DUOMENŲ STRUKTŪROS IR ALGORITMAI

MARIUS GŽEGOŽEVSKIS

REKURSIJA

Rekursija yra viena iš pirminių ir pagrindinių matematikos ir informatikos sąvokų. Rekursija matematikoje vadinamas toks funkcijų apibrėžimo metodas, kai funkcijos reikšmės, atitinkančios bet kokius argumentus, yra apibrėžiamos naudojant tos pačios funkcijos reikšmes, atitinkančias mažesnius argumentus.

REKURSIJA

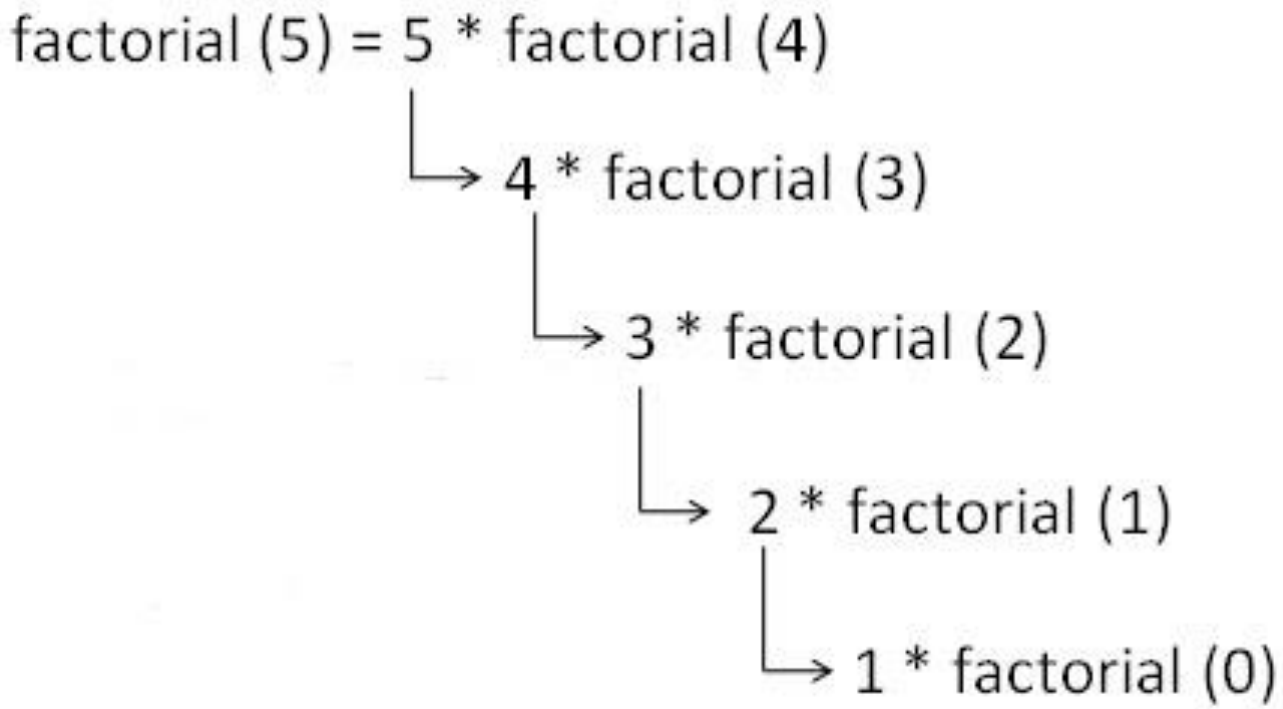
Analogiškai programavime, rekursija yra programų (procedūrų, algoritmų) apibrėžimo (sudarymo) metodas, kai programa kreipiasi pati į save, esant mažesnėms argumentų (parametrų) reikšmėms. Rekursyviai programai reikia papildomai apibrėžti tą atvejį, kai pasiektos mažiausios galimos argumentų reikšmės (baigmės sąlyga).

