Team 3 Presentation

문제 1: 숫자 필터링 및 정렬

문자열과 숫자가 혼합된 리스트에서 정수로 변환 가능한 값들만 추출한 뒤, 중복을 제거하고 삽입 정렬을 구현**하여 오름차순으로 정렬하였다.
최종적으로 정렬된 값들의 총합을 계산하여 PASSCORD를 도출하였다.

- 사용 알고리즘: 삽입 정렬 (직접 구현)
- 핵심 로직: 필터링 → 중복 제거 → 삽입 정렬 → 합산

```
shuffled_list = [
    "-900", "3", "baz", "six^", "2", "65", "comma,", "0", "adipiscing", "-500",
    "space ", "foo", "15", "75", "-100", "10", "one!", "lorem", "one", "-600",
    "-450", "bar", "world", "five%", "star*", "ten", "four$", "500", "95", "thousand",
    "300", "-50", "-25", "70", "minus", "xyz", "-200", "abc!", "do", "-888",
    "35", "incididunt", "400", "7", "700", "-20", "100", "500", "100", "hello",
    "3a", "55", "three#", "-350", "-10", "five", "4", "-4", "-700", "2",
    "abc", "xyz", "-250", "2", "zero", "-650", "-15", "ten)", "consectetur", "-1",
    "45", "eight*", "elit", "-77", "-50", "ten", "0", "underscore_", "four", "-100",
    "-300", "-10", "40", "-850", "seven&", "700", "-200", "amet", "-25", "1",
    "33", "85", "dolore", "nine(", "99", "dolor", "6", "-999", "3", "baz",
    "lorem", "six^", "bar", "seven&", "100", "xyz", "two@", "adipiscing", "-200", "abc!",
    "-1000", "nine(", "5", "-850", "consectetur", "999", "bar", "xyz", "-700", "999",
    "foo", "plus", "70", "-900", "-20", "-250", "incididunt", "three#", "one!", "hello",
    "six^", "1", "-777", "abc", "comma,", "cat", "bar", "baz", "baz", "3a",
    "abc123", "-500", "7", "-650", "-300", "hello", "bar", "NaN", "999", "space ",
    "999", "star*", "ten", "-450", "underscore_", "three#", "baz", "five", "-100", "-1000",
    "-100", "baz", "one", "1.5", "nine(", "-15", "five%", "5", "six^", "-300",
    "consectetur", "-100", "cat", "eight*", "baz", "baz", "space ", "baz", "baz", "baz"
]
```

```
# 정답
# 리스트의 합: -5967
# 삽입 정렬 시간: 0.000084초
# [-1000, -999, -900, -888, -850, -777, -700, -650, -600, -500, -450,
```

```
# 문자를 숫자로 바꾸고 해당 과정에서 에러가나는 경우는 제외
processed_list = []
for item in shuffled_list:
    try:
        # 숫자로 변환 가능한 경우 float으로 변환 (정수든 실수든 다 커버)
        number = int(item)
        processed_list.append(number)
    except (ValueError, TypeError):
        continue

# 기존 processed_list에서 중복 제거 (순서 유지)
arr = [] # 중복되지 않는 요소들을 모을 리스트
seen = set() # 중복을 확인하기 위한 set 생성

for num in processed_list:
    if num not in seen: # set을 이용해서 처음 이후 두번째 만나는 값들은 모두 무시
        arr.append(num)
        seen.add(num)

import time
```

사용한 정렬 알고리즘: 삽입정렬

데이터 수가 작다 (N이 작다)

- 퀵 정렬은 평균적으로 O(N log N) 복잡도를 가지지만, **"N이 매우 작을 때는"** 이점이 크지 않다.
- 오히려 퀵 정렬은 **재귀 호출**, **피벗 선택**, **파티션 분할** 같은 부가적인 오버헤드가 있어서 소규모 데이터에서는 **삽입 정렬**이 이긴다.
- 51개 정도는 아직 "소규모"라 삽입 정렬의 간결한 구조가 유리하다.

문제 2: 확장 이진 탐색

주어진 정수 리스트를 오름차순으로 정렬한 뒤, **특정 target 값의 시작 인덱스와 끝 인덱스를 이진 탐색으로 각각 찾아** 두 값을 더하여 PASSCORD로 도출하였다.
만약 해당 값이 리스트에 존재하지 않는 경우 `"NOT FOUND"`를 출력하고, `"notfound"`를 PASSCORD로 사용하였다.

