정렬 - 6장

정렬이란?

데이터를 특정한 기준에 따라 순서대로 나열한 것을 의미한다.

선택 정렬, **삽입 정렬**, **퀵 정렬**, **계수 정렬** 이 대표적이다.

선택 정렬

가장 작은 데이터를 선택해 맨 앞에 있는 데이터와 바꾸고, 그 다음 작은 데이터를 선택해 앞에서 두 번째 데이터와 바꾸는 과정을 반복한다. 가장 원시적인 방법으로 매번 '가장 작은 것을 선택'한다는 의미에서 선택 정렬이라고 한다.

다음과 같은 코드로 작성할 수 있다.

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

# 맨 앞에서부터 최소값으로 바꿔준다.
for i in range(len(arr)):
  min_idx = i

  for j in range(i + 1, len(arr)):
    if arr[min_idx] > arr[j]:
      min_idx = j

  arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

시간 복잡도는 O(N^2) 이다. 이는 다른 정렬, 심지어 sort() 함수와 비교해봐도 성능이 매우 떨어진다. 하지만, 특정 리스트에서 가장 작은 데이터를 찾는 일이 코딩 테스트에서 잦으므로 선택 정렬 소스코드 형태에도 익숙해질 필요가 있다.

삽입 정렬

데이터를 하나씩 확인하며, 각 데이터를 적절한 위치에 삽입하는 방법이다. 삽입 정렬 또한 선택 정렬처럼 직관적으로 이해가 쉬운 알고리즘이다. 물론 구현 난이도는 보다 높지만 실행 시간 측면에서 더 효율적인 알고리즘으로 알려져 있다. 특히 삽입 정렬은 **필요할 때만 위치를 바꾸므로 '데이터가 거의 정렬 되어 있을 때' 훨씬 효과적**이다.

특정한 데이터가 적절한 위치에 들어가기 이전에, 그 앞까지의 데이터는 이미 정렬되어 있다고 가정해야 한다. 정렬되어 있는 데이터 리스트에서 적절한 위치를 찾은 뒤에, 그 위치에 삽입된다는 점이 특징이다.

즉, 현재 데이터보다 앞의 데이터들에 대해 현재 데이터를 비교해 위치를 찾아 넣는 것이다.

예를 들어, [7, 5, 4, 1, 3] 이 있다면

1. 5를 7앞에 둔다. → [5, 7, 4, 1, 3]
2. 4를 5 앞에 둔다. → [4, 5, 7, 1, 3]
3. 1을 4 앞에 둔다. → [1, 4, 5, 7, 3]
4. 3을 1과 4 사이에 둔다. → [1, 3, 4, 5, 7]

다음과 같은 코드로 작성이 가능하다.

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

for i in range(1, len(arr)):
  for j in range(i, 0, -1):
    if arr[j] < arr[j - 1]: # 현재 값이 이전 값보다 작으면 위치를 바꿔준다.
      arr[j], arr[j - 1], arr[j - 1], arr[j]
    else:
      break
```

삽입 정렬 또한 시간 복잡도는 O(N^2) 이다. 하지만 위에서도 말했다시피 삽입 정렬은 현재 데이터가 거의 정렬 되어 있는 경우 최선일 때 O(N)의 시간 복잡도를 가진다.

퀵 정렬

퀵 정렬은 대부분의 프로그래밍 언어에서 정렬 라이브러리의 근간이 되는 알고리즘이다. **퀵 정렬은 기준을 설정한 다음 큰 수와 작은 수를 교환한 후 리스트를 반으로 나누는 방식**으로 동작한다.

퀵 정렬에서는 **피벗(Pivot)**이 사용된다. 큰 숫자와 작은 숫자를 교환할 때 교환하기 위한 '기준'을 바로 피벗이라고 한다. 퀵 정렬을 수행하기 전에는 피벗을 어떻게 설정할 것인지 미리 명시해야 한다. 피벗을 설정하고 리스트를 분할하는 방법에 따라 여러 가지 방식으로 퀵 정렬을 구분하는데, 책에서는 가장 대표적인 분할 방식인 **호어 분할 방식**을 기준으로 퀵 정렬을 설명한다.

호어 분할 방식은 리스트에서 첫 번째 데이터를 피벗으러 정한다.

이와 같이 피벗을 설정한 이후 왼쪽에서부터 피벗보다 큰 데이터를 찾고, 오른쪽에서부터 피벗보다 작은 데이터를 찾는다. 그다음 큰 데이터와 작은 데이터의 위치를 서로 교환해준다. 이러한 과정을 반복하면 피벗에 대해 정렬이 수행된다.

그렇게 교환하다가 왼쪽에서부터 진행과 오른쪽에서부터의 진행이 엇갈리게 되면 현재 진행에서의 '작은 데이터'와 피벗의 위치를 서로 변경한다.

이 과정에서 첫 피벗 값보다 작은 데이터는 모두 피벗의 왼쪽에, 큰 데이터는 모두 피벗의 오른쪽에 위치하게 된다. 이러한 상태에서 다시 왼쪽 파트에 대해 피벗을 설정하고 정렬, 오른쪽 파트에서 피벗을 설정하고 정렬을 반복하다 보면 정렬이 되게 된다.

퀵 정렬의 코드는 다음과 같다. (파이썬에서만 이렇게 할 수 있다)

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

def quick_sort(array):
  # 리스트가 하나 이하의 원소만을 담고 있다면 종료
  if len(array) <= 1:
    return array

  pivot = array[0]
  tail = array[1:] # 피벗을 제외한 리스트

  # 분할
  left_side = [x for x in tail if x <= pivot]
  right_side = [x for x in tail if x > pivot]

  # 합병
  return quick_sort(left_side) + [pivot] + quick_sort(right_side)
```

