

Midterm Review

Team 5: 정이삭 양수민 최승리 허지윤

문제 1

정렬 알고리즘 선택 및 구현

PASSCORD:

정수형으로 변환 가능한 원소만 추출하여 정
수형으로 변환 후 중복 제거를 하고 남은 원소
들의 합



Algorithm

```
25 #quick 정렬
26 def quick_sort(arr):
27     if len(arr)<=1:    # 배열의 길이가 1보다 작거나 같다면 그냥 배열자체를 반환한다
28         return arr
29     pivot=arr[len(arr)//2] # pivot을 배열 길이를 2로 나눈 뒷으로 설정
30     left=[x for x in arr if x<pivot] # pivot보다 작으면
31     middle=[x for x in arr if x==pivot] # pivot이랑 같은 값
32     right=[x for x in arr if x>pivot ] # pivot보다 큰 값
33     return quick_sort(left)+middle+quick_sort(right)
34
```

```
"-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"
```

```
result=quick_sort(shuffled_list) # 쿠션트 사용하여 주어진 리스트를 정렬
```

정렬 알고리즘 선택 및 정렬

- 퀵정렬 사용

(평균 시간 복잡도가 $O(n \log n)$ 이고
추가 메모리 사용이 필요하지 않음)

- shuffled_list를 quick sort 함수
의 파라미터로 사용하여 정렬



```
def is_int_convertible(x):
    try:
        int(x)      # 해당 값이 정수형으로 변환 가능한지
        return True # 가능하다면 True 반환
    except (ValueError, TypeError): # 가능하지 않다면 False 반환
        return False
```

```
# 각 항목이 정수로 변환 가능한지 "YES" / "NO"로 출력
for item in result:
    if is_int_convertible(item): # is_int_convertible을 사용하여 정렬된 리스트의 값이 정수형으로 반환되는지 if 조건문 이용하여 판별
        print(f"{item!r}: YES") # 가능하면 단어와 함께 YES 출력
    else:
        print(f"{item!r}: NO") # 그렇지 않다면 NO 출력
```

정수형으로 변환 가능여부 출력

- `is_int_convertible` 함수 생성
정수형으로 변환 가능하면 `True`, 그렇지
않으면 `False` 반환
 - `for` 반복문 사용하여 정렬된 리스트
내부를 돌며 각 원소가 정수형으로
변환 가능한지 여부를 YES NO로
출력

Algorithm

```
# 정수로 변환 가능한 값만 추출하여 리스트로 변환  
convertible_strings = [int(x) for x in result if is_int_convertible(x)]  
  
result=[]  
for i in convertible_strings:  
    if i not in result: # 해당 데이터가 없다면 result 리스트에 추가해주고, 이미 존재한다면 넘어간다.  
        result.append(i) # 중복 된 값은 리스트에 담기지 못한다.  
  
# 정렬된 숫자 리스트  
result1=quick_sort(result)  
print(result1)  
total_sum = sum(result1) # 중복제거되고 남은 값을 합  
  
# 총합 출력  
print("중복 제거 후 남은 값들의 총합:", total_sum)
```

```
[DECODED]  
[-1000, -999, -900, -888, -850, -777, -700, -650, -600, -500, -450, -350, -300, -250, -200, -100, -77, -50, -25, -20, -15, -10, -4, -1, 0,  
1, 2, 3, 4, 5, 6, 7, 10, 15, 33, 35, 40, 45, 55, 65, 70, 75, 85, 95, 99, 100, 300, 400, 500, 700, 999]  
중복 제거 후 남은 값들의 총합: -5967
```

정수형으로 변환 가능한 값을 추출 후 중복제거 및 최종 원소들의 합 계산

- 정수로 변환 가능한 값만 정수형으로 변환하여 추출
- result 리스트를 만들고 for 반복문을 이용해 추출된 리스트의 원소가 리스트에 없으면 result 리스트에 append하여 중복제거 후 출력
- 중복제거되고 남은 값을 sum 함수 이용해서 합을 구하고 출력

