**동적 계획법(Dynamic Programming)**

#1010 – 다리 놓기

|  |
| --- |
| 재원이는 한 도시의 시장이 되었다. 이 도시에는 도시를 동쪽과 서쪽으로 나누는 큰 강이 흐르고 있다. 하지만 재원이는 다리가 없어서 시민들이 강을 건너는데 큰 불편을 겪고 있음을 알고 다리를 짓기로 결심하였다. 강 주변에서 다리를 짓기에 적합한 곳을 사이트라고 한다. 재원이는 강 주변을 면밀히 조사해 본 결과 강의 서쪽에는 N개의 사이트가 있고 동쪽에는 M개의 사이트가 있다는 것을 알았다. (N ≤ M)  재원이는 서쪽의 사이트와 동쪽의 사이트를 다리로 연결하려고 한다. (이때 한 사이트에는 최대 한 개의 다리만 연결될 수 있다.) 재원이는 다리를 최대한 많이 지으려고 하기 때문에 서쪽의 사이트 개수만큼 (N개) 다리를 지으려고 한다. 다리끼리는 서로 겹쳐질 수 없다고 할 때 다리를 지을 수 있는 경우의 수를 구하는 프로그램을 만들어 각 테스트 케이스에 대해 주어진 조건하에 다리를 지을 수 있는 경우의 수를 출력한다. |

|  |  |
| --- | --- |
| t = int(input())  *def* bridge(n,m):      dp = [[0]\*(m-n+1) *for* \_ in range(n)]  *for* i in range(n):  *for* j in range(m-n+1):  *if* i == 0:                  dp[i][j] = 1  *elif* j == 0:                  dp[i][j] = 1  *else*:                  dp[i][j] = dp[i][j-1] + dp[i-1][j]  *return* sum(dp[-1])  *for* i in range(t):      n,m = map(int, input().split())  *print*(bridge(n,m)) | 동적 계획법 문제들은 배열의 각 상태의 의미와 정확한 점화식만 세울 수 있다면 코드를 짜는 것 자체에는 무리가 별로 없다.  이 문제의 경우에는  dp[i][j] = i번째 사이트가 j번째에 연결될 경우의 수 이고  dp[i][0] = 1, dp[0][j] = 1임을 무조건 만족시켜야 하며  나머지의 경우  dp[i][j] = dp[i-1][j] + dp[i][j-1]이다.  이렇게 구한 뒤 마지막에는 dp리스트의 마지막 줄의 수들의 합을 출력하면 된다. |

#14501 – 퇴사

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 상담원으로 일하고 있는 백준이는 퇴사를 하려고 한다. 오늘부터 N+1일째 되는 날 퇴사를 하기 위해서, 남은 N일 동안 최대한 많은 상담을 하려고 한다. 백준이는 비서에게 최대한 많은 상담을 잡으라고 부탁을 했고, 비서는 하루에 하나씩 서로 다른 사람의 상담을 잡아놓았다. 각각의 상담은 상담을 완료하는데 걸리는 기간 Ti와 상담을 했을 때 받을 수 있는 금액 Pi로 이루어져 있다. N = 7인 경우에 다음과 같은 상담 일정표를 보자.     |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | |  | 1일 | 2일 | 3일 | 4일 | 5일 | 6일 | 7일 | | Ti | 3 | 5 | 1 | 1 | 2 | 4 | 2 | | Pi | 10 | 20 | 10 | 20 | 15 | 40 | 200 |   1일에 잡혀있는 상담은 총 3일이 걸리며, 상담했을 때 받을 수 있는 금액은 10이다. 5일에 잡혀있는 상담은 총 2일이 걸리며, 받을 수 있는 금액은 15이다. 상담을 하는데 필요한 기간은 1일보다 클 수 있기 때문에, 모든 상담을 할 수는 없다. 예를 들어서 1일에 상담을 하게 되면, 2일, 3일에 있는 상담은 할 수 없게 된다. 2일에 있는 상담을 하게 되면, 3, 4, 5, 6일에 잡혀있는 상담은 할 수 없다. 또한, N+1일째에는 회사에 없기 때문에, 6, 7일에 있는 상담을 할 수 없다. 퇴사 전에 할 수 있는 상담의 최대 이익은 1일, 4일, 5일에 있는 상담을 하는 것이며, 이때의 이익은 10+20+15=45이다. 상담을 적절히 했을 때, 백준이가 얻을 수 있는 최대 수익을 구하는 프로그램을 작성하시오. |

