Яндекс Лицей

Урок Recursion

Ռեկուրսիա

Անոտացիա

Այս դասում մենք կծանոթանանք ռեկուրսիայի հասկացության հետ, ցույց կտանք թե ինչպես կարելի է այն կապել արդեն մեզ ծանոթ կառուցվածքների (ցիկլերի և ֆունկցիաների) հետ: Ինչպես նաև կքննարկենք առավել հաճախ հանդիպող սխալները և մի քանի «դասական» օրինակներ:

1. Զուգորդությունների վերաբերյալ խնդիրներ

Այդ խնդրի պատասխանը կլինի հետևյալ մեծությունը.

$$C_n^k = \frac{n!}{k! * (n-k)!}$$

որն անվանում են n տարրերից k-ական զուգորդություն: n! գրառումը նշանակում է $1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$, որը կոչվում է n թվի ֆակտորիալ (մենք արդեն բազմիցս հանդիպել ենք այս հասկացության հետ), ընդ որում համարվում է, որ 0! = 1

```
# Խնդիր զուգորդությունների վերաբերյալ


def f(n):
    res = 1
    for i in range(2, n + 1):
        res *= i
```

Чаты

```
    return res


time=(f(10) / (f(3) * f(10 - 3))) * 2
print(time)
```

2. Ռեկուրսիայի սահմանումը, աշխատանքի սկզբունքները

Սակայն այս խնդիրը կարելի է լուծել մեկ այլ եղանակով: Մաթեմատիկայում հաճախ հաշվարկները պարզեցնելու նպատակով խնդիրը ներկայացնում են ավելի պարզ տեսքով:

Ռեկուրսիա

Արդյունքում կարելի է գալ նրան, որ կստանանք նախնական խնդիրը, բայց ավելի պարզեցված տեսքով: Այս հնարքը կոչվում է ռեկուրսիա, լատիներեն recurcio - «անդրադարձ» բառից: Այսպիսով, ծրագրավորման մեջ ռեկուրսիան ենթածրագիր է, որը կանչում է ինքն իրեն (անմիջական եղանակով կամ ենթածրագրերի շղթայի միջոցով):

Վերադառնանք մեր խնդրին և դիտարկենք ֆակտորիալի հաշվման խնդիրը մեկ այլ տեսանկյունից. փորձենք կիրառել ռեկուրսիա: Հայտնի է, որ 0! = 1, 1! = 1: Իսկ ինչպե՞ս հաշվել n! արժեքը մեծ n-երի դեպքում: Եթե մեզ հաջողվեր հաշվել (n-1)!, մենք հեշտությամբ կհաշվեինք նաև n!, քանի որ n! = n · (n-1)!: Իսկ ինչպե՞ս հաշվել (n-1)!: Եթե մեզ հաջողվեր հաշվել (n-2)!, մենք հեշտությամբ կհաշվեինք նաև (n-1)!, քանի որ (n-1)! = (n-1) · (n-2)!: Իսկ ինչպե՞ս հաշվել (n-2)!: Եթե ... Վերջ ի վերջո, մենք կհասնենք 0!, որը հավասար էր 1: Այսպիսով, ֆակտորիալի հաշվման համար կարող ենք օգտագործել ավելի փոքր թվի ֆակտորիալը:

Գրենք համապատասխան ֆունկցիա.

```
# Ֆակտորիալի հաշվում


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

Ռեկուրսիվ ֆունկցիաների տրամաբանական բարդությունը կայանում է ենթածրագրի հաջորդական ինքն իրեն դիմելու արդյունքում պարամետրերի փոփոխության և միջանկյալ արժեքների ստացման առանձնահատկությունների մեջ: Իրականացվում են քայլերի երկու հաջորդականություններ: Առաջին հաջորդականությունը ծրագրի ինքն իր մեջ խորացման քայլերն են, մինչև ընտրված պարամետրը հասնի իր սահմանային արժեքին (ռեկուրսիայի խորություն): Երկրորդ հաջորդականությունը ռեկուրսիվ ելքի քայլերն են մինչև ընտրված պարամետրի արժեքը ընդունի իր նախնական արժեքը: Սա էլ, որպես կանոն, ապահովում է միջանկյալ և վերջնական արդյունքների ստացումը: Վերը նշվածի լավ օրինակ է հետևյալ ծրագիրը.