REKURSIJA

Rekursyvių programų pavyzdžiai medžiams numeruoti ar viršūnėms perrinkti jau buvo pateikti anksčiau. Žinomi klasikiniai rekursyvių algoritmų pavyzdžiai - tai faktorialo skaičiavimas ar Fibonačio skaičių sekos apibrėžimas:

```
3   def faktorialas(n):
4       if n == 0: return 1
5
6       else: return n * faktorialas(n-1)
7
8   print(faktorialas(3))
```

5 faktorialo skaičiavimas naudojant rekursiją



Fibonačio skaičių seka

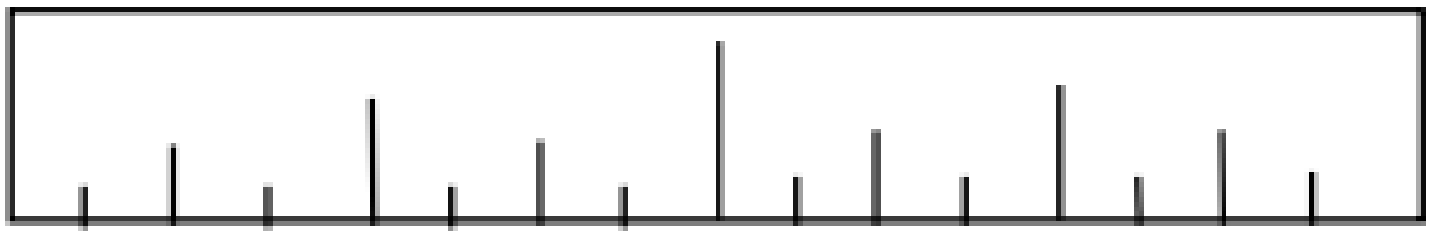
$$F_n = \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n > 1. \end{cases}$$

```
3   def f():
4       n = int(input("Iveskite skaičių:"))
5       print(fib(n))
6
7   def fib(n):
8       if n == 0: return 0
9
10      elif n == 1: return 1
11
12      else: return fib(n-1)+fib(n-2)
13  f()
```

Rekursijos stiprybė - ji įgalina žymiai suprastinti algoritmo formulavimą ar programos konstrukciją. Tačiau ja naudoti reikia atsargiai. Pavyzdžiui, Fibonačio skaičių sekos apibrėžimas naudojant rekursiją, kaip ką tik pateiktame pavyzdyje, šią programą daro neefektyvią: skaičiai fibonacci(N-1) ir fibonacci(N-2) gaunami nepriklausomai vienas nuo kito, tuo tarpu juos nesunku gauti vieną iš kito. Fibonačio algoritmas, formuluojamas be rekursijos, yra efektyvesnis (operacijų skaičius proporcingas N):

Rekursija glaudžiai susijusi su tam tikra algoritmų konstravimo metodika, vadinamąja skaldyk ir valdyk (angl. divide and conquer) paradigma. Pagal tokią paradigmą sukurtas algoritmas naudoja du rekursyvius programos kvietimus, tačiau nesikertančioms argumentų reikšmėms. Jos pavyzdys - pateikiama liniuotės gradavimo programa:

```
3  def liniuotė(l, r, h):
4      if h > 0:
5          m = (l+r) // 2
6          žymė(m, h)
7          liniuotė(l, m, h-1)
8          liniuotė(m, r, h-1)
9  print(liniuotė(10, 20, 30))
```



"Skaldyk ir valdyk" paradigma apima tris tipinius kiekvieno rekursijos lygio žingsnius:

1. išskaidyti uždavinį į kelis dalinius uždavinius;
2. valdyti rekursyviai kiekvieno dalinio uždavinio sprendimą, jei uždavinys tapo paprastas, išspręsti jį tiesiogiai;
3. sujungti dalinių uždavinių sprendinius ir suformuoti išeities uždavinio sprendinį.