|  |  |
| --- | --- |
| #dfs이용해서 모든 경우 확인하기(dp나 메모이제이션을 이용 안함)  *import* sys  n = int(input())  day, cost = [],[]  *for* \_ in range(n):      a,b = map(int, sys.stdin.readline().split())      day.append(a)      cost.append(b)  ans = 0  *def* choose(idx, money):  *global* ans  *if* idx == n:  *return*  *else*:  *for* k in range(idx, n):              a,b = day[k], cost[k]  *if* a + k <= n:                  ans = max(ans, money+b)                  choose(k+a, money+b)  choose(0,0)  *print*(ans) | #dp를 이용해서 해결하는 방법(이 경우도 dfs와 혼합)  *import* sys  n = int(input())  day, cost = [], []  *for* \_ in range(n):      a,b = map(int, sys.stdin.readline().split())      day.append(a)      cost.append(b)  ans = 0  dp = [0]\*(n+1)  *def* dfs(v):  *if* v == n:  *return*  *else*:          a,b = day[v], cost[v]  *if* a+v <= n:              dp[a+v] = max(dp[a+v], dp[v]+b)              dfs(a+v)          dp[v+1] = max(dp[v], dp[v+1])          dfs(v+1)  dfs(0)  *print*(max(dp)) |

#1082 – 방 번호

|  |
| --- |
| 이번에 VIP 회장으로 새로 부임한 백은진은 빅뱅의 위대함을 세계에 널리 알리기 위해서 사무실을 하나 임대했다.빅뱅은 위대하기 때문에, 사무실의 번호도 되도록이면 커야 한다고 생각한다. 따라서 지금 가지고 있는 돈 전부를 가지고 방 번호를 만들려고 한다. 1층에 있는 문방구에서는 숫자를 판다. 각 숫자의 가격은 서로 다를 수 있기 때문에, 현재 가지고 있는 돈을 이용해서 만들 수 있는 가장 큰 숫자를 만들려고 한다. 예를 들어, 문방구에서 파는 숫자가 0, 1, 2이고, 각 숫자의 가격이 6, 7, 8이고, 백은진이 현재 가지고 있는 돈이 21이라면, 백은진이 만들 수 있는 가장 큰 수는 210(8+7+6=21)이다. |

|  |  |
| --- | --- |
| *import* sys  n = int(sys.stdin.readline())  cost = list(map(int, sys.stdin.readline().split()))  money = int(sys.stdin.readline())  dp = [*'0'*]\*(money+1)  *for* i in range(n-1,-1,-1):      c = cost[i]  *for* j in range(c, len(dp)):          dp[j] = dp[j-c]+str(i) *if* int(dp[j]) <= int(dp[j-c]+str(i)) *else* dp[j]  *print*(int(dp[-1])) | 그리디 알고리즘적인 관점에서 볼 때 자리수가 많아야 하고 앞자리일수록 더 큰 수가 와야 한다. 그래서 비용이 작은 수로 자리 수를 최대한 길게 구성한 뒤에 앞자리부터 가능한 큰 수로 바꾸는 작업을 해야 한다.  그러나 여기서 dp리스트를 2차원 리스트로 설정하였을 때에는 답이 나오지 않았다.  모든 자리가 0인 경우에는 다른 수로 대체가 가능한지의 여부를 반드시 확인해 보야야 한다.  계속해서 오류가 발생했기 때문에 한번 더 생각해 봐야 할 문제이다. |

|  |
| --- |
| -RESOLVE- |

#1932 – 정수 삼각형

|  |  |
| --- | --- |
| #bfs로 모든 경우를 다 해 보았더니 당연하게도 시간 초과  *import* sys  n = int(sys.stdin.readline())  board = [list(map(int, sys.stdin.readline().split()))  *for* \_ in range(n)]  answer = 0  *def* dfs(x,y,ans):  *global* answer  *if* x+1 == n:          answer = max(answer, ans)  *return*  *else*:          dfs(x+1,y,ans+board[x+1][y])          dfs(x+1,y+1,ans+board[x+1][y+1])    dfs(0,0,board[0][0])  *print*(answer) | #dp를 이용하여 푸는 것보다는 메모이제이션을 이용  *import* sys  n = int(sys.stdin.readline())  board = [list(map(int, sys.stdin.readline().split())) *for* \_ in range(n)]  *for* i in range(n-1):  *for* j in range(len(board[i+1])):  *if* j == 0:              board[i+1][j] += board[i][j]  *elif* j == len(board[i+1])-1:              board[i+1][j] += board[i][-1]  *else*:              board[i+1][j] += max(board[i][j], board[i][j-1])  *print*(max(board[-1])) |

