10 - Rekurzivne funkcije

January 28, 2024

1 Rekurzivne funkcije

Preden sploh začnemo, si oglejmo tale nesmiselni program.

[1]:
```
def f(x):
    return g(x) + 1

def g(x):
    return 2 * x
```

Sme funkcija `f` poklicati funkcijo `g`, še preden je slednja definirana?

Ne. Vendar je tudi ne. Funkcija `f` bo poklicala funkcijo `g` šele, ko pokličemo `f`. Recimo zdaj.

[2]:
```
f(5)
```

[2]:
```
11
```

Python vidi definicijo funkcije `f` in si jo zapomni; izvajati je ne poskuša, zato (še) ne naleti na klic funkcije `g`. Nato vidi definicijo funkcije `g`. Ko potem pokličemo `f`, `g` že obstaja in vse je v redu.

Narobe bi bilo, če bi funkcijo poklicali prezgodaj.

[3]:
```
def h(x):
    return j(x) + 1

h(5)

def j(x):
    return 2 * h
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[3], line 4
      1 def h(x):
      2     return j(x) + 1
----> 4 h(5)
      6 def j(x):
      7     return 2 * h
```

```
Cell In[3], line 2, in h(x)
      1 def h(x):
----> 2     return j(x) + 1

NameError: name 'j' is not defined
```

Ampak tega ne bomo počeli.

1.1 Sodolihi in lihosodi

Ni, da bi govoril. Bila sta oni dan dva lena študenta, Jože in Benjamin. Za domačo nalogo bi morala napisati naslednji funkciji.

- `sodolihi(s)` prejme seznam celih števil in vrne `True`, če se v njem izmenjujejo **soda in liha števila, začenši s sodim**. Če ni tako, vrne `False`. Za prazen seznam vrne `True`.
- `lihosodi(s)` prejme seznam celih števil in vrne `True`, če se v njem izmenjujejo **liha in soda števila, začenši s lihim**. Če ni tako, vrne `False`. Za prazen seznam vrne `True`.

Ker sta imela domačih nalog iz programiranja dovolj, sta se dogovorila, da vsak sprogramira eno funkcijo, potem pa si ju izmenjata in malo preimenujeta spremenljivke v naivni veri, da ju ne bomo zalotili.

Jože ni bil le len, temveč tudi zvit. Razmišljal je, da bo Benjamin sprogramiral funkcijo `lihosodi`, torej ne bo nič narobe, če njegova funkcija kliče Benjaminovo. In je naredil kar takole.

[4]:
```
def sodolihi(s):
    if s == []:
        return True
    if s[0] % 2 == 1:
        return False
    return lihosodi(s[1:])
```

Preveril je prazen seznam in prvi element, ostanek - v katerem se morajo izmenjevati lihi in sodi, začenši z lihim - pa prepustil Benjaminu.

Tudi Benjamin ni le len, zvit. Zvit je natančno na isti način kot Jože. Češ, poskrbim za prazen seznam in za prvi element, ostalo lahko tako ali tako prepustim Jožetu. In je sprogramiral

[5]:
```
def lihosodi(s):
    if s == []:
        return True
    if s[0] % 2 == 0:
        return False
    return sodolihi(s[1:])
```

Jožetova funkcija bo očitno delovala, če bo delovala Benjaminova. In obratno: Benjaminova bo delovala, če bo Jožetova. Skratka, obe funkciji bosta delovali, če bosta delovali obe funkciji.

Vprašanje je samo … bo Python to res dovolil? Bo. Funkcije v Pythonu smejo klicati druge funkcije. In nič jim ne preprečuje, da slednje ne bi poklicale nazaj. (Za globlje misleče: da, lahko si zamislimo jezik - oziroma način izvajanja - v katerem to ne bi šlo. Takšne so bile zelo stare različice Fortrana;

omejitev je bila v tem, da so bili argumenti funkcije zapisani na določenem mestu v pomnilniku. Kmalu so odkrili, da tako ne gre in je argumente in lokalne spremenljivke bolj smiselno shranjevati na skladu.)

