

ICPC Sinchon



2022 Winter Algorithm Camp

1주차. 시간복잡도와 정렬 - 서강대학교 김성현

2022 Winter Algorithm Camp

1주차. 시간복잡도와 정렬

목차

1. 강사 소개, 강의 소개
2. 강의의 방향
3. 알고리즘의 개념과 발상
4. 알고리즘의 효율과
시간복잡도
5. 정렬과 시간복잡도
6. 정렬의 의미

2022 Winter Algorithm Camp

강사 소개

* 김성현(dart)

- 서강대학교 기계공학과 / 컴퓨터공학과
- 2021 겨울 신촌지역 대학교 프로그래밍 동아리 연합 알고리즘 캠프 초급반 멘토
- 2021 여름 신촌지역 대학교 프로그래밍 동아리 연합 알고리즘 캠프 콘테스트 초급 3등

2022 Winter Algorithm Camp

강의 커리큘럼 소개

1회차 시간복잡도, 정렬

2회차 문자열

3회차 DP

4회차 그리디

5회차 스택, 큐, 덱

6회차 백트래킹, 완전탐색

7회차 분할정복, 이분탐색

8회차 그래프, 그래프 순회

9회차 트리, PQ, BST

10회차 다익스트라, 벨만포드, 플로이드 알고리즘

11회차 UF, MST

강의의 방향

* 강의의 대상

- 프로그래밍 언어를 하나 이상 알고 있지만 알고리즘에 대해 모르는 사람
- 초급 알고리즘에 대한 기본적인 지식이 있으나 응용이 잘 되지 않는 사람

* 강의의 목적

- 초급 알고리즘에 대한 지식 전달
- 단순한 지식을 넘어 그 지식을 적절한 상황에 응용할 수 있는 감각을 전달
- '문제 해결'로서 알고리즘 다루기

2022 Winter Algorithm Camp

알고리즘과 문제 해결

* 알고리즘이란 무엇인가?

- 문제를 해결하는 방법이다.
- 그것이 우리에게 구체적으로 어떤 의미를 가지는가?

알고리즘과 문제 해결

* 알고리즘이란 무엇인가?

- 문제를 해결하는 방법이다.
- 그것이 우리에게 구체적으로 어떤 의미를 가지는가?

* 프로그래밍을 하는 자의 알고리즘 - 단순한 절차들의 결합으로 문제 해결

- 작은 것이라도 우리는 언제나 문제 해결 속에서 살고 있음
- 지도를 보고 길을 찾는 것과, 그것을 특정 방향으로 일정 거리만큼씩 이동하는 것의 연속으로 서술하는 것은 아주 다른 일
- 우리가 너무 자연스럽게 하나의 덩어리로 생각하는 문제 해결을 해체하여 더 명확하고 단순하게 서술하고, 그것을 컴퓨터가 이해할 수 있을 만큼 작은 단위의 연산들과 구체적인 절차들의 연속으로 나타내는 것
- 그 절차들을 수행할 때의 효율과 예외 상황들에 대한 고려도 필요

문제 해결의 예시 - 이름 찾기

* 캠프 초급반 슬랙에서 '김성현'을 찾으려면?

- 내가 직접 한다면 어떻게 할까?
- 단순하고 당연한 접근 : 약 200개의 이름을 다 검토하면 찾을 수 있을 것
- 컴퓨터에게 이를 어떻게 지시할 것인가?

문제 해결의 예시 - 이름 찾기

* 캠프 초급반 슬랙에서 '김성현'을 찾으려면?

- 내가 직접 한다면 어떻게 할까?
- 단순하고 당연한 접근 : 약 200개의 이름을 다 검토하면 찾을 수 있을 것
- 컴퓨터에게 이를 어떻게 지시할 것인가?

