#stucy #algorithms

집합

집합이란?

- 집합은 순서도 중복도 없는 원소들을 갖는 자료 구조
  - {1, 6, 6, 4, 3} 은 { 6 } 이 중복이므로 집합이 아니다
  - {1, 6, 4, 3} 은 {6, 1, 4, 3} 과 같은 집합이다. 왜냐하면 집합은 순서를 따지지 않기에 중복 없이 원소가 동일하면 같은 집합으로 분류한다

상호 배타적 집합

- 상호 배타는 서로 겹치는 원소가 없는 집합 관계를 말한다. 즉, A와 B 집합이 있고 A와 B 집합 간에 공통 원소가 없으면 상호 배타적 집합이다
- 상호 배타적 집합 특성을 활용한 작업
  - 이미지 분할, 도로 네트워크 구성, 최소신장트리, 게임 개발, 클러스터링 작업 등

어떻게 구현하는 가?

- 단순하게 표현하면 트리를 배열로 표현하는 것과 유사하게 만든다
- 구현 방식
  - 집합에서 가장 큰 원소값 기준으로 배열 크기 설정(max_index == max_element_value)
  - 집합에서 존재하는 원소값과 동일한 배열 인덱스 값에 해당 원소값을 넣는다. 그리고 원소값이 없는 배열 인덱스에는 집합에는 없는 원소값을 넣어서 구분한다.
  - 최상위 노드에 들어갈 기준값을 집합에서 뽑는다. 그리고 해당 값과 간선으로 이어질 자식 노드의 값과 동일한 배열의 인덱스 위치에 부모 노드의 값을 넣는다.
  - 마찬가지로 그 후의 자식 노드도 자기 부모 노드의 값을 삽입, 삽입, 삽입 ...
  - 위와 같이 하면 연결된 집합과 연결되지 않은 집합을 구분할 수 있다

유니온 - 파인드 알고리즘

- 상호 배타적 집합의 특징을 이용하여 노드 간 연결 관계를 규명하고 특징을 찾는 알고리즘이다
- 파인드(find) 연산
  - 특정 노드의 최상위 노드를 탐색하는 방법

```
// node는 내가 찾고자 하는 특정 노드값이다.
int find(int node) {
    // 특정 노드에 위치한 값과 인덱스 값이 같으면 최상위 노드다
    if (arr[node] == node) {
        return node;
    }
```

```
    // 최상위 노드를 찾기 위해 부모 노드값으로 탐색을 한다. 그리고 찾은 최상위 노드 값을 넣는다
    // 만약 바로 최상위 노드를 저장하지 않고 부모 노드값을 남겨 놓아야 할 이유가 있으면 지역변수에 값을 저장하고 해당 값을 반환할 것
    arr[node] = find(arr[node]);

    // 최상위 노드 값을 반환
    return arr[node];
}
```

- 유니온(union) 연산
  - 두 집합을 합친다. 즉, 두 집합의 최상위 노드를 동일한 값으로 조정하는 것. 두 집합 중에서 어느 노드 값을 동일한 값으로 만들 지는 규칙과 조건에 따라 다르다.

```
// 첫번째 집합 노드값과 두번째 집합 노드값을 매개변수로 받는다.
void union(int first_set_node, int second_set_node) {
    // find 함수를 통해 각 집합의 최상위 노드 값을 찾는다.
    int first_set_root_node = find(first_set_node);
    int second_set_root_node = find(second_set_node);

    // 두 집합의 최상위 노드가 일치하지 않는 경우, 한 쪽의 값으로 최상위 노드를 변경한다.
    if (first_set_root_node != second_set_node) {
        arr[second_set_second_node] = first_set_root_node;
    }
}
```

풀어보자

- 폰켓몬, https://school.programmers.co.kr/learn/courses/30/lessons/1845
- 섬 연결하기, https://school.programmers.co.kr/learn/courses/30/lessons/42861

정렬

정렬이란?

