

Solving Talk

Solving Club

HOME Ta

'24년 하계 대학생 S/W 알고리즘 역량강화 과정



Club 지수 6272

205

개설일 : 2024-07-15 15:30

비공개

전체보기

Notice

Q&A

Free

Problem Box

Club Problem box 01(0)

클럽 탈퇴

Notice

목록

[24년 하계 대학생 S/W 알고리즘] 9. 구간트리



조회 610 작성일

안녕하세요, 하계 대학생 S/W 알고리즘 9강 주제는 Segment Tree/Sqrt Decomposition 입니다.
우리말로 구간트리 및 제곱근분할(평방분할)이라고도 부릅니다.

1. 기초 강의

이번 시간에는 기초 강의는 제공되지 않습니다. 학습 자료로 개인 학습을 진행하신 후, 학습 내용과 관련하여 질·
코드배틀의 Q&A 게시판을 이용해주세요.

※ 문제풀이는 수료조건에 반영됩니다.

2. 학습 자료

Segment Tree & SQRT Decomposition

Part1 구간트리

구간 트리 개요

데이터가 연속적으로 존재할 때, 특정 구간 데이터를 효율적으로 계산

데이터 집합이 주어지고, 특정 구간의 합, 최대, 최소 등을 구할 때 좋음.



- 이름에서 나타나듯 트리형태 자료구조입니다.

- point update와 range query를 모두 $O(\log N)$ 의 시간에 처리합니다.

- 이번 슬라이드는 **기초 세그먼트 트리에 대해서만 다룹니다.**

:여기서 말하는 기초 : 결합법칙이 성립하는 연산에 대한 range query를 처리하는 구간 트리

$$(a+b)+c = a+(b+c)$$

a	b	c
---	---	---

a	b	c
---	---	---

구간 합 구하기 예

배열의 구간 합을 구해보자. 예: 주어진 배열의 INDEX 6~9의 합은?

- 방법1 (선행적)

단순히 순차적으로 더한다.. 7+8+9+10 결과는 34. 시간 복잡도는 $O(N)$ 이다.

아주 쉽게 구할 수 있다는 장점이 있지만. 인덱스 1~9 구하기, 인덱스 2~4 구하기, 인덱스 3~9 구하기 등 반복 명령이

주어졌을 때마다 다시 재계산을 해야한다. 이 때 마다 계속 $O(N)$ 시간적으로 비효율적이다. ☺

• 방법2 (방법1을 개선한 선형 계산법)

i번째까지의 합을 저장하는 배열 arr를 하나 더 만들어 놓는다.

