درین رستمی – ۹۷۲۴۳۰۳۴

Initial state: در حالت اولیه مار در یک خانه قرار گرفته و الگوریتم موردنظر ران میشود تا جهت حرکت معلوم شود.

Action: سپس در آن جهت مار حرکت میکند و دوباره الگوریتم از این مکان جدید ران میشود. اگر به خانه‌ای برسد که در آن سیب باشد، ار تعداد سیب‌هایی که بهشون نرسیدیم کم میشود اما سیب از نقشه حذف نمیشود چراکه وقتی حرکت بعدی را بیابیم تازه میفهمیم در کدام جهت باید به بدن مار اضافه کنیم. پس برای حرکت کردن اگر مار در خانه‌ای قرار داشته باشد که در آن سیب باشد، اول سیب را میخورد (سیب از روی نقشه حذف میشود) و سپس در جهت بدست آمده یک واحد به مار اضافه میشود. اگر سیب نباشد، سر مار به آن جهت میرود و همه قسمت‌های بدن مار یک واحد جابجا میشوند (اگر ۰ را سر مار در نظر بگیریم، برای همه قسمت‌های بدن مار قسمت nام میرود جای قسمت n-1ام)

- با هر حرکت مار الگوریتم اجرا میشود زیرا ممکن است اول با اجرای الگوریتم با توجه به اینکه بدنش سر راهش بوده یک سیب انتخاب شود ولی در حرکت بعدی چون بدنش دیگر سر راه نیست سیب دیگری نزدیکتر باشد.

Goal state: وقتی تعداد سیب‌هایی که به آنها نرسیدیم برابر ۰ شود.

فقط BFS و IDS پیاده‌سازی شده که وقتی هر بار عمق IDS را یکی اضافه کنیم (الگوریتم فقط برای IDS با step=1 پیاده سازی شده است) در حقیقت جواب BFS را میابد با این تفاوت که اگر جواب در شاخه‌های چپ تر درخت باشد از حافظه کمتری استفاده میشود. ولی در IDS هربار درخت از اول تولید میشود و وریش DFS اجرا میشود، پس زمانش بیشتر است. اگر step را در IDS ۲ یا بیشتر بگیریم ممکن است جوابی دورتر را اول پیدا کند (برای step=2 ممکن است جواب پیدا شده ۱ واحد عمیق‌تر از جواب بهینه باشد، برای step=3 ۲ واحد عمیق‌تر و...) چون با DFS جلو میرود یعنی اول تا آخر عمق یک شاخه را میبیند و بعد به شاخه بعد میرود. به همین دلیل که نزدیکترین سیب را میخواهیم فقط step=1 پیاده سازی شده است.

برای هردو الگوریتم متغیرها و توابع زیر یکسان است:

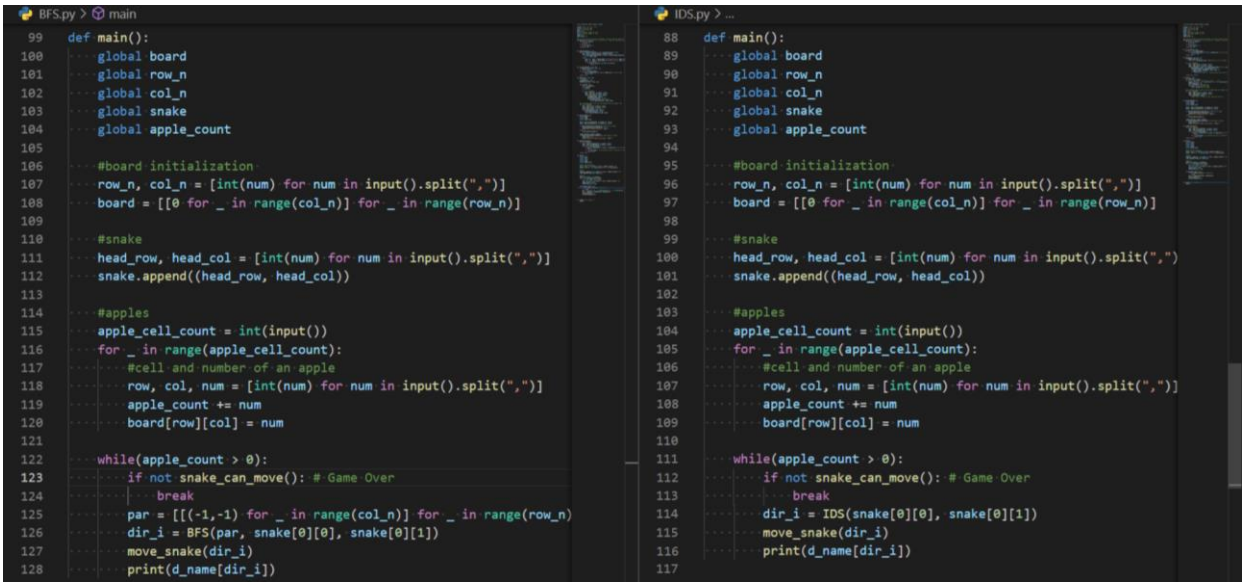
d_row و d_col جابه‌جایی‌هایی هستند که با آنها در یک آرایه ۲ بعدی یعنی board میتوان به ۴ جهت حرکت کرد. d_name هم فقط اسم جابه‌جایی‌های مذکور است. row_n و col_n تعداد سطر و ستون‌هاست که در ورودی داده میشوند. snake لیستیست که جایگاه قسمت‌های مختلف بدن مار را نگه میدارد به طوری که المنت ۰ سر مار و المنت len(snake)-1 دم مار است. apple_count هم تعداد سیب‌هایی است که به آنها نرسیدیم.