Sicer pa: poskusimo.

[6]:
```
sodolihi([4, 7, 6, 1, 8])
```

[6]:
```
True
```

[7]:
```
sodolihi([4, 7, 6, 2, 1])
```

[7]:
```
False
```

Če hočemo slediti poteku, lahko naredimo funkciji zgovornejši.

[8]:
```
def sodolihi(s):
    print("Sodolihi preverja", s)
    if s == []:
        print("- prazen seznam: ok")
        return True
    if s[0] % 2 == 1:
        print("- prvi element ni ok")
        return False
    print("- začatek je v redu, potrebno bo preveriti ostanek")
    ok = lihosodi(s[1:])
    print(f"Sodolihi je izvedel, da je ostanek {ok} in sporoča, da je {s} {ok}.
 ↪")
    return ok

def lihosodi(s):
    print("Lihosodi preverja", s)
    if s == []:
        print("- prazen seznam: ok")
        return True
    if s[0] % 2 == 0:
        print("- prvi element ni ok")
        return False
    print("- začatek je v redu, potrebno bo preveriti ostanek")
    ok = sodolihi(s[1:])
    print(f"Lihosodi je izvedel, da je ostanek {ok} in sporoča, da je {s} {ok}.
 ↪")
    return ok
```

[9]:
```
sodolihi([4, 7, 6, 1, 8])
```

```
Sodolihi preverja [4, 7, 6, 1, 8]
- začatek je v redu, potrebno bo preveriti ostanek
Lihosodi preverja [7, 6, 1, 8]
```

```
- začatek je v redu, potrebno bo preveriti ostanek
Sodolihi preverja [6, 1, 8]
- začatek je v redu, potrebno bo preveriti ostanek
Lihosodi preverja [1, 8]
- začatek je v redu, potrebno bo preveriti ostanek
Sodolihi preverja [8]
- začatek je v redu, potrebno bo preveriti ostanek
Lihosodi preverja []
- prazen seznam: ok
Sodolihi je izvedel, da je ostanek True in sporoča, da je [8] True.
Lihosodi je izvedel, da je ostanek True in sporoča, da je [1, 8] True.
Sodolihi je izvedel, da je ostanek True in sporoča, da je [6, 1, 8] True.
Lihosodi je izvedel, da je ostanek True in sporoča, da je [7, 6, 1, 8] True.
Sodolihi je izvedel, da je ostanek True in sporoča, da je [4, 7, 6, 1, 8] True.
```

[9]:
```
True
```

1.2 Dodatna naloga: samisodi

Opogumljena z uspehom sta se Jože in Benjamin lotila dodatne naloge:

- napiši funkcijo `samisodi(s)`, ki vrne `True`, če podani seznam celih števil ne vsebuje nobenega lihega števila (in `False`, če ga).

Zdaj je bilo potrebno napisati le eno funkcijo, tako da si nista mogla deliti dela. Idejo pa sta obdržala. Poglejmo, kako jima je uspelo.

[10]:
```
def samisodi(s):
    if s == []:
        return True
    if s[0] % 2 == 1:
        return False
    return samisodi(s[1:])
```

[11]:
```
samisodi([6, 8, 10, 2, 4, 6])
```

[11]:
```
True
```

[12]:
```
samisodi([6, 4, 5, 2, 0])
```

[12]:
```
False
```

Ideja je, torej enaka. Funkcija preveri prazen seznam in prvi element, ostalo delo pa prepusti, no, sama sebi.

