알고리즘_3~4일차 (1/31~2/1)

목차

배열(List)2 (Array 2)

---

1. 배열 : 2차원 배열

1-1. 2차원 배열

> 1차원 List를 묶어놓은 List
>
> 2차원 이상의 다차원 List는 차원에 따라 index를 선언
>
> 2차원 List의 선언 : 세로길이(행 개수), 가로길이(열 개수)를 필요로 함
>
> Python에서는 데이터 초기화를 통해 변수선언 및 초기화 가능

> arr = [[0, 1, 2, 3], [4, 5, 6, 7]]
> (2행 4열의 2차원 List)

```
'''input을 이용한 2차원 리스트 코드 예시
3
1 2 3
4 5 6
7 8 9
를 코드로 입력하려면'''

N = int(input)
arr = [list(map(int, input().split())) for i in range(N)]

# 1 2 3이 아닌 123 처럼 공백 없이 주어지면 .split()을 제외한다.

# 빈 2차원 배열 만드는 방법 (중요!!!!!!)
empty_arr = [[0] * N for _ in range(N)]
```

1-2. 배열의 접근

배열 순회

> n X m 배열의 n*m개의 모든 원소를 빠짐없이 조사하는 방법
> 종류 : 행 우선 조회, 열 우선 조회, 지그재그 순회

```
# i = 행의 좌표, j = 열의 좌표
# n = 행 갯수, m = 열 갯수

# 1. 행 우선 조회
for i in range(n):
    for j in range(m):
        f(array[i][j]) # 필요한 연산 수행

# 2. 열 우선 조회
for j in range(m):
    for i in range(n):
        f(array[i][j]) # 필요한 연산 수행

# 3. 지그재그 순회
for i in range(n):
    for j int range(m):
        f(array[i][j + (m-1-2*j) * (i%2)])
```

델타를 이용한 2차 배열 탐색

> 2차 배열의 한 좌표에서 4방향의 인접 배열 요소를 탐색하는 방법
>
> 인덱스 (i, j)인 칸의 상하좌우 칸(ni, nj)

2차 배열의 한 좌표에서 4방향의 인접 배열 요소를 탐색하는 방법

```
arr[0...N-1][0...N-1] # NxN 배열
di[] <- [0, 1, 0, -1] #
dj[] <- [1, 0, -1, 0]

for i : 0 -> N-1:
    for j : 0 -> N-1:
        for k in range(4):
            ni <- i + di[k]
            nj <- j + dj[k]
            if 0 <= nu < N and 0 <= nj < N
            # 유효한 인덱스면
            f(arr[ni][nj])
```

> 교재 8~9쪽

```
# 강의시간 선생님 실습 교재 8쪽
i = 3
```

```
j = 4

# 방향별로 더할 값
di = [0, 1, 0, -1]
dj = [1, 0, -1, 0]

for k in range(4):
    ni = i + di[k]
    nj = j + dj[k]
    print(ni, nj)

# 특정 칸의 상하좌우 4칸 좌표를 출력하는 방법 (중요!!)
N = 5
arr = [[0] * N for _ in range(N)]

for i in range(N):
    for j in range(N):
        for k in range(4):
            ni = i + di[k]
            nj = j + dj[k]
            if 0 <= ni < N and 0 <= nj < N:
                print(arr[ni][nj], end='')
        print()
```

전치 행렬

> 1 2 3
> 4 5 6
> 7 8 9
> 에서 2와 4, 3과 7, 6과 8의 자리를 바꾸는 것

```
# i : 행의 좌표, len(arr)
# j : 열의 좌표, len(arr[0])
arr = [[1,2,3], [4,5,6], [7,8,9]]   # 3x3 행렬

for i in range(3):
    for j in range(3):
        if i < j:
            arr[i][j], arr[j][i] = arr[j][i], arr[i][j]
print(arr)
```

연습문제 1-1 (교재 12쪽)

> 5x5 2차 배열에 25개 숫자를 저장하고, 대각선 원소의 합을 구하시오.

```
N = int(input())
arr = [list(map(int, input().split())) for _ in range(N)]
```

```
total = 0

for i in range(N):
    total += arr[i][i]  # 오른쪽 아래 대각선
    total += arr[i][N-1-i]
if N % 2:   # 크기가 홀수인 경우
    total -= arr[N//2][N//2]    # 중심원소가 중복되므로
print(total)
```

