**이니로 스터디 3주차 보고서**

**팀명: I am Brute**



|  |  |  |  |
| --- | --- | --- | --- |
| **회차** | 3회차 | **날짜** | 2021/04/05(월) |
| **스터디 주제** | Chapter5, 6 코드 리뷰 및 발표 | | |
| **스터디 장소/시간** | 학술 정보관 / 18:00~19:00 | | |
| **스터디 내용** | 참여인원( 팀원 전원 )   Part05 미로 탈출문제 이해하기 동빈이는 NxM 크기의 직사각형 형태의 미로에 갇혀 있다. 미로에는 여러 마리의 괴물이 있어 이를 피해 탈출해야한다. 동빈이의 위치는 (1, 1)이고 미로의 출구는 (N,M)이다. 한 번에 한 칸씩 이동할 수 있다. 미로는 반드시 탈출할 수 있다. 이때 동빈이가 탈출하기 위해 움직여야 하는 최고 칸의 갯수를 구하라. 칸을 셀 때는 첫 번째 칸과 마지막 칸을 모두 포함하여 계산한다. 문제 접근 방법 BFS로 길을 탐색한다면, 트리의 깊이가 해당 문제의 답이 될 것이다. BFS의 깊이를 구하는 방법을 찾아보자  처음 시작할 때의 타일의 값은 1이다. 그리고 한 걸음 걸음 마다 해당 타일의 값을 1씩 더해주면 몇 자국에 해당위치를 갈 수 있는지 파악할 수 있다. 그리고 마지막 위치인 (N, M)의 타일 값을 return 해주면 된다. 접근 방법을 적용한 코드 In [15]:  **from** **collections** **import** deque  dx = [1, 0, -1, 0]  dy = [0, 1, 0, -1]  N, M = map(int, input().split())  graph = []  **for** \_ **in** range(N):  graph.append(list(map(int, input())))  **def** bfs(graph, x, y):  q = deque()  q.append((x, y))  **while** q:  pos = q.popleft()  **for** i **in** range(4):  next\_pos = (pos[0] + dx[i], pos[1] + dy[i])  **if** next\_pos[0] < 0 **or** next\_pos[0] >= N **or**\  next\_pos[1] < 0 **or** next\_pos[1] >= M:  **continue**  **if** graph[next\_pos[0]][next\_pos[1]] != 0 \  **and** graph[next\_pos[0]][next\_pos[1]] == 1:  graph[next\_pos[0]][next\_pos[1]] += graph[pos[0]][pos[1]]  q.append((next\_pos))  **if** next\_pos[0] == N-1 **and** next\_pos[1] == M-1:  **return** graph[N-1][M-1]  print(bfs(graph, 0, 0)) DFS approach  * deque를 stack으로만 바꾸면 된다!   In [27]:  dx = [1, 0, -1, 0]  dy = [0, 1, 0, -1]  **def** dfs(graph, x, y, step):  stack = []  stack.append((x, y))  **while** stack:  pos = stack.pop()  **for** i **in** range(4):  next\_pos = (pos[0] + dx[i], pos[1] + dy[i])  **if** next\_pos[0] < 0 **or** next\_pos[0] >= N **or** next\_pos[1] < 0 **or** next\_pos[1] >= M:  **continue**  **if** graph[next\_pos[0]][next\_pos[1]] == 1:  graph[next\_pos[0]][next\_pos[1]] += graph[pos[0]][pos[1]]  stack.append(next\_pos)  **if** next\_pos[0] == N-1 **and** next\_pos[1] == M-1:  **return** graph[N-1][M-1]  N, M = map(int, input().split())  graph = []  **for** \_ **in** range(N):  graph.append(list(map(int, input())))  print(dfs(graph, 0, 0, 1)) Part05 실전 문제: 음료수 얼려 먹기 NxM 크기의 얼음 틀이 있다. 