2 Kako (ne) pisati rekurzivne funkcije

Funkcijam, ki kličejo same sebe, pravimo *rekurzivne funkcije*. Študenti se jih iz nekega razloga bojijo. Morda zato, ker zahtevajo drugačen način razmišljanja, ali pa jih je, preprosto, strah,

kako bo to delovalo. Če kaj, potem rekurzivne funkcije zahtevajo razsvetljenje. V nekem trenutku postanejo, preprosto *normalne* in jih pišemo enako brezskrbno kot "navadne".

Prej pa nam mora priti nekaj stvari, povezanih z njimi v rutino.

2.1 Korak bližje rešitvi

Razlog, da rekurzivne funkcije delujejo, je, da smo z vsakim klicem bližje rešitvi. Poglejmo tale primer nesrečne funkcije, ki naj bi računala vsoto elementov seznama `s`.

[13]:
```
def vsota(s):
    return vsota(s)
```

[14]:
```
vsota([2, 5, 8])
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
Cell In[14], line 1
----> 1 vsota([2, 5, 8])

Cell In[13], line 2, in vsota(s)
      1 def vsota(s):
----> 2     return vsota(s)

Cell In[13], line 2, in vsota(s)
      1 def vsota(s):
----> 2     return vsota(s)

    [… skipping similar frames: vsota at line 2 (2971 times)]

Cell In[13], line 2, in vsota(s)
      1 def vsota(s):
----> 2     return vsota(s)

RecursionError: maximum recursion depth exceeded
```

To ne gre. Ta funkcija je prepustila *vse* delo sama sebi. Kliče se skoraj v nedogled - oziroma toliko časa, dokler se Pythonu zazdi, da je to že preveč in sproži napako "maximum recursion depth exceeded".

Pravilna rešitev je

[15]:
```
def vsota(s):
    if s == []:
        return 0
    return s[0] + vsota(s[1:])
```

[16]:
```
vsota([2, 5, 8])
```

[16]: 15

Ta, druga različica, naredi en korak: k prvemu elementu prišteje vsoto *ostalih*. Seznam, ki ga je potrebno sešteti, je v vsakem klicu za en element krajši; prej ko slej bomo prišli do praznega seznama.

Razliko med (napačno) prvo in (pravilno) drugo obliko vidimo, če funkciji opišemo v slovenščini.

- Druga pravi, da vsoto seznama dobimo tako, da k prvemu elementu prištejemo vsoto ostalih.
- Prva pravi, da vsoto seznama dobimo tako, da izračunamo vsoto seznama.

Druga zveni kot načrt. Prva zveni kot neumnost.

Pravilno napisano rekurzivno funkcijo se da navadno tudi lepo opisati. Seznam vsebuje same sode elemente, če je prazen, ali pa je prvi element sod in so sodi tudi vsi ostali.

2.2 Robni pogoj

Rekurzivna funkcija bo torej z vsakim klicem bližje koncu. Na koncu, v nekem klicu (ali več klicih, kot bomo videli na primerih rekurzije na rodbinskem drevesu) ne bo več klica naprej. To se bo zgodilo takrat, ko bo problem (na primer podani seznam) že tako preprost, da bomo vedeli odgovor brez računanja. Prazni seznami ne vsebujejo lihih elementov. Vsota elementov praznih seznamov je 0.

Rekurzivna funkcija se bo zato pogosto začela z nekim *robnim pogojem*, ki bo preverjal, ali je delo končano in nadaljnji klici niso več potrebni. Robni pogoj bo včasih impliciten, skrit. Tako bo recimo, ko bo v preiskovanju rodbinskega drevesa funkcija poklicala samo sebe za vse otroke določene osebe. Za osebe brez otrok se pač ne bo več poklicala. Tam ne bo nobenega `if`, pač pa se bo klic nahajal znotraj zanke, ki se morda ne bo izvedla nikoli.

2.3 Rekurzija in lokalne spremenljivke

Ena od težav pri razumevanju rekurzije izhaja iz napačne predstave o lokalnih spremenljivkah. Poglejmo si bolj zgovorno različico funkcije `sum`.

