**프로그래머스 Lv.3 + BOJ 알고리즘별 문제**

#[1차]추석 트래픽

이 문제의 경우 해결하는데 오랜 시간이 걸렸다. 처음에는 시간:분:초 이런 형태로 정리 되어 있는 시간에서 주어진 시간(단위:ms)만큼 이전으로 가야 한다는 점에서 너무 복잡했으며, 60으로 나누어 몫과 나머지를 계속 갱신해야 했기 때문에 생각하기가 너무 까다로웠다. 그러나 문제의 출력 조건이 시간이 아니라 해당되는 범위의 개수이므로 그냥 전부 초 단위로 바꾸어 주었다. 그 이후에 일일이 리스트에 값을 모두 넣어야 하나 싶었지만 그것이 아니라 어차피 시행되는 개수는 어떤 로그의 시작지점과 끝나는 지점에서 바뀌기 때문에 로그의 시작 지점~1초후 라는 구간에 대해 주어진 다른 범위의 끝 < 시작지점 이거나 범위의 시작 >= 1초후가 아닐 경우에만 해당 되도록 한다. 그래서 최대 값을 구하게 되었는데

일단 시간 관련문제에 있어서 단위가 헷갈리고 환산이 복잡하다면 한 단위로 통일하는 것을 고려해 보고

구간별로 나누어서 최댓값/최솟값을 구하는 문제가 나온다면 변화하는 지점에 주목하자. 그리고 그것을 기준으로 전부 탐색해보자. 미적분 문제를 풀 때 변곡점/미분값이 0이 되는 지점 등에 주목하는 것과 비슷한 논리이다. 무작정 소수점 세번째 자리까지 범위를 쪼개어 구하라고 할 리가 없다.

def **change**(data):

    time = list(**map**(float, data[1].split(':')))

    spent = float(data[2][:-1])

    alltime = 0

    for i in **range**(2, -1,-1):

        alltime += time[i] \* (60\*\*(2-i))

    alltime = '%0.3f' %(float(alltime))

    start = '%0.3f' %(float(alltime) - spent+0.001)

    return([start, alltime])

def **solution**(lines):

    file = []

    for j in **range**(**len**(lines)):

        lines[j] = list(**map**(str, lines[j].split(' ')))

    for i in **range**(**len**(lines)):

        file.append(change(lines[i]))

        file[i][0] , file[i][1] = float(file[i][0]), float(file[i][1])

    answer = 0

    for i in **range**(**len**(file)):

        count1, count2 = 0,0

        a,b = file[i][0], file[i][1]

        for j in **range**(**len**(file)):

            if not (file[j][1] < a or file[j][0] >= a+1):

                count1 += 1

            if not (file[j][1] < b or file[j][0] >= b+1):

                count2 += 1

        answer = **max**(answer, count1, count2)

    return answer

#BOJ 1010번 다리 놓기

이 문제의 경우에는 그냥 누가 봐도 조합 문제였기 때문에 DP문제라는 것을 무시하고 펙토리얼 값을 구하는 함수를 정의해서 조합의 값을 계산해 주려 했다. 그러나 그렇게 하니 시간초과가 발생했고, 따라서 이항 정리의 파스칼 법칙을 이용하기로 했다. Dp[n][m]을 nCm의 값이라고 생각하여 nCr = n-1Cr-1 + n-1Cr임을 이용해서 동적 계획법을 적용했더니 시간 초과 없이 답이 간단하게 나왔다.

#BOJ 11404번 플로이드

n개의 도시, m개의 버스를 이용해서 한도시에서 다른 도시로 이동하는데 드는 비용을 알려주고 최소 시간을 각 도시 사이에서 구하는 것이다. 당연하게도 DFS를 이용해서 채우는 것이었는데, 2중 for문을 한번만 사용해서는 나중에 갱신된 값으로 이전에 계산 못한 값이 갱신이 되지 않아서 한번 for문을 사용할 때에 board[i][j] 와 board[j][i]를 모두 DFS함수로 갱신할 수 있도록 하였다.

#BOJ 2309번 일곱 난쟁이

개인적으로 그렇게 복잡한 문제는 아니었던 것 같다. 브루트 포스, 즉 완전 탐색 문제였는데 일단 모든 경우를 다 해보는 것이므로 제일 효율적으로 시간을 단축시킬 수 있는 방법을 생각해야 한다. 이 문제의 경우에는 9명의 정보를 주고 그중 7명을 선택해 키의 합이 100인 경우 중 하나만 찾아서 키를 모두 출력값으로 나오도록 하는 것이다. 그래서 전체중에 2명만 뽑아 전체 sum()에서 빼 주어서 출력하게 했다. 다만 여기서 100이 나오면 그만 탐색 해도 되는데 2중 for문이라 break로는 멈추기 어려워서 그냥 함수로 정의해서 return문을 사용해 loop에서 탈출 하였다.

#BOJ 3055번 탈출

BFS와 시뮬레이션이 섞인 문제였다. 이 문제를 푸는데 시간이 매우 오래 걸렸는데, 우선 물과 고슴도치 모두 매 분 사방으로 움직이는 상황이기 때문에 두개의 큐 자료 구조를 만들어야 했다. 이 상황에서 고슴도치가 움직인 이후 경과 시간을 표시해야 했는데 이를 위해서 check라는 리스트를 하나 더 만들고 지도 역할을 하는 board리스트는 주어진 기호만으로 채워서 갱신해 나갔다. 여기서 주의해야 할 점은 고슴도치는 물이 흐를 예정인 곳으로 이동해서는 안되기 때문에 먼저 물을 흐르게 한 뒤에 뒤이어 고슴도치의 움직임이 있도록 하는 것이다.

알고리즘을 순서대로 나열해 보자면

1. 2중 for문을 이용하여 ‘D’의 index, file이라는 큐에 ‘\*’의 index, move이라는 큐에 ‘S’의 index를 넣는다(‘S’는 ‘.’로 바꿔줌)

2. move큐에 아무 것도 없을 때까지 탐색을 반복한다.

1. 우선 len(file)만큼 for문을 돌려서 file안에 있는 값을 popleft()한다.

2. 이후에 dx, dy를 이용해 사방으로 이동시킨 뒤에 ‘.’이라면 ‘\*’으로 바꿔 준 뒤에 index를 file에 input()한다.

3. file을 모두 탐색 했다면 똑 같은 방법으로 move도 탐색한다.

1. 만약 움직인 board의 좌표가 ‘.’이면 check의 해당 좌표에 이전 부모노드의 check값 + 1로 갱신한다.

2. 좌표가 ‘D’이면 값만 갱신 해주고 break 한다.

3. ‘D’의 index에 해당하는 check의 값이 -1 이면 ‘CACTUS’를, 아니면 check값을 출력한다.

r,c = **map**(int, **input**().split())

from collections import deque

board = []

idx1, idx2 = 0,0

file, move = deque(), deque()

check = [[-1]\*c for x in **range**(r)]

for i in **range**(r):

    board.append(list(**input**().strip()))

for i in **range**(r):

    for j in **range**(c):

        if board[i][j] == '\*':

            file.append((i,j))

        if board[i][j] == 'S':

            move.append((i,j))

            check[i][j] = 0

            board[i][j] = '.'

        if board[i][j] == 'D':

            idx1, idx2 = i,j

dx, dy = [-1,1,0,0], [0,0,-1,1]

while move:

    num = **len**(file)

    for l in **range**(num):

        a,b = file.popleft()

        for i in **range**(4):

            x,y= a+dx[i], b+dy[i]

            if 0 <= x < r and 0<= y < c and board[x][y] == '.':

                board[x][y] = '\*'

                file.append((x,y))

    count = **len**(move)

    for l in **range**(count):

        e,f = move.popleft()

        for i in **range**(4):

            p,q = e+dx[i], f+dy[i]

            if 0<= p<r and 0<= q<c and check[p][q] == -1:

                if board[p][q] == '.':

                    check[p][q] = check[e][f] + 1

                    move.append((p,q))

                elif board[p][q] == 'D':

                    check[p][q] = check[e][f]+1

                    break

if check[idx1][idx2] == -1:

    print('KAKTUS')

else:

    print(check[idx1][idx2])

#프로그래머스 lv3.단어 변환

이 문제는 BFS를 이용해서 해결 할 수 있는 문제였다. 전반적으로 수월하게 진행 되었으나, 한 개의 테스트 케이스만 계속 답이 나오지를 않았다. 그 이유는 겹치지 않는 단어의 개수가 1개임을 확인하는 과정에서 set()를 이용하는데 중복된 알파벳, 즉 한 문자열안에 같은 알파벳이 여러 개 있을 것이라는 가능성을 무시 해 버린 것이다. 따라서 for문을 이용해서 두 리스트의 각 요소를 비교하는 것으로 방법을 바꾸었다.

#BOJ 2529 부등호

처음에는 제일 큰 수, 제일 작은 수를 greedy 알고리즘을 이용해 제일 처음에 나온 수들이 각각 최대, 최소가 될 수 있도록 최적화 시켜서 함수를 만들었으나, 계속해서 리스트의 길이가 원하는 것보다 길게 나왔다. 시간 초과를 우려하고 백트래킹(완전 탐색) 기법을 사용하지 않은 건데, 재귀를 바탕으로 백트래킹을 해야 답이 나왔다. 최대 최소는 매번 리스트의 길이가 k일 때마다 비교 후 갱신을 해 주었으며, 굳이 check리스트를 이용해서 중복을 검사 할 필요 없이 각각의 리스트에 있는 값인지 아닌지만 판단 해 주면 된다. 따라서 하나의 함수로도 충분히 시간 초과 없이 문제를 풀 수 있었던 것이다. 그리고 BFS의 큐에 첫 수를 넣는 것처럼 이 경우에도 리스트에 첫번째 수를 넣는 것을 for문을 이용해 0부터 9까지 진행한다 이전에 내가 푼 방법으로는 그 부분에서 문제가 생겼다. 무엇 보다 이런 문제에서는 첫번째에 최적화된 답이 나오게 하기는 어렵다. 그리고 모든 경우를 탐색하지 않으려다 보면 빼먹는 경우가 무조건 생기게 된다.

k = int(**input**())

file = list(**map**(str, **input**().split(' ')))

maxstr,minstr,maxnum,minnum = '','',-1,99999999999

def **DFS**(select, i):

    global maxstr, minstr, maxnum, minnum,k

    if i == k:

        num = int(''.join(**map**(str, select)))

        if num > maxnum:

            maxnum = num

            maxstr = select

        if num < minnum:

            minnum = num

            minstr = select

    else:

        if file[i] == '<':

            for l in **range**(select[i] + 1, 10):

                if l not in select:

                    DFS(select+[l],i+1)

        else:

            for l in **range**(select[i]):

                if l not in select:

                    DFS(select+[l],i+1)

for i in **range**(10):

    DFS([i], 0)

print(''.join(**map**(str, maxstr)))

print(''.join(**map**(str, minstr)))

#2020KAKAO BLIND RECRUITMENT 자물쇠와 열쇠

정답율이 7.4%인 문제로 생각해 내기까지 상당히 어려웠다. 처음에는 DFS기법을 이용해서 key리스트와 lock리스트의 각각 1인 좌표와 0인 좌표를 리스트에 모두 입력해 넣고 리스트들에 변화를 주는데, 이때 for x in range(4)를 해서 회전은 4번 시키고 반복해서 이동을 시켰는데, 한 칸 이동할 때마다 매번 모든 리스트의 요소를 뽑아서 바꾸고 다 바꾸면 lock리스트와 비교하는 과정이 매우 복잡했다.

