حل سودوکو به روش بازگشت به عقب

گرداورندگان: ایمان شیروی، سجاد سالمی، نگار فرهنگ

استاد: دکتر زارع‌پور

ساختمان داده:

ما این مساله را به صورت بازگشتی وبا استفاده از لیست دو بعدی حل کردیم. یعنی در یک جدول دو بعدی مقادیر مشخص را عددی (بین ۱_۹) قرار دادیم و مقادیر نامشخص را (0) قرار دادیم و با استفاده از الگوریتم بازگشتی این مساله را حل کردیم.

الگوریتم:

از الگوریتم بازگشتی استفاده شده است. الگوریتم بازگشتی استفاده شده به گونه ای است که به ترتیب در خانه های خالی عددی بین ۱ تا ۹ قرار میدهد و برای هر کدام از اعداد بررسی میکند که آیا این عدد قوانین سودوکو را رعایت میکند یا نه اگر حداقل یکی از اعداد، قوانین را رعایت نکرد الگوریتم متوقف می‌شود و جدول غیر قابل حل کردن است.

کد:

تابع get_input : مقادیر را مانند مثال به صورت سطری دریافت میکند. کاربر باید از (.) برای نمایش خانه های خالی و از (1_9) برای خانه های پر استفاده کند در اخر تابع مقادیر (.) را به 0 تبدیل میکند و درون یک لیست دو بعدی قرار میدهد.

```
def get_input():
    """Get the Puzzle in one line format.
    Example's:
    - ....49..2
    - .5.4....."""
    board = []
    print("Enter puzzle in one line format:\n")
    for i in range(9):
        i = input(">")
        tmp = []
        for j in i:
            if j ==".":
                tmp.append(0)
            else:
                tmp.append(int(j))
        board.append(tmp)
    return board
```

تابع print_bord : لیست دوبعدی که مقادیر برد در آن قراردارد را دریافت میکند و آن را به یک فرم زیبا چاپ میکند.

```
def print_board(board):
    """Prints the given `board`"""
    print("\t\tSOLVED PUZZLE:")
    print("+","-"*5,"+","-"*5,"+","-"*5,"+")
    for i in range(9):
        print("|", end=" ")
        for j in range(9):
            print(board[i][j] , end=" ")
            if (j+1)%3==0 and j>0 and j <7:
                print("|", end=" ")
        print("|")
        if (i+1)%3==0 and i>0:
            print("+","-"*5,"+","-"*5,"+","-"*5,"+")
```

تابع is_right : بررسی میکند که ایا عدد داده شده ی (x) در سطر یا ستون لیست (bord) وجود دارد یا نه.

همچنین بررسی میکند که در هر سطر و ستون (3*3) از هر عدد (1_9) فقط یکی امده باشد و قوانین سودوکو رعایت شده باشد و اگر قوانین رعایت نشده باشد مقدار (false) برمیگرداند.

```
def is_right(board, row, column, x):
    """check if x is valid for the cell in the given `row` and `column`"""
    # check row and column
    if x in board[row]:
        return False

    for i in range(9):
        if board[i][column] == x:
            return False

    # check the smaller square
    row_s = 3 * (row // 3)
    column_s = 3 * (column // 3)
    for i in range(row_s, row_s + 3):
        for j in range(column_s, column_s + 3):
            if board[i][j] == x:
                return False
    return True
```

تابع empty : به دنبال خانه‌های خالی در لیست میگردد و اگر خانه ای خالی پیدا کرد شاخص های سطر و ستون آن خانه را برمیگرداند و اگر خانه ای خالی نبود '-','-' را برمیگرداند.

```
def empty(board):
    """it will look for an empty cell and in case couldn't find it, it will return '-','-' """
    for i in range(9):
        for j in range(9):
            if board[i][j] == 0:
                return i,j
    return "-","-"
```

تابع solver : تابع اصلی است که مساله را به صورت بازگشتی حل میکند یعنی در هر خانه خالی (x) را به ترتیب از (1_9) قرار میدهد و قوانین سودوکو را بررسی میکند اگر درست بود مقدار x را در خانه مورد نظر در لیست bord قرار میدهد و اگر هیچ عددی درون ان خانه به صورت درست قرار نگرفت مقدار false را برمیگرداند که نشان دهنده عدم موفقیت در حل سودوکو است.

```
def solver(board):
    """solver is used to solve the puzzle"""
    row,column = empty(board)
    if row == '-':
        return True
    for x in range(1,10):
        if is_right(board, row, column, x):
            board[row][column] = x
            if solver(board) :
                return board
            else:
                board[row][column] = 0
    return False
```

تابع runner : تابعی است که ابتدا ورودی را با استفاده از تابع get_input دریافت کرده و سپس با فراخوانی تابع حل کننده سودوکو تلاش میکند پازل را حل کند. اگر مقدار برگردانده شده هر چیزی بجز false بود آنگاه متوجه می‌شود که سودوکو حل شده است جواب پایانی یا همان سودوکو حل شده را با استفاده از تابع print_board چاپ میکند و در صورتی که نتوانست سودوکو را حل کند، در ترمینال این موضوع را اطلاع داده و برنامه به پایان می‌رسد.

```python
def runner():
    board = get_input()
    result = solver(board)
    if result :
        print("HUH i solve it. i guess im brilliant :/")
        print_board(board)
    else:
        print("cant solve it.")
```

پیچیدگی زمان:

برای محاسبه پیچیدگی زمان بدترین حالت ممکن را در نظر میگیریم که در اینجا می‌شود 9 (اعداد 0 تا 9) بار صدا زدن تابع به صورت بازگشتی. در این صورت برای درجه های مختلف سختی سودوکو این زمان متفاوت است ولی همگی از فرمول O(9^m) پیروی می‌کنند که m تعداد خانه‌های خالی در جدول می‌باشد.

- خیلی ساده : O(9^10)
- ساده : O(9^15)
- متوسط : O(9^27)
- سخت : O(9^42)
- خیلی سخت : O(9^60)

* لازم به ذکر است که در هر حالت میانگین تعداد خانه خالی محاسبه شده است.

باید اشاره کرد که در عمل، زمان حل بسیار میتواند سریعتر باشد با توجه به موقعیت های خانه های پر در جدول و بدترین حالت های در نظر گرفته شده به ندرت دیده می‌شوند. پیچیدگی زمان این الگوریتم با افزایش تعداد خانه‌های خالی به سرعت افزایش پیدا میکند که دلیل آن زیاد شدن تهداد احتمال‌های اعداد برای خانه‌های جدید است.

پیچیدگی فضا:

در این الگوریتم دو فضا مطرح می‌شود:

- فضای ثابت برای ذخیره ورودی: از آنجا که این مقدار وابستگی به داده گرفته شده از کاربر ندارد و ثابت است، پیچیدگی زمانی O(1) را به خودش اختصاص می‌دهد.

- فضای استفاده شده برای فراخوانی تابع به صورت بازگشتی: تابع به صورت بازگشتی فراخوانی میشود و این مقدار ارتباط مستقیمی با تعداد خانه‌های خالی دارد چرا که بیشترین میزان حافظه تخصیص داده شده زمانی است که تابع بازگشتی به اندازه تعداد خانه‌های خالی فراخوانی شوند پس پیچیدگی زمانی این بخش $O(m)$ است که m تعداد خانه‌های خالی است.

که در کل پیچیدگی فضای این الگوریتم برابر است با $O(m)$.