

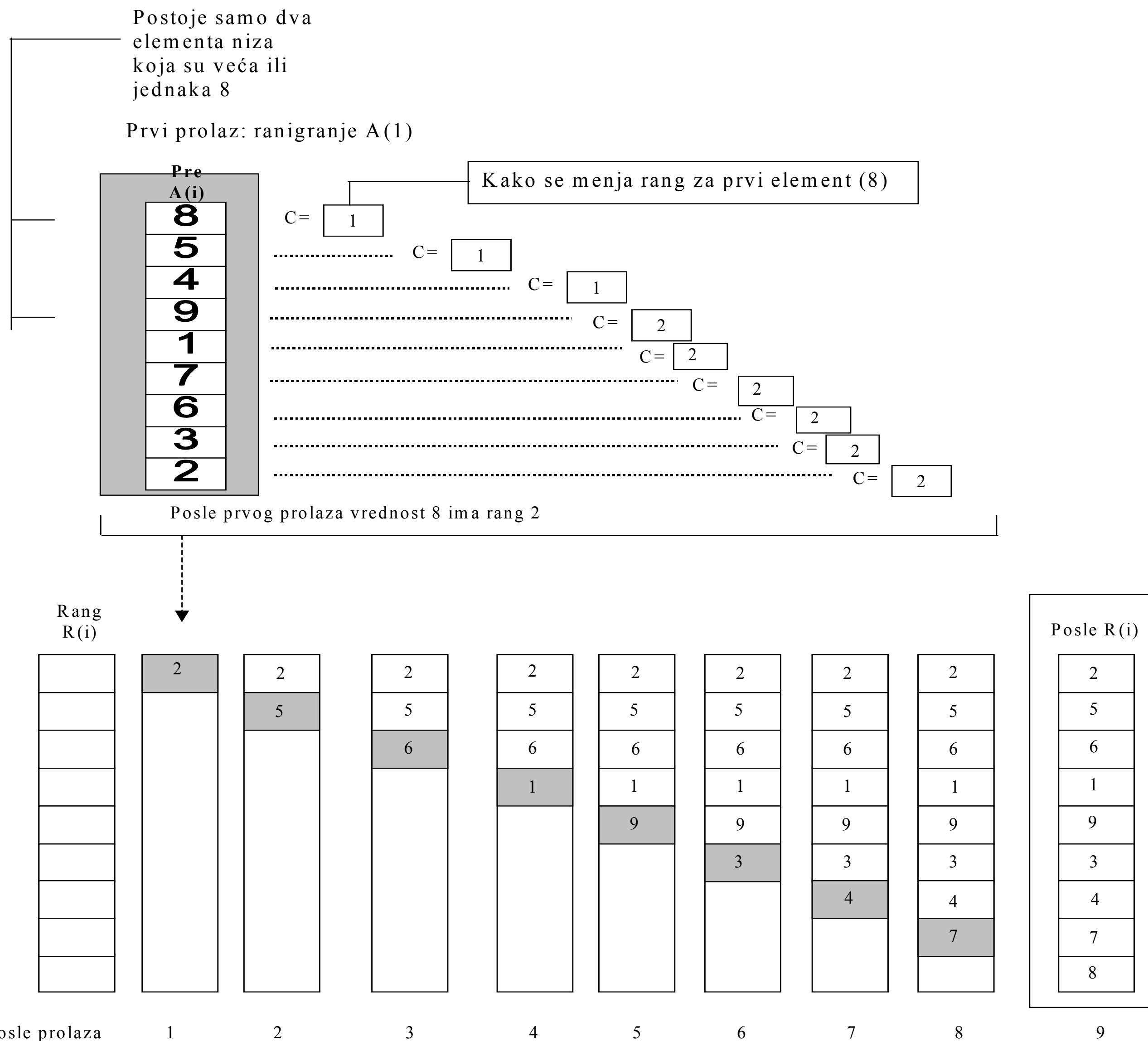
Ovi algoritmi se vrlo lako mogu modifikovati, ako želite da sortirate nizove karaktera, po redosledu u abecedi. Algoritme ćemo ilustrovati na nizu od 9 brojeva, od 1 do 9. Originalan niz je:

```
8, 5, 4, 9, 1, 7, 6, 3, 2
```

Sortiranje sa prebrojavanjem

Ovo je jedan od najjednostavnijih metoda sortiranja. Kod ovog metoda se pronalazi rang svih vrednosti u nizu. Najveća vrednost ima rang 1, dok najmanja ima rang N (ako su sve vrednosti različite).