- 사용 알고리즘: 이진 탐색 (반복문 기반)
- 핵심 로직: 정렬 → 시작 인덱스 탐색 → 끝 인덱스 탐색 → 결과 처리

```
lst = [31, -36, -47, 44, -15, -19, -22, -33, 44, -37, 36, 44, 19, -39, 25, 4, -46, -47, -39, -23,
 -21, 14, 27, -47, 21, -25, 41, 33, 39, 19, 3, -22, 7, 25, -15, -50, 47, -30, 39, 4,
 -7, -15, -31, -23, 47, -7, -37, -39, -2, -38, -5, -6, 27, -17, -45, 43, 8, 18, -35, -2,
 -40, 20, -13, 30, 29, -4, 23, -26, 40, -42, -45, 34, -21, 48, -13, -40, -21, -38, -2, -15,
 8, 31, -4, -30, -3, -5, -24, 35, -16, 39, 37, 32, -41, 27, 31, -29, 18, 43, -19, -30]
```

```
# 정답
# [40, 41, 42, 43]
# 83
```

```python
def find_target_indices(lst, target):
    lst = sorted(lst)  # 이진 탐색을 위해 정렬
    left, right = 0, len(lst) - 1 # left right 설정
    found = False # 기존 설정값은 False

    # 이진 탐색으로 target 존재 여부 확인
    while left <= right:
        mid = (left + right) // 2
        if lst[mid] == target:
            found = True
            break
        elif lst[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    # Target이 없다면 출력할 문구
    if not found:
        print("Not Found")
        return

    # 정렬된 리스트에서 target의 모든 인덱스 찾기
    indices = [i for i, val in enumerate(lst) if val == target]
    print(indices) # 위치 리스트 출력
    print(indices[0] + indices[-1]) # 첫+마지막 인덱스 합 출력
```

문제 3: 등장 빈도 + 우선순위 정렬

정렬되지 않은 0 이상의 정수 리스트에서 collections.Counter를 사용해 각 숫자의 등장 빈도를 계산하고, 빈도가 높은 순(내림차순) 가장 우선순위로 하고, 빈도가 같으면 다음 우선 순위인 숫자가 큰 순으로 sorted() 함수를 이용해 복합 정렬하였다. 정렬된 리스트의 첫 번째 항목의 숫자와 빈도를 합산하여 PASSCORD를 도출하였다.

사용 알고리즘: Counter + 복합 정렬 (key=lambda x: (-x[1], -x[0]))
핵심 로직: 빈도 계산 → key 함수 정렬 → PASSCORD 계산

```
nums = [
5, 2, 4, 3, 9, 0, 4, 1, 3, 5,
6, 8, 4, 2, 7, 6, 0, 9, 3, 5,
2, 1, 4, 7, 8, 2, 9, 3, 5, 0,
1, 1, 6, 5, 4, 4, 3, 9, 2, 2,
8, 7, 0, 6, 5, 5, 3, 4, 1, 2,
0, 0, 9, 9, 6, 3, 2, 4, 7, 8,
5, 1, 1, 0, 0, 6, 8, 9, 2, 3,
5, 7, 4, 0, 1, 2, 8, 5, 6, 3,
9, 0, 4, 2, 3, 1, 7, 6, 5, 4,
8, 9, 0, 2, 1, 3, 5, 4, 6, 7]
```

```
# 최종 출력 결과
'''
[(5, 12), (4, 12), (2, 12), (3, 11), (0, 11), (1, 10), (9, 9), (6, 9), (8, 7), (7, 7)]
PASSCORD = 5 + 12 = 17
'''
```

```python
from collections import Counter

def frequency_sort(nums: list[int]) -> list[tuple[int,int]]:
    # Counter 클래스로 nums 안 숫자들의 등장 횟수를 센다. (예: nums = [5,3,5,2] → Counter({5:2, 3:1, 2:1}))
    freq = Counter(nums)

    # freq.items() = [(5,2), (3,1), (2,1)] 형태의 리스트를 가져온 다음,
    # sorted()로 '(횟수, 숫자)' 기준 정렬을 내림차 순으로 한다(-) → 횟수 큰 순, 횟수 같으면 숫자 큰 순
    sorted_items = sorted(freq.items(), key=lambda x: (-x[1], -x[0]))

    # 정렬된 (숫자,횟수) 리스트 반환
    return sorted_items


# 함수 호출
result = frequency_sort(nums)

# 정렬된 숫자, 횟수 리스트 결과 출력
print(result)

# 정렬된 리스트의 첫 번째 항목에 대해 숫자와  빈도수의 합(PASSCORD) 출력
passcord = result[0][0] + result[0][1]
print(f"PASSCORD =", result[0][0], "+", result[0][1], "=", passcord)
```

문제 4: 순환 기반 탈출 경로 찾기 - 최소 경로 가중치 포함

주어진 10 by 10 그리드를 (0,0)부터 출발하여 0을 찾아가는 프로그램 작성. 재귀 함수를 사용하여 한 번 움직일 때 가장 작은 값으로 이동하면서 0을 찾아간다. 그 과정에서 찍히는 모든 셀들을 더한 값이 Passcord가 된다.