인덱스 1~9 구하기는 $cumsum[9] - cumsum[0]$

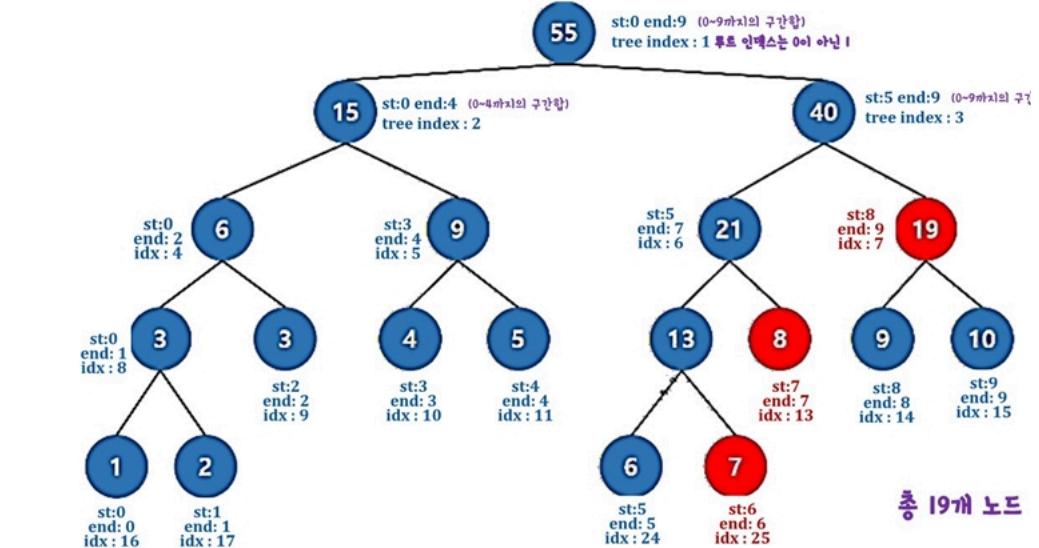
인덱스 2~4 구하기는 $cumsum[4] - cumsum[1]$

인덱스 6~9하기는 $cumsum[9] - cumsum[5] = 55 - 21 = 34$

방법1보다는 훨씬 빠르다. 하지만, 기존 배열의 원소가 갱신 되는 경우, 전체 재계산이 필요하여 시간이 많이 걸릴 수 있다.

▼ cumsum
1 3 6 10 15 21 28 36 45 5

```
for(int i = 1; i <= n; i++) {
    cin >> arr[i];
    sum[i] = sum[i-1] + arr[i];
}
```



• 방법3 (트리이용)

원래 범위를 반씩 분할하며 값을 저장한다.

- 루트에는 전체 노드를 더한 55. 리프를 제외한 모든 노드는 자식이 2개가 있다.

- 루트의 첫째 자식이자 두번째 노드 인덱스 0~4까지를 더한 값 15

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

- 루트의 둘째 자식이자 세번째 노드는 인덱스 5~9까지를 더한값 40

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

etc... 그럼 총 19개의 원소가 나올 것이다. **트리 노드 개수 != 배열 크기** 임을 다시 한 번 주의하자

이제 트리가 준비되어 데이터탐색을 $O(\log N)$ 에 할 수 있다. ☺

인덱스 구간 6~9의 합은 19 + 8 + 7 3개 노드 합으로 34이다.

구간 트리 특징 요약:

- Complete Binary Tree에 가까움
- (리프를 제외한) 모든 노드는 자식이 2명
- (리프를 제외한) 모든 노드는 왼쪽/오른쪽의 합이다 (구간합 구간트리니까)
- 트리 높이는 $O(\log N)$

구간 트리 구현 (구간합 찾기)

■ 트리 초기화

```
int init(int* tree, int node, int s, int e) {
    // s & e : arr의 시작 & 끝 인덱스
    if (s == e) return tree[node] = arr[s];

    // 재귀로 반씩 나누어 초기화
    int m = (s + e) / 2;
    int a = init(tree, node * 2, s, m);
    int b = init(tree, node * 2 + 1, m + 1, e);
    return tree[node] = a + b;
}

int main() {
    int N = sizeof(arr) / sizeof(arr[0]); // 배열 크기
    int segTree[N * 4] = {};
    init(segTree, 1, 0, N - 1);

    for (int i = 1; i < N * 4; i++) {
        printf("tree[%d] = %d\n", i, segTree[i]);
    }
}
```

- 트리는 데이터의 4배 크기의 배열로 만들기
- 정확히는 $\lfloor \frac{N}{2} \rfloor * 2$ 크기의 요구하지만 4배 해도 무방
- 데이터크기를 2 거듭제곱수로 올림한 후 두 배
- 말단 노드는 배열의 수 자체를 가리키게 된다.
- (말단 제외) 각 노드는 자식이 2명
- (말단 제외) 각 노드는 왼쪽 자식과 오른쪽 자식의 합
- 좌측자식은 2^k , 우측자식은 $2^k + 1$ ← 그래서 index 시작은 0이 아닌 1

tree[1] = 55	tree[11] = 5
tree[2] = 15	tree[12] = 13
tree[3] = 40	tree[13] = 8
tree[4] = 6	tree[14] = 9
tree[5] = 9	tree[15] = 10
tree[6] = 21	tree[16] = 1
tree[7] = 19	tree[17] = 2
tree[8] = 3	tree[18] = 0
tree[9] = 3	tree[19] = 0
tree[10] = 4	tree[20] = 0

구간 트리 구현 (구간합 찾기)