Na gornjem levom delu naredne slike je prikazan originalan niz A, pre sortiranja. Rang neke vrednosti X se pronalazi poređenjem te vrednosti sa svim vrednostima u nizu od N elemenata, kao i brojanjem vrednosti koje su veće ili jednake od vrednosti X. Za najveću vrednost postoji samo jedna vrednost koja je veća ili jednaka od nje (to je ona sama), tako da je njen rang 1. Za drugu najveću vrednost postoje dve vrednosti koje su veće ili jednake od nje, tako da ona ima rang 2 itd. Za najmanju od N vrednosti postoji N vrednosti koje su veće ili jednake od nje, tako da ona ima rang N.



Prvi prolaz je prikazan u gornjem delu slike. U prvom prolazu se uzima vrednost 8 i poredi sa ostalim vrednostima u nizu (uključujući i nju samu). Svaki put kad se pronađe vrednost koja je veća ili jednaka od 8, brojač se povećava za 1.

1. U našem slučaju se uzima 8, poredi sa prvom vrednošću 8 i brojač dobija vrednost 1.
2. Kada se naiđe na 9 brojač dobija vrednost 2.

Ovim se određuje da prva vrednost 8 ima rang 2, pošto su samo dve vrednosti veće ili jednake sa njom. Na donjem delu slike je prikazan rezultat svakog prolaza. Ukupno ima 9 prolaza (N=9).

Sada možemo da počnemo sa razvojem algoritma. Proces ćemo početi odozgo nadole. Prvo ćemo definisati grubi okvir algoritma.

Pseudo kod ovog algoritma izgleda ovako:

```
CountSort(A, Rang)
   for prolaz=1 do brojStavki po 1
          |broji vrednosti veće ili jednake od A[prolaz]     Akcija za
          |podešava Rang[prolaz] na prebrojane vrednosti     svaki prolaz
kraj
```

Ako sada razložimo korake koje smo dali u prethodnoj petlji, to izgleda ovako:

```
CountSort(A, Rang)
  | for prolaz=1 do brojStavki po 1
  |        Vrednost se podešava na A[prolaz]
  |        Brojac se podešava na 0
  |        for indeks=1 do brojStavki po 1          Akcija za
  |             | if A[indeks] >= Vrednost          svaki prolaz
  |             |        povećaj Brojac za 1
  |        Podesi Rang[prolaz] na Brojac
Kraj
```

1. Tokom svakog prolaza se selektuje vrednost.
2. Ta vrednost se zatim poredi sa svim članovima niza. U promenljivu brojac se smesta broj elemenata koji su veći ili jednaki sa promenljivom vrednost.
3. brojac koji se dobije na kraju se smešta u niz Rang.

Obratite pažnju na to da su sve vrednosti u ovom primeru različite, tako da se dobijaju različite vrednosti za rang, koje se kreću od 1 do N. Ako bi se neke vrednosti ponavljale, onda bi se i neki rangovi ponavljal, dok neki rang ne bi imao odgovarajuću vrednost. Na primer, ako bismo u prethodnom primeru, promenili element 8 na 9, postojala bi dva elementa koja imaju rang 2, a nijedan ne bi imao rang 1.

Analiza algoritma

Kad god pravite algoritam trebalo bi da uradite dve stvari:

1. Da proverite da li radi
2. Da izvršite analizu da biste videli koliko dobro radi

Performanse algoritma se mogu meriti na više načina. Najčešće se analiza vrši na osnovu vremena izvršenja i prostora koji je potreban za izvršenje.

Vreme izvršenja je direktno povezano sa brojem operacija koje se moraju obaviti. U našem primeru se spoljašnja petlja ponavlja N puta, dok se u svakom od tih prolaza unitrašnja petlja ponavlja N puta. Ukupan broj prolaza je prema tome N x N. Pošto u našem primeru imamo 9 elemenata, to će biti 81 prolaz. Svaki prolaz znači jedno poređenje, tako da će biti ukupno 81 poređenje.

