UNIVERZITET METROPOLITAN BEOGRAD

CS203 - ALGORITMI I STRUKTURE PODATAKA

Uvod u algoritme

Lekcija 01

PRIRUČNIK ZA STUDENTE

CS203 - ALGORITMI I STRUKTURE PODATAKA

Lekcija 01

UVOD U ALGORITME

UVOD

Stvarne performanse bilo kog softverskog sistema zavise od: izabranog algoritama i od prikladnosti i efikasnosti njihove implementacije

Algoritmi su osnova za računarstvo i softversko inženjerstvo. Stvarne performanse bilo kog softverskog sistema zavise od: (1) izabranih algoritama i (2) njihove prikladne i efikasne implementacije. Dobro projektovan algoritam je stoga presudan za performanse svih softverskih sistema. Štaviše, proučavanje algoritama pruža uvid u suštinu problema i moguće tehnike rešavanja problema koje su nezavisne od programskog jezika, programske paradigme, računarskog hardvera ili bilo kog drugog aspekta implementacije.

Važan deo računarstva je sposobnost odabira algoritama koji odgovaraju određenoj svrsi i sposobnost njihove primene, ali i razumevanje da za određeni problem može i da ne postoji algoritam koji je dobar i efikasan. Ovo je naravno moguće, ali to proizilazi iz poznavanja mogućnosti i slabosti odgovarajućih algoritama, i pogodnosti primene u odgovarajućim situacijama. Zato je proučavanje efikasnosti algoritama široko rasprostranjena tema u ovom području.

U cilju korišćenja računara za efikasno rešavanje problema, studenti treba da budu sposobni da čitaju i pišu programe različitih programskih jezika. Pored programerskih veština, studenti takođe treba da budu sposobni da pravilno projektuju i analiziraju algoritam, izaberu adekvatne primere (i da upotrebe moderne alate za razvoj i testiranje kompjuterskih programa. Ovaj predmet ima za cilj da studentima približi projektovanje i analizu algoritama, osnovne programerske koncepte i strukture podataka.

Poglavlje 1

Osnovi o algoritmima

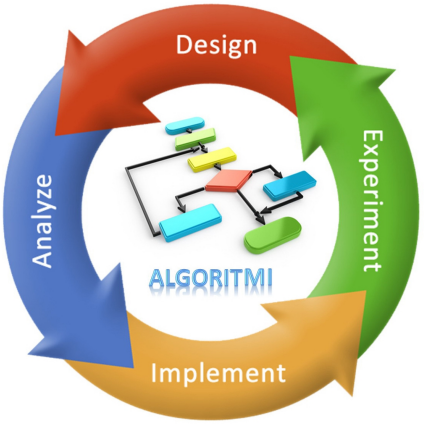
DEFINICIJA ALGORITMA

Algoritam je opis za rešavanje nekog problema. To je konačna i precizno definisana procedura (niz dobro definisanih pravila) kojom se ulazne vrednosti transformišu u izlazne

Algoritam je konačna i precizno definisana procedura, tj. uputstvo, kojom se opisuje izvršavanje nekog postupka, zadatka ili problema.

Tako se i uputstvo za slanje čoveka na Mesec i uputstvo za pravljenje ruske salate sastoji od niza koraka, postupaka, koje treba uraditi i koji vode ispunjenju cilja ili rešavanju problema. Uobičajeni koraci za projektovanje pravilnog algoritma su prikazani na Slici 1.1.

Različiti algoritmi mogu rešiti isti problem različitim nizom postupaka uz manje ili više napora, za kraće ili duže vreme. S obzirom da je rezultat isti, algoritmi se mogu porediti po svojoj efikasnosti, brzini ili složenosti.



Koraci u razvoju algoritma:

1. Projektovanje (Design)
2. Analiza (Analyze)
3. Implementacija (Implement)
4. Provera (Experiment)

Slika 1.1 Koraci u konstrukciji pravilnog algoritma: Projektovanje, analiza, implementacija i provera [9]

PREDSTAVLJANJE ALGORITAMA

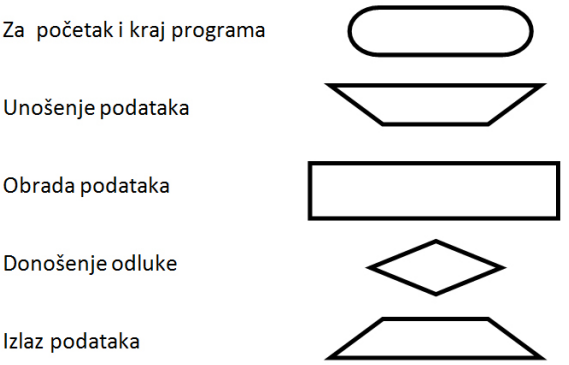
Algoritmi se obično predstavljaju na dva načina: grafički i pomoću pseudokoda

Algoritam se mogu predstaviti na različite načine i to najčešće:

- prirodnim jezikom (razumljiv samo govornicima tog jezika)
- grafički, dijagramom toka (blok-dijagramom) ili strukturnim dijagramima ili

- tekstualno - pseudokodom, veštačkim precizno definisanim jezikom koji liči na programski jezik
- odgovarajućim programskim jezikom.

Na Slici 1.2 su prikazani osnovni elementi grafički predstavljenih algoritama.



Slika 1.2 Osnovni elementi za predstavljanje algoritama [izvor: autor]

Pseudokod (engl. pseudocode) je tekstualna reprezentacija algoritma koja aproksimira konačni izvorni kod. U nastakvu je jedan primer:

```
DECLARE CHARACTER ch
DECLARE INTEGER count = 0
DO
  READ ch
  IF ch IS '0' THROUGH '9' THEN
    count++
  END IF
UNTIL ch IS '\n'
PRINT count
END
```

U okviru ovog predmeta ćemo algoritme najčešće predstavljati u programskom jeziku Java, ali neretko ćemo koristiti i pseudo kodove.

Poglavlje 2

Algoritamske strategije

OSNOVNE ALGORITAMSKE STRATEGIJE

Algoritmi se obično razvrstavaju prema metodologiji projektovanja ili primenjenom obrascu. Ovom problematikom se bavi oblast koja se zove algoritamske strategije

Jedan način razvrstavanja algoritama je po metodologiji projektovanja ili primenjenom obrascu. Ovo problematikom se bavi oblast koja se naziva ***algoritamske strategije*** .

