멘사 알고리즘 스터디 최종 보고서

202155650 윤소현

1. 학습 개요:

1.1 학습 배경:

멘사 알고리즘 스터디는 정보컴퓨터공학부 학생 7명이 모여 일주일 동안 공부한 알고리즘에 대해 얘기하고, 문제 풀이에 대해 한 명씩 발표하며 알고리즘 문제 해결 능력을 향상시키는 스터디이다.

1.2 기존 문제점:

스터디를 만들기 전에는 혼자 공부하기에 동기가 부족하고, 공부를 계속해서 미룰 수 있다는 단점이 있었다. 이를 해결하기 위해 멘사 알고리즘 스터디를 만들어 매주 의무적으로 모여 공부한 내용을 나눔으로써 한 학기 동안 의지를 잃지 않고 알고리즘 공부를 할 수 있었다.

2. 학습 환경:

파이썬과 C++을 이용해서 알고리즘 문제를 풀고, 깃허브로 코드를 공유했다.

3. 주 차별 학습 내용:

1주 차) 복잡도

알고리즘 공부를 본격적으로 시작하기 전에, 알고리즘의 효율성을 평가하는 복잡도가 무엇인지 알아야 했다. 복잡도(complexity)란, 알고리즘의 성능을 나타내는 척도로 시간 복잡도(time complexity)와 공간 복잡도(space complexity)로 나눌 수 있다. 시간 복잡도는 특정한 크기의 입력에 대해 얼마의 알고리즘 수행 시간 걸리는지를 의미하고, 공간 복잡도는 특정한 크기의 입력에 대하여 알고리즘이 얼마나 많은 메모리 공간을 차지하는지를 의미한다. 시간 복잡도는 빅오 표기법을 사용한다. 빅오 표기법은 상수 계수나 낮은 차수 항은

생략하고 가장 높은 차수 항으로 표현한다. 예를 들어 T(n) = 3n + 2는 O(n)으로 표현된다. 공간 복잡도도 시간 복잡도와 마찬가지로 빅오 표기법을 사용한다.

```
array = [1, 2, 3]
sum = 0
for i in array:
   sum += i
print(sum)
```

위 코드에서 프로그램의 시간을 결정하는 것은 array 리스트의 크기(n)이다. 따라서 시간 복잡도는 O(n)이 된다.

```
array = [1, 2, 3]
for i in range(len(array)):
   for j in range(len(array)):
       print(f"Pair: ({array[i]}, {array[j]})")
```

이번엔 array 리스트 안의 원소로 만들 수 있는 2개의 숫자 쌍들을 모두 출력하는 코드를 살펴보자. 첫 번째 for 루프에서도 array 리스트 크기 n만큼 반복하고, 두 번째 for 루프에서도 n만큼 반복된다. 따라서 시간 복잡도는 O(n^2)이 된다.

1주 차) Union-Find 알고리즘

[1주 차 문제 1717 - 집합의 표현]

이 문제는 0, a, b 또는 1, a, b로 입력을 받는다. 첫 번째 입력으로 들어오는 숫자가 0인 경우에는 합집합(Union) 연산을, 1인 경우에는 a와 b 두 원소가 같은 집합에 포함되어 있는지 확인(Find)하는 연산을 한다. 이 문제를 해결하기 위해 Union-Find 알고리즘을 사용했다.

Union-Find 알고리즘은 이름 그대로 집합의 합집합(Union)과 Find 연산을 빠르게 처리하는 데 사용된다. Find 연산은 루트 노드에 도달할 때까지 parent 배열을 재귀적으로 호출하여 해당 원소가 속한 집합의 루트 노드를 반환한다. Union 연산은 말 그대로 두 원소가 속한 집합을 합친다. 각 집합은 트리로 표현된다.

Union-find 알고리즘에서 Tree의 깊이가 깊어질 경우에, O(N)의 시간복잡도를 가지게 된다. 이를 해결하여 Union-Find 알고리즘의 성능을 향상시키기 위해 경로 압축 기법을 사용할 수 있다. 경로 압축 기법은 Find 연산에서 트리의 깊이를 최소화하기 위해, Find 연산을 수행할 때 각 원소가 루트 노드를 가리키도록 parent를

업데이트하는 것이다. 즉, find한 값을 배열에 저장하고 값을 반환해 주는 것이다. 이를 통해 다음 find 연산에서 빠르게 루트 노드를 찾을 수 있게 된다. 약간의 코드 수정으로 시간 복잡도를 O(logN)로 줄일 수 있게 된다.