그래서 lock배열을 확장한 expandlist를 만들어서 해결했다. 가운데에는 lock리스트를 고정하고 key리스트를 움직이면서 구한 것이다. 회전은 0, 90, 180, 270도 만큼 한 뒤에 겹치는 부분의 시작점을 lock리스트의 시작점부터 마지막 점까지 고려를 해주면서 putinkey라는 함수를 이용해서 해당 경우에 홈이 모두 채워지는 지 확인했다. 매번 새로운 리스트board를 만들어서 해당 칸 수만킁 이동하고 회전 한 상태를 모든 경우를 완전탐색을 통해 확인하는 것이다. 여기서 중요한 것은 board안에서 lock의 시작점과 끝점의 좌표를 자주 이용해야 하기 때문에 미리 변수에 저장을 해 두어서 헷갈리지 않는 것이다. 회전 하는 것은 따로 함수를 만들어서 for문을 이용해 실행 했다.

def **putinkey**(x,y,key,lock,length,start, end):

    n = **len**(key)

    board = [[0]\*length for x in **range**(length)]

    for i in **range**(x, n+x):

        for j in **range**(y, n+y):

            board[i][j] += key[i-x][j-y]

    for i in **range**(start, end):

        for j in **range**(start, end):

            board[i][j] += lock[i-start][j-start]

            if board[i][j] != 1:

                return False

    return True

def **rotateKey**(key):

    a= **len**(key)

    file = [[0]\*a for x in **range**(a)]

    for i in **range**(a):

        for j in **range**(a):

            file[a-j-1][i] = key[i][j]

    return file

def **solution**(key, lock):

    start =**len**(key)-1

    end = start+**len**(lock)

    n = start\*2+**len**(lock)

    for k in **range**(4):

        for i in **range**(end):

            for j in **range**(end):

                if putinkey(i,j,key, lock,n,start, end):

                    return True

        key = rotateKey(key)

    return False

완전 탐색은 하지만 리스트의 길이가 한정되어 있기 때문에 시간 복잡도는 큰 문제가 되지 않는다.

#프로그래머스 연습문제 2\*n타일링

처음에 그냥 dp = [0]\*n을 이용해서 리스트를 만든 뒤에 dp[i] = dp[i-1] + dp[i-2]임을 이용해서 값을 갱신했는데, 이유가 뭔진 모르겠지만 효율성테스트에서 2문제나 시간 초과가 발생했다. 그래서 그 이후에 그냥 리스트를 길이가 2가 되도록 새롭게 생성해서 n의 짝/홀 여부에 따라서 dp[0]과 dp[1]각각에 값을 갱신해 주었다. 마지막에 n번쨰를 구할 때에도 짝/홀여부에 맞게 나누어서 구해 주었다. 아무래도 자료가 길어지다 보면 dp[i-1], dp[i-2]일일히 검색하는 것이 O(n)의 사간 복잡도를 가지기 때문에 매번 검색하는 것이 시간 초과 발생의 원인이었을 것 같다.

#BOJ 14502번 연구소

이 문제는 DFS, BFS< 브루트포스(완전탐색)을 모두 사용해야 하는 문제였다. 문제 풀이 순서는 간단했다. 바이러스를 막을 벽 3개세우기(DFS), 벽을 세운 후에 바이러스 확산 경과 보기(BFS), 마지막에 최댓값을 갱신 해 주기 였다. 모든 경우를 탐색해서 최댓값을 구하는 것이었으므로 완전 탐색이 필요했다. 여기서 제일 까다로웠던 것은 원래의 입력 받은 board에 변화를 주지 않으면서 모든 경우를 해 보는 것이었다. 그래서 총 3개의 벽을 모두 세웠을 때 copy.deepcopy()를 이용해서 안의 객체까지도 다 복사를 하여서 check()함수에 넣어서 계산을 했다. 처음에는 board를 굳이 함수의 변수들에 넣어서 계산을 했었는데, 그렇게 하면 더 복잡하며 나중에 변수를 copy하기도 애매한 상황이었다.

특히 DFS를 이용해서 구할 때는 값에 변화를 주고 DFS로 재귀적으로 부른 뒤에 다시 그 값을 원래대로 돌려놓으면 이미 함수를 부른 것이기 때문에 다시 그 값에 변화가 있지 않기 때문에 걱정 할 필요가 없다.

n,m = **map**(int, **input**().split())

from collections import deque

board = []

import copy

import sys

for i in **range**(n):

    board.append(list(**map**(int, sys.stdin.readline().split())))

file = deque()

answer = 0

def **check**(board):

    dx,dy = [-1,1,0,0],[0,0,-1,1]

    countwall = 0

    for i in **range**(n):

        for j in **range**(m):

            if board[i][j] == 2:

                file.append((i,j))

            elif board[i][j] == 1:

                countwall += 1

    count = **len**(file)

    while file:

이 문제에 나온 것 처럼 어떤 특정 값, 이 문제의 경우 1이 나올때까지 계속해서 0을 2로 바꿔주는데 , 그 과정을 4 방으로 한다. 이럴 때 dx, dy를 이용해서 한번에 한 칸 씩, 다만 큐에 index를 pop, append하여서 BFS()로 계산하였다. While 1일 떄 까지 이런 식으로 하면 오히려 BFS를 사용하는 의미가 없어진다.

        a,b = file.pop()

        for i in **range**(4):

            da,db = a+dx[i], b+dy[i]

            if 0<= da<n and 0<=db<m:

                if board[da][db] == 0:

                    board[da][db] = 2

                    count += 1

                    file.append((da,db))

    return n\*m-countwall-count

answer = 0

def **setwall**(start, wallcount):

    global n,m,answer

    if wallcount == 0:

copyfile = copy.deepcopy(board)

        answer = **max**(answer, check(copyfile))

        return

    for i in **range**(start, n\*m):

        x,y = i//m, i%m

        if board[x][y] == 0:

            board[x][y] = 1

            setwall(i+1, wallcount-1)

            board[x][y] = 0

setwall(0,3)

print(answer)

#BOJ 1080 행렬

이 문제는 그리디 알고리즘 문제였다. 최적의 방법을 구하라고 했으며, 매번 최적의 상황을 선택할 때에 답이 나올 것이라는 것이 증명 가능했기 때문이다. 처음에는 매번 n-3, m-3의 for문을 사용하여 반복하여서 그 때마다 바꿔주면 이로운지 아닌지 확인해 주는 방법으로 했는데, 당연히 그렇게 하면 답이 나오질 않았다. 여기서의 핵심은 A, B두 행렬의 각각의 좌표가 같은지 아닌지 이기 때문에 그것을 확인해 주는 check라는 리스트를 만든 뒤에 같을 떄는 1, 다를 때는 0을 함으로서 확인 해주었다. 그렇게 순서대로 하다가 3\*3행렬의 마지막 좌표 일 때는 남은 수들이 모두 1이거나 0일 때에만 이어주었고 아니면 A와 B는 같을 수 없음을 의미하기 때문에 그냥 -1을 return하였다.

전부 탐색한 후에는 마지막 2줄을 확인 했는데, 여기서 n>3인 경우에만 확인하는 줄 알고 계속 오답이 나왔었다.

그리고 처음부터 행렬이 3\*3보다 작은 경우에는 A==B일 때는 0을, 아니면 -1을 출력하도록 해 주었다.

def **change**(a,index):

    x,y = index[0], index[1]

    for i in **range**(x,x+3):

        for j in **range**(y,y+3):

            if a[i][j] == 0:

                a[i][j] = 1

            else:

                a[i][j] = 0

def **answer**(n,m):

    global count

    for i in **range**(n-2):

        for j in **range**(m-2):

            if j == m-3:

                if **sum**(check[i][j:j+3]) == 0 or **sum**(check[i][j:j+3]) == 3:

                    if check[i][j] == 0:

                        count += 1

                        change(check,[i,j])

                    else:

                        continue

                else:

                    return(-1)

            else:

                if check[i][j] == 0:

                    count += 1

                    change(check,[i,j])

                else:

                    continue

    for i in **range**(n-2,n):

        for j in **range**(m-2,m):

            if check[i][j] == 0:

                return(-1)

    return count

n,m = **map**(int, **input**().split())

board =[]

for x in **range**(2):

    file = []

    for i in **range**(n):

        file.append(list(**map**(int, **input**().strip())))

    board.append(file)

A,B = board[0], board[1]

count = 0

if n < 3 or m< 3:

    if A!= B:

        print(-1)

    else:

        print(0)

else:

    check = [[0]\*m for x in **range**(n)]

    for i in **range**(n):

        for j in **range**(m):

            if A[i][j] == B[i][j]:

                check[i][j] = 1

    print(answer(n,m))

**그리디 알고리즘을 풀 때에 제일 중요한 것은 이렇게 매 순간 최적의 선택을 한 후에 나오는 답이 최적의 답이 되는지 증명이 가능해야 한다. 정답이 아닐 경우를 대비해서 모든 경우를 메모이제이션을 이용해 탐색해보는 DP알고리즘과 다른 점이다.**

#BOJ 11724 연결 요소의 개수

이 문제가 계속 시간 초과와 런타임 에러가 발생 했는데, 시간 초과는 input()대신 sys.stdin readline()으로 바꿔주니 해결이, 런타임 에러는 재귀의 깊이를 sys.setrecursionlimit(100000)을 선정해 주니 해결이 되었다.

이 문제와 같이 간선/노드와 관련된 문제들은 DFS로 풀면 쉽게 풀리는데, 중요한 것은 한번 제일 깊은 데까지 탐색 한 후에 다시 처음부터 조건에 맞는 것을 찾고 무엇보다 한번 함수를 호출 하면 함수를 더 이상 재귀적으로 부르지 않도록 하는 조건을 잘 설정해야 한다는 것이다.