Postoji više algoritamskih strategija koje se primenjuju kada je potrebno razviti algoritam za neki problem. Ipak, ne postoje garancije da će neka strategija dati tačno rešenje (a iako je rešenje tačno ne mora biti efikasno). Razlog za primenu je taj što su se ove strategije u prošlosti pokazale uspešnim za rešavanje mnogih problema pa je verovatno da će neka od njih dati dovoljno dobro rešenje. Najpoznatije strategije su:

- ***Gruba sila*** (brute-force) ili iscrpna pretraga
- ***"Podeli i osvoji"*** algoritmi (divide and conquer).
- ***Pohlepni metod*** (greedy method).
- ***Dinamičko programiranje***
- ***Bektreking*** ili ***obrnuta pretraga*** - iscrpna pretraga rešenja uz ispitivanje svih puteva.
- ***Preobrazi i osvoji*** (Transform and Conquer)
- ***Heuristike***
- ***Redukcija*** ili svođenje problema na postojeće.

ISCPNA PRETRAGA

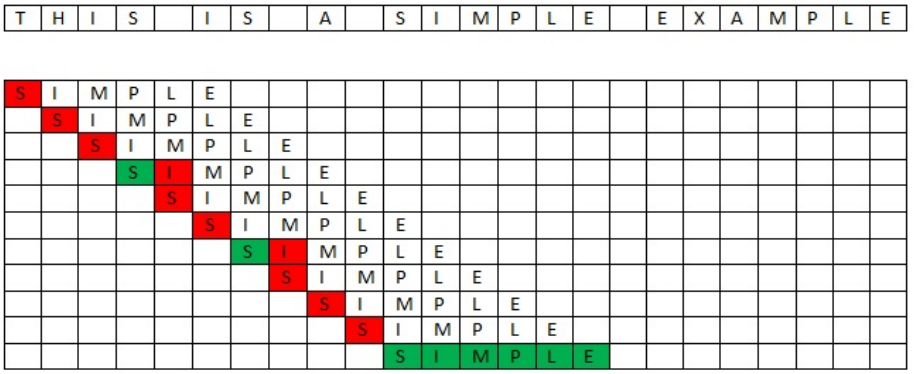
Iscrpna pretraga je opšta tehnika rešavanja problema koja se sastoji od sistematičnog nabrajanja svih kandidata kao mogućih rešenja i provere da li svaki zadovoljava rešenje

Brute-force **pretraga (gruba sila)** ili ***iscrpna pretraga*** (brute force search), je tehnika rešavanja problema koja se sastoji od sistematičnog nabrajanja svih mogućih kandidata kao potencijalnih rešenja i provere da li svaki kandidat zadovoljava rešenje problema.

Primer 1: Da bi algoritam iscrpne pretrage pronašao delioce prirodnog broja *n*, on prolazi kroz sve cele brojeve od *1* do *n/2*, i proverava da li svaki od njih deli *n* bez ostatka.

Dok je iscrpna pretraga jednostavna za primenu, i uvek će pronaći rešenje ako postoji, njena cena srazmerna je broju kandidata rešenja – što u mnogim praktičnim problemima nije primenljivo. Ova metoda se takođe koristi kada je jednostavnost implementacije važnija od brzine. Iscrpna pretraga je takođe korisna i kao ***"osnovni"*** metod kod testiranja drugih algoritama.

Primer 2: Pretpostavimo da želimo da ispitamo da li se u stringu dužine *N* nalazi odgovarajući podstring dužine *M*. Algoritam iscrpne pretrage bi se sastojao iz provere svih pozicija teksta od *0* do *n-m*, sa ciljem da se ispita da li podstring počinje od tog mesta ili ne. Zatim, nakon svakog neuspešnog pokušaja poklapanja šablona, bi se pomerili na sledeće mesto u stringu u desno i ponovili isti postupak ispitivanja (Slika 2.1).



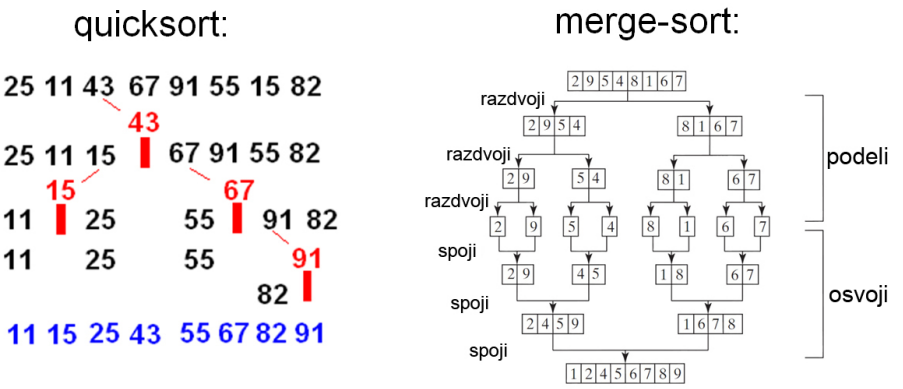
Slika 2.1.1 Primer upotrebe algoritma iscrpne pretraga za pronalaženje podstringa u datom stringu [10]

PODELI I OSVOJI“ ALGORITAM

Podeli i osvoji algoritam smanjuje stepen složenosti problema podelom na dva ili više manjih problema iste vrste dok od problema ne ostane toliko mali deo da se može jednostavno rešiti

Podeli i osvoji (lat. divide et impera*. ili* ***„zavadi pa vladaj“***) je algoritam u kojem se veći problem deli na više manjih problema koji su jednostavniji za sagledavanje i pojedinačno rešavanje. Podela na manje se vrši dok potproblemi ne postanu dovoljno jednostavni da se mogu direktno rešiti. Rešenja potproblema se kombinuju da bi se dobilo rešenje početnog problema.

Primeri ***podeli-i-osvoji*** algoritama su kviksort (quicksort), sortiranje spajanjem (merge sort), Euklidov algoritam, binarna pretraga itd. Primer rada dva algoritma sortiranja koja koriste ovu strategiju je prikazan na Slici 2.2.



Slika 2.1.2 Primeri "podeli i osvoji" algoritma [2]

U cilju razumevanja osnovnih koncepata ***"podeli i osvoji"*** algoritma možete pogledati sledeći video.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

POHLEPNI ALGORITAM

Pohlepni algoritam u svakom koraku bira u tom trenutku naizgled najbolje rešenje, bez obzira da li će to uticati na tačnost konačnog rešenja problema

Pohlepni algoritam (engl.greedy algorithm) u svakom koraku bira u tom trenutku naizgled najbolje rešenje. Redom pravimo jedan pohlepni izbor za drugim, i svodimo svaki dati problem na manji problem. Pogledajmo sledeći primer.

Problem kusura

Trgovac treba da vrati mušteriji kusur od 41 cent, a na raspolaganju su mu novčići od 25, 10, 5 i 1 centi. Instinktivno će većina ljudi vratiti 1x25, 1x10 i 5x1, i 1x1 cent. Takav algoritam vraća tačan iznos uz najkraću moguću listu novčanica (Slika 2.3).

