Rekurzija

© Goodrich, Tamassia, Goldwasser

Katedra za informatiku, Fakultet tehničkih nauka, Univerzitet u Novom Sadu

2019.

Rekurzija kao šablon

- rekurzija: kada funkcija poziva samu sebe
- klasičan primer: faktorijel

$$n! = 1 \cdot 2 \cdot 3 \cdot ... (n-1) \cdot n$$

- rekurzivna definicija:

$$n! = \begin{cases} 1 & \text{ako } n = 0 \\ n(n-1)! & \text{inače} \end{cases}$$

Faktorijel pomoću rekurzije

```python
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

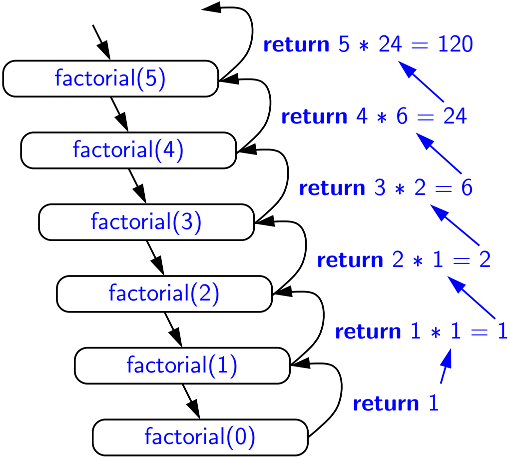
Sadržaj rekurzivne funkcije

- osnovni slučajevi
  - vrednosti ulaznih promenljivih za koje ne pravimo rekurzivne pozive
  - mora postojati bar jedan
- rekurzivni pozivi
  - poziv iste funkcije
  - svaki rekurzivni poziv bi trebalo definisati tako da predstavlja napredovanje prema osnovnom slučaju

Vizuelizacija rekurzije

- trag rekurzije
  - pravougaonik za svaki rekurzivni poziv
  - strelica od pozivača ka pozvanom
  - strelica od pozvanog ka pozivaču sa rezultatom koji se vraća

Vizuelizacija rekurzije: faktorijel



Primer rekurzije: „engleski lenjir“

- odštampati crtice i brojeve tako da se dobije izgled lenjira

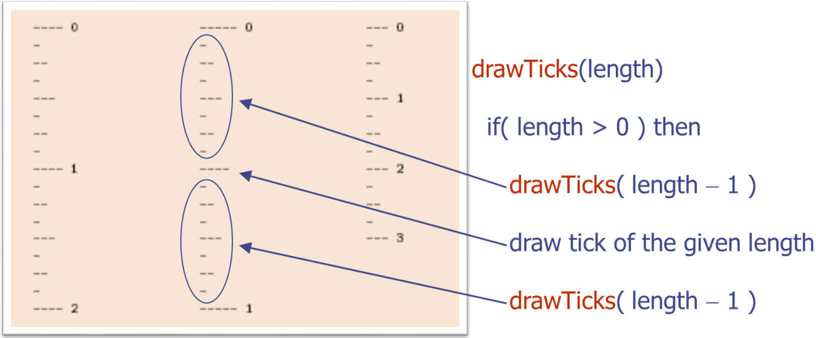
```
---- 0
-
--
-
---
-
--
-
---- 1
-
--
-
---
-
--
-
---- 2
```