비슷한 논리를 이용해도 하나의 방법은 시간 초과였고 아닌 게 있는데, 우선 이미 지나갔음을 의미하는 visit리스트를 만들지 않으면 안된다.

import sys

sys.setrecursionlimit(10000)

n,m = **map**(int, sys.stdin.readline().split())

visit = [0]\*n

file = []

board = [[0]\*n for x in **range**(n)]

for i in **range**(m):

    A = (list(**map**(int, sys.stdin.readline().split())))

    a,b = A[0],A[1]

    board[a-1][b-1] = 1

    board[b-1][a-1] = 1

    file.append(A)

def **DFS**(a):

    for i in **range**(n):

        if visit[i] == 0 and board[a][i] == 1:

            DFS(i)

count = 0

for i in **range**(n):

    if visit[i] == 0:

        count += 1

        DFS(i)

print(count)

*#DFS문제를 풀 때에 제일 중요한 것은 한 번 어떤 시작점으로부터 끝까지 '깊이 우선'으로*

*#전부 탐색을 했다면 다시 원점으로 돌아가 재귀를 멈추고*

*#또 다시 DFS로 탐색을 하는 형식이다*

*#이 문제와 같이 노드/간선연결 관련 문제는 DFS로 푸는 것이 훨씬 빠르다.*

*#다만 파이썬으로 풀이 할 경우 재귀제한이 걸려 있을 수 있어서 setrecursionlimit()를 해야 한다.*

*#방문하지 않은 것을 방문하는 것이 제일 중요하다*

*#DFS던 BFS던 탐색 문제에서 check/visit의 기능을 지닌 리스트는 필수이다.*

#BOJ 11403번 경로 찾기

이전에 풀었던 노드를 이용한 경로 탐색 문제였다. DFS문제였으나 BFS로 푸는 것이 더 효율적인 듯 보였다. 다만 다른 경우와 다른 이유는 두개의 노드를 모두 입력해야 했으며, 계속 무한 loop가 반복되지 않도록 하기 위해서 좌표가 같은 경우에는 큐에 새롭게 넣지 않고, 나머지 경우에는 계속 큐에 넣어서 반복해서 탐색을 시도했다. 한상 마지막에 출력 형식을 주의 할 것!

n = int(**input**())

from collections import deque

board = []

for i in **range**(n):

    board.append(list(**map**(int, **input**().split())))

file = deque()

for i in **range**(n):

    for j in **range**(n):

        if board[i][j] == 1:

            file.append((i,j))

while file:

    a,b = file.popleft()

    for i in **range**(n):

        if board[b][i] == 1 and board[a][i] == 0:

            board[a][i] = 1

            if a != i:

                file.append((a,i))

            if b != i:

                file.append((b,i))

for i in **range**(n):

    for j in **range**(n):

        print(board[i][j],end = ' ')

    print()

#BOJ 6603번 로또

이전에 풀었던 n과 m문제와 매우 흡사했으나, 반복되는 리스트에 맞게 문자열을 출력해야 한다는 점에서 상이했다. 처음에는 이 문제를 어떻게 해결 하는 것이 제일 효율적일지 감이 안왔는데, 역시나 DFS를 이용해서 재귀적으로 해결하면 되는 문제였다. 재귀 함수에서 항상 헷갈리던 것이 재귀함수를 새롭게 호출 하면 그 함수의 처리가 모두 끝난 다음에 다음으로 넘어간다는 것을 파악한 이후로 해결 되었다. 여기서 중요한 것은 오름차순으로 중복 없이 출력해야 하기 때문에 check[]라는 사용 여부를 기록해 놓는 리스트를 만드는 것이었다.

#BOJ 2468번 안전지역

이 문제는 처음에 BFS로 풀었을 때 계속해서 6%에서 머무르다 틀렸다고 했기 때문에 오랜 시간이 걸렸던 문제였다. 처음에는 해당 높이보다 낮은 높이를 모두 일치하게 표시한 뒤에 나머지를 모두 다른 index의 값으로 바꾼 뒤에 인접 영역일 경우에 BFS를 이용해서 더 높은 높이를 탐색해 가며 인접 영역은 부모 노드와 같은 index로 바꾸어서 나중에 set)를 이용해 총 다른 값의 개수를 구해 주었는데, 모든 높이가 같은 경우 등의 경우에는 해결을 하기가 어려웠다.

그래서 결국에는 기준 높이보다 높고 아직 탐색하지 않은 높이일 때에 새로운 영역의 시작으로 보고 그 지점을 시작으로 dx,dy를 이용해 인접 영역을 찾아주고 다른 영역은 다시 새롭게 for문을 이용해서 탐색을 한다. 그리고 여기서 같은 영역을 찾는 탐색 과정이 DFS와 BFS 모두로 각각 구현이 가능한 것이다.

#BFS를 이용하여 탐색한 방법

n = int(input())  
import sys  
from collections import deque  
board = [list(map(int,sys.stdin.readline().split())) for x in range(n)]  
answer = 0  
dx,dy = [-1,1,0,0],[0,0,-1,1]  
for i in range(100):  
 count = 0  
 check = [[0]\*n for x in range(n)]  
 file = deque()  
 for j in range(n):  
 for k in range(n):  
 if board[j][k] > i and check[j][k] == 0:  
 check[j][k] = 1  
 file.append((j,k))  
 count += 1  
 while file:  
 a,b = file.popleft()  
 for l in range(4):  
 x,y = a+dx[l], b+dy[l]  
 if 0<= x<n and 0<= y<n and board[x][y] > i and check[x][y] == 0:  
 check[x][y] = 1  
 file.append((x,y))  
 if count == 0:  
 break  
 answer = max(answer, count)  
print(answer)

#DFS를 이용하여 탐색한 방법

def DFS(a,b,c):  
 dx,dy = [-1,1,0,0],[0,0,-1,1]  
 for i in range(4):  
 x, y= b+dx[i], c+dy[i]  
 if 0<= x<n and 0<=y<n and board[x][y] > a and re[x][y] == 1:  
 re[x][y] = 0  
 DFS(a, x,y)

결과적으로 내가 생각한 대로 작은 높이의 값을 우선적으로 통일 시켜 준 뒤에 set()를 이용하는 것은 나중에 아무 것도 잡기지 않을 경우에 문제가 될 수 있다. 그리고 해당 높이보다 클 때의 좌표를 모두 큐에 넣었기 때문에 불필요한 중복된 계산이 많았을 것이다.

DFS/BFS 탐색을 시행하는 횟수가 곧 안전 영역의 개수이다.

#BOJ 1707 이분 그래프

DFS, BFS 모두를 이용하여 해결이 가능한 문제였다. 우선 처음에 시도한 방법은 DFS를 이용해 재귀적으로 함수를 호출하여 file1과 file2로 나누어서 해당 노드를 양쪽 모두의 리스트에 넣는 것이 불가능 하다면 재귀 함수를 다시 호출하지 않고, 아니라면 해당 노드를 넣은 리스트를 file1로 취급하여 checkgraph(j, board, file2, file1)로 재귀 함수를 호출하였다. 그러나 그렇게 한 결과 메모리 초과와 런타임 에러가 모두 발생하였고, 비록 코드의 알고리즘적인 논리는 모두 맞았지만 비효율적이라는 사실은 변함이 없었다.

k = int(input())  
import sys  
def checkgraph(i, board, file1, file2):  
 global v,e,ans  
 if i == v-1:  
 ans.append(0)  
 return  
 for j in range(i+1, v):  
 if board[i][j] == 1:  
 count = 0  
 for l in file2:  
 if board[j][l] == 1:  
 count += 1  
 if count == 0:  
 file2.append(j)  
 checkgraph(j, board, file2, file1)  
 else:  
 count = 0  
 for l in file1:  
 if board[l][j] == 1:  
 count += 1  
 if count == 0:  
 file1.append(j)  
 checkgraph(j, board, file1, file2)  
 file1.pop()  
 count = 0  
 for k in file2:  
 if board[k][j] == 1:  
 count += 1  
 if count == 0:  
 file2.append(j)  
 checkgraph(j, board, file2, file1)  
for \_ in range(k):  
 v, e = map(int, sys.stdin.readline().split())  
 board = [[0]\*v for x in range(v)]  
 for \_ in range(e):  
 a,b = map(int, sys.stdin.readline().split())  
 board[a-1][b-1], board[b-1][a-1] = 1,1  
 ans = []  
 checkgraph(0,board,[0],[])  
 if len(ans) > 0:  
 print(**'YES'**)  
 else:  
 print(**'NO'**)

그래서 어차피 두개의 집합으로 나누어서 분류를 할 것이기 때문에 길이가 v인 리스트를 만들어서 각각의 노드들이 포함된 집합을 1,2로 표시하고 일종의 check리스트의 역할을 하도록 방문하지 않은 노드는 0으로 표시 되도록 하였다.

k = int(input())  
import sys  
from collections import deque  
def BFS(i, color, check, file):  
 q = deque()  
 q.append(i)  
 check[i] = 1  
 color[i] = 1  
 while q:  
 now = q.popleft()  
 for j in file[now]:  
 if check[j] == 0:  
 color[j] = 3-color[now]  
 check[j] = 1  
 q.append(j)  
 else:  
 if color[j] == color[now]:  
 return False  
 return True  
  
for \_ in range(k):  
 v, e = map(int, sys.stdin.readline().split())  
 file = [[] for x in range(v)]  
 color, check = [0] \* v, [0]\*v  
 for \_ in range(e):  
 a,b = map(int, sys.stdin.readline().split())  
 file[a-1].append(b-1)  
 file[b-1].append(a-1)  
 ans = True  
 for i in range(v):  
 if check[i] == 0:  
 if not BFS(i, color, check, file):  
 ans = False  
 break  
 if ans == True:  
 print(**'YES'**)  
 else:  
 print(**'NO'**)

처음에 위에서 푼 방법에서는 DFS를 이용해서 끊임없이 재귀 함수를 불러 냈었지만, 그렇게 하니 메모리 초과가 sys.setrecursionlimit()으로도 복구가 어려워서(그러나 알고리즘은 맞고 반례도 없었다) BFS로 방법을 바꾸어 보았다. 각각의 노드가 연결된 노드들의 번호를 저장해 놓은 뒤에, 0번쨰 노드부터 순서대로, 아직 방문한 노드가 아닐 경우에 BFS를 이용해서 탐색을 시도한다. 색을 정해줄 때에 if else문을 이용할 필요가 없었던 건 두 색의 합이 3이므로 3-color[i]이런 식으로 진행하면 되는 것이었다.

이분 그래프의 핵심은 ‘간선으로 연결 되어있는 노드의 색이 같아서는 안된다는 것’이다. 다만 방문을 이미 했는지 안 했는지의 여부가 중요했고 전혀 어떠한 노드 와도 연결이 되어있지 않은 노드의 경우에는 어떠한 집단에 들어가도 상관이 없기 때문에 굳이 따로 예외적으로 고려를 해 줄 필요는 없다.

DFS와 BFS의 목적이 ‘탐색’이라는 것을 기억할 때 ‘연결되었는데 같은 색’인 노드를 ‘탐색’하는 것으로 함수를 만들었어야 한다.

#BOJ 1389 케빈베이컨 테스트

역시나 사람을 이용하였지만 연결 노드와 그래프 관련 문제였다. 이 문제의 경우 BFS로 풀었고, 앞서 푼 문제와 비슷한 방법으로 입력 값을 받았다. 원래 같으면 board에만 입력을 받았겠지만, 이번에는 리스트를 이용해 각각이 연결된 노드의 번호를 넣어서 BFS를 진행 하였다. 그리고 양방향 연결 노드인 것을 잊고 한쪽만 입력하는 실수를 저질러서 BFS탐색 과정에서 한다리 건너 만나는 것 이상으로 호출이 되지 않았다. 그것을 수정한 후에 정상적으로 작동이 되었다.

무엇보다 이런 문제에서 무한 루프가 발생하는 일이 없도록 방문 리스트를 꼭 만들어서 중복이 발생하지 않도록 하는 것이 중요하다. 그리고 혹시나 모를 상황을 위해 min을 이용해 케빈 베이컨 값을 갱신 했는데, 굳이 그럴 필요는 없었다. 어차피 이미 방문한 노드는 다시 케빈 베이컨 값을 찾을 일도 없고 한쪽에서 최소 다리이면 상대방도 마찬가지이기 때문이다.