"Skaldyk ir valdyk" paradigma

Ši paradigma taikoma algoritmams, kaip operacijų sekoms, ir duomenų struktūroms konstruoti. Kiti du plačiai naudojami metodai - tai rekursija ir iteracija. Visos toliau nagrinėjamos struktūros ar su jomis susiję algoritmai bus pagrįsti vienu iš šių metodų.

REKURSIJA IR ITERACIJA

Tiek rekursija, tiek iteracija reiškia veiksmų kartojimą:

1. iteracija kartojimo scenarijų naudoja išreikštiniu būdu,
2. rekursija – neišreikštiniu būdu, pakartotinai iškviesdama funkcijos kopijas.
3. Tiek rekursija, tiek iteracija atlieka pabaigos tikrinimą: iteracija baigiasi, kai ciklo kartojimo sąlyga yra patenkinama.
4. rekursija yra baigiama, yra pasiekiamas pagrindinis atvejis.

REKURSIJA IR ITERACIJA

Abiem atvejais rezultatas yra pasiekiamas pažingsniui:

1. iteracija keičia ciklo skaitliuką,
2. rekursija gamina vis paprastesnį atvejį.

Abiem atvejais kartojimai gali būti amžini:

1. iteracija gali patekti į amžiną ciklą, nes ciklo skaitliukas nepasiekia ribinių reikšmių
2. rekursija gali tęstis amžinai, jei žingsnis nesuprastina užduoties iki pagrindinio atvejo.

REKURSIJOS TRŪKUMAI

- Jis daug kartų inicijuoja funkcijos iškvietimo mechanizmą ir eikvojami tam skirti resursai, būtent procesoriaus **laikas** ir **atmintis**;
- kiekvienam iškvietimui kintamiesiems yra skiriama atmintis ir tai gali reikalauti daug atminties išteklių;
- **Iteracija paprastai** yra atliekama vienos funkcijos ribose, todėl šių resursų ji nereikalauja. Kaip rinktis kartojimo aprašymo būdą?

Kaip rinktis kartojimo aprašymo būdą? REKURSIJA AR ITERACIJA ?

- Rekursiniai algoritmai yra geresni tuomet, kai jie natūraliau aprašo užduoties sprendimo eigą.
- Kai kada iteracijos sprendinys gali būti neakivaizdus, tuomet geriau yra rinktis rekursinį algoritmą.
- Jei yra reikalaujama, kad užduoties sprendimas būtų efektyvus, tuomet yra renkamasi iteraciją.

Rekursija – Hanojaus bokštų uždavinys

Duoti trys stulpai ir N diskų. Ant vieno (pradinio) stulpo sumauti diskai didėjimo tvarka, einant iš viršaus į apačią. Reikia visus diskus perkelti nuo pradinio stulpo ant laisvo (tikslo) stulpo, pasinaudojant atsarginiu stulpu.

APRIBOJIMAI

Per vieną ėjimą galima nuimti tik vieną diską ir jį būtina iš karto uždėti ant kito stulpo. Didesnio disko negalima dėti ant mažesnio.

Visai nesunku sugalvoti rekursinį šio uždavinio sprendimo algoritmą:

Jei diskas tik vienas (N=1), uždavinys elementarus – tiesiog perkeliame šį 1 diską nuo pradinio stulpo ant tikslo stulpo.

APRIBOJIMAI

Tarkime, kad mes mokame išspręsti uždavinį su N-1 disku. Tada norėdami perkelti N diskų elgiamės taip:

1. N-1 diską nuo pradinio stulpo perkeliame ant atsarginio stulpo (pagal prielaidą tai mes jau mokame padaryti);
2. tada ant pradinio stulpo likusį 1 diską (didžiausią) tiesiog perkeliame ant tikslo stulpo;
3. N-1 diską nuo atsarginio stulpo perkeliame ant tikslo stulpo (pagal prielaidą tai mes jau mokame padaryti).