Slika 2.1.3 Primena pohlepnog algoritma kod problema kusura [7]

Dakle, postupak je:

Izabere se najveća novčanica koja ne prelazi ukupnu sumu (25 centi), sračuna se ostatak za vraćanje (41 - 25 = 16 centi), zatim se bira sledeća novčanica koja ne prelazi ostatak koji treba vratiti (10 centi), i opet se računa novi ostatak za vraćanje (16 - 10 = 6 centi). Ovo se vrši sve dok se ne vrati tačan iznos.

Strategija pohlepnog pristupa je u konkretnom primeru sa 41 centa dovela do najefikasnijeg rešenja problema vraćanja novca. Međutim, pohlepni algoritmi vrlo rano donose određene odluke ili ne obrađuju temeljno sve mogućnosti, što ih može sprečiti da kasnije nađu najbolje rešenje.

Recimo, u partiji šaha, pohlepni algoritam će uvek odigrati onaj potez koji izgleda najbolje u datom trenutku, ne obazirući se na posledice. Na primer, ukoliko najveću vrednost za igrača u datoj situaciji ima potez u kome on uzima protivničkog lovca, on će to i učiniti, sve i ako taj potez otvara protivniku mogućnost da matira igrača. Ostali primeri primene pohlepnog algoritma su:

- Kruskalov i Primov algoritam za nalaženje minimalnog obuhvatnog stabla,
- Dijkstra algoritam za nalaženje najkraćih puteva u grafu iz jednog početka,
- Algoritam za nalaženje optimalnog Hafmanovog stabla (kompresija podataka).

DINAMIČKO PROGRAMIRANJE

Ideja je sačuvati rešenja onih problema koji su već rešeni, da bi se kasnije rezultati mogli iskoristiti umesto da se problemi pon ovo rešavaju.

Dinamičko programiranje je pristup koji izbegava ponovno rešavanje potproblema koja su već rešena.

Analizirajmo problem određivanja Fibonačijevih brojeva, gde se svaki sledeći Fibonačijev broj određuje na osnovu prethodna dva korišćenjem formule:

f(n) = f(n-1) + f(n-2).

Implementacija funkcije za traženje n-tog člana Fibonačijevog niza se bazira na prethodnom izrazu, i koristi princip rekurzije:

```
function fib(n)
    if n = 0 or n = 1
        return n
    else
        return fib(n − 1) + fib(n − 2)
```

Primetimo da, ako se pozove funkcija fib(5), odgovarajuće funkcije za fib(3) i fib(2) će biti pozvane po nekoliko puta. Na primer, treći i četvrti broj mogu biti sračunati kao $F_3 = F_1+F_2$

i $F_4 = F_2+F3$, pa računanje svakog broja zahteva nalaženje F_2. Kako su F_2 i F_4 potrebni za nalaženje F_5 ovim pristupom bi za računanje F_5 bilo potrebno nalaženje F_2 nekoliko puta.

Kako bi se ovo izbeglo, potrebno je sačuvati rešenja onih problema koji su već rešeni, da bi se kasnije mogla iskoristiti.

Postupak čuvanja rešenja se zove ***memoizacija*** , koja je osnova dinamičkog programiranja.

U odgovarajućim slučajevima, ako je očigledno da rešeni potproblemi nisu više potrebni, mogu se odbaciti kako bi se sačuvao memorijski prostor.

PRISTUPI DINAMIČKOG PROGRAMIRANJA

Postoje dva pristupa kod dinamičkog programiranja: odgozgo na dole i odozdo na gore.

Ovaj algoritam koristi najčešće jedan od sledeća dva pristupa:

- ***Odozgo na dole*** : Problem se rastavi na potprobleme, potproblemi se reše i pamte se njihova rešenja, za slučaj njihove kasnije upotrebe. Ovaj pristup predstavlja kombinovanje rekurzije i memoizacije.

Ako se koristi pristup ***odozgo na dole*** (memoizacija) tada se potproblemi rešavaju tačno jednom, jer se njihove vrednosti pamte:

```
var m = map(0 → 1, 1 → 1)
function fib(n)
    if n not in keys(m)
        m[n] = fib(n − 1) + fib(n − 2)
    return m[n]
```

- **Odozdo na gore**: Svi potproblemi se rešavaju redom i koriste za nalaženje sledećeg koji se rešava. Ovaj pristup je bolji u odnosu na pristup "odozgo na dole" zbog čuvanja memorijskog prostora na steku.

Koristeći pristup ***odozdo na gore*** , polazimo od rešenja osnovnih slučajeva, i dalje ih koristimo za rešavanje narednih potproblema, i na kraju i konačnog rešenja:

```
function fib(n)
    var previousFib = 1, currentFib = 1
    repeat n − 1 times
        var newFib  = previousFib + currentFib
        previousFib = currentFib
        currentFib  = newFib
    return currentFib
```

Najpoznatiji primeri primene algoritma dinamičkog programiranja su: algoritmi sa stringovima (najduža zajednička podsekvenca, najduži zajednički podstring, itd...), kao i Dijkstra , Belman-Ford i Flojd-Varšal algoritmi za određivanje najkraćih putanja pri radu sa grafovima, itd.

2.1 Ostale algoritamske strategije

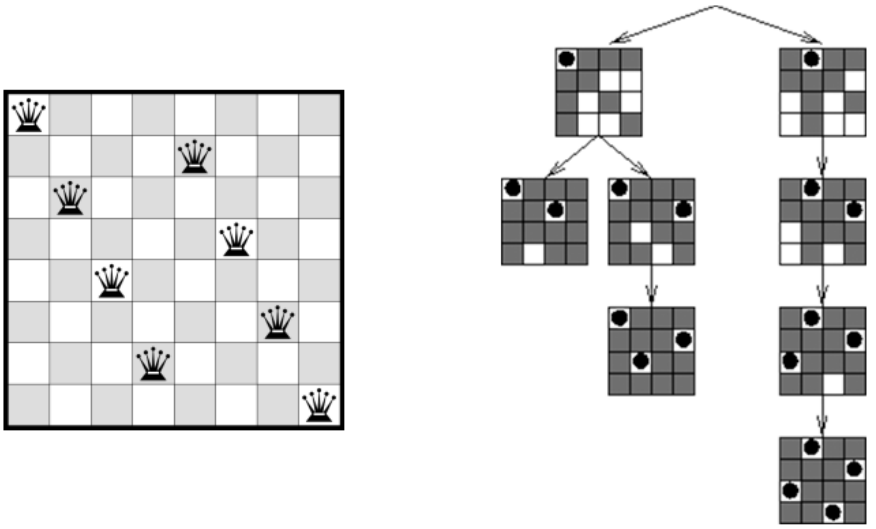
BEKTREKING ALGORITAM

Bektreking je algoritam grube sile kod koga se grade kandidati za rešenje, a odbacuju se svi kandidati za koje se ispostavi da ne vode do tačnog rešenja

Metod obrnute pretrage (eng. Backtracking) predstavlja pristup *grube sile* u traženju rešenja gde se isprobavaju sve moguće kombinacije. Ideja je sledeća:

- Postepeno se grade kandidati za rešenje, a odbacuju se svi kandidati za koje se ispostavi da ne vode do tačnog rešenja.