#BOJ 1937 욕심쟁이 판다 (시뮬레이션)

처음에는 그냥 BFS를 반복적으로 적용하면 답이 나왔으나, 실제 온라인으로 코드를 돌렸을 때 계속해서 시간 초과가 발생하였다. 파이썬이 느린 것은 맞지만, pypy3으로도 시간 초과가 발생한 것으로 보아 코드의 알고리즘은 맞더라도 시간 복잡도를 해결해야만 했다. 그래서 메모이제이션을 이용해서 문제를 해결하려 했고, 그럼에도 불구하고 무언가 잘못됬는지 시간 초과가 계속 발생했다. 그러나 이 문제를 해결하려면 dp와 메모이제이션을 효율적으로 구현해야 한다.

처음에 dp를 구현할 때에 BFS 탐색을 이용해서 구하였고 dp[i][j]자리가 거기까지 가는데까지 걸린 시간이었기 때문에 오류가 발생했다. Dp[i][j]를 그 장소로부터 갈 수 있는 최대 날의 수로 저장하는 것으로 코드를 다시 바꾸어서 구현하였다.

import sys  
sys.setrecursionlimit(300000)  
n = int(input())  
board = [list(map(int, sys.stdin.readline().split())) for x in range(n)]  
dx, dy = [-1,1,0,0],[0,0,-1,1]  
def DFS(i,j,dp):  
 tree = board[i][j]  
 if dp[i][j]:  
 return dp[i][j]  
 dp[i][j] = 1  
 for k in range(4):  
 x,y = i + dx[k], j+dy[k]  
 if 0 <= x < n and 0<= y < n and tree < board[x][y]:  
 dp[i][j] = max(dp[i][j], DFS(x,y,dp)+1)  
 return dp[i][j]  
  
dp = [[0]\*n for \_ in range(n)]  
for i in range(n):  
 for j in range(n):  
 dp[i][j] = DFS(i,j, dp)  
print(max(sum(dp, [])))

DFS를 재귀적으로 호출하는 목적은 결국에 가장 멀리까지 이동 가능한 날의 수를 계산하는 것이기 때문에 dp롸 DFS를 이용하기 위해서 우선 이미 dp[i][j]의 값에 0이 아닌 다른 값이 존재 한다면 (0은 False로 인식) 그 값이 이전에 계산한 최댓값일 것으로 인식한다. 계속 board의 값이 더 큰 값으로 DFS를 호출 하다 보면 만약에 움직일 곳이 없으면 1로 그대로 출력이 될 것이고 dp의 원리를 이용해 그 값+1과 원래 dp자리의 값 중 더 큰 값을 이용하면 된다.

Dp/LIS(최장 증가 수열 알고리즘)/정렬 알고리즘 모두를 이용하되 탐색할 때에는 DFS를 함꼐 이용해준다. 그러나 여전히 핵심은 dp이다. 다만 나는 이 상황에서 DFS를 이용할 생각만 했지 정확히 어떻게 구현에 이용할지 제대로 생각하지 못했다.

#BOJ 빙산

이 문제는 BFS나 DFS로 풀 수 있는 문제였다. 우선 메모리 초과와 시간 초과의 장벽을 넘기가 너무 힘들었는데, 그것을 해결하기 위해 board의 빙하의 높이를 갱신해 주는 것과 연결된 덩어리의 개수를 구해주는 두개의 함수를 한번에 가능하도록 큐 자료구조를 두번 이용하였다.

import sys  
n,m = map(int, input().split())  
board = [list(map(int, sys.stdin.readline().split())) for x in range(n)]  
dx,dy = [-1,1,0,0],[0,0,-1,1]  
from collections import deque  
def BFS(i,j):  
 file = deque()  
 res = deque()  
 file.append((i,j))  
 check[i][j] = 1  
 while file:  
 i,j = file.popleft()  
 count = 0  
 for k in range(4):  
 x,y = dx[k] + i, dy[k] + j  
 if 0<= x<n and 0<= y < m and check[x][y] == 0 and board[x][y] != 0:  
 file.append((x,y))  
 check[x][y] = 1  
 elif 0<= x < n and 0<= y < m and check[x][y] ==0 and board[x][y] == 0:  
 count += 1  
 if count:  
 res.append((i,j,count))  
 return res  
year = 0  
while True:  
 piece = 0  
 check = [[0]\*m for x in range(n)]  
 for i in range(1, n-1):  
 for j in range(1, m-1):  
 if board[i][j] != 0 and check[i][j] == 0:  
 piece += 1  
 file = BFS(i,j)  
 while file:  
 a,b,c = file.popleft()  
 board[a][b] = max(0, board[a][b] - c)  
 if piece == 0:  
 print(0)  
 break  
 if piece >= 2:  
 print(year)  
 break  
 year += 1

#BOJ문자열 계산

파이썬 내장 함수 중에 eval 이라는 함수로 문자열이 계산이 되는데, 이때 주의해야 될 점은 사칙연산의 우선순위에 따라서 계산이 진행이 된다는 점이다. 그것만을 수정한 뒤에 함수를 두개를 만드니 해결이 전부 가능한 문제였다.

#BOJ 적록색약

이 문제는 BFS를 이용해서 푸는 것이 제일 수월하다. 일단 파이썬이라는 언어의 특성상 DFS를 이용하면 sys.setrecursionlimit(10\*\*6)을 설정하여서 재귀의 깊이 및 시간 초과 문제가, BFS를 이용하게 되면 메모리 초과가 발생하기 쉽다. 따라서 check리스트는 필수로, 최대한 각 문제당 탐색은 한번에, 그리고 동시에 하는 방법을 택하자. 이 경우 불가피하게 탐색은 두 번 해야 했으나, 진행하기 위해서 R과 G를 다른 색으로 판단하지 않는 적록 색약 탐색을 위해 board에 변화를 주는 과정은 정상인 사람을 탐색하는 도중에 바뀌도록 최대한 시간 복잡도를 줄였다.

n = int(input())  
import sys  
board = [list(map(str, sys.stdin.readline().strip())) for x in range(n)]  
from collections import deque  
ans = [0]\*2  
file = deque()  
dx, dy = [-1,1,0,0],[0,0,-1,1]  
def DFS(i,j):  
 check[i][j] = 1  
 c = board[i][j]  
 now = c  
 if board[i][j] == **'R'**:  
 board[i][j] = **'G'** for k in range(4):  
 x,y = i+dx[k], j+dy[k]  
 if 0<= x < n and 0<= y< n and check[x][y] == 0:  
 if now == board[x][y]:  
 DFS(x,y)  
  
  
for k in range(2):  
 check = [[0] \* n for x in range(n)]  
 for i in range(n):  
 for j in range(n):  
 if check[i][j] == 0:  
 DFS(i,j)  
 ans[k]+= 1  
print(ans[0], ans[1])

#BOJ 2294 동전2

이 문제는 동적 프로그래밍 문제로, 만약에 동전의 값들이 5단위로 나누어 떨어지는 등의 특수한 상황이라면 그리디를 쓰는게 맞겠지만 그게 아닐 경우에는 dp를 쓰는 것이 깔끔하다. 사실 이 문제를 해결하기 위해서 dp board를 n\*k의 크기의 리스트를 설정하면 안된다.

그리고 dp board를 처음에 들어있는 수를 0으로 하면 min을 점화식으로 실행 할 때에 자꾸 세부 사항까지 고려하게 만들고 만약에 그렇게 할 경우에 조건문이 너무 길어지게 된다. 그리고 굳이 나누어 떨어질 때 등등의 세부 사항은 고려 할 필요 없이 처음에 dp = [10001]\*(k+1)로 만들고 첫 수는 0으로 바꿔 준 뒤에 dp[j] = min(dp[j-coin[i]]+1, dp[j])를 코드를 짜면 답이 나온다.

n,k = map(int, input().split())  
coin = []  
**for** \_ **in** range(n):  
 coin.append(int(input()))  
coin.sort()  
dp = [10001]\*(k+1)  
dp[0] = 0  
**for** p **in** range(n):  
 i = coin[p]  
 **for** j **in** range(i,k+1):  
 dp[j] = min(dp[j-i]+1, dp[j])  
**if** dp[-1] == 10001:  
 print(-1)  
**else**:  
 print(dp[-1])

#BOJ 2146 다리 만들기

일단 BFS를 이용해 문제를 해결하려 할 때에는 방문 처리를 할 때 정점을 큐에 ‘넣을 때’ 진행해야 한다. 그 문제를 해결 해 주었더니 시간 초과가 한방에 해결이 되는 경이로운 경험을 할 수 있게 되었다. 그리고 DFS는 재귀를 반복 진행하므로 진행중인 모든 재귀를 멈출 수 있는 방법이 딱히 없어서 시간 초과가 발생하기 쉽다. 그러나 BFS는 그냥 return이나 break 한번으로 빠져나가는 것이 가능하므로 BFS를 애용하는 것이 나을 것도 같다.

num+= 1  
**def** bfs(a,b,now):  
 **global** count,ans  
 check[a][b] = 1  
 file.append((a,b,0))  
 **while** file:  
 i,j,c = file.popleft()  
 **for** k **in** range(4):  
 x,y = i+dx[k], j+dy[k]  
 **if** 0<=x<n **and** 0<=y<n **and** check[x][y] == 0:  
 **if** board[x][y] == 0:  
 file.append((x,y,c+1))  
 check[x][y] = 1  
 **elif** board[x][y] != now:  
 ans = min(ans, c)  
 **return**ans = 99999999  
**for** i **in** range(n):  
 **for** j **in** range(n):  
 **if** board[i][j] != 0:  
 check = [[0] \* n **for** \_ **in** range(n)]  
 file = deque()  
 bfs(i,j,board[i][j])  
print(ans)

n = int(input())  
**import** sys  
sys.setrecursionlimit(10\*\*8)  
board = [list(map(int, sys.stdin.readline().split())) **for** \_ **in** range(n)]  
dx,dy = [-1,1,0,0],[0,0,-1,1]  
**from** collections **import** deque  
**def** BFS(num, i,j):  
 board[i][j] = num  
 file = deque()  
 file.append((i,j))  
 **while** file:  
 a,b = file.popleft()  
 **for** k **in** range(4):  
 x,y = a+dx[k], b+dy[k]  
 **if** 0<= x<n **and** 0<=y<n:  
 **if** board[x][y] == 1:  
 file.append((x,y))  
 board[x][y] = num  
  
  
num = 2  
**for** i **in** range(n):  
 **for** j **in** range(n):  
 **if** board[i][j] == 1:  
 BFS(num,i,j)

#BOJ 1138 한 줄로 서기

이 문제는 그리디 알고리즘을 이용한 문제로, 처음에는 각각의 사람별로 더 큰 사람을 확인해 주어야 하는 줄 알았는데, 알고 보니 그게 아니라 그냥 deque()를 이용해서 해결하면 되는 문제였다. 그래서 out와 ans, 두개의 deque()를 이용해서 큰 수부터 큐에 넣고 그냥 그떄 그때 왼쪽에 있는 더 큰 사람의 개수를 맞추어 주었다. Deque 자료구조에서 appendleft()와 extend()라는 두개의 모듈이 있다는 것을 새롭게 알게 되었다. appendleft()는 시간 복잡도가 매우 낮게 맨 앞으로 밀어 넣는 것이고, extend는 두개의 큐를 연결하여 연장 시키는 것이다.