HANOJAUS BOKŠTŲ UŽDAVINIO REALIZAVIMO BŪDAI

Žemiau pateikiami 2 variantai šį algoritmą realizuojančios rekursinės procedūros pseudokodo (naudojami pažymėjimai: ***count*** - diskų skaičius, ***source*** - pradinis stulpas, ***dest*** - tikslo stulpas, ***spare*** - tarpinis stulpas):

1 BŪDAS

```
Towers (count, source, dest, spare)
if count = 1 then tiesiog perkelti 1 diską nuo pradinio stulpo (source)
ant tikslo stulpo (dest)
 else
  begin
     Towers (count-1, source, spare, dest)
     Towers (1, source, dest, spare)
     Towers (count - 1, spare, dest, source)
  End
```

2 BŪDAS

```
Towers (count, source, dest, spare)
if (count > 0)
  begin
        Towers (count - 1, source, spare, dest)
        perkelti 1 diską nuo pradinio stulpo (source) ant tikslo stulpo
  (dest)
     Towers (count - 1, spare, dest, source)
  end
```

Dabar nustatysime, kiek reikės žingsnių N diskų perkėlimui. Pažymėkime **žingsniai(N)** žingsnių skaičių, reikalingą perkelti N diskų.

Kai N = 1, tai **žingsniai(1)** = 1.

Kai N > 1, procedūra *Towers* iškviečiama 3 kartus: perkelti N-1 diską, perkelti 1 diską ir vėl perkelti N-1 diską. Taigi galime apskaičiuoti **žingsniai(N)** pagal tokią rekurentinę formulę:

žingsniai(N) = **žingsniai(N-1)** + **žingsniai(1)** + **žingsniai(N-1)** = 2 * **žingsniai(N-1)** + 1

Žingsnių skaičių galima apskaičiuoti pagal tokią formulę:

žingsniai(N) = 2^N - 1 ($\forall N \geq 1$)

Šios formulės teisingumą įrodysime matematinės indukcijos metodu.

Patikriname jos teisingumą, kai N=1.

žingsniai(1) = 2^1 - 1 = 2 - 1 = 1

Tarkime, kad ji teisinga su N-1 (t.y. **žingsniai(N-1)** = $2^{N-1} - 1$), tada:

žingsniai(N) = 2 * **žingsniai(N-1)** + 1 = 2 * ($2^{N-1} - 1$) + 1 = $2^N - 2 + 1 = 2^N$ - 1

Įrodymas baigtas.

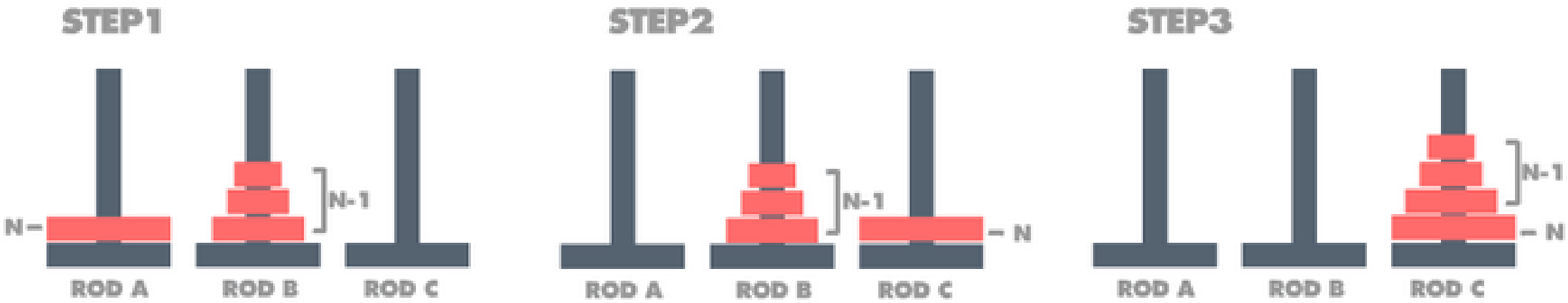
HANOJAUS BOKŠTŲ UŽDAVINIO REALIZACIJA PYTHON PROGRAMAVIMO KALBA

```
def hanoi(ndisks, startPeg=1, endPeg=3):
    if ndisks:
        hanoi(ndisks-1, startPeg, 6-startPeg-endPeg)
        print "Move disk %d from peg %d to peg %d" % (ndisks, startPeg, endPeg)
        hanoi(ndisks-1, 6-startPeg-endPeg, endPeg)

hanoi(ndisks=4)
```