Ako izvršite dalju analizu zaključićete da dupliranje veličine niz (od 9 na 18) neće jednostavno duplirati broj poređenja, već će taj broj kvadratno porasti (od 81 na 324). Ovaj kvadratni rast rezultuje vrlo sporim algoritmom, posebno ako su u pitanju dugački nizovi.

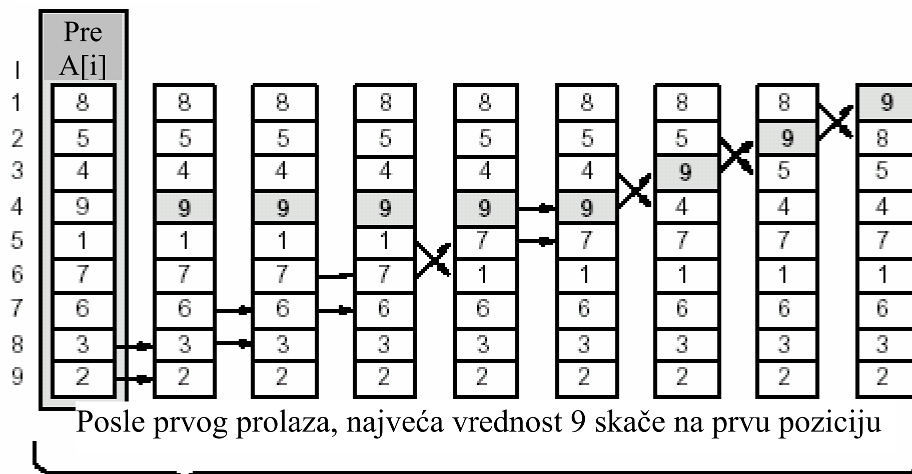
Na osnovu podele koju smo dali, može se zaključiti da ovaj algoritam pripada kategoriji kvadratnih algoritama.

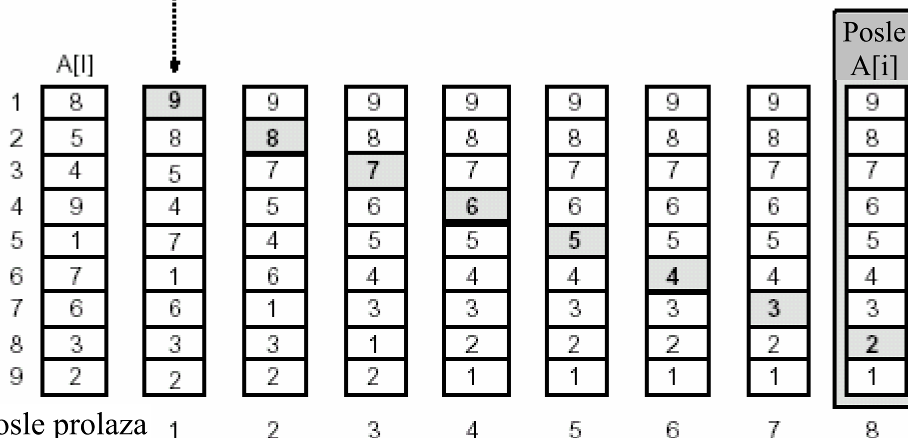
Sa stanovišta efikasne upotrebe memorijskog prostora, ovaj algoritam nije baš najefikasniji. Kao što ste videli potreban je drugi niz koji čuva rang koji se dodeljuje pojedinim elementima niza. U slučaju da originalni niz ima velike elemente (zapise), onda je u poređenju sa njim prostor potreban za dodatni niz mali, jer se u taj niz smeštaju celi brojevi. U našem slučaju je dodatni niz iste veličine, kao i početni, pošto oba niza sadrže cele brojeve.

Sortiranje sa zamenom

Kod ove familije algoritama postoji veliki broj varijacija na istu temu. Mi ćemo ovde predstaviti algoritam poznat kao mehurasto sortiranje (bubble sort).

Ovaj algoritam se sastoji od poređenja susednih vrednosti u nizu koji se sortira i zameni mesta, ako vrednosti nisu sređene. Na gornjem delu naredne slike je prikazan prvi prolaz kroz niz. U to prolazu se vrši poređenje svih susednih stavki. Nakon prvog prolaza se može reći da je niz malo više sortiran. Primetićete da je najveća vrednost (originalno na mestu 4) sada došla na prvu poziciju, gde i treba da bude.