 **Algorithm**

```
|  
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"  
]
```

Input

PASSCORD = -5967

Output

문제 2

확장 이진 탐색:
중복 요소와 예외 처리

PASSCORD:
타켓의 리스트 안에서의 시작 인덱스와
끝 인덱스의 합



```
# Lower bound index 함수
def lower_bound_index(arr, target):
    # left는 0으로 right은 마지막 인덱스로 설정
    left, right = 0, len(arr) - 1
    # 가운데 값이 타겟보다 작으면 left를 mid+1로, 그 외의 경우에는 right를 mid-1로 하면
    # left > right이 되는 순간 left는 첫번째 타겟의 인덱스를 가르키게 된다.
    # arr[mid] < target 조건이 left를 첫번째 타겟 인덱스로 옮긴다.
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return left

# upper bound index 함수
def upper_bound_index(arr, target):
    # left는 0으로 right은 마지막 인덱스로 설정
    left, right = 0, len(arr) - 1
    # 가운데 값이 타겟보다 같거나 작으면 left를 mid+1로, 그 외의 경우에는 right를 mid-1로 하면
    # left > right이 되는 순간 left는 타겟의 마지막 타겟 인덱스 다음을 가르키게 된다.
    # arr[mid] <= target 조건이 left를 마지막 타겟 인덱스 다음으로 밀어낸다.
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] <= target:
            left = mid + 1
        else:
            right = mid - 1
    return left
```

Lower/Upper bound index 함수 생성

- 정렬된 리스트에서 타겟의 시작 인덱스(Lower bound)와 타겟 다음 아이템의 인덱스(Upper bound)를 찾아주는 함수
- 이진 탐색
- mid가 target보다 작을 경우 left를 옮기면 최종 left는 Lower bound가 됨
- mid가 target보다 같거나 작을 경우 left를 옮기면 최종 left는 Upper bound가 됨

```
# 메인 함수
def find_target_index_sum(numbers, target):
    # 정렬을 먼저 한다
    numbers.sort()
    # lower bound와 upper bound를 구한다.
    start_idx = lower_bound_index(numbers, target)
    end_idx = upper_bound_index(numbers, target)

    # 값이 존재하지 않는 경우
    # 값이 존재하지 않는 경우 마지막 인덱스 바깥 혹은 첫 인덱스 바깥으로
    # lower bound와 upper bound는 같은 값을 가지게 된다.
    if start_idx == end_idx:
        print("NOT FOUND")
        print("PASSCORD = 'notfound'")
    # 값이 존재하는 경우
    else:
        print(f'{start_idx} {end_idx-1}')
        print(f"PASSCORD = {start_idx} + {end_idx-1} {start_idx + (end_idx - 1)}")
```

메인 함수 생성

- 리스트 정렬
- Lower bound와 Upper bound 구하기
- 타켓이 리스트에 존재하지 않을 경우 Lower bound와 Upper bound는 리스트 바깥 인덱스를 출력하며 동일한 값을 지니게 됨. 이 경우 notfound 출력.
- 값이 존재할 경우 시작 인덱스와 끝 인덱스 합 출력

 **Algorithm**

```
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]
```

target = -15

Input

40 43

PASSCORD = 40 + 43 83

Output

문제 3

등장 빈도 + 우선순위 정렬

PASSCORD:

(숫자, 빈도수) 형태의 튜플로 구성된 정렬된 리스트의 첫 번째 항목에 대해 숫자와 빈도수의 합



```
# import Counter
from collections import Counter

# 함수 생성
def count_unique(input_list):
    # Counter 사용하여 아이템을 key로 카운트를 value로 반환
    counts = Counter(input_list)
    # sorted를 사용하여 value를 기준으로 내림차순 정렬
    sorted_counts = sorted(counts.items(), key=lambda x: (x[1], x[0]), reverse=True)
    result = [(num, count) for num, count in sorted_counts]
    return result
```

```
output = count_unique(nums)
print(output)
print(f"PASSCORD = {output[0][0]} + {output[0][1]} = {output[0][0] + output[0][1]}")
```