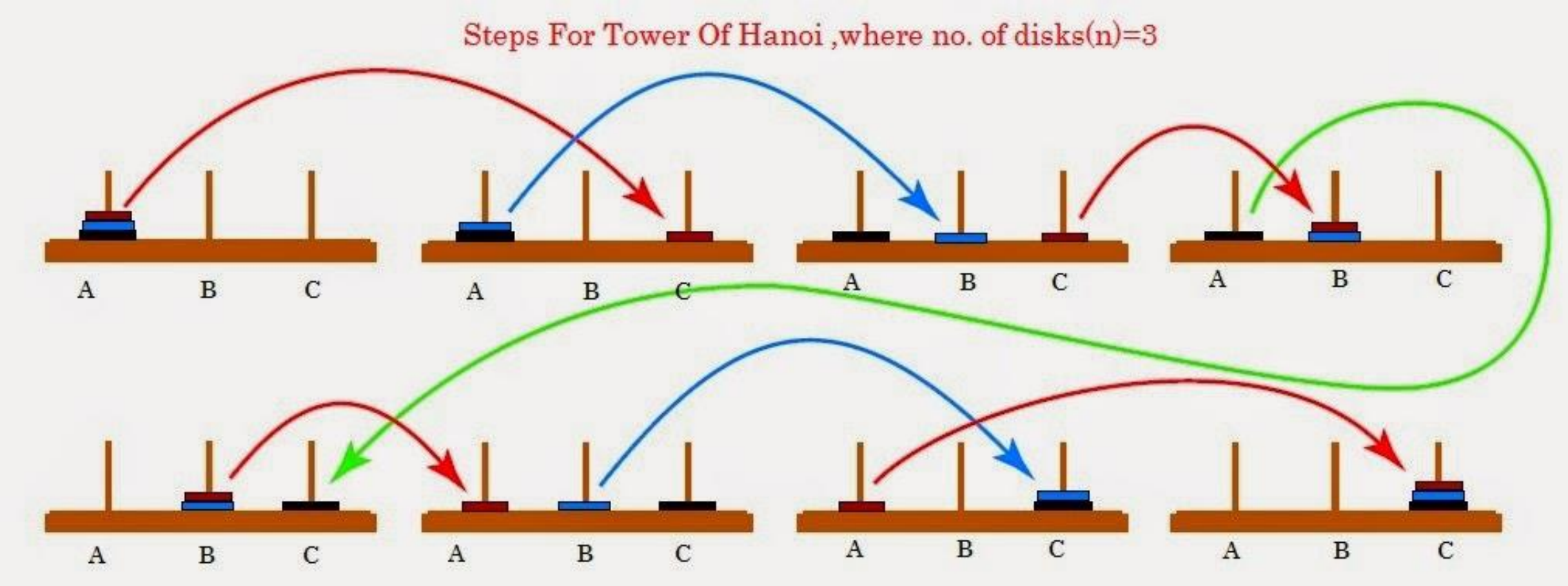
for ndisks=2

```
Move disk 1 from peg 1 to peg 2
Move disk 2 from peg 1 to peg 3
Move disk 1 from peg 2 to peg 3
```