Algoritam se često sporo izvršava, pa se koriste algoritmi prolagođeniji za dati problem.



Slika 2.2.1 Postavljanje kraljica na šahovskoj tabli da se ne napadaju [1]

Klasičan primer bektreking algoritma je problem osam dama (Slika 3.1). Kod ovog problema traga se za rasporedom osam dama na standardnoj šahovskoj tabli, tako da nijedna dama ne napada bilo koju drugu.

Pogledajmo uprošćeni primer sa tablom 4x4. Gde je ovde obrnuta pretraga?

Prvo postavimo 1. kraljicu na poziciju i u 1.vrsti. Zatim postavimo 2. kraljicu na poziciju j u 2. vrsti. Pokušamo da postavimo 3. kraljicu u 3. vrstu. Ukoliko nije moguće vraćamo se (backtrack) na 2. kraljicu, probamo da je postavimo na j+1-u poziciju, itd...

Obrnuta pretraga može biti od koristi samo u problemima kod kojih možemo primeniti koncept **"*odabira odgovarajućih kandidata*"** i kod kojih relativno brzo možemo ispitati da li određen kandidat može biti validni deo rešenja.

PRIMERI BEKTREKING ALGORITMA

Osnovni primeri primene algoritma obrnute pretrage su problem postavljanja kraljica na šahovskoj tabli, obilazak table skakačem, sudoku, ...

Osnovni primeri primene algoritma obrnute pretrage:

- Postavljanje kraljica na šahovskoj tabli (Video 1)
- Obilazak table skakačem (Video 2)
- Rešavanje sudoku problema (Video 3)

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Naredni video klipovi preuzeti sa linkova: https://en.wikipedia.org/wiki/File:Knight%27s_tour_anim_2.gif

https://en.wikipedia.org/wiki/Backtracking#/media/File:Sudoku_solved_by_bactracking.gif.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

HEURISTIKE

Cilj heuristike je da se brzo dođe do rešenja koje je dovoljno dobro za problem koji se rešava

Složen problem se često ne može rešiti tačno, i zato koristimo približno rešenje! Heuristika je jedan vid rešavanja složenih problema.

Umesto da se izlistaju sva rešenja nekog problema i među njima nađe najbolje rešenje, cilj heuristike je da se brzo dođe do rešenja koje je dovoljno dobro za problem koji se rešava. To rešenje ne mora biti nužno najbolje , ili čak može biti samo aproksimacija tačnog rešenja. I pored toga takvo rešenje je vredno zato što za njegovo nalaženje nije potrošeno preterano mnogo vremena.

Primer 1 (Heuristika za vožnju kroz Pariz)

Pretpostavimo da ste u poseti prijatelju u Parizu i da ste iznajmili auto na aerodromu. Vaš cilj je da po što kraćoj putanji dođete od aerodroma do stana vašeg prijatelja.

Rešenje: Posmatramo sledeći problem: Od svih mogućih putanja od aerodroma do stana prijatelja, izabrati najkraću. U zavisnosti od rasporeda ulica i njihove prohodnosti, problem može biti manje-više složen. Npr. da su ulice Pariza međusobno paralelne i iste dužine, problem bi bio jednostavan. Kako to nije slučaj sa Parizom, problem je složen. Heuristika bi,

u ovom slučaju, dala rešenje po kome bismo se kretali samo po glavnim ulicama. Ovo nije najbolje rešenje, ali je dovoljno dobro.

Primer 2 (Putovanje po Evropi)
Pretpostavimo da želite da putuje po Evropi i da na tom putovanju obilazite sve glavne gradove a da troškovi puta budu minimalni.
Rešenje: Problem je poznat kao problem Trgovačkog putnika (rastojanje između gradova se zameni troškovima puta između dva grada).
Heuristika bi dala sledeće rešenje: Iz grada A kretati se ka onom gradu do kog su troškovi puta najjeftiniji.

Mnogi antivirusi koristi heuristička pravila za detektovanje virusa i drugih tipova malvera. Heuristička pretraga traži kodove i/ili šablone ponašanja koji mogu da pripadaju klasi ili familiji virusa, sa različitim skupom pravila za različite viruse. Ako posmatrani fajl, ili proces koji se izvršava sadrži poklapajuće uzorke koda, i/ili izvršava odgovarajući skup aktivnosti, onda skener zaključuje da je fajl zaražen.

PREOBRAZI I OSVOJI (TRANSFORM AND CONQUER)

U prvoj fazi se problem transformiše na lakši oblik a u drugoj fazi se transformisani problem rešava

Ovu tehniku zovemo transform-and-conquer jer je to procedura koja radi u dva koraka. Prvo, u fazi transformacije, instanca problema se modifikuje tako da bude, iz datog razloga, pogodnija za rešavanje. Zatim, u drugoj fazi (osvajanja), vrši se rešavanje problema.

Primer 1. Razmotrimo slučaj množenja dva broja XII i IV, koji su deo rimskog brojevnog sistema. Pošto mnogi nisu upoznati sa ovim brojevnim sistemom, ideja je da problem transformišemo na drugi koji se svodi na korišćenje arapskih umesto rimskih brojeva.

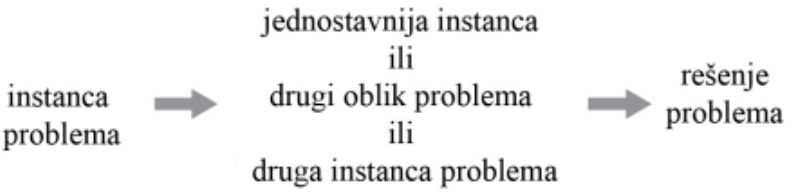
1. U prvoj fazi, brojevi XII i IV se transformišu na arapske brojeve 12 i 4.

