10c - Rekurzija naprej in nazaj

January 28, 2024

1 Rekurzija naprej in nazaj

1.1 Potenčna množica

Napišimo funkcijo `potencna(s)`, ki za podani seznam elementov `s` vrne seznam vseh njegovih podseznamov. `potencna([1, 2, 3])` mora vrniti `[[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]`. Vrstni red ni pomemben. Namesto s seznami bi lahko delali z množicami, vendar ne bomo, ker je s seznami lažje; tako ali tako bomo imeli dovolj drugih težav.

1.1.1 Ne preveč lepa rekurzija: nabiranje

Če bi dal to nalogo študentom, bi jo večinoma rešili takole. (No, večinoma je ne bi znali rešiti; tisti, ki bi jo, pa bi jo tako.)

Najprej rešimo enostavnejšo nalogo: izpišimo vse možne podsezname. Funkcija bo imela dva argumenta: seznam `s` in še vrečo (`vreca`), v katero bomo nabirali elemente.

- Najprej sestavimo vse podmnožice `s[1:]` (torej `s` brez prvega elementa), ki vsebujejo `s[0]`: pokličemo `potencna(s[1:], vreca + [s[0]])`, s čimer vržemo `s[0]` v vrečo (k tistemu, kar se je morda že nabralo v njej) in prepustimo rekurzivnemu klicu, da vanjo doda - ali pa ne - še kaj od preostalih elementov.
- Nato sestavimo še podmnožice `s[1:]`, ki ne vsebujejo `s[0]`: pokličemo `potencna(s[1:], vreca)`, tako da se bo v vreči morda nabralo kaj iz `s[1:]`, `s[0]` pa ne.

Ko je `s` prazen, se je rekurzija iztekla in le še izpišemo, kar se je nabralo v vreči.

[1]:
```
def potencna(s, vreca):
    if not s:
        print(vreca)
    else:
        potencna(s[1:], vreca + [s[0]])
        potencna(s[1:], vreca)

potencna([1, 2, 3, 4], [])
```

```
[1, 2, 3, 4]
[1, 2, 3]
[1, 2, 4]
[1, 2]
[1, 3, 4]
[1, 3]
```

```
[1, 4]
[1]
[2, 3, 4]
[2, 3]
[2, 4]
[2]
[3, 4]
[3]
[4]
[]
```

Zdaj pa predelajmo funkcijo tako, da bo vrnila seznam seznamov, namesto da jih izpisuje.

- Če je `s` prazen, mora očitno vrniti seznam, ki vsebuje en podseznam, namreč vse, kar se je nabralo v vreče; `[vreca]`.
- Sicer pa pokliče `potencna(s[1:], vreca)` in `potencna(s[1:], vreca + [s[0]])`. Tako dobi seznam vseh podmnožic brez `s[0]` in z `s[0]`. Sešteje in vrne.

[2]:
```
def potencna(s, vreca):
    if not s:
        return [vreca]
    else:
        return potencna(s[1:], vreca) + potencna(s[1:], vreca + [s[0]])

potencna([1, 2, 3, 4], [])
```

[2]:
```
[[],
 [4],
 [3],
 [3, 4],
 [2],
 [2, 4],
 [2, 3],
 [2, 3, 4],
 [1],
 [1, 4],
 [1, 3],
 [1, 3, 4],
 [1, 2],
 [1, 2, 4],
 [1, 2, 3],
 [1, 2, 3, 4]]
```

1.1.2 Lepa rekurzija: pobiranje

Tako, kot smo naredili zgoraj, naredimo, kadar imamo za to kak razlog.

Problem gornjega razmišljanja je, da razmišljamo naprej namesto nazaj. Vzeli smo prvi element, naredili odločitev v zvezi z njim in mu potem pridružili potenčno množico ostanka. Rekurzijo je

boljše načrtovati v drugo smer. Najprej rešimo preprostejši problem (potenčna množica seznama brez prvega elementa), nato pa rešitev razširimo v rešitev, ki upošteva tudi ta, prvi element.