با توجه به اینکه صفحه دیوار ندارد با استفاده از تابع get_co بدست میاوریم که اگر یک واحد در سطر یا ستون جابجا شویم به کدام سطر یا ستون میرویم. (نحوه استفاده از آن در توابع بعدی قابل مشاهده است)

```
d_name = ['U', 'R', 'D', 'L']
d_row = [-1, 0, 1, 0]
d_col = [0, 1, 0, -1]
board = []
row_n = 0 #total number of rows
col_n = 0 #total number of cols
snake = []
apple_count = 0

#get row/col based on current coordinates, direction and max number of rows/cols on the board
def get_co(co, d, max_co): # d = -1 or 0 or 1 , max_cow = row_n or col_n
    if co + d == max_co:
        return 0
    if co + d == -1:
        return max_co - 1
    return co + d
```

تابع main در هردو الگوریتم ورودی‌ها را میگیرد و الگوریتم را صدا میکند.



```python
def main():
    global board
    global row_n
    global col_n
    global snake
    global apple_count

    #board initialization
    row_n, col_n = [int(num) for num in input().split(",")]
    board = [[0 for _ in range(col_n)] for _ in range(row_n)]

    #snake
    head_row, head_col = [int(num) for num in input().split(",")]
    snake.append((head_row, head_col))

    #apples
    apple_cell_count = int(input())
    for _ in range(apple_cell_count):
        #cell and number of an apple
        row, col, num = [int(num) for num in input().split(",")]
        apple_count += num
        board[row][col] = num

    while(apple_count > 0):
        if not snake_can_move(): # Game Over
            break
        par = [[(-1,-1) for _ in range(col_n)] for _ in range(row_n)
        dir_i = BFS(par, snake[0][0], snake[0][1])
        move_snake(dir_i)
        print(d_name[dir_i])
```

```python
def main():
    global board
    global row_n
    global col_n
    global snake
    global apple_count

    #board initialization
    row_n, col_n = [int(num) for num in input().split(",")]
    board = [[0 for _ in range(col_n)] for _ in range(row_n)]

    #snake
    head_row, head_col = [int(num) for num in input().split(",")
    snake.append((head_row, head_col))

    #apples
    apple_cell_count = int(input())
    for _ in range(apple_cell_count):
        #cell and number of an apple
        row, col, num = [int(num) for num in input().split(",")]
        apple_count += num
        board[row][col] = num

    while(apple_count > 0):
        if not snake_can_move(): # Game Over
            break
        dir_i = IDS(snake[0][0], snake[0][1])
        move_snake(dir_i)
        print(d_name[dir_i])
```

فقط قبل صدا کردن الگوریتم چک میکند که اگر مار هر ۴ جهتش بدن خودش باشد بازی تمام میشود.

```python
def snake_can_move():
    for i in range(4):
        adjx = get_co(snake[0][0], d_row[i], row_n)
        adjy = get_co(snake[0][1], d_col[i], col_n)
        for i in range(len(snake)):
            if adjx == snake[i][0] and adjy == snake[i][1]: #is the snake
                break
            if i == len(snake) - 1: #reached the end of snake but it wasn't equal to current adj
                return True
    return False
```