2-3, 5-6주 차) 그리디 알고리즘

그리디 알고리즘이란, 현재 단계에서 가장 최적이라고 판단되는 선택을 반복적으로 수행하여 해답을 구하는 알고리즘이다. 현재 단계에서 가장 최적인 선택을 하기 때문에 현재의 선택이 나중에 어떤 영향을 미칠지는 고려하지 않는다. 따라서 최종으로 구해진 해답이 항상 최적의 해가 된다고 보장할 수는 없으며, 그리디 선택 속성과 최적 부분 구조와 같은 문제에서만 최적해를 보장한다. 그리디 알고리즘은 최소 스패닝 트리(MST)를 찾는 문제(크루스칼, 프림 알고리즘) 또는 최단 경로 구하기(다익스트라 알고리즘) 문제 등에서 사용된다. 2주 차에 1946, 3주 차에 2170, 5주 차에 11501, 6주 차에 11000번을 풀었다.

[2주 차 문제 1946 - 신입사원]

이 문제는 다른 모든 지원자와 비교했을 때 서류 심사 성적과 면접 시험 성적 중 적어도 하나가 다른 지원자보다 떨어지지 않는 사람만 채용한다는 조건 하에서 신규 사원 채용에서 선발할 수 있는 최대 인원의 수를 구하는 문제이다. 이 문제를 그리디 알고리즘으로 해결할 수 있는 이유는, 서류 성적을 기준으로 오름차순으로 정렬하여 면접 성적을 기준으로 이전에 합격한 사람의 면접 순위보다 높은 순위의 사람만 합격자로 추가해 주면 되기 때문이다.

[3주 차 문제 2170 - 선 긋기]

이 문제는 자의 한 점에서 다른 한 점까지 선을 긋는데, 이미 선이 있는 위치에 겹쳐서 그릴 경우에는 차이를 구별하지 않고 여러 선을 그었을 때, 그려진 선들의 총 길이를 구하는 문제이다. 선의 시작점을 기준으로 먼저 정렬하고, 선들이 겹칠 때는 끝점이 확장된다면 이 값으로 갱신한다. 만약 겹치지 않는다면 이전 선의 길이를 더하고 새로운 선을 기준선으로 갱신하는 식으로 그리디 전략을 사용하여 문제를 풀 수 있다.

[5주 차 문제 11501 - 주식]

이 문제는 날 별로 주식의 가격을 알려주었을 때, 최대 이익이 얼마가 되는지 계산하는 문제이다. 주식 가격을 내림차순으로 정렬하고 현재 상태의 주식 가격 리스트에서 가장 큰 주식의 가격을 찾아 해당 주식을 팔아 최대 이익을 얻는다. 따라서 그리디 알고리즘을 적용하여 문제를 해결할 수 있었다.

[6주 차 문제 11000 - 강의실 배정]

이 문제는 주어진 N개의 수업을 위해 최소 강의실을 사용하기 위해 강의실을 배정하는 문제이다. 이 문제에서는 가장 빨리 종료되는 강의실을 먼저 사용하는 그리디 전략을 통해 현재 강의의 시작 시간이 가장 빨리 끝나는 강의실의 종료 시간보다 늦다면 그 강의실을 재사용할 수 있도록 한다. 현재 강의의 시작 시간이 가장 빨리 끝나는 강의실의 종료 시간보다 빠르면 새로운 강의실을 배정받아야 하므로 해당 강의의 종료 시간을 최소 힙에 추가했다. 최종적으로 최소 힙의 길이가 사용된 강의실의 수가 된다. 여기서 최소힙을 사용한 이유는 가장 빨리 끝나는 강의실을 쉽게 찾을 수 있게 하기 위해서이다.

[후기]

4주에 걸쳐 그리디 알고리즘 문제를 풀어보면서, 그리디 알고리즘으로 풀 수 있는 유형을 빠르게 식별하는 방법을 터득할 수 있었다. 문제에서 구해야 하는 답이 단계별로 최적의 선택을 해야하는 경우에는 먼저 그리디 알고리즘을 떠올려 보는 것이 중요하다는 것을 느꼈고, 앞으로 알고리즘 문제를 풀 때에는 문제를 빠르게 분석하여 어떤 알고리즘을 적용하면 좋을지 판단하는 능력을 더 키워야겠다는 생각이 들었다.