Problem je očitno najpreprostejši, če je seznam `s` prazen. V tem primeru potenčna množica vsebuje eno samo množico, namreč prazno množico.

```
    if not s:
        return [[]]
```

Sicer pa naredimo tole: najprej sestavimo potenčno množico `s`-ja brez prvega elementa, `brez = potencna(s[1:])`. Dobili smo … pač nekaj množic. Zdaj ta seznam podvojimo: enkrat ga vzamemo takšnega kot je (`brez`), enkrat pa k vsem njegovim elementom dodamo še `s[0]` (`[[s[0]] + b for b in brez]`). To potem vrnemo.

[3]:
```
def potencna(s):
    if not s:
        return [[]]
    else:
        brez = potencna(s[1:])
        return brez + [[s[0]] + b for b in brez]

potencna([1, 2, 3, 4])
```

[3]:
```
[[],
 [4],
 [3],
 [3, 4],
 [2],
 [2, 4],
 [2, 3],
 [2, 3, 4],
 [1],
 [1, 4],
 [1, 3],
 [1, 3, 4],
 [1, 2],
 [1, 2, 4],
 [1, 2, 3],
 [1, 2, 3, 4]]
```

Ta rešitev je obrnjena v pravo smer in zato precej krajša in brez dodatnega argumenta, vreče. Poleg tega pa ima še eno lepo lastnost. "Grda" različica dvakrat pokliče sama sebe. Za potenčno množico n elementov se pokliče 2n-krat. Ta, druga, se pokliče le enkrat. Za potenčno množico petih elementov se pokliče n-krat.

To sicer ne pomeni (nujno), da je hitrejša: prva različica vsebuje le eno seštevanje, druga pa zanko. Na koncu koncev morata obe sestaviti množico, ki vsebuje 2n elementov, kar bo moralo zahtevati 2n korakov. No, v praksi bo druga hitrejša zato, ker so rekurzivni klici počasnejši od zanke.

A hitrost še niti ni tako pomembna: kar tule šteje, je eleganca. :)

1.2 Akumulator

Očitna razlika med gornjima funkcijama je v tem, da ima prva še dodaten argument, v katero se tekom rekurzivnih klicev nabirajo vrednosti, nekakšen "akumulator". Kako ta princip deluje v splošnem, bomo najlepše videli ob funkciji, ki vrne vsoto elementov seznama.

[4]:
```
def vsota(s, a):
    if s == []:
        return a
    else:
        return vsota(s[1:], a + s[0])

vsota([5, 3, 4], 0)
```

[4]:
```
12
```

Tu funkcija rekurzivno kliče samo sebe in ob vsakem klicu nekaj prišteje k drugemu argumentu. Ko se rekurzija izteče, funkcija vrne, kar se je nabralo.

Funkcija ima dodatni argument. Ker gre za število, torej `int`, ki je nespremenljiv (*immutable*) se tu z lahkoto zmažemo tako, da mu damo privzeto vrednost.

[5]:
```
def vsota(s, a=0):
    if s == []:
        return a
    else:
        return vsota(s[1:], a + s[0])

vsota([5, 3, 4])
```

[5]:
```
12
```

Trik s privzeto vrednostjo bo včasih vžgal, včasih pa se bo potrebno malo bolj potruditi. A to ni poanta. Primerjajmo to funkcijo z vsoto, ki smo jo sprogramirali na predavanjih (ali pa je vsaj v zapiskih o rekurziji na seznamih).

[6]:
```
def vsota(s):
    if s == []:
        return 0
    return vsota(s[1:]) + s[0]

vsota([5, 3, 4])
```

[6]:
```
12
```

Tu gre za popolnoma drugačno filozofijo: tu vsote ne nabiramo tja grede, temveč jo dopolnjujemo nazaj grede. Takšna rekurzija je tipično lepša.