**import** sys  
**from** collections **import** deque  
n = int(input())  
people = list(map(int, sys.stdin.readline().split()))  
people = people[::-1]  
ans = deque()  
**for** i **in** range(n):  
 a = people[i]  
 b = n-i  
 count = 0  
 out = deque()  
 **while** ans:  
 **if** count == a:  
 **break** c = ans.popleft()  
 out.append(c)  
 **if** c > b:  
 count += 1  
 ans.appendleft(b)  
 out.extend(ans)  
 ans = out  
**for** i **in** ans:  
 print(i, end = **' '**)

#BOJ 7562 나이트의 이동

이 문제를 풀 때에 처음에는 큐 자료 구조에 일명 ‘노드의 깊이’를 넣어서 count += 1등의 방법 없이 횟수를 세어보려 했다. 그러나 그렇게 하니 메모리 초과가 발생 하였고, 그냥 x와 y좌표만 넣는 방법으로는 못 진행할까 고민을 한 결과 어차피 중복 방지를 위해서 check리스트가 필요한데 이 리스트에다가 +=1을 하는 방법으로 하면 되었다.

그리고 while 문과 for문을 이용한 반복문 중에 while문이 훨씬 시간 복잡도가 적음을 알 수 있게 되었다.

a,b = map(int, input().split())

x,y = map(int, input().split())  
 **if** a == x **and** y == b:  
 n -= 1  
 print(0)  
 **else**:  
 check = [[0]\*k **for** \_ **in** range(k)]  
 n -= 1  
 print(BFS(a,b,x,y))

**from** collections **import** deque  
n = int(input())  
dx,dy = [1,1,2,2,-1,-1,-2,-2], [2,-2,1,-1,2,-2,1,-1]  
**def** BFS(a,b,x,y):  
 **global** k,check  
 file = deque()  
 file.append((a,b))  
 check[a][b] = 1  
 **while** file:  
 c,d = file.popleft()  
 **if** c == x **and** d == y:  
 **return** check[c][d] -1  
 **else**:  
 **for** p **in** range(8):  
 i,j = c+dx[p], d+dy[p]  
 **if** 0 <= i < k **and** 0 <= j < k **and** check[i][j] == 0:  
 check[i][j] = check[c][d] + 1  
 file.append((i,j))  
**while** n:  
 k = int(input())

#BOJ 11048 이동하기

이 문제는 동적 계획법을 이용하는 문제였는데, 처음에는 어떤 식으로 해야 할지 고민이었고 더욱이 그동안 푼 동적 계획법이 한줄의 dp만으로 해결이 가능했기에 시행착오가 가능했으나 이 문제는 사방이 아니라 계속 오른쪽이나 아래로만 이동하고, 첫번쨰 타일과 마지막 타일을 반드시 통과해야 한다는 특이성을 갖고 있기 때문에 나름 쉽게 해결 할 수 있었다.

먼저 첫번쨰 가로줄과 세로줄 각각을 연속으로 더해준 뒤에 나머지를 DP로 해결 하니 쉽게 풀렸다.

**import** sys  
n,m = map(int, input().split())  
board = [list(map(int, input().split())) **for** \_ **in** range(n)]  
new = [[0]\*m **for** \_ **in** range(n)]  
new[0][0] = board[0][0]  
**for** i **in** range(1, m):  
 new[0][i] = new[0][i-1] + board[0][i]  
**for** i **in** range(1, n):  
 new[i][0] = new[i-1][0] + board[i][0]  
**for** i **in** range(1, n):  
 **for** j **in** range(1,m):  
 *#점화식* new[i][j] = board[i][j] + max(new[i-1][j-1], new[i][j-1], new[i-1][j])  
print(new[-1][-1])

#BOJ 15686 치킨 배달

이 문제는 애초에는 브루트 포스 문제였는데, 완전 탐색과 DFS/BFS를 이용해서 문제를 해결하려 했었다. 특히나 치킨집을 선택해야 하는데 선택하려는 m개를 일일히 정하는데에 DFS/BFS의 탐색 방법을 이용하려 했었다. 그러나 시간 초과가 발생하는 경우가 있어서 from itertools import combination이라는 모듈을 불러서 해결을 하기로 했다. 따라서 조합을 먼저 구한 뒤에 진행을 한 것이다. 그리고 최대 m개를 선택한다고 해서 0~m개인줄 알았는데 무조건 m개만 선택해야 했다.

문제를 풀기 전에는 조합을 구할 때 dfs를 이용하려 했는데, 그렇게 하는 것은 한 줄씩 print를 하는 등의 경우에만 가능하고 이렇게 경우의 수를 모두 저장 할 때에는 itertools를 이용하는 것이 훨씬 편리하다. 그러나 dfs를 이용하여 최소 치킨거리까지 구하는 방법은 존재한다.

**import** math,sys  
**from** itertools **import** combinations  
n,m = map(int, input().split())  
a,b = [],[]  
**for** i **in** range(n):  
 now = list(map(int, input().split()))  
 **for** j **in** range(n):  
 **if** now[j] == 1:  
 a.append((i,j))  
 **elif** now[j] == 2:  
 b.append((i,j))  
length = [[0]\*len(b) **for** \_ **in** range(len(a))]  
**for** i **in** range(len(a)):  
 **for** j **in** range(len(b)):  
 x,y,k,p = a[i][0], a[i][1], b[j][0], b[j][1]  
 length[i][j] = (abs(x-k)+abs(y-p))  
  
ans = 999999999  
file = list(combinations(b, m))  
**for** i **in** range(len(file)):  
 answer = [99999]\*len(a)  
 **for** j **in** range(len(file[i])):  
 num = b.index(file[i][j])  
 **for** k **in** range(len(a)):  
 answer[k] = min(answer[k], length[k][num])  
 ans = min(ans, sum(answer))  
print(ans)

**import** math,sys  
n,m = map(int, input().split())  
a,b = [],[]  
**for** i **in** range(n):  
 now = list(map(int, input().split()))  
 **for** j **in** range(n):  
 **if** now[j] == 1:  
 a.append((i,j))  
 **elif** now[j] == 2:  
 b.append((i,j))  
  
**def** dfs(v, next):  
 **global** ans  
 **if** v == m:  
 print(num)  
 add = 0  
 **for** j **in** range(len(a)):  
 x1,y1 = a[j][0], a[j][1]  
 now = 99999999999  
 **for** k **in** num:  
 x2,y2 = b[k][0], b[k][1]  
 now = min(now, abs(x1-x2)+abs(y1-y2))  
 add += now  
 **if** add < ans:  
 ans = add  
 **else**:  
 **for** i **in** range(next, len(b)):  
 num[v] = i  
 dfs(v+1, i+1)  
num = [0]\*m  
ans = 999999999  
dfs(0,0)  
print(ans)

#BOJ 2589 보물섬

BFS를 이용해서 해결하는 문제였는데, 처음에는 기존의 탐색 문제들이 많이 묻던 ‘최단/최소’를 묻는 문제가 아니어서 훨씬 복잡할 것이라고 생각했지만, 역시나 탐색알고리즘의 정석대로 풀면 간단히 해결되는 문제였다. 당연하게도 check리스트를 제대로 사용하지 않는다면 while 문에서 무한 루프를 돌게 될 것이기 때문에 육지가 나올 때마다 각 육지별로 가장 멀리 있는 육지까지 걸리는 시간을 측정하면 되고, 첫 욱지를 count에 포함하는 것을 방지하기 위해서 1을 뺀 값으로 갱신을 해 주었다.

**import** sys  
n,m = map(int, input().split())  
board = [list(map(str, sys.stdin.readline().strip())) **for** \_ **in** range(n)]  
**from** collections **import** deque  
answer = 0  
dx,dy = [-1,1,0,0],[0,0,-1,1]  
**def** dfs(i,j):  
 ans = 0  
 file = deque()  
 file.append((i,j))  
 check[i][j] = 1  
 **while** file:  
 a,b = file.popleft()  
 **for** k **in** range(4):  
 x,y = a+dx[k], b+dy[k]  
 **if** 0<= x<n **and** 0<=y<m **and** board[x][y] == **'L'**:  
 **if** check[x][y] == 0:  
 check[x][y] = check[a][b]+1  
 ans = max(ans, check[x][y]-1)  
 file.append((x,y))  
 **return** ans  
  
**for** i **in** range(n):  
 **for** j **in** range(m):  
 **if** board[i][j] == **'L'**:  
 check = [[0]\*m **for** \_ **in** range(n)]  
 answer = max(answer, dfs(i,j))  
 print(check)  
print(answer)

#BOJ 15663번 N과M(9)

이 문제는 한동안 조합 관련해서 구하는 것이 잘 안풀려서 bfs의 원리를 좀더 확실히 이해하고자 푼 문제이다. 이 문제를 풀기 위해 처음에는 set라는 자료구조에 넣어서 중복을 없게 하도록 했었는데, 그렇게 하다 보니 오름차순으로 정렬하도록 하는데에 어려움을 겪었다. 그러나 dfs로 탐색을 시행 할 때마다 출력을 하게 되면 자연스럽게 오름차순이 되기 떄문에 그렇게 따로 넣고 뺄 필요가 없었다. 그리고 탐색 문제에 걸맞게 check리스트를 이용하여 중복으로 선택은 피할 수 있도록 해 주었다.

**import** sys  
n,m = map(int, input().split())  
file = list(map(int, sys.stdin.readline().split()))  
file.sort()  
ans = set()  
**def** dfs(v,i,now):  
 **if** v == m:  
 **if** now **not in** ans:  
 print(now)  
 ans.add(now)  
 **return  
 elif** i == n:  
 **return  
 elif** v < m:  
 **for** k **in** range(n):  
 **if** check[k] == 0:  
 check[k] = 1  
 dfs(v+1, k,now+**' '** + str(file[k]))  
 check[k] = 0  
**for** i **in** range(n):  
 check = [0]\*n  
 check[i] = 1  
 dfs(1,i, str(file[i]))

#BOJ 11055번 가장 큰 증가 부분 수열

이 문제의 해결책은 찾았었으나 처음에는 답이 계속 틀리게 나왔다. 그러나 반례 하나를 찾고 보니 처음에 dp리스트를 [0]\*n이 아니라 그냥 입력 받은 file을 deepcopy해서 동적 프로그래밍을 시작해야 했다.

**import** sys,copy  
n = int(input())  
file = list(map(int, sys.stdin.readline().split()))  
dp = copy.deepcopy(file)  
**for** i **in** range(1, n):  
 **for** j **in** range(i):  
 **if** file[i] >file[j]:  
 dp[i] = max(dp[i], dp[j]+file[i])  
print(max(dp))

#BOJ 2225 합분해

이 문제는 사실 동적 계획법이라는 말이 없었더라면 훨씬 더 해결하기 어려웠을 것 같다. 일단 어떤 알고리즘을 이용해야 할지 알고 난 이후에 dp에 맞는 점화식을 찾는 데에 어려움을 느꼈다. 그러나 생각해 보니 어차피 합을 구하는 것이기 때문에 동적 계획법의 특성상 메모이제이션을 적극 활용해야 하기 때문에 풀어내기가 좀 쉬웠던 것 같다.