U ovoj fazi su sve vrednosti već sortirane. Ostali prolazi su nepotrebni.

U donjem delu slike su prikazani rezultati svakog prolaza. Kao što se vidi posle prvog prolaza je najveća vrednost (9), došla na svoje mesto na početak niza. Nakon drugog prolaza je druga najveća vrednost (8) došla na drugu poziciju. Ovo se nastavlja sve dok se niz ne poređa.

Primetićete da je posle četvrtog prolaza niz već sortiran, ali se algoritam nastavlja. Razlog je taj što u je u najgorem scenariju potrebno N-1 prolaz, za sortiranje N vrednosti. Ovo možete proveriti na primeru niza koji je već uređen po rastućim vrednostima (A[9] = 9, A[8] = 8 itd).

Da pogledamo sada kako izgleda algoritam. Počećemo sa najopštijom verzijom:

```
BubleSort(A, veličina)
   for prolaz+1 do veličina-1 po 1
          najveću vrednost vadi na vrh
Kraj
```

Sada treba da razložimo akciju u petlji. Tu treba da prođemo kroz parove susednih vrednosti, da ih uporedimo i zamenimo im mesta ako je potrebno. Sledeća verzija izgleda ovako:

```
BubleSort(A, veličina)
   for prolaz=1 do veličina-1 po 1
        | for indeks=veličina-1 do po -1
        |   Poredi susedne vrednosti i menja im mesta ako je potrebno
Kraj
```

Kompletan algoritam dobijamo specifikacijom detalja poređenja i zamene.

```
BubleSort(A, veličina)
   for prolaz=1 do veličina-1 po 1
        | for indeks=veličina-1 do po -1
        |        | if A[indeks] < A[indeks+1)
        |        |         zameni A[indeks] i A[indeks+1]
Kraj
```

Analiza algoritma

Iz koda koji smo napravili može se videti da ovaj algoritam zahteva (N-1) prolaza (u našem slučaju 8). U svakom od tih prolaza se dešava ukupno (N-1) poređenja, što je ukupno (N-1) x (N-1) poređenja. U našem primeru imamo 9 elemenata koje želimo da sortiramo što znači da je potrebno 64 poređenja. Već sada je ovaj algoritm bolji od prethodnog sa prebrojavanjem, ali se on može dodatno poboljšati.

Postoji više načina da se poboljša ovaj algoritam. Najlakše poboljšanje se može dobiti ako se primeti da svaki prolaz može biti kraći za jedno poređenje. Razlog je što je na kraju svakog prolaza jedan element postavljen na svoju finalnu poziciju, tako da ga više ne treba uzimati u obzir. To znači da ćemo u prvom prolazu i dalje imati 8 poređenja, ali nam zato u prolazu 2 treba samo sedam poređenja, pošto je jedan element (9) već na svojoj finalnoj poziciji. U trećem prolazu nam treba šest poređenja itd. Ukupan broj poređenja u ovom primeru je:

$$C = (8 + 7 + 6 + 5 + 4 + 3 + 2 + 1) = 36$$

što je manje od polovine poređenja u prethodnom algoritmu sa prebrojavanjem.

Sada možemo generalizovati rezultat za N vrednosti. Broj poređenja za poboljšani algoritam sa zamenom je:

$$C = (N-1) + (N-2) + (N-3) + \ldots + 3 + 2 + 1$$

ili

$$C = \frac{N(N-1)}{2}$$

Da uporedimo sada novi poboljšani algoritam sa zamenom i prethodni sa prebrojavanjem. Kod sortiranja sa prebrojavanjem je bilo potrebno N^2 poređenja. Kod poboljšanog algoritma sa zamenom je potrebno N(N-1)/2 poređenja.

