UNIVERZITET METROPOLITAN BEOGRAD

CS203 - ALGORITMI I STRUKTURE PODATAKA

Rekurzija - Napredna analiza

Lekcija 04

PRIRUČNIK ZA STUDENTE

CS203 - ALGORITMI I STRUKTURE PODATAKA

Lekcija 04

REKURZIJA - NAPREDNA ANALIZA

UVOD

Ova lekcija treba da ostvari sledeće ciljeve:

U okviru ove lekcije studenti se upoznaju sa:

- Primenom rekurzije kod Euklidovog algoritma za određivanje NZD dva broja
- Kreiranjem fraktala primenom rekurzije
- Rekurzivnim ***„podeli i osvoji“*** algoritmima i master teoremom
- Primenom rekurzije kod ***dinaminičkog programiranja***

Rekurzija u programiranju označava situaciju kada metod, procedura ili funkcija poziva samu sebe.

Nadovezujući se na znanja koja su usvojena u okviru predmeta CS202, vezano za rekurziju, u okviru ove lekcije studenti se upoznaju sa mnoštvom korisnih primera gde rekurzija nalazi svoju efikasnu primenu.

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

Poglavlje 1

Primer: Euklidov algoritam

DEFINICIJA I IMPLEMENTACIJA EUKLIDOVOG ALGORITMA

Mnogo efikasniji algoritam za određivanje NZD dva broja je otkrio Euklid oko 300. god. PNE

Jedan od najefikasnijih algoritama za određivanje NZD dva broja je otkrio Euklid oko 300. god. pre nove ere. Ovo je jedan od najstarijih i najpoznatijih algoritama. On može biti rekurzivno definisan na sledeći način:

Neka je sa *gcd(m, n)* označen NZD brojeva *m* i *n*:

- Ako je *m % n* jednako *0*, onda je *gcd (m, n)* jednako *n*.
- U suprotnom, *gcd(m, n)* je jednako *gcd(n, m % n)*.

Nije uopšte teško dokazati tačnost ovog algoritma.

Pretpostavimo da je *m % n = r*. Stoga, *m = qn + r*, gde je *q* količnik od *m / n*. Bilo koji broj koji je delilac *m* i *n* mora biti i delilac broja *r*. Dakle,*gcd(m, n)* će biti isto kao i *gcd(n, r)*, gde je *r = m % n.*

```
public static int gcd(int m, int n) {
    if (m % n == 0)
       return n;
    else
       return gcd(n, m % n);
}
```

Program koji određuje NZD primenom ovog rekurzivnog algoritma je dat u nastavku:

```
import java.util.Scanner;
public class GCDEuclid {

  public static int gcd(int m, int n) {
    if (m % n == 0)
    return n;
    else
    return gcd(n, m % n);
  }

  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
```

```
    System.out.print("Unesi prvi broj: ");
    int m = input.nextInt();
    System.out.print("Unesi drugi broj: ");
    int n = input.nextInt();

    System.out.println("Najveci zajednicki delilac za " + m +
    " i " + n + " je " + gcd(m, n));
  }
}
```

Moguće izvršavanje programa:

```
Unesi prvi ceo broj: 2525 (ENTER)
Unesi drugi ceo broj: 125 (ENTER)
Najveci zajednici delilac za 2525 i 125 je 25
```

ANALIZA NAJBOLJEG I NAJGOREG SLUČAJA

U najboljem slučaju kada je m % n jednako 0 algoritam zahteva samo jedan korak da odredi NZD. U najgorem slučaju, vremenska kompleksnost gcd metoda je ustvari O(logn)

U najboljem slučaju, kada je *m % n* jednako *0*, algoritam zahteva samo jedan korak da odredi NZD.

Međutim, malo je teže analizirati prosečan slučaj. Kakogod, možemo dokazati da je vremenska složenost najgoreg slučaja reda O(logn).

Ako pretpostavimo da je *m>=n*, možemo pokazati da je *m % n < m / 2*, na sledeći način:

- Ako je *n <= m / 2*, onda je *m % n < m / 2*, pošto ostatak broja m podeljen sa *n* je uvek manji od *n*.
- Ako je *n > m / 2*, onda je *m % n = m – n < m / 2*. Stoga, *m % n < m / 2*.

Euklidov algoritam rekurzivno poziva metod *gcd*. On prvo poziva *gcd(m, n)*, zatim poziva *gcd(n, m % n)*, pa onda *gcd(m % n, n % (m % n))*, i tako dalje, kao što je prikazano u nastavku:

gcd(m, n)

= gcd(n, m % n)

= gcd(m % n, n % (m % n))

= ...

Pošto je *m % n < m / 2* i *n % (m % n) < n / 2*, argument prosleđen metodi *gcd* se redukuje za polovinu u svakoj sledećoj iteraciji. Nakon dva poziva metode *gcd*, drugi parametar je sigurno manji od n/2. Nakon 4 poziva metode *gcd*, drugi parametar postaje manji od n/4. Nakon šest poziva metode *gcd*, drugi parametar postaje manji od $n/2^3$. Neka sa *k* označimo broj poziva

metode *gcd*. Stoga, nakon k-tog poziva metode *gcd*, drugi parametar je manji od $n/2^{k/2}$ što je veće ili jednako 1. Stoga imamo:

$$\frac{n}{2^{k/2}} \geq 1 \Rightarrow n \geq 2^{k/2} \Rightarrow logn \geq k/2 \Rightarrow k \leq 2logn$$

Literatura:Liang D., Introduction to Java Programming, 12th edition, 2020.

PRIMER NAJGOREG SLUČAJA

Najgori slučaj nastaje kada za dva broja imamo najveći broj podela i provera. Ispostavlja se da ustvari dva susedna Fibonačijeva broja vode do najvećeg broja podela i provera

Najgori slučaj nastaje kada za dva broja imamo najveći broj podela i provera. Ispostavlja se da ustvari dva susedna Fibonačijeva broja vode do najvećeg broja podela i provera. Podsetimo se da Fibonačijeva serija počinje brojevima 0 i 1, i svaki sledeći član se dobija kao zbir prethodna dva, pa imamo sledeću seriju brojeva: 0 1 1 2 3 5 8 13 21 34 55 89 . . .

Ova serija može rekurzivno biti definisana kao

fib(0) = 0;

fib(1) = 1;

fib(index) = fib(index - 2) + fib(index - 1); index >= 2

Za dva susedna Fibonačijeva broja *fib(index)* i *fib(index - 1)* :

gcd(fib(index), fib(index - 1))

= gcd(fib(index - 1), fib(index - 2))

= gcd(fib(index - 2), fib(index - 3))

= gcd(fib(index - 3), fib(index - 4))

= ...

= gcd(fib(2), fib(1))

= 1

Primer:

```
gcd(21, 13)
= gcd(13, 8)
= gcd(8, 5)
= gcd(5, 3)
= gcd(3, 2)
```

```
= gcd(2, 1)
= 1
```

Vidimo da je broj poziva metoda ***gcd*** isti kao i indeks Fibonačijevog broja. Možemo dokazati da važi da je *index* ≤1.44 log*n* pri čemu je *n* = fib(index - 1). Ovo je mnogo jača granica nego *index* ≤log*n*.

Na Slici 1. 1 su sumirane složenosti tri algoritma za određivanje NZD dva broja:

Kompleksnost	Opis
$O(n)$	Brute force, provera svih mogućih delilaca
$O(n)$	Provera polovine mogućih delilaca
$O(\log n)$	Euklidov algoritam

Slika 1.1 Poređenje složenosti NZD algoritama [1]

Literatura: Liang D., Introduction to Java Programming, 12th edition, 2020.