4, 8주 차) Dynamic Programming

DP(= 다이나믹 프로그래밍)는 기본적으로 하나의 큰 문제를 여러 개의 작은 문제로 나누어 그 결과를 저장하여 다시 큰 문제를 해결할 때 사용한다. 일반적으로 DP는 재귀와 유사하지만, 재귀를 사용할 때 동일한 작은 문제들이 여러번 반복되어 비효율적인 계산이 되는 것을 막아준다. 피보나치 수열을 예로 들 수 있는데, 피보나치 수열을 구할 때 f(n) = f(n-1) + f(n-2)로 호출하면 동일한 값을 2번씩 구하게 되므로 연산량이 매우 늘어난다. 이 때 DP를 사용하여 한 번 구한 작은 문제의 결과 값들을 저장해 두고 재사용할 수 있어 계산된 값을 다시 계산할 필요가 없어진다. DP로 문제에서는 점화식을 세우는 것이 핵심이다. 4주 차에는 백준 떡 먹는 호랑이 문제(2502)를 풀었다. 8주 차에는 백준 평범한 배낭 문제(12865)를 풀었다.

[4주 차 - 2502 떡 먹는 호랑이]

문제에서는 하루에 한 번 산을 넘어가는 떡 장사 할머니가 호랑이에게 어제 받은 떡의 개수와 그저께 받은 떡의 개수를 더한 만큼의 떡을 줘야만 산을 무사히 넘어갈 수 있다. 할머니가 넘어온 날(D)과 그날 호랑이에게 준 떡의 개수(K)가 주어졌을 때, 첫날에 준 떡의 개수와 둘째 날에 준 떡의 개수를 구하는 문제이다.

```
for i in range(1, D) :
   coeff.append([coeff[i - 1][0] + coeff[i][0], coeff[i - 1][1] + coeff[i][1]])
```

위와 같은 점화식을 사용해 coeff[i][0]과 coeff[i][1] 각각에 i 번째 날에서 준 떡의 값을 첫 번째 날과 두 번째 날에 준 떡의 개수의 계수를 저장하도록 한다.

[8주 차 - 12865 평범한 배낭 문제]

이 문제에는 물건의 개수(N)와 배낭의 최대 무게(K), 각 물건의 무게와 가치가 주어진다. 이 때, 배낭에 넣을 수 있는 물건들의 최대 가치의 합을 구해야 한다. 다이나믹 프로그래밍으로 풀 수 있는 대표적인 예제인 배낭 문제를 연습하기 위해 이 문제를 선택해 보았다.

```
for w, v in score.items():
       if input_W + w <= K and input_V + v > score.get(input_W + w, 0):
           tmp[input_W + w] = input_V + v
   score.update(tmp)
```

배낭 문제에서 사용되는 점화식은 위와 같다. 새로운 물건의 무게 input_W와 가치 input_V를 선택하여 기존 상태 (w, v)에 추가한다. 이때, 새로운 무게가 배낭의 최대 무게 K를 초과하지 않고, 해당 무게에서 가치가 기존 가치보다 더 크면 상태를 갱신한다. 이러한 과정을 통해 모든 가능한 경우를 탐색하여 최종적으로 배낭에 넣을 수 있는 물건들의 최대 가치를 출력하도록 할 수 있다.

[후기]

다이나믹 프로그래밍 문제를 해결하기 위해서는 큰 문제를 어떻게 작은 하위 문제로 나눌 것인지, 어떤 값을 dp 테이블에 저장할 것인지, 점화식은 어떻게 도출해야 할 것인지 결정하는 것이 핵심이다. 이 과정에 익숙해지기 위해서는 조금 더 여러 문제를 풀어 보면서 반복적인 연습이 필요하다는 것을 느꼈다.

7주 차) 이분 탐색 / 매개변수 탐색

이진 탐색(이분 탐색) 알고리즘은 정렬되어 있는 리스트에서 탐색 범위를 절반씩 좁혀가며 데이터를 탐색하는 방법이다. 반드시 리스트의 내부 데이터가 정렬되어 있어야 사용할 수 있다. 이분 탐색은 start, end, mid 3개의 변수를 사용하여 탐색을 진행한다. 정렬된 리스트에서 특정 값을 찾을 때 정중앙에 위치한 값을 활용하여 빠른 속도로 탐색을 끝낸다.