연습문제 1-2 (교재 13쪽)

> 5x5 2차 배열에 25개 숫자를 저장하고, 25개의 각 요소에 대해서 그 요소와 이웃한 요소와의 차의 절대값을 구하시오.
> 아래 코드 공식마냥 외워두기!!!

```
di = [0, 1, 0, -1]
dj = [1, 0, -1, 0]

N = int(input())
arr = [list(map(int, input().split())) for _ in range(N)] # 5x5 2차 배열에 25개의 숫자를 저장하고
total = 0
for i in range(N): # 25개의 각 요소에 대해서  arr[i][j]
    for j in range(N):
        for k in range(4):
            ni = i + di[k]
            nj = j + dj[k]
            if 0 <= ni < N and 0 <= nj < N:
                # 차의 절대값을 구하시오, 총합을 구하시오.
                total += abs(arr[ni][nj] - arr[i][j])
print(total)
```

---

2. 부분집합

2-1. 부분집합 합 (Sunset Sum) 문제

> 유한 개의 정수로 이루어진 집합이 있을 때,
> 이 집합의 부분집합 중에서 그 집합의 원소를 모두 더한 값이 0이 되는 경우가 있는지를 알아내는 문제

> 예를 들어, [-7, -3, -2, 5, 8]인 집합이 있을 때,
> [-3, -2, 5]는 이 집합의 부분집합이면서 (-3) + (-2) + 5 = 0 이므로 답은 참이다.

2-2. 부분집합 생성하기

> 완전검색 기법으로 부분집합 합 문제를 풀기 위해서는,
> 우선 집합의 모든 부분집합을 생성한 후에 각 부분집합의 합을 계산해야 한다.

부분집합의 수

> 집합의 원소가 n개일 때, 공집합을 포함한 부분집합의 수는 2^n개 이다.

> 이는 각 원소를 부분집합에 포함시키거나 포함시키지 않는 2가지 경우를 모든 원소에 적용한 경우의 수와 같다.

> 예를 들어, {1, 2, 3, 4} 는 (2 x 2 x 2 x2) = 16가지

부분집합 생성 코드

> 각 원소가 부분집합에 포함되었는지를 loop를 이용하여 확인하고 부분집합을 생성하는 방법

```
bit = [0, 0, 0, 0]
for i in range(2):
    bit[0] = i  # 0번 원소
    for j in range(2):
        bit[1] = j  # 1번 원소
        for k in range(2):
            bit[2] = k  # 2번 원소
            for l in range(2):
                bit[3] = l  # 3번 원소
                print_sunset(bit)
                # 생성된 부분집합 출력
```

비트 연산자

연산자	내용
&	비트 단위로 AND 연산을 한다.
|	비트 단위로 OR 연산을 한다.
<<	피연산자의 비트 열을 왼쪽으로 이동시킨다.
>>	피연산자의 비트 열을 왼쪽으로 이동시킨다.

<< 연산자

> **1 << n**
> 원소가 n개일 경우의 모든 부분집합의 수를 의미 (2^n)

&연산자

> **i & (1<<j)**
> i의 j번째 비트가 1인지 아닌지를 검사한다.

비트연산자 사용한 부분집합 생성 코드

> 보다 간결하게 부분집합을 생성하는 방법

```
arr = [3,6,7,1,5,4]

n = len(arr)    # n: 원소의 개수

for i in range(1<<n):   # 1<<n: 부분집합의 개수
    for j in range(n):  # 원소의 수만큼 비트를 비교
        if i & (1<<j):  # i의 j번 비트가 1인 경우
            print(arr[j], end=" ")  # j번 원소 출력
    print()
print()
```

연습문제 2 (교재 22쪽)

> 10개의 정수를 입력 받아 부분집합의 합이 0이 되는 것이 존재하는지를 계산하는 함수를 작성해보자.