구멍이 뚫려 있는 부분은 0, 칸막이가 존재하는 부분은 1로 표시된다. 구멍이 뚫려 있는 부분끼리 상하좌우로 붙어 있는 경우 서로 연결되어 있는 것으로 간주한다. 이때 얼음 틀의 모양이 주어졌을 때 생성되는 총 아이스크림의 개수를 구하는 프로그램을 작성하라. 문제 이해하기 해당 문제는 DFS 접근법으로 풀 수 있다. 먼저 주어진 모든 얼음판에 대해서 여기에 아이스크림을 만들 수 있는지 없는지 확인하면된다.  먼저 (0,0) 부터 음료수를 부으면 아이스크림을 몇 개 만들 수 있는지 확인 한다. 여기서 음료수를 붓는 행위가 DFS가 되는거다. DFS를 진행하면서 visit하는 node는 이제는 다른 칸막이처럼 1이 된다.  이렇게 음료수를 다 채우면 return 을 True로 주어 아이스크림 하나 완성 이라고 알려준다. 그리고 다음 비어있는 얼음칸을 찾아 for문을 돈다. 문제 접근 방법 DFS구현 배경 지식  DFS접근 방법을 적용한 코드  N, M = map(int, input().split())  graph = []  **for** \_ **in** range(N):  graph.append(list(map(int, input().split())))  **def** dfs(x, y):  **if** x < 0 **or** x >= N **or** y < 0 **or** y >= M:  **return** **False**  **if** graph[x][y] == 1:  **return** **False**  **else**:  graph[x][y] = 1  dfs(x-1, y); dfs(x+1, y); dfs(x, y-1); dfs(x, y+1);  **return** **True**  cnt = 0  **for** x **in** range(N):  **for** y **in** range(M):  **if** dfs(x, y) == **True**:  cnt += 1  print(cnt)  **정렬(Sorting)**  데이터를 특정 기준에 따라 순서대로 나열하는 것.   * 정렬 알고리즘으로 데이터 정렬하면 이진탐색(Binary Search)가능. * 이진탐색의 전처리 과정   **선택 정렬(Selection Sort)**  가장 작은 데이터를 선택해 맨 앞에 있는 데이터와 바꾸고, 그 다음 작은 데이터를 선택해 앞에서 두 번째 데이터와 바꾸는 과정 반복.  arr = [7,5,9,0,3,1,6,2,4,8]  for i in range(len(arr)):  min\_index = i # 가장 작은 원소 인덱스  for j in range(i+1, len(arr)):  if arr[min\_index] > arr[j]:  min\_index = j  arr[i], arr[min\_index] = arr[min\_index], arr[i] # swap   * 시간 복잡도 : O(N^2) * 데이터 10,000개 이상이면 정렬 속도 급격히 느려짐. * 비효율적이지만, 특정 리스트에서 가장 작은 데이터 찾는 일이 코테에서 잦으므로 해당 소스코드 형태 익숙해질 필요 있음   **삽입 정렬(Insertion Sort)**  데이터를 하나식 확인하며, 각 데이터를 적절한 위치에 삽입.   * 데이터가 거의 정렬 되어 있을 때 효율적. * 첫 번째 데이터는 그 자체로 정렬되어 있다고 판단하기 때문에 두 번째 데이터부터 시작.   arr = [7,5,9,0,3,1,6,2,4,8]  for i in range(1, len(arr)):  for j in range(i, 0, -1): # index i부터 1까지 감소하며 반복  if arr[j]< arr[j-1]: # 한 칸씩 왼쪽으로 이동  arr[j], arr[j-1] arr[j-1], arr[j]  else: # 자기보다 작은 데이터 만나면 그 위치에서 멈춤  break   * 시간 복잡도 : O(N^2)   + 최선의 경우 O(N)   + 거의 정렬되어 있는 상태라면 퀵정렬보다 강력.   **퀵 정렬(Quick sort)**  기준 데이터를 설정하고 그 기준 데이터보다 큰 데이터와 작은 데이터 위치를 바꿈   * 지금까지 배운 정렬 중 가장 많이 사용됨. 퀵 정렬만큼 빠른 정렬='병합 정렬(merge sort)' * 기준을 설정한 뒤 큰 수와 작은수를 교환하고 리스트를 반으로 나누는 방식으로 동작. * 피벗(pivot)이 사용됨. 