Iterativni i rekurzivni postupci

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:

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

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;
}
```

Opšti rekurentni nizovi

- Često je *r* mali broj, pa se tada neke petlje mogu zameniti nizom ekvivalentnih naredbi
- Takođe, izračunavanje narednog elementa često je jednostavno, pa se metod `f` može izostaviti
- Kao primer opšteg rekurentnog niza navodimo Fibonačijeve brojeve
- Svaki naredni Fibonačijev broj je zbir prethodna dva Fibonačijeva broja

Fibonačijevi brojevi

- Rekurentni niz Fibonačijevih brojeva definisan je sa:

$$f_n = f_{n-1} + f_{n-2}, n > 1$$
$$f_0 = 0, f_1 = 1$$

- Primenićemo opšti postupak za izračunavanje rekurentnih nizova od *r* promenljivih, uz nekoliko pojednostavljenja:
  - Nećemo definisati metod `f` zbog jednostavnosti izračunavanja novih elemenata
  - Nećemo koristiti niz početnih vrednosti, nego ćemo početne vrednosti odmah staviti u niz `f`
  - Izbacićemo neke nepotrebne dodele promenljivoj `ok`, jer ćemo povratnom vrednošću `-1` signalizirati grešku
  - Elementi niza biće tipa `int`
- Dobija se sledeći metod

Fibonačijevi brojevi

```
static int fibonacci(int n) {
  final int GRANICA = 50;
  int[] f = new int[3];
  boolean ok = true;
  if (0 <= n && n <= GRANICA) {
    f[0] = 0;
    f[1] = 1;
    f[2] = f[1] + f[0];
    if (n <= 2)
      return f[n];
    else {
      int i = 2;
      do {
        f[0] = f[1];
        f[1] = f[2];
        ok = Integer.MAX_VALUE - f[1] > f[0];
        if (ok) f[2] = f[1] + f[0];
        i++;
      } while (i < n && ok);
      if (ok) return f[2];
    }
  }
  return -1;
}
```

Fibonačijevi brojevi

- U slučaju Fibonačijevih brojeva izračunavanje može da se organizuje malo efikasnije koristeći činjenicu da je veza među njima jednostavna

- Naime, ako se sabiranje vrši i u `f[0]` i u `f[1]`, tako da u `f[0]` budu Fibonačijevi brojevi sa parnim indeksima, a u `f[1]` sa neparnim, tada se korak u petlji može povećavati za dva, što će skratiti broj prolazaka kroz petlju na pola

- Takođe, ne mora se koristiti niz, nego obične promenljive

- Radi jednostavnosti, eliminisaćemo i proveru gornje granice

Fibonačijevi brojevi

```
static int fibonacci1(int n) {
  int f0, f1;
  boolean ok = true;
  if (0 <= n) {
    f0 = 0;
    f1 = 1;
    int i = 1;
    while (i < n && ok) {
      ok = Integer.MAX_VALUE - f1 - f1 > f0;
      // dva puta oduzimamo f1 jer cemo ga dva puta dodati
      if (ok) {
        f0 = f0 + f1;
        f1 = f0 + f1;
        i += 2;
      }
    }
    if (ok)
      if (n % 2 == 1) return f1;
      else return f0;
  }
  return -1;
}
```

Fibonačijevi brojevi

- Fibonačijevi brojevi mogu da se izračunavaju i rekurzivno, direktnim korišćenjem definicije:

```
static int fibonacci2(int n) {
  if (0 <= n)
    return fib(n);
  else
    return -1;
}
static int fib(int n) {
  if (n <= 1)
    return n;
  else
    return fib(n-1) + fib(n-2);
}
```

Fibonačijevi brojevi

- Prethodni način izračunavanja je vrlo neefikasan, jer se rekurzivnim pozivima metoda `fib` prethodni elementi rekurentnog niza nepotrebno izračunavaju više puta
- Npr. za izračunavanje $f_{25} = 75025$ izvršava se 242785 poziva metoda `fib`, dok se iterativnim postupkom u metodi `fibonacci1` koristi samo 26 sabiranja (ne računajući proveru prekoračenja opsega i povećanje brojača)
- Korišćenjem tehnike akumulirajućih parametara moguće je rekurzivno rešenje po efikasnosti, a i po prirodi, približiti iterativnom
  - Za računanje f_n biće potrebno samo n+1 poziva metoda
  - Metodom `fib` ćemo u stvari simulirati `while` petlju, jer će u svakoj “iteraciji” (rekurzivnom pozivu) parametar `f1` dobiti vrednost `f0 + f1`, parametar `f0` će dobiti vrednost `f1`, a poslednji parametar će igrati ulogu brojača

Fibonačijevi brojevi

```
static int fibonacci3(int n) {
  if (0 <= n)
    return fib(1, 0, n);
  else
    return -1;
}

static int fib(int f1, int f0, int n) {
  if (n == 0)
    return f0;
  else
    return fib(f0+f1, f1, n-1);
}
```