Iterativni i rekurzivni postupci

Uvod

- Sastavni delovi svakog programa su **algoritmi** i **strukture podataka**
- **Algoritam** je diskretan, jednoznačan, čisto mehanički postupak, koji je definisan konačnim brojem pravila i koji dovodi do rešenja svakog konkretnog problema iz nekog skupa problema u konačno mnogo koraka
- Reč “algoritam” potiče od latiniziranog imena arapskog matematičara Muhameda ibn-Muse al-Hvarizmija koji je živeo u IX veku i između ostalog napisao knjigu o “računu Indijaca”, tj. o računanju sa arapskim ciframa
- Prvobitno je reč “algoritam” označavala pravila za sabiranje, oduzimanje, množenje i deljenje, a kasnije se pojam algoritma proširio da obuhvata sve mehaničke postupke

Uvod

- Jedini sačuvani latinski prevod odlomka te knjige počinje rečima “Dixit Algorismi...” (Algorizmi je rekao...), po čemu je i današnji pojam algoritam dobio ime

- Svi programi koji se izvode na računarima su realizacija algoritama - da bi se problem mogao rešiti na računaru mora da bude algoritamski rešiv

- Ukoliko neki problem ima algoritamsko rešenje, tada obično ima više različitih načina za njegovo rešavanje pa se prirodno postavlja pitanje koji je od mogućih algoritama za rešavanje problema bolji

Uvod

Kriterijumi po kojima se algoritmi razlikuju, i po kojima se može vršiti izbor pravog algoritma za dati problem, su sledeći:

- **Korektnost:** algoritam treba da ispravno reši postavljeni problem. Ovaj uslov zahteva da problem bude dovoljno precizno postavljen da bismo uopšte bili u stanju da analiziramo da li je problem rešen korektno
- **Količina memorije** potrebna za rešavanje problema. Ovaj kriterijum vremenom gubi na značaju jer je memorija postala dovoljno jeftina, tako da se često ne isplati gubiti vreme na skraćivanje programa ili optimizaciju potrošene memorije na uštrb jasnoće koda
- **Jasnoća i jednostavnost.** Ovaj kriterijum sve više dobija na značaju, jer se danas potencira zahtev da se problem što pre reši na što jasniji način, te da to rešenje bude što shvatljivije i lako za izmene i prilagođavanja
- **Efikasnost.** Pretpostvalja se da je efikasniji algoritam bolji, a da bi se definisalo šta se pod efikasnim algoritmom podrazumeva, uvodi se pojam složenosti algoritma. Kao mera složenosti uzima se broj koraka koji je potrebno izvesti da bi se do rešenja došlo. Broj koraka za rešavanje jednog problema najčešće zavisi i od veličine početnih podataka pa se broj koraka izražava kao funkcija veličine polaznih podataka

Rekurentni nizovi jedne promenljive

- U mnogim oblastima primene računara javlja se potreba za izračunavanjem elemenata matematičkog niza
- Jedna vrsta matematičkih nizova pogodnih za izračunavanje su rekurentni nizovi, tj. nizovi $\{s_n\}$, gde je n-ti element niza, s_n, definisan preko prethodnih elemenata niza (s_{n-1}, s_{n-2}, ...)
- Posmatraćemo prvo najjednostavniji slučaj kada je element niza definisan preko tačno jednog prethodnog:

$$s_n = \begin{cases} a, & n = 0 \\ f(s_{n-1}), & n \geq 1 \end{cases}$$

  gde funkcija f može da ima još neke parametre koji zavise od n
- Zbog toga što elementi niza zavise od prethodnih kaže se "**rekurentni**", a za gornji slučaj pošto element zavisi od tačno jednog prethodnog kaže se "**jedne promenljive**"

Rekurentni nizovi jedne promenljive

- Indekse (matematičkog) niza predstavićemo Javinim tipom `int`, a elemente tipom `double`
- Funkciju *f* za računanje sledećeg elementa niza predstavićemo metodom:
  `static double f(double stari, int n)`
- Metod `f` će za dati indeks `n` i prethodni element niza s_{n-1} prosleđen u parametru `stari` vratiti izračunatu vrednost novog elementa s_n
- Moguće je da metod `f` osim indeksa `n` koristi i još neke dodatne parametre koji su zbog jednostavnosti izostavljeni