■ 쿼리 수행 (예: sum)

```
int query(int* tree, int node, int ts, int te, int qs, intqe) {
    // qs~qe : arr의 질의구간
    // ts~te : tree의 구간

    // 경우1 : 트리구간이 질의구간에 완전히 포함됨
    if (ts >= qs && qe >= te) return tree[node];

    // 경우2 : 트리구간과 질의구간이 전혀 겹치지 않음
    if (te < qs || qe < ts) return 0;

    // 경우3 : 일부 겹침
    int m = (ts + te) / 2;
    int a = query(tree, node * 2, ts, m, qs, qe);
    int b = query(tree, node * 2 + 1, m + 1, te, qs, qe);
    return a + b;
}
```

- 쿼리 수행 역시 재귀적으로 구현
- 질의 하는 범위안에 노드 범위가 완전히 쓱 포함될 트리가 값을 내주는 방식
- 포함 하지 못하면: 0을 리턴
- 부분적으로 포함 : 쓱 포함될 때까지 재탐색

```
query(segTree, 1, 1, N, 1, 5)
query(segTree, 1, 1, N, 2, 4)
query(segTree, 1, 1, N, 9, 10)
query(segTree, 1, 1, N, 7, 10)

sum of idx 0~4: 15
sum of idx 1~3: 9
sum of idx 8~9: 19
sum of idx 6~9: 34
```

실행결과 (일부 랜덤)

구간 트리 구현 (구간합 찾기)

■ 갱신 예: 3번째 ~ 6번째 원소 모두 +3씩 하라고 한다면?

```
int update(int* tree, int node, int s, int e,
          int ii, int value) {
    // ii : 수정할 인덱스
    // value : 더해서 갱신

    // 대상 범위 안에 있을 때만 갱신 진행
    if (s > ii || ii > e) return tree[node];
    tree[node] += value;
    if (s == e) return tree[node];

    int m = (s + e) / 2;
    int a = update(tree, node * 2, s, m, ii, value);
    int b = update(tree, node * 2 + 1, m + 1, e, ii, value);
    return a + b;
}
```

- 갱신 역시 재귀적으로 구현
- 대상 원소 값만 바꾸는 것이 아니라, 대상 원소를 포함한 모든 구간(부모, 할아버지 etc)을 바꾸어 주어야한다.
- 갱신 대상이 범위 밖에 있을 때엔, 원래 값이 그대로 리턴
- 범위 안이데 s == e란 것은 leaf 노드란 소리이므로, 갱신 후 추가 탐색을 멈춘다.

(복습) 누적합 배열 사용이 안좋은 이유:
아래처럼 누적합 배열만 만들어둔 경우, 단순 반복으로 query는 구간 트리보다 빠를 수 있겠으나, 원소를 갱할 때 모두 다시 갈아 엎어야 해서 비효율적이다)

1	3	6	10	15	21	28	36	45
<pre>for(int i = 1; i <= n; i++) { cin >> arr[i]; sum[i] = sum[i-1] + arr[i]; }</pre>								

구간트리 진가는 갱신할 때 더욱 발휘한다~

구간 트리 구현 (구간합 찾기)

■ 갱신 실행 결과

```
update(segTree, 1, 1, N, 5, 100); // 인덱스 4 변경 + 100
cout << "sum of idx 6~9: " << query(segTree, 1, 1, N, 7, 10) << endl;
update(segTree, 1, 1, N, 6, 100); // 인덱스 5 변경 + 100
cout << "sum of idx 6~9: " << query(segTree, 1, 1, N, 7, 10) << endl;
update(segTree, 1, 1, N, 7, 100); // 인덱스 6 변경 + 100
cout << "sum of idx 6~9: " << query(segTree, 1, 1, N, 7, 10) << endl;
update(segTree, 1, 1, N, 8, 100); // 인덱스 7 변경 + 100
cout << "sum of idx 6~9: " << query(segTree, 1, 1, N, 7, 10) << endl;
```

```
sum of idx 6~9: 34
sum of idx 6~9: 34
sum of idx 6~9: 134
sum of idx 6~9: 234
```

[추가] 구간 트리 구현 (최솟값 찾기)

RMQ = Range Minimum Query.

구간트리에서 특정 구간 최솟값 찾기

기초 구간 트리에서 가장 많이 다루는 내용은 구간합구하기 + RMQ입니다.

