PNU 스터디 그룹 10조 중간보고서

201925111 김건호

202155525 김문경

201924515 유승훈

201924429 김민혁

202055614 최성민

202155655 정진택

202155650 윤소현

0/1 배낭 문제로 알아보는 동적 계획법

정보컴퓨터공학부 201925111 김건호 PNU 스터디그룹 중간 보고서

서론

기존에 알고리즘 문제(PS)를 해결하면서, 시간 복잡도가 높은 경우에 대해 주로 `동적 계획법(Dynamic Programming)`을 사용해서 풀곤 했다. 하지만 `어떤 상태`에 대해 값을 `어떻게 저장해야 하는지` 설정하고, 어떻게 `점화식`을 세워야 할 지 어려움을 느꼈다. 이번 글에선 `언제 그리고 왜 동적 계획법을 쓰는가`에 대해 기술할 예정이다.

동적 계획법이란

`동적 계획법(Dynamic Programming, DP)`는 작은 하위 문제들을 통해, 복잡한 전체 문제를 해결하는 방식이다. 이를 위한 전제는, 문제의 최적 해결책이 하위 문제들의 최적 해결책으로 구성되어 있어야 한다. 이 명제는 전체 문제를 최적의 부분 문제로 나눌 수 있어야 함을 암시한다. 이는 상위 문제는 하위 문제에 의존함 또한 암시한다.

하위 문제를 통해 전체 문제를 해결

동적 계획법 : 하위 문제를 통해, 전체 문제를 해결하면 점이 무엇인가에 대해 생각해보자.

동적 계획법이 유리할 때는, `중복되는 계산 구조`가 있을 때 유리하다. 동적 계획법에서 해결한 부분 문제는 최적임을 의미하는데, `중복되는 계산 구조`를 발견한다면, 기존에 계산했던 값을 사용하면 되기 때문이다. 이는 해결한 부분 문제에 대해 정답을 `메모`했다가, 그대로 사용하는 방식과 유사하기 때문에 `메모이제이션(memoization)`이라고 부르기도 한다. `key`값은 현재 상태, `value`는 정답을 메모한다. 또한, 작은 문제에서부터 시작하기에, 문제의 크기가 줄어들고, 이는 `계산이 간단해짐`을 의미한다. 이렇게 찾은 최적의 부분 해들을 사용하여 `최적의 전체 해`를 얻을 수 있다.

동적 계획법(Dynamic Programming)을 사용해야 할 때

알고리즘 스터디에서 문제를 해결하며 느낀 바는 동적 계획법은 입력에 대해 지수 시간복잡도를 갖는 문제를, 입력에 대해 선형 시간복잡도로 변환할 수 있다는 것이다. 대표적인 문제,

`0/1 Knapsack problem(배낭 문제)`를 통해 알아보자.
[문제 링크 : 백준 12865(평범한 배낭)](https://www.acmicpc.net/problem/12865)
`0/1 배낭 문제`는 각 물건은 한 번만 선택할 수 있고, 주어진 물건들 중 일부를 선택하여 배낭에 넣을 때, 물건들의 무게 합이 배낭의 용량을 초과하지 않으면서 가치의 합을 최대화하는 문제이다. `물건을 n개, 가방이 버틸 수 있는 최고 무게를 w라고 하자.` 만약 문제를 `brute-force(완전 탐색)`으로 해결하려면 이는 입력에 대해 지수 시간복잡도를 갖는다. 왜냐하면 n개의 물건에 대해 전부 넣음과 넣지 않음을 계산하기 때문이다. 이는 O(2^n)으로 나타낼 수 있다.
<추가 설명>

a번째 물건을 n_a라고 하자. 첫 번째 물건에 대해 배낭에 넣거나 넣지 않거나 이다. 즉 2가지의 경우의 수를 갖는다. 2번째도 마찬가지다. Na번쨔 물건도 2가지 경우의 수를 갖는다. 이는 O(2^n)를 의미한다.
`이를 동적 계획법을 통해 해결해보자.` 본격적으로 문제 해결에 앞서, 현재 상태를 어떻게 정의해야 할까? 현재 상태를 잘 설정해야, 문제를 해결할 수 있다. 이 부분이 동적 계획법의 핵심이다. 그 다음으로, 설정한 상태를 토대로 전체로 도달할 수 있는 `점화식`을 세워야 한다.

탐욕법

>`탐욕법`은 눈 앞만 바라본다. 하지만 전체 조합을 고려해야 한다.

최고 값어치를 구하는 문제기에, 어떤 물건을 담았는지는 중요하지 않다. 현재 상태에서 하나 확정된 것은 현재 나의 배낭 무게이다. 또한, 현재 어떤 물건을 담았는지도 중요하다. 아래와 같은 예시를 살펴보자.
직관적으로 보았을 때, 4번 물건은 무조건 담아야 한다. 왜냐하면 무게 비율 가치가 제일 높기 때문이다. 그렇다면 이를
`탐욕적이게(greedy)`로 해결하면 될까? 정답은 `아니다`이다. 왜냐하면 아래 예시를 보자. `배낭의 용량을 9`이라고 하자.

물건	무게	가치	가치/무게 비율 (가치/무게)
1	3	4	1.33
2	4	5	1.25
3	5	7	1.4
4	2	3	1.5

`탐욕법`을 통해 문제를 해결하면 3번과 4번 물건을 넣을 것이다. 왜냐하면 가치/무게 비율이 제일 좋기 때문이다. 과연 이것이 정답일까? 즉, `10의 가치가 최대일까?` 정답은 `아니다.` 왜나하면, `1번 2번 4번` 물건을 넣는 것이 더 가치가 높기 때문이다. 즉, `탐욕적`으로 해결할 수 없다.

이러한 이유는, `탐욕법`은 각 물건을 부분적으로 담는 것에만 집중한다. 그러나, `0/1 배낭 문제`는 전체 물건을 한 번에 봐야 하는 조합적인 문제기 때문이다. 알고리즘이 지향하는 바가 완전히 다르기에 잘못된 접근이다.

0/1 배낭 문제 상태 설정

> `선택하지 않는 경우`와 `선택하는 경우`를 다 고려하며 상태를 설정해야 한다.

상태 설정을 해보자. 앞선 탐욕법의 실패에서 배낭 문제는 조합적인 문제임을 알았다. 일단 문제를 부분 문제로 나눠보자. 어떻게 나눌까? 일단 어떤 `물건를 보는가와 가방 무게`를 고려해야 한다. 하지만, `현재 어떤 물건`을 담았는지 상태를 다 기록하는 것은 결국 지수의 공간 복잡도를 가질 것이다. 왜냐하면 이는 경우의 수와 똑같은 문제기 때문이다.