```
----- 0
-
--
-
---
-
--
-
----
-
--
-
---
-
--
-
----- 1
```

```
--- 0
-
--
-
--- 1
-
--
-
--- 2
-
--
-
--- 3
```

Crtanje „engleskog lenjira“

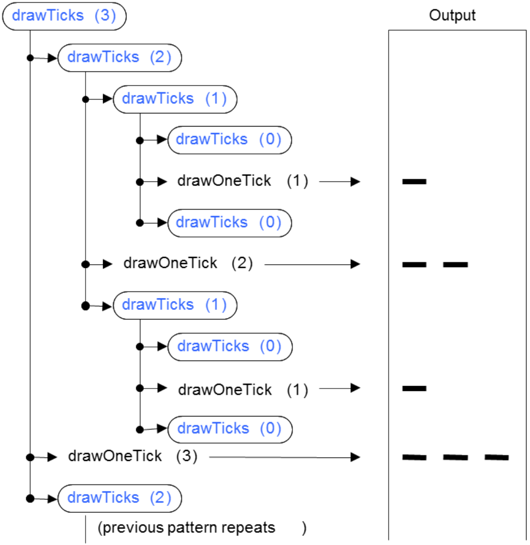
- `drawTicks(length)`
- ulaz: dužina crtice
- izlaz: lenjir sa crticom date dužine u sredini i manji lenjiri sa leve i desne strane



Crtanje „engleskog lenjira“

- interval sa centralnom crticom dužine $L \geq 1$ sastoji se od
  - intervala sa centralnom crticom dužine $L - 1$
  - crtice dužine L
  - intervala sa centralnom crticom dužine $L - 1$

Crtanje „engleskog lenjira“



Crtanje „engleskog lenjira“: Python implementacija

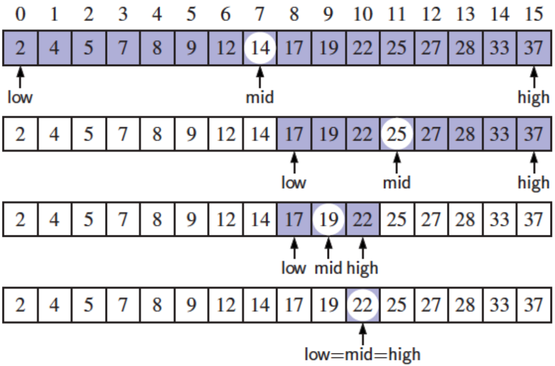
```python
def draw_line(tick_length, tick_label=''):
    """Draw one line with given tick length (followed by optional label)."""
    line = '-' * tick_length
    if tick_label:
        line += tick_label
    print(line)

def draw_interval(center_length):
    """Draw tick interval based upon a central tick length."""
    if center_length > 0:                   # recursively draw top ticks
        draw_interval(center_length - 1)    # draw center tick
        draw_line(center_length)            # recursively draw bottom ticks
        draw_interval(center_length - 1)

def draw_ruler(num_inches, major_length):
    """Draw English ruler with given number of inches, major tick length."""
    draw_line(major_length, '0')            # draw inch 0 line
    for j in range(1, 1+num_inches):
        draw_interval(major_length - 1)     # draw interior ticks for inch
        draw_line(major_length, str(j))     # draw inch j line and label
```

Binarna pretraga

```
def binary_search(data, target, low, high):
  """Return True if target is found in indicated portion of a Python
  list. The search only considers the portion from data[low] to
  data[high] inclusive.
  """
  if low > high:
    return False                    # interval is empty; no match
  else:
    mid = (low + high) // 2
    if target == data[mid]:         # found a match
      return True
    elif target < data[mid]:
      # recur on the portion left of the middle
      return binary_search(data, target, low, mid - 1)
    else:
      # recur on the portion right of the middle
      return binary_search(data, target, mid + 1, high)
```

Vizuelizacija binarne pretrage

- `target == data[mid]` – našli smo ga
- `target < data[mid]` – ponavljamo pretragu u levoj polovini
- `target > data[mid]` – ponavljamo pretragu u desnoj polovini



Analiza binarne pretrage

- radi u $O(\log n)$ vremenu
- veličina preostale liste je `high-low+1`
- posle jednog poređenja, to postaje

$$(mid - 1) - low + 1 = \left\lfloor \frac{low + high}{2} \right\rfloor \leq \frac{high - low + 1}{2}$$

$$high - (mid + 1) + 1 = high - \left\lfloor \frac{low + high}{2} \right\rfloor \leq \frac{high - low + 1}{2}$$

- $\Rightarrow$ svaki rekurzivni poziv deli region pretrage na pola; prema tome, može biti najviše $\log n$ nivoa

Linearna rekurzija

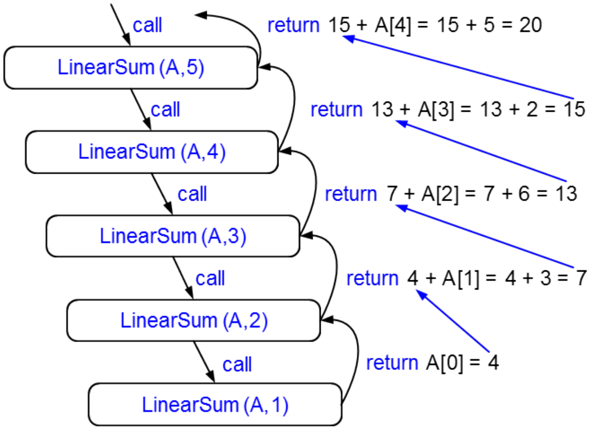
- testiranje baznih slučajeva
  - početi sa testiranjem baznih slučajeva (mora biti bar jedan)
  - obrada baznog slučaja ne sme koristiti rekurziju
  - svaki mogući lanac rekurzivnih poziva mora se završiti dolaskom do baznog slučaja
- rekurzivno jednom
  - napraviti jedan rekurzivni poziv
  - možemo napraviti grananje sa odlukom da se izabere jedan od mogućih rekurzivnih poziva
  - svaki mogući rekurzivni poziv treba da se približi baznom slučaju

Primer linearne rekurzije

LinearSum(A, n)
Input: A: niz celih brojeva
Input: n: broj brojeva u nizu koje treba sabrati
Output: suma prvih n brojeva u A

if $n = 1$ **then**
  return $A[0]$
else
  return LinearSum(A, $n - 1$) $+ A[n - 1]$

Primer linearne rekurzije



Obrtanje redosleda u nizu

ReverseArray(A, i, j)
Input: A: niz brojeva
Input: i, j: nenegativni indeksi, $i < j$
Output: obrnut redosled u A počevši od indeksa i do indeksa j