퀵 정렬의 평균 시간 복잡도는 O(NlogN) 으로 앞서 설명한 선택, 삽입 정렬에 비해 매우 빠른 편이다. 즉, 만약 데이터가 100만 개일 경우 NlogN ~= 2000만으로 퀵정렬은 1초의 시간이 걸린다.

단, 퀵 정렬의 시간 복잡도에 대해 기억할 것이 있는데, 그건 바로 최악의 시간이 아닌 평균 시간이 O(NlogN) 이라는 것이다. 최악의 경우는 퀵 정렬 또한 O(N^2)가 걸린다. 데이터가 무작위로 입력되는 경우 퀵 정렬은 매우 빠르게 동작하겠지만 이미 데이터가 정렬되어 있는 경우에는 매우 느리게 동작한다. 이는 삽입 정렬과 정반대이다.

하지만 파이썬의 기본 정렬 라이브러리는 기본적으로 O(NlogN)을 보장해주기 때문에 크게 걱정할 필요는 없다.

계수 정렬

특정한 조건이 부합할 때만 사용할 수 있지만 매우 빠른 정렬 알고리즘이다. 모든 데이터가 양의 정수인 상황을 가정해보자. 데이터의 개수가 N, 데이터 중 최대값이 K일 때, 계수 정렬은 최악의 경우에도 수행 시간 O(N + K)를 보장한다. 계수 정렬은 이처럼 매우 빠르게 동작할 뿐만 아니라 원리 또한 매우 간단하다. 다만, 계수 정렬은 **'데이터의 크기 범위가 제한되어 정수 형태로 표현할 수 있을 때'**만 사용할 수 있다. 예를 들어 데이터의 값이 무한한 범위

를 가질 수 있는 실수형 데이터가 주어지는 경우 계수 정렬은 이용하기 어렵다. 일반적으로 가장 큰 데이터와 가장 작은 데이터의 차이가 100만을 넘지 않을 때 효과적인 사용이 가능하다.

예를 들어 0 이상 100 이하인 성적 데이터를 정렬할 때 계수 정렬이 효과적이다. 다만, 가장 큰 데이터와 가장 작은 데이터의 차이가 너무 크다면 계수 정렬은 사용이 불가능하다. 계수 정렬이 이러한 특징을 가지는 이유는, 계수 정렬을 이용할 때는 **'모든 범위를 담을 수 있는 크기의 리스트를 선언'** 해야 하기 때문이다.

계수 정렬은 앞서 다루었던 3가지 정렬 알고리즘처럼 직접 데이터의 값을 비교한 뒤에 위치를 변경하며 정렬하는 방식이 아니다. 일반적으로 별도의 리스트를 선언하고 그 안에 정렬에 대한 정보를 담는다는 특징이 있다. 즉, 만약 [7, 5, 9, 0, 3, 1, 6, 2, 9, 1, 4, 8, 0, 5, 2](**arr1**)가 있다면 크기가 10인 배열(**arr2**)을 0으로 초기화 시키고 각 데이터의 인덱스에 값을 +1씩 해준다. 그렇게 되면 arr2는 결국 [2, 2, 2, 1, 1, 2, 1, 1, 2]가 될 것이고 인덱스 개수만큼 숫자를 출력하면 [0, 0, 1, 1, 2, 2, 3, 4, 5, 5, 6, 7, 8, 9, 9]가 될 것이다.

이를 소스코드로 구현하면 다음과 같다.

```
array = [7, 5, 9, 0, 3, 1, 6, 2, 9, 1, 4, 8, 0, 5, 2]
count = [0] * (max(array) + 1)

for i in range(len(array)):
  count[array[i]] += 1

for i in range(len(count)):
  for j in range(count[i]):
    print(i, end=' ')
```

계수 정렬은 현존하는 정렬 알고리즘 중에서 기수 정렬과 더불어 가장 빠르다고 볼 수 있다. 물론 기수 정렬은 코딩테스트에 출제되지 않으므로 넘어간다.