* 문제 상황을 추상적으로, 해결 방식은 순차적/구체적으로

- 문제의 상황을 더 추상적이고 일반적인 상황으로 모델링 : 주어진 데이터 중 특정 데이터와 같은 것이 있는지 찾는 문제
- 모든 데이터를 하나씩 순차적으로 검토하면서, 찾는 데이터와 같은지 대조하여 풀기
- 이름들 사이에서 김성현 찾기 -> 모든 데이터를 순차적으로 보면서 찾는 데이터와 대조
- 예외의 고려도 필요 - 동명이인에 대한 처리 등 (다루지는 않겠지만 이런 부분의 사고가 필요하다는 것)
- 그럼 이 방법은 얼마나 효율적일까?

알고리즘의 성능 따지기

* 알고리즘의 성능을 어떻게 측정할까?

- 알고리즘을 구현한 코드가 특정 입력에 대해 수행되는 시간을 측정해볼 수 있음
- 만약 한 코드가 다른 코드보다 느리다면 더 안 좋은 알고리즘이라고 추측 가능
- 단순한 시간 측정은 실험 환경, 최적화의 차이, 코드 구현의 차이 등 여러 변수 존재

알고리즘의 성능 따지기

* 알고리즘의 성능을 어떻게 측정할까?

- 알고리즘을 구현한 코드가 특정 입력에 대해 수행되는 시간을 측정해볼 수 있음
- 만약 한 코드가 다른 코드보다 느리다면 더 안 좋은 알고리즘이라고 추측 가능
- 단순한 시간 측정은 실험 환경, 최적화의 차이, 코드 구현의 차이 등 여러 변수 존재

* 알고리즘 성능에 대한 객관적인 척도 - 계산량

- 알고리즘을 수행할 때 몇 번의 계산이 발생하는지
- 같은 데이터에 대해 알고리즘을 수행할 때 더 적은 계산이 필요하면 더 좋은 알고리즘

알고리즘의 성능 따지기

* 계산량을 어떻게 따질 것인가?

- 앞에서 다룬 이름 찾기 문제
- 가나다순으로 정렬되어 있는 초급반 수강생 데이터에서 '김성현'을 찾을 땐 상대적으로 적은 데이터만 검토
- '임지환' 이나 '한다현'을 찾을 때는 더 많은 데이터의 검토(즉, 더 많은 계산) 필요

* 알고리즘의 계산량 나타내기 - 최악의 경우에 대하여

- 임의의 n 개의 데이터에 대해서 알고리즘 수행시 발생할 수 있는 최대의 계산량
- n 개의 데이터를 검토하고, 각 데이터의 검토가 1번의 계산이라 하면 최악의 경우 n 번 계산 필요
- 만약 다른 누군가가 짠 알고리즘은 $\log(n)$ 번의 계산만 필요하다면 그 알고리즘이 더 좋다

시간복잡도

* 계산량을 좀더 일반적으로 표현할 수는 없을까?

- n 개의 데이터에 대해 작동하는 알고리즘에서 n 번, $n+3$ 번, $n/2$ 번 등의 계산을 구분하는 건 의미있는 일이지만 번거로움
- 비슷한 계산량들을 묶어서 표현할 수 있는 방법 필요
- 빅 오 표기법의 등장

* Big-O Notation

- $f(n)$ 이 $O(g(n))$ 이라면 어떤 양수 n_0 , c 가 있어 n_0 이상의 모든 n 에 대해 $f(n) \leq c * g(n)$
- ex) $2n^2 + 3n + 3 = O(n^2)$
- 계수를 무시하고 계산량에 가장 큰 영향을 미치는 가장 높은 차수의 항만 따지는 것
- 계산량이 $10000n$ 이라도 $O(n)$
- 최고차항 미만의 항과 상수 계수는 무시되지만 실제로는 영향을 미치기도 함(소위 '상수 커팅')

시간복잡도의 활용

* 각각의 시간복잡도는 어느 정도의 성능인가?