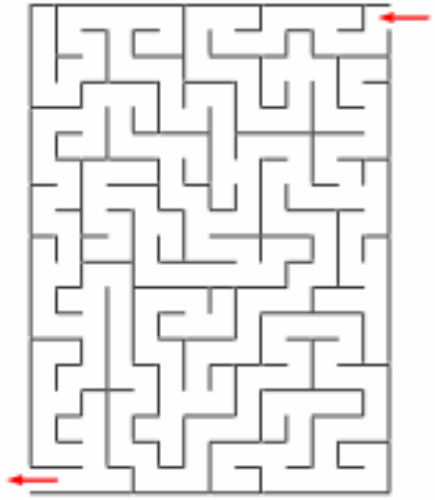

```
FUNCTION SOLVE(N, SOURCE, DEST, SPARE)
IF N == 1, THEN
    move N from SOURCE to DEST
ELSE
    SOLVE(N - 1, SOURCE, SPARE, DEST)   STEP1
    move N from SOURCE to DEST          STEP2
    SOLVE(N - 1, SPARE, DEST, SOURCE)   STEP3
END IF
```

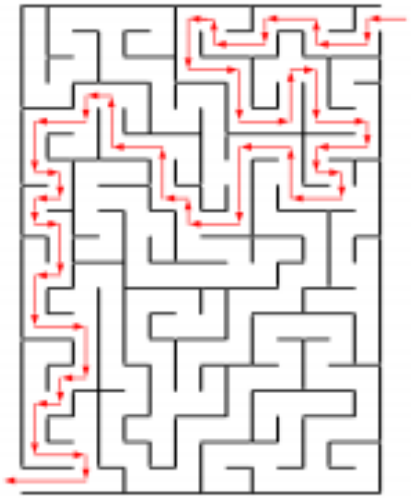
http://www.mclelun.com/labs/#as3toh

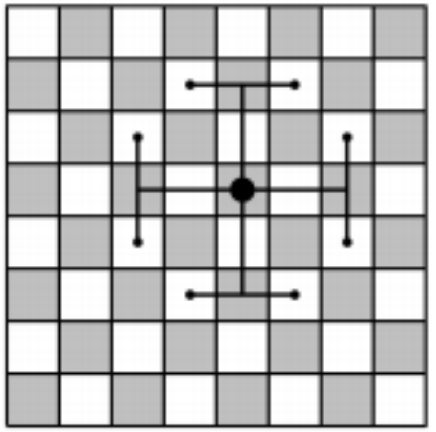

Steps For Tower Of Hanoi ,where no. of disks(n)=3
A
B
C
A
B
C
A
B
C
A
B
C
A
B
C
A
B
C
A
B
C
A
B
C

Variantų perrinkimas grįžtant atgal

Rekursija yra labai naudinga sprendžiant uždavinius variantų perrinkimu su grįžimu atgal (angl. backtrack). Pavyzdžiui,







55	42	15	12	57	32	17	10
14	1	56	43	16	11	34	31
41	54	13	64	33	58	9	18
2	25	46	51	44	63	30	35
47	40	53	26	59	36	19	8
24	3	50	45	52	27	62	29
39	48	5	22	37	60	7	20
4	23	38	49	6	21	28	61

1) Kaip pereiti per labirintą? 2) Žirgo maršrutas šachmatų lentoje? Sudarysime rekursinį algortimų šabloną (angl. template) tokių uždavinių sprendimui.

Variantų perrinkimo šablonas grįžtant atgal

- Sprendžiant konkretų uždavinį turime apibrėžti duomenų struktūrą pozicija, kurioje saugosime informaciją apie uždavinio sprendimo eigą (pvz., praeitą maršrutą labirinte arba šachmatų lentoje). Sprendinio paieška tęsiama iš P taško (pvz., P - lentos langelis).

- Turime apibrėžti funkciją S=BandymųAibė(P, pozicija), kuri generuoja visus naujus ėjimus, kuriuos galima atlikti iš taško P. Leistinų ėjimų aibę S apibrėžia uždavinio sąlygos ir duomenų struktūroje pozicija saugoma informacija apie jau aplankytas paieškos vietas.

Variantų perrinkimo šablonas grįžtant atgal

- Jei padarę kažkurį ėjimą gauname, kad naujos pozicijos S yra tuščia arba netiko nei vienas iš naujos S ėjimų (pranešime apie tai grąžindami reikšmę 0), tai turime grįžti atgal, atstatyti seną poziciją (SenaPozicija(U, pozicija)) ir tikrinti kitą leistiną ėjimą.