```
def depth(n):
    print('Խորացում ', n)
    if (n > 1):
        depth(n - 1)
    else:
        print('_________________')
    print('Ելք', n)


depth(6)
```

Գործնականում անհրաժեշտ է համոզվել, որ ռեկուրսիայի խորությունը մեծ չէ:

3. Ռեկուրսիվ սահմանումներ: Ռեկուսրսիան, որպես ցիկլի ընհանրացում

Ռեկուրսիվ սահմանում

Ռեկուրսիվ սահմանումը մաթեմատիկայում հաճախ օգտագործվող ֆունկցիաների որոշման եղանակ է, որի ժամանակ ֆունկցիայի որոնելի արժեքը տվյալ կետում որոշվում է նախորդ կետերում նրա ունեցած արժեքների միջոցով:

Ռեկուրսիվ սահմանման հզորությունը նրանում է, որ այն թույլ է տալիս վերջավոր արտահայտության միջոցով սահմանել օբյեկտների անվերջ բազմություն: Օրինակ, թվի ոչ բացասական ամբողջ ցուցիչով աստճանի ֆունկցիան կարելի է ներկայացնել հետևյալ կերպ.

$$x^n = \begin{cases} 1, & \text{if } n = 0 \\ x \cdot x^{n-1}, & \text{if } n > 0 \end{cases}$$

Եթ հայտնի է, որ ինչ-որ բան կարելի է նկարագրել ռեկուրսիվ եղանակով, ապա այն կարելի է լուծել ռեկուրսիայով, բայց դա ամենևին էլ չի նշանակում որ ճիշտ է այդպես վարվելը:

Ռեկուրսիան հանդիսանում է ցիկլի ընդհանրացում: Ստորև բերված օրինակը ցույց է տալիս ցիկլի պարզ փոխարինումը ռեկուրսիայով: Սակայն ավելի հաճախ հարկ է լինում կատարել հակառակ անցումը, քանի որ ռեկուրսիայի օգտագործումը պահանջում է լրացուցիչ հիշողություն և դանդաղեցնում է ծրագրերի աշխատանքը:

Ցիկլ.

```
def iter():
  global i, s1
  while i < 5:
      i += 1
      s1 += i


i, s1 = 0, 0
```

```
iter()
print('Ցիկլեր՝ ', s1)
```

Ռեկուրսիա.

```
def rec():
    global i, s2
    if i < 5:
        i += 1
        s2 += i
        rec()


i, s2 = 0, 0
rec()
print('Ռեկուրսիա՝ ', s2)
```

4. Ռեկուրսիվ ալգորիթմների օգտագործման վտանգները

Կարևոր է

Ռեկուրսիայի օգտագործման ժամանակ ամենատարածված սխալը անվերջ ռեկուրսիան է, երբ ֆունկցիաների կանչերի շղթան երբեք չի ավարտվում և շարունակվում է այնքան ժամանակ, մինչև վերջանա համակարգչի հիշողությունը:

Սահմանենք անվերջ ռեկուրսիայի երկու ամենատարածված պատճառները թվի ամբողջ ոչ բացասական աստիճանը հաշվող ոչ կոռեկտ գրված ֆունկցիայի օրինակով (որի ռեկուրսիվ սահմանումը մենք այսօր արդեն դիտարկել էինք):