- 알고리즘 문제를 풀 땐 보통 컴퓨터가 1초에 1억번 정도의 연산이 가능하다고 생각
- 예를 들어 $O(n^2)$ 는 1초에 1만 개 정도의 입력을 처리 가능
- 단순한 연산은 약 5억 번 정도까지도 가능하지만 일반적인 경우 문제도 1초에 약 1억 번,
넉넉하게는 약 1천만 번의 연산으로 문제를 해결 가능하도록 문제가 출제됨 : 컴퓨팅 비용이 비싼 연산을 사용하거나 입력/출력에 필요한 시간 등이 있기 때문

- 그 범위 내에 있는 계산 횟수를 이용하여 문제를 풀어내는 것이 우리의 목적


* 문제 입력의 범위와 알고리즘의 시간복잡도를 고려하여 문제를 풀어야 함

- 문제의 입력이 최대 10만개까지 들어오면 $O(n^2)$ 로 푸는 것이 정해가 아닐 것
- 문제에서 요구되는 시간복잡도보다 더 느린 풀이법을 먼저 떠올리는 것도
좋지만 최적화를 해야 한다는 사실을 알고는 있어야 함

기초 알고리즘 - 정렬

* 주어진 데이터를 손으로 정렬해야 한다면?

- 가장 작은 것부터 하나씩 뽑아 가면서 순서대로 나열하는 것 : 선택 정렬의 아이디어
- 정렬된 부분을 점진적으로 늘려 나가는 삽입 정렬도 직관적이다
- 모든 원소를 다 보아야 하고 각 원소를 다른 모든 원소들과 비교해야 하기 때문에 둘 다 $O(n^2)$



5	1	3	7	2	9
1	5	3	7	2	9
1	2	3	7	5	9
1	2	3	7	5	9
1	2	3	5	7	9
1	2	3	5	7	9

* 다른 방법 - 퀵 소트, 머지 소트

- $O(n \log n)$ 으로 정렬할 수 있다
- 같은 문제를 푸는 다양한 방법이 존재
- 중요한 것은 그 중에 현재 상황에 맞고 효율적인 것을 택하는 것

기초 알고리즘 - 정렬

* 주어진 데이터를 손으로 정렬해야 한다면?

- 가장 작은 것부터 하나씩 뽑아 가면서 순서대로 나열하는 것 : 선택 정렬의 아이디어
- 정렬된 부분을 점진적으로 늘려 나가는 삽입 정렬도 직관적이다
- 모든 원소를 다 보아야 하고 각 원소를 다른 모든 원소들과 비교해야 하기 때문에 둘 다 $O(n^2)$

* 다른 방법 - 퀵 소트, 머지 소트

- $O(n \log n)$ 으로 정렬할 수 있다
- 같은 문제를 푸는 다양한 방법이 존재
- 중요한 것은 그 중에 현재 상황에 맞고 효율적인 것을 택하는 것

```
sort(arr, arr + n);
```


정렬의 의미

* 데이터들을 정렬한다는 것은 어떤 의미를 가지는가?

- 규칙이 없었던 데이터들에 순서와 방향을 부여하는 것
 - 문제 해결에 필요한 과정에서 '순서대로 무엇인가를 해야 할' 때
 - 꼭 크기 순서는 아니어도 된다
 - 순서에 대해 다루는 방법이 정렬만 있는 것은 아니지만...

정렬의 의미

* 데이터들을 정렬한다는 것은 어떤 의미를 가지는가?

- 규칙이 없었던 데이터들에 순서와 방향을 부여하는 것
 - 문제 해결에 필요한 과정에서 '순서대로 무엇인가를 해야 할' 때
 - 꼭 크기 순서는 아니어도 된다
 - 순서에 대해 다루는 방법이 정렬만 있는 것은 아니지만...

* 순서에 아무 상관없이 모든 경우를 탐색하는 문제는 상대적으로 적음

- 문제의 상황에서 가장 작은 것부터, 가장 큰 것부터 등등 특정한 순서대로 처리하는 것이 효율적인지를 따지기
 - 문제를 효율적으로 해결할 수 있는 방향으로 데이터들을 정렬한 후 그 순서로 처리
 - 정렬만이 주가 되는 문제는 적지만 쓰이는 문제는 아주 많다 : 문제 해결 절차를 생각하는 연습 필요

BOJ 18870 : 좌표 압축

문제

수직선 위에 N 개의 좌표 X_1, X_2, \dots, X_N 이 있다. 이 좌표에 좌표 압축을 적용하려고 한다.

