#그리디 알고리즘

그리디 알고리즘은 현재 상황에서 지금 당장 좋은 것만 고르는 방법을 의미합니다.

일반적인 그리디 알고리즘은 문제를 풀기 위한 최소한의 아이디어를 떠올릴 수 있는 능력을 요구합니다.

그리디 해법은 그 정당성 분석이 중요합니다.

단순히 가장 좋아 보이는 것을 반복적으로 선택해도 최적의 해를 구할 수 있는지 검토합니다.

대표적인 문제:

1. 거스름돈 🡪 큰 화폐로 계속 나누기

2. 1이 될 때까지 🡪 빼고 나누기를 해서 가장 횟수를 줄이게끔 하는 것

3. 곱하기 혹은 더하기 🡪 0,1 더하기, 나머지 곱하기

4. 모험가 길드 🡪 정렬, 숫자 늘려가면서 같은 수인지 컨펌

#구현: 시뮬레이션과 완전탐색

구현이란, 머릿속에 있는 알고리즘을 소스코드로 바꾸는 과정이다

흔히 알고리즘 대회에서 구현 유형의 문제란 무엇을 의미할까요?

풀이를 떠올리는 것은 쉽지만 소스코드로 옮기기 어려운 문제를 지칭합니다.

구현 유형의 예시는 다음과 같습니다.

알고리즘은 간단한데 코드가 지나칠 만큼 길어지는 문제

실수 연산을 다루고, 특정 소수점 자리까지 출력해야 하는 문제

문자열을 특정한 기준에 따라서 끊어 처리해야 하는 문제

적절한 라이브러리를 찾아서 사용해야 하는 문제(콤비네이션, 퍼뮤테이션)

대표적인 문제

1. 매트릭스에서 동서남북으로 옮겨 가기 🡪 이동하는 기능을 만들어 준 뒤에 적용

2. 시각 🡪 표시되는 시간의 문자열화.

3. 왕실의 나이트 🡪1번 유형이랑 비슷하나, 이동하는 기능을 다르게 표현.

4. 문자열 재정렬

#그래프 탐색 알고리즘

탐색이란 많은 양의 데이터 중에서 원하는 데이터를 찾는 과정을 말합니다.

대표적인 그래프 탐색 알고리즘으로는 DFS와 BFS가 있습니다.

DFS/BFS는 코딩테스트에서 매우 자주 등장하는 유형이므로 반드시 숙지해야 합니다.

프로그래밍에서는 그래프, 트리 등의 자료 구조 안에서 탐색을 하는 문제를 자주 다룬다. 대표적인 탐색 알고리즘으로 DFS와BFS를 꼽을 수 있는데 이 두 알고리즘의 원리를 이해해야 코딩 테스트의 탐색 문제 유형을 풀 수 있다. 그런데 DFS와BFS를 제대로 이해하려면 기본 자료구조인 스택과 큐에 대한 이해가 전제되어야 하므로 사전 학습으로 스택과 큐, 재귀함수를 간단히 정리하고자 한다.

스택 자료구조

먼저 들어 온 데이터가 나중에 나가는 형식(선입후출)의 자료구조입니다.

입구와 출구가 동일한 형태로 스택을 시각화할 수 있습니다.

Append, pop을 활용. 리스트로 스택 활용.

Print(stack[::-1]) #최상단 원소부터 출력

print(stack) #최하단 원소부터 출력

큐 자료구조

먼저 들어온 데이터가 먼저 나가는 형식(선입선출)의 자료구조입니다.

큐는 입구와 출구가 모두 뚫려 있는 터널과 같은 형태로 시각화 할 수 있습니다.

From collections import deque

#큐를 구현을 위해 deque 라이브러리 사용

Queue = deque() 🡪 시간복잡도 때문에 리스트로 큐를 만드는 것보다는 deque를 쓰자.

Queue.popleft

Print(queue) # 먼저 들어온 순서대로 출력

Queue.reverse() # 역순으로 바꾸기

Print(queue) # 나중에 들어온 원소부터 출력

재귀함수

재귀함수란 자기 자신을 다시 호출하는 함수를 의미합니다.

DFS를 실질적으로 구현하고자 할때, 자주 사용하는 방법이므로 알아야함.

재귀함수의 종료 조건

재귀함수를 문제 풀이에서 사용할 때는 재귀 함수의 종료 조건을 반드시 명시해야합니다.

종료 조건을 제대로 명시하지 않으면 함수가 무한히 호출될 수 있습니다.

재귀함수 사용 예제

1. 팩토리얼

2. 최대공약수 계산(유클리드 호제법)

재귀함수를 잘 활용하면 복잡한 알고리즘을 간결하게 작성할 수 있습니다.

