

# 累積和

## 区間和を高速に求める

Keiya Sato  
Yokoyama Lab, M1  
April 27, 2023

# 目次

- はじめに
- 累積和
- 実装
- 発展的課題
- 例題

# はじめに

以下の問題が出てきたらどのようにして解きますか？

## 問題文

長さ  $N$  の数列  $A = A_0, A_1, \dots, A_{N-1}$  が与えられます。

以下の形式の  $Q$  個のクエリ进行处理してください。

- 整数  $l_i, r_i$  が与えられるので, 区間和  $A_{l_i} + A_{l_i+1} + \dots + A_{r_i-1}$  を求めてください。

## 制約

- ◆  $0 < N < 10^5$
- ◆  $0 \leq A_i < 10^9$  ( $i = 0, 1, \dots, N-1$ )
- ◆  $0 < Q < 10^5$
- ◆  $0 \leq l_i \leq r_i \leq N$

# はじめに

## 愚直に解く場合

- 各クエリで区間和を求めるのに  $O(N)$
- 全体で  $O(NQ)$
- 制約的に間に合わない

じゃあどうする？ → 累積和で解決

# 累積和

## 累積和とは

適切な前処理をしておくことで、配列上の区間の総和を求めるクエリを爆速で処理できるようになる手法<sup>1</sup>

- 前処理に  $O(N)$  かかるが、各クエリは  $O(1)$  で答えられるようになる
- 先ほどの問題を  $O(N + Q)$  で解ける

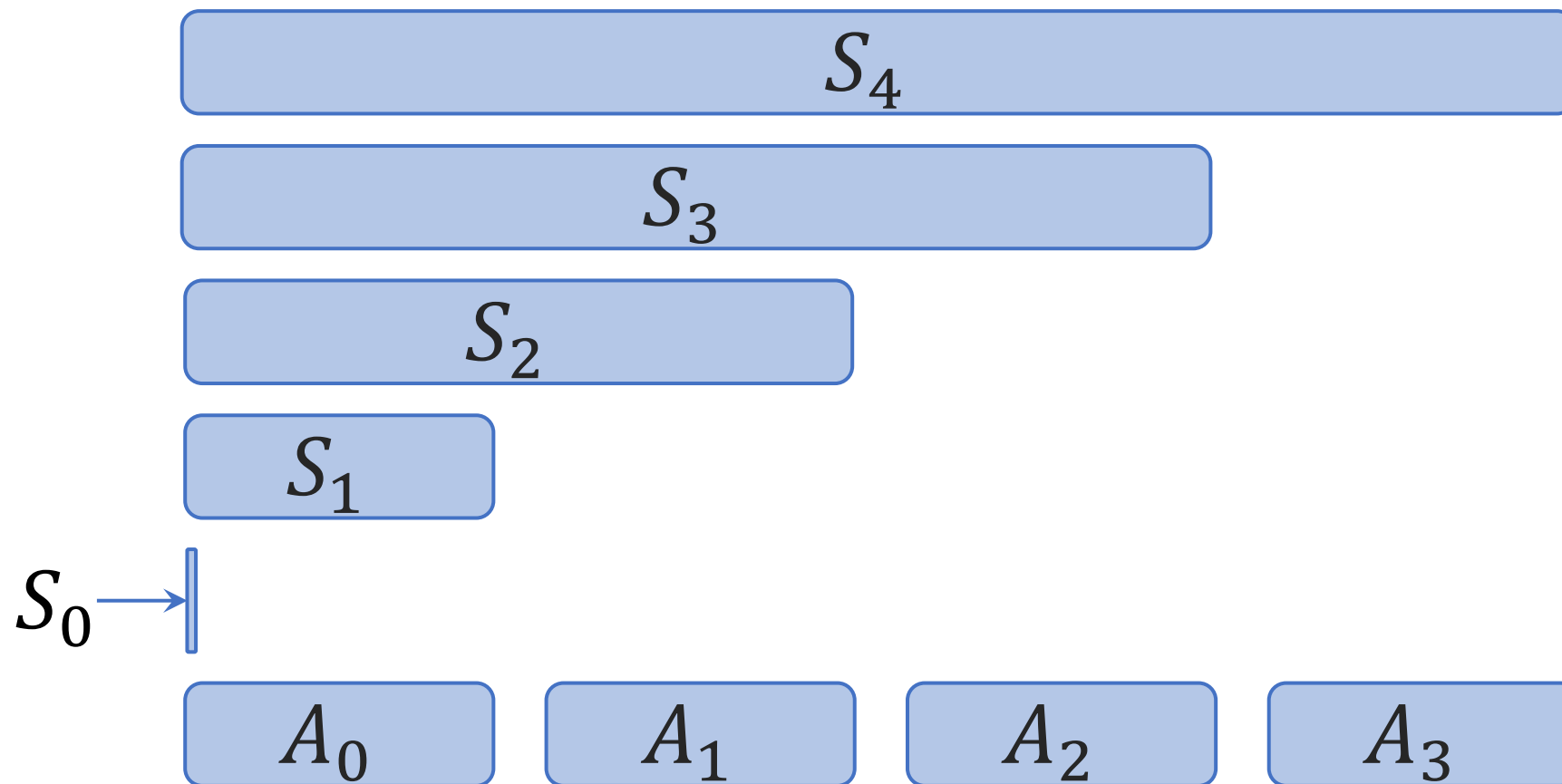
1. drken. Qiita. 「累積和を何も考えずに書けるようにする！」 . <https://qiita.com/drken/items/56a6b68edef8fc605821>