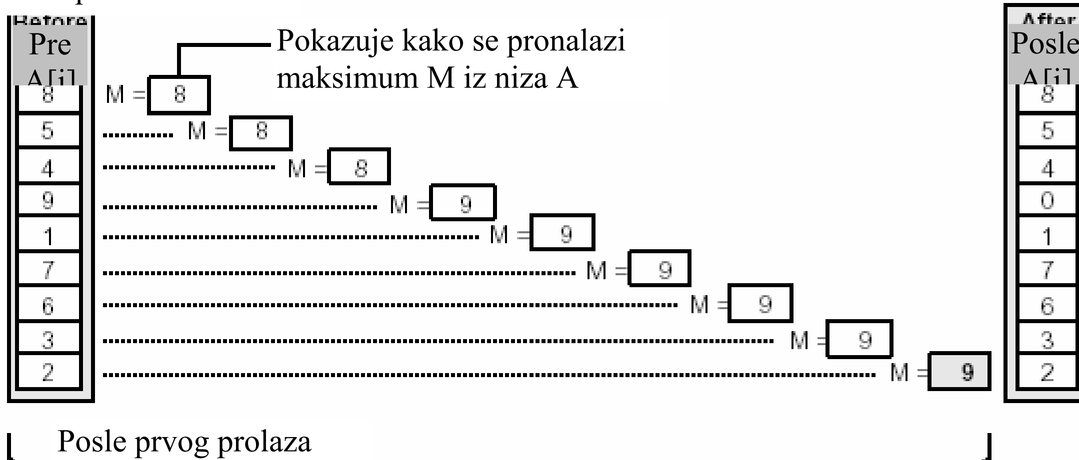
Ako sortiramo velike nizove, gde je N veoma vliki broj, možemo da primetimo sledeće:

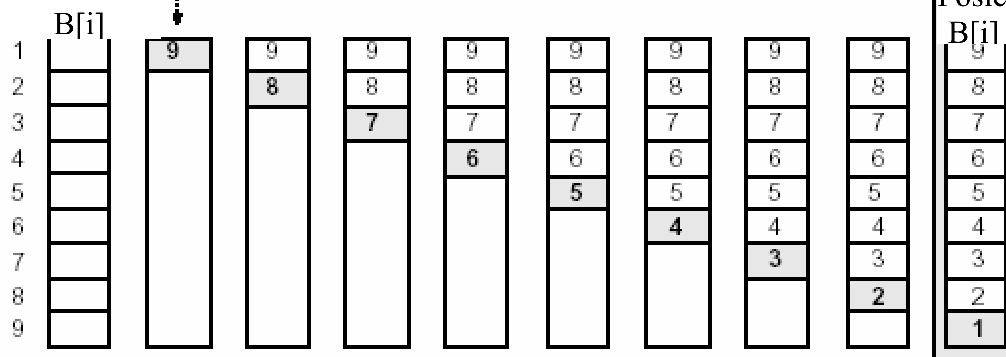
- Sortiranje sa prebrojavanjem traži N^2 poređenja.
- Poboljšani algoritam sa zamenom traži polovinu poređenja.

U skladu sa ovim možemo reći da je sortiranje sa zamenom skoro dva puta brže od sortiranja sa prebrojavanjem, ali da oba sortiranja pripadaju kvadratnim algoritmima.

Sortiranje sa selekcijom

Ovaj algoritam se zasniva na izboru ekstremnih vrednosti, bilo da je u pitanju minimalna, bilo maksimalna vrednost. Algoritam bi, na primer, mogao početi određivanjem maksimalne vrednosti u nizu koji se sortira. Ta vrednost se obeležava, stavlja u drugi niz i briše iz originalnog. Proces se zatim ponavlja na originalnom nizu. Ponovo se bira maksimalna vrednost, ona se zapisuje i izbacuje. Na taj način se originalni niz polako prazni, a drugi niz, rezultat, polako puni. Ciklus se ponavlja sve dok se svih N vrednosti iz niza ne zapišu.





Gornji deo prethodne slike prikazuje prvi prolaz kroz niz celih brojeva. Prva maksimalna vrednost koja se pronalazi je 9. To je izlaz i ona se smešta u drugi niz. Njena vrednost se zamenjuje nulom.

U donjem delu slike se vidi rezultujući niz B nakon svakog prolaza. Obratite pažnju na to da se prvobitni niz A polako uništava, odnosno postaje niz sa nulama.

U algoritmu ćemo imati N prolaza kroz glavnu petlju. Svaki prolaz se odnosi na pronalaženje jedne vrednosti.