단, 오히려 다른 사람이 이해하기 어려운 형태의 코드가 될 수도 있으므로 신중하게 사용해야합니다.

모든 재귀함수는 반복문을 이용하여 동일한 기능을 구현할 수 있습니다.

재귀함수가 반복문보다 유리한 경우도 있고 불리한 경우도 있습니다.

컴퓨터가 함수를 연속적으로 호출하면 컴퓨터 메모리 내부의 스택 프레임에 쌓입니다.

그래서 스택을 사용해야 할 때 구현상 스택 라이브러리 대신에 재귀 함수를 이용하는 경우가 많습니다.

DFS(DEPTH-FIRST SEARCH)

DFS는 깊이 우선 탐색이라고도 부르며 그래프에서 깊은 부분을 우선적으로 탐색하는 알고리즘입니다. DFS는 스택 자료구조(혹은 재귀함수)를 이용하며, 구체적인 동작 과정은 다음과 같습니다.

1. 탐색 시작 노드를 스택에 삽입하고 방문 처리를 합니다.

2. 스택의 최상단 노드에 방문하지 않은 인접한 노드가 하나라도 있으면 그 노드를 스택에 넣고 방문 처리합니다. 방문하지 않은 인접 노드가 없으면 스택에서 최상단 노드를 꺼냅니다.

3. 더 이상 2번의 과정을 수행할 수 없을 때까지 반복합니다.

보통은 번호가 낮은 인접 노드부터 방문을 가정.

BFS

BFS는 너비우선 탐색이라고도 부르며, 그래프에서 가까운 노드부터 우선적으로 탐색하는 알고리즘입니다. BFS는 큐 자료구조를 이용하며, 구체적인 동작 과정은 다음과 같습니다.

1. 탐색 시작 노드를 큐에 삽입하고 방문 처리를 합니다.

2. 큐에서 노드를 꺼낸 뒤에 해당 노드의 인접 노드 중에서 방문하지 않은 노드를 모두 큐에 삽입하고 방문 처리합니다.

3. 더 이상 2번의 과정을 수행할 수 없을 때까지 반복합니다.

특정 조건에서 해당 최단 경로를 찾는데 적합한 BFS

대표 문제

음료수 얼려 먹기 DFS

1. 특정한 지점의 주변 상, 하, 좌, 우를 살펴본 뒤에 주변 지점 중에서 값이 ‘0’이면서 아직 방문하지 않은 지점이 있다면 해당 지점을 방문합니다.

2. 방문한 지점에서 다시 상, 하, 좌, 우를 살펴보면서 방문을 진행하는 과정을 반복하면, 연결된 모든 지점을 방문할 수 있습니다.

Def dfs(x,y):

If x<=-1 or x>=n or y<=-1 or y>=m:

Return False

If graph[x][y] ==0:

Graph[x][y] ==1

Dfs(x-1,y)

Dfs(x,y-1)

Dfs(x+1,y)

Dfs(x,y+1)

Return True

Return False

N,m = map(int, input().split())

Graph=[]

For I in range(n):

Graph.append(list(map(int,input())))

Result =0

For I in range(n):

For j in range(m):

If dfs(I,j) =True:

Result +=1

Print(result)

미로탈출

BFS는 시작 지점에서 가까운 노드부터 차례대로 그래프의 모든 노드를 탐색합니다.(최단 거리 탐색 유리, 맹목적인 탐색)

상,하,좌,우로 연결된 모든 노드로의 거리가 1로 동일합니다.

따라서 (1,1) 지점부터 BFS를 수행하여 모든 노드의 최단 거리값을 기록하면 해결할 수 있습니다. 예시로 다음과 같이 3\*3 크기의 미로가 있다고 가정합시다.

From collections import deque

N,m = map(int, input().split())

Graph=[]

For I in range(n):

Graph.append(list(map(int, input())))

Dx =[-1,1,0,0]

Dy=[0,0,-1,1]

Def bfs(x,y):

Queue = deque()

Queue.append((x,y))

# 큐가 빌때까지 반복하기

While queue:

X,y = queue.popleft()

#현재 위치에서 4가지 방향으로의 위치 확인

For I in range(4):

Nx = x+ dx[i]

Ny = y+dy[i]

#미로 찾기 공간을 벗어난 경우 무시

If nx <0 or nx>=n or ny <0 or ny>=m:

Continue

#벽인 경우 무시

If graph[nx][ny] ==0:

Continue

#해당 노드를 처음 방문하는 경우에만 최단 거리 기록

If graph[nx][ny] ==1:

Graph[nx][ny] = graph[x][y] + 1

Queue.append((nx,ny))

# 가장 오른쪽 아래까지의 최단 거리 반환

Return graph[n-1][m-1]