[17]:
```
def vsota(s):
    print("Računam vsoto", s)
    if s == []:
        print("Prazen seznam, vračam 0.")
        return 0
    prvi = s[0]
    ostanek = s[1:]
    print("Izvedeti moram vsoto", ostanek)
    vsota_ost = vsota(ostanek)
    print(f"Izvedel sem, da je vsota {ostanek} enaka {vsota_ost}; vračam {prvi}␣
 ↪+ {vsota_ost} = {prvi + vsota_ost}")
    return prvi + vsota_ost
```

[18]:
```
vsota([2, 5, 3, 1])
```

```
Računam vsoto [2, 5, 3, 1]
Izvedeti moram vsoto [5, 3, 1]
Računam vsoto [5, 3, 1]
Izvedeti moram vsoto [3, 1]
Računam vsoto [3, 1]
Izvedeti moram vsoto [1]
Računam vsoto [1]
Izvedeti moram vsoto []
Računam vsoto []
Prazen seznam, vračam 0.
Izvedel sem, da je vsota [] enaka 0; vračam 1 + 0 = 1
Izvedel sem, da je vsota [1] enaka 1; vračam 3 + 1 = 4
Izvedel sem, da je vsota [3, 1] enaka 4; vračam 5 + 4 = 9
Izvedel sem, da je vsota [5, 3, 1] enaka 9; vračam 2 + 9 = 11
```

[18]: `11`

Opazujte zadnje izpise: funkcija izpiše vrednost `ostanek`, in ta je najprej `[]`, potem `[1]`, nato `[3, 1]` in potem `[5, 3, 1]`.

Lokalne spremenljivke niso *lokalne spremenljivke funkcije*, temveč *lokalne spremenljivke klica funkcije*. Nova spremenljivka `ostanek` nastane ob vsakem klicu funkcije in stari, prejšnji, "čakajoči" klic obdrži staro spremenljivko s staro vrednostjo. (Da ne bo zmotilo koga, ki razume kaj več, se moram še popraviti: ime `ostanek` se v vsakem klicu nanaša na drug objekt.)

Ko torej znotraj rekurzije rečemo `ostanek = s[1:]`, ne pokvarimo vrednosti `ostanek` iz "čakajočega" klica.

2.4 Rekurzija in globalne spremenljivke

Če dam študentom napisati rekurzivno funkcijo v slogu

- napiši funkcijo `sodi(s)`, ki vrne seznam, ki vsebuje vse sode elemente seznama `s`,

se navadno zgodi nekaj takšnega.

[19]:
```
def sodi(s):
    t = []
    if s == []:
        return t
    if s[0] % 2 == 0:
        t.append(s[0])
    sodi(s[1:])
    return t
```

Logika je v tem, da funkcija poskrbi za prvi element - če je sod, ga doda v `t` - nato pa se rekurzivno pokliče z namenom, da bi v `t` dodala še vse ostale sode elemente.

Žal ne deluje.

[20]:
```
sodi([2, 5, 3, 8, 0])
```

[20]: [2]

[21]:
```
sodi([3, 4, 8, 0])
```

[21]: []

Težava je prav v tem, da je `t` lokalna spremenljivka *klica*. V vsakem klicu dobimo nov `t`, vanj morda dodamo en element - v rekurzivnem klicu pa se zgodba ponovi z novim `t`.

Študent zadevo reši tako, da `t` spremeni v globalno spremenljivko.

[22]:
```
t = []

def sodi(s):
    if s == []:
        return t
    if s[0] % 2 == 0:
        t.append(s[0])
    sodi(s[1:])
    return t
```

To "deluje".

[23]:
```
sodi([2, 5, 3, 8, 0])
```

[23]: [2, 8, 0]

Ampak samo enkrat.