배낭 문제에서 중요한 점은, 각 물건을 `선택하거나 선택하지 않는` 방식으로 해결할 수 있다는 것이다. 이때, `선택하지 않는 경우`와 `선택하는 경우`를 다 고려하며, `메모`를 잘 해보자. 현재 n번째 물건을 담았고, 가방의 무게를 Wn라고 하자. n+1번째 물건을 담는다면 가방의 무게는 Wn+Wn+1이다. 만약 담지 않는다면, 여전히 가방의 무게는 Wn일 것이다. 즉 물건을 순차적으로 고려하고, 다음 상태로 나아간다고 가정하면 물건의 번호와, 가방의 무게가 필요하다. 이를 `상태`로 정의해보자. 즉 메모를 기록할 자료 구조를 2차원 배열로 설정할 수 있게 된다. 배열 이름을 `dp`라고 하자.

```
dp[현재 몇 번째 물건을 고려하는지][현재 가방의 무게] = 최고 값어치
```

우리는 2차원 배열의 제일 끝을 보면 정답을 찾을 수 있게 되었다.

0/1 배낭 문제 점화식

> `선택하지 않는 경우`와 `선택하는 경우`를 다 고려하며 점화식을 세워보자. 조합적인 모든 것을 고려해야 한다. n번째 물건과 현재 가방의 무게를 w라고 하자. 다음 상태 즉, n+1번째 물건을 고려할 때, 넣거나, 넣지 않아도 된다. 즉 이를 점화식으로 써보자.

```
dp[n+1][w] = max(dp[n][w] , dp[n][w-(현재 보는 물건의 무게)] + (현
```

로 쓸 수 있다. dp[n][w]는 `n+1번째 물건을 넣지 않는다`이고, 후자는 `n+1번째 물건을 넣는 경우`이다. 이 때, `IndexOutOfBoundsException`을 조심하도록 해야 한다. 이제 이를 코드로 작성해보자.

소스 코드

`python3` 을 기준으로 작성하였다.

```python
import sys

    n,k=map(int,sys.stdin.readline().split()) #물건 수, 제한
    dp=[[0 for _ in range(k+1)] for _ in range(n+1)]
    answer=0
    for i in range(1,n+1) :
        w,v = map(int,sys.stdin.readline().split()) #weight,
        for j in range(1,k+1) :
            if j>=w :
                dp[i][j] = max(dp[i-1][j], dp[i-1][j-w] + v)
            else :
                dp[i][j] = dp[i-1][j]

    print(dp[n][k])
```

마무리하며

`동적 계획법`은 계산이 중복되는 구조에서 사용할 수 있다. 또한, `부분은 최적을 가져야 한다`는 원칙을 지켜야 한다. 또한 `지수 시간 복잡도`를 갖는 문제에서 `선형 시간 복잡도`로 변환할 수 있다. `0/1 배낭 문제` 같은 경우는 처음에 상태를 정의하는 것이 꽤나 까다로웠다. 특히, `현재 가방의 무게라는 것`이 처음엔 직관적으로 와닿지 않았다. `동적 계획법`문제의 알파이자 오메가는 `상태 정의와 저장하는 값`이라고 생각한다. 이를 잘못 정의하면, 잘못된 값들이 저장되어 전체 해결이 불가능해지기 때문이다. 또한, 계산된 값을 어떻게 저장하고 재사용할지 결정하는 것이 중요하다.

스터디 후기

알고리즘 스터디를 통해 여러 알고리즘 문제를 다루었다. 최장 부분 증가 수열(LIS), 최장 부분 공통 수열(LCS), 최단 경로 집중 공략, BFS, DFS 유형 분석, 투포인터 등등...

각자 공부해온 문제를 최대한 자세하게 설명하였다. 특히 필자는 `동적 계획법`을 주로 공부하였다.

동적 계획법을 통해 여러 복잡한 문제들을 효율적으로 해결할 수 있는 방법을 배웠고, 특히 문제의 **상태 정의**와 **중복 계산을 피하기 위한 값 저장**이 중요하다는 것을 깨달았다. 또한, 동적 계획법을 적용하는 문제들은 처음에는 어떻게 상태를 나누고, 어떤 방식으로 값을 저장할지 고민이 많았지만, 여러 문제를 풀어보며 점차적으로 직관이 생겼다.

스터디에서 다룬 `0/1 배낭 문제`와 같은 문제들은 상태 정의와 최적화 문제를 어떻게 분리하고, 중복 계산을 없애는지를 명확히 보여주는 좋은 예시였다. `동적 계획법`을 풀 때마다, 문제를 어

떻게 분할하고 최적의 해결 방법을 도출할지 고민하면서 점점 더 알고리즘적인 사고가 발전하는 것을 느꼈다.

알고리즘 스터디 보고서

202155525 김문경

1. [선수과목 – 백준 14567] : 위상정렬을 이용한 문제 해결

주어진 그래프(방향성 있는 비순환 그래프, DAG)에서 각 노드에 대해 위상 정렬 순서를 기반으로 한 값을 계산하는 프로그램입니다.
특히, 그래프의 위상 정렬 순서를 따라 노드가 가질 수 있는 최소 값을 결정하고 출력합니다.

[주요 코드 동작]

입력 처리

- 입력 값 n(노드 개수)과 m(간선 개수)을 읽습니다.
- 각 간선 정보를 읽어 node 배열에 간선을 저장하고, 각 노드의 진입 차수(degree)를 업데이트합니다.

위상 정렬 알고리즘

- **초기화**:
    - 모든 노드의 진입 차수를 확인하여, 진입 차수가 0인 노드를 큐에 삽입.
    - 진입 차수가 0인 노드는 시작 노드로 간주하며, 결과 배열 result에서 값을 1로 설정.
- **반복 처리**:
    - 큐에서 노드를 꺼내 해당 노드의 모든 인접 노드를 방문.
    - 방문 시 인접 노드의 진입 차수를 감소시키고, 진입 차수가 0이 되면 큐에 추가.
    - 새로운 노드의 result 값은 현재 노드의 result 값 + 1.

결과 출력

- 각 노드의 결과값(result[i])을 순서대로 출력

2. [작업 – 백준 2056] : 위상정렬 알고리즘

여러 작업이 주어지고, 각 작업은 소요 시간과 선행 작업을 가집니다. 특정 작업을 수행하기 위해서는 해당 작업의 모든 선행 작업이 완료되어야 하며, 전체 작업을 완료하는 데 걸리는 최소 시간을 구합니다.

[주요 코드 동작]

입력 처리

- 작업 수 n을 입력받고 각 작업의 정보를 저장합니다.
- 작업의 소요시간과 작업이 완료되기 위해 필요한 선행 작업 수, 작업 x를 선행 작업으로 요구하는 작업들의 리스트 데이터를 저장합니다.