#2293 – 동전 1

|  |  |
| --- | --- |
| *import* sys  n,k = map(int, sys.stdin.readline().split())  num = [int(sys.stdin.readline()) *for* \_ in range(n)]  num.sort()  dp = [[0]\*(k+1) *for* \_ in range(2)]  dp[0][0], dp[1][0] = 1,1  *for* i in range(n):      now = num[i]  *for* j in range(1,k+1):  *if* i == 0:              dp[i%2][j] = 1  *else*:  *if* j >= now:                  dp[i%2][j] = dp[(i-1)%2][j]+dp[i%2][j-now]  *else*:                  dp[i%2][j] = dp[(i-1)%2][j]  *print*(dp[(n-1)%2][-1]) | 일단 dp 리스트의 크기를 n개만큼 하면 메모리 초과가 발생한다. 그러나 2개로 해서 2로 나눈 나머지를 indx로 설정하면 메모리 초과가 발생할 일은 없다.  처음에는 dp[i][j]를 i번째까지의 수를 이용해서 j를 만드는 경우의 수로 설정하여 각 행의 마지막 수를 모두 더하려 했으나 생각해 보니 당연히 중복이 발생할 수 밖에 없는 방법이었다.  그래서 일반적인 냅색 알고리즘 처럼 해당 값보다 작을 때에는 이전 행의 값을 이용하고 해당 값보다 크거나 같은 경우에는 dp[i][j] = dp[i-1][j] + dp[i][j-cost[i]]를 해준다.  Python3, pypy3모두 시간 초과없이 통과가 되었다. |

#1149 – RGB 거리

|  |  |
| --- | --- |
| *import* sys,copy  n = int(sys.stdin.readline())  board = [list(map(int, sys.stdin.readline().split()))  *for* \_ in range(n)]  dp = [[0]\*3 *for* \_ in range(n)]  *for* i in range(3):      dp[0][i] = board[0][i]  *for* v in range(1,n):  *for* i in range(3):          dp[v][i] = min(dp[v-1][(i+1)%3], dp[v-1][(i+2)%3]) + board[v][i]  *print*(min(dp[-1])) | 함수를 이용해서 구하면 왜 런타임 에러가 계속 발생하는지 모르겠다.  check(v,dp)함수를 설정해서 for문으로 구현할 수 있는 것을 함수에 밀어 넣었을 뿐인데 말이다.  어쨌든 이 문제는 매우 쉽고, 딱히 머리를 사용해 dp를 구현하니 않아도 처음에 주어지는 자료의 세로 길이가 총 집의 개수이고 가로길이가 RGB임을 헷갈리지 않고 파악하면 쉽게 풀 수 있다. |

#9251 - LCS

|  |  |
| --- | --- |
| *import* sys  a = list(map(str, sys.stdin.readline().strip()))  b = list(map(str, sys.stdin.readline().strip()))  length\_a, length\_b = len(a), len(b)  dp = [[0]\*(length\_a+1) *for* \_ in range(length\_b+1)]  answer = 0  *for* i in range(1, length\_b+1):      now = b[i-1]  *for* j in range(1, length\_a+1):          dp[i][j] = dp[i-1][j-1]+1 *if* now == a[j-1] *else* max(dp[i][j-1], dp[i-1][j])          answer = max(answer, dp[i][j])  *print*(answer) | 주의해야 할 사항들이 존재한다.  우선 두 문자열 a,b의 길이가 다를 수도 있다는 점을 잊어서는 안된다.  그리고 나는 일치하는 ‘최장 증가 수열’이라고 할 때 두 문자열 사이의 같은 문자가 존재할 때 만약 a의 4번째와 b의 2번째 문자가 같을 때 2번째 이전의 dp값만 고려해야 한다고 생각 해서 1차원 리스트를 이용했는데 그게 아니라 2차원 리스트를 이용해서 해결했어야 했던 문제였다.  결국에 문제를 풀기 위해 이용한 점화식은  dp[i][j]는 1번 리스트의 i번째, 2번 리스트의 2번째 문자를 대응점에 둘 때 제일 긴 공통 순열이다. **만약 같은 문자가 존재하면 dp[i][j] = dp[i-1][j-1]+1이고 없다면 dp[i][j] = max(dp[i-1][j], dp[i][j-1])이다.** |