[24]:
```
sodi([3, 4, 8, 0])
```

[24]: [2, 8, 0, 4, 8, 0]

Seznama `t` nihče ne sprazni. Plan C je, da spremenljivko `t` spraznimo na začetku funkcije.

[25]:
```
t = []

def sodi(s):
    t.clear()
    if s == []:
        return t
    if s[0] % 2 == 0:
        t.append(s[0])
    sodi(s[1:])
    return t
```

Tudi to ne gre.

[26]:
```
sodi([2, 5, 3, 8, 0])
```

[26]: []

Zdaj vsak klic pobriše vse, kar se je zgodilo doslej - do zadnjega, ki prav tako izprazni seznam in vanj ne doda ničesar, saj je `s` prazen.

Rekurzija in globalne spremenljivke se ne marajo. (Če ne veste točno, kaj delate.)

Pravilna rešitev je takšna: če je prvi element sod, vrnemo seznam, ki vsebuje ta element in vse sode elemente ostanka. Sicer vrnemo sode elemente ostanka.

[27]:
```
def sodi(s):
    if s == []:
       return []
    if s[0] % 2 == 0:
        return [s[0]] + sodi(s[1:])
    else:
        return sodi(s[1:])
```

[28]:
```
sodi([2, 5, 3, 8, 0])
```

[28]: [2, 8, 0]

V zadnji rešitvi smo "razmišljali rekurzivno": funkcijo smo poklicali s krajšim seznamom in k dobljeni rešitvi dodali, kar je bilo treba (prvi element) (če je bilo treba). V prejšnjih, napačnih poskusih, je bila funkcija rekurzivna, razmišljanje pa ne. Rekurzivni klici so dodajali v isti seznam, vendar je rekurzija služila le kot zamenjava za zanko.

Razlika se vidi: v pravilni rešitvi smo uporabili rezultat rekurzivnega klica. V napačnih smo ga zavrgli in nekako računali na "stranski učinek" klica - dodajanje v seznam.

2.5 Akumulator

Zanemariti rezultat in uporabiti stranski učinek sicer ni nujno narobe. Vendar si je potrebno v tem primeru celotno funkcijo zamisliti drugače. Napisali bomo funkcijo, ki kot argument prejme seznam celih števil in še en seznam, v katerega mora zložiti vse sode elemente prvega seznama

[29]:
```
def sodi(s, t):
    if s == []:
        return
    if s[0] % 2 == 0:
        t.append(s[0])
    sodi(s[1:], t)
```

[30]:
```
e = []
sodi([2, 5, 3, 8, 0], e)

e
```

[30]: [2, 8, 0]

Študenti znajo sproducirati nekaj takšnega. Nekateri bi poskušali na podoben način reševati tudi prvo nalogo in sicer tako, da bi ignorirali navodila naloge, ki pravijo, da bo `sodi` dobil le en argument, ter mu dodali `t`, ki ima privzeto vrednost `[]`. To ne deluje, ker spreminjanje `t`-ja potem spreminja privzeto vrednost. Pač pa gre tako:

[31]:
```
def sodi(s, t=None):
    if t is None:
        t = []
    if s == []:
        return t
    if s[0] % 2 == 0:
        t.append(s[0])
    return sodi(s[1:], t)
```

[32]:
```
sodi([2, 5, 3, 8, 0])
```

[32]:
```
[2, 8, 0]
```

V prvi rešitvi, kjer smo imel `sodi(s, t)`, je bila rekurzija praktično zamenjava za zanko. Druga je še grša, saj je čuden križanec med temi, ki dopolnjujejo podani seznam in onimi, ki morajo vračati (nov) seznam.

Če nam je všeč ali ne: pišemo tudi takšne rekurzivne funkcije. Argument, ki si ga podajamo, je "akumulator", v katerega se tekom rekurzije nabira rezultat in zadnji klic ga vrne, da si ga ostali podajajo nazaj.