Rekurentni nizovi jedne promenljive

- Uz navedene pretpostavke dobija se sledeći metod za računanje elementa niza:

```
static double s(int n, double pocetnaVrednost) {
  double tekucaVrednost = pocetnaVrednost;
  for (int i = 1; i <= n; i++) {
    tekucaVrednost = f(tekucaVrednost, i);
  }
  return tekucaVrednost;
}
```

Rekurentni nizovi jedne promenljive

- Sledećim programskim fragmentom ispisuje se vrednost izračunatog elementa niza:

```
System.out.print("Unesite n: ");
int n = Svetovid.in.readInt();
System.out.print("Unesite pocetnu vrednost: ");
double a = Svetovid.in.readDouble();
System.out.println("s(n) = " + s(n, a));
```

Primer: niz parnih prirodnih brojeva

- Niz parnih prirodnih brojeva može se definisati na sledeći način:

$$s_n = \begin{cases} 0, & n = 0 \\ s_{n-1} + 2, & n \geq 1 \end{cases}$$

- Lako se vidi da je $s_0 = 0$, $s_1 = 2$, $s_2 = 4$, ...

- Kad se primeni opisana metodologija dobija se sledeći program za računanje *n*-tog elementa ovog niza

Primer: niz parnih prirodnih brojeva

```
class Parni {
  static double f(double stari, int n) {
    return stari + 2;
  }
  static double s(int n, double pocetnaVrednost) {
    double tekucaVrednost = pocetnaVrednost;
    for (int i = 1; i <= n; i++) {
      tekucaVrednost = f(tekucaVrednost, i);
    }
    return tekucaVrednost;
  }
  public static void main(String[] args) {
    System.out.print("Unesite n: ");
    int n = Svetovid.in.readInt();
    System.out.println("s(n) = " + s(n, 0));
  }
}
```

Primer: binomni koeficijenti

- Prethodni primer je jednostavan i ne ilustruje punu snagu pristupa preko rekurentnih nizova, jer postoji trivijalan odnos između indeksa i elementa niza: $s_n = 2n$
  - Stoga je u ovom slučaju pristup preko rekurentnih nizova previše komplikovan i neefikasan
- Zato ćemo obraditi i malo složeniji primer gde se bolje vide prednosti ovakvog pristupa - računanje vrednosti binomnih koeficijenata
- Za date prirodne brojeve *n* i *k*, binomni koeficijent je broj koji označava koliko različitih podskupova od *k* elemenata je moguće izdvojiti iz skupa od *n* elemenata
  - Standardna oznaka: $\binom{n}{k}$
- Formula po kojoj se računa binomni koeficijent:

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

Primer: binomni koeficijenti

- Vrednost binomnog koeficijenta se može izračunati na jednostavan način: izračunaju se vrednosti svih faktorijela, pa se pomnože/podele
- Međutim, ovaj "naivni" pristup ima jedan veliki nedostatak
  - Faktorijeli su veliki brojevi, pa lako dolazi do prekoračenja opsega brojevnih tipova pri njihovom računanju
  - S druge strane, binomni koeficijenti nisu toliko veliki brojevi kao faktorijeli, i bilo bi poželjno izbeći rad sa velikim brojevima
- Zato ćemo binomne koeficijente posmatrati kao rekurentni niz, gde će nam n biti fiksan broj, a indeksi niza biće po k, tako da će element niza biti:

$$s_k = \binom{n}{k}$$

- Preostalo je da odredimo:
  - Početnu vrednost rekurentnog niza, s_0
  - Funkcionalnu zavisnost (f) između elemenata s_k i s_{k-1} za $k \geq 1$