- 사용자가 정의한 순서대로 데이터를 나열하는 것. 그리고 이 나열을 어떻게 하면 효율적(주로, 시간복잡도 측면에서)으로 나열할 수 있을 지 고민한다.

삽입 정렬(insertion sort)

- 가장 기본적인 정렬 방식
- 정렬을 정렬된 영역과 정렬되지 않은 영역으로 구분하고 정렬되지 않은 영역에서 원소를 하나씩 정렬된 영역의 값과 비교하여 적절한 위치에 삽입하는 것
- 시간 복잡도 : 최대 O(N^2). 단, 배열로 할 때는 중간에 삽입하면서 원소들을 밀어야 하므로 O(N^3)이다.

```
// 연결리스트를 사용한다고 가정한다
void insertion_sort() {
    // 정렬되지 않은 영역을 탐색
    for (int i = 0; i < arr.size(); i++) {
        // 정렬되지 않은 원소를 넣기 위해 정렬된 영역에서 적절한 위치를 탐색
        for (int j = 0; j < i; j++) {
            if (arr[i] < arr[j]) {
                // 여기서 push는 (넣을 값, 넣을 위치)
                arr.push(arr[i], j);
                // 값을 찾아 넣었으니 탈출하고 다음 정렬되지 않은 원소를 탐색
                break;
            }
        }
    }
}
```

병합 정렬(merge sort, divide and conquer sort)

- 정렬되지 않은 영역을 쪼개서 각각의 영역을 정렬하고 다시 이를 합치며 정렬한다. 즉, 절반 씩 쪼개가며 원소 배열을 분할한 다음에 더이상 쪼갤 수 없을 때까지 가면 그 때부터 합치면서 정렬해 나간다.
- 시간복잡도 : 분할은 원소를 절반씩 쪼개므로 O(log N)이고 정복은 주어진 원소 배열에서 정렬을 수행하므로 O(N)이다. 즉, O(N log N)이다
- 병합 정렬의 단점은 재귀를 이용하기에 재귀 깊이가 너무 깊으면 메모리 문제가 발생할 수 잇다.

```
// 정렬을 해야 할 원소가 모여 있는 배열 arr 가정

// divide는 원소를 분할하는 과정이다.
void divide(int left, int right) {
    // 왼쪽 원소 번호와 오른쪽 원소 번호의 가운데 기준으로 짜르는 데 왼쪽 원소 번호가 더 커지면 문제가 있는 거다
    if (left >= right)
        return ;

    // 분할할 지점을 설정
    int mid = (left + right) / 2;

    // 먼저 왼쪽부터 쪼개 나가고 왼쪽을 다 쪼개면 오른쪽을 쪼개기 시작한다
    divide(left, mid);
    divide(mid + 1, right);

    // 쪼갤 수 있는 데 까지 쪼개고 나면 그 때부터 정복 시작
    conquer(left, mid, right);
}

// conquer는 정복 과정으로 실질적으로 원소들을 정렬한다
void conquer(int left, int mid, int right) {
    int l = left;
```

```
    int r = mid + 1;
    int i = l;
    int tmp[right - left];

    // 중앙값을 기준으로 정렬을 한다
    while (l <= mid || r <= right) {
        // 중앙값 기준 오른쪽 원소를 다 썼거나 왼쪽 원소가 오른쪽 원소보다 작으면 임시 정렬에 넣는다. 아니면 오른쪽 원소를 임시 정렬에 넣는다.
        if (r > right || (l < mid && arr[l] < arr[r])) {
            tmp[i] = arr[l];
            l++;
        } else {
            tmp[i] = arr[r];
            r++;
        }
        i++;
    }

    // 임시 정렬에 넣은 값들을 배열 위치에 맞게 넣는다.
    for (int i = left; i <= right; i++) {
        arr[i] = tmp[i];
    }
}
```

힙 정렬(heap sort)