```
if i < j then
    swap A[i], A[j]
    ReverseArray(A, i + 1, j − 1)
```

Definisanje elemenata za rekurziju

- prilikom dizajniranja rekurzivnih funkcija važno je definisati ih tako da je rekurzija jednostavna
- ponekad to znači da treba definisati dodatne parametre funkcije
- na primer, definisali smo ReverseArray(A, i, j) umesto ReverseArray(A)

Definisanje elemenata za rekurziju

- Python implementacija

```
def reverse(S, start, stop):
  """Obrni elemente u isečku S[start:stop]."""
  # ako ima bar dva elementa
  if start < stop - 1:
    # zameni im mesta
    S[start], S[stop-1] = S[stop-1], S[start]
    # rekurzivno obrni ostatak
    reverse(S, start+1, stop-1)
```

Stepenovanje

- funkciju za stepenovanje $p(x, n) = x^n$ možemo definisati rekurzivno:

$$p(x, n) = \begin{cases} 1 & \text{ako } n = 0 \\ x \cdot p(x, n-1) & \text{inače} \end{cases}$$

- na ovaj način ćemo dobiti funkciju koja radi u $O(n)$ vremenu (jer pravimo n poziva)
- može li brže?

Stepenovanje

- možemo napraviti brži linearno rekurzivan algoritam pomoću ponavljanog kvadriranja:

$$p(x,n) = \begin{cases} 1 & \text{ako je } x = 0 \\ x \cdot p(x, \frac{n-1}{2})^2 & \text{ako je } x > 0 \text{ neparan} \\ p(x, n/2)^2 & \text{ako je } x > 0 \text{ paran} \end{cases}$$

- na primer:
  $2^4 = 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16$
  $2^5 = 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32$
  $2^6 = 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64$
  $2^7 = 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128$

Rekurzivno stepenovanje

Power(x, n):

Input: broj x i njegov stepen n

Output: vrednost x^n

```
if n = 0 then
    return 1
if n je paran then
    y ← Power(x, (n − 1)/2)        {svakim pozivom polovimo n}
    return x · y · y
else
    y ← Power(x, n/2)
    return y · y                    {promenljiva umesto duplog poziva funkcije}
```

„Repna“ rekurzija

- kada je rekurzivni poziv poslednji korak u funkciji
- primer: ReverseArray(A, i, j)
- lako se preradi u iterativni postupak

IterativeReverseArray(A, i, j)

Input: niz A i nenegativni indeksi i i j, $i < j$

Output: obrnut redosled elemenata u A počevši od indeksa i do j

while $i < j$ **do**
  swap $A[i], A[j]$
  $i \leftarrow i + 1$
  $j \leftarrow j - 1$

Binarna rekurzija

- kada postoje dva rekurzivna poziva za svaki bazni slučaj
- primer: engleski lenjir