Primer: binomni koeficijenti

- Početna vrednost:

$$s_0 = \binom{n}{0} = 1$$

- Funkcionalnzu vezu s_k i s_{k-1} je najjednostavnije naći preko količnika s_k / s_{k-1} :

$$\frac{s_k}{s_{k-1}} = \frac{\frac{n!}{k!\,(n-k)!}}{\frac{n!}{(k-1)!\,(n-k+1)!}} = \frac{n-k+1}{k}$$

tako da je $s_k = s_{k-1}(n-k+1)/k$

Primer: binomni koeficijenti

- Pre nego što pređemo na implementaciju računanja, obratimo pažnju na nekoliko faktora specifičnih za binomne koeficijente
- Prethodna izvođenja pošla su od pretpostavke da je $k \leq n$. Ako je $k > n$ tada je binomni koeficijent po definiciji 0
- Pri izračunavanju elemenata rekurentnog niza može se smanjiti broj iteracija, ako se uzme u obzir da je $s_k = s_{n-k}$ (za $0 \leq k \leq n$)
  - Tako da računamo $s_{\min(k, n-k)}$
- Ako je $n < 0$ ili $k < 0$ binomni koeficijent nije definisan, što ćemo signalizirati vraćanjem vrednosti -1 iz metoda

Primer: binomni koeficijenti

```
static double f(double stari, int n, int k) {
  return stari * (n - k + 1) / k;
}
static double bk(int n, int k) {
  if (n >= 0 && k >= 0) {
    if (k > n) {
      return 0;
    }
    else {
      if (k > n - k) k = n - k;
      double tekucaVrednost = 1;
      for (int i = 1; i <= k; i++) {
        tekucaVrednost = f(tekucaVrednost, n, i);
      }
      return tekucaVrednost;
    }
  }
  else {
    return -1;
  }
}
```

Sume i proizvodi

- Važan specijalan slučaj rekurentnih nizova su sume i proizvodi:

$$S = \sum_{k=1}^{n} a_k \quad \text{se definiše sa} \quad s_n = \begin{cases} 0, & n = 0 \\ s_{n-1} + a_n, & n > 0 \end{cases}$$

$$P = \prod_{k=1}^{n} b_k \quad \text{se definiše sa} \quad p_n = \begin{cases} 1, & n = 0 \\ p_{n-1} \cdot b_n, & n > 0 \end{cases}$$

- Predstavićemo prvo dva rekurzivna načina da se izračuna vrednost sume (računanje proizvoda je analogno)

- Pretpostavimo da je definisan metod koji računa vrednost sabirka a_n:

```
static double sabirak(int n)
```

Sume i proizvodi

- Izračunavanje sume može se implementirati sledećim rekurzivnim metodom:

```
static double suma(int n) {
  if (n == 0)
    return 0.0;
  else
    return suma(n-1) + sabirak(n);
}
```

- Međutim, ova implementacija je neefikasna, jer se čuvaju svi međurezultati izračunavanja u lancu rekurzivnih poziva

Sume i proizvodi

- Moguće je rekurzivno izračunavanje sume organizovati tako da se u samom metodu ne radi ništa nego se izračunavanje izvodi u zaglavlju metoda pri prenošenju parametara
- Ovako korišćeni parametri nazivaju se **akumulirajući parametri**

```
static double suma(double zbir, int i, int n) {
  if (i > n)
    return zbir;
  else
    return suma(zbir + sabirak(i), i + 1, n);
}
public static void main(String[] args) {
  System.out.print("Unesite n: ");
  int n = Svetovid.in.readInt();
  System.out.println("suma(n) = " + suma(0.0, 1, n));
}
```

Sume i proizvodi

- U većini programskih jezika sume i proizvodi se izračunavaju efikasnije korišćenjem petlji, najčešće varijacijom sledećih opštih postupaka:

```
suma = 0.0;
i = 1;
while (i <= n) {
  izracunati sabirak ai;
  suma += ai;
  i++;
}
```

```
proizvod = 1.0;
i = 1;
while (i <= n) {
  izracunati cinilac bi;
  proizvod *= bi;
  i++;
}
```

ili korišćenjem odgovarajuće `for` ili `do-while` petlje

- Gornji opšti postupak ilustrovaćemo na nekoliko primera

Suma recipročnih vrednosti

- Treba napisati metod koji računa sumu recipročnih vrednosti prvih *n* prirodnih brojeva ($n \geq 1$)
- Pošto je izračunavanje sabiraka jednostavno, nećemo za to koristiti posebnan metod
- Koristićemo `for` petlju
- Ako korisnik unese *n* koje nije u dozvoljenom opsegu, metod će vratiti -1 i tako signalizirati grešku