함수 생성

- Counter 함수를 사용하여 (숫자, 빈도수) 리스트 반환
- sorted을 사용하여 첫번째로 빈도 수를 기준으로, 두번째로 숫자를 기준으로 내림차순 정렬
- 첫번째 아이템의 숫자와 빈도수 합

 **Algorithm**

```
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]
```

Input

```
[(5, 12), (4, 12), (2, 12), (3, 11), (0, 11),  
(1, 10), (9, 9), (6, 9), (8, 7), (7, 7)]
```

PASSCORD = 5 + 12 = 17

Output

문제 4

순환 기반
탈출 경로 찾기

PASSCORD:

- 탈출 경로를 찾고, 가능할 경우 경로 상의 모든 셀 값을 더한 최소값
- 탈출이 불가능한 경우에는 "FAIL" 출력



```
# dfs 기반 함수 정의
def dfs(x, y, path):
    current_time = time.time() # 현재 시간 측정하고 1초 지나면 종료되도록
    if current_time - start_time > 1.0:
        print("FAIL (시간 초과)")
        sys.exit()

    current_mem = tracemalloc.get_traced_memory()[1] / (1024 * 1024) # 현재 메모리 사용량 측정, MB 단위, 256MB 초과시 종료되도록
    if current_mem > 256:
        print("FAIL (메모리 초과)")
        sys.exit()

    if x < 0 or x >= n or y < 0 or y >= n: # 주어진 크기의 범위를 벗어나지 못하도록
        return
    if visited[x][y]: # 무한 루프 방지(이미 지나간 자리 다시 돌아가지 않도록)
        return
    if grid[x][y] == 0: # 도착지점(0)에 도달한 경우 경로 저장하고 종료
        path.append((x, y))
        all_paths.append(path.copy())
        path.pop()
        return

    visited[x][y] = True # 현재 자리 방문 표시
    path.append((x, y)) # 현재 자리를 경로에 추가
    step = grid[x][y] # 현재 자리의 숫자 만큼 이동 가능

    for dx, dy in [(-step, 0), (step, 0), (0, -step), (0, step)]: # 상하좌우 방향으로 현재 자리의 숫자 만큼 이동 가능
        dfs(x + dx, y + dy, path)

    visited[x][y] = False # 다른 경로 탐색을 위해 방문 표시 지우고 경로에서도 제거
    path.pop()
```

DFS 알고리즘 구현

- 재귀 함수를 이용하여 DFS 구현
 - 현재 위치에서 이동 가능한 모든 경로를 탐색
- 경로 저장 및 방문 기록 관리
- 시간 및 메모리 제한 처리
 - 1초, 256MB
- 도달 가능한 경로들을 하나씩 출력하고, 가장 적은 칸 수로 도달한 경로와 그 총 합도 함께 출력



```
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  
]
```

Input

PASSCORD = FAIL (시간 초과)

Output

문제 6

순환 기반
탈출 경로 찾기

PASSCORD:

- 출발점 S에서 도착점 E까지 이동 가능한 경로가 존재하는지 탐색
- 경로가 존재한다면 경로의 길이를 출력, 존재하지 않으면 "NO PATH"를 출력



```
def dfs(x, y, path, visited):
    # 현재 위치가 범위를 벗어나는 경우 더 이상 진행 못하도록
    if x < 0 or x >= n or y < 0 or y >= n:
        return False

    # 벽이 있는 곳이거나 이미 방문한 적이 있는 곳이면 더 이상 진행 못하도록
    if grid[x][y] == '#' or visited[x][y]:
        return False

    # 도착 지점이면 경로에 추가하고 TRUE 리턴
    if grid[x][y] == 'E':
        path.append((x, y))
        return True

    visited[x][y] = True # 현재 위치 방문 표시
    path.append((x, y)) # 현재 위치 경로에 추가

    for dx, dy in [(-1,0), (1,0), (0,-1), (0,1)]: # 상하좌우 네 방향으로 이동 가능
        nx, ny = x + dx, y + dy # 새로운 위치 계산
        if dfs(nx, ny, path, visited): # 재귀 함수 사용하여 다음 칸 탐색
            return True # 다음칸에서 도착지점까지 연결된 경로가 있는 경우 TRUE 반환

    # 네 방향 모두 실패한 경우, 현재 위치를 경로에서 제거하고 방문 표시한 것도 없애기
    path.pop()
    visited[x][y] = False
    return False
```