Poglavlje 2

Fraktali

POSTAVKA PROBLEMA – TROUGAO SIERPINSKOG

Fraktal je geometrijski lik koji se može razložiti na manje delove tako da je svaki od njih, makar približno, umanjena kopija celine. Najpoznatiji fraktal je Trougao Sierpinskog

Fraktal je geometrijski lik koji se može razložiti na manje delove tako da je svaki od njih, makar približno, umanjena kopija celine.

Zadatak: Ako je data dimenzija *N* napisati program u Java-i koji definiše ***Trougao Sirepinskog*** reda *N*.

Postoji veliki broj interesantnih primera kada su u pitanju fraktali. Ovde ćemo opisati prost fraktal pod nazivom ***Trougao Sierpinskog*** (eng. Sierpinski triangle) koji je dobio ime po čuvenom poljskom matematičaru. Trougao Sierpinskog se može kreirati na sledeći način.

1. Poći od jednakostraničnog trougla, koji se smatra Trouglom Sierpinskog, reda (nivoa) *0*, kao što je prikazano na Slici 2. 1a .

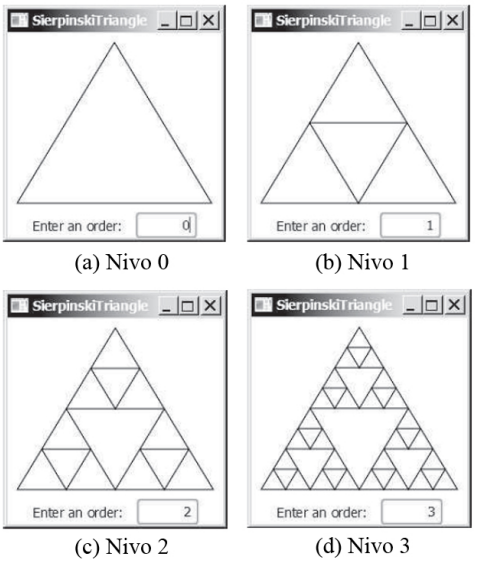
2. Spojiti središnje tačke stranica trougla reda nula i kreirati Trougao Sierpinskog reda *1* (Slika 2.1b).

3. Ostaviti centralni trougao netaknut. Spojiti linijama središnje tačke stranica ostala tri trouglai kreirati Trougao Sirepinskog reda *2* (Slika 2.1c).

4. Zatim rekurzivno ponoviti isti proces sa ciljem da se kreira Trougao Sierpinskog reda *3*, *4*, . . . , i tako dalje do *n-1* (Slika 2.1d).

Problem je po prirodi rekurzivan. Postavlja se pitanje kako razviti rekurzivno rešenje za njega?

Razmotrimo prvo osnovni slučaj reda *0*. Nesumnjivo trougao reda *0* je veoma lako nacrtati nacrtati (Slika 2.1a).



(a) Nivo 0 (b) Nivo 1

(c) Nivo 2 (d) Nivo 3

Slika 2.1 Trougao Sierpinskog kao šablon rekurzivnih trouglova [1]

IMPLEMENTACIJA ALGORITMA ZA KREIRANJE FRAKTALA

Problem je po prirodi rekurzivan. Iscrtavanje Trougla Sierpinskog reda n može biti svedeno na iscrtavanje tri Trougla Sierpinskog reda n-1

Trougao Sierpinskog reda *0* je veoma lako nacrtati (Slika 2.1a, prethodna sekcija). Kako sada kreirati trougao reda *1*?

Problem može biti sveden na crtanje 3 trougla reda *0*. Kako nakon toga kreirati Trougao Sierpinskog reda *2*?

Problem može biti sveden na iscrtavanje tri trougla reda *1*, pa stoga iscrtavanje Trougla Sierpinskog reda *n* može biti svedeno na iscrtavanje tri Trougla Sierpinskog reda *n-1*.

Naredni listing kreira Java FX aplikaciju koja prikazuje Trougao Sierpinskog bilo kod reda, kao što je prikazano na Slici 2. 1 (iz prethodne sekcije). Korisnik u polje za unos može da unese bilo koji red i nakon toga će biti kreiran i iscrtan Trougao Sierpinskog upravo tog reda:

```
import javafx.application.Application;
import javafx.geometry.Point2D;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Polygon;
import javafx.stage.Stage;
```

```
public class SierpinskiTriangle extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    SierpinskiTrianglePane pane = new SierpinskiTrianglePane();
    TextField tfOrder = new TextField();
    tfOrder.setOnAction(
      e -> pane.setOrder(Integer.parseInt(tfOrder.getText())));
    tfOrder.setPrefColumnCount(4);
    tfOrder.setAlignment(Pos.BOTTOM_RIGHT);

    // Podesi Pane koji ima label, text field, i button
    HBox hBox = new HBox(10);
    hBox.getChildren().addAll(new Label("Unesi red trougla: "), tfOrder);
    hBox.setAlignment(Pos.CENTER);

    BorderPane borderPane = new BorderPane();
    borderPane.setCenter(pane);
    borderPane.setBottom(hBox);

    // Kreiraj prikaz (scene) i postavi ga na scenu (stage)
    Scene scene = new Scene(borderPane, 200, 210);
    primaryStage.setTitle("SierpinskiTriangle"); // Setuj naziv scene
    primaryStage.setScene(scene); // Postavi prikaz na scenu (stage)
    primaryStage.show(); // Prikazi scenu

    pane.widthProperty().addListener(ov -> pane.paint());
    pane.heightProperty().addListener(ov -> pane.paint());
  }

  /** Pane za prikazivanje trouglova */
  static class SierpinskiTrianglePane extends Pane {
    private int order = 0;

    /** Setuj novi red trougla (order) */
    public void setOrder(int order) {
      this.order = order;
      paint();
    }

    SierpinskiTrianglePane() {
    }

    protected void paint() {
      // Izaberi tri tacke koje su u proporciji sa velicinom panela
      Point2D p1 = new Point2D(getWidth() / 2, 10);
      Point2D p2 = new Point2D(10, getHeight() - 10);
      Point2D p3 = new Point2D(getWidth() - 10, getHeight() - 10);

      this.getChildren().clear(); // Obrisi Pane pre reiscrtavanja

      displayTriangles(order, p1, p2, p3);
    }
```

```
    private void displayTriangles(int order, Point2D p1,
        Point2D p2, Point2D p3) {
      if (order == 0) {
        // Nacrtaj trougao pomocu tri tacke
        Polygon triangle = new Polygon();
        triangle.getPoints().addAll(p1.getX(), p1.getY(), p2.getX(),
          p2.getY(), p3.getX(), p3.getY());
        triangle.setStroke(Color.BLACK);
        triangle.setFill(Color.WHITE);

        this.getChildren().add(triangle);
      }
      else {
        // Pronadji sredisnu tacku na svakoj ivici trougla
        Point2D p12 = p1.midpoint(p2);
        Point2D p23 = p2.midpoint(p3);
        Point2D p31 = p3.midpoint(p1);

        // Rekurzivno prikazi tri manja trougla
        displayTriangles(order - 1, p1, p12, p31);
        displayTriangles(order - 1, p12, p2, p23);
        displayTriangles(order - 1, p31, p23, p3);
      }
    }
  }

  /**
   * Metod main() je potreban samo kod IDE-a sa ogranicenom
   * JavaFX podrskom. Nije potreban za pokretanje iz komandne linije.
   */
  public static void main(String[] args) {
    launch(args);
  }
}
```