Funkcije v tej obliki znajo biti učinkovitejše. Zgodi se, da je lažje dopolnjevati akumulator kot stalno nekaj seštevati (ali karkoli že). Predvsem pa znajo nekateri jeziki pospešiti izvajanje *repne rekurzije*.

2.6 Repna rekurzija

Tole nima zveze s Pythonom: Python tega ne zna. Po trditvah njegovega avtorja tudi nikoli ne bo znal, saj to za Python ni zanimivo.

Gre pa za to: ko rekurzivna funkcija kliče samo sebe, mora biti ohranjena "sled" klicev: vedeti je potrebno, kdo je klical koga in komu mora vrniti rezultat. Poleg tega je potrebno ohranjati lokalne spremenljivke - kot recimo `ostanek`, ki smo ga videli nekaj razdelkov nazaj.

Če je rekurzivni klic zadnja, ampak čisto čisto zadnja stvar, ki jo naredimo v funkciji, in na koncu zgolj vrnemo rezultat tega klica - ne da bi mu, recimo, še kaj prišteli ali pa ga prišteli k čemu - potem takšni rekurziji rečemo *repna rekurzija*. V tem primeru

- ni potrebno pomnjenje lokalnih spremenljivk kličočih funkcij. Nobenega "`ostanka`" se nikoli nihče več ne bo dotikal.
- ni potrebno vračati rezultata tistemu, ki je poklical funkcijo, temveč tistemu, ki je, če citiramo Vonneguta, vse skupaj začel.

Zanimivo je predvsem slednje. Pokličemo funkcijo `sodi`. Ta pokliče sebe. Pokliče sebe. Pokliče sebe … In rezultati se vračajo med temi klici nazaj – vrne sebi, vrne sebi, vrne sebi … in na koncu vrne nam. Če je rekurzija repna, lahko na koncu vrne kar nam in preskoči vse "sebi".

Takšna rekurzija je veliko hitrejša. Jeziki, v katerih ni spremenljivk temveč le konstante in zato nimajo zank temveč le rekurzijo, zato praktično *morajo* optimizirati repno rekurzijo - programerji pa sestavljati funkcije, v katerih je rekurzivni klic zadnje, kar naredijo. Python ni eden od teh jezikov, zato tega ne optimizira. Z repno rekurzijo v Pythonu ne pridobimo ničesar.

2.7 Rekurzija lahko eksplodira

Zdaj pa napišimo funkcijo, ki vrne največje število v (nepraznem) seznamu.

[33]:
```
def maks(s):
    if len(s) == 1:
        return s[0]
    if s[0] > maks(s[1:]):
        return s[0]
    else:
        return maks(s[1:])
```

[34]:
```
maks([2, 3, 10, 5, 8])
```

[34]: 10

[35]:
```
maks([2, 3, 8, 10])
```

[35]: 10

Deluje že, deluje. Ampak za kakšno ceno! Ob vsakem klicu izpišimo zvezdico

[36]:
```
def maks(s):
    print("*")
    if len(s) == 1:
        return s[0]
    if s[0] > maks(s[1:]):
        return s[0]
    else:
        return maks(s[1:])
```

[37]:
```
maks([2, 3, 8, 10])
```

```
*
*
*
*
*
*
*
*
*
*
*
```

```
*
*
*
*
```

[37]: 10

Človek bi pričakoval nekako štiri zvezdice ne 15.

[38]:
```
maks([2, 3, 5, 8, 10])
```

```
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
```

[38]: 10

Tule pa jih je, če prav štejem, 31! En element več - dvakrat več zvezdic!

Težava je v tem, da funkcija `maks` dvakrat pokliče samo sebe. Tudi v običajnih funkcijah ni lepo, če dvakrat pokličemo neko funkcijo z enakimi argumenti. Če gre za rekurzivno funkcijo, pa je to

usodno. Funkcija `maks` dvakrat pokliče sebe. Ta v vsakem od teh dveh klicev dvakrat pokliče sebe - pa imamo štiri klice. Če vsak od teh štirih klicev dvakrat pokliče sebe, smo pri osmih klicih, iz teh osmih jih nastane šestnajst in iz šestnajstih dvaintrideset. (Eden manj je zato, ker ne gremo do praznega seznama temveč do seznama z enim elementom.)