하지만 문제는 계수 정렬의 공간 복잡도에 있다. 예를 들어 데이터가 0과 999999 밖에 없어도 count 배열의 크기는 100만이 된다. 따라서 항상 사용가능한 알고리즘은 아니며, 동일한 값을 가지는 데이터가 여러 개 등장할 때 적합하다. 예를 들어 성적의 경우 100점을 맞은 학생이 여러명일 수 있기 때문에 계수 정렬이 효과적이다. 반면 앞서 설명한 퀵 정렬은 일반적

인 경우에서 평균적으로 빠르게 동작하기 때문에 데이터의 특성을 파악하기 어렵다면 퀵 정렬을 이용하는 것이 유리하다.

파이썬의 정렬 라이브러리

파이썬은 기본 정렬 라이브러리인 sorted() 함수를 제공한다. sorted()는 퀵 정렬과 동작 방식이 비슷한 병합 정렬을 기반으로 만들어 졌는데, 병합 정렬은 일반적으로 퀵 정렬보다 느리지만 최악의 경우에도 시간 복잡도 O(NlogN)을 보장한다는 특징이 있다.

사실 정렬 라이브러리는 이미 잘 작성된 함수로 우리가 직접 퀵 정렬을 구현할 때보다 더욱 효과적이다. 앞서 파이썬은 병합 정렬 기반이라 했지만 정확히는 병합 + 삽입 정렬의 아이디어를 더한 하이브리드 방식의 알고리즘을 사용하고 있다. 책에서 자세히 다루지는 않지만, 문제에서 별도의 요구가 없다면 단순히 정렬해야 하는 상황에서는 기본 정렬 라이브러리를 사용하고, 데이터의 범위가 한정되어 있으며 더 빠르게 동작해야 할 때는 계수 정렬을 사용하자.

코딩 테스트에서 정렬 알고리즘이 사용되는 경우를 일반적으로 3가지 문제 유형으로 나타낼 수 있다.

1. **정렬 라이브러리로 풀 수 있는 문제**
2. **정렬 알고리즘의 원리에 대해 물어보는 문제**
3. **더 빠른 정렬이 필요한 문제** : 퀵 정렬 기반의 정렬 기법으로는 풀 수 없으며 계수 정렬 등의 다른 정렬 알고리즘을 이용하거나 문제에서 기존에 알려진 알고리즘의 구조적 개선을 거쳐야 풀 수 있다.

문제 1. 위에서 아래로

하나의 수열에는 다양한 수가 존재한다. 이러한 수는 크기에 관계없이 나열되어 있다. 이 수를 큰 수부터 작은 수의 순서로 정렬해야 한다. 수열을 내림차순으로 정렬하는 프로그램을 만들어라.

```
import sys

N = int(sys.stdin.readline())

arr = []
for _ in range(N):
  arr.append(int(sys.stdin.readline()))

arr.sort(reverse=True)

for data in arr:
  print(data, end=' ')
```

💡 문제 2. 성적이 낮은 순서로 학생 출력하기

N 명의 학생 정보가 있다. 학생 정보는 학생의 이름과 학생의 성적으로 구분된다. 각 학생의 이름과 성적 정보가 주어졌을 때 성적이 낮은 순서대로 학생의 이름을 출력하라.

성적이 동일할 경우 자유롭게 출력한다.

💡 문제 3. 두 배열의 원소 교체

동빈이는 2개의 배열 A와 B를 가지고 있다. 두 배열은 N개의 원소로 구성되어 있으며, 배열의 원소는 모두 자연수이다. 동빈이는 최대 K번의 바꿔치기 연산을 수행할 수 있는데, 바꿔치기 연산이란 배열 A에 있는 원소 하나와 배열 B에 있는 원소 하나를 골라 두 원소를 바꾸는 것을 말한다. 동빈이의 최종 목표는 배열 A의 모든 원소의 합이 최대가 되도록 하는 것이며, 여러분은 동빈이를 도와야 한다.

N, K, 그리고 배열 A, B 의 정보가 주어졌을 때, 최대 K번의 바꿔치기 연산을 수행하여 만들 수 있는 배열 A의 모든 원소의 합의 최대값을 출력하는 프로그램을 작성하라.

```
import sys

N, K = map(int, sys.stdin.readline().strip().split())
A = list(map(int, sys.stdin.readline().strip().split()))
B = list(map(int, sys.stdin.readline().strip().split()))

A.sort()
B.sort(reverse=True)

for i in range(K):
    if A[i] < B[i]:
        A[i], B[i] = B[i], A[i]

print(sum(A))
```