#1003 – 피보나치 수열

|  |  |
| --- | --- |
| *import* sys  t = int(sys.stdin.readline())  dp = [[0]\*2 *for* \_ in range(41)]  dp[0][0], dp[0][1], dp[1][0], dp[1][1] = 1,0,0,1  *for* i in range(2,41):      dp[i][0], dp[i][1] = dp[i-1][0]+dp[i-2][0], dp[i-1][1]+dp[i-2][1]  *for* \_ in range(t):      n = int(sys.stdin.readline())  *print*(dp[n][0],dp[n][1]) | 문제가 뭘 원하는지만 파악하면 너무나 쉬운 문제였다.  피보나치 수열의 기본이 재귀 함수를 이용하는 것은 맞지만 이 문제 또한 매번 재귀를 이용하면 시간 초과가 발생할 수 밖에 없다.  따라서 어차피 n이 40이하의 수로 한정이 되어 있기 때문에 dp의 길이를 처음부터 41로 정한 뒤 0과 1의 개수를 미리 채워서 필요한 tc마다 출력해 주면 된다. |

#1915 – 가장 큰 정사각형

|  |  |
| --- | --- |
| *import* sys  n,m = map(int, sys.stdin.readline().split())  board = [list(map(int, sys.stdin.readline().strip())) *for* \_ in range(n)]  dp = [[0]\*m *for* \_ in range(n)]  *for* i in range(m):      dp[0][i] = board[0][i]  *for* i in range(n):      dp[i][0] = board[i][0]  *for* i in range(1,n):  *for* j in range(1,m):  *if* board[i][j] == 1:              a,b,c = dp[i-1][j],dp[i-1][j-1], dp[i][j-1]              now = min(a\*\*0.5,b\*\*0.5,c\*\*0.5)              dp[i][j] = (now+1)\*\*2  ans = 0  *for* i in range(n):  *for* j in range(m):          ans = max(ans,int(dp[i][j]))  *print*(ans) | 이 문제는 dp[i][j]를 (i,j)번째 칸을 정사각형의 오른쪽 끝 점으로 쳐서 만들 수 있는 가장 큰 정사각형의 크기를 dp리스트에 저장하도록 하는 방법으로 풀면 해결이 가능하다.  잠깐 고민 했었으나 생각해 보니 dp[i][j]가 1이라면 dp[i-1][j], dp[i][j-1], dp[i-1][j-1]에 기록된 최대 정사각형의 크기중에 제일 작은 값에 제곱근을 씌워 1을 더해 제곱을 해서 저장하면 되는 것이었다.  따라서 결국 마지막에 dp에 저장된 값 중 제일 큰 값을 출력하도록 하였다.  이 문제는 특히 직접 재귀적으로 퍼져나가도록 사각형을 그려보는 과정에서 해결 방법을 찾을 수 있었다. |

#2098 – 외판원 순회

외판원 순회 문제는 영어로 Traveling Salesman problem (TSP) 라고 불리는 문제로 computer science 분야에서 가장 중요하게 취급되는 문제 중 하나이다. 여러 가지 변종 문제가 있으나, 여기서는 가장 일반적인 형태의 문제를 살펴보자.

1번부터 N번까지 번호가 매겨져 있는 도시들이 있고, 도시들 사이에는 길이 있다. (길이 없을 수도 있다) 이제 한 외판원이 어느 한 도시에서 출발해 N개의 도시를 모두 거쳐 다시 원래의 도시로 돌아오는 순회 여행 경로를 계획하려고 한다. 단, 한 번 갔던 도시로는 다시 갈 수 없다. (맨 마지막에 여행을 출발했던 도시로 돌아오는 것은 예외) 이런 여행 경로는 여러 가지가 있을 수 있는데, 가장 적은 비용을 들이는 여행 계획을 세우고자 한다.

각 도시간에 이동하는데 드는 비용은 행렬 W[i][j]형태로 주어진다. W[i][j]는 도시 i에서 도시 j로 가기 위한 비용을 나타낸다. 비용은 대칭적이지 않다. 즉, W[i][j] 는 W[j][i]와 다를 수 있다. 모든 도시간의 비용은 양의 정수이다. W[i][i]는 항상 0이다. 경우에 따라서 도시 i에서 도시 j로 갈 수 없는 경우도 있으며 이럴 경우 W[i][j]=0이라고 하자.