V času epidemije so nekateri prvič slišali besedno zvezo *eksponentna rast* in jo zdaj na veliko uporabljajo kot sinonim za *hitro rast*. To ni isto. Nekaj je eksponentno, če se njegova velikost (trajanje, vrednost, karkoli že) v vsakem koraku pomnoži z nekim faktorjem. No, to kar se dogaja tule, je primer *eksponentne rasti*.

Zdravilo je enostavno. Funkcija se mora izogniti podvojenim klicem. Namesto

[39]:
```
def maks(s):
    print("*")
    if len(s) == 1:
        return s[0]
    if s[0] > maks(s[1:]):
        return s[0]
    else:
        return maks(s[1:])
```

napišemo

[40]:
```
def maks(s):
    print("*")
    if len(s) == 1:
        return s[0]
    m = maks(s[1:])
    if s[0] > m:
        return s[0]
    else:
        return m
```

pa bo.

[41]:
```
maks([2, 3, 5, 8, 10])
```

```
*
*
*
*
*
```

[41]: 10

2.8 Izogibanje neizogibni eksploziji

Imamo `n` stolov, na katere bomo posedli fante. Koliko jih posedemo - enega, dva, pet, več … - je vseeno. Lahko posedemo tudi nobenega, se pravi, pustimo vse stole prazne. Pomembno je le tole: ker so fantje poredni, mora biti med dvema zasedenima stoloma vsaj en prost, da se ne bodo tepli.

Vzemimo, recimo, šest stolov. Možni razporedi so

```
*.*.*. (1)
*.*..* (2)
*.*... (3)
*..*.* (4)
*..*.. (5)
*...*. (6)
*....* (7)
*..... (8)
.*.*.* (9)
.*.*.. (10)
.*..*. (11)
.*...* (12)
.*.... (13)
..*.*. (14)
..*..* (15)
..*... (16)
...*.* (17)
...*.. (18)
....*. (19)
.....* (20)
...... (21)
```

V nekaterih primerih smo posedli tri fante, v nekaterih manj, v zadnjem celo nobenega. Vseh možnih razporedov je 21.

Znamo napisati funkcijo, ki izračuna število razporedov za `n` stolov?

Znamo, in preprosteje bo, če bo rekurzivna. Takole razmišljamo: prvi sedež bo bodisi zaseden bodisi prazen. Razporedov, v katerih je prvi stol zaseden, je toliko, kolikor je razporedov na `n - 2` stolih. Če je prvi zaseden, bo namreč drugi prazen in razporejamo lahko le še `n - 2` stolov. Če pa je prvi prazen, je razporedov toliko, kolikor je razporedov na `n - 1` stolih.

Ostane nam še robni pogoj. Če je stolov 0, je razpored le `1`. Če je stol eden, sta razporeda dva: stol je poln ali prazen.

[42]:
```
def razporedov(n):
    if n == 0:
        return 1
    if n == 1:
        return 2
    return razporedov(n - 1) + razporedov(n - 2)
```

[43]:
```
razporedov(6)
```

[43]: 21

Upam, da si, dragi bralec, med tistimi, ki so prepoznali gornjo funkcijo. Gre za Fibonacijevo zaporedje: vsak element zaporedja je enak vsoti prejšnjih.

[44]:
```
for i in range(10):
    print(razporedov(i))
```

```
1
2
3
5
8
13
21
34
55
89
```

Fibonacijevega zaporedja nismo privlekli zaradi kakega predpisa, ki veleval, da gre za obvezno sestavino vsakega predavanja o rekurziji. Ne. Privlekli smo ga zaradi zvezdic.

