Iterativni i rekurzivni postupci

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 $s_{k\text{-}1}$ je najjednostavnije naći preko količnika s_k / $s_{k\text{-}1}$:

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

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:

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

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

- 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
- Korističemo `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
- Korističemo `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 zahtena 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;
  }
}
```

Izračunavanje a^n

- S obzirom da smo uspeli pronaći način za efikasnije računanje a^{n^2}, postavlja se pitanje da li je slično moguće uraditi i u opštem slučaju kod računanja a^n
  - Tačnije, da li postoji način da se izračuna a^n korišćenjem (mnogo) manje od n operacija množenja?
- Odgovor je potvrdan
- Ranije smo pri računanju a^n u stvari koristili funkcionalnu vezu $a^n = a^{n-1}\,a$
- Sad je ideja da koristimo vezu $a^n = a^{n-k}\,a^k$
- Postupak je obično najefikasniji za $k = n / 2$, pa dobijamo dekompoziciju $a^n = a^{n/2}\,a^{n/2} = (a^{n/2})^2$
- Postupak gde računanje a^n svodimo na računanje $a^{n/2}$ kog zatim kvadriramo, naziva se **uzastopno kvadriranje**

Izračunavanje a^n

- S obzirom da je izložilac *n* ceo broj mora se voditi računa o njegovoj parnosti, a stepen se dobija rekurzivno sledećim izračunavanjem:

```
if (n % 2 == 1)
  return a * sqr(rekStepen(a, n/2));
else
  return sqr(rekStepen(a, n/2));
```

- Trivijalan slučaj i izlaz iz rekurzije je *a* za *n* = 1
- Na osnovu ove ideje može se napisati rekurzivan metod i ako se ne vodi računa o eventualnom prekoračenju, dobija se sledeći kod

Izračunavanje a^n

```
static double sqr(double a) {
  return a * a;
}

static double rekStepen(double a, int n) {
  if (n == 1)
    return a;
  else if (n % 2 == 1)
    return a * sqr(rekStepen(a, n/2));
  else
    return sqr(rekStepen(a, n/2));
}
```

Izračunavanje a^n

```
static double stepen(double a, int n) {
  if (a == 0.0 && n <= 0)
    return Double.NaN;
  else {
    if (a == 0.0)
      return 0.0;
    else if (n == 0 || a == 1.0)
      return 1.0;
    else {
      if (n < 0) {
        a = 1.0 / a;
        n = Math.abs(n);
      }
      return rekStepen(a, n);
    }
  }
}
```

Izračunavanje a^n

- Može se napraviti i iterativna verzija:

```
static double stepen(double a, int n) {
  if (a == 0.0 && n <= 0)
    return Double.NaN;
  else {
    if (a == 0.0)
      return 0.0;
    else if (n == 0 || a == 1.0)
      return 1.0;
    else {
      if (n < 0) {
        a = 1.0 / a;
        n = Math.abs(n);
      }
      double stepen = 1.0;
      while (n > 0) {
        if (n % 2 == 1)
          stepen *= a;
        n /= 2;
        a *= a;
      }
      return stepen;
    }
  }
}
```

Opšti rekurentni nizovi

- Potpuno analogno rekurentnom nizu jedne promenljive definisanim sa $s_n = f(s_{n-1})$, može se definisati opšti postupak za izračunavanje elemenata niza $\{s_n\}$ datog rekurentnom relacijom r-tog reda ($r \geq 1$)
- Elementi niza izračunavaju se rekurentnim izrazom sa fiksnim brojem od r argumenata, tj. izrazom oblika:

$$s_n = f(s_{n-1}, s_{n-2}, \dots, s_{n-r}), n \geq r,$$
$$s_0 = pv_0, s_1 = pv_1, \dots, s_{r-1} = pv_{r-1},$$

gde su pv_i, $0 \leq i \leq r - 1$, date početne vrednosti

Opšti rekurentni nizovi

- Neka su elementi niza tipa `double` i neka je `R` zadat kao konstanta:
  `static final int R = 5;`
- Pretpostavićemo da je funkcionalna zavisnost realizovana metodom:
  `static boolean f(double[] s)`
- Za razliku od metoda `f` kod rekurentnih nizova jedne promenljive, parametar `s` sadrži (Javin) niz od `R` + 1 elementa:
  - Prvih `R` elemenata niza `s` predstavljaju `R` elemenata rekurentnog niza koji prethode elementu sa indeksom *n* kog računamo
  - Poslednji element niza `s` sadržaće novoizračunati element:
    `s[R] =` *f*(`s[R-1]`, `s[R-2]`, ..., `s[0]`)
  - Metod `f` vratiće logičku vrednost koja označava da li je došlo do neke greške
- Neka su početne vrednosti zadate nizom `pv`, tako da je `pv[0]` = pv_0, ..., `pv[R-1]` = $pv_{R\text{-}1}$
- Za prirodan broj *n* iz nekog dozvoljenog intervala ($0 \le n \le granica$) *n*-ti element rekurentnog niza može se izračunati sledećim programskim fragmentom

Opšti rekurentni nizovi

```
if (0 <= n && n <= GRANICA) {
  ok = true;
  if (n < R)
    rezultat = pv[n];
  else {
    for (int i = 0; i < R; i++) s[i] = pv[i];
    ok = f(s);
    if (ok) {
      if (n == R)
        rezultat = s[R];
      else {
        int i = R;
        do {
          for (int j = 1; j <= R; j++)
            s[j-1] = s[j];
          ok = f(s);
          i++;
        } while (i < n && ok);
        if (ok) rezultat = s[R];
      }
    }
  }
}
else {
  ok = false;
}
```