ANALIZA SIERPINSKI ALGORITMA

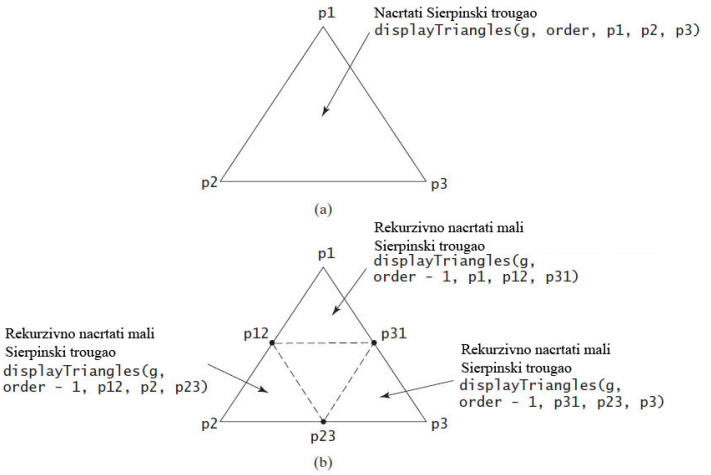
Rekurzivno pozivamo metod displayTriangles sa nivoom umanjenim za jedan, da bi prikazali tri manja Trougla Sierpinskog, čiji je nivo upravo taj nivo umanjen za jedan

Inicijalni (početni) trougao ima tri tačke koje su postavljene u prostoru proporcinalno veličini panela za iscrtavanje (linije 44–46). Ukoliko je *order == 0*, metod *displayTriangles(g, order, p1, p2, p3)* prikazuje trougao koji je dobijen spajanjem tačaka *p1*, *p2*, i *p3* u linijama 55–57, kao što je prikazano na Slici 2. 2a . U suprotnom, biće sprovedeni sledeći koraci:

1. Odrediti središnju tačku između *p1* i *p2* (linija 61), središnju tačku između *p2* i *p3* (linija 62), i središnju tačku između *p3* i *p1* (linija 63), kao što je prikazano na Slici 2.2b.

2. Rekurzivno pozvati metod *displayTriangles* sa nivoom umanjenim za jedan, da bi prikazali tri manja Trougla Sierpinskog, čiji je nivo upravo taj nivo umanjen za jedan (linije 66–68). Primetimo da je svaki manji Trougao Sierpinskog po strukturi identičan originalnom većem Trouglu Sierpinskog, osim što je nivo manjeg trougla za jedan manji od nivoa većeg trougla, kao što je prikazano na Slici 2. 2b .

Trougao Sierpinskog će biti prikazan u panelu *SierpinskiTrianglePanel*. Vrednost polja order u unutrašnjoj klasi SierpinskiTrianglePanel predstavlja nivo Trougla Sierpinskog. Klasa *Point* služi za definisanje koordinata tačaka. Metod *midpoint(Point p1, Point p2)* kao rezultat vraća središnju tačku duži određene tačkama p1 i p2 (linije 72–74).



Slika 2.2 Iscrtavanje Trougla Sierpinskog rekurzivnim pozivanjem tri nova metoda za iscrtavanja malih Trouglova Sierpinskog nižeg reda [1]

Poglavlje 3

Rekurzija i "Podeli i osvoji" strategija

MAKSIMALNA SUMA: PODELI I OSVOJI ALGORITAM

Niz delimo na dva dela i ispitujemo tri različita slučaja koji mogu da se jave: najveća sekvenca se nalazi isključivo u levoj ili desnoj polovini, ili počinje u levoj a završava u desnoj

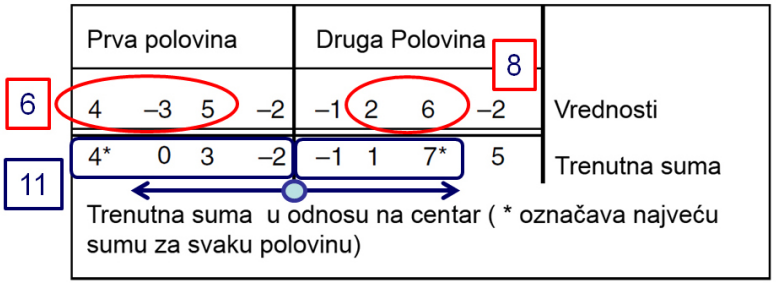
Analizirajmo ponovo problem maksimalne sume podsekvence niza koga smo prikazali u prošloj lekciji.

Razmotrimo algoritam koji se bazira na principu ***"podeli i osvoji"*** . Pretpostavimo da imamo sledeći ulazni niz {4, –3, 5, –2, –1, 2, 6, –2}. Podelimo ovaj niz na dva dela, kao što je prikazano na Slici 3.1 . Najveća suma kontinualne podsekvence može da se javi u tri slučaja:

Slučaj 1: Nalazi se isključivo u prvoj polovini niza.

Slučaj 2: Nalazi se isključivo u drugoj polovini niza.

Slučaj 3: Počinje u prvoj polovini i nastavlja se u drugoj polovini niza.



Slika 3.1 Deljenje problema najveće sume kontinualnih podsekvenci na dva dela [2]

Započnimo analizu sa slučajem 3:

Ono što želimo je da izbegnemo ugnježdene petlje koje rezultuju iz analize svih N/ 2 početnih tačaka i krajnjih tačaka N/2 nezavisno. To možemo uraditi zamenom dve ugnježdene petlje sa dve uzastopne petlje.

Uzastopne petlje, svaka veličine *N*/2, se kombinuju da ostvare isključivo posao linearne vremenske složenosti.

Mi možemo da izvršimo ovu zamenu jer: bilo koja kontinualna podsekvenca koja počinje u prvoj polovini i završava se u drugoj polovini mora da sadrži poslednji element prve polovine i prvi element druge polovine niza.

Slika 3. 1 pokazuje da za svaki element u prvoj polovini niza možemo da odredimo sume kontinualne podsekvence koja sadrži i poslednji član. To radimo pretragom s desna na levo, počevši od granice između dve polovine niza.

Slično tome, možemo da odredimo sume svih kontinualnih sekvenci u drugoj polovini koje sadrže prvi element s leva, i među njima odredimo najveću sumu.

Na kraju, možemo da spojimo ove dve podsekvence u najveću kontinualnu podsekvencu koja se prostire s obe strane granice.

U ovom primeru, rezultujuća podsekvenca se prostire od prvog elementa s leva u prvoj polovini do pretposlednjeg elementa u drugoj polovini. Ukupna suma je ustvari suma ove dve podsekvence a to je 4 + 7 = 11. Ova analiza pokazuje da slučaj 3 može biti izvršen za linearno vreme.

IMPLEMENTACIJA "PODELI I OSVOJI" ALGORITMA

Svaka polovina niza (leva ili desna) se zatim po istom principu deli ponovo na polovine, itd. Stoga je ovaj problem po intuiciji rekurentan i za njega se može kreirati rekurzivan algoritam

Slučaj 3 može biti izvršen za linearno vreme. Ali, šta je sa slučajevima 1 i 2?

S obzirom da postoji N/2 elemenata u svakoj polovini, primena algoritma iscrpne pretrage i dalje zahteva kvadratnu složenost za svaku od polovina. Kao poboljšanje, u slučajevima 1 i 2 možemo da primenimo istu ***„podeli i osvoji***" strategiju – podelimo polovinu na dve četvrtine. Zatim možemo da ove četvrtine delimo dalje sve dok deljenje postane nemoguće.

Primena prethodnog pravila će smanjiti vreme izvršavanja algoritma zbog čuvanja podrešenja tokom izvršavanja algoritma.