Suma recipročnih vrednosti

```
static double sumaRV(int n) {
  final int DONJA_GR = 1;
  if (n >= DONJA_GR) {
    double suma = 1.0;
    for (int i = 2; i <= n; i++) {
      suma += 1.0 / i;
    }
    return suma;
  }
  else {
    return -1.0;
  }
}
```

Suma kvadrata

- Treba napisati metod koji računa sumu kvadrata prvih *n* prirodnih brojeva ($0 \leq n \leq 2000$)
- Pošto je izračunavanje sabiraka jednostavno, opet nećemo za to koristiti posebnan metod
- Koristićemo `while` petlju
- Ako korisnik unese *n* koje nije u dozvoljenom opsegu, metod će vratiti -1 i tako signalizirati grešku
- Takođe, metod će vratiti -1 ako dođe do prekoračenja opsega pri računanju sume
  - Sabirci ne mogu prekoračiti opseg, jer je $2000^2 <$ `Integer.MAX_VALUE`

Suma kvadrata

```
static int sumaKvad(int n) {
  final int DONJA_GR = 1;
  final int GORNJA_GR = 2000;
  if (DONJA_GR <= n && n <= GORNJA_GR) {
    boolean ok = true;
    int i = 1;
    int suma = 0;
    while (ok && i <= n) {
      int sabirak = i * i;
      ok = Integer.MAX_VALUE - suma > sabirak;
      if (ok) suma += sabirak;
      i++;
    }
    if (ok) return suma;
    else return -1;
  }
  else {
    return -1;
  }
}
```

Izračunavanje a^{n^2}

- Treba napisati metod koji računa za dati realan broj *a* i ceo broj *n* računa a^{n^2} koristeći množenje

- Kako je stepenovanje specijalan slučaj proizvoda, problem možemo rešiti jednostavnim programskim fragmentom:

```
int rez = 1;
for (int i = 1; i <= n*n; i++) {
  rez *= a;
}
```

- Ovaj postupak jeste jednostavan, ali zahteva n^2 množenja

Izračunavanje a^{n^2}

- Ukoliko primenimo opšti postupak za izračunavanje elemenata rekurentnog niza jedne promenljive, $s_n = f(s_{n-1})$, treba naći funkcijsku vezu f
- U ovom slučaju (kao što smo imali i kod binomnih koeficijenata) pogodno je tražiti vezu oblika $s_n = y\, s_{n-1}$, za neko nepoznato y koje treba odrediti
- Za $a \neq 0$ dobijamo:

$$s_n = a^{n^2}, \qquad \frac{s_n}{s_{n-1}} = \frac{a^{n^2}}{a^{(n-1)^2}} = a^{2n-1}$$

pa je:

$$s_n = a^{2n-1}\, s_{n-1}, \qquad n > 0$$
$$s_0 = 1$$

Izračunavanje a^{n^2}

- Ako se sad i izračunavanje a^{2n-1} realizuje postupkom za izračunavanje elemenata rekurentnog niza, dobijamo:

$$x_n = a^{2n-1}, \qquad \frac{x_n}{x_{n-1}} = \frac{a^{2n-1}}{a^{2n-3}} = a^2$$

$$x_n = a^2\, x_{n-1}, \qquad n > 1$$
$$x_0 = 1/a$$

- Konačno se dobija da se a^{n^2} izračunava pomoću dva rekurentna niza $\{s_n\}$ i $\{x_n\}$
- Za izračunavanje je potrebno samo $2n$ množenja, umesto n^2 množenja iz prvobitne verzije
- Radi jednostavnosti nećemo proveravati prekoračenje opsega

Izračunavanje a^{n^2}

```
static double an2(double a, int n) {
  if (a == 0.0) {
    return 0.0;
  }
  else {
    n = Math.abs(n);
    double s = 1.0;
    double x = 1.0 / a;
    double cinilac = a * a;
    for (int i = 1; i <= n; i++) {
      x *= cinilac;
      s *= x;
    }
    return s;
  }
}
```