Dp[i][j]는 i개의 숫자를 이용하여 j라는 숫자를 만드는 것인데, 그렇다면 i-1개를 이용한 j보다 작거나 같은 수들의 경우의 수의 합을 구하게 된다면 답이 나올 것이라고 생각을 했다. 그리고 무조건 dp문제를 풀 때에는 dp의 맨 앞줄과 같은 첫줄 세팅을 제대로 해 주어야 한다.

n,k = map(int, input().split())  
dp = [[0]\*(n+1) **for** \_ **in** range(k)]  
**for** i **in** range(n+1):  
 dp[0][i] = 1  
**for** i **in** range(1,k):  
 dp[i][0] = 1  
 **for** j **in** range(1, n+1):  
 **for** k **in** range(j+1):  
 dp[i][j] += dp[i-1][k]  
  
print(dp[-1][-1]%1000000000)

그리고 알고리즘 문제를 풀 때에 답이 너무 길어지면 특정 수로 나눈 나머지를 구하라고 하는데 그럴 경우 주의를 해 주어야 한다.

#BOJ 14889 스타트와 링크

브루트 포스 문제 답게 그냥 bfs를 이용하면 금방 풀리는 문제이다. 그런데 n//2명씩 묶어서 조합을 구하려 할 때에 from intertools import collections 를 해 주는 것이 더 시간이 단축이 될 뿐 아니라 정확했다. 왜 내가 만든 bfs함수로 구하면 20%에서 틀렸다고 하는지 잘 모르겠다.

**import** sys, math  
n = int(input())  
board = [list(map(int, sys.stdin.readline().split())) **for** \_ **in** range(n)]  
answer = 99999999  
**def** addscore(file):  
 score = 0  
 **for** i **in** range(len(file)):  
 **for** j **in** range(i+1, len(file)):  
 score += board[file[i]][file[j]]  
 score += board[file[j]][file[i]]  
 **return** score  
**from** itertools **import** combinations  
people = [int(i)**for** i **in** range(n)]  
file = list(combinations(people, n//2))  
**for** i **in** range(len(file)):  
 left = []  
 **for** k **in** people:  
 **if** k **not in** file[i]:  
 left.append(k)  
 answer = min(answer, abs(addscore(left)-addscore(file[i])))  
print(answer)

#BOJ 14500번 테트로미노

처음에는 주어진 5개의 테트리스의 모양일 하나하나 구하도록 함수를 5개를 만들어야 하나 싶었다. 그러나 생각해 보니 그냥 기본 탐색처럼 dx,dy = [-1,1,0,0],[0,0,-1,1] 으로 하는 것이 맞았다. 그러나 이렇게 하면 ㅗ 모양의 테트리스 모양에 도달하는 것은 불가능했기에 그 경우만 따로 움직여야 하는 좌표의 변화를 리스트에 기록하였다. 이렇게 하니 풀리긴 했으나 pypy3으로만 맞았다고 나와서 python3로는 시간초과가 발생한다고 하여 파이썬의 시간 복잡도의 측면에 있어서의 한계를 확실히 알 게 되었다. 그리고 탐색은 dfs를 이용하였으며, check 리스트는 무조건 존재해야 하는 것이다.

**import** sys  
n,m = map(int, input().split())  
board = [list(map(int, sys.stdin.readline().split())) **for** \_ **in** range(n)]  
dx,dy = [1,-1,0,0],[0,0,-1,1]  
answer = 0  
**def** dfs(i,j,v,add):  
 **global** answer  
 **if** v == 4:  
 answer = max(add, answer)  
 **else**:  
 **for** k **in** range(4):  
 x,y = i+dx[k], j+dy[k]  
 **if** 0<= x<n **and** 0<= y< m:  
 **if** check[x][y] == 0:  
 check[x][y] = 1  
 dfs(x,y,v+1, add+board[x][y])  
 check[x][y] = 0  
file = [[[1,0,1],[0,1,-1]],[[0,1,-1],[1,0,1]],[[1,0,1],[0,-1,1]],[[0,-1,1],[1,0,1]]]  
**def** other(i,j):  
 **global** answer  
 **for** l **in** range(4):  
 add = board[i][j]  
 x,y = i,j  
 **for** p **in** range(3):  
 x,y = x+file[l][0][p], y+file[l][1][p]  
 **if** 0<=x<n **and** 0<=y<m:  
 add += board[x][y]  
 answer = max(add, answer)  
   
check = [[0] \* m **for** \_ **in** range(n)]  
**for** i **in** range(n):  
 **for** j **in** range(m):  
 check[i][j] =1  
 dfs(i,j,1,board[i][j])  
 other(i,j)  
 check[i][j] = 0  
print(answer)

#BOJ 1969 DNA

이 문제는 그리디 알고리즘을 이용하는 문제였는데, 처음에는 정답률이 57%정도여서 매우 쉬울 것으로 예상하였으나 생각보다 그렇지 만도 않았다. 우선 이 문제를 푸는 데에 있어서 각 세로, 즉 각 열마다 제일 자주 존재하는 알파벳을 이어 붙인 문자열을 만들 되 개수가 여러 개이면 알파벳 순으로 가장 앞에 있는 것이 선택 될 수 있도록 처리를 해 주었다. 그러다 보니 사실 처음에는 그렇게 이어 붙인 문자열이 입력으로 받은 board의 문자열 중에는 존재하지 않을 것이라고 생각했으나 알고 보니 그 문자열이 존재하기는 했다. 그래서 마지막에 한번 더 이중 for문을 돌리면서 hamming distance를 구해 주었다.

그런데 내가 푼 방법으로는 코드가 너무 길었고 계속해서 for문을 진행 시켜주어야 했기 때문에 dna의 종류가 ‘ACGT’의 4가지 밖에 없음을 이용해서 그 안에서 길이가 4인 리스트를 통해 값을 구해주니 숏코딩으로도 해결이 가능한 문제였다.

**import** sys  
n,m = map(int, input().split())  
board = [list(map(str,sys.stdin.readline().strip())) **for** \_ **in** range(n)]  
ans = 0  
dna = **'ACGT'**res = **''  
  
for** i **in** range(m):  
 file = [0,0,0,0]  
 **for** j **in** range(n):  
 file[dna.find(board[j][i])] += 1  
 a = file.index(max(file))  
 ans += file[a]  
 res += dna[a]  
print(res)  
print(n\*m-ans)

#BOJ 1965 상자 넣기

이 문제는 dp문제였는데, dp문제의 특성상 알고리즘을 잘 떠올리기만 하면 제대로 점화식을 세우면 간단하게 코딩이 끝난다. 이 문제도 마찬가지였으며, 문제의 규칙상 결국에 앞에 있는 상자의 크기 <= 뒤에 있는 상자의 크기를 만족하면 이어 나갈 수 있는 것이므로 순서대로 for문을 적용하여 본인보다 작은 크기의 상자가 나면 메모이제이션과 max()를 이용해 답을 구해 주었다.

#BOJ 1309 동물원

이 문제도 마찬가지로 dp 문제였다. 처음에는 각 칸 마다 dp로서 저장을 하여서 푸는 문제일 것이라고 생각했었다. 그런데 점화식이 나오지 않아서 n의 값을 기준으로 점화식을 세웠고, 실제로 하는 방법은 dp[i] = dp[i-1]\*2+dp[i-2]라는 것도 있었으나 뿐만 아니라 n번째 열에서 사자가 없는 경우, 오른쪽에 있는 경우, 왼쪽에 있는 경우 (가로로 분으면 안되므로 양쪽에 있는 경우는 불가능)으로 나누어서 계산을 해 주었더니 점화식이 쉽게 완성 되었다. 그리고 마지막에는 항상 조건 잘 보기! 9901로 나눈 나머지를 출력하지 않아서 출력 초과가 발생 했었다.

n = int(input())  
dp = [[0]\*3 **for** \_ **in** range(2)]  
dp[0][0], dp[0][1], dp[0][2] = 1,1,1  
**for** k **in** range(1,n):  
 i = k%2  
 g = (k-1)%2  
 dp[i][0], dp[i][1], dp[i][2] = sum(dp[g]), dp[g][0]+dp[g][2], dp[g][0]+ dp[g][1]  
a = sum(dp[(n-1)%2])  
print(9901%a)

#BOJ 1449 수리공항승

이 문제는 그리디 알고리즘을 이용하는 문제라고 인식을 하고 푸는 느낌은 아니었다. 그러나 풀다 보니 매 순간 최선의 선택을 한다는 감이 있기는 했다. 매 순간 길이 l의 테이프를 최소한으로 사용한다는 것이 곧 길이 1 만큼 사용하는 것이었기 때문에 남는 길이를 file이라는 리스트에 담아서 구했고, 그렇게 구한 이후에 file에 저장한 값과 이전 구멍의 위치와 현재 구멍의 위치를 비교하여 충분하지 않으면 ans += 1를 함으로서 구현을 했다.

**import** sys  
n,l = map(int, input().split())  
board = list(map(int, sys.stdin.readline().split()))  
board.sort()  
ans = 1  
file = [0]\*(n)  
file[0] = l-1  
**for** i **in** range(1, n):  
 **if** file[i-1] - (board[i]-board[i-1]) >= 0:  
 file[i] = file[i-1] - (board[i]-board[i-1])  
 **else**:  
 ans += 1  
 file[i] = l-1  
print(ans)

#BOJ 2503 숫자 야구

브루트 포스 문제 답게 모든 경우의 수를 탐색하는 문제여서 웬만하면 시간 초과에 걸릴 위험 없이 제대로 구현만 해주면 된다.처음에는 각 스트라이트/볼 조건에 맞는 숫자 리스트를 하나하나 만드는게 나을까도 생각했으나 그것 보다는 파이썬의 permutations 모듈을 이용하는 것이 편할 것 같아 그것을 이용하기로 했다. 그리고 그렇게 숫자 리스트를 구하고 if – in – 방법을 사용하려 했었으나 str과 int의 자료 형태가 헷갈려서 그냥 이중 for문으로 비교를 하여 스트라이트와 볼을 일일이 구했다.

Itertools 클래스에는 permutations, combinations, combinations\_with\_replacement가 있는데 순서대로 반복 불가능+모든 순서, 반복없음 + 정렬된 순서, 반복 있음 + 정렬된 순서 의 요소들을 합하여 리스트로 출력하여 준다.

**import** sys  
n = int(input())  
board = [list(map(int, sys.stdin.readline().split())) **for** \_ **in** range(n)]  
**from** itertools **import** permutations  
num = [int(i) **for** i **in** range(1, 10)]  
nums = list(permutations(num, 3))  
file = [1]\*len(nums)  
**def** check(a,b,c):  
 **global** file  
 **for** i **in** range(len(nums)):  
 **if** file[i] == 1:  
 now = nums[i]  
 p,q = 0,0  
 **for** x **in** range(3):  
 **for** y **in** range(3):  
 **if** int(now[x]) == int(a[y]):  
 **if** x == y:  
 p+=1  
 **else**:  
 q+=1  
 **if** p!=b **or** q!=c:  
 file[i] = 0  
  
**for** k **in** range(n):  
 a, b,c =str(board[k][0]), board[k][1], board[k][2]  
 check(a,b,c)  
ans = 0  
**for** i **in** range(len(file)):  
 **if** file[i] == 1:  
 ans += 1  
print(ans)

#BOJ 3085 사탕게임

솔직히 문제를 이해를 정확히 못해서 틀리는 경우가 더 많을 거 같다. 외국 문제를 번역한 것이다 보니 오류가 생긴것으로 보였는데, 우선 나는 모든 경우에서의 최대 길이를 찾는 줄 알았으나 그게 아니라 ‘인접한 두개의 칸’을 기준으로만 설정하고 그 칸에서의 가로열, 세로열만을 따져주어서 최대값을 갱신하면 된다. 그리고 해당 칸에서의 가로방향이면 오른쪽 왼쪽, 세로 방향이면 위 아래 모두 고려해야 한다. 브루트 포스문제여서 dfs나 bfs가 쓰일줄 알았는데 그렇지는 않았다.