Pseudo kod u najopštijem slučaju izgleda ovako:

```
SelectSort(Tabela, Veličina, Rezultat)
   for prolaz=1 do veličina po 1
     |     Podesi vrednost za Maksimum
     |     upotrebi Maksimum
     |     eliminiši Maksimum
Kraj
```

Sada možemo da definišemo detalje algoritma. Tokom svakog prolaza se pronalazi maksimalna vrednost, ona se smešta u drugi niz i onda eliminiše iz originalnog niza.

```
SelectSort(Tabela, Veličina, Rezultat)
   for prolaz=1 do veličina po 1
   |       Podesi Maksimum na 0                              ─┐
   |       for indeks=1 do veličina po 1                      │  Pronalaženje
   |       |       Uporedi Maksimum i Tabela[indeks]          ├─ maksimuma u A
   |       |       Ažuriraj Maksimum i Poziciju ako je potrebno│
   |       podesi Rezultat[prolaz] na Maksimum               ─┘
   |       Podesi Tabela[pozicija] na 0
Kraj
```

Sada možemo da napravimo završni oblik algoritma. Promenljiva maksimum se poredi sa svakom vrednošću u nizu i ažurira u skladu sa pozicijom.

```
SelectSort(Tabela, Veličina, Rezultat)
   for prolaz=1 do veličina po 1
   |       Podesi Maksimum na 0                              ─┐
   |       for indeks=1 do Veličina po 1                      │
   |       |       if Maksimum < Tabela[indeks]               │  Pronalaženje
   |       |       |       Podesi maksimum na Tabela[indeks]  ├─ maksimuma u A
   |       |       |       Podesi poziciju na indeks         ─┘
   |       podesi Rezultat[prolaz] na Maksimum
   |       Podesi Tabela[pozicija] na 0
Kraj
```

Sortiranje sa zamenom koje smo ovde pokazali radi samo sa pozitivnim celim brojevima, pošto se izbačene vrednosti zamenjuju nulom. Ako podaci sadrže i negativne vrednosti, morali bismo da modifikujemo algoritam, tako da obrisanim elementima dodeljuje najmanju negativnu vrednost (iz niza).

Analiza algoritma

Analizom ovog algoritma možemo zaključiti da postoji N prolaza i da se u svakom prolazu ispituje N elemenata iz tabele. U skladu sa tim postoji N^2 poređenja, što je slično sa prvim algoritmom za sortiranje koji smo radili (sortiranje sa prebrojavanjem). Kod ove verzije za čuvanje sortiranih vrednosti koristi drugi niz, što je rešenje koje nije ekonomično u pogledu memorijskog prostora.

Ovaj algoritam se može poboljšati na nekoliko načina:

Prvo bismo mogli da incijalnu vrednost promenljive maksimum podesimo na prvu vrednost iz niza i tako broj poređenja smanjima na N-1.

Mogli bismo i da iskombinujemo selekciju sa zamenom, kao što smo pokazali u narednom primeru. Umesto da prvi pronađeni maksimum smeštamo u drugi niz, možemo ga zameniti sa prvim elementom u nizu. Nakon toga se pronalazi maksimum ostatka niza i menja sa drugim elementom niza. Ovo se nastavlja sve dok se ne sortira ceo niz.

Ovakvo sortiranje se obavlja samo u originalnom nizu. Tokom sortiranja možemo smatrati da je taj niz podeljen na dva dela. Jedan deo je sortiran i on je inicijalno prazan, a drugi je niz koji nije sortiran. Sortirani deo raste, sve dok ne obuhvati ceo niz.

Sortiranje sa umetanjem

Kod ovog metoda se takođe jedan po jedan element iz originalnog niza A prebacuje u sortirani niz B. Pri tome se postojeći elementi u sortiranom nizu B pomeraju da bi se napravilo mesto za novi element. Ovo je slično sa ređanjem karata u preferansu ili bridžu. Igrači karte najčešće slažu prema bojama i po veličini. Određena karta se prebacuje na njeno mesto u nizu, a ostale se pomeraju za po jedno mesto.