위상 정렬 알고리즘

- **초기 작업 큐 설정**:
    - 선행 작업이 없는 작업(storage[i] == 0)을 큐에 추가하고 작업을 시작합니다.
- **BFS를 활용한 작업 수행**:
    - 큐에서 작업을 꺼내면서 해당 작업의 소요 시간을 확인합니다.
    - 해당 작업을 선행 작업으로 요구하는 작업들의 storage 값을 1씩 감소시키고, 각 작업의 **최소 시작 시간**(minCost)을 업데이트합니다.
    - 모든 선행 작업이 완료된 작업(storage[num] == 0)은 큐에 추가하고, 해당 작업의 **완료 시간**을 계산하여 큐에 삽입합니다.
- **최대 시간 계산**:
    - totalTime 변수는 현재까지 완료된 모든 작업 중 가장 오래 걸린 작업의 완료 시간을 저장하며, BFS 과정에서 갱신됩니다.

결과 출력

- BFS가 종료되면 totalTime을 출력합니다. 이는 모든 작업이 완료되는 데 필요한 최소 시간입니다.

[문제 풀이에서 헤멨던 부분]

시간을 total time (현재까지 실행된 시간) 과 nowTime (queue에서 꺼낸 작업의 진행시간) 개념으로 사용하고 있었는데, queue에 push 할 때 현재까지 일의 cost에 어떤 값을 더해서 저장 해야 할지 고민이 많았습니다.

결론 : 각 작업을 실행하기 위해 필요한 최대 시간을 minCost[num] 에 넣어서 저장했습니다.
minCost[num] = max(minCost[num], nowTime);
그리고 queue에 push 할 땐 minCost[num] + 현재 일의 cost 를 더해서 사용했습니다.

```
int n, totalTime=0;
int storage[10001]={0};
int minCost[10001]={0};
map<int,vector<int> > mmap;
map<int, int> cost;
queue<pair<int,int> > q;

void init(){
    cin>>n;
    for(int i=1; i<=n; i++){
        int cost_input,cnt;
        cin>>cost_input>>cnt;
        for(int j=0; j<cnt; j++){
            int x;
            cin>>x;
            mmap[x].push_back(i);
        }
        storage[i] = cnt;
        cost[i] = cost_input;
    }
}

int main() {
    init();
    int cnt = 0;
    for(int i=1; i<=n; i++){
        if(storage[i] == 0){
            q.push(make_pair(i, cost[i]));
            storage[i] = -1;
        }
    }
    while(!q.empty()){
        int number = q.front().first;
        int nowTime = q.front().second;
        q.pop();
        if(totalTime < nowTime){
            totalTime = nowTime;
        }
        for(int num : mmap[number]){
            storaalTime<<endl;
}
```

3. [1학년 – 백준 5557] : 동적 계획법 (DP)

숫자 배열이 주어지고, 첫 번째 숫자에서 시작하여 주어진 연산(더하기, 빼기)을 통해 목표 값(마지막 숫자)을 만드는 방법의 수를 구합니다. 단, 중간 연산 결과는 항상 0 이상 20 이하의 범위를 유지해야 합니다.

[주요 코드 동작]

입력 처리

- 숫자 배열 num을 입력 받습니다.
- 숫자 배열의 각 값을 배열 형태로 저장합니다

동적 계획법 (DP) 알고리즘

- **동적 계획법(DP) 테이블 정의:**
    - dp[i][j]: 배열의 i번째 숫자까지 연산했을 때, 값 j를 만들 수 있는 경우의 수.
    - dp[1][num[1]] = 1: 첫 번째 숫자는 초기 값으로 설정.
- **점화식:**
    - dp[i][j + num[i]] += dp[i-1][j] (더하기 연산 가능하면 업데이트).
    - dp[i][j - num[i]] += dp[i-1][j] (빼기 연산 가능하면 업데이트).
    - 단, 결과 값이 항상 0 이상 20 이하 범위 내에 있어야 함.
- **계산 과정:**
    - 첫 번째 숫자에서 시작해 순차적으로 배열의 끝까지 연산.
    - 마지막 숫자를 목표로 하는 경우의 수를 누적 계산.

결과 출력

- 마지막 숫자를 목표 값으로 만드는 경우의 수 dp[n-1][num[n]]를 출력합니다.

[알고리즘 – 동적 계획법(DP)]

이전 상태(`i-1`)를 기준으로 현재 상태(`i`)를 계산하며 가능한 모든 연산 결과를 업데이트 진행

-

- **[전화번호 목록 – 백준 5052] : 정렬**

여러 테스트 케이스에서 전화번호부 목록이 주어졌을 때, **어떤 번호가 다른 번호의 접두사(prefix)**인지 확인합니다. 만약 접두사가 존재하지 않으면 "YES"를, 존재하면 "NO"를 출력합니다.

[주요 코드 동작]

입력 처리

- 첫 번째 입력값 n (테스트 케이스의 개수)을 입력받습니다.
- 각 테스트 케이스에 대해 m개의 전화번호를 입력합니다.

문제풀이 (정렬) 알고리즘

- 전화번호 정렬:
    - 각 테스트 케이스의 전화번호를 사전 순 정렬.
    - 접두사를 확인하기 위해 작은 길이의 번호부터 비교가 용이하도록 정렬.
- 접두사 확인:
    - 정렬된 리스트에서 연속된 두 번호를 비교:
        - v[k]가 v[k+1]의 접두사인지 확인:
        - 조건: v[k] == v[k+1].substr(0, v[k].length()).
    - 접두사가 발견되면 check = false로 설정하고 종료.
- 결과 저장:
    - 접두사가 없으면 "YES", 있으면 "NO" 출력.

결과 출력

- 각 테스트 케이스마다 "YES" 또는 "NO" 출력.

[정렬 알고리즘]

- **정렬 기반 접근**:
    - 정렬을 통해 접두사 확인 시, 인접한 두 번호만 비교하면 충분.
- **효율성** (시간 복잡도)
    - 정렬: O(mlog m)O(m \log m)O(mlogm).
    - 비교: O(m·l)O(m \cdot l)O(m·l) (l은 전화번호의 평균 길이).

5. 스터디 후 느낀점

스터디를 진행하며 인상깊었던 문제들을 대표로 가져와 보고서를 작성했습니다. 알고리즘 스터디 인 만큼 풀고 싶은 알고리즘, 문제들을 가져와 발표하는 시간을 가졌으며 발표 후엔 질의응답을 통해 서로 모르는 것을 물어보고 아는 내용을 심화적으로 알아갈 수 있는 시간이 되었던 것 같습니다.
학교에서 배웠던 알고리즘들을 활용하기도 하고, 이전에 풀어봤었던 문제들을 다시 풀어 보기도 하며 전보다 더 성장해 나가는 제 모습을 볼 수 있는 기회였던 것 같습니다. 함께 스터디를 진행하는 팀원들의 질문과 적극적인 참여 덕분에 더 재미있게 알고리즘 공부를 할 수 있어 좋았습니다.