```
---- 0
-
--
-
---
-
--
-
---- 1
-
--
-
---
-
--
-
---- 2
```

```
----- 0
-
--
-
---
-
--
-
----
-
--
-
---
-
--
-
----- 1
```

```
--- 0
-
--
-
--- 1
-
--
-
--- 2
-
--
-
--- 3
```

Binarna rekurzija: sumiranje elemenata

BinarySum(A, i, n)

Input: niz A i celi brojevi i i n

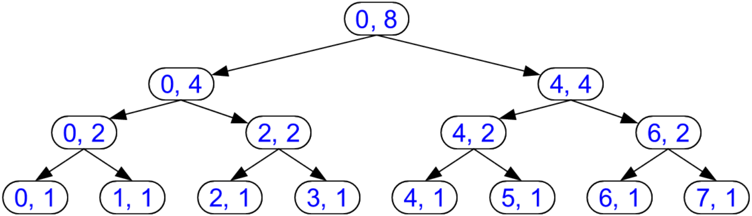
Output: zbir n brojeva iz A počevši od i-tog

if n=1 **then**

  return $A[i]$

else

  return BinarySum$(A, i, n/2)$ + BinarySum$(A, i + n/2, n/2)$



Fibonačijevi brojevi

- definišu se rekurzivno:
  $F_0 = 0$
  $F_1 = 1$
  $F_i = F_{i-1} + F_{i-2} \quad$ za $i > 1$
- rekurzivni algoritam (prvi pokušaj):

BinaryFib(k)

Input: nenegativan ceo broj k

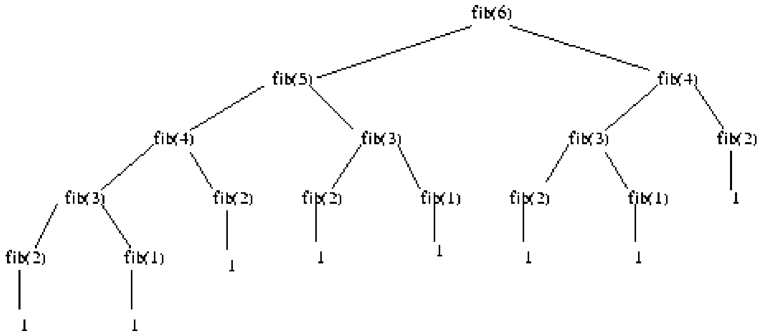
Output: k-ti Fibonačijev broj F_k

```
if k ≤ 1 then
    return k
else
    return BinaryFib(k − 1) + BinaryFib(k − 2)
```

Fibonačijevi brojevi

- neka je n_k broj rekurzivnih poziva funkcije BinaryFib(k):
  - $n_0 = 1$
  - $n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 41$
  - $n_8 = n_7 + n_6 + 1 = 67$
- n_k se svaki drugi put više nego duplira!
- dakle, $n_k \geq 2^{k/2}$
- broj poziva raste eksponencijalno!

Fibonačijevi brojevi – ponavljanje rekurzivnih poziva



Fibonačijevi brojevi v2

- pomoću linearne rekurzije

LinearFib(k)

Input: pozitivan ceo broj k
Output: par Fibonačijevih brojeva (F_k, F_{k-1})

```
if k = 1 then
    return (k, 0)
else
    (i, j) ← LinearFib(k − 1)
    return (i + j, i)
```

- ima samo $k-1$ rekurzivnih poziva!

Višestruka rekurzija

- primer problema: zagonetke sabiranja
  - $pot + pan = bib$
  - $dog + cat = pig$
  - $boy + girl = baby$
- višestruka rekurzija
- potencijalno pravi puno rekurzivnih poziva
- ne samo jedan ili dva

Višestruka rekurzija

PuzzleSolve(k, S, U)

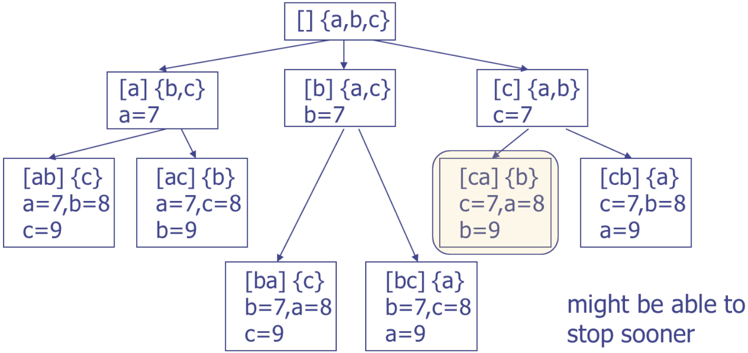
Input: ceo broj k, sekvenca S, skup U (skup svih elemenata)
Output: lista svih proširenja S dužine k korišćenjem elemenata iz U bez ponavljanja

```
for all e ∈ U do
   ukloni e iz U                                   {e se sada koristi}
   dodaj e na kraj S
   if k = 1 then
      test da li S predstavlja rešenje
      if S predstavlja rešenje then
         return 'Solution found:', S
   else
      PuzzleSolve(k − 1, S, U)
      dodaj e u U                                  {e se više ne koristi}
      ukloni e sa kraja S
```

Višestruka rekurzija

$cbb + ba = abc$

$799 + 98 = 897$



Višestruka rekurzija