- Kai surandame sprendinį RadomeSprendinį (pozicija) == taip, grąžiname reikšmę 1, taip panešdami šią funkciją iškvietusiai funkcijai, kad sprendinys rastas ir ji irgi gali nutraukti savo darbą grąžindama 1.

Rekursinis šablonas

```
int   Tikrink (int n, Point P, Inf pozicija)
begin
  (1)   if   ( RadomeSprendinį (pozicija) == taip ) then
  (2)        Spausdink (pozicija);
  (3)        return(1);
  (4)   else
  (5)        S = BandymųAibė(P, pozicija)
  (6)        while   ( S ≠ ∅ ) do
  (7)                U = NaujasBandymas(S);
  (8)                NaujaPozicija(U, pozicija);
  (9)                if   ( Tikrink (n+1, U, pozicija) == 1 ) return (1);
  (10)               else
  (11)                  SenaPozicija(U, pozicija);
                     end if
             end do
  (12)       return (0);
        end if
end   Tikrink
```

Skaldyk ir valdyk metodas

Daugelio uždavinių efektyvius sprendimo algoritmus sudarome tokiu metodu:

1. Uždavinį skaidome į kelis mažesnius uždavinius.
2. Randame šių uždavinių sprendinius.
3. Iš jų sudarome viso uždavinio sprendinį.

Skaldyk ir valdyk metodas

Dalinius uždavinius vėl galime spręsti tokiu pačiu metodu. Taip skaidome tol, kol gautieji uždaviniai yra lengvai išsprendžiami. Taip gauname rekursinį skaldyk ir valdyk metodo (angl. **divide and conquer**) algoritmą. Sprendžiant konkretų uždavinį reikia apibrėžti, kaip realizuojami šie trys žingsniai. Tai galima daryti įvairiai, todėl dažnai tam pačiam uždaviniui skaldyk ir valdyk metodu galime sudaryti kelis skirtingus sprendimo algoritmus.

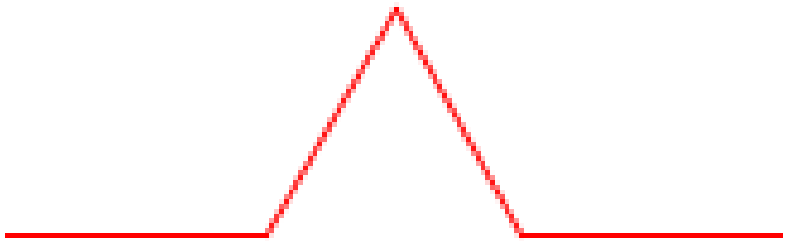
Rekursinių skaldyk ir valdyk algortimų šablonas

Sprendžiame uždavinį, kurį apibūdina duomenų struktūrą A.

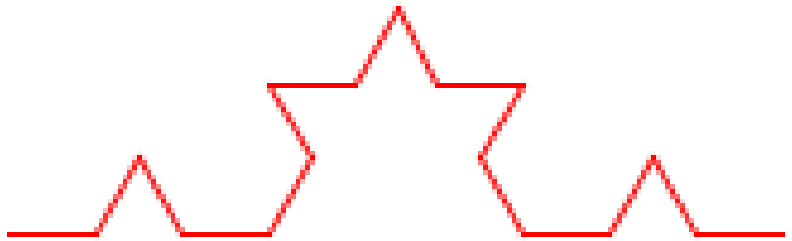
```
SprendTipas   SkaldykIrValdyk (Inf A)
begin
  (1)  if  ( |A| > ε ) then
  (2)      (A_1, A_2, ..., A_M) = Skaldyk (A);
  (3)      for all ( A_j, j=1, 2,..., M ) do
  (4)              S_j = SkaldykIrValdyk (A_j);
           end do
  (5)      S = Valdyk (S_1, S_2, ..., S_M);
       else
  (6)      S = Sprendinys(A)
       end if
  (7)  return (S);
end   SkaldykIrValdyk
```