DFS 알고리즘 구현

- 재귀 함수를 이용하여 DFS 구현
- 경로 저장 및 방문 기록 관리
- 도달 가능 경로를 찾은 경우 프린트문과 함께 총 이동 거리를 출력
- 도달 가능 경로가 없는 경우 찾지 못했다는 프린트문을 출력

 **Algorithm**

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

Input

Path found!
Path length(BONUS_ANSWER) = 47

Output

문제 5

제한 시간 내에 100만개의
대용량 숫자 리스트를 정렬

PASSCORD:

3가지 조건을 만족하는지 여부
(TRUE or FALSE)



```
1 # 필요한 라이브러리 import 및 1,000,000개의 랜덤 숫자 생성
2 import pandas as pd
3 import random
4 import time
5
6 numbers = [random.randint(0, 10**6) for _ in range(10**6)]
7 print(f"정렬되지 않은 숫자 리스트: \n {numbers[0:10]}...")
8 print(f"정렬되지 않은 리스트 길이: {len(numbers)}")
9
10 numbers.sort()
11 sorted_numbers = numbers
12
13 print(f"sort: \n {sorted_numbers[0:10]}")
14 print("=*65")
15 print(f"정렬된 리스트 길이: {len(sorted_numbers)}")
```

```
sort:
[1, 2, 3, 4, 5, 5, 7, 8, 10, 10]
=====
정렬된 리스트 길이: 1000000
```

조건 1&2&4&5(1)(3)

1~100만 이하의 값으로 구성된
정수형 데이터 100만개를 정렬하고

*정렬 정확성 및 *출력값을 검사

- 조건을 만족하는 정수형 데이터를 random 라이브러리의 randint로 생성하여 numbers로 저장
- 기본 내장 함수 .sort()를 사용하여 numbers를 정렬하고, sorted_numbers로 다시 저장
- *정렬 정확성 및 *출력값을 검사하기 위해 sorted_numbers의 앞 10개 항목과 길이를 출력하여 확인

```
20 # 알고리즘 시간 복잡도 분석
21 times_timSort = []
22
23 start_time = time.time()
24 numbers.sort()
25 end_time = time.time()
26 times_timSort.append(end_time - start_time)
27
28 print(f"실행 시간: {times_timSort[0]}")
```

조건 3. 리스트를 제한 시간 3초 이내에 정렬

- time 라이브러리의 time()를 사용하여 시작 시간을 start_time에, 종료 시간을 end_time에 저장
- end_time에서 start_time의 차이를 계산하여 times_timSort[]에 저장
- times_timSort[]에 저장된 값이 정렬에 소요된 시간

```
38 if times_timSort[0] <= 3:  
39     print("실행시간: ", times_timSort[0])  
40     print(sorted_numbers[0:10])  
41     print('PASSCORD =', "'TRUE'")  
42 else:  
43     print('PASSCORD =', "'FALSE'")
```

조건 3&5.

*실행 시간이 3초 이내면 “TRUE”
초과하면 “FALSE”를 출력

- 앞에서 *정렬 정확성 및 *출력값에 대한 조건을 만족했기에 *실행 시간에 대한 조건(≤ 3)을 만족하는지 검사
- 만족할 경우 실행시간과 정렬된 리스트의 앞 10개 항목, “TRUE”를 반환

정렬되지 않은 숫자 리스트:

[291514, 759907, 78452,
276582, 472312, 503558,
223281, 808474, 38330,
52635]...

정렬되지 않은 리스트 길이:

1000000

Input

실행시간: 0.05045604705810547
[1, 2, 3, 4, 5, 5, 7, 8, 10, 10]
PASSCORD = 'TRUE'

Output

Thank You

Team 5: 정이삭 양수민 최승리 허지윤