2. U drugoj fazi se vrši množenje brojeva 12 x 4 = 48, i rezultat se konvertuje u rimski broj XLVIII.

Primer 2. Drugi, veoma poznati, primer primene ove strategije je određivanje najmanjeg zajedničkog sadržaoca (LCM) korišćenjem najvećeg zajedničkog delioca (GCD). Na primer, ako je poznat GCD dva broja, onda se LCM može odrediti korišćenjem sledeće jednačine:

$$lcm(m,n) = \frac{mn}{GCD(m,n)}$$

Postoje sledeće glavne varijacije ove strategije koje se razlikuju prema tipu transformacije instance problema (Slika 3.2).



Slika 2.2.2 Postupak preobrazi i osvoji algoritma [izvor: autor]

Primeri primene: prethodno sortiranje niza pre ulaska u glavni algoritam (presorting), Balansirana stabla pretraživanja, AVL stablo, 2-3 stablo, Crveno-crno (stablo), itd...

Poglavlje 3

Primer: Najveći zajednički delilac

ALGORITAM ISCRPNE PRETRAGE ZA ODREĐIVANJE NZD DVA BROJA

Algoritam iscrpne pretrage za određivanje NZD dva broja se sastoji iz provere svih mogućih delioca dva broja, počevši od broja 1, i čuvanje onog koji je trenutno najveći

Kao što je već poznato sa osnovnih kurseva o programiranju, najveći zajednički delilac (***NZD*** , ili ***GCD*** - greatest common divisor) brojeva *4* i *2* je *2*. Najveći zajednički delilac brojeva *16* i *24* je *8*. Ono što se postavlja kao pitanje je kako izgleda kreiranje algoritma koji će odrediti najveći zajednički delilac dva broja?

Osnovni algoritam koji koristi princip **iscrpne pretrage** (brute-force) se može napisati na sledeći način [1]:

- Neka su uneta dva cela broja *n1 i n2*.
- Znamo da je broj *1* sigurno zajednički delilac dva uneta broja, ali ne mora biti najveći. Stoga je potrebno ispitati da li je *k* (gde je*k* = *2*, *3*, *4*, itd) najveći zajednički delilac brojeva *n1* i *n2*, sve dok *k* ne postane veći od *n1* ili *n2*.
- Koristićemo pomoćnu promenljivu *nzd* da bi u njoj čuvali NZD. Inicijalno, *nzd* je *1*.
- Kad god pronađemo zajednički delilac, u petlji gde *k* ide od *2*, do manjeg broja *n1* i *n2*, on postaje novi najveći zajednički delilac.
- Kada ispitamo sve moguće zajedničke delioce između brojeva *2* i *n1* ili *n2*, vrednost koja je sačuvana u *nzd* će ustvari biti najveći zajednički delilac.

Ova ideja može biti pretočena u sledeći kod:

```
int nzd = 1; // Pocetno gcd je 1
int k = 2; // Potencijalni nzd
while (k <= n1 && k <= n2) {
    if (n1 % k == 0 && n2 % k == 0)
        nzd = k; // Azuriraj nzd
    k++; // Sledeci potencijalni nzd
}
```

IMPLEMENTACIJA ALGORITMA U JAVA-I

Veoma je bitno dobro razmisliti o problemu pre samog pisanja koda. Razmišljanje o problemu vam omogućava da kreirate logično rešenje problema bez brige o tome kako ćete napisati kod

U sledećem listingu je program koji od korisnika zahteva da unese dva pozitivna cela broja, a zatim određuje njihov najveći zajednički delilac:

```
import java.util.Scanner;
public class GreatestCommonDivisor {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Unesi prvi broj: ");
        int n1 = input.nextInt();
        System.out.print("Unesi drugi broj: ");
        int n2 = input.nextInt();
        int nzd = 1;
        int k = 2;
        while (k <= n1 && k <= n2){
            if (n1 % k == 0 && n2 % k == 0)
                nzd = k;
            k++;
        }

        System.out.println("NZD za " + n1 +
                " i " + n2 + " je " + nzd);
    }
}
```

Moguće rešenje je:

```
Unesi prvi broj: 125
Unesi drugi broj: 2525
NZD za 125 i 2525 je 25
```

Postavlja se pitanje kako bi ste napisali ovaj programi? Da li bi odmah krenuli sa pisanjem koda? Odgovor treba da bude NE! Naime, veoma je bitno dobro razmisliti o problemu pre bilo kakvog pisanja koda. Razmišljanje o problemu vam omogućava da kreirate logično rešenje problema bez brige o tome kako ćete napisati kod. Jednom kada ste sigurni da ste osmislili logično rešenje, ostaje da algoritam prevedete na odgovarajući programski jezik (Java, C, C++, itd...). Prevod na programski jezik nije jedinstven, pa je tako umesto ***while*** moguće koristiti ***for*** petlju, i isečak koda koji određuje NZD bi mogao da ima sledeći oblik:

```
for (int k = 2; k <= n1 && k <= n2; k++){
    if (n1 % k == 0 && n2 % k == 0)
        nzd = k;
}
```

Svaki problem obično ima više rešenja, pa tako se i problem najvećeg zajedničkog delioca može rešiti na više načina. Mnogo efikasnije rešenje ovog problema je ***Euklidov algoritam*** , ali o njemu će biti više reči u lekciji o rekurziji.

Poglavlje 4

Primer: Prikaz prostih brojeva

ALGORITAM ISCRPNE PRETRAGE ZA PRIKAZ PROSTIH BROJEVA

U ovoj sekciji je predstavljen program koji na ekranu prikazuje prvih 50 prostih brojeva, i to u 5 linija, pri čemu u svakoj liniji ima po 10 prostih brojeva

Pošto će u narednoj lekciji o ***Analizi algoritama*** biti uvedeni efikasniji algoritmi pronalaženja prostih (engl. prime) brojeva, krenućemo od ovog najjednostavnijeg, ali i najsporijeg algoritma. Kao što je poznato, pozitivan ceo broj veći od *1* je ***prost*** ako je deljiv samo sa *1* i sa samim sobom. Tako su na primer brojevi *2*, *3*, *5*, i *7* prosti brojevi, dok *4*, *6*, *8*, i *9* nisu. Cilj ovog zadatka je da se na ekranu prikaže prvih 50 prostih brojeva, i to u 5 linija, gde svaka linija sadrži 10 prostih brojeva. Problem se može podeliti u nekoliko sledećih koraka, odnosno podzadataka [1]:

- Napisati kod koji određuje da li je proizvoljni broj prost.
- Zatim za brojeve *number* = *2*, *3*, *4*, *5*, *6*, ..., ispitati da li su prosti.
- Vršiti brojanje prostih brojeva.
- Prikazati svaki prost broj, ali tako da ih ima 10 u jednoj prikazanoj liniji na ekranu.

Na osnovu prethodnih koraka, očigledno je da morate napisati petlju koja će iznova ispitivati da li je novi broj *number* prost. Ukoliko je *number* prost uvećati brojač *count* za *1*. Naravno brojač *count* je inicijalno jednak *0*. Kada brojač dostigne *50*, petlja se prekida.