U nastavku je sumiran glavni deo ovog algoritma:

1. Rekurzivno odrediti najveću sumu kontinualne podsekvence koja se isključivo nalazi u prvoj (levoj) polovini niza.

2. Rekurzivno odrediti najveću sumu kontinualne podsekvence koja se isključivo nalazi u drugoj (desnoj) polovini niza

3. Odrediti, korišćenjem uzastopnih petlji maksimalnu sumu kontinualne podsekvence koja počinje u prvoj a završava se u drugoj polovini

4. Izabrati najveću od 3 izračunate sume.

Rekuzivni postupak računanja podniza sa maksimalnom sumom je dat u nastavku:

```
private static int maxSumRec( int [ ] a, int left, int right ) {
 int maxLeftBorderSum = 0, maxRightBorderSum = 0;
 int leftBorderSum = 0, rightBorderSum = 0;
 int center = ( left + right ) / 2;

 if( left == right )  // Osnovni slucaj
  return a[ left ] > 0 ? a[ left ] : 0;

  int maxLeftSum  = maxSumRec( a, left, center );
  int maxRightSum = maxSumRec( a, center + 1, right );

  for( int i = center; i >= left; i-- )  {
   leftBorderSum += a[ i ];
   if( leftBorderSum > maxLeftBorderSum )
    maxLeftBorderSum = leftBorderSum;
  }

  for( int i = center + 1; i <= right; i++ )   {
   rightBorderSum += a[ i ];
   if( rightBorderSum > maxRightBorderSum )
    maxRightBorderSum = rightBorderSum;
  }

  return max3( maxLeftSum, maxRightSum,
    maxLeftBorderSum + maxRightBorderSum );
}

private static int max3( int a, int b, int c ) {
 return a > b ? a > c ? a : c : b > c ? b : c;
}
```

ANALIZA "PODELI I OSVOJI" REKURENTNOSTI

Vreme izvršavanja „podeli i osvoji“ algoritma za određivanje maksimalne sume će biti O(NlogN)

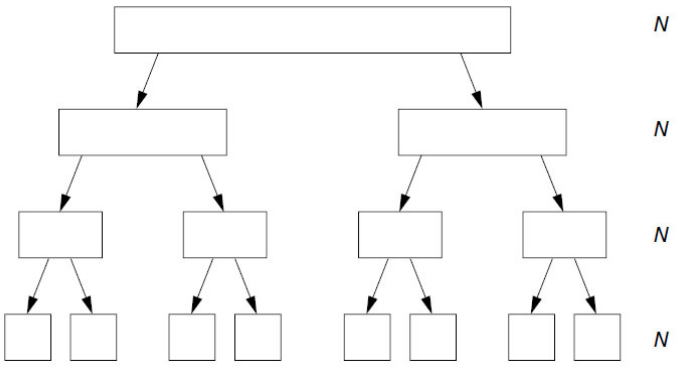
Rekurzivni algoritam maksimalne sume kontinualne podsekvence se izvršava obavljanjem linearnog algoritma za izračunavanje sume koja obuhvata i granicu dva domena, i zatim obavljanjem dva rekurzivna poziva. Ovi pozivi sve skupa izračunavaju sumu koja obuhvata centralnu granicu, obavljaju naredne rekurzivne pozive itd. Ukupan posao koji obavlja ovaj algoritam je tada proporcionalan ispitivanju koje se vrši nad svim rekurzivnim pozivima.

Slika 3. 2 grafički ilustruje kako algoritam radi za slučaj N = 8 elemenata. Svaki pravougaonik predstavlja poziv funkcije *maxSumRec*, a veličina pravougaonika je proporcionalna veličini podniza nad kojim se operiše u toku jednog rezurzivnog poziva.

Početni poziv je prikazan u prvoj liniji. Veličina podzniza je N što predstavlja troškove ispitivanja slučaja 3. Iz početnog poziva se zatim vrše dva rekurzivna poziva, svaki nad polovinom tekućeg podniza, pa se problemi svode na ispitivanje nizova dužina N/2. Vreme

izvršavanja slučaja 3 je polovina od troškova celog niza, ali s obzirom da ima dva takva rekurzivna poziva, kombinovani troškovi ovih rekurzivnih poziva su takođe veličine N.

Svaka od ove dve rekurzivne instance ponovo pravi dva rekurzivna poziva, pa tako sada imamo 4 podniza čija je veličina 4 puta manja od ukupne veličine niza. Stoga, ukupni troškovi slučaja 3 za sve do sada izvršene pozive je i dalje N.



Slika 3.2 Praćenje rekurzivnih poziva za rekurzivno rešenje problema najveće sume kontinualne podsekvecne, za slučaj kada je N=8 [2]

U jednom trenutku, dostižemo osnovni slučaj. Svaki osnovni slučaj je veličine 1, a ukupno ima N osnovnih slučajeva. Naravno, kod ovog slučaja nema troškova izvršavanja slučaja 3, ali trošimo vreme na proveru da li je taj element pozitivan ili negativan. Ukupni troškovi su stoga N za svaki korak rekurzije. U svakom koraku se veličina problema smanjuje na pola, pa po prirodi možemo zaključiti da imamo log N nivoa. U stvari, broj nivoa je 1 + [log N] (što je 4 kada je N jednako 8). Stoga očekujemo da će vreme izvršavanja algoritma biti O(N log N) .

Ova analiza nam daje intuitivno objašnjenje zašto je vreme izvršavanja algoritma O(N log N).

Poglavlje 4

Master teorema

OSNOVI MASTER TEOREME

Master teorema predstavlja osnovu za rešavanje rekurentnih jednačina koje se javljaju pri analizi mnogih podeli-pa-vladaj algoritama

Analiza iz prethodne sekcije je pokazala da se, kada je problem podeljen na dve jednake polovine koje se mogu rešiti rekurzivno - uz dodatni trošak od O(N), dobija algoritam reda O(N log N). Šta se dešava ako podelimo problem na tri jednake polovine, uz linearne dodatne troškove, ili na sedam jednakih polovina uz dodatne troškove kvadratne složenosti?

U ovoj sekciji ćemo dati opštu formula koja se koristi za izračunavanje vremena izvršavanja "podeli i osvoji algoritama", i ona se naziva Master teorema. Formula zahteva tri parametra:

- A, što predstavlja broj potproblema.
- B, što predstavlja relativnu veličinu potproblema (na primer, B = 2 predstavlja potproblem koji je jednak polovini od ukupne veličine)
- k, koeficijent koji predstavlja dodatni posao za svaki potproblem reda $O(N^k)$

Formula je predstavljena sledećom teoremom (Master teorema):

Teorema : $Rešenje jednačine\ T(n) = AT(N/B) + O(N^k), gde je A \geq 1 i B > 1, je$

$$T(N) = \begin{cases} O(N^{log_B A}) & for A > B^k \\ O(N^k log N) & for A = B^k \\ O(N^k) & for A < B^k \end{cases}$$

N/B ustvari predstavlja veličinu svakog potproblema (Ovde se pretpostavlja da su svi potproblemi u suštini iste veličine), dok $O(N^k)$ predstavlja cenu operacija obavljenih van rekurzivnih poziva (obuhvata operacije deljenja problema na potprobleme i spajanja rešenja dobijenih kao rešenja potproblema).

Dokaz ove formule zahteva poznavanje geometrijskih suma, ali poznavanje dokaza nije neophodno da bi se ova koristila.

Pogledajmo sada konkretnu primenu ove teoreme.