N과 비용 행렬이 주어졌을 때, **가장 적은 비용을 들이는 외판원의 순회 여행 경로**를 구하는 프로그램을 작성하시오.

|  |
| --- |
| **-개념 1. 비트 마스킹** |
| 정의: 어떤 숫자에 ‘마스크’라고 불리는 비트 값을 비트 연산의 숫자의 특정 비트 구간 값을 변형시키거나 그 값을 구현  사용: 실제로 업계와 실무, 특히 컴퓨터 네트워크 부문에서 많이 쓰이는 방법이라고 한다. 그런데 일단 지금 내 수준에서는 알고리즘 문제를 풀 떄에 있어서 탐색, dfs, bfs등의 문제를 풀 때 방문 여부를 0/1, 혹은 on/off로 표기한다고 할 때 유용하게 쓰일 수 있다. |
| ‘**<<**’ 연산: 왼쪽 연산자의 비트를 오른쪽 연산자수만큼 왼쪽으로 당기는 것  ‘**(1 << n)**’ 연산: n번째 비트를 키는 연산. 순수한 정수 1은 비트 연산에서는 0번쨰 비트가 켜졌음을 의미한다. 따라서 켜진 0번째 비트를 왼쪽으로 n만큼 옮기면 n번쨰 비트가 켜진다.  ‘**(1 << n) -1**’ 연산: n개의 비트를 모두 켠다.  **‘visited & (1 << n)**’ 연산: n번째 비트가 켜지지 않은 상태이면 True아니면 False  ‘**visited | (1 << n)**’ 연산: n번쨰 비트를 켜준다. |

|  |
| --- |
| -**개념 2. TSP Algorithm** |
| 도시의 개수와 각 도시간의 거리들이 주어질 때 모든 도시를 방문하고 원래 도시로 돌아올 수 있는 최단 경로의 거리를 구하는 문제  해결 방법이 모두 비트 마스킹을 이용하는데, 완전 탐색과 dp 두가지 방법으로 구현이 가능하다.  그러나 완전 탐색을 이용하면 pypy3을 이용해도 시간초과가 발생한다. |

**Sol 1. 완전 탐색**

|  |  |
| --- | --- |
| *import* sys  *from* collections *import* deque  n = int(sys.stdin.readline())  board = [list(map(int, sys.stdin.readline().split())) *for* \_ in range(n)]  inf = float(*'inf'*)  visited\_all = (1<<n)-1  ans = inf  *def* find\_path(start, now, visited, temp\_dist):  *global* ans  *if* visited == visited\_all:          curr\_dist = board[now][start] or inf          ans = min(ans, curr\_dist+temp\_dist)  *return*  *for* i in range(n):  *if* visited & (1 << i) == 0 and board[now][i]!= 0:              find\_path(start, i, visited | (1<<i), temp\_dist+board[now][i])  *for* k in range(n):      find\_path(k,k,1<<k,0)  *print*(ans) | #완전 탐색 + 비트 마스킹 이용 |

**Sol 2. 동적 계획법**

|  |  |
| --- | --- |
| *import* sys  *from* collections *import* deque  n = int(sys.stdin.readline())  board = [list(map(int, sys.stdin.readline().split())) *for* \_ in range(n)]  inf = float(*'inf'*)  visited\_all = (1<<n)-1  *#동적 계획법을 위해 dp리스트를 초기화 한다.*  *#range(n)을 통해 도시의 개수 n에 대응하고 1<<n을 통해 방문한 도시 집합 v에 대응한다.*  *#dp에 비트 마스킹을 이용해서 'visited'라는 상태를 기준으로 저장하는 것이다.*  dp = [[0]\*(1<<n) *for* \_ in range(n)]  *def* find\_path(last, visited):  *#이미 전부 방문한 상태라면 board[last][0]이 0일 경우에는 inf를, 즉 무한을 출력해 갱신이 불가하도록*  *if* visited == visited\_all:  *return* board[last][0] or inf  *#해당 값이 0이 아니라면 이미 방문한 적 있다는 뜻이기 때문에 그냥 바로 출력(시간 복잡도 감소)*  *if* dp[last][visited] != 0:  *return* dp[last][visited]  *#temp의 최솟값을 갱신해 주기 위해서 무한으로 설정해주고 최소로 갱신된 값을 캐시,즉 dp에 저장해 준다.*      temp = inf  *for* i in range(n):  *if* visited & (1<<i) == 0 and board[last][i] != 0:              temp = min(temp, find\_path(i,visited | (1<<i))+board[last][i])      dp[last][visited] = temp  *return* temp  *#어차피 순회하는 경로이기 때문에 시작점이 어디던 간에 상관이 없다.*  *#이 부분이 완전 탐색과 다르고 효율성이 높을 수 있는 이유이다.*  *print*(find\_path(0,1<<0)) | #비트 마스킹 + 동적 계획법을 이용해서 해결한 방법이다.  #무조건 visit리스트를 따로 만들어서 매번 갱신하고 다시 초기 상태로 돌려 놓는 등의 방법을 사용하니까 항상 복잡했었는데 이번 기회에 비트 마스킹을 배워서 방문 추적이 필요한 다양한 문제에서 써먹을 수 있도록 해야겠다. |