직전에 구간합 쿼리를 수행했다면, 이번에는 최솟값 구하기 쿼리를 수행해보자. RMQ 라고 부르기도 하다.

구간합 구하기와 특별하게 다르게 가야하는 것은 아니다. 대부분의 경우, 합수가 100% 같거나 매우 비슷하다.

- 네이티브 기능 4번 그시노 `segtree`를 배울 즈음에 간접하기
- 재귀로 반씩 나누가며 트리 초기화 및 수행 등 구현 방식은 **똑같다.**

단, 달라지는 점은:

- 여기서의 쿼리는 합이 아닌 최소값 찾기 이므로, 함수가 수행될 때, `min(좌구간, 우구간)`을 반환
(복습: 구간합 때에는 좌구간+우구간이 반환되었다.)
- 또한, 트리구간과 쿼리구간이 전혀 겹치지 않을 경우, 아주 큰 값을 반환
(복습: 구간합 때에는 0 반환)

■ 트리초기화

```
#include <iostream>
using namespace std;
int arr[10] = {15, 17, 11, 16, 13, 14, 12, 19, 20, 18};

int init(int* tree, int node, int s, int e) {
    // s & e : arr의 시작 & 끝 인덱스
    if (s == e) return tree[node] = arr[s];

    // 재귀로 반씩 나누어 초기화
    int m = (s + e) / 2;
    int a = init(tree, node * 2, s, m);
    int b = init(tree, node * 2 + 1, m + 1, e);
    return tree[node] = min(a, b);
}
```

RMQ

```
#include <iostream>
using namespace std;
int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int init(int* tree, int node, int s, int e) {
    // s & e : arr의 시작 & 끝 인덱스
    if (s == e) return tree[node] = arr[s];

    // 재귀로 반씩 나누어 초기화
    int m = (s + e) / 2;
    int a = init(tree, node * 2, s, m);
    int b = init(tree, node * 2 + 1, m + 1, e);
    return tree[node] = a + b;
}
```

구간합

■ 쿼리수행

```
int query(int* tree, int node, int ts, int te,
          int qs, int qe) {
    // qs~qe : arr의 질의구간
    // ts~te : tree의 구간

    // 경우1 : 트리구간이 질의구간에 완전히 포함됨
    if (ts >= qs && qe >= te) return tree[node];

    // 경우2 : 트리구간과 질의구간이 전혀 겹치지 않음
    if (te < qs || qe < ts) return 99999;

    // 경우3 : 일부 겹침
    int m = (ts + te) / 2;
    int a = query(tree, node * 2, ts, m, qs, qe);
    int b = query(tree, node * 2 + 1, m + 1, te, qs, qe);
    return min(a, b);
}
```

RMQ

```
int query(int* tree, int node, int ts, int te,
          int qs, int qe) {
    // qs~qe : arr의 질의구간
    // ts~te : tree의 구간

    // 경우1 : 트리구간이 질의구간에 완전히 포함됨
    if (ts >= qs && qe >= te) return tree[node];

    // 경우2 : 트리구간과 질의구간이 전혀 겹치지 않음
    if (te < qs || qe < ts) return 0;

    // 경우3 : 일부 겹침
    int m = (ts + te) / 2;
    int a = query(tree, node * 2, ts, m, qs, qe);
    int b = query(tree, node * 2 + 1, m + 1, te, qs, qe);
    return a + b;
}
```

구간합

■ 갱신

```
int update(int* tree, int node, int s, int e, int ii, int value) {
    // ii : 수정할 인덱스
    // value : 더해서 갱신

    // 대상 범위 안에 있을 때만 갱신 진행
    if (s > ii || ii > e) return tree[node];
    tree[node] += value;
    if (s == e) return tree[node];

    int m = (s + e) / 2;
    int a = update(tree, node * 2, s, m, ii, value);
    int b = update(tree, node * 2 + 1, m + 1, e, ii, value);
    return tree[node] = min(a, b);
}
```

RMQ

```
int update(int* tree, int node, int s, int e, int ii, int value) {
    // ii : 수정할 인덱스
    // value : 더해서 갱신

    // 대상 범위 안에 있을 때만 갱신 진행
    if (s > ii || ii > e) return tree[node];
    tree[node] += value;
    if (s == e) return tree[node];

    int m = (s + e) / 2;
    int a = update(tree, node * 2, s, m, ii, value);
    int b = update(tree, node * 2 + 1, m + 1, e, ii, value);
    return tree[node] = (a + b);
}
```

구간합

Part2 제곱근 분할

제곱근 분할 개요

Sqrt Decomposition (aka 제곱근 분할, 평방 분할)

이름 그대로 원소들을 **SQRT(square root)** 단위로 **분할**하는 것이다..



- 원소가 16개라면 4개씩, 49개라면 7개씩. 원소가 10개라면 3.xx개니까 3개 혹은 4개씩이다.
- 각 그룹은 대푯값을 가지고 있다. (주어지는 query가 sum이라면 그룹의 합이 대푯값)
- 쿼리 수행 시간은 $O(\sqrt{N})$ 이다. 각 그룹의 대푯값을 들고오면 되기 때문이다.

예: 쿼리(구간합) 수행

- ▷ 구간이 딱 맞아 떨어질 때 : 인덱스 4~11번의 합 → 대푯값 $14 + 24 = 38$ 로 간단히 해결
- ▷ 구간이 딱 맞아 떨어지지 못함 : 인덱스 3~12의 합 → 4~11번째 원소는 대푯값으로 해결하지만, 3번과 12번은 별도 계산. 쿼리를 날리는 구간의 왼쪽 몇 개 오른 쪽 몇 개 수준이므로 여전히 총 $O(\sqrt{N})$ 시간으로 본다.

제곱근 분할 SQRT Decomposition

● 쿼리(구간합) 수행 : $O(\sqrt{N})$

```
long long query(int lo, int hi) {
    long long ret = 0;
    // 구간이 딱 맞아 떨어지지 않을 경우. 왼쪽에 걸쳐있는 묶음의 원소들을 모두 더해준다
    while (lo % sz != 0 && lo <= hi) {
        ret += A[lo++];
    }
    // 오른쪽에 걸쳐있는 묶음의 원소들을 모두 더해준다
    while ((hi + 1) % sz != 0 && lo <= hi) {
        ret += A[hi--];
    }
    // 구간이 딱 맞아 떨어질 때
    while (lo <= hi) {
        ret += bucket[lo / sz];
        lo += sz;
    }
    return ret;
}
```

제곱근 분할 SQRT Decomposition

● 간접 O(1)

그룹을 지어놓았기 때문에, 바꿔는 원소의 그룹과 대푯값만 찾아 바꾸면 된다.





```
void update(int pos, long long val) {
    // 원래 원소의 값과 갱신해야 하는 값의 차이를 계산
    long long diff = val - A[pos];
    // 기존 원소를 새로운 값으로 대체
    A[pos] = val;
    // 기존 원소가 속해있는 묶음에 갱신으로 인해 생기는 차이만큼의 값을 더해준다
    bucket[pos / sz] += diff;
}
```

3. 참고 자료

[Visual Reference Code - Tree](#)

Link : <https://swexpertacademy.com/main/learn/referenceCode/referenceCodeDetail.do?referenceId=TREE&category=undefined>

4. 기본 문제

- Segment Tree 연습 - 1

5. 응용 문제

- 출근길 라디오

* 응용문제에 대한 Live 해설 특강이 14:00시 부터 진행됩니다.

댓글 (0)

댓글을 입력해 주세요.

0 / 1000자

About

SW Expert Academy
포인트와 랭킹
학습 가이드

Support

공지사항
FAQ
Q&A
사이트 오류 제보
사이트 맵

이용 약관 | 개인정보처리방침 | 이메일 무단 수집 거부

1366*768 이상 해상도에 최적화되어 있으며 Google Chrome 브라우저를 권장합니다. (Internet Explorer 11 이상 지원)

본 사이트의 콘텐츠는 저작권법의 보호를 받는 바 무단 전재, 복사, 배포 등을 금합니다.

© 2017 SAMSUNG. ALL RIGHTS RESERVED