- 힙이라는 자료 구조를 이용하는 정렬
- 힙
  - 부모 노드와 자식 노드 간의 대소 관계를 이용한 이진 트리
  - 최대 힙과 최소 힙으로 나뉘며 최대 힙은 부모 노드가 무조건 자식 노드보다 커야하고 최소힙은 부모 노드가 무조건 자식 노드보다 작은 값이어야 한다.
- 정렬 방법(최대 힙 기준)
  - 현재 노드와 자식 노드 값 비교
  - 현재 노드의 값이 자식 노드보다 가장 크지 않으면 자식 노드 중 가장 큰 값과 바꾼다
  - 바꾼 자식 노드 위치를 현재 노드로 한다.
  - 만약, 현재 노드에서 비교했을 때 자식 노드보다 크면 연산 종료
- 시간 복잡도 : 정렬되지 않은 값 N개로 이루어진 이진 트리의 탐색은 log N이다. 거기에 참조할 데이터는 최대 N개이므로 O(N log N)

```
// 여기서 사용하는 heap은 이진 트리 배열로 가정한다

void heap_sort(int heap_size) {
    for (int tree_size = heap_size - 1; tree_size >= 0; tree_size--) {
        heapify(heap_size);
        swap(tree_size, 0);
    }
```

```
}

void heapify(heap_size) {
    int last_parent = heap_size / 2 - 1;

    for (int cur = last_parent; cur >= 0; cur--) {
        while (cur <= last_parent) {
            int child = cur * 2 + 1;
            int sibling = child + 1;
            if (sibling < heap_size && heap[child] < heap[sibling])
                child = sibling;
            if (heap[cur] < heap[child]) {
                swap(cur, child);
                cur = child;
            } else {
                break;
            }
        }
    }
}

void swap(int under, int top) {
    int value = heap[top];

    heap[top] = heap[under];
    heap[under] = value;
}
```

우선순위 큐(priority queue)

- 우선 순위에 따라 데이터를 차출하는 큐. 즉, 우선순위에 맞춰서 원소 삽입부터 대기열을 조정하는 것이다.
- 우선순위 큐는 힙 구조로 동작한다. 왜냐하면 우선순위를 크기라고 생각했을 때, 최대힙은 자식 노드가 무조건 부모보다 크기가 작으므로 큰 값일 수록 상위에 위치하기 때문이다.
- 우선순위 큐 구현
  - 우선순위 기준을 숫자로 체계화
  - 체계화한 순위를 기준으로 힙 구조를 만들고 정렬한다.

위상 정렬(topological sort)

- 일의 순서가 있는 작업을 순서에 맞춰서 정렬
- 위상 정렬 구현
  - 위상 정렬은 자신을 향한 간선(즉, 부모 노드의 개수)을 진입 차수로 정의한다.
  - 진입 차수가 0인 작업을 전부 큐에 넣고 하나씩 작업을 끝내며 줄인다.
  - 작업을 줄일 때마다 해당 작업 노드와 연결된 자식 노드들의 진입 차수를 줄인다.
  - 그 다음 진입 차수가 0인 노드를 넣는다.

    - 위 과정을 반복하여 모든 노드가 차수가 0이 되면 작업을 행한 순서대로 노드를 늘어놓는다.
- 시간 복잡도 : 위상 정렬은 모든 정점(v, node)과 간선(e)을 단 한 번만 지난다. 그렇기에 O(|V| + |E|)

계수 정렬(counting sort)

- 데이터의 빈도수를 기준으로 정렬한다. 각 데이터마다 동일한 값이 몇 번 나오는 지 검사한 뒤에 많이/적게 나온 순서대로 값을 정렬.
- 시간 복잡도 : 데이터를 세는 건 N번, 빈도수 배열은 최솟값과 최댓값 범위로 따지므로 해당 범위를 K라 하면 K번. 즉, O(N + K).

풀어보자

- 문자열 내 마음대로 정렬하기, https://schoool.programmers.co.kr/courses/30/lessons/12933
- 가장 큰 수, https://school.programmers.co.kr/courses/30/lessons/42746
- (난이도 상) 지형 이동, https://school.programmers.co.kr/courses/30/lessons/62050