```
def pow(n):
     return n * pow(n - 1)


def pow(n):
    if n == 0:
        return 1
    else:
        return n * pow(n)
```

Այսպիսով, ռեկուրսիվ ֆունկցիա գրելու ժամանակ ամենից առաջ պետք է ձևակերպել ռեկուրսիայի ավարտի պայմանները և մտածել այն մասին, թե ինչու ռեկուրսիան որևէ պահի կավարտի իր աշխատանքը:

Կարևոր է

Եվս մեկ խնդիր, որը կապված է ռեկուրսիվ ֆունկցիաների օգտագործման հետ. դա ալգորիթ
բարդության և արդյունավետության գնահատաման ոչ տրիվիալ լինելն է: Այդ ալգորիթմների

Чаты

բարդությունը կախված է ոչ միայն ներքին ցիկլերի բարդությունից, այլ նաև ռեկուրսիայի իտերացիաների քանակից: Ռեկուրսիվ ֆունկցիան կարող է բավականաչափ պարզ տեսք ունենալ, բայց այն կարող է էականորեն բարդացնել ծրագրի աշխատանքը՝ ինքն իրեն բազմակի կանչելու հետևանքով:

5. Արդյո՞ք գեղեցկությունը զոհեր է պահանջում

Գրենք ծրագիր, որը տասական հաշվարկման համակարգի թիվը կներկայացնի երկուական (իրականում նաև ցանկացած այլ հիմք ունեցող դիրքային) հաշվարկման համակարգում:

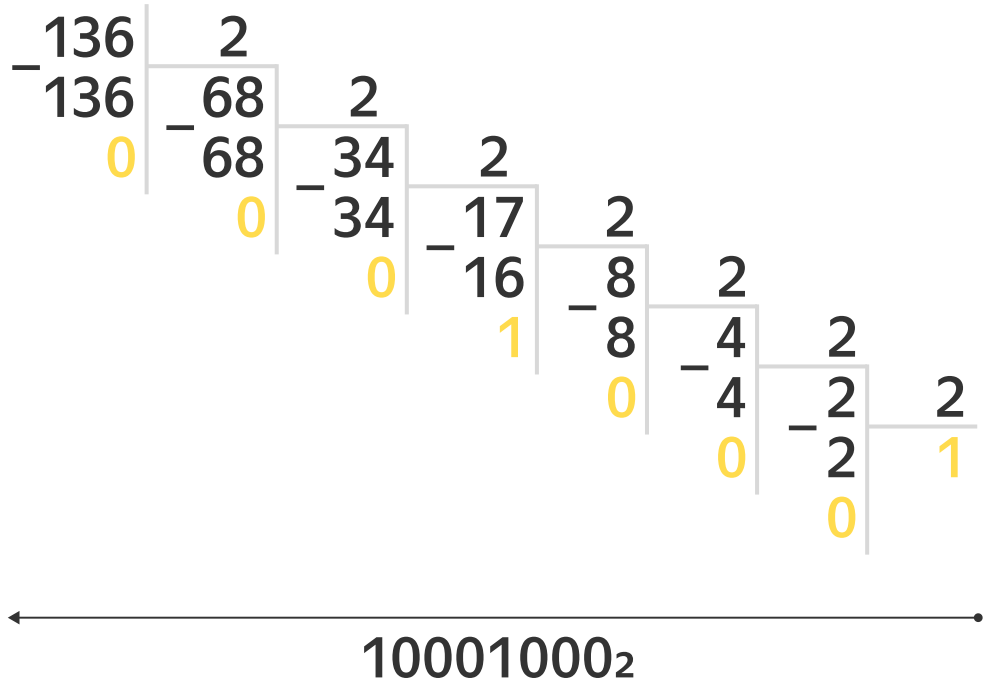
Սկզբի համար հիշենք անցման բազային ալգորիթմը.

Քայլ 1. Բաժանել թիվը այն հաշվարկման համակարգի հիմքի վրա, որին պետք է անցում կարատել:

Քայլ 2. Եթե բաժամնամ արդյունքը մեծ է կամ հավասար 2-ի, ապա շարունակում ենք այն բաժանել 2-ի այնքան ժամանակ, մինչև արդյունը լինի 1:

Քայլ 3. Մեկ տողով դուրս գրել վերջին բաժանման արդյունքը և բաժանումից ստացված մնացորդները՝ հակառակ կարգով:

Դիտարկենք 136 թիվը երկուական հաշվարկման համակարգով ներկայացնելու օրինակը:



$136_{10} = 10001000_2$

Ռեկուրսիայի օգնությամբ ալգորիթմը կարելի է ներկայացնել շատ կարճ և գեղեցիկ կոդի միջոցով:

```
def bin(a):
    if a > 1:
        bin(a // 2)
    print(a % 2)
```

Чаты

6. Մի քանի ռեկուրսիվ ճյուղեր: Ծառեր

Որպես օրինակ դիտարկենք Ֆիբոնաչիի թվերը հաշվող ֆունկցիան:

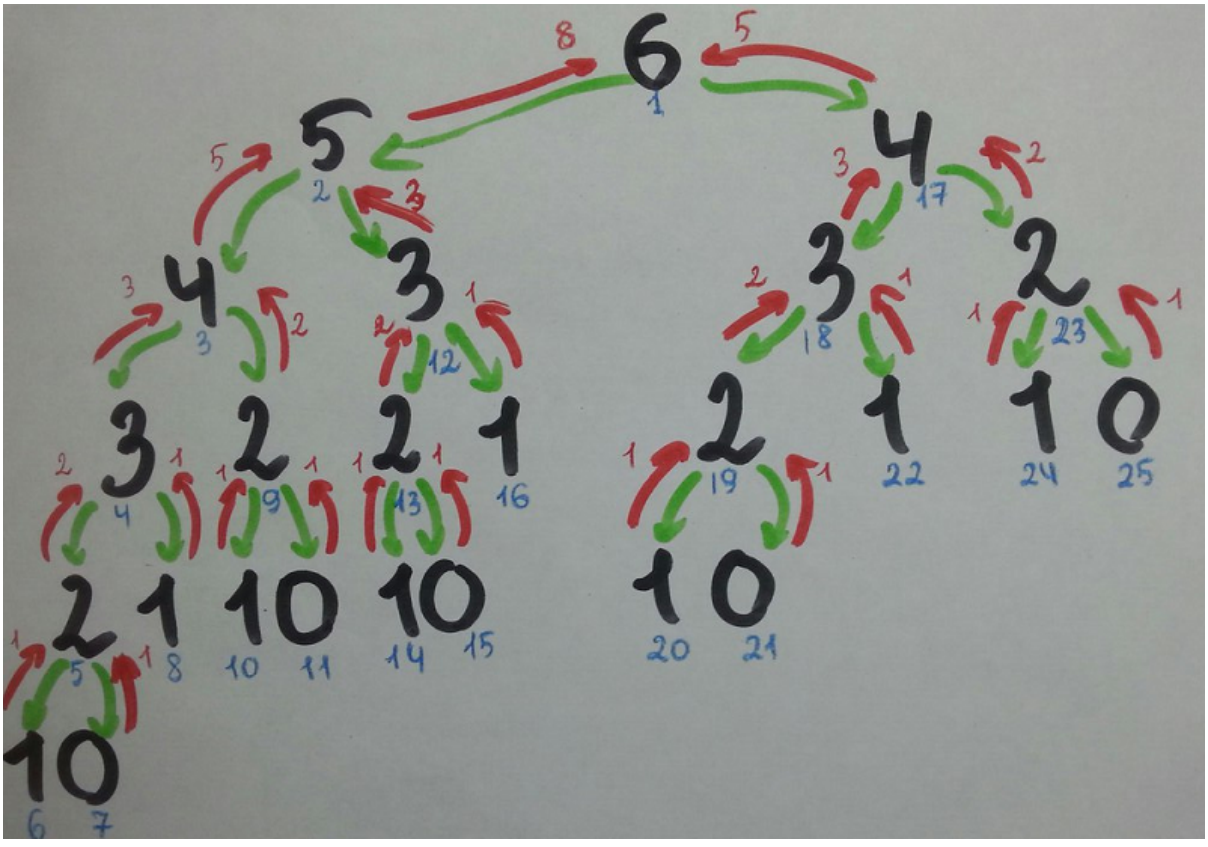
Ֆիբոնաչիի թվերը՝ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89... թվերի շարքն է, որտեղ առաջին երկու տարրերը հավասար են 1, իսկ յուրաքանչյուր հաջորդ տարր հավասար է իրեն նախորդող երկու տարրերի գումարին: Հատկանշական է, որ երկու հարևան Ֆիբոնաչիի թվերի հարաբերությունը ձգտում է ոսկե հատույթի թվին՝ 1,6180339887:

Կազմենք այդ թվերի ռեկուրսիվ սահմանումը և անմիջապես գրենք այն ֆունկցիայի տեսքով:

```
def fib(n):
    if n <= 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Զարմանալի է, բայց ծրագիրը բառ առ բառ համընկնում է Ֆիբոնաչիի թվերի սահմանման հետ:

Սակայն այս օրինակում մենք հանդիպեցինք ռեկուրսիայի նոր տեսակի հետ, որտեղ ֆունկցիան ստեղծում է երկու ռեկուրսիվ ճյուղեր: Ոչ բացահայտ տեսքով ծրագրի կատարման ժամանակ մենք իրականացնում ենք ծառի շրջանցում ըստ խորքի: Դիտարկենք այս ամենը հետևյալ օրինակի վրա՝ **fib(6)**



Կարևոր է հիշել, որ ֆունկցիայի ներկայացուցիչները կատարվում են ոչ թե զուգահեռ, այլ դետերմինացված (այսինքն որոշակի) հերթականությամբ՝ սկզբում ձախ ենթածառը, իսկ ապա ամբողջ աջ ենթածառը կամայական գագաթից: Կապույտ գույնով ցույց են տրված անցումները, կարմիրով՝ վերադարձվող արժեքները, իսկ կանաչ գույնով՝ ծառի կառուցվածքը:

Հետաքրքիր է այն փաստը, որ Ֆիբոնաչիի թվի մեծացման հետ զուգահեռ ծառը շատ արագ ա
բերում է ծրագրի աշխատանքի դանդաղեցմանը և հիշողության ծախսի:

Ի դեպ, յուրաքանչյուր հաջորդ ֆիբոնաչիի թիվ հաշվարկվում է ճիշտ ոսկե հատույթի թվով անգամ ավելի դանդաղ, քան նախորդը: Այսպիսով, **fib(500)** կհաշվարկվի միայն Արեգակնային համակարգի անհետացումից հետո:

Հաջորդ օրինակը ցուցադրում է այդ փաստը: Գործարկեք այն և համոզվոք.

```
from time import time


for i in range(20, 30):
    s = time()
    print(i, fib(i), "%.03f" % (time() - s))
```

Ինչպես տոսնում ենք արժեքների **քեշավորում** (նախորդ հաշվարկների արժեքների մտապահում) տեղի չի ունենում: Նմանատիպ օպտիմիզացիա առկա է որոշ ֆունկցիոնալ ծրագրավորման լեզուներում (LISP, Haskell): Դժբախտաբար, Python-ը նման օպտիմիզացիա ինքնուրույն չի կատարում, բայց արհեստական ձևով կարելի է իրականացնել այն:

Հետևյալ օրինակում մտապահվում են fib ֆունկցիայի վերջին 1000 կանչերը:

```
from functools import lru_cache


@lru_cache(maxsize=1000)
def fib(n):
    if n <= 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)


print(fib(100))
```