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 알고리즘이 얼마나 오래 걸리는지를 의미하고, 공간 복잡도는 특정한 크기의 입력에 대하여 알고리즘이 얼마나 많은 메모리를 차지하는지를 의미한다. 알고리즘 문제에는 시간 제한을 두어 해당 시간 안에 동작하는 프로그램을 작성 해야 정답으로 인정된다. 따라서 문제 해결 시에 시간 복잡도를 고려해야 한다. 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

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 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. 이를 통해 다음 find 연산에서 빠르게 루트 노드를 찾을 수 있게 된다. 백준 1717번 문제를 풀었다.

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.

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 문제를 풀었다. 8주 차에는 백준 평범한 배낭 문제를 풀었다.

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.

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 미로 찾기 등의 문제를 해결하기 위해 사용한다. 9주 차에선 백준 2606, 10주 차에선 2178번 문제를 풀었다.

알고리즘 스터디 보고서

201924515 유승훈

진행기간: 약 3개월

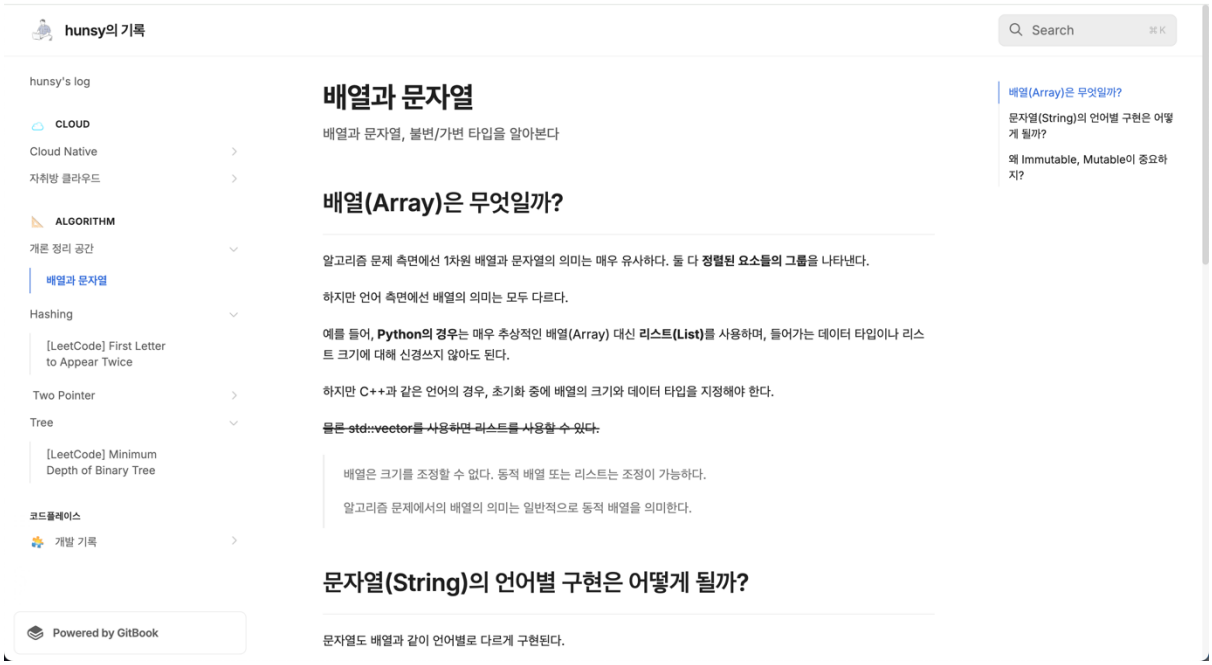
1. **보고서 개요**

진행기간 동안 프로그래머스와 백준, 그리고 LeetCode 문항들을 중점적으로 학습하였습니다.

프로그래머스와 백준에서는 단순 구현부터, 그래프 탐색 문제들을 위주로 학습하였으며, LeetCode 플랫폼을 통해선 알고리즘의 기초적인 자료구조나 방법론을 다루는 문제들 위주로 풀었습니다.

2. 학습 개요

먼저 알고리즘의 가장 근본이 되는 배열, 그리고 리스트에 대해서 정리하는 시간을 가졌습니다.



https://hunsy.seung.site/algorithm/undefined/undefined

배열과 리스트에 대한 차이를 이해하고 나서, 기초적인 자료구조나 방법론을 공부하였습니다.

크게 Hashing, Two pointer, Tree에 대해서 학습하였고, 각각 Leetcode에서 해당 자료구조나 방법론과 연계된 문제 3문항을 풀었습니다.

1. [Leetcode] First Letter to Appear Twice -> Hashing 관련
2. [Leetcode] Counting Elements -> Two Pointer 관련
3. [Leetcode] Minimum Depth of Binary Tree -> Tree 관련

3. 학습 상세

(1) [Leetcode] First Letter to Appear Twice

문제 설명

Given a string `s` consisting of lowercase English letters, return the first letter to appear twice.

참고 사항

A letter `a` appears twice before another letter `b` if the second occurrence of `a` is before the second occurrence of `b`.

문제를 보고 처음 한 생각

1. 문자열을 앞에서부터 뒤로 훑으면서 스택(파이썬 리스트)에 넣고, 스택의 head를 확인
2. 스택의 head가 처음으로 지금 탐색 중인 character와 같다면, 지금 탐색 중인 character를 바로 리턴 후 종료

위 두 가지 과정을 거쳐 해결하려고 하였습니다.

```
class Solution:
    def repeatedCharacter(self, s: str) -> str:
        stack = [s[0]]
        for i in range(1, len(s)):
            if stack[-1] == s[i]:
                return s[i]
            stack.append(s[i])
```

위 생각을 바탕으로 위와 같은 코드를 짰지만, 무조건 처음으로 붙어있는 문자를 찾는 것이 아니라, 붙어있지 않더라도, 탐색 중 처음으로 중복되는 문자열이 발견되면 return 하라는 문제인 것을 파악하게 되어

```
class Solution:
    def repeatedCharacter(self, s: str) -> str:
        dic = {}
        for i in range(len(s)):
            if s[i] in dic:
                return s[i]
            dic[s[i]] = -1
```

1. 딕셔너리에 Key: (탐색 중인 Character), Value에는 쓰레기 값을 넣어준다.
2. in 키워드를 이용하여 이미 키 값이 있는지 확인하고, 있다면 바로 키 값과 함께 바로 리턴한다

이 방식을 이용해서 개선된 풀이를 제출하여 통과하였습니다.

(2) [Leetcode] Counting Elements

문제 설명

Given an integer array `arr`, count how many elements `x` there are, such that `x + 1` is also in `arr`. If there are duplicates in `arr`, count them separately.

문제를 보고 처음 한 생각

```
def countElements(self, arr: List[int]) -> int:
    count = 0
    for x in arr:
        if x + 1 in arr:
            count += 1
    return count
```

처음 봤을 땐 단순히 in 메서드를 이용해서 풀면 바로 풀릴거라고 생각했습니다. 물론 이렇게 간단한 방법으로도 풀 수 있지만, in 메서드는 주어진 arr를 한번 순회하므로, arr의 length가 N이라고 했을 때 **시간 복잡도는 O(N^2)**가 걸리게 됩니다.