**import** sys  
n = int(input())  
board = [list(map(str, sys.stdin.readline().strip())) **for** \_ **in** range(n)]  
ans = 0  
dx,dy = [1,0],[0,1]  
**def** check(file,i,j):  
 **global** ans  
 count1, count2 = 1,1  
 **for** x **in** range(i+1, n):  
 **if** file[x-1][j] == file[x][j]:  
 count1 += 1  
 **else**:  
 **break  
 for** x **in** range(i-1,-1,-1):  
 **if** file[x+1][j] == file[x][j]:  
 count1 += 1  
 **else**:  
 **break  
 for** y **in** range(j+1, n):  
 **if** file[i][y-1] == file[i][y]:  
 count2 += 1  
 **else**:  
 **break  
 for** y **in** range(j-1,-1,-1):  
 **if** file[i][y+1] == file[i][y]:  
 count2 += 1  
 **else**:  
 **break** ans = max(ans, count1, count2)  
  
**for** i **in** range(n):  
 **for** j **in** range(n):  
 **for** k **in** range(2):  
 **if** 0<= i+dx[k] < n **and** 0<= j+dy[k] < n:  
 board[i][j], board[i+dx[k]][j+dy[k]] = board[i+dx[k]][j+dy[k]], board[i][j]  
 check(board,i,j)  
 check(board, i+dx[k], j+dy[k])  
 board[i][j], board[i+dx[k]][j+dy[k]] = board[i+dx[k]][j+dy[k]], board[i][j]  
  
  
print(ans)

#프로그래머스 lv 3 여행경로

DFS, BFS를 이용해서 해결이 가능한 문제였으며, 우선 연결 노드 문제를 푸는 것 처럼 board라는 2차원 리스트에 연결된 노드에는 1을 넣어주었다. 그리고 나머지는 처음에는 bfs로 해결하려 했으나, 모든 경우를 다 탐색해 보아야 하는 백트래킹도 알고 보니 섞여 있었기 때문에 dfs로 해결하기로 마음을 바꿨다. 그 이후에 1번 tc만 통과가 되지 않아 ‘질문하기’를 참고해 보았는데, 그 결과 같은 경로/항공권이 2개 있을 수 있다는 내용이 있었고, 그 내용을 바탕으로 board를 1이 아니라 += 1, -= 1을 해주는 것으로 바꾸었더니 정답으로 처리가 되었다.

answer = []  
**def** dfs(v,now, ans,place,board,tickets):  
 **global** answer  
 **if** v == len(tickets):  
 answer.append(ans)  
 **return  
 else**:  
 **for** i **in** range(len(place)):  
 **if** board[now][i] == 1:  
 board[now][i] = 0  
 dfs(v+1, i, ans+str(i),place,board,tickets)  
 board[now][i] = 1  
  
**def** solution(tickets):  
 **global** answer  
 tickets = sorted(tickets, key = **lambda** x : (x[0],x[1]))  
 n = len(tickets)  
 place = set()  
 **for** i **in** range(n):  
 place.add(tickets[i][0])  
 place.add(tickets[i][1])  
 place = list(place)  
 place.sort()  
 board = [[0]\*len(place) **for** \_ **in** range(len(place))]  
 **for** i **in** range(n):  
 a,b = place.index(tickets[i][0]), place.index(tickets[i][1])  
 board[a][b] = 1  
 **for** i **in** range(len(place)):  
 **if** board[place.index(**'ICN'**)][i] == 1:  
 board[place.index(**'ICN'**)][i] = 0  
 dfs(1,i, str(place.index(**'ICN'**))+str(i),place,board,tickets)  
 board[place.index(**'ICN'**)][i] = 1  
 res = []  
 **for** i **in** range(len(answer)):  
 **if** len(answer[i]) == n+1:  
 **for** k **in** range(len(answer[i])):  
 res.append(place[int(answer[i][k])])  
 **break** print(res)

#BOJ 14503 로봇 청소기

이 문제는 삼성 기출 문제로, 시뮬레이션 문제였다. 솔직히 시뮬레이션 문제에 취약하고 오래 걸리는데, 그 이유는 다 꼼꼼히 모든 조건을 살펴보지 않기 때문이다. 그러나 이런 문제를 풀 때에 하나하나 알고리즘을 설계해 나가면 비교적 쉽게 풀 수 있다. 50%정도의 정답률을 갖고 있음에도 2시간이나 걸렸다는 점에서 반성할 필요가 있어 보인다. 우선 다 좋은데 ‘후진’이라는 키워드에서 실수가 있었다. 현재 방향에 따라 후진을 한 이후의 상태가 바뀔 텐데 무조건 가로축의 좌표가 1만큼 커지는 상황만 생각했기 때문에 답이 제대로 나오지 않았다. 그러나 break모멘트까지 잘 고려해 주고 오류까지 수정하고 나니 한번에 맞았다고 떴다. 제일 중요한 조건의 정리를 무조건 먼저 할 수 있도록 하자.

**import** sys  
n,m = map(int, input().split())  
r,c,head = map(int, input().split())  
board = [list(map(int, sys.stdin.readline().split())) **for** \_ **in** range(n)]  
dx,dy = [-1,0,1,0],[0,1,0,-1]  
direct = [[3,2,1,0],[0,3,2,1],[1,0,3,2],[2,1,0,3]]  
backx, backy = [1,0,-1,0],[0,-1,0,1]  
board[r][c] = 2  
clean = 1  
**def** check(r,c,head):  
 **global** clean  
 count = 0  
 **for** k **in** range(5):  
 **if** k < 4:  
 now = direct[head][k]  
 x,y = r + dx[now], c + dy[now]  
 **if** 0<= x < n **and** 0<= y< m:  
 **if** board[x][y] == 0:  
 board[x][y] = 2  
 clean += 1  
 check(x,y,now)  
 **break** count += 1  
 **else**:  
 **if** count == 4:  
 a,b = r+backx[head], c+backy[head]  
 **if** 0<= a<n **and** 0<= b<m:  
 **if** board[a][b] == 1:  
 **return  
 else**:  
 check(a,b,head)  
  
check(r,c,head)  
print(clean)

#BOJ 14499 주사위 굴리기

처음에는 전개도를 이용해서 좌표를 정해 앞면, 뒷면, 밑면 순서대로 생각 했었는데, 그렇게 하여서 뒤로 굴리면 앞면의 좌표는 ((a1+3)%4, a2)이런 식으로 정하는 등의 방법을 사용했었다. 그러나 그렇게 하니 헷갈렸고 자꾸만 주사위의 전개도가 아닌 부분에 숫자가 새겨지는 오류가 발생하였다. 따라서 그러한 오류가 발생하지 않도록 주사위의 순서대로 dice[0], dice[1], …dice[5]까지 정하고 dice[0]을 주사위의 현재 위치에서의 바닥면으로 설정하여 각 방향으로 굴릴 때의 변화를 노가다로 입력했다. 그리고 중요한 포인트 하나는 만약에 주사위를 굴릴 때에 지도 밖으로 나가게 되면 거기서 출력을 멈추어야 한다는 것이다. 그것만 조심하면 간단히 시뮬레이션 문제를 해결 할 수가 있었다.

**import** sys  
n,m,x,y,k = map(int, sys.stdin.readline().split())  
board = [list(map(int, sys.stdin.readline().split())) **for** \_ **in** range(n)]  
move = list(map(int, input().split()))  
dice = [0]\*6  
dice[0] = board[x][y]  
dx,dy = [0,0,-1,1],[1,-1,0,0]  
*#dice[0] is the bottom on the dice***def** copy(n):  
 **if** board[x+dx[n]][y+dy[n]] == 0:  
 board[x+dx[n]][y+dy[n]] = dice[0]  
 **else**:  
 dice[0] = board[x+dx[n]][y+dy[n]]  
 board[x+dx[n]][y+dy[n]] = 0  
**for** i **in** range(k):  
 check = 0  
 **if** move[i] == 1:  
 **if** 0 <= x+dx[move[i]-1] < n **and** 0 <= y+dy[move[i]-1] < m:  
 dice[0], dice[2], dice[4], dice[5] = dice[5], dice[4], dice[0], dice[2]  
 copy(move[i]-1)  
 check += 1  
 **elif** move[i] == 2:  
 **if** 0 <= x+dx[move[i]-1] < n **and** 0 <= y+dy[move[i]-1] < m:  
 dice[0], dice[2], dice[4], dice[5] = dice[4], dice[5], dice[2], dice[0]  
 copy(move[i]-1)  
 check += 1  
 **elif** move[i] == 3:  
 **if** 0 <= x+dx[move[i]-1] < n **and** 0 <= y+dy[move[i]-1] < m:  
 dice[0], dice[1], dice[2], dice[3] = dice[3], dice[0], dice[1], dice[2]  
 copy(move[i]-1)  
 check += 1  
 **else**:  
 **if** 0 <= x+dx[move[i]-1] < n **and** 0 <= y+dy[move[i]-1] < m:  
 dice[0], dice[1], dice[2], dice[3] = dice[1],dice[2], dice[3], dice[0]  
 copy(move[i]-1)  
 check += 1  
 **if** check == 1:  
 x,y = x+dx[move[i]-1], y+dy[move[i]-1]  
 print(dice[2])

#BOJ 15683 감시

이 문제는 브루트 포스를 이용해서 풀어야 하는 문제였다. 그런데 일단 그렇게 풀 때에 처음에는 그냥 board에 cctv가 있을 때마다 해당 위치를 기준으로 사방으로 빈공간을 계산해 cctv의 종류에 따라서 max의 값을 구해전체 개수에서 뺐었는데, 그렇게 했더니 중복되는 곳을 관찰하는 경우를 제외할 수가 없었다. 그래서