1.3 Repna rekurzija

V kontekstu vsega tega lahko povemo še nekaj o malo naprednejši temi: repni rekurziji.

Problem rekurzije je, da si mora Python (ali katerikoli jezik že) beležiti, kdo je poklical koga, s kakšnimi argumenti in kakšne so njegove lokalne spremenljivke. (Tehnično: strukturi, v katero se to shranjuje, se reče *sklad*.) To jemlje pomnilnik. In čas. Python pusti 1000 nivojev rekurzije; ne sicer toliko zaradi pomnilnika, temveč zato, ker ob 1000 nivojih predpostavi, da je šlo nekaj narobe in se je program zaciklal. (To mejo je mogoče spremeniti s klicem `sys.setrecursionlimit`.)

Včasih pa je to v bistvu nepotrebno. Če je rekurzivni klic čisto zadnja operacija, ki se zgodi, si ni potrebno zapomniti lokalnih spremenljivk in vsega v zvezi s trenutno funkcijo. Namesto, da bi se izvajanje po rekurzivnem klicu vrnilo v to funkcijo, ki ga je naredila, se lahko vrne v kličočo funkcijo.

Se pravi: recimo, da imamo prvo različico funkcije `vsota`, tisto, ki ima dva argumenta. Recimo, da znotraj neke funkcije `f` pokličemo funkcijo `vsota([5, 3, 4], 0)`. V okviru tega se pokliče `vsota([3, 4], 5)`. Rezultat tega klica bo že končni rezultat, zato se lahko pravzaprav vrne kar neposredno funkciji `f` in ne "funkciji" `vsota([5, 3, 4], 0)`, ki ga bo tako ali tako nespremenjenega samo vrnila funkciji `f`. Reč gre naprej: `vsota([3, 4], 5)` pokliče `vsota([4], 8)`. Rezultat tega klica bo 12 in bi ga spet lahko vrnili kar direktno funkciji `f`, ne pa tistemu, ki je klical. Klic `vsota([4], 8)` pokliče `vsota([], 12)`. Ta klic naredi samo `return a`. Ta `return` bi lahko letel naravnost v `f`.

Optimizacijski trik, ki lahko naredi prevajalnik, je torej ta, da ob rekurzivnem klicu pozabi na vmesni klic, temveč preprosto zamenja vrednosti argumentov. Kot sem napovedal na začetku, je ta tema nekoliko zahtevnejša in je zato ni tako lahko razložiti v enem odstavku, sploh, če ne vemo, kako delujejo programski jeziki in zato ne moremo uporabljati primernih izrazov. :) Če bo kaj pomagalo, je tule povezava na Wikipedijo.

Na ta način se rekurzivni klici bistveno pospešijo: rekurzija je navadno kar znatno počasnejša od ekvivalentnega nerekurzivnega programa (kar nikakor ne pomeni, da je ne uporabljamo, kot so me nekoč narobe razumeli študenti in me špecali kolegom v višjih letnikih!). S tem trikom je praktično enako hitra.

Ta trik je možen pri funkcijah, napisani na prvi način, z akumulatorjem, pri drugi funkciji pa ne, ker mora po rekurzivnem klicu še nekaj prišteti k rezultatu. Prva oblika, ki sem jo označil kot gršo, ima torej včasih praktično prednost pred drugo.

Slaba novica: Python tega ne podpira. Tudi če napišemo funkcijo, ki bi se jo dalo optimizirati na ta način, Python tega ne bo storil. Ker se avtorjem ne zdi pomembno vlagati truda v to. Python ni jezik, ki bi temeljil na rekurziji. Pač pa je to tipično za jezike, ki nimajo običajnih zank - tipično so to jeziki, ki nimajo spremenljivk, temveč le konstante in je vsako prirejanje "dokončno".