그래서 투포인터를 이용해서 시간복잡도를 개선하였습니다.

```
def countElements(self, arr: List[int]) -> int:
        answer = 0
        arr.sort()
        i=0 # slow pointer
        j=1 # fast pointer
        while j < len(arr):
            if arr[j] - arr[i] == 0:
                j += 1
            else:
                if arr[j]-arr[i] == 1:
                    answer += 1
                i+=1

        return answer
```

문제는 특이하게도 x에게 x+1이 있는지 찾는 문제이기 때문에, 두 개의 Slow, Fast 포인터를 두고

1. 두 포인터의 arr 값이 같으면 Fast Pointer + 1
2. **두 포인터의 arr 값의 차이가 1(우리가 찾는 값)이라면**, answer + 1, Slow Pointer +1
3. 두 포인터의 arr 값의 차이가 0도, 1도 아니라면, 단순히 Slow Pointer

이 방식으로 문제를 해결하면, x 와 x + 1이 arr안에 각각 어떤 위치에 존재하던지 간에, x+1을 가진 x의 개수를 셀 수 있었습니다.

(3) [Leetcode] Minimum depth of binary tree

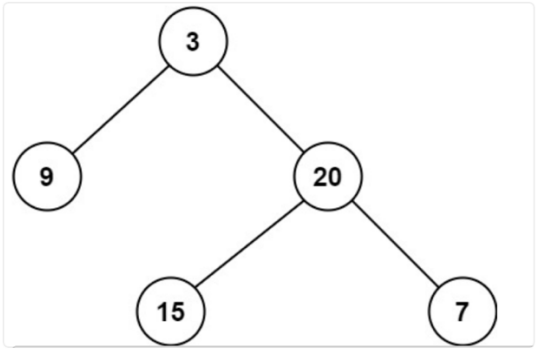
문제 설명

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

Note: A leaf is a node with no children

입력 예제:



example

```
Input: root = [3,9,20,null,null,15,7]
Output: 2
```

문제를 보고 처음 한 생각

binary tree 문제이고, DFS를 통해 Sub Tree의 Leaf Node까지의 cost를 계산하고, 이 중에서 min() 함수를 통해 최소인 cost를 선택해준다면 된다고 생각하였습니다.

```python
class Solution:

    def dfs(self, node: TreeNode) -> int:
        if node == None: # node가 없다면 return
            return 0

        if node.left == None: #만약 node의 left child가 없다면
            return 1 + self.dfs(node.right) # right child로 dfs

        elif node.right == None: #만약 node의 right child가 없다면
            return 1 + self.dfs(node.left) # left child로 dfs

        else:
            # 양쪽 다 있으면 dfs(leftNode), dfs(rightNode)중 작은 것 선택
            return 1 + min(self.dfs(node.left), self.dfs(node.right))

    def minDepth(self, root: Optional[TreeNode]) -> int:
        return self.dfs(root)
```

dfs 함수는 반환 타입으로 int를 가지며, node가 없는 경우를 제외한 경우에선 dfs(next_node)와 1을 더해서 return 하는 방식으로 경로의 누적 cost를 계산하여 이 문제를 통과하였습니다.

알고리즘 스터디 보고서

201924429 김민혁

ㄱ

1. 알고리즘 스터디 진행 방식

   매주 주말, 대면으로 스터디를 진행하였습니다. 각자 문제를 1-2개 준비해와서 각자의 풀이를 공유하고, 더 나은 풀이나 다른 풀이법이 있는 문제라면 그것까지 학습해와서 공유하는 방식으로 스터디를 진행했습니다. 문제의 난이도나 알고리즘의 종류에는 크게 제약이 없었지만, 다수의 참여자가 소수의 특정 알고리즘 문제를 준비하거나 너무 쉬운 문제를 지속적으로 풀어오는 것은 지양하는 것이 원칙이었습니다.

2. 스터디 준비 내용

   저는 주로 백준 온라인 저지(Baekjoon Online Judge)의 Gold 등급 문제들을 준비해왔습니다. 그래프 탐색, 그리디, DP, 구현, 수학 등 다양한 알고리즘의 문제들을 준비하려고 노력했습니다. 스터디를 진행하면서 실제로 백준 사이트의 문제 풀이 등급도 많이 상승했으며, 실질적인 알고리즘 풀이 실력이 많이 향상되었습니다.

   저는 특히 여러 가지 풀이가 있는 문제들을 준비하거나, 테마를 정해서 여러 문제를 준비해 가는 경우가 많았습니다. 가령 다음과 같은 테마로 문제를 준비했습니다.

   - 최장 부분 증가 수열(LIS)
   - 최장 부분 공통 수열(LCS)
   - 최단 경로 집중 공략
   - BFS, DFS 유형 분석

   …

   이렇게 주제별로 풀이를 학습했을 때, 스터디원들의 반응이 좋았습니다. 스스로도 해당 알고리즘에 대해 체계적으로 학습할 수 있었고, 학습 내용이 오래 기억되어 좋았습니다.

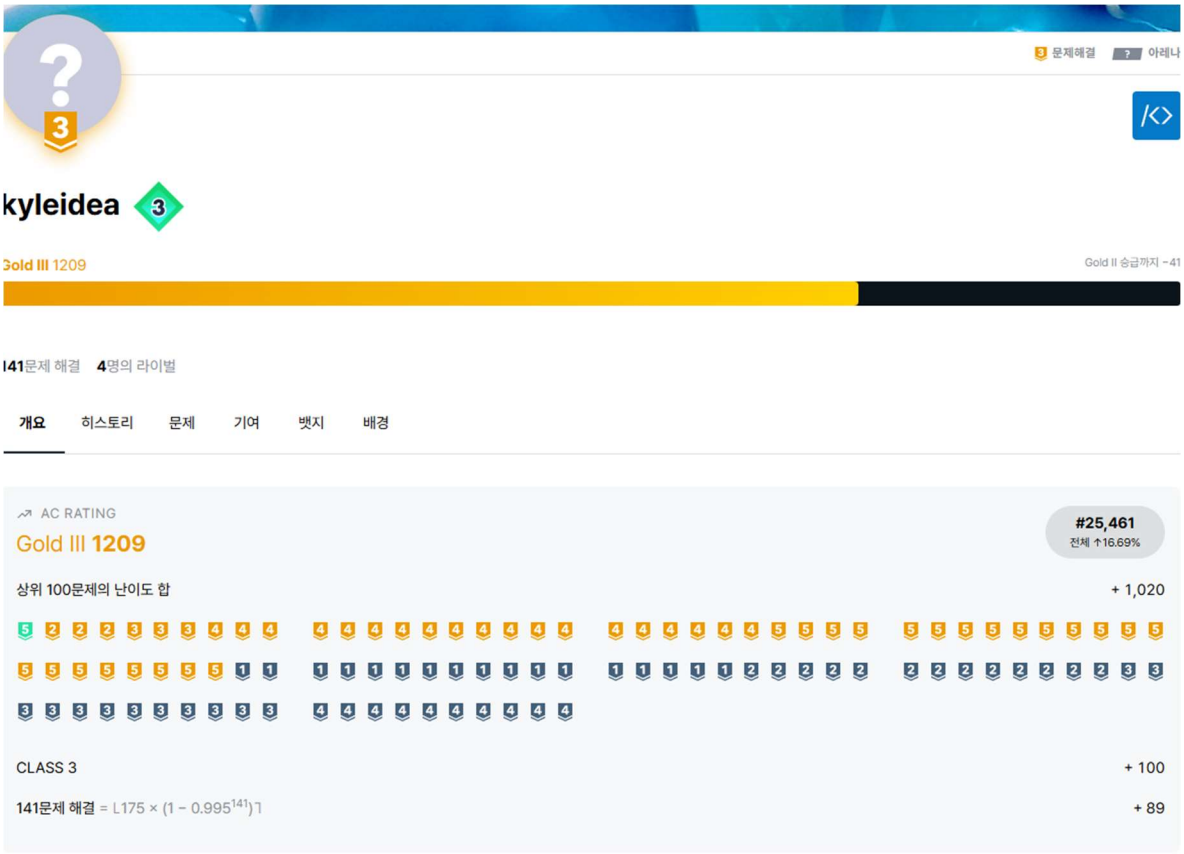
   .