X_i 를 좌표 압축한 결과 X'_i 의 값은 $X_i > X_j$ 를 만족하는 서로 다른 좌표의 개수와 같아야 한다.

X_1, X_2, \dots, X_N 에 좌표 압축을 적용한 결과 X'_1, X'_2, \dots, X'_N 를 출력해보자.

제한

- $1 \leq N \leq 1,000,000$
- $-10^9 \leq X_i \leq 10^9$

예제 입력 1 복사

```
5
2 4 -10 4 -9
```

예제 출력 1 복사

```
2 3 0 3 1
```

BOJ 18870 : 좌표 압축

문제

수직선 위에 N 개의 좌표 X_1, X_2, \dots, X_N 이 있다. 이 좌표에 좌표 압축을 적용하려고 한다.

X_i 를 좌표 압축한 결과 X'_i 의 값은 $X_i > X_j$ 를 만족하는 서로 다른 좌표의 개수와 같아야 한다.

X_1, X_2, \dots, X_N 에 좌표 압축을 적용한 결과 X'_1, X'_2, \dots, X'_N 를 출력해보자.

- 모든 좌표에 대하여 그보다 작은 좌표의 개수를 세어 가면서 출력하는 접근 가능
- n 개의 좌표 각각에 대해서 $n-1$ 개의 다른 좌표들을 검토해야 하므로 $O(n^2)$
- n 이 최대 100만까지이므로 $O(n^2)$ 으로는 시간 초과 발생
- 접근 자체는 맞음. 어떻게 특정 좌표보다 작은 좌표의 개수를 빠르게 셀 수 있을까?

BOJ 18870 : 좌표 압축

문제

수직선 위에 N 개의 좌표 X_1, X_2, \dots, X_N 이 있다. 이 좌표에 좌표 압축을 적용하려고 한다.

X_i 를 좌표 압축한 결과 X'_i 의 값은 $X_i > X_j$ 를 만족하는 서로 다른 좌표의 개수와 같아야 한다.

X_1, X_2, \dots, X_N 에 좌표 압축을 적용한 결과 X'_1, X'_2, \dots, X'_N 를 출력해보자.

- 만약 좌표가 크기 순으로 정렬되어 있다면 가장 작은 것부터 번호를 매겨주면서 순서대로 볼 수 있음
- 그렇게 매긴 순서를 원소 별로 저장해 주면 된다
- `map stl`, `pair` 배열 등을 사용
- 단순한 풀이로부터 시작해서 효율적인 방식을 찾아 나가기

sort STL : 다양한 순서로 정렬하기

* 비교 함수를 정의하기

```
sort(arr, arr + n, compare);
```

- sort 함수의 세번째 인자로 비교 함수를 전달할 수 있다
- 그러면 그 비교 함수대로 오름차순 정렬이 됨
- greater<>() 등 template으로 제공되는 함수도 있다

```
bool compare(coordinate a, coordinate b){  
    if(a.y==b.y){  
        return a.x<b.x;  
    }  
    return a.y<b.y;  
}
```

* 연산자 오버로딩을 사용하기

- C++에서는 구조체에 대해서도 연산자 오버로딩을 지원한다
- 따라서 부등호 < 에 대해서 비교의 조건을 다시 정의해 놓으면 정렬에도 적용된다
- 다른 방향의 부등호나, 같은 원소에 대한 처리 등은 알아서 이루어짐

sort STL : 다양한 순서로 정렬하기

* 연산자 오버로딩을 사용하기

- C++에서는 구조체에 대해서도 연산자 오버로딩을 지원한다
- 따라서 부등호 < 에 대해서 비교의 조건을 다시 정의해 놓으면 정렬에도 적용된다

다른 범함인 부등호나 다른 연산에 대한 함수도 만들어서 이루어짐

```
typedef struct coordinate{
    int x,y;

    bool operator<(coordinate other){
        if(y==other.y){
            return x<other.x;
        }
        return y<other.y;
    }
}coordinate;
```

- x,y 두 개의 좌표로 이루어진 구조체를 y좌표 기준으로,
y좌표가 같으면 x좌표를 기준으로 정렬하도록 연산자를 오버로딩한 코드