PRIMENA PRVOG SLUČAJA TEOREME

Master teorema mogućava da se na osnovnu broja i veličine potproblema,i količine dodatnog posla odredi opšta forma vremenske složenosti algoritma

Primer 1. Ako imamo problem koji se može podeliti na 3 potproblema veličine 1/2 ukupnog problema, uz linearni dodatni posao, onda imamo parametre A = 3, B = 2, i k = 1, pa se primenjuje prvi slučaj teoreme. Rezultat je $O(N^{\log_2 3}) = O(N^{1.59})$. U ovom slučaju, dodatni posao van rekurzije ne doprinosi ukupnoj vremenskoj složenosti algoritma. Bilo koji dodatni posao manji od $O(N^{1.59})$ će proizvesti isto vreme izvršavanja za rekurzivni algoritam.

Primer 2. Neka je data sledeća jednačina: $T(n) = 8T\left(\frac{n}{2}\right) + 1000n^2$. Iz ove jednačine se može zaključiti da promenljive uzimaju sledeće vrednosti:

$$a = 8,\ b = 2,\ f(n) = 1000n^2$$
$$f(n) = O\left(n^c\right), gde\ je\ c = 2$$

Dalje proveravamo da li je zadovoljen uslov prvog slučaja master teoreme:

$$\log_b a = \log_2 8 = 3, dakle : c < \log_b a$$

Zadovoljen je uslov, pa iz prvog slučaja master teoreme sledi:

$$T(n) = O\left(n^{\log_b a}\right) = O\left(n^3\right)$$

Prema tome, data rekurentna jednačina T(n) pripada $O(n^3)$. Ovaj rezultat se može potvrditi direktnim rešavanjem rekurentne jednačine: $T(n) = 1001n^3 - 1000n^2 T(n) = 1001n^3 - 1000n^2$, pod pretpostavkom da je $T(1) = 1 T(1) = 1$.

PRIMENA DRUGOG SLUČAJA TEOREME

Demonstrira se drugi oblik primene master teoreme u narednom primerima

Primer 1: Za problem maksimalne sume podniza, imamo dva problema, dve polovine i linearni dodatni posao u svakom potproblemu. Stoga, vrednosti koje primenjujemo u formuli su A = 2, B = 2, i k = 1. To znači da primenjujemo drugi slučaj teoreme, pa dobijamo da je vreme izvršavanja O(N log N), što se slaže sa izračunavanjem koju smo ranije sproveli.

Primer 2: Neka je data jednačina:

$$T(n) = 2T\left(\frac{n}{2}\right) + 10n$$

Iz prethodne jednačine promenljive uzimaju sledeće vrednosti:

$$a = 2,\ b = 2,\ c = 1,\ f(n) = 10n$$
$$f(n) = O\left(n^c \log n\right)\ gde\ je\ c = 1$$

Dalje proveravamo da li je zadovoljen uslov drugog slučaja master teoreme:

$$\log_b a = \log_2 2 = 1, dakle : c = \log_b a$$

Zadovoljen je uslov, pa iz drugog slučaja master teoreme sledi:

$$T(n) = \Theta\left(n^{\log_b a} \log n\right) = \Theta\left(n^1 \log n\right) = \Theta\left(n \log n\right)$$

Prema tome, data rekurentna jednačina T(n) pripada Θ(n log n). Ovaj rezultat se može potvrditi direktnim rešavanjem rekurentne jednačine:

$$T(n) = n + 10n \log_2 nT(n) = n + 10n \log_2 n,$$

pod pretpostavkom da je $T(1) = 1T(1) = 1$

PRIMENA TREĆEG SLUČAJA TEOREME

Demonstrira se treći oblik primene master teoreme u narednom primerima

Primer 1: Algoritam koji rešava tri potproblema veličine 1/2 originalnog problema, uz kvadratni dodatni posao, će zahtevati vreme O(N^2) jer se tada primenjuje treći slučaj teoreme. Naime, dodatni posao postaje dominantan kada jednom pređe prag od O($N^{1.59}$). Za definisanje praga, vodeći činilac je logaritamski faktor koji je prikazan u drugom slučaju.

Primer 2: Neka je data jednačina:

$$T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

Iz prethodne jednačine promenljive uzimaju sledeće vrednosti:

$$a = 2,\ b = 2,\ f(n) = n^2$$
$$f(n) = \Theta\left(n^c\right),\ gde\ je\ c = 2c = 2$$

Dalje proveravamo da li je zadovoljen uslov trećeg slučaja master teoreme:

$$\log_b a = \log_2 2 = 1, dakle : c > \log_b a$$

Zadovoljen je uslov, pa iz trćeg slučaja master teoreme sledi:

$$T\left(n\right) = \Theta\left(n^c\right) = \Theta\left(n^2\right).$$

Prema tome, data rekurentna jednačina T(n) pripada $\Theta(n^2)$, što je u skladu sa f (n) iz početne jednačine. Ovaj rezultat se može potvrditi direktnim rešavanjem rekurentne jednačine: $T(n) = 2n^2 - nT(n) = 2n^2 - n$, pod pretpostavkom da je $T(1) = 1T(1) = 1$.

Poglavlje 5

Rekurzija i dinamičko programiranje

PROBLEM VRAĆANJA KUSURA (ENG. CHANGE-MAKING PROBLEM)

Za odgovarajuću valutu sa novčićima od C1, C2, ... Cn (centi) koji je minimalni broj novčića neopodan da se kupcu vrati kusur od K centi?

Koji je minimalni broj novčića neophodan da se kupcu vrati kusur od K centi za odgovarajuću valutu sa novčićima od C_1, C_2,...,C_N (centi) ?

Američka valuta ima novčiće od 1, 5, 10 i 25 centi (novčić od 50 centi se retko koristi). Da bi vratili kusur od 63 centa, možemo da koristimo dva novčića od 25 centi, jedan od 10 i 3 novčića od 1 centa, što znači da smo ukupno iskoristili 6 novčića.

U slučaju da se valutni sistem sastoji iz novčića od 21 centi, pohlepni algoritam će dati rešenje od 6 novćića, ali optimalno rešenje u ovom novom slučaju bi bilo 3 novčića (tri od 21 centa).

Pitanje tada postaje: *Kako rešiti problem za proizvoljan skup novčića?*

Pretpostavimo da uvek imamo na raspolaganju novčić od 1 centa tako da rešenje uvek postoji. Jednostavna strategija formiranja kusura od *K* centi se rekurzivno može formulisati na sledeći način:

1. Ako možemo kreirati kusur koristeći tačno jedan novčić, to je i minimalni broj novčića.

2. U suprotnom, za svaku moguću vrednost *i* možemo izračunati minimalni broj novčića potrebnih da napravimo *i* centi kusura, i *K* - *i* centi kusura samostalno. Zatim izaberemo *i* koje minimizira broj iskorišćenih novčića.

Da vidimo kako možemo, korišćenjem prethodnog rekurzivnog razmišljanja, napraviti kusur od 63 centa.