تابع move_snake هم همان action مار است که صفحه اول گزارش توضیح داده شد. پرینت‌ها هم برای نشان دادن state مار است که دارد سیب را میخورد و از روی آن کنار میرود یا تازه به سیب رسیده است (بعد این جملات یکی از حروف نشان‌دهنده جهت چاپ خواهند شد). اگر هیچکدام از این جملات چاپ نشوند یعنی سر مار به هیچ سیبی نرسیده و مار فقط روی صفحه حرکت کرده است)

```python
def move_snake(dir_i):
    global board
    global snake
    global apple_count

    destx = get_co(snake[0][0], d_row[dir_i], row_n)
    desty = get_co(snake[0][1], d_col[dir_i], col_n)

    if board[snake[0][0]][snake[0][1]] != 0: #was on apple
        board[snake[0][0]][snake[0][1]] -= 1
        print("ate an apple and went ", end="")
    else:
        snake.pop(len(snake)-1)

    snake.insert(0, (destx, desty))
    if board[snake[0][0]][snake[0][1]] != 0: #going on apple
        apple_count -= 1
        print("reached an apple when going ", end="")
```

الگوریتم BFS :

موقع صدا زده شدن سر مار به آن پاس داده میشود و از آنجا شروع میکند الگوریتم BFS را اجرا کردن. چون نیاز داریم بدانیم از کدام جهت به سیب رسیده (اگر رسیده باشد) به جای آرایه visited که معمولا در این الگوریتم استفاده میشه، از parent استفاده شده که والد هر خانه را نگه داریم و هر موقع به سیب رسیدیم با تابع get_dir آرایه را از سیب شروع کنیم به طی کردن تا به سر مار برسیم (خط ۵۸).

اگر کل board طی شد ولی سیبی یافت نشد به این معنی است که در حال حاضر بدن مار روی سیب‌های باقی مانده است پس به صورت رندم به یک جهتی حرکت میکند تا از روی سیب کنار برود.

```
33  def is_valid(parent, row, col):
34      if parent[row][col] != (-1, -1):
35          return False
36      for i in range(len(snake)):
37          if row == snake[i][0] and col == snake[i][1]: #is the snake
38              return False
39      return True
40
41  def BFS(parent, row, col):
42      q = queue()
43      q.append((row, col))
44      parent[row][col] = (row, col)
45
46      while(len(q) > 0):
47          cell = q.popleft()
48          x = cell[0]
49          y = cell[1]
50
51          for i in range(4):
52              adjx = get_co(x, d_row[i], row_n)
53              adjy = get_co(y, d_col[i], col_n)
54              if(is_valid(parent, adjx, adjy)):
55                  q.append((adjx, adjy))
56                  parent[adjx][adjy] = (x, y)
57                  if board[adjx][adjy] != 0: #apple
58                      return get_dir(parent, adjx, adjy)
59
60      #no reachable apple right now even though there are apples left
61      for i in range(4):
62          adjx = get_co(row, d_row[i], row_n)
63          adjy = get_co(col, d_col[i], col_n)
64          parent[adjx][adjy] = (-1,-1)
65          if(is_valid(parent, adjx, adjy)):
66              parent[adjx][adjy] = (x, y)
67              return get_dir(parent, adjx, adjy)
```

تابع get_dir : چون فقط یک حرکت بعدی مار را میخواهیم، هر موقع به جایی رسید که سر مار و خانه بعدی در حرکت را داشت، اندیس متناظر حرکت با توجه به آرایه‌های d_row و d_col را بر میگرداند.

```
def get_dir(parent, x, y):
    while(not(parent[x][y][0] == x and parent[x][y][1] == y)):
        adjx, adjy = parent[x][y][0], parent[x][y][1]
        if(not(parent[adjx][adjy][0] == adjx and parent[adjx][adjy][1] == adjy)):
            x, y = adjx, adjy
        else:
            drow = x - adjx if abs(x-adjx) <= 1 else -1 if x > adjx else 1
            dcol = y - adjy if abs(y-adjy) <= 1 else -1 if y > adjy else 1
            for i in range(4):
                if drow == d_row[i] and dcol == d_col[i]:
                    return i
```

الگوریتم IDS :

در main تابع IDS با مختصات سر مار صدا زده میشود و این تابع عمق درخت را از ۱ تا m/2 در نظر میگیرد (چون وقتی صفحه دیوار ندارد حداکثر فاصله بین دو خانه برابر نصف ماکسیمم طول و عرض صفحه است) و DFS با عمق محدود را صدا میزند. اگر از DFS جواب گرفت (سیب پیدا شده بود)، جهت را بازمیگرداند وگرنه مار روی سیب است و یک جهت رندم برای حرکت مار برمیگرداند.

```
def IDS(row, col):
    m = max(col_n, row_n)
    for i in range(1,m//2 + 1): # i = 1, 1+1, ..., m/2
        depth = [[-1 for _ in range(col_n)] for _ in range(row_n)]
        depth[row][col] = 0
        ans = DFS(depth, row, col, i)
        if ans[1] == True: #apple found
            return ans[0] #dir to apple

    #no reachable apple right now even though there are apples left
    for i in range(4):
        adjx = get_co(row, d_row[i], row_n)
        adjy = get_co(col, d_col[i], col_n)
        if(is_valid(adjx, adjy)):
            return i #i = random dir that snake can move to
```

تابع DFS به صورت بازگشتی پیاده سازی شده است پس parent نیاز نیست و فقط نیاز است عمق خانه‌ها را داشته باشیم تا هرموقع به حد عمق رسیدیم ادامه ندهیم. چون IDS عمق را یکی یکی افزایش میدهد پس اگر تا l-1 سیب پیدا نکرده باشیم فقط ممکن است در l سیب باشد پس فقط در همین عمق دنبال سیب میگردیم.

