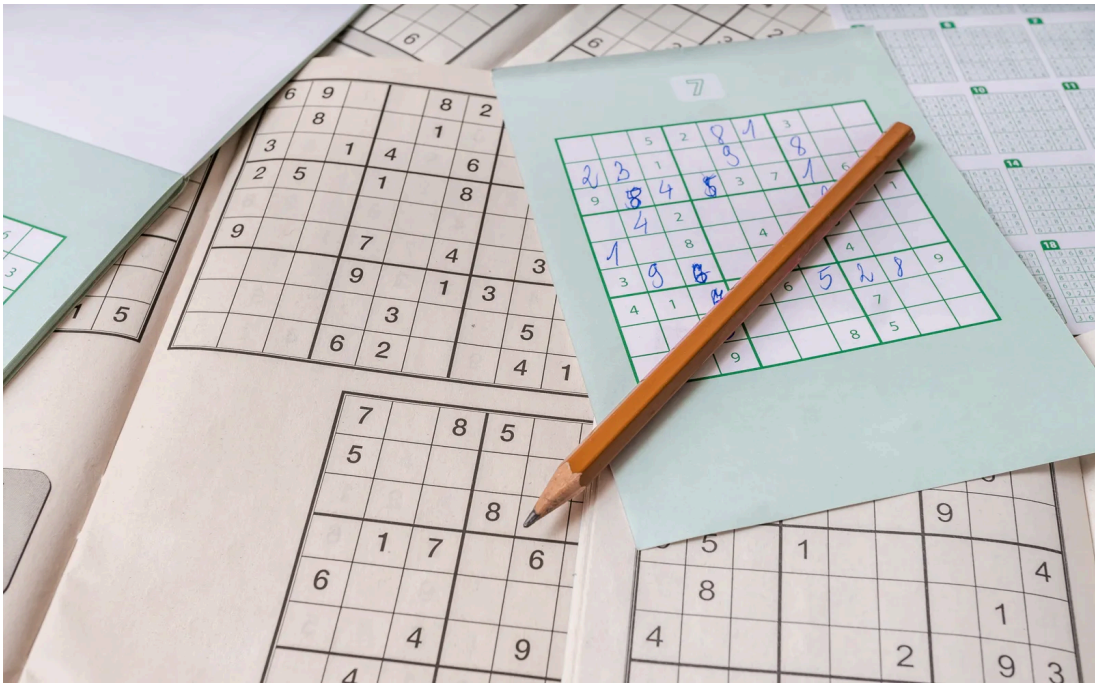
quera.org/course/14904/
مبانی برنامه نویسی پایتون
assignments/64107

پروژه حل سودوکو با استفاده از زبان python



دانشجو : پارسا سلیمانی/رشته آمار/۶۱۰۳۰۲۱۵۴
استاد : دکتر گودرزی

۱ بهمن ۱۴۰۲

فهرست

مقدمه:

این پروژه به عنوان تمرین کلاس مبانی برنامه نویسی انجام شده است.
در حل این مسئله از زبان پایتون کمک گرفته شده و الگوریتم این سوال بر پایه ایده backtrack کردن شکل گرفته که از توابع بازگشتی بهره میبرد

الگوریتم:

بک ترک (backtrack) به عنوان الگوریتمی بهینه جهت حل سوالاتی شناخته میشود که نیازمند بررسی یک فضای نمونه نسبتا بزرگ زمانی که الگوریتم های brute force و یا گونه های دیگر الگوریتم هایی که بر ایده complete search شکل گرفته اند سرعت پایینی دارند.
زمانی بک ترک به عنوان گزینه ای ایدئال در نظر گرفته میشود که درخت (tree) حالات مختلف فضای نمونه قابلیت محدود سازی داشته باشد (مثلا در پروژه سودوکو قوانین این بازی این نقش را ایفا میکنند)
در این کد در هر مرحله تابع **solver** فضا های خالی را با استفاده از **free_places** و **possibles** شناسایی و پر میکند و در هر مرحله در صورت وجود خطایی (خانه ای بدون هیچ عدد مناسب برای پر کردن) با متوقف کردن محاسبات مربوط به شاخه
از هدر رفت منابع جلوگیری میکند
و در نهایت **print_board** لیست دو در دو را به صورت مورد نظر به ترمینال نمایش میدهد

پیاده سازی:

تابع **get_input** وظیفه گرفتن ورودی از کاربر و خروجی دادن یک لیست دو در دو را دارد و تابع **print_board** با گرفتن یک لیست دو در دو خروجی را به صورت مشخص شده در صورت سوال نمایش میدهد

```python
blank = 0
def get_input():
    """
    should return a 9*9 list:
    """
    l=[]
    for i in range(9):
        l.append(list(map(int,input().split())))
    return l

def print_board(l):
    for i in range(9):
        for j in range(9):
            print(l[i][j],end = " ")
        print()
```

تابع **free_place** جهت نمایش اولین خانه خالی جدول به کار میرود
تابع **possibles** با بررسی کردن شرایط جدول لیستی از همه اعدادی که در آن خانه خالی میتوانند باشند باز میگرداند

```python
```

```
def free_place(l):
    f = []
    for i in range(9):
        for j in range(9):
            if l[i][j] == blank:
                return [i,j]


def possibles(l,place):
    possible=[i for i in range(1,10)]
    for i in range(9):
        if l[place[0]][i] in possible:
            possible.remove(l[place[0]][i])
    for i in range(9):
        if l[i][place[1]] in possible:
            possible.remove(l[i][place[1]])
    x= place[0]//3
    y= place[1]//3
    for i in range(x*3,x*3+3):
        for j in range(y*3,y*3+3):
            if [i,j]!=place:
                if l[i][j] in possible:
                    possible.remove(l[i][j])
    return possible
```

```

تابع **solver** به عنوان بخش اصلی الگوریتم با بهره گیری از توابع نامبرده با محاسبه درخت جایگشت های اعداد جدول در صورت وجود یک جدول کامل به عنوان جواب آن را خروجی میدهد

```python
def solver(l):
    free = free_place(l)
    if not free:
```
```

```
        return l
    poss = possibles(l,free)
    if len(poss)==0:
        return None
    for val in poss:
        l[free[0]][free[1]] = val
        meow = solver(l)
        if meow:
            return meow
        l[free[0]][free[1]] = 0

print_board(solver(get_input()))
```

```
```

چالش ها :

در الگوریتم این کد یک بازنگری اساسی در بخش **solver** صورت گرفت
در ابتدا بنا بر این بود که یک لوپ جهت بررسی فضا های خالی در این تابع استفاده گردد اما به علت time limit خوردن کد بازنگری به اینصورت انجام شد که تابع **free_places** به جای یک لیست از همه فضای های خالی اولین فضای خالی را به **solver** برگرداند و به جای لوپ کردن در آن فضا های خالی متفاوت نیز به طور بازگشتی پر شوند

نتیجه گیری

این پروژه برای شخص نویسنده ارزش آموزشی بالایی داشت و از آنجا که احتمالا آخرین پروژه این کلاس خواهد بود میخواهم از صمیم قلب از آقای پرورش و دکتر گودرزی به خاطر این کلاس تشکر کنم

ممنون از وقت شما