REKURSIJOS PAVYZDŽIAI

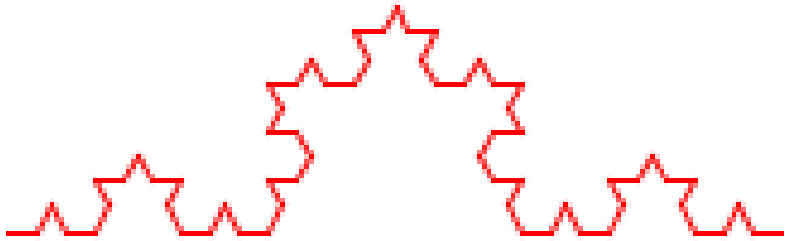
recursionLevel = 1



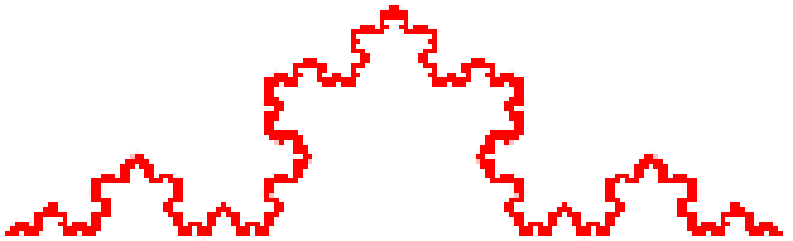
recursionLevel = 2



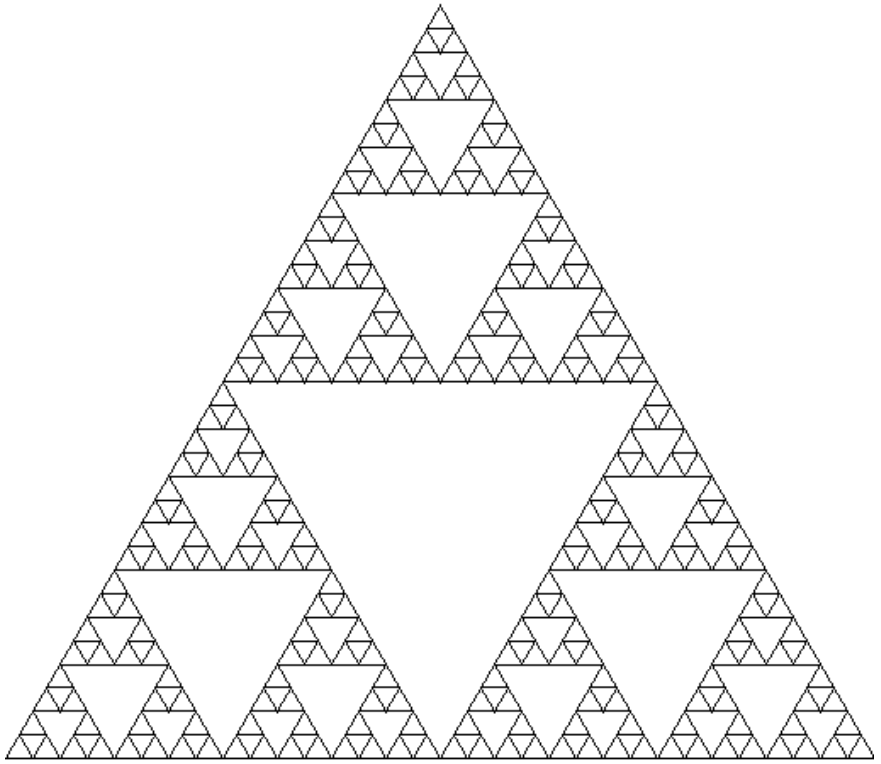
recursionLevel = 3



recursionLevel = 8



REKURSIJOS PAVYZDŽIAI



REKURSIJOS PAVYZDŽIAI