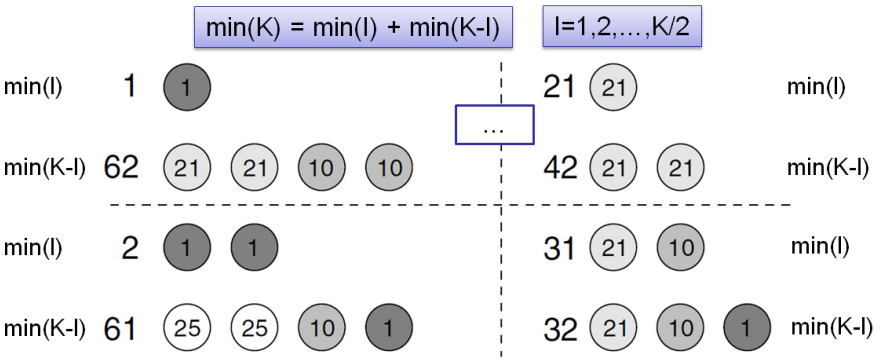
IMPLEMENTACIJA PRIRODNOG REKURZIVNOG ALGORITMA

Kod pohlepnog algoritma, u toku svake faze, odluka se donosi sa ciljem da bude optimalna, bez obzira na dalje posledice. Nažalost, pohlepni algoritam ne radi baš u svim slučajevima

Ako razmišljamo rekurzivno, možemo nezavisno izračunati broj potrebnih novčića za kusur od 1 cent, a onda i za ostatak od 62 centi (Slika 5.1). To su 1 i 4 novčića, redom.

Ove rezultate dobijamo rekurzivno, tako da oni moraju biti uzeti kao optimalni (dobićemo da se kusur od 62 centa optimalno formira pomoću dva novčića od 21 cent i dva od 10 centi). Kao rešenje imamo pet novčića.

Ako podelimo problem na 2 centa i 61 centi, rekurzivni algoritam će dobiti rešenje 2 i 4, redom, pa ćemo imati ukupno 6 novčića. Nastavljamo da isprobavamo sve mogućnosti, od kojih su neke prikazane na Slici 5. 1.



Slika 5.1 Neki od rekurzivnih podproblema kod problema vraćanja kusura [2]

Na kraju imamo slučaj da je ukupan kusur od 63 centa podeljen na delove od 21 i 42 centa, što se može predstaviti pomoću jednog odnosno 2 novčića, što je ukupno tri novčića.

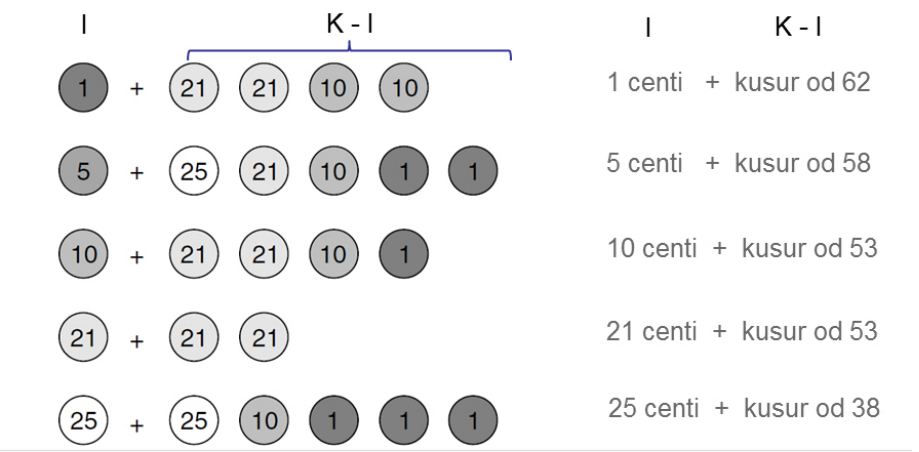
Poslednja podela koju treba isprobati je 31 i 32 centa. Mi možemo kusur od 31 cent formirati pomoću 2 novčića, a kusur od 32 centa pomoću 3 novčića, što u zbiru daje 5. Naravno, minimalni broj i dalje ostaje tri novčića. Ponovo svaki od ovih problema rešimo rekurzivno.

Ovaj algoritam zahteva dosta ponavljanja i suvišnih poslova, i neće se završiti u razumnom vremenu za slučaj kusura od 63 centa

KAKO POBOLJŠATI REKURZIVNI ALGORITAM?

Alternativni algoritam obuhvata smanjenje problema rekurzivno navodeći jedan od novčića koji koristimo. Pošto imamo 5 različitih novčića u valutnom sistemu imaćemo i 5 rekurzivnih poziva

Alternativni algoritam obuhvata smanjenje problema rekurzivno navodeći jedan od novčića koji koristimo. Na primer, kusur od 63 centa možemo formirati na neki od sledećih načina (Slika 5.2):



Slika 5.2 Alternativni rekurzivni algoritam za problem vraćanja kusura [2]

- Novčić od 1 centi , plus kusur od 62 centa rekurzivno određen
- Novčić od 5 centi, plus kusur od 58 centa rekurzivno određen
- Novčić od 10 centi, plus kusur od 53 centa rekurzivno određen
- Novčić od 21 centi, plus kusur od 42 centa rekurzivno određen
- Novčić od 25 centi, plus kusur od 38 centa rekurzivno određen

Pomoću ovog uprošćenja dobijamo samo 5 rekurzivnih poziva, po jedan za svaki mogući novčić. Opet, naivna rekurzivna implementacija je i dalje vrlo neefikasna, jer više puta određuje kusur za iste vrednosti.

Na primer, u prvom slučaju ostaje nam da rešimo kusur od 62 centa. U ovom potproblemu, jedan od rekurzivnih poziva bira 10 centi a zatim rekurzivno rešava podproblem od 52 centa. U trećem slučaju ostaje nam 53 centa. Jedan od njegovih rekurzivnih poziva uklanja novčić od 1 cent a zatim rešava problem od 52 centa koji smo već imali u prvom slučaju.

I ovde imamo suvišan posao koji opet dovodi do prekomernog vremena izvršavanja algoritma.

PRIMENA DINAMIČKOG PROGRAMIRANJA

Možemo primeniti pristupe odozgo na dole, i odozdo na gore kako bi smo rešili problem primenom dinamičkog programiranja

Da bi napravili algoritam koji radi razumnom brzinom treba da sačuvamo rešenja potproblema u niz. Ukupno rešenje će zavisiti isključivo od rešenja manjih potproblema. Prvi način je princip ***"odozgo na dole"*** , gde uvodimo niz *used [change]* u kome pamtimo koji je minimalni broj novčića za kreiranje kusura *change*.

```
public static int makeChange( int [ ] coins, int change) {
    if (used[change ] > 0) return used[change ];
    int minCoins = change;

    for( int j = 0; j < coins.length; j++ )
        if( coins[ j ] == change )
```

```
            return 1;

    for( int i = 1; i <= coins.length; i++) {
        int thisCoins = coins[ i ] +
            makeChange( coins, change - coins[ i ] );

        if( thisCoins < minCoins )
            minCoins = thisCoins;
    }
    used [change ] = minCoins;
    return minCoins;
}
```

Drugi način je pristup ***"odozdo na gore",*** i njegovo rešenje je efikasnije. Pogledajmo kod:

```
public static void makeChange( int [ ] coins, int differentCoins,
        int maxChange, int [ ] coinsUsed, int [ ] lastCoin ) {
    coinsUsed[ 0 ] = 0; lastCoin[ 0 ] = 1;

    for( int cents = 1; cents <= maxChange; cents++ ) {
        int minCoins = cents;
        int newCoin  = 1;

        for( int j = 0; j < differentCoins; j++ ) {
            if( coins[ j ] > cents ) // Ne mozemo koristi novcic j
                continue;
            if( coinsUsed[ cents - coins[ j ] ] + 1 < minCoins ) {
                minCoins = coinsUsed[ cents - coins[ j ] ] + 1;
                newCoin  = coins[ j ];
            }
        }

        coinsUsed[ cents ] = minCoins;
        lastCoin[ cents ]  = newCoin;
    }
}
```

Kod ovog pristupa se problem svodi na to da odredimo način kako da predstavimo kusur od 1 cent, a zatim 2 centa, pa 3 centa, itd.