> 정수 : -7, -5, 2, 3, 8, -2, 4, 6, 9

```
def f(arr, N):
    for i in range(1, 1 << n):
        s = 0
        for j in range(n):
            if i & (1<<j):
                s += arr[j]
        if s == 0:
            return True
    return False

N = int(input())
arr = list(map(int, input().split()))

print(f(arr, N))
```

---

3. 검색

> 저장되어 있는 자료 중에서 원하는 항목을 찾는 작업

> 목적하는 탐색 키를 가진 항목을 찾는 것
> 탐색 키(search key) : 자료를 구별하여 인식할 수 있는 키

> 검색의 종류 : 순차 검색, 이진 검색, 해쉬

3-1. 순차 검색 (Sequential Search)

> 일렬로 되어 있는 자료를 순서대로 검색하는 방법

> 가장 간단하고 직관적인 검색 방법
> 배열, 연결리스트 등 순차구조로 구현된 자료구조에서 원하는 항목을 찾을 때 유용함
> 알고리즘이 단순하여 구현이 쉽지만, 검색 대상의 수가 많은 경우에는 수행시간이 급격히 증가해 비효율적

> 2가지 경우

1. 정렬되어 있지 않은 경우
2. 정렬되어 있는 경우

정렬되어 있지 않은 경우

> **검색과정**

1. 첫 번째 원소부터 순서대로 검색 대상과 키 값이 같은 원소가 있는지 비교하며 찾는다.
2. 키 값이 동일한 원소를 찾으면 그 원소의 인덱스를 반환
3. 자료구조의 마지막에 이를 때까지 검색 대상을 찾지 못하면 검색 실패

> 찾고자 하는 원소의 순서에 따라 비교 횟수가 결정됨

> 첫 번째 원소를 찾을 때는 1번, 두 번째 원소는 2번...
>
> 정렬되지 않은 자료에서의 순차 검색의 평균 비교 횟수
> = (1/n) * (1+2+3+...+n) = (n+1) / 2

> 시간 복잡도: O(n)

구현 예시

```
def sequential_search(a, n, key):
    i <- 0
    while i < n and a[i] != key:
        i <- i + 1
    if i < n:
        return i
    else:
        return -1
```

정렬되어 있는 경우

> **검색과정**

1. 자료가 오름차순으로 정렬된 상태에서 검색을 실시한다고 가정
2. 자료를 순차적으로 검색하면서 키 값을 비교

3. 원소의 키 값이 검색대상의 키 값보다 크면 찾는 원소가 없다는 것이므로 검색 종료

> 정렬되어 있으므로, 검색 실패를 반환하는 경우 평균 비교 횟수가 반으로 줄어듬

> 시간 복잡도: O(n)

```
def sequential_search(a, n, key):
    i <- 0

    while i < n and a[i] < key:
        i <- i + 1
    if i < n and a[i] == key:
        return i
    else:
        return -1

# for문으로 쓰려면
    for i in range(N):
        if a[i] == key:
            return i
        else:
            break
    return -1
```

3-2. 이진 검색 (바이너리 서치, Binary Search)

> 자료의 가운데에 있는 항목의 키 값과 비교하여 다음 검색의 위치를 결정하고 검색을 계속 진행하는 방법

> 이진 검색을 하기 위해서는 **자료가 정렬된 상태**여야 한다.

이진 검색 검색 과정

1. 자료 중앙에 있는 원소를 고른다.
2. 중앙 원소 값과 찾고자 하는 목표 값을 비교.
3. 목표 값이 더 크면 오른쪽 반에 대해서 새로 검색을 수행
   3-1. 목표 값이 더 작으면 왼쪽 반에 대해서 새로 검색을 수행
4. 찾고자 하는 값을 찾을 때까지 1~3 과정 반복

이진 검색 알고리즘

1. 구현

> 검색 범위의 시작점과 종료점을 이용하여 검색을 반복 수행

> 이진 검색의 경우, 자료에 삽입 또는 삭제가 발생했을 때 배열의 상태를 항상 **정렬상태로 유지하는 추가 작업 필요**

이진 검색 알고리즘 코드

```
def binary_search(arr, N, key):
    start = 0   # 구간 초기화
    end = N - 1
    while start <= end:     # 검색 구간이 유효(s <= e)하면 반복해라
        middle = (start + end) // 2     # 중앙원소 인덱스
        if arr[middle] == key:    # 중앙값이 key랑 같으면 : 검색성공
            return middle
        elif arr[middle] > key:   # 중앙값이 key보다 크면
            end = middle - 1
        else:                     # 중앙값이 key보다 작으면
            start = middle + 1
    return -1    # 검색실패
```

2. 재귀 함수 이용

> 재귀 함수를 이용해서 이진 검색을 구현할 수도 있다. (추후 자세히 배운다.)