매개변수 탐색은 이진 탐색과 상당히 유사한 알고리즘으로, 답이 이미 결정되어 있다고 보고 문제를 푸는 “결정 문제”로 풀 수 있는 문제에서 사용할 수 있다. 추가로, 어떤 시점까지는 조건을 만족하지만, 그 시점 이후로는 조건을 만족하지 않는 경우에 최댓값을 찾거나 최소값을 찾는 문제에도 사용된다.

7주 차에서는 백준 2343번 기타 레슨 문제를 풀었다.

[7주 차 문제 - 2343 기타 레슨]

이 문제에서는 기타 강의의 길이가 강의 순서대로 분 단위로 주어질 때, 녹화 가능한 블루레이의 크기 중 최소를 출력해야 한다.

```
if blu_ray_counting <= blu_ray:
    end = mid - 1
else:
    start = mid + 1
```

이분 탐색을 진행하고 있는 부분이다.

[후기]

이 문제는 풀이에 사용하고 있는 이분 탐색뿐만 아니라, 매개변수 탐색으로도 풀 수 있다고 한다. 이분 탐색 문제에는 이미 익숙해져 있는 상태였지만, 매개변수 탐색은 처음 들어보는 알고리즘 유형이었다. 이 문제를 풀어봄으로써 이분 탐색 알고리즘을 복습하고, 매개변수 탐색이라는 새로운 알고리즘을 학습할 수 있었다.

9, 10주 차) BFS / DFS

BFS는 너비 우선 탐색 알고리즘으로, 큐를 이용하여 구현할 수 있다. BFS는 시작 노드에서부터 인접한 노드를 모두 탐색한 후, 다음 노드로 이동한다. BFS는 두 노드 사이의 최단 경로 혹은 임의의 경로를 찾고 싶을 때 이 방법을 선택한다.

DFS는 깊이 우선 탐색 알고리즘으로, 스택을 이용하여 구현한다. DFS는 탐색 시작 노드를 스택에 삽입하고, 방문 처리한다. 스택의 최상단 노드에 방문하지 않은 인접한 노드가 하나라도 있으면 그 노드를 스택에 넣고 방문 처리한다. 방문하지 않은 인접 노드가 없으면 스택 최상단 노드를 꺼낸다. 더이상 이 과정을 수행할 수 없을 때까지 반복하여 그래프를 탐색한다. DFS는 트리 순회 등의 문제를 해결하기 위해 사용한다.

9주 차에선 백준 2606, 10주 차에선 2178번 문제를 풀었다.

[9주 차 - 2606 바이러스]

이 문제는 여러 대의 컴퓨터가 네트워크 상으로 서로 연결되어 있을 때, 1번 컴퓨터가 바이러스에 걸릴 경우 해당 컴퓨터에 의해 바이러스에 걸리게 되는 컴퓨터의 수를 구하는 문제이다. 1번 컴퓨터에서 부터 시작하여 연결된 모든 컴퓨터의 수를 출력함으로써 풀 수 있는 문제라고 생각이 들었고 깊이 우선 탐색 알고리즘을 사용하여 쉽게 풀 수 있었다.

[10주 차 - 2178 미로 탐색]

이 문제는 (N, M)과 미로가 입력될 때, (1, 1) 에서 미로를 통과하여 (N, M)으로 가기 위해 지나야 하는 최소 칸의 수를 구해야 하는 문제이다. 가중치가 없는 그래프에서 최단 경로를 구하는 경우에는 너비 우선 탐색을 사용하여 해결할 수 있다는 것을 알게되었고, BFS를 사용해서 미로를 통과하는 알고리즘을 구현하여 문제를 풀 수 있었다.

[후기]

깊이 우선 탐색과 너비 우선 탐색 문제를 풀어보면서 두 알고리즘의 차이를 명확하게 알 수 있었고, 어떤 문제 상황에서 어떤 알고리즘을 선택하여 풀면 좋은지 학습할 수 있었다. 깊이 우선 탐색의 경우 모든 경로를 탐색하게 되므로 경로가 많아지면 공간과 시간이 급증할 수 있기 때문에 종료 조건을 적절히 설정하거나, 입력의 크기를 유의해야 할 것 같다. 너비 우선 탐색도 큐에 노드를 넣고 꺼내는 방식으로 동작하게 되므로, 모든 경로를 탐색할 필요가 없는 문제에서 불필요한 탐색을 줄이기 위해 유의해야겠다는 생각이 들었다.