ANALIZA PRISTUPA "ODOZDO NA GORE"

Korišćenjem principa dinamičkog programiranja mi čuvamo rešenja potproblema u niz. Problem se onda svodi na to da odredimo način kako da predstavimo kusur od 1,2,3... centi

Analizirajmo prethodni kod. Niz *LastCoin* se koristi da nam kaže koji novčić je poslednji put korišćen da se kreira optimalan kusur. U suprotnom, mi pokušavamo da kreiramo broj centi za koji možemo da vratimo kusur, za vrednosti u centima od 1 do *maxChange*. Da bi kreirali takav kusur, isprobavamo svaki novčić kao što je prikazano u

for(int j = 0; j < differentCoins; j++)

iskazu:

- Ako je suma koju dobijemo korišćenjem trenutno izabranih novčića veća od ukupnog kusura koji treba da vratimo onda prekidamo dalju pretragu za taj slučaj.
- U suprotnom, vršimo ispitivanja sa ciljem da odredimo da li je broj novčića koje koristimo da kreiramo kusur potproblema plus jedan novčić za zbir do *maxChange* manji od onog koji smo do tada dobili za postavljeni potproblem; Ako je tako, vršimo ispravku.

Kada se petlja završi za trenutni broj centi koji rešavamo, vrednost minimalnog broja novčića se unosi u član niza *coinsUsed* koji odgovara toj vrednosti kusura. Na kraju algoritma, *coinsUsed[i]* predstavlja minimalan broj potrebnih novčića da bi se kreirao kusur od *i* centi (*i == maxChange* je posebno rešenje koje tražimo).

int [ ] coins = { 1, 5, 10, 21, 25 }; min(K) = I + min(K-I) I=1,5,10,21,25

cents	1	2	3	4	5	6	7	8	9	10
coinsUsed[i]	1	2	3	4	1	2	3	4	5	1
lastCoin[i]	1	1	1	1	5	1	1	1	1	10

cents	11	12	13	14	15	16	17	18	19	20
coinsUsed[i]	2	3	4	5	2	3	4	5	6	2
lastCoin[i]	1	1	1	1	5	1	1	1	1	10

cents	21	22	23	24	25	26	27	28	29	30
coinsUsed[i]	1	2	3	4	1	2	3	4	5	2
lastCoin[i]	1	1	1	1	25	1	1	1	1	5

Slika 5.3 Postupak odredjivanja minimalnog kusura za prvih 30 brojeva [izvor: autor].

Ako pratimo kako se menjaju članovi niza *lastCoin*, možemo da zaključimo koji su novčići korišćeni da bi se postiglo rešenje (Slika 5.3). Na primer, za broj 15 *lastCoin* je 5. Možemo da oduzmemo 15-5 i dobijemo 10. Iz tabele vidimo da je za broj 10 *lastCoin* = 10. Kada oduzmemo 10-10 dobijamo 0, i završilo smo proveru novčića. Znači, za broj 15 koristili smo 10 i 5.

Vreme rada algoritma je ustvari vreme izvršavanja dve ugnježdene petlje i stoga O(NK), gde je N broj različitih novčića valutnog sistema a K je vrednost kusura koji pokušavamo da kreiramo.

Poglavlje 6

Vežbe

ZADATAK 1 (10 MIN)

Cilj zadatka je primena rekurzije kod matematičkih sumiranja

Zadatak 1. Napisati program koji računa zbir prvih N brojeva korišćenjem rekurzivne funkcije. Matematički je suma prvih n brojeva predstavljena jednačinom (10 min):

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n$$

Opšti oblik matematičke jednačine koja služi za rešavanje sume prvih N brojeva korišćenjem rekurzije se može napisati na sledeći nacin:

$$\sum_{i=1}^{n} i = n + \sum_{i=1}^{n-1}$$

```
public class RecSuma {
    // Izracunati zbir prvih N celih brojeva - rekurzivno
    public static long sum_rec( int n ) {
        if( n == 1 )
            return 1;
        else
            return sum_rec( n - 1 ) + n;
    }

 // Izracunati zbir prvih N celih brojeva - iterativno
    public static long sum_iter( int n ) {
     long sum_iter = 0;
     for(int i = 1; i <= n; i++)
      sum_iter += i;
     return sum_iter;
    }

    // Testiranje programa
    public static void main( String [ ] args ) {
        for( int i = 1; i <= 10; i++ )
            System.out.println( sum_rec( i ) );
        System.out.println( sum_rec( 8882 ) );
        System.out.println( sum_iter( 8882 ) );
    }
}
```

ZADATAK 2 (5 MIN)

Cilj zadatka je primena rekurzije kod brojanja pojavljivanja određenih karaktera u stringu

Zadatak 2 (5 min). Napisati rekurzivni metod koji određuje broj pojavljivanja odgovarajućeg slova u stringu, pri čemu metod ima zaglavlje:

```
public static int count(String str, char a)
```

Na primer, *count("Welcome", 'e')* vraća rezultat 2. Napisati test program koji od korisnika zahteva da unese string i karakter, a zatim određuje i prikazuje broj pojavljivanja datog karaktera u unetom stringu.

```
import java.util.Scanner;

public class Zadatak4 {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Unesi string: ");
    String s = input.nextLine();
    System.out.print("Unesi karakter: ");
    char ch = input.nextLine().charAt(0);
    int times = count(s, ch);
    System.out.println(ch + " appears " + times +
        (times > 1 ? " times " : " time ") + "in " + s);
  }

  public static int count(String str, char a) {
    int result = 0;
    if (str.length() > 0)
    result = count(str.substring(1), a) +
        ((str.charAt(0) == a) ? 1 : 0);

    return result;
  }
}
```

ZADATAK 3 I 4 (20 MIN)

Cilj zadataka je primena rekurzije kod brojanja velikih slova u stringu i nizu karaktera

Zadatak 3 (10 min) (*Određivanje broja velikih slova u stringu*) Napisati rekurzivni metod koji kao rezultat vraća broj velikih slova u stringu. Napisati i test program koji od korisnika zahteva da unese string, a zatim ispisuje broj velikih slova stringa.

```
import java.util.Scanner;

public class Zadatak5 {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Unesi string: ");
    String s = input.nextLine();
    System.out.println("Broj velikih slova u " +
        s + " je " + countUppercase(s));
  }

  public static int countUppercase(String str) {
    return countUppercase(str, str.length() - 1);
  }

  public static int countUppercase(String str, int high) {
    if (high < 0)
    return 0;
    else
    return countUppercase(str, high - 1) +
        (Character.isUpperCase(str.charAt(high)) ? 1 : 0);
  }
}
```

Zadatak 4 (10 min) (*Određivanje broja velikih slova u nizu*) Napisati rekurzivni metod koji kao rezultat vraća broj velikih slova u nizu karaktera. Neophodno je da kreirate sledeća dva metoda. Drugi je pomoćni rekurzivni metod

public static int count(***char***[] chars)

public static int count(***char***[] chars, ***int*** high)

Napisati i test program koji od korisnika zahteva da unese listu karaktera u jednoj liniji, a zatim ispisuje broj velikih slova u listi karaktera.

```
public class Zadatak6 {
  public static void main(String[] args) {
    System.out.println(count(new char[]{'W', 'e', 'l', 'c', 'o', 'm', 'E'}));
  }

  public static int count(char[] chars) {
    return count(chars, chars.length - 1);
  }

  public static int count(char[] chars, int high) {
    if (high >= 0) {
    return count(chars, high - 1) +
        (Character.isUpperCase(chars[high]) ? 1 : 0);
    }
    else
    return 0;
  }
}
```

ZADATAK 5 (10 MIN)

Cilj zadatka je primena rekurzije sa logičkim rezultatom

Zadatak 5 (10 minuta) Napisati rekurzivni metod koji određuje da li je suma cifara broja parna.