<LIS 발표자료>

```python
# 11053 Silver 2
# Dp Solution_O(n^2)
# Find the length of LIS

import sys
input = sys.stdin.readline

n = int(input())
l = list(map(int, input().strip().split()))

dp = [1]*(n)

for i in range(1,n):
    for j in range(i):
        if l[i] > l[j]: # 증가 시에만 dp[j]가 dp[i]에 비해 커질 여지가 있음
            if dp[i] < dp[j]+1: # dp[j]+1 은 l[j]를 LIS의 구성요소로 채택함을 의미
                dp[i] = dp[j] + 1

print(max(dp)) # 결론적으로 dp 테이블은, 각 자리수까지 LIS의 길이만을 담음
```

```python
# 14002 Gold 4
# Dp Solution_O(n^2)
# Find one specific LIS

import sys
input = sys.stdin.readline

n = int(input())
l = list(map(int,input().split()))

dp = [1]*n

for i in range(1,n):
    for j in range(i):
        if l[j] < l[i]:
            dp[i] = max(dp[i],dp[j]+1) # 여기까지 동일.

lis = []

max_dp = max(dp) # 역추적을 위한 DP 최댓값
max_idx = dp.index(max_dp) # 역추적을 위한 DP 최댓값의 index
while max_idx >= 0:
    if dp[max_idx] == max_dp: # LIS의 끝 원소부터 역추적 시작
        lis.append(l[max_idx])
        max_dp -= 1 # dp 최댓값이 줄어들면서
    max_idx -= 1 # 역방향으로 계속해서 탐색.

lis.reverse() # lis에는 실제 LIS의 역순이 들어있음. 뒤집어!

print(len(lis))
print(*lis)
```

3. 성과



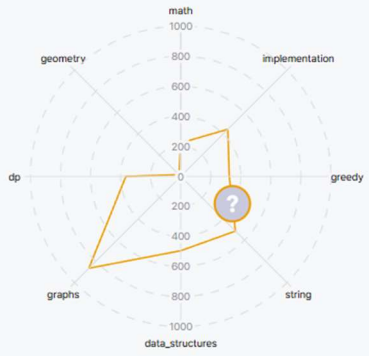
스터디 시작 전의 목표였던 백준 티어 골드를 달성하였고, 나아가 개인적으로도 많은 문제를 풀어 목표를 초과 달성할 수 있었습니다. 스터디를 하지 않았다면 팀원들을 위해 공부하는 보람과 강제성이 부여되지 않아 조금 힘들었을 것 같습니다.

4. 앞으로의 다짐

많은 문제를 풀고 목표도 달성했지만, 해결한 문제의 분포를 보면 특정 알고리즘에 치우쳐진 경향이 있습니다. 상대적으;로 부족한 동적계획법, 기하, 수학, 구현 문제 풀이 비중을 조금 더 늘리고, 기업 코딩테스트 문제를 많이 풀어봐야겠다는 생각을 하였습니다.

<현재 해결한 문제 알고리즘 분포>

태그 분포



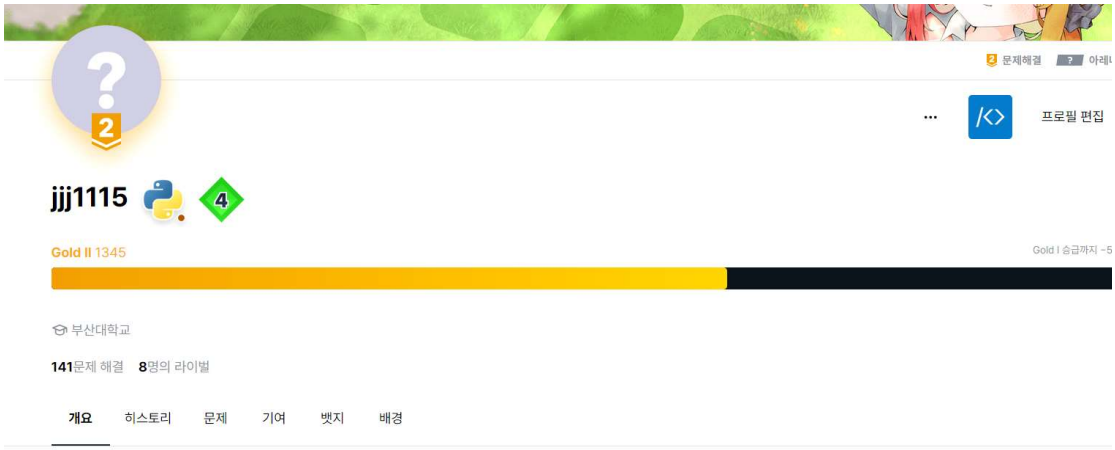
태그	문제		레이팅
#그래프 이론	38	27.0%	5 865
#그래프 탐색	30	21.3%	1 666
#너비 우선 탐색	27	19.1%	2 596
#문자열	32	22.7%	2 515
#자료 구조	24	17.0%	3 495
#구현	35	24.8%	3 443
#정렬	23	16.3%	3 421
#다이나믹 프로그래밍	17	12.1%	4 365
#그리디 알고리즘	17	12.1%	4 323
#최단 경로	13	9.2%	4 316

더 보기 (+22)

멘사 활동 보고서

202155655 정진택

저의 현재 백준 티어입니다.



이번 학기 멘사 활동을 하면서 공부한 내용은 다음과 같습니다.