큰 숫자와 작은 숫자 교환할 때 교환하기 위한 기준을 피벗이라 함. * 호어 분할 방식(Hoare Parition) : 리스트에서 첫 번째 데이터를 피벗으로 설정   arr = [7,5,9,0,3,1,6,2,4,8]  def quick\_sort(arr, start, end):  if start >= end: # 원소 1개인 경우 종료  return  pivot = start # 피벗은 첫 번째 원소  left = start + 1  right = end  while left <= right:  # 피벗보다 큰 데이터 찾을 때 까지 반복  while left <= end and arr[left] <>= arr[pivot]:  left += 1  # 피벗보다 작은 데이터 찾을 때 까지 반복  while right>start and arr[right] >= arr[pivot]:  right -= 1  if left > right: # 엇갈렸다면 작은 데이터와 피벗 교체  arr[right], arr[pivot] = arr[pivot], arr[right]  else: # 엇갈리지 않았다면 작은 데이터와 큰 데이터 교체  arr[left], arr[right] =arr[right], arr[left]  # 분할 이후 왼쪽 부분과 오른쪽 부분에서 각 정렬 수행  quick\_sort(arr, start, right-1)  quick\_sort(arr, right+1, end)  quick\_sort(arr, 0, len(arr)-1)  print(arr)  파이썬의 장점을 살린 퀵정렬 소스코드  arr = [7,5,9,0,3,1,6,2,4,8]  def quick\_sort(arr):  # 리스트가 하나 이하 원소만 담고 있다면 종료  if len(arr) <= 1:  return arr    pivot = arr[0] # 피벗은 첫 번째 원소  tail = arr[1:] # 피벗을 제외한 리스트  left\_side = [x for x in tail if x <= pivot]  # 분할된 왼쪽 부분  right\_sied = [x for x in tail if x > pivot]  # 분할된 오른쪽 부분  # 분할 이후 왼쪽부분, 오른쪽 부분 각각 정렬  수행하고 전체 리스트 반환  return quick\_sort(left\_side) + [pivot] + quick\_sort(right\_side)   * 시간 복잡도 : O(NlogN)   + 최악 : O(N^2)   + 이미 데이터가 정렬되어 있는 경우 ㄴ리가 동작.   + 따라서 피벗값 설정할 때 추가적 로직 더해줌.   + 파이썬 기본 정렬 라이브러리 O(NlogN)   **계수 정렬(Count Sort)**  특정 조건 부합할 때만 사용할 수 있지만 매우 빠른 정렬 알고리즘.   * 데이터 개수 N, 데이터 중 최댓값 K일 때 * **시간 복잡도 : O(N+K)** 최악일때도! * 다만 데이터의 크기 범위가 제한되어 정수 형태로 표현할 수 있을 때만 사용 가능. * 가장 큰 데이터와 가장 작은 데이터 차이가 1,000,000 넘지 않을 때 사용 * 비교 기반 정렬 방식(선택정렬, 삽입정렬, 퀵정렬) 아님. * 데이터의 크기가 제한되어 있을 때 데이터 개수 많아도 아주 빠르게 동작.  1. 리스트 인덱스가 모든 범위 포함할 수 있도록 리스트를 생성함 2. 데이터를 하나씩 확인하며 데이터 값과 동일한 인덱스 데이터를 1씩 증가시킴 3. 그 후 리스트 첫번째 데이터부터 그 값만큼 인덱스 출력   arr = [7,5,9,0,3,1,6,2,4,8]  # 모든 범위 포함하는 리스트 선언 & 0으로 초기화  count = [0] \* (max(arr)+1)  for i in range(len(arr)):  count [arr[i]] += 1 # 각 데이터에 해당하는 인덱스 값 증가  for i in range(len(count)): # 리스트에 기록도니 정보 확인  for j in range(count[i]):  print(i, end= ' ') #띄어쓰기 구분으로 등장한 횟수만큼 인덱스 출력   * 시간 복잡도 : O(N+K) * 현존하는 정렬 알고리즘 중 기수정렬(Radix Sort)과 더불어 가장 빠름. * 공간 복잡도 : O(N+K) 비효율적.   + 동일한 값을 가지는 데이터가 여러 개 등장할 때 효과적.   + 퀵 정렬 : 일반적인 경우에서 빠르게 동작.   **계수 정렬(count sort)은 데이터 크기 한정되어 있고, 데이터가 많이 중복되어 있을 수록 유리함**  **파이썬의 정렬 라이브러리**  sorted() : 퀵 정렬과 동작 방식이 비슷한  **병합 정렬(merge sort)** 기반으로 만들어짐.   * 시간 복잡도 : O(NlogN)   sort() : 리스트 객체 내장 함수. 내부 원소 바로 정렬됨 | | |
| **기타** | 작성자 : 김준호 | | |