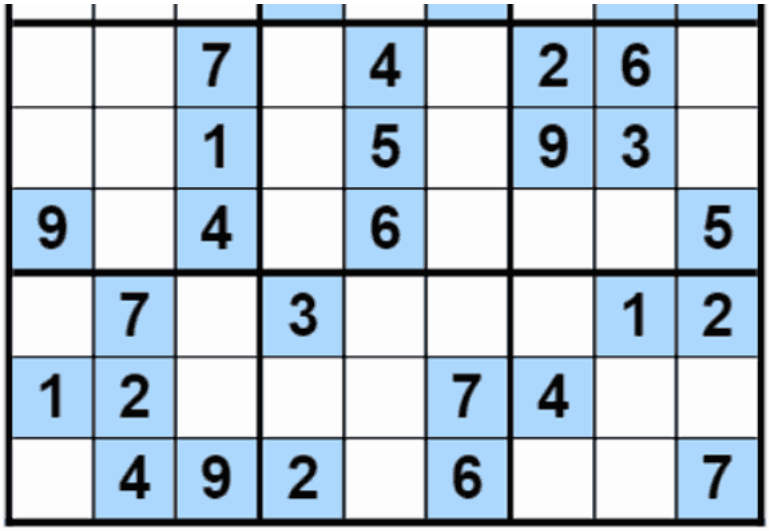
Գլխավոր եզրակացությունը հետևյալն է.

Կարևոր է

Ռեկուրսիվ մեթոդն ապահովում է ցուցակի, ծառի կամ գրաֆի շրջանցման համար հարմար եղանակ՝ ապահովելով տարրերով տեղաշարժը և նախորդ վիճակներին վերադարձը: Այս ամենը կարելի է օգտագործել բազմաթիվ մաթեմատիկական և կիրառական խնդիրներ լուծելու ժամանակ:

7. Բոնուս. լուծենք Սուդոկու

7	8		4			1	2	
6				7	5			9
			6		1		7	8



Հավանաբար նախկինում դուք արդեն հանդիպել եք այս խնդիր հետ: Հիշեք, թե ինչպես ենք այն լուծել: Այժմ փորձենք առաջարկել նոր լուծման եղանակ:

Ենթադրենք, պետք է գրել ծրագիր, որը լուծում է Սուդոկու: Դաշտի մոդելավորումը իրականացնենք ամբողջ թվերից կազմված ցուցակների ցուցակի միջոցով:

```
field = [
    [0,0,0,0,0,0,0,0,0],
    [0,1,0,0,2,0,0,3,0],
    [0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0],
    [0,4,0,0,5,0,0,6,0],
    [0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0],
    [0,7,0,0,8,0,0,9,0],
    [0,0,0,0,0,0,0,0,0],
]
```

Ձևակերպենք սուդոկուի լուծման ռեկուրսիվ ալգորիթմը:

- Եթե սուդոկուի դաշտում չկա դատարկ վանդակ, ապա համարենք, որ այն լուծված է և դաշտը վերադարձնենք որպես լուծում:
- Եթե կան դատարկ վանդակներ, ապա պետք է հաշվարկել որևէ դատարկ վանդակի արժեք, որի համար հնարավոր արժեքների քանակը մինիմալ է: Հերթականությամբ ստուգել այն բոլոր հնարավոր արժեքները և լուծումը գտնելու դեպքում վերադարձնել այն:

Ինքը՝ ֆունկցիան, այս նկարագրությունից երկար չէ:

```
from pprint import pprint
from copy import deepcopy
from random import shuffle
from time import clock

"""
Բոլոր վանդակների համար ըստ սահմանափակումների վերադարձնում է
հնարավոր արժեքների ցուցակը: օրինակ՝
(0, 0, {2, 3, 4, 5, 6, 7, 8, 9})
```

```
(0, 1, {2, 3, 5, 6, 8, 9})
(0, 2, {2, 3, 4, 5, 6, 7, 8, 9})
(0, 3, {1, 3, 4, 5, 6, 7, 8, 9})
(0, 4, {1, 3, 4, 6, 7, 9})
(0, 5, {1, 3, 4, 5, 6, 7, 8, 9})
(0, 6, {1, 2, 4, 5, 6, 7, 8, 9})
(0, 7, {1, 2, 4, 5, 7, 8})
(0, 8, {1, 2, 4, 5, 6, 7, 8, 9})
(1, 0, {4, 5, 6, 7, 8, 9})
"""


def get_variants(sudoku):
    variants = []
    for i, row in enumerate(sudoku):
        for j, value in enumerate(row):
            if not value:
                # արժեքները տողում
                row_values = set(row)
                # արժեքները սյունակում
                column_values = set([sudoku[k][j] for k in range(9)])
                # 3x3 չափերով ո՞ր քառակուսու մեջ է գտնվում վանդակը:
                # Այդ քառակուսու կոորդինատները
                sq_y = i // 3
                sq_x = j // 3
                square3x3_values = set([
                    sudoku[m][n]
                    for m in range(sq_y * 3, sq_y * 3 + 3)
                    for n in range(sq_x * 3, sq_x * 3 + 3)
                ])
                exists = row_values | column_values | square3x3_values
                # ո՞ր արժեքներն են մնացել
                values = set(range(1, 10)) - exists
                variants.append((i, j, values))

    return variants


def solve(sudoku):
    # Եթե սուդոկուն լրացված է, ապա սա է պատասխանը
    if all([k for row in sudoku for k in row]):
        return sudoku

    # Հակառակ դեպքում դիտարկենք բոլոր տարբերակները
    variants = get_variants(sudoku)

    # Ընտրեք այն վանդակը, որն ունի հնարավորինս քիչ հնարավոր արժեքներ
    x, y, values = min(variants, key=lambda x: len(x[2]))
```

```
    # Ըստ հերթականության ստուգենք բոլոր արժեքները
    for v in values:
        # deepcopy-ն ստեղծում է ցուցակի ամբողջական պատճենը՝ հաշվի
        # առնելով բոլոր ներդրված ցուցակները:
        new_sudoku = deepcopy(sudoku)
        new_sudoku[x][y] = v
        # Եթե լուծումը գտնվել է, վերադարձնենք պատասխանը:
        s = solve(new_sudoku)
        if s:
            return s


s = clock()
pprint(
    solve([
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 0, 0, 2, 0, 0, 3, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 4, 0, 0, 5, 0, 0, 6, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 7, 0, 0, 8, 0, 0, 9, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0],
    ]))
print('Ծախսված ժամանակը՝ ', clock() - s)
```

Ահա խնդրի ռեկուրսիվ լուծման ևս մեկ օրինակ: Այն ավելի կարճ է, բայց նրա մեջ առկա են **պիտոնական** հնարքներ:

```
from random import shuffle
from copy import deepcopy
from pprint import pprint


def make_assumptions(sudoku):
    for i, row in enumerate(sudoku):
        for j, value in enumerate(row):
            if not value:
                values = set(row) \
                         | set([sudoku[k][j] for k in range(9)]) \
                         | set([sudoku[m][n]
                                for m in range((i // 3) * 3,
                                               (i // 3) * 3 + 3)
                                for n in range((j // 3) * 3,
                                               (j // 3) * 3 + 3)])
                yield i, j, list(set(range(1, 10)) - values)
```

```
def solve(sudoku):
    if all([k for row in sudoku for k in row]):
        return sudoku
    assumptions = list(make_assumptions(sudoku))
    shuffle(assumptions)

    x, y, values = min(assumptions, key=lambda x: len(x[2]))

    for v in values:
        new_sudoku = deepcopy(sudoku)
        new_sudoku[x][y] = v
        s = solve(new_sudoku)
        if s:
            return s


pprint(solve(field))
```

Նախորդ երկու օրինակները ցույց են տալիս ռեկուրսիայի առավելությունները՝ կարճ և հեշտ ընթերցվող ծրագրերի կազմումը:

Փորձեք համեմատել այս ծրագրերը իմպերատիվ (առանց ռեկուրսիայի օգտագործման) տարբերակի հետ:

Ռեկուրսիային դուք դեռ շատ կհանդիպեք: Միայն հիշեք, որ այն **համադարման չէ,** բայց հնարավորություն է տալիս գեղեցիկ և էֆեկտիվ լուծել խնդիրների մի լայն շրջանակ:

Առայժմ այսքանը: Անցնենք խնդիրների լուծմանը: