SW Certi. Pro 대비 SRG

Orientation

05/07/2020

Intro

What to know

- 최적화
  - 시간 / 공간복잡도
  - 최적화 기법
- 자료구조
  - 링크드리스트, 트리 (이진 트리 포함), 해시
- 알고리즘
  - 정렬, 탐색, 분할정복, 메모이제이션

Intro

What to learn today

- 시간 / 공간 복잡도
- 비트마스크
- 포인터 / 구조체
- 자료구조 / 알고리즘 맛보기
- 구간트리 (Segment tree)

Intro

SW Certi. Pro

- main과 solution 이렇게 2개의 파일로 구성 (우리는 solution만 작성)
- 입출력, 라이브러리 사용할 일 없음 (malloc.h만 예외) - 이것도 안 씀
- 프로그램을 다 짜는게 아니라 함수를 짠다
- 문제 내에 모든 제약조건이 명시되지 않으므로, 주어지는 코드를 분석해야 함
- 요즘은 구현량을 늘리는 추세 -> 연습 많이 해야 할 듯

시간/공간 복잡도

개괄

- 시간 : 50개 테스트케이스를 합쳐서 C++의 경우 10초 / Java의 경우 10초
- 메모리 : 힙, 정적 메모리 합쳐서 256MB 이내, 스택 메모리 1MB 이내

- 시간 : 50가지 케이스가 10초 -> 1 케이스당 0.2초
- 공간 : 스택 1MB, 메모리 256MB

- 어디까지 가능한지 아는 방법

시간/공간 복잡도

시간복잡도

- 입력에 크기와 문제해결 시간의 상관관계
- 대략적인 느낌만 가져가자
- 시간복잡도를 먼저 따진 후 풀자

- Pro 시험은 N을 log N으로 낮추는 시험
- 자료구조 / 알고리즘마다 복잡도 알기

시간복잡도	예시	N = 10000	1초 컷
O(N^2)	기본 정렬	1억	10,000
O(N log N)	빠른 정렬	14만	1,000,000
O(N)	선형탐색	1만	1억
O(log N)	이진탐색 / 힙	14	Pro 뿌셔
O(1)	해시	1	Pro 뿌셔

시간/공간 복잡도
공간복잡도

- 스택메모리
  - 터뜨릴 일이 거의 없음
  - 재귀를 잘못 쓰는 경우
  - 지역변수를 무리하게 많이 쓰는 경우
- 메모리
  - 마찬가지로 터뜨릴 일이 거의 없음
  - 그래도 계산은 하자

자료형	N = 1백만	N = 1천만
char	1 MB	10 MB
short	2 MB	20 MB
int(long)	4 MB	40 MB
long long	8 MB	80 MB

시간/공간 복잡도

예시

자료구조	랜덤 원소 삽입/삭제	맨 뒤에 원소 삽입/삭제	N번째 원소 접근
배열	O(N)	O(N)	O(1)
이중 링크드리스트	O(1)	O(1)	O(N)
동적배열	O(N)	O(1)	O(1)

자료구조	원소 삽입	원소 삭제	메모리
힙	O(log N)	O(log N)	N
해시	O(1)	O(1)	10N 이상

비트마스크

- 정수의 이진수 표현을 자료구조로 이용

- O(1)의 수행시간 - 빠르다
- 간결한 코드
- 적은 메모리 사용량

비트마스크

비트연산 리뷰

- 비트연산자 & | ^ ~ >> <<
- 비교연산자보다 낮은 우선순위
- >> 연산의 경우 자료형에 따라 최상위비트의 처리가 달라질 수 있으므로 주의

- dir = ++dir & 3;
- dir = --dir & 3;

비트마스크

예시

- 원소 추가 : array |= (1 << p);
- 원소 포함여부 : if (array & (1 << p))
- 원소 삭제 : array &= ~(1 << p);
- 원소 토글 : array ^= (1 << p);

- 합집합 : (a | b)
- 교집합 : (a & b)
- 차집합 : (a & ~b)
- 둘 중 하나만 포함된 집합 : (a ^ b)

포인터 / 구조체

포인터

- 포인터 연산자 * & (둘은 역함수)
- 2개 쓸 일은 Pro에선 없지 않을까요?

- int* p로 선언시 p는 포인터변수로 주소 저장
- *p로 p가 가르키는 주소에 저장된 값에 접근

```
void func(void)
{
    int n = 10;
    int * p;
    int * q = &n;
    p = &n;
    printf("%d %d\n", *p, *q);
}
```

포인터 / 구조체

구조체

- Pro시험의 필수요소
- 멤버참조연산자 . ->
- 현재 포인터면 -> 아니면 .
- ptr->key == (*ptr).key

- 구조체는 선언 시 중괄호 끝에 ;을 붙인다
- 변수 선언하면서 초기화는 예시 참조

```
struct NODE {
    int key;
    struct NODE * ptr;
};

void func(void)
{
    struct NODE p = (struct NODE) {1, NULL};
    struct NODE q = {.key = 2, .ptr = NULL};
    struct NODE * r = &p;

    p.ptr = &q;

    printf("%d %d\n", p.key, r->key);
    printf("%d %d\n", p.ptr->key, r->ptr->key);
}
```

자료구조 맛보기
링크드리스트

```
struct NODE {
    int key;
    NODE * next;
};

int node_idx = 0;  // 매 tc마다 초기화 필수
NODE a[MAX_NODE];

NODE * newNode(int key)
{
    return &(a[node_idx++] = {.key = key, .next = NULL});
}
```

```
void func(void)
{
    NODE * p = newNode(2);
    NODE * q = newNode(1);
    q->next = p;
    p = q;

    printf("%d %d\n", p->key, p->next->key);
    printf("%d %d\n", a[0].key, a[1].key);
}
```

자료구조 맛보기
이진탐색트리 (BST)

```
struct NODE {
    int key;
    NODE * left, * right;
};

int node_idx = 0;  // 매 tc마다 초기화 필수
NODE a[MAX_NODE];

NODE * newNode(int key)
{
    return &(a[node_idx++] = \
                {.key = key, .left = NULL, .right = NULL});
}
```

```
void addNode(NODE * p, int key)
{
    if (p == NULL)  // if ( ! p )
    {
        return p = newNode(key);
    }
    if (N < p->value) { p->left = addNode(p->left, key); }
    else { p->right = addNode(p->right, key); }
    return p;
}

// NODE * root = addNode(NULL, key); // 처음 1번만 실행
// addNode(root, key);                          // 이후엔 계속 이렇게

// 더 방어적으로 코딩하려면
// if ( !findKey(root, key) ) { addNode(root, key); }
```

알고리즘 보기

병합정렬 (Merge Sort)

```
int a[MAX_N], b[MAX_N];

void merge_sort(int * a, int * b, int s, int e)
{
    int m = s + (s + e) / 2
    if (s + 1 >= e) return;

    merge_sort(a, b, s, m);
    merge_sort(a, b, m, e);

    merge(a, b, s, m, e);
}
```

```
void merge(int * a, int * b, int s, int m, int e)
{
    int i, j, k;
    i = k = s;
    j = m;

    while (i < m && j < e)
    {
        if (a[i] < a[j]) { b[k++] = a[i++]; }
        else { b[k++] = a[j++]; }
    }
    while (i < m) { b[k++] = a[i++]; }
    while (j < e) { b[k++] = a[j++]; }

    for (k = s; k < e; ++k) { a[k] = b[k]; }
}
```

구간트리 (Segment Tree)

왜 쓸까?

- 크기가 N인 배열에서 구간 [x, y]의 합을 구하려면? O(N) 전처리를 하면 O(1)
- 배열의 원소를 하나 바꾸기 + 구간을 바꿔 합을 구하기를 M번 반복하면? O(MN)
- 여기서 M = N 이라면? O(N^2) -> Pro 탈락

- 전처리에 O(N)을 사용해서 구간트리를 만들면
- 구간 합 구하기와 갱신이 각각 O(log N)
- 따라서 문제를 O(N log N)에 풀 수 있다!

구간트리 (Segment Tree)
구현해보자

```
int a[N];
int tree[4 * N];

int init(int * a, int * tree, int node, int s, int e)
{
    if (s == e) { return tree[node] = a[s]; }
    else
    {
        int m = (s + e) / 2;
        return tree[node] = init(a, tree, node*2, s, m) \
                          + init(a, tree, node*2 + 1, m + 1, e);
    }
}
```

```
int find_sum(int * tree, int node, int s, int e, int x, int y)
{
    int m = (s + e) / 2;
    if (x > e || y < s) { return 0; }
    if (x <= s && e <= y) { return tree[node]; }
    return find_sum(tree, node*2, s, m, x, y) \
         + find_sum(tree, node*2 + 1, m + 1, e, x, y);
}

void renew(int * tree, int node, int s, int e, int idx, int diff)
{
    if (idx < s || idx > e) { return; }
    tree[node] += diff;
    if (s != e)
    {
        int m = (s + e) / 2;
        renew(tree, node*2, s, m, idx, diff);
        renew(tree, node*2 + 1, m + 1, e, idx, diff);
    }
}
```