사용 알고리즘: 재귀 알고리즘 핵심 로직: 주어진 그리드로 재귀함수 실행 -> 좌표를 path list에 기록하며 이동할 수 있는 곳 중 가장 작은 값으로 이동 -> 0을 만날 때까지 반복 -> 0을 만나면 재귀함수 종료

1. 주어진 그리드에서 (0,0) -> (9,9)를 찾아가는 프로그램
2. 네 방향(상하좌우)로 움직일 수 있으며 그 중 가장 작은 값을 선택해서 이동
3. 각 시행마다 가장 작은 값을 찾아 0이 나올 때까지 반복시행
4. 0을 만나면 경로와 경로에 포함된 각 셀의 숫자 총합을 출력하고 프로그램 종료

```
grid = [
    [3, 4, 1, 2, 5, 2, 3, 2, 1, 1],
    [1, 2, 3, 2, 1, 4, 2, 2, 3, 2],
    [2, 1, 1, 3, 2, 1, 1, 3, 1, 2],
    [3, 2, 4, 2, 3, 1, 2, 1, 4, 2],
    [1, 3, 2, 1, 1, 2, 4, 3, 2, 3],
    [2, 2, 1, 4, 3, 3, 1, 2, 3, 1],
    [1, 1, 2, 1, 2, 4, 3, 1, 2, 1],
    [3, 3, 1, 2, 3, 1, 1, 4, 2, 2],
    [2, 1, 2, 3, 2, 2, 1, 2, 3, 1],
    [1, 2, 1, 1, 1, 1, 1, 3, 2, 0]  # (9, 9)이 도착점
]



# 실행 함수
def run_escape_algorithm(size):
    print("\n경로 추적:")
    visited = [[False]*size for i in range(size)] # 방문 여부 저장
    if not dfs(grid, 0, 0, visited, []):
        print("NO")
```

size x size의 그리드를 만들어 방문여부를 저장하는 리스트 생성.
dfs라는 재귀함수 실행시키고,
‘TRUE’면 조건문 종료
‘FALSE’이면 ‘NO’ 출력

이동하려는 셀이 그리드 밖에 있거나 이미 방문한 셀인 경우 함수 탈출

```
def dfs(grid, x, y, visited, path):
    size = len(grid) # grid의 크기 행 개수로 정의
    # 경계 조건: x, y가 grid의 범위를 벗어나거나 이미 방문한 경우
    if x < 0 or x >= size or y < 0 or y >= size or visited[x][y]:
        return False

    # 경로 좌표를 path에 저장 후 True로 지정
    path.append((x, y))
    visited[x][y] = True

    # 0을 만났을 때
    if grid[x][y] == 0:
        print("YES") # 0 도달 여부 출력
        print(" -> ".join(f"{p}={grid[p[0]][p[1]]}" for p in path) + " -> 0 도달 -> 성공") # 도달 경로 출력
        print(f'최소값 이동한 셀: {", ".join(f"{p}={grid[p[0]][p[1]]}" for p in path)}') # 최솟값으로 이동한 셀 출력
        print(f'총합: {sum(grid[p[0]][p[1]] for p in path)}') # 경로의 총합 출력
        return True
```

path 리스트에 이동하는 좌표 저장 / 0을 만났을 때 위와 같이 출력

셀의 숫자만큼 이동시키는 부분

```
# 몇 칸을 어디로 옮길 것인지 결정
step = grid[x][y] # 현재 셀 숫자가 이동할 칸 수
directions = [(-step, 0), (step, 0), (0, -step), (0, step)]  # 상, 하, 좌, 우

candidates = [] # 후보 리스트 (상, 하, 좌, 우)
# 움직일 셀 좌표를 candidates에 저장
for dx, dy in directions:
    nx, ny = x + dx, y + dy
    # 후보 좌표가 grid의 범위 내에 있고, 방문하지 않은 경우 후보로 등록
    if 0 <= nx < size and 0 <= ny < size and not visited[nx][ny]:
        candidates.append(((dx, dy), grid[nx][ny]))
```