Naravno, svaki put kada je brojač *count* deljiv sa 10, treba prebaciti kursor u sledeći red. Stoga, algoritam za rešenje ovog problema ima oblik:

```
Broj prostih brojeva koji treba biti odstampan je jednak
vrednosti konstante NUMBER_OF_PRIMES;
Primeni count da pratis broj prostih brojeva, i
Postavi pocetno count na 0;
Postavi pocetno number na 2;
while (count < NUMBER_OF_PRIMES) {
 Proveri da li je number prost;
 if number je prost {
   Prikazi number i uvecaj count;
 }
 Uvecaj broj za 1;
}
```

Da bi smo ispitali da li je broj prost, ispitaćemo da li je deljiv sa nekim brojem u intervalu od *2* do *number / 2*. Ukoliko je pronađen delilac onda broj nije prost. Algoritam može biti napisan na sledeći način:

```
Primeni bool promenljivu isPrime da oznacis da li je
number prost broj ; Pocetno, isPrime ima vrednost true;
for (int divisor = 2; divisor <= number / 2; divisor++) {
 if (number % divisor == 0) {
  Setuj isPrime na false
  Izadji iz petlje;
 }
}
```

ANALIZA ALGORITMA ZA ODREĐIVANJE PROSTIH BROJEVA

Ključ za kreiranje izvodljivog rešenja ovog i mnogih drugih problema je cepkanje složenog problema na potprobleme, a zatim i rešavanje svakog od prostijih potproblema posebno

U nastavku je kompletan kod za određivanje prostih brojeva:

```
public class PrimeNumber {
  public static void main(String[] args) {
    final int NUMBER_OF_PRIMES = 50;
    final int NUMBER_OF_PRIMES_PER_LINE = 10;
    int count = 0;
    int number = 2;

    System.out.println("Prvih 50 prostih brojeva su \n");
    while (count < NUMBER_OF_PRIMES) {
        boolean isPrime = true; // Da li je trenutni broj prost?
        for (int divisor = 2; divisor <= number / 2; divisor++) {
            if (number % divisor == 0) { // Ako je true, number nije prost
                isPrime = false;
                break; // Izadji iz for petlje
            }
        }
        if (isPrime) {
            count++; // Uvecaj brojac count

            if (count % NUMBER_OF_PRIMES_PER_LINE == 0)
                System.out.println(number);
            else
                System.out.print(number + " ");
        }
        number++;
    }
```

```
  }
}
```

Ovo je naravno problem koji je veoma složen za početnike u programiranju. Ključ za kreiranje izvodljivog rešenja ovog i mnogih drugih problema je cepkanje složenog problema na potprobleme, a zatim i rešavanje svakog od prostijih potproblema posebno. Stoga je preporuka da se ne razvija kompletno rešenje problema u jednom pokušaju (iteraciji). Umesto toga, započeti pisanje koda sa ciljem da se prvo odredi da li je neki broj prost ili ne, pa tek onda kada ste sigurni da algoritam radi za jedan broj, kreirajte petlju u kojoj ćete ispitati da li su ostali brojevi prosti ili ne.

Da bi ste odredili da li je neki broj prost, prvo ispitajte da li je deljiv sa brojevima između *2* i *number/2*, uključujući i njih**.** Ukoliko je to tačno broj nije prost, a u suprotnom jeste. Ukoliko je broj prost odštampajte ga na ekran, i uvećajte brojač za 1. Ukoliko je brojač deljiv sa 10 pređite u sledeći red. Program se završava kada brojač dostigne *50*.

Program koristi iskaz ***break*** u liniji 19 da bi smo izašli iz ***for*** petlje onog trenutka kada zaključimo da broj nije prost. Na taj način ubrzavamo rad koda jer ne vršimo nepotrebno ispitivanje ostalih brojeva do *number/2*.

Poglavlje 5

Vežbe

PRIMER 1. ODREĐIVANJE JEDINSTVENOSTI ELEMENTA U NIZU (10 MIN)

Određivanje jedinstvenog elementa u nizu

Razmotrimo problem određivanja jedinstvenog elementa u nizu. Problem može biti definisan na sledeći način: Za zadatu listu slučajnih brojeva, proveriti da li ima duplikata.

Algoritam grube sile (brute force)

Algoritam grube sile uključuje proveru svakog para elemenata u cilju odredjivanja duplikata. Ovo znači poredjenje jednog elementa sa svim ostalim elementima u nizu. Algoritam grube sile može biti predstavljen na sledeći način:

```
function Uniqueness
    for each x  belongs  A
        for each y   belongs  {A-x}
            if x=y then
                return not unique
            endif
    return unique
```

Preobrazi i osvoji (transform and conquer)

Kao što je prikazano na predavanjima, možemo u ovom slučaju da primenimo princip uprošćavanja samog problema. Ili da jednostavno pre same faze provere izvršimo sortiranje niza. Prednost korišćenja sortiranja je u tome da je nakon toga dovoljno proveriti samo susedne elemente niza.

Tako se algoritam svodi na:

1. Sortirati brojeve.

2. Proveriti susedne brojeve. Ako su brojevi jednaki vratiti da jedinstvenost (uniqueness) nije zadovoljena (tj. vratiti *false*).

3. Kraj.

Napisano korišćenjem pseudo-koda, to bi izgledalo ovako:

```
function ElementUniqueness_Presorting(A[1..n])
    Ulaz: Array A
    Izlaz: Unique ili not
```

```
    Begin
    Sort A
    for i = 1 to n-1
        if A[i] = A[i+1] return not unique
    End for
    return unique
```

Zadatak za samostalni rad: Napisati Java kod za oba algoritma.

PRIMER 2. MOD NIZA (10 MIN)

Mod je definisan kao element koji se najčešće javlja u nizu

Mod (mode) je element koji se najčešće pojavljuje u nizu. Na primer, razmotrimo sledeći niz elemenata, A = [5, 1, 5, 5,5,7, 6, 5, 7,5] . Mod niza A je 5 jer se on najviše puta pojavljuje u nizu. Ukoliko ima više elemenata koji se pojavljuju identičan broj puta, a više od ostalih elemenata niza, onda svaki od njih predstavlja mod niza.