참고 : 인덱스

> Database에서 유래했으며, 테이블에 대한 동작 속도를 높여주는 자료 구조를 일컫는 말
> (다른 분야에서는 Look up table 등의 용어 사용)

> 인덱스를 저장하는데 필요한 디스크 공간은 보통 테이블을 저장하는데 필요한 디스크 공간보다 작다.
> (인덱스는 키-필드만 가지고 있고, 테이블의 다른 세부 항목들은 가지고 있지 않기 때문이다.)

> 대량 데이터의 정렬 시 성능 저하 문제를 해결하기 위해 배열 인덱스를 사용한다.

4. 선택 정렬 (Selection Sort)

> 주어진 자료들 중 가장 작은 값의 원소부터 차례대로 선택하여 위치를 교환하는 방식
> (셀렉션 알고리즘을 전체 자료에 적용한 것)

4-1. 정렬 과정

1. 주어진 리스트 중 최소값을 찾는다.
2. 그 값을 리스트의 맨 앞에 위치한 값과 교환
3. 맨 처음 위치를 제외한 나머지 리스트를 대상으로 반복 수행

시간 복잡도 : O(n^2)

코드 예시

```
# 알고리즘
def SelectionSort(a[], n):
```

```
    for i from 0 to n-2:
        a[i], ..., a[n-1] 원소 중 최소값 a[k]를 찾음
        a[i]와 a[k]를 교환
```

```
# 선택 정렬
def selectionSort(arr, N):
    for i in range(N-1): # i 주어진 구간의 시작
        minidx = i # 구간의 최솟값 위치 minidx, 첫 원소를 최소로 가정
        for j in range(i+1, N): # 실제 최솟값 위치 검색
            if arr[minidx] > arr[j]:
                minidx = j
        arr[minidx], arr[i] = arr[i], arr[minidx] # 최솟값을 구간의 맨 앞으로 이동
    return

N = 5
arr = [1, 3, 2, 5, 4]
selectionSort(arr, N)
print(arr)
```

---

5. 셀렉션 알고리즘 (Selection Algorithm)

> 저장되어 있는 자료로부터 k번째로 큰 혹은 작은 원소를 찾는 방법
> (최소값, 최대값, 중간값을 찾는 알고리즘을 의미하기도 함)

선택과정

> 셀렉션은 아래와 같은 과정을 통해 이루어진다.

1. 정렬 알고리즘을 이용하여 자료 정렬하기
2. 원하는 순서에 있는 원소 가져오기

K번째로 작은 원소를 찾는 알고리즘

> 1번부터 K번째까지 작은 원소들을 찾아 배열 앞쪽으로 이동시키고, 배열의 K번째를 반환한다.
> K가 비교적 작을 때 유용하며 수행시간은 O(Kn)이다.

```
def select(arr, K):
    for i in range(0, K):
        minindex = i
        for j in range(i+1, len(arr)):
            if arr[minindex] > arr[j]:
                minindex = j
        arr[i], arr[minindex] = arrr[minindex], arr[i]
    return arr[k-1]
```

연습문제 3 (교재 56쪽)

> 다음 그림의 왼쪽과 같이 2차 배열을 초기화한 후 오른쪽 2차 배열과 같이 정렬하여 2차 배열형태로 출력하시오.

```
'''
9  20 2  18 11          1  2  3  4  5
19 1  25 3  21          16 17 18 19 6
8  24 10 17 7     ->    15 24 25 20 7
15 4  16 5  6           14 23 22 21 8
12 13 22 23 14          13 12 11 10 9

※ 시계방향으로 골뱅이 모양으로 돌기
'''
우선 왼쪽 박스를 오름차순 배열하기.
그 숫자들을 골뱅이 모양으로 돌아가게 배열하기.
swea에 코드란에서 달팽이 검색해서 풀기
```