تابع is_valid برای این الگوریتم با BFS فرق دارد چراکه در این الگوریتم ما از چند را به یک خانه میتوانیم برسیم و ممکن است اول از راه طولانی‌تر به آن رسیده باشیم. پس فقط اینکه خانه را قبلا مشاهده نکرده باشیم مدنظر نیست (depth[adj]==-1) بلکه اگر الان با عمق بهتری به آن خانه میرسیم هم باید باز الگوریتم را روی آن خانه اجرا کنیم.

```
def is_valid(row, col):
    for i in range(len(snake)):
        if row == snake[i][0] and col == snake[i][1]: #is the snake
            return False
    return True

def DFS(depth, row, col, l):
    if depth[row][col] == l:
        return (-1, board[row][col] != 0) #true if it's an apple

    for i in range(4):
        adjx = get_co(row, d_row[i], row_n)
        adjy = get_co(col, d_col[i], col_n)
        if(is_valid(adjx, adjy) and (depth[adjx][adjy] == -1 or depth[adjx][adjy] > depth[row][col] + 1)):
            depth[adjx][adjy] = depth[row][col] + 1
            _, apple_found = DFS(depth, adjx, adjy, l)
            if apple_found == True:
                return (i, True)  #i = the direction which led to the adj cell where an apple was found

    return (-1, False)
```

تست اول با هردو الگوریتم:

```
PS N:\uni\AI\CA1> python .\BFS.py
5,5
0,0
4
3,1,1
3,2,1
1,4,2
4,3,1
D
reached an apple when going L
ate an apple and went U
U
reached an apple when going L
ate an apple and went U
reached an apple when going L
ate an apple and went reached an apple when going L
ate an apple and went U
U
L
reached an apple when going L
```

```
PS N:\uni\AI\CA1> python .\IDS.py
5,5
0,0
4
3,1,1
3,2,1
1,4,2
4,3,1
D
reached an apple when going L
ate an apple and went U
U
reached an apple when going L
ate an apple and went U
reached an apple when going L
ate an apple and went reached an apple when going L
ate an apple and went U
U
L
reached an apple when going L
```

تست دوم:

```
PS N:\uni\AI\CA1> python .\BFS.py
10,12
0,0
6
0,1,1
9,11,1
0,11,1
9,0,1
5,6,1
6,10,1
reached an apple when going U
ate an apple and went reached an apple when going L
ate an apple and went reached an apple when going D
ate an apple and went R
reached an apple when going R
ate an apple and went U
U
U
U
L
L
reached an apple when going L
ate an apple and went U
L
L
L
reached an apple when going L
```

```
PS N:\uni\AI\CA1> python .\IDS.py
10,12
0,0
6
0,1,1
9,11,1
0,11,1
9,0,1
5,6,1
6,10,1
reached an apple when going U
ate an apple and went reached an apple when going L
ate an apple and went reached an apple when going D
ate an apple and went R
reached an apple when going R
ate an apple and went U
U
U
U
L
L
reached an apple when going L
ate an apple and went U
L
L
L
reached an apple when going L
```

تست سوم:

```
PS N:\uni\AI\CA1> python .\BFS.py
10,10
0,0
5
3,4,2
2,2,1
9,1,1
5,7,2
4,9,1
U
reached an apple when going R
ate an apple and went R
D
D
reached an apple when going D
ate an apple and went R
R
reached an apple when going D
ate an apple and went R
R
R
D
reached an apple when going D
ate an apple and went R
U
reached an apple when going R
ate an apple and went U
R
R
R
R
reached an apple when going R
ate an apple and went R
R
R
D
reached an apple when going D
```

```
PS N:\uni\AI\CA1> python .\IDS.py
10,10
0,0
5
3,4,2
2,2,1
9,1,1
5,7,2
4,9,1
U
reached an apple when going R
ate an apple and went R
D
D
reached an apple when going D
ate an apple and went R
R
reached an apple when going D
ate an apple and went R
R
R
D
reached an apple when going D
ate an apple and went R
U
reached an apple when going R
ate an apple and went U
R
R
R
R
reached an apple when going R
ate an apple and went R
R
R
D
reached an apple when going D
```