Algoritam grube sile (brute force)

Algoritam grube sile se svodi na skeniranje celog niza i brojanje pojavljivanja svakog elementa. Pristup baziran na odredjivanju učestalosti pojavljivanja može biti napisan na sledeći način:

```
Korak 1: Find length of List A
         max  = max(A)
Korak 2: Set freq[1..max] to  0
Korak 3: for each x  from  A
            freq[x] = freq[x] + 1
Korak 4: mode  =  freq[1]
Korak 5: for i  from  2 to max
            if freq[i] > freq[mode] mode  = i
Korak 6: return mode
```

Preobrazi i osvoji (transform and conquer) pristup
Umesto prethodnog pristupa, moguće je prvo sortirati niz, a zatim odrediti najdužu sekvencu identičnih brojeva u tom sortiranom nizu. Algoritam bi se sveo na sledeći postupak:
1) Sortirati elemente niza.
2) Odrediti sekvence identičnih brojeva
3) Odštampati element koji ima najdužu sekvencu pojavljivanja
4) Kraj.
Korišćenjem pseudo koda prethodni algoritam može biti napisan kao:

```
Sort A
i = 0
frequency = 0
while i <= n-1
    runlength = 1; runvalue = A[i]
    while i+runlength ≤ n-1 and A[i+runlength] = runvalue
        runlength = runlength + 1
```

```
    if runlength > frequency
        frequency = runlength
        modevalue = runvalue
    i = i + runlength
return modevalue
```

Zadatak za samostalni rad: Napisati Java kod za oba algoritma.

ZADATAK 1 - PROST PALINDROMSKI BROJ (10 MIN)

Prost palindromski broj je prost broj koji je istovremeno i palindrom

Prost palindromski broj je prost broj koji je istovremeno i palindrom. Na primer, 131 je prost ali istovremeno i palindrom, kao i brojevi 313 i 757. Napisati program koji prikazuje prvih 100 prostih brojeva koji su i palindromi. Prikazati 10 brojeva u jednoj liniji, pri čemu su brojevi u liniji razdvojeni tačno jednom prazninom.

Primer:

```
2 3 5 7 11 101 131 151 181 191

313 353 373 383 727 757 787 797 919 929
```

Kompletan kod primera je dat u sledećem listingu:

```
public class Zadatak1 {
    public static void main(String[] args) {
        int count = 1;

        for (int i = 2; true; i++) {
            // Proveri da li je broj prost i palindrom:
            if (isPrime(i) && isPalindrome(i)) {
                System.out.print(i + " ");

                if (count % 10 == 0) {
                    System.out.println();
                }

                if (count == 100) {
                    break;
                }

                count++;
            }
        }
    }
    // provera da li je broj prost
    public static boolean isPrime(int num) {
        for (int i = 2; i <= num / 2; i++) {
```

```
            if (num % i == 0) {
                return false;
            }
        }

        return true;
    }
    // obrtanje redosleda cifara u broju
    static int reversal(int number) {
        int result = 0;

        while (number != 0) {
            int lastDigit = number % 10;
            result = result * 10 + lastDigit;
            number = number / 10;
        }

        return result;
    }
    // glavna funkcija koja proverava da li je broj palindrom
    static boolean isPalindrome(int number) {
        return number == reversal(number);
    }
}
```

ZADATAK 2 - ANAGRAMI (10 MIN)

Dve reči su anagrami ako sadrže ista slova, bez obzira na redosled i raspored

Napisati metod koji proverava da li su dve reči anagrami. Dve reči su anagrami ako sadrže ista slova, bez obzira na redosled i raspored. Na primer, reči ***silent*** i ***listen*** su anagrami. Zaglavlje metode je:

```
public static boolean isAnagram(String s1, String s2)
```

Napisati test program koji od korisnika zahteva da unese dva stringa i da prikaže poruku ***“dva stringa su anagrami”,*** odnosno ***“dva stringa nisu anagrami”,*** u zavisnosti od rezultata ispitivanja.

Kompletan kod primera je dat u sledećem listingu. Kao što možete primetiti, neophodno je da funkciju *SelectionSort:sort()* koja je obrađena na predavanjima transformišete tako da radi sa stringovima umesto sa realnim brojevima.

```
public class Zadatak2 {
    public static void main(String args[]) {
        // Zatrazi od korisnika da unese stringove
        java.util.Scanner input = new java.util.Scanner(System.in);
        System.out.print("Unesi prvi string: ");
```

```
        String first = input.nextLine();

        System.out.print("Unesi drugi string: ");
        String second = input.nextLine();
        System.out.println(
                first + " i " + second + " string " +
                (isAnagram(first, second) ?
                "su anagrami." : "nisu anagrami."));
    }

    public static boolean isAnagram(String s1, String s2) {
        String newS1 = SelectionSort.sort(s1);
        String newS2 = SelectionSort.sort(s2);

        if (newS1.length() != newS2.length()) return false;

        for (int i = 0; i < newS1.length(); i++) {
            if (newS1.charAt(i) != newS2.charAt(i))
                return false;
        }

        return true;
    }
}
```

ZADATAK 3 - SUMA PRVE I POSLEDNJE CIFRE BROJA (10 MIN)

Suma prve i poslednje cifre broja

Sa standardnog ulaza učitava se ceo broj.

Napisati program koji na standardni izlaz ispisuje sumu njegove prve i poslednje cifre. Ukoliko je uneti broj jednocifren, program treba da na standardni izlaz ispiše -1.

PRIMER: X=9, output: -1

X=123, output: 4

X=1235, output: 6

```
package cs103l01z3;

import java.util.Scanner;

public class Zadatak3 {

    public static void main(String[] args) {
        sumaPrveIPosednjeCifre();
    }
```

```
    private static void sumaPrveIPosednjeCifre() {
        int n;
        Scanner scan = new Scanner(System.in);
        // Učitavamo broj
        System.out.println("Uneti broj za proveru:");
        n = scan.nextInt();
        int prva_cifra, poslednja_cifra;

        // Proveravamo da li je broj jednocifren ili negativan
        if (n < 0) {
            System.out.println("Rezultat: -2");
            return;
        } else if (n >= 0 && n < 10) {
            System.out.println("Rezultat: -1");
            return;
        }
        // Ako nije, izdvajamo poslednju cifru
        poslednja_cifra = n % 10;

        // Izdvajamo prvu cifru
        while (n >= 10) {
            n = n / 10;
        }
        prva_cifra = n;

        System.out.println("Rezultat: " + (prva_cifra + poslednja_cifra));
    }
}
```

Poglavlje 6

Zadaci za samostalni rad

ZADACI ZA SAMOSTALNO VEŽBANJE

Na osnovu materijala za ovu nedelju uraditi samostalno sledeće zadatke:

Zadatak 1. (*Markov matrica* - 30 min) Neka *n x n* matrica se naziva Markov matrica, ako je svaki element matrice pozitivan i ako je suma elemenata u svakoj koloni jednaka 1. Napisati sledeći metod koji proveravamo da li je neka matrica Markov matrica. Koristiti sledeće zaglavlje:

```
public static boolean isMarkovMatrix(double[][] m)
```