움직일 수 있는 셀들을 셀의 숫자와 함께 리스트에 저장

오름차순으로 정렬하여 값이 가장 작은 칸으로 이동

```
    # 값 기준으로 정렬 (작은 값 우선)
candidates.sort(key=lambda item: item[1])

# 후보 좌표로 이동
for (dx, dy), _ in candidates:
    nx, ny = x + dx, y + dy
    if dfs(grid, nx, ny, visited, path): # 한 번 움직였을 때 0에 도착하지 않았다면 함수 다시 호출
        return True
```

이동 후 같은 함수 한 번 더 호출 (재귀)

문제5: 고성능 정렬 (1,000,000개 숫자 리스트 생성 및 정렬)

백만개 이상의 숫자를 무작위로 생성하고 이에 대한 리스트로 나열한다. 그리고 내장 함수를 통해 숫자 리스트를 정렬하였다. 무작위 숫자를 만들고 내장 함수의 기본 개념인 Timsort로 빠르게 시간 내에 정렬할 수 있었다.

- 사용 알고리즘: Timsort 정렬 (내장함수: sort()사용)
- 핵심 로직: 무작위 숫자 생성 -> 내장함수 사용(정렬) -> PASSCORD 도출

```
import time
import random
import csv

numbers = []

def generate_number():          #숫자 무작위 생성
  for i in range(1,1000001):        #1,000,000번으로의 반복으로 1,000,000개 숫자 생성
    a = random.randint(1,1000000)         #random으로 1부터 1,000,000 사이의 숫자로 만들기
    numbers.append(a)         #무작위 숫자 list으로 생성
```

```
def sorting():
    start_time = time.time()
    generate_number()

    numbers.sort()

    end_time = time.time()

    elapsed = end_time - start_time

    print(numbers[:10])
    print(f"실행시간 : {elapsed:.2f}초")

    if elapsed and elapsed <3:
        print('PASSCORD = ','TRUE')

        with open("output.csv", mode="w",newline='') as file:
          writer = csv.writer(file)
          for item in numbers:
            writer.writerow([item])

    else:
        print('PASSCORD = ','FALSE')

sorting()     #함수 실행
```

아쉬운 점:

1. numbers 리스트를 만들 때, sorting 함수에서 generate_number로 리스트를 생성하기

2. sorting함수와 csv파일 만드는 함수를 나눠서 역할 분배하기

문제 6: 미로 탐색

10x10 크기의 미로에서 `S`에서 출발하여 `E`에 도달하는 경로를 찾는 로직을 작성하였다.
현재 위치가 `E`인지 확인하고, **오른쪽이나 아래쪽으로 이동 가능한 경로(`.` 또는 `E`)가 있는 경우** 재귀적으로 이동을 시도한다.
최종적으로 `E`에 도달하면 해당 위치와 이동 횟수를 출력하고 탐색을 종료하도록 하였다.

- 사용 알고리즘: 재귀 탐색
- 핵심 로직: 현재 위치 검사 → 오른쪽 이동 시도 → 아래쪽 이동 시도 → 종료 또는 실패

```
grid = [
    list("S...#....."),
    list("###.#.###."),
    list("........#."),
    list(".######.#."),
    list("......#.#."),
    list(".#.#..#.#."),
    list(".#.####.#."),
    list("........#."),
    list(".######.#."),
    list("........E.")
]
```

```
# 시작점 (0,0)에서 탐색 시작
move(grid, 0, 0)

# 정답
# 도착! 위치: (8, 9), 이동 횟수: 17
```

아쉬운 점

E에 도달하는 모든 경우의 수를 비교하고

가장 빨리 도달하는 수를 도출할 수 있도록 했다

```python
def move(maze, x, y, count=0):
    # 현재 위치가 도착 지점 'E'라면 종료
    if maze[y][x] == 'E':
        print(f"도착! 위치: ({x}, {y}), 이동 횟수: {count}")
        return True

    # 오른쪽으로 이동 가능한 경우 ('.' 또는 'E')이면 오른쪽 재귀 호출
    if x + 1 < 10 and maze[y][x + 1] in ['.', 'E']:
        if move(maze, x + 1, y, count + 1):  # x+1: 오른쪽, y: 같은 행
            return True

    # 아래쪽으로 이동 가능한 경우 ('.' 또는 'E')이면 아래쪽 재귀 호출
    if y + 1 < 10 and maze[y + 1][x] in ['.', 'E']:
        if move(maze, x, y + 1, count + 1):  # x: 같은 열, y+1: 아래쪽
            return True

    # 도달 불가능한 경우 False 반환 (백트래킹 없음, 단순 종료)
    return False
```