U slučaju da se prosledi 0, vraćamo true, jer je 0 paran broj. Kroz rekurzivne korake uklanjamo cifre iz broja pomoću celobrojnog deljenja. Kada imamo poslednju cifru (podatak da li je parna - a), i sumu ostatka broja (podatak da li je suma parna - b) možemo da proverimo da li će ukupna suma biti parna ili neparna. Ako imamo paran i neparan broj, onda će suma biti neparna. Ukoliko su oba parna ili oba neparna, onda je suma parna.

```
public class Rekurzija {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println(parnaSuma(123));
        System.out.println(parnaSuma(23));
        System.out.println(parnaSuma(5));
        System.out.println(parnaSuma(2));
    }

    public static boolean parnaSuma(int n) {

        if (n == 0) {
            return true;
        }

        int cifra = n % 10;
        boolean a = cifra % 2 == 0;
        boolean b = parnaSuma(n / 10);

        return a && b || !a && !b;

    }

}
```

Poglavlje 7

Zadaci za samostalni rad

ZADACI ZA SAMOSTALNO VEŽBANJE (75 MIN)

Na osnovu materijala sa predavanja i vežbi uraditi samostalno sledeće zadatke:

Zadatak 1 (10 min). Napisati rekurzivni algoritam koji određuje zbir sledeće serije brojeva:

$$m(i) = \frac{1}{3} + \frac{2}{5} + \frac{3}{7} + \frac{4}{9} + \frac{5}{11} + \frac{6}{13} \ldots + \frac{i}{2i+1}$$

Napisati test progam koji prikazuje sumu m(i) za i = 1, 2, . . . , 10.

Zadatak 2 (10 min). (Štampanje karaktera stringa u obrnutom poretku) Napisati rekurzivni metod koji prikazuje string u inverznom poretku u konzoli, korišćenjem sledećeg zaglavlja:

```
public static void reverseDisplay(String value)
```

Na primer, reverseDisplay("abcd") će prikazati rezultat dcba. Napisati test program koji od korisnika zahteva da unese string a zatim ga prikazuje u inverznom poretku.

Zadatak 3 (15 min). (Trougao Sierpinskog) Preraditi program sa predavanja tako da se kreira applet koji omoguća korisniku da pomoću dugmića + i – uvećava i umanjuje trenutni red fraktala za 1, kao što je prikazano na slici 1-a. Inicijalni red je 0. Ukoliko je trenutno red 0 onda pritisak na dugme - nema efekta.

Zadatak 4 (10 min). (*Permutacije stingova*) Napisati iterativni i rekurzivni metod koji štampa sve permutacije stringova. Na primer, za string **abc**, biće odštampano:

abc, acb, bac, bca, cab, cba

Zadatak 5 (15 min). Napisati rekurzivnu funkciju koja ispituje da li je uneti string palindrom. Testirati rad funkcije u glavnom programu.

Zadatak 6 (15 min). Napisati rekurzivnu funkciju koja štampa sve permutacije brojeva od 1 do 4. Npr, 1234 je jedna permutacija, 2134 druga, itd... Testirati rad funkcije u glavnom programu. Kako bi implementirali funkciju koja radi sa N različitih cifara (N < 9)?

Zadatak 7. (15 min) (Igra: Osam kraljica) Problem sa osam kraljica je pronaći rešenje za postavljanje dame u svaki red na šahovskoj tabli tako da dve dame ne mogu da napadnu jedna drugu. Napišite program za rešavanje problema Osam kraljica pomoću rekurzije.

NOVČANI SISTEM (15 MIN)

Na osnovu materijala sa predavanja i vežbi uraditi samostalno sledeći zadatak

Zadatak 8 (15 min). Dat je novčani sistem sa N novčanica, čije su vrednosti d[1], d[2], ... , d[N]. Napisati program koji ispituje da li je moguće isplatiti sumu M pomoću datog sistema. U slučaju kada je sumu moguće ispatiti štampati i način isplate.

Input> U prvom redu ulaza nalaze se dva prirodna broja N i M, broj novčanica i tražena suma, redom. U naredom redu nalazi se N prirodnih brojeva, koji označavaju vrednosti novčanica.

Output> U prvom izlaza štampati ’DA’ ukoliko je moguće isplatiti datu sumu, odnosno ’NE’ ukoliko nije. U slučaju da je moguće isplatiti datu sumu, u drugom redu ispisati N brojeva, C[1], ..., C [N], dovojenih jednim znakom razmaka, tako da je

C [1] * d [1] + ... + C [N] * d [N] = M.

Način isplate nije jedinstven, drugim rečima i rešenja sa isplatom "1 0 2" i "0 4 0" su korektna.

Ako se unesu brojevi 3, 12 - odgovor je DA,

Unosi se > 2 3 5 - odgovor je 2 1 1

Poglavlje 8

Domaći zadatak

DOMAĆI ZADATAK - PRAVILA

Pravila za izradu domaćeg zadatka

Svaki student dobija od asistenta sopstvenu kombinaciju domaćeg zadatka.

Online studenti bi trebalo mail-om 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 folderCS203-DZ04-Ime-Prezime-BrojIndeksa. Zipovani folder CS203-DZ04-Ime-Prezime-BrojIndeksa poslati predmetnom asistentu (lazar.mrkela@metropolitan.ac.rs) u mejlu sa naslovom (subject)CS203-DZ04, inače se neće računati.

Studenti iz Niša predispitne obaveze predaju asistentu 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:

Rekurzija u programiranju označava situaciju kada metod, procedura ili funkcija poziva samu sebe. Poziv može biti direktan ili indirektan. Svaki rekurzivni poziv mora na kraju dovesti do slučaja zaustavljanja inače dolazi do beskonačne rekurzije.

Primenom rekurzije mogu se rešiti mnogo problemi. Jedan od njih je Euklidov algoritam za određivanje NZD dva broja. Dalje, postoje problemi koji se veoma teško mogu rešiti bez korišćenja rekurzije. Takav od primera je određivanje veličine direktorijuma.

Fraktal je geometrijski lik koji se može razložiti na manje delove tako da je svaki od njih, makar približno, umanjena kopija celine. Najpoznatiji fraktal je Trougao Sierpinskog. Ovaj problem je u prirodi rekurzivan i najlakše se može rešiti rekurzijom.

Jedna veoma važna tehnika za rešavanje problema koja koristi rekurziju je "***podeli i osvoji***" . ***“Podeli i osvoji”*** algoritam je efikasan rekurzivni algoritam koji se sastoji od dva dela: podeli (podela na dva dela, odnosno dva podproblema koja se zatim rešavaju zasebno, i rekurzivno), i osvoji (konačno rešenje se dobija spajanjem rešenja podproblema).

Rekurzija nekad može da dovede do problema. Da bi smo izbegli te probleme, često radije koristimo principe ***dinamičkog programiranja*** , gde rekurzivni algoritam pretvaramo u nerekurzivni koji sistematski čuva rešenja podproblema u tabeli.

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] Nenad Filipovic, Algoritmi i strukture podataka, Masinski fakultet u Kragujevcu, 2010.

[4] T H. Cormen, C E. Leiserson, R L. Rivest, C. Stein, Introduction to Algorithms, 3rd edition, The MIT Press, 2009.

[5] https://www.geeksforgeeks.org/backtracking-set-1-the-knights-tour-problem/

[6] https://en.wikipedia.org/wiki/Knight%27s_tour#Computer_algorithms