11주 차) 다익스트라

다익스트라 알고리즘은 대표적인 최단 경로 탐색 알고리즘이다. 다익스트라 알고리즘을 사용하여 최단 경로를 탐색할 때에는 음의 가중치를 가지는 간선을 포함할 수 없다. 모든 간선의 가중치가 음이 아닐 때, 한 정점에서 다른 모든 정점으로 가는 최단 경로를 알려준다. 다익스트라 알고리즘의 동작은 다음과 같다.

출발 정점과 도착 정점을 선택하고, 최단 거리를 저장할 배열을 만들어 초기화해 둔다. 이때, 배열의 값을 계속해서 최단 경로로 갱신할 것이기 때문에 초기값은 무한대로 설정해 두고, 출발 정점의 거리는 0으로 설정한다. 추가로 정점의 방문 여부를 구별할 배열도 하나 준비해 둔다.

그리고 아래와 같은 탐색 과정을 반복해야 한다.

현재까지 최단 거리 배열 중에서 방문하지 않은 정점들 중 최단 거리 값이 가장 작은 정점을 선택하여 방문 처리한다. 선택된 정점과 인접한 정점들에 대해 간선을 따라 이동했을 때의 거리(=현재 정점의 최단 거리 + 간선의 가중치)를 계산한다. 계산된 거리와 인접한 정점의 기존 최단 거리 값을 비교하여, 더 작은 값으로 최단 거리 배열을 갱신한다. 방문하지 않은 정점이 없거나, 도착 정점까지 최단 경로가 확보되면 반복을 종료한다. 이때 최단 거리 배열에 저장된 값들이 시작 정점에서 각 정점까지 도달할 수 있는 최단 거리이다.

11주 차에는 백준 1238 문제를 풀었다.

[11주 차 - 1238 파티] 이 문제에서는 N개의 숫자로 구분된 각각의 마을에 한 명의 학생들이 살고 있다. 총 M개의 단방향 도로가 있고, i 번째 길을 지나는데 T_i의 시간이 소비되는 각 마을이 있을 때, N명의 학생이 파티에 참석하기 위해 파티 장소 X 마을에 갔다가 다시 그들의 마을로 돌아가는데 가장 많은 시간을 소비하는 학생이 누구인지 구하는 문제이다. 따라서 각 학생이 살고있는 마을에서 출발하여 파티가 열리는 X 마을까지의 최단 경로를 구하기 위해 학생별로 다익스트라를 실행하고, 각 학생이 파티에 참석하고 돌아오는 시간을 추가로 계산했다.

```
def dijkstra(start):
    queue = []
    heapq.heappush(queue, (0, start))
    distance[start] = 0
    while queue:
        dist, now = heapq.heappop(queue)
        if distance[now] < dist:
            continue
        for next in graph[now]:
            cost = dist + next[1]
            if cost < distance[next[0]]:
                distance[next[0]] = cost
                heapq.heappush(queue, (cost, next[0]))
    return distance
```

다익스트라 알고리즘은 위와 같다.

```
for i in range(1, students + 1):
    distance = [INF] * (students + 1)
    result.append(dijkstra(i))
for i in range(1, students + 1):
    cost_lst.append(result[i][time] + result[time][i])
```

학생별로 다익스트라를 실행하고, 마을 X 까지 갔다가 다시 돌아오는 비용을 더하고 있다.

[후기]

학생의 수만큼 다익스트라를 실행하는 알고리즘으로 문제를 해결하긴 했으나, 학생의 수가 더 많아지면 해당 풀이로는 문제를 주어진 시간 안에 해결할 수 없을 것 같다는 생각이 든다. 조금 더 효율적으로 해결할 수 있는 방법을 추가적으로 생각해 보아야 할 필요성이 있음을 느꼈다. 다익스트라 알고리즘에 대해 이론적으로 알고 있었지만, 직접 구현해 보면서 다익스트라 알고리즘의 동작 방식을 더 잘 이해할 수 있게 되었다.

4. 알고리즘 스터디 활동 결과:

매주 조원들과 함께 한 주 동안 공부한 알고리즘에 대해 논의하고, 각자의 풀이법을 공유하며 학습했다. 이를 통해 여러 알고리즘에 대한 다양한 접근법을 익힐 수 있었다.

한 학기 동안 조원들과 공부 내용을 꾸준히 공유하고 함께 학습한 덕분에 지속적으로 알고리즘 공부를 이어갈 수 있었고, 백준에서 골드 5를 달성할 수 있었다.