[45]:
```
def razporedov(n):
    print("*")
    if n == 0:
        return 1
    if n == 1:
        return 2
    return razporedov(n - 1) + razporedov(n - 2)
```

[46]:
```
razporedov(5)
```

```
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
```

[46]:
```
13
```

[47]:
```
razporedov(6)
```

```
*
```

```
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
```

[47]: 21

Spet je počilo. Eksplodiralo. Vendar tokrat nismo nič krivi. Prej smo brez potrebe dvakrat zapored poklicali isto funkcijo z istimi argumenti. Zdaj ne - zdaj jo pokličemo z različnimi.

Razlog za eksplozijo je, seveda, v tem, da pri reševanju za `n` stolov pokličemo `n - 1` in `n - 2`. Za `n - 1` pa spet pokličemo `n - 2`. Vsak od dveh klicev za `n - 2` pokliče `n - 3` (ki ga sicer pokliče tudi `n - 1`) tako naprej. (Razmislite, koliko klicev je to. Namig: odgovor bo spet povezan s Fibonacijevimi števili, le ritensko.)

Ko gre za Fibonacijeva števila, je rešitev očitna: namesto da bi jih računali rekurzivno, nazaj, jih računamo iterativno, naprej, tako kot vsi normalni ljudje in to že nekje od drugega tedna predavanj.

Zgodi pa se, da naletimo na nalogo - študenti računalništva jih vidijo nesluteno število v drugem letniku pri Algoritmih in podatkovnih strukturah, predvsem pri dinamičnem programiranju - kjer je računanje nazaj, z rekurzijo, veliko preprostejše kot iterativno, naprej.

2.9 Memoizacija

Tu nas reši trik: funkcija si mora zapomniti rezultat klica z določenimi argumenti. Ko jo naslednjič pokličemo z enakimi argumenti, ničesar ne računa, temveč le vrne, kar je naračunala in vrnila v prejšnjem klicu s takimi argumenti.

Kako izvesti *memoizacijo*, je odvisno od jezika. V Pythonu funkcijo dekoriramo, ovijemo v drugo funkcijo, ki prestreza klice in pomni rezultate. Dekoriranje funkcij v Pythonu je splošnejši meh-

anizem, ki ga lahko uporabimo za marsikaj. O njem se ne bomo učili (tole je vendarle tečaj programiranja, ne Pythona), tule bomo le pogledali, kaj nam je storiti, da preprečimo pretirano število rekurzivnih klicev.

[48]:
```
from functools import cache

@cache
def razporedov(n):
    print("*")
    if n == 0:
        return 1
    if n == 1:
        return 2
    return razporedov(n - 1) + razporedov(n - 2)
```

[49]:
```
razporedov(6)
```

```
*
*
*
*
*
*
*
```

[49]: 21

Šest stolov, sedem zvezdic. Zadnja je za 0 stolov.

Skratka: pred funkcijo napišemo `@cache`, pri čemer je `cache` funkcija, ki jo je potrebno uvoziti iz `functools`.

Tehnični detajl, ki se ga je potrebno zavedati: `cache` deluje tako, da shranjuje argumente in pripadajoče rezultate v slovar. (Očitno. Kam drugam bi jih lahko? Le tako lahko hitro preveri, ali je določene argumente že kdaj videl in kakšen je bil pripadajoči rezultat.) Ker pa morajo biti ključi slovarja nespremenljive reči, dekoratorja `cache` ne moramo uporabiti za funkcije, ki sprejemajo sezname, slovarje ali množice (ali kaj drugega nespremenljivega). V tem primeru spremenimo funkcijo tako, da namesto seznamov sprejema terke, namesto množic zamrznjene množice (`frozenset`), namesto slovarjev pa, tough luck, sezname parov. No, slednje se mi še ni zgodilo, tako da ne boli preveč.