**import** sys  
**from** collections **import** deque  
n,m = map(int, input().split())  
board = [list(map(int,sys.stdin.readline().split())) **for** \_ **in** range(n)]  
empty = 0  
*#중복을 없애주기 위햐서 set, 즉 집합 자료구조 이용하기*up, down, left, right = 0,1,2,3  
dx,dy = [-1,1,0,0],[0,0,-1,1]  
all = []  
*#cctv의 종류에 맞게 모든 방향에서 관찰이 가능한 좌표/위치를 set()에 넣어 겹치지 않도록 한다.  
#while문을 사용했는데 break문을 제대로 사용하지 않아서 계속 무한 루프를 돌았었다.  
#while(1)과 while True는 같은 것을 의미한다.***def** watch(x,y,direction):  
 file = set()  
 **for** i **in** direction:  
 a,b = x,y  
 **while**(1):  
 a,b = a+dx[i], b+dy[i]  
 **if not**(0<= a< n **and** 0<= b<m):  
 **break  
 if** board[a][b] == 6:  
 **break  
 if** board[a][b] == 0:  
 file.add((a,b))  
 **return** file  
*#dfs,즉 깊이 우선 탐색 방법을 이용해서 가장 찾는 경우가 큰 것을 선택해 보았다.  
#여기서 항상 난제였던 것을 해결할 방법이 생겼다.  
#set끼리만 가능한 연산으로,*a|b *#합집합*a&b *#교집합*a-b *#차집합***def** dfs(n,union\_set):  
 **global** watched  
 **if** n == len(all):  
 watched = max(watched, len(union\_set))  
 **return  
 else**:  
 **for** i **in** all[n]:  
 dfs(n+1, union\_set|i)  
*#모든 board의 위치들 마다 cctv가 있을 때 종류에 맞게 가능한 모든 direction을 all이라는 리스트에 append한다.***for** i **in** range(n):  
 **for** j **in** range(m):  
 **if** board[i][j] == 0:  
 empty += 1  
 **elif** board[i][j] == 1:  
 all.append([watch(i,j,[up]),watch(i,j,[right]),watch(i,j,[left]),watch(i,j,[down])])  
 **elif** board[i][j] == 2:  
 all.append([watch(i,j,[up,down]), watch(i,j,[right, left])])  
 **elif** board[i][j] == 3:  
 all.append([watch(i,j,[up, right]), watch(i,j,[right, down]), watch(i,j,[down, left]),watch(i,j,[left, up])])  
 **elif** board[i][j] == 4:  
 all.append([watch(i,j,[up, right, down]), watch(i,j,[right, down, left]), watch(i,j,[down, left, up]), watch(i,j,[left, up, right])])  
 **elif** board[i][j] == 5:  
 all.append([watch(i,j,[up, right, left, down])])  
  
watched = 0  
dfs(0,set())  
print(empty - watched)

이렇게 풀어 보았다. 집합의 연산 법칙에 대해서도 알게 되었으며, 브루트 포스는 역시 경우마다 탐색, 조건에 따라 함수를 만들어서 구하는 것이 제일 편하다는 것을 알게 되었다.

#BOJ 2096 내려가기

이 문제를 풀 수 있는 방법은 거의 동적 계획법밖에 없을 것이라고 충분히 확신이 가능했다. 그래서 그냥 어차피 가로 길이가 3으로 전부 일정할 뿐만 아니라 각각의 경우에 더해지는 값이 일정하기 때문에 구하기가 쉬울 것이라고 생각했다. 각각의 칸에 하나의 값이 아니라 하나는 최대, 다른 하나는 최소로 갱신을 하도록 저장을 했다. 그럼에도 불구하고 정답률이 33%밖에 안된다는 점에서 좀 의아 했는데, 아마 메모리 초과 때문이 아니었을까 싶다. 그래서 그냥 열의 길이를 2로 해서 구했더니 답이 맞았다고 나왔다.

**import** sys  
n = int(input())  
board = [list(map(int, sys.stdin.readline().split())) **for** \_ **in** range(n)]  
dp = [[[0,0]**for** \_ **in** range(3)] **for** \_ **in** range(2)]  
  
**for** i **in** range(3):  
 dp[0][i][0], dp[0][i][1] = board[0][i], board[0][i]  
**for** i **in** range(1, n):  
 a,b = i%2, (i-1)%2  
 **for** j **in** range(3):  
 **if** j == 0:  
 dp[a][j][0] = board[i][0] + min(dp[b][0][0], dp[b][1][0])  
 dp[a][j][1] = board[i][0] + max(dp[b][0][1], dp[b][1][1])  
 **elif** j == 1:  
 dp[a][j][0] = board[i][j] + min(dp[b][0][0], dp[b][1][0], dp[b][2][0])  
 dp[a][j][1] = board[i][j] + max(dp[b][0][1], dp[b][1][1], dp[b][2][1])  
 **else**:  
 dp[a][j][0] = board[i][j] + min(dp[b][1][0], dp[b][2][0])  
 dp[a][j][1] = board[i][j] + max(dp[b][1][1], dp[b][2][1])  
big, small = 0,999999999  
now = (n-1)%2  
**for** i **in** range(3):  
 big = max(big, dp[now][i][1])  
 small = min(small, dp[now][i][0])  
print(big, small)

#BOJ 10844-쉬운 계단 수, 1562-계단 수

일단 동적 계획법 문제를 푸는 데에 제일 중요한 것은 ‘어떤 것을 저장해주고 어떤 것을 꺼내 써야 할까?’이다. 10844번 문제의 이 질문에 대한 해답은 ‘일의 자리에 0~9까지의 경우의 수를 1로 다 초기화 시킨 다음에 자릿수를 하나씩 늘려가면서 이전 자릿수의 경우의 수를 받아오는 것이다.

이런 식으로 해결하면 되는 문제였는데, 중요한 것은 내가 세로를 끝나는 숫자로, 가로를 자리수로 했더니 답이 나오지 않았다는 것이다. 왼쪽 코드는 세로를 시작하는 숫자, 가로를 자릿수로 설정해서 구했다. 주의할 점은 마지막에 답을 구할 때 0으로 시작하는 경우는 더하는 것에 포함시키지 말아야 한다는 것이다.

내가 원래 한대로 풀면 왜 답이 안 나오는지 의문이다.

n = int(input())  
dp = [[0]\*n **for** \_ **in** range(10)]  
**for** i **in** range(1,10):  
 dp[i][0] = 1  
**for** i **in** range(10):  
 **for** j **in** range(1,n):  
 **if** i > 0:  
 dp[i][j] += dp[i-1][j-1]  
 **if** i < 9:  
 dp[i][j] += dp[i+1][j-1]  
ans = 0  
**for** i **in** range(10):  
 ans += dp[i][n-1]  
print(ans%1000000000)

이렇게 했는데 뭐가 문제인지는 아직도 찾지 못했다.

n = int(input())  
dp = [[0]\*10 **for** \_ **in** range(n)]  
**for** i **in** range(10):  
 dp[0][i] = 1  
**for** i **in** range(1,n):  
 **for** j **in** range(10):  
 **if** j > 0:  
 dp[i][j] += dp[i-1][j-1]  
 **if** j < 9:  
 dp[i][j] += dp[i-1][j+1]  
ans = 0  
**for** i **in** range(1,10):  
 ans += dp[-1][i]  
print(ans%1000000000)

1562번은 솔직히 말하면 도무지 어떤 방법으로 0~9까지의 모든 수를 사용하였음을 표현 가능한지 너무 해결하기 어려워서 정답 코드를 참고 했다. 이 코드를 공부하면서 파이썬을 좀더 효율적으로, 그리고 코드 짤 때에 좀더 짧게 조건문을 다룰 수 있는 방법을 여러 개 터득 할 수 있었다.

이 문제를 해결하면서 알게 된 것이 바로 비트연산자(Bitwise Operators)이다.

#Bitwise Operators

a = 0011 1100 이고 b = 0000 1101이라 할때

| **Operator** | **Description** | **Example** |
| --- | --- | --- |
| & | AND 연산. 둘다 참일때만 만족 | (a & b) = 12 → 0000 1100 |
| | | OR 연산. 둘 중 하나만 참이여도 만족 | (a | b) = 61 → 0011 1101 |
| ^ | XOR 연산. 둘 중 하나만 참일 때 만족 | (a ^ b) = 49 → 0011 0001 |
| ~ | 보수 연산. | (~a) = -61 → 1100 0011 |
| << | 왼쪽 시프트 연산자. 변수의 값을 왼쪽으로 지정된 비트 수 만큼 이동 | a << 2 = 240 → 1111 0000 |
| >> | 오른쪽 시프트 연산자. 변수의 값을 오른쪽으로 지정된 비트 수 만큼 이동 | a >> 2 = 15 → 0000 1111 |

n = int(input())  
re = 0  
dp = [[0 **for** \_ **in** range(1024)]**for** \_ **in** range(10)]  
  
**for** i **in** range(1, 10):  
 *#1<<i 라는 것은 1의 비트를 i만큼 왼쪽으로 이동시키는 것을 의미(이진수응 뒤에서부터 연산 시작함)* dp[i][1<<i] = 1  
**for** i **in** range(1,n):  
 dp\_next = [[0 **for** \_ **in** range(1024)] **for** \_ **in** range(10)]  
 **for** k **in** range(10):  
 *#0~9까지의 모든 수들을 이용해서 수를 만들기 위해서 자리수의 크기를 계속 하나씩 늘려가며 + 비트 연산자를 이용해서 아직 사용하지 않은 숫자를 찾아 떠남* **for** j **in** range(1024):  
 **if** k < 9:  
 dp\_next[k][j | (1<<k)] = (dp\_next[k][j | (1<<k)] + dp[k+1][j]) % 1000000000  
 **if** k > 0:  
 dp\_next[k][j | (1 << k)] = (dp\_next[k][j | (1 << k)] + dp[k - 1][j]) % 1000000000  
 dp = dp\_next  
*#해보니 비트 연산자는 1024가 이진법으로 1111111111이기 때문에 무조건 길이가 1024일 경우에 사용이 가능하다.  
#이 문제의 경우와 같이 bool자료 구조를 이용해서 하나하나 check리스트를 만들면서 확인하기 힘든 경우에 사용하면 좋다.*print(sum([dp[i][1023] **for** i **in** range(10)]) % 1000000000)

#BOJ 1051 숫자 정사각형

이 문제를 푸는 방법은 브루트 포스였다. 각 방향으로 쭉 끝까지 이어 나갔을 때 그 자리가 초기 board의 값과 같으면 리스트에 +=1을 하도록 한 뒤에 나중에 list의 값이 3인 것들로만 big = max(big, i+2)로 해서 구했다. 그러나 계속 31%에서 멈추길래 보니까 크기가 1인 경우에 정사각형으로 고려해 주지 않았기 때문이었고, 그 부분만 수정해 주었더니 정답 처리 되었다.

**import** sys  
n,m = map(int, input().split())  
board = [list(map(int, sys.stdin.readline().strip())) **for** \_ **in** range(n)]  
dx,dy = [1,0,1],[0,1,1]  
**def** bfs(i,j):  
 ans = [0]\*(max(n-i-1,m-j-1))  
 **for** k **in** range(i+1, n):  
 **if** board[k][j] == board[i][j]:  
 ans[k-i-1] += 1  
 **for** k **in** range(j+1, m):  
 **if** board[i][k] == board[i][j]:  
 ans[k-j-1] += 1  
 x,y = i+1,j+1  
 **while** 0<= x<n **and** 0<=y<m:  
 **if** board[x][y] == board[i][j]:  
 ans[x-i-1]+= 1  
 x,y = x+1, y+1  
 big = 1  
 **for** i **in** range(len(ans)):  
 **if** ans[i] == 3:  
 big = max(big, i+2)  
 **return** big  
  
ans = 0  
**for** i **in** range(n):  
 **for** j **in** range(m):  
 ans = max(ans, (bfs(i,j))\*\*2)  
print(ans)