1. 다익스트라 알고리즘 (Dijkstra's Algorithm)

다익스트라 알고리즘은 **가중 그래프에서 단일 시작점에서 다른 모든 정점까지의 최단 경로를 찾는 알고리즘**입니다.
그래프의 간선 가중치가 모두 비음수여야 동작합니다.

동작 과정

- 시작 노드에서의 거리를 0으로 설정하고, 나머지 노드의 거리를 무한대로 초기화합니다.
- 방문하지 않은 노드 중 가장 거리가 짧은 노드를 선택하여 처리합니다.
- 해당 노드와 인접한 노드의 거리를 갱신합니다.
- 모든 노드를 방문할 때까지 2~3단계를 반복합니다.

시간 복잡도

- 인접 리스트와 **우선순위 큐**를 사용하면 O((V+E)logV)
  (V: 정점 수, E: 간선 수)

활용 예

- GPS 시스템의 최단 경로 탐색
- 네트워크 패킷 전송 경로 최적화

2. 프림 알고리즘 (Prim's Algorithm)

프림 알고리즘은 **가중치가 있는 연결 그래프에서 최소 신장 트리(MST)를 찾는 알고리즘**입니다.
시작 정점에서 출발하여 트리를 확장하며 최소 비용 간선을 선택합니다.

동작 과정

- 임의의 시작 정점을 선택하고, 방문한 노드 집합에 추가합니다.
- 현재 방문한 노드와 인접한 간선 중 가장 가중치가 작은 간선을 선택합니다.
- 선택한 간선이 연결하는 정점을 방문한 노드 집합에 추가합니다.
- 모든 정점이 방문될 때까지 2~3단계를 반복합니다.

시간 복잡도

- 인접 리스트와 **우선순위 큐**를 사용하면 O(ElogV)

활용 예

- 네트워크 연결 비용 최적화
- 전력망 구축

3. 백트래킹 (Backtracking)

백트래킹은 **모든 가능한 해를 탐색하는 방법으로, 불필요한 탐색을 가지치기(pruning)하여 효율성을 높이는 알고리즘**입니다.
주로 재귀를 사용해 상태 공간 트리를 탐색합니다.

동작 과정

- 현재 단계에서 가능한 모든 선택지를 시도합니다.
- 조건을 만족하지 않으면 탐색을 중단하고 이전 단계로 되돌아갑니다.

- 조건을 만족하면 결과를 저장하거나 다음 단계로 진행합니다.
- 모든 경로를 탐색할 때까지 반복합니다.

시간 복잡도

- 상태 공간 크기에 따라 다름. 일반적으로 O(bd)
  (b: 노드의 분기 수, d: 탐색 깊이)

활용 예

- N-Queen 문제
- 미로 탐색
- 조합 및 순열 생성

4. 깊이 우선 탐색 (DFS, Depth-First Search)

DFS는 **그래프나 트리에서 깊은 부분을 우선적으로 탐색하는 알고리즘**입니다.
스택 또는 재귀를 사용하여 구현합니다.

동작 과정

- 시작 정점에서 출발하여 방문하지 않은 인접 정점을 탐색합니다.
- 더 이상 방문할 정점이 없을 때 이전 정점으로 돌아옵니다.
- 모든 정점을 방문할 때까지 반복합니다.

시간 복잡도

- 인접 리스트를 사용하면 O(V+E)

활용 예

- 그래프의 연결 요소 탐색
- 순환 여부 판단
- 위상 정렬

5. 너비 우선 탐색 (BFS, Breadth-First Search)

BFS는 **그래프나 트리에서 너비를 우선적으로 탐색하는 알고리즘**입니다.
큐를 사용하여 구현하며, 최단 경로를 구할 때 유용합니다.

동작 과정

- 시작 정점을 큐에 삽입하고 방문 처리합니다.
- 큐에서 정점을 하나씩 꺼내고, 방문하지 않은 인접 정점을 큐에 추가합니다.
- 큐가 빌 때까지 반복합니다.

시간 복잡도

- 인접 리스트를 사용하면 O(V+E)

활용 예

- 그래프에서 최단 경로 탐색(가중치가 동일할 때)
- 레벨 탐색(계층적 탐색)

중간보고서

멘사스터디 202055614 최성민

1. 공부한 내용:

이번 스터디를 통해서 공부한 내용은 그래프 탐색에 관한 내용이다. DFS란 깊이 우선 탐색으로 한 경로를 따라 끝까지 탐색한 후 다른 경로를 탐색하는 방식이며, 스택 또는 재귀를 활용하여 구현한다. 모든 경로를 탐색하므로 완전 탐색에 적합하다. BFS는 너비 우선 탐색으로 시작 정점으로부터 가까운 정점부터 탐색하는 방식이며, 큐를 활용하여 구현한다. 최단 경로 탐색에 적합하다.

이를 이용해서 그래프 탐색 문제에서 어떤 부분에서 DFS와 BFS를 구별하여 사용할 지에 대해 알아볼 수 있었다.

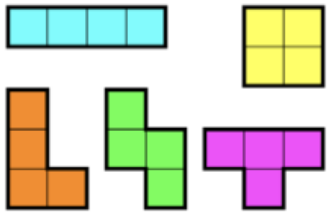
2. 14500.py(테트로미노)

- 문제:

폴리오미노란 크기가 1×1인 정사각형을 여러 개 이어서 붙인 도형이며, 다음과 같은 조건을 만족해야 한다.

- 정사각형은 서로 겹치면 안 된다.
- 도형은 모두 연결되어 있어야 한다.
- 정사각형의 변끼리 연결되어 있어야 한다. 즉, 꼭짓점과 꼭짓점만 맞닿아 있으면 안 된다.

정사각형 4개를 이어 붙인 폴리오미노는 테트로미노라고 하며, 다음과 같은 5가지가 있다.



아름이는 크기가 N×M인 종이 위에 테트로미노 하나를 놓으려고 한다. 종이는 1×1 크기의 칸으로 나누어져 있으며, 각각의 칸에는 정수가 하나 쓰여 있다.

테트로미노 하나를 적절히 놓아서 테트로미노가 놓인 칸에 쓰여 있는 수들의 합을 최대로 하는 프로그램을 작성하시오.

서식 지정함: 글꼴: 14 pt

서식 있음: 들여쓰기: 왼쪽: 1.41 cm, 글머리 기호 또는 번호 없이

서식 지정함: 글꼴: 10 pt

서식 지정함: 글꼴: 12 pt

테트로미노는 반드시 한 정사각형이 정확히 하나의 칸을 포함하도록 놓아야 하며, 회전이나 대칭을 시켜도 된다.