Peti prolaz

Početak B[i]				Kraj B[i]
9	9	9	9	9
8	8	8	8	8
5	5	5	5	7
4	4	4	5	5
1	1	4	4	4
0	1	1	1	1
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

posle petog prolaza

prolaz	1	2	3	4	5	6	7	8
	8	8	9	9	9	9	9	9
	5	5	8	8	8	8	8	8
	0	4	5	5	7	7	7	7
	0	0	4	4	5	6	6	6
	0	0	0	1	4	5	5	5
	0	0	0	0	1	4	4	4
	0	0	0	0	0	1	3	3
	0	0	0	0	0	0	1	2
	0	0	0	0	0	0	0	1

Kod za ovaj algoritam (u Javi) izgleda ovako:

```
public void insertionSort(int[]a,int[]b)   {
   int in, out;
   b[0] = a[0];
   for(out=1; out<nElems; out++)  {
     long temp = a[out];       // proverava se ovaj element
     in = out;            // pomeranje počinje od elementa out
     while(in>0 && b[in-1] <= temp){ // sve dok je neki manji
      b[in] = b[in-1];       // element se pomera udeso
      --in;             // ide se jednu poziciju u levo
     }
     b[in] = temp;          // označeni element se ubacuje
  }
}
```

Analiza algoritma

U prvom prolazu ovog algoritma se poredi maksimalno jedan element. U drugom prolazu se porede maksimalno dva elementa itd., sve do maksimalnog broja poređenja N-1 u poslednjem prolazu. Ukupno je to

```
1 + 2 + 3 + ... + N-1 = N*(N-1)/2
```

Pošto se u svakom prolazu ipak ne porede svi elementi, odnosno nemamo maksimalan broj poređenja, to se može uzeti srednja vrednost, odnosno maksimalan broj poređenja se može podeliti sa 2, što daje:

```
N*(N-1)/4
```

Pored poređenja imamo i kopiranje. Broj kopiranja je približno isti kao broj poređenja. Sa druge strane proces kopiranja ne uzima toliko vremena kao zamena, tako da je ovaj algoritam brži u odnosu na sortiranje sa zamenom.

U svakom slučaju, kao i prethodni algoritmi za sortiranje i ovaj algoritam pripada klasi kvadratnih algoritama.

Sortiranje sa umetanjem mnogo bolje radi ako su u pitanju podaci koji su već sortirani ili delimično sortirani. Ako su podaci uređeni, onda uslov u while petlji nikad nije tačan, tako da se sve svodi na spoljašnju petlju, koja se izvršava N-1 puta. U ovom slučaju ovo postaje algoritam sa vremenom izvršenja N. Ako su podaci skoro sortirani, onda je i ovo algoritam sa vremenom izvršenja skoro N.

Sa druge strane, ako su podaci uređeni po inverznom redosledu, onda se vrši svako moguće poređenje i kopiranje, tako da se ovaj algoritam izvršava slično kao prethodni.

Iz ove analize se može zaključiti da sortiranje sa umetanjem treba koristiti kod problema kod kojih su elementi već delimično uređeni.

Algoritam Quicksort (brzo sortiranje)

Ovo je algoritam koji se u praksi verovatno najviše koristi. Osnovni algoritam je 1960 godine razvio Hoare, a od tada je on doživeo mnoge implementacije i poboljšanja. Ovaj algoritam je popularan zato šti nije težak za implementaciju, radi generalno sortiranje (radi dobro u različitim situacijama) i zauzima manje resursa nego mnogi drugi metodi sortiranja.

Dobre osobine ovog algoritma su da se sortiranje vrši na licu mesta (koristi se samo mali pomoćni stek), da zahteva NlogN operacija. Ovo vreme je srednje vreme potrebno za sortiranje N elemenata.

Nedostaci ovog algoritma su da je rekurzivan (implementacija postaje komplikovana ako se ne može da koristi rekurzija), da postoji najgori slučaj kada je potrebno N^2 operacija, kao i da je vrlo krhak (mala greška u implementaciji može ostati neprimećena i može dovesti da se u nekim slučajevima algoritam ne ponaša kako treba).

Ovaj algoritam radi na principu podele niza. Niz koji se sortira se deli na dva dela, a onda se svaki deo nezavisno sortira. Tačna podela zavisi od inicijalnog rasporeda elemenata u nizu.