Napisati test program koji od korisnika zahteva da unese matricu realnih brojeva tipa double, a zatim i da testira da li je uneta matrica Markov matrica ili ne. U nastavku su primeri izvršavanja programa (Slika 6.1):

```
Enter a 3-by-3 matrix row by row:
0.95 -0.875 0.375 Enter
0.65 0.005 0.225 Enter
0.30 0.22 -0.4 Enter
It is not a Markov matrix
```

```
Enter a 3-by-3 matrix row by row:
0.15 0.875 0.375 Enter
0.55 0.005 0.225 Enter
0.30 0.12 0.4 Enter
It is a Markov matrix
```

Slika 6.1 Mogući primer izvršavanja programa [1]

Zadatak 2. (*Najveći blok* -30 min) Za zadatu kvadratnu matricu čiji su elementi samo 0 i 1, napisati program koji pronalazi najveću kvadratnu podmatricu čiji su elementi samo jedinice. Korisnik treba da unese broj vrsta matrice, i matricu (vrstu po vrstu).

Zadatak 3. Napisati funkciju *int suma_delilaca(int n)* koja određuje sumu pravih delilaca celog broja n (tj.sumu svih delilaca broja n, ne računajući sam broj n). (30 min)
Napisati potom program koji sa standardnog ulaza učitava ceo
broj n>1 i ispisuje na standardni izlaz sumu njegovih delilaca, ne računajući sam taj broj, korišćenjem implementirane
funkcije.

Ako n nije veće od 1, program treba da na standardni izlaz za grešku ispiše -1.

Program treba kao rezultat da prikaže lokaciju prvog elementa najveće kvadratne podmatrice, kao i broj vrsta najveće podmatrice. Program treba da implementira i da koristi sledeći metod za određivanje najveće kvadratne podmatrice:

```
public static int[] findLargestBlock(int[][] m)
```

Poglavlje 7

Domaći zadatak

PRAVILA ZA DOMAĆI ZADATAK

Detaljno proučiti pravila za izradu domaćih zadataka

Svaki student dobija od asistenta sopstvenu kombinaciju domaćeg zadatka.

Online studenti bi trebalo mailom da se najave, kada budu želeli da krenu sa radom na predmetu i prikupljanjem predispitnih obaveza.

Odgovarajući NetBeans (Eclipse ili Visual Studio) projekat koji predstavlja rešenje domaćeg zadatka smestiti u folder CS203-DZ01-Ime-Prezime-BrojIndeksa. Zipovani folder CS203-DZ01-Ime-Prezime-BrojIndeksa poslati predmetnom asistentu (lazar.mrkela@metropolitan.ac.rs) u mejlu sa naslovom (subject)CS203-DZ01, inače se neće računati.

Studenti iz Niša predispitne obaveze predaju asistentima u Nišu jovana.jovanovic@metropolitan.ac.rs i uros.lazarevic@metropolitan.ac.rs.

Student tradicionalne nastave ima 7 dana, od dana kada je dobio mail sa domaćim zadatkom, da uradi i pošalje rešenje za maksimalan broj poena.

Ukoliko student pošalje domaći nakon tog roka, najviše može da ostvari 50% od maksimalnog broja poena.

Studenti online nastave imaju rok da predaju rešene domaće zadatke 10 dana pre termina ispita u ispitnom roku u kome polažu CS203 Algoritmi i strukture podataka.

Vreme izrade: 2h.

Zaključak

REZIME

Na osnovu svega obrađenog možemo zaključiti sledeće:

Algoritam je precizno definisan postupak za rešavanje nekog problema. Prosto rečeno, to je konačna i precizno definisana procedura (niz dobro definisanih pravila) kojom se ulazne vrednosti transformišu u izlazne. Algoritmi se obično predstavljaju na dva načina: grafički i pomoću pseudokoda.

Algoritmi se obično razvrstavaju prema metodologiji projektovanja ili primenjenom obrascu. Ovom problematikom se bavi oblast koja se zove algoritamske strategije**.** Postoji veliki broj algoritamskih strategija, i bitno je poznavati principe svake od strategija kako bi izabrali najpogodniji algoritam za postavljeni problem.

Iscrpna pretraga je opšta tehnika rešavanja problema koja se sastoji od sistematičnog nabrajanja svih kandidata kao mogućih rešenja i provere da li svaki kandidat zadovoljava rešenje problema.

Podeli i osvoji algoritam smanjuje stepen složenosti problema podelom na dva ili više manjih problema iste vrste dok od problema ne ostane toliko mali deo da se može jednostavno rešiti.

Pohlepni algoritam u svakom koraku bira u tom trenutku naizgled najbolje rešenje, bez obzira da li će to uticati na tačnost konačnog rešenja problema.

Kada se optimalno rešenje problema može konstruisati iz rešenja potproblema, i preklapanjem potproblema, glavni problem možemo rešiti brzo korišćenjem dinamičkog programiranja. Dinamičko programiranje je, stoga, pristup koji izbegava ponovno izračunavanje rešenja koja su već izračunata.

Pretraživanje je proces traženja odgovarajućeg elementa u nizu. Osnovni tipovi pretraživanja su linearno i binarno pretraživanje. Linearno pretraživanje je algoritam iscrpne pretrage, dok binarno pretraživanje spada u specijalan slučaj "podeli i osvoji" algoritma.

Sortiranje, kao i pretraživanje, je veoma čest zadatak u programiranju. Metoda selekcije pronalazi najmanji element u listi i zamenjuje ga sa prvim elementom. Zatim se u ostatku liste ponovo pronalazi najmanji element i zamenjuje sa drugim članom liste.

REFERENCE

Korišćena literatura

[1] D. Liang, Introduction to Java Programming, Comprehensive Version, 10th edition, Prentice Hall, 2014

[2] M.A. Weiss, Data Structures and Problem Solving Using Java, 3rd edition, Addison Wesley, 2005.

[3] http://www.tutorialspoint.com/data_structures_algorithms/

[4] R. Sedgewick, K.Wayne, Algorithms, 4th edition, Pearson Education, 2011.https://en.wikipedia.org/wiki/Divide_and_conquer_algorithm

[5] 6. C. Shaffer, Data Structures and Alghoritm Analysis, Virgina Tech, 2012.

[6] https://en.wikipedia.org/wiki/Divide_and_conquer_algorithm

[7] https://en.wikipedia.org/wiki/Greedy_algorithm

[8] https://code.energy/8-queens-problem/

[9] http://openclassroom.stanford.edu/MainFolder/CoursePage.php?course=IntroToAlgorithms

[10] http://somemoreacademic.blogspot.com/2012/09/brute-forcenaive-string-matching.html