- 해결방안

N x M 크기의 종이에서 테트로미노를 만족하면서 그 수의 합이 최대로 되어야하는 문제이다. 테트로미노 모양을 지키며 수들의 합을 계산해 나가야 하므로 그래프 탐색 기법을 사용해야겠다고 생각했다. 4<=N,M<=500이므로 크기가 그다지 크지 않기 때문에 한 점에 대해서 테트로미노 모양으로 최대 4번씩만 확장하며 완전 탐색을 해야 겠다고 생각했다. 따라서 방식은 DFS로 재귀를 이용해 4번 확장이 되면 그만 확장하고 global변수로 수의 합의 최댓값을 구하는 것으로 구현하였다. 여기서 중요한 점은 'ㅗ'모양은 위 탐색에서 만들어지지 않기 때문에 따로 이 모양의 계산을 할 수 있게끔 함수를 하나 더 만들었다.

- 답안 코드

```python
import sys  #dfs, 구현 14500.py 테트로미노
input = sys.stdin.readline
dx = [1,0,-1,0]
dy = [0,1,0,-1]

N, M = map(int, input().split())
graph = [list(map(int, input().split())) for _ in range(N)]
visited = [[0]*M for _ in range(N)]
max_ans = 0

def dfs(x, y, n, temp):  #ㅗ모양 제외 나머지 테트로미노
    global max_ans
    if n==4:
        max_ans = max(max_ans, temp)
        return
    for i in range(4):
        nx = x + dx[i]
        ny = y + dy[i]
        if 0<=nx<N and 0<=ny<M and not visited[nx][ny]:
            visited[nx][ny] = 1
            dfs(nx,ny,n+1,temp+graph[nx][ny])
            visited[nx][ny] = 0
```

서식 있음: 목록 단락, 왼쪽, 글머리 기호 + 수준:1 + 맞춤 위치: 0.78 cm + 들여쓰기 위치: 1.41 cm

서식 있음: 왼쪽

```
def fk(x,y,temp): #ㅗ모양 4방위 값중 가장 작은 값 제외한 합으로 계산
    global max_ans
    f = []
    for i in range(4):
        nx = x + dx[i]
        ny = y + dy[i]
        if 0<=nx<N and 0<=ny<M:
            f.append(graph[nx][ny])
    if len(f)==4:  #한 방향 가장 작은 값 빼기
        max_ans = max(max_ans, sum(f) - min(f) + temp)
    elif len(f)==3: # 3방향 합
        max_ans = max(max_ans, sum(f)+temp)
    else:
        return

for i in range(N):
    for j in range(M):
        visited[i][j] = 1
        dfs(i, j, 1, graph[i][j])
        fk(i,j,graph[i][j])
        visited[i][j] = 0

print(max_ans)
```

서식 있음: 왼쪽, 오른쪽: 0.71 cm

서식 있음: 표준, 들여쓰기: 왼쪽: 0 cm

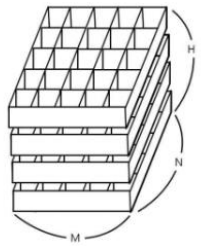
서식 지정함: 글꼴: 12 pt

서식 있음: 목록 단락, 번호 매기기 + 수준:1 + 번호 스타일: 1, 2, 3, … + 시작 번호: 1 + 맞춤: 왼쪽 + 맞춤 위치: 0.78 cm + 들여쓰기 위치: 1.41 cm

3. 7576, 7569.py (토마토)

- 문제:

철수의 토마토 농장에서는 토마토를 보관하는 큰 창고를 가지고 있다. 토마토는 아래의 그림과 같이 격자모양 상자의 칸에 하나씩 넣은 다음, 상자들을 수직으로 쌓아 올려서 창고에 보관한다.



창고에 보관되는 토마토들 중에는 잘 익은 것도 있지만, 아직 익지 않은 토마토들도 있을 수 있다. 보관 후 하루가 지나면, 익은 토마토들의 인접한 곳에 있는 익지 않은 토마토들은 익은 토마토의 영향을 받아 익게 된다. 하나의 토마토에 인접한 곳은 위, 아래, 왼쪽, 오른쪽, 앞, 뒤 여섯 방향에 있는 토마토를 의미한다. 대각선 방향에 있는 토마토들에게는 영향을 주지 못하며, 토마토가 혼자 저절로 익는 경우는 없다고 가정한다. 철수는 창고에 보관된 토마토들이 며칠이 지나면 다 익게 되는지 그 최소 일수를 알고 싶어 한다.

토마토를 창고에 보관하는 격자모양의 상자들의 크기와 익은 토마토들과 익지 않은 토마토들의 정보가 주어졌을 때, 며칠이 지나면 토마토들이 모두 익는지, 그 최소 일수를 구하는

프로그램을 작성하라. 단, 상자의 일부 칸에는 토마토가 들어있지 않을 수도 있다.

- 해결 방안:

7576번 문제는 위 문제에서 높이(H)가 없는 문제이다. 이 문제는 하루마다 익은 토마토가 주변의 토마토를 익게 만들어서 토마토가 전부 익게 되는 최소 날짜를 구하는 문제이므로 하루가 지날 때마다 1을 더해서 그래프 탐색이 끝났을 때 가장 큰 값을 구하면 된다고 생각했다. 최단 경로 탐색이라는 점에서 BFS로 진행을 하되, 높이도 고려해야 하므로 주변에 익지 않은 토마토를 확인하고 그래프 탐색을 이어가는 과정을 유심하게 구현했다. 그래프 탐색이 끝난 후 토마토가 전부 익을 수 없는 상태일 때에는 -1을 출력해야 하므로 반복문을 통해 -1이 있는지 확인과정을 거쳤다.

- 답안 코드:

서식 있음: 글머리 기호 + 수준:1 + 맞춤 위치: 0.78 cm + 들여쓰기 위치: 1.41 cm

서식 있음: 들여쓰기: 왼쪽: 1.55 cm

서식 있음: 왼쪽

```
import sys #tomato bfs
from collections import deque
input = sys.stdin.readline

dx = [1,0,-1,0,0,0]
dy = [0,1,0,-1,0,0]
dz = [0,0,0,0,1,-1]

col, row, h = map(int, input().split()) #가로,세로,높이
tomato = [[list(map(int, input().split()))for _ in range(row)] for __ in range(h)]
q = deque([])

for i in range(h):
    for j in range(row):
        for k in range(col):
            if tomato[i][j][k] == 1: #익은 토마토 탐색
                q.append((i,j,k))

while q:
    z,x,y = q.popleft()
    for l in range(6):
        nz = z + dz[l]
        nx = x + dx[l]
        ny = y + dy[l]
        if 0<=nx<row and 0<=ny<col and 0<=nz<h:
            if tomato[nz][nx][ny] == 0:
                tomato[nz][nx][ny] = tomato[z][x][y] + 1 # years표시
                q.append((nz,nx,ny))
```

```python
years = 0
for i in range(h):
    for j in range(row):
        for k in range(col):
            if tomato[i][j][k] == 0:
                print(-1)
                exit(0)
            else:
                years = max(years, tomato[i][j][k])

print(years-1) #처음 1년 길게 나옴 -> -1
exit(0)
```

서식 지정함: 글꼴: 12 pt

서식 있음: 왼쪽, 들여쓰기: 왼쪽: 0.78 cm