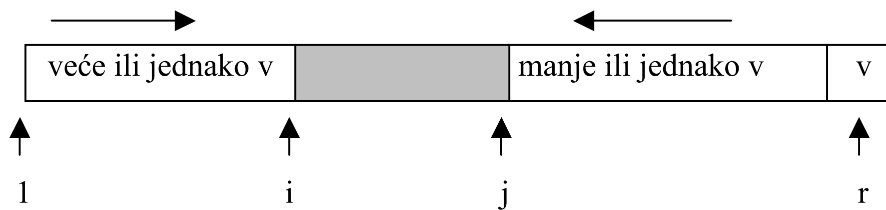
Globalni program izgleda ovako:

```
static void quicksort(int[] a, int lev, int des){
    if (des <= lev) return;
    int i = podela(a, lev, des);
    quicksort(a, lev, i-1);
    quicksort(a, i+1, des);
}
```

Ako niz ima jedan ili nijedan element ne dešava se ništa. Ako niz ima više elemenata, niz se obrađuje pomoću procedure podela. Ova procedura postavlja element a[i] na poziciju između lev i des (uključujući i njih), a ostalim elementima menja mesta u skladu sa sortiranjem i njihovim vrednostima.

Podela se implementira na sledeći način. Prvo se proizvoljno izabere element a[r], koji će podeliti niz. To je element koji će se postaviti na svoju krajnju poziciju. Nakon toga se pretražuje niz, počevši od levog kraja sve dok se ne nađe element koji je veći od elementa podele. Niz se pretražuje i sa desne strane sve dok se ne nađe element koji je manji od elementa podele. Elementi koji su zaustavili pretraživanje očigledno nisu na mestima na kojima bi trebalo da se nađu u sortiranom nizu. Zbog toga se njima menjaju mesta. Proces se nastavlja i dalje, čime se osigurava da levo od levog indeksa nema elemenata koji su manji od elementa podele, a desno od desnog indeksa nema elemenata koji su veći od elementa podele.

Ovo se može predstaviti sledećim dijagramom:



Na dijagramu v predstavlja vrednost elementa podele, i je levi indeks, a j je desni indeks. Kao što se vidi sa dijagrama, najbolje je pretražvanej zaustaviti kada su elementi veći ili jednaki ili manji ili jdenaki vrednosti v. Kada se indeksi i i j ukrste, pretraživanje se prekida, a a[r] menja mesto sa krajnjim levim elementom desnog dela (element na koji ukazuje levi indeks i).

Procedura podele izgleda ovako:

```
private static int podela(int[] a, int lev, int des) {
    int i = lev - 1;
    int j = des;
    while(true) {
        while (a[++i] > a[des]) // trazi se element sa leve strane
                                 //kojem se menja mesto
            ;                    // a[des] je uslov petlje
        while (a[des] > a[--j])   trazi se element sa desne strane
                                 //kojem se menja mesto
            if (j == lev) break;     // ne prelaziti granice
        if (i >= j) break;     // provera da li se indeksi ukrštaju

        int swap = a[i];
        a[i] = a[j];      // dva elementa menjaju mesta
        a[j] = swap;
    }
    int swap = a[i];
    a[i] = a[des]; // element podele menja mesto
    a[des] = swap;
    return i;
}
```

Prva while petlja je implementirana kao beskonačna petlja. Izlaz iz ove petlje se ostvaruje preko iskaza break, koji se poziva ako su se indeksi ukrstili. Promenljiva i služi za brojanje sa leve strane i prva unutrašnja while petlja traži prvu manju vrednost od početka. Promenljiva j služi za brojanje sa desne strane, a druga unutrašnja while petlja traži prvu veću vrednost od kraja. U svakom prolazu se pronađenim elementima menjaju mesta.

Kada se indeksi ukrste podela se završava zamenom elementa koji je služio za podelu sa krajnjim levim elementom, desnog dela (menjaju mesta a[des] i a[i]).

Analiza algoritma

Osnovni quicksort algoritam ima ograničene mogućnosti. Za neka sortiranja koja se mogu javiti u praksi on može postati prilično neefikasan. Na primer, ako se primeni na niz veličine N, koji je već sortiran, onda će se desiti N poziva programa, pri čemu se u svakom prolazu briše samo po jedan element.

Ovaj algoritam u najgorem slučaju koristi $N^2/2$ poređenja. To je slučaj kada je niz već sortiran. Ovakvo ponašanje znači da ovaj algoritam ne samo da traži više vremena u ovom slučaju, već raste i memorijski prostor potreban za rukovanje rekurzijom, što kod velikih nizova može biti problem.

Najbolji slučaj kod ovog algoritma je kada svaka faza podele podeli niz tačno na pola. Na taj način bi se dobio broj poređenja od približno N log N.

Generalno se može reći da u najvećem broju slučajeva algoritam quicksort koristi 2 N log N poređenja.