# 累積和：前処理

数列  $A = A_0, A_1, \dots, A_{N-1}$  に対して数列  $S = S_0, S_1, \dots, S_N$  を以下の  
ように定める.

- $S_0 = 0$
- $S_1 = A_0$
- $S_2 = A_0 + A_1$
- $\vdots$
- $S_N = A_0 + A_1 + \dots + A_{N-2} + A_{N-1}$

# 累積和：前処理



# 前処理：クエリ

- $A_l$  から  $A_{r-1}$  までの和を求めたいとき

$S_r - S_l$  を計算すると答えが出る

- $S_r = A_0 + A_1 + \cdots + A_{l-1} + A_l + \cdots + A_{r-1}$
- $S_l = A_0 + A_1 + \cdots + A_{l-1}$
- $S_r - S_l = A_l + A_{l+1} + \cdots + A_{r-1}$



# 実装

- $S_i = A_0 + \dots + A_{i-1}$  だから,  
 $S_{i+1} = S_i + A_i$  で計算できる
- 添え字に注意して区間和を求める

```
// 前処理
vector<int> s(n + 1, 0);
for (int i = 0; i < n; i++)
{
    s[i + 1] = s[i] + a[i];
}

// クエリ
int l, r;
cin >> l >> r;
cout << s[r] - s[l] << endl;
```

# 実装

- Python だとこんな感じ
- C++ も Python も 0-indexed なのか 1-indexed なのかを注意して実装する必要がある
  - 問題文によって異なる場合がある
  - AtCoder は基本的に 1-indexed

```
# 前処理
s = [0 for i in range(n + 1)]
for i in range(n):
    s[i + 1] = s[i] + a[i]

# クエリ
l, r = int(input().split())
print(s[r] - s[l])
```

# 発展的話題：二次元累積和

- 二次元のデータに対しても累積和を用いることで区間和を求めることができる
- 考え方は一次元累積和と一緒
- 同様に三次元, 四次元の累積和も考えることができる（問題として出ることほとんどない）
- ABC で出るとしたら E 以上だと思うので, まずは一次元累積和をぱっと書けるようにすべき

# 発展的話題：似ているデータ構造

- **Sparse Table**

- 区間に対する  $\min, \max, \gcd, \text{lcm}$  などのクエリを処理することができる
- 区間和や区間積など冪等則を満たさないものは計算できない
- 前処理  $O(N \log N)$ , クエリ  $O(1)$

- **Disjoint Sparse Table**

- $\min, \max, \gcd, \text{lcm}$  などに加え区間和, 区間積なども求められる Sparse Table の完全上位互換
- 前処理  $O(N \log N)$ , クエリ  $O(1)$

# 発展的話題：似ているデータ構造

- Segment Tree (セグ木)

- 区間和, 区間積,  $\min$ ,  $\max$ ,  $\gcd$ ,  $\text{lcm}$  などが求められる
- 値の更新も可能 (一点更新)
- 前処理  $O(N \log N)$ , クエリ  $O(\log N)$ , 更新  $O(\log N)$
- 区間更新一点取得が可能な双対セグ木や, 区間更新区間和取得が可能な遅延セグ木なども存在する

# 例題

## ABC084 D - 2017-like Number

- $l_i$  以上  $r_i$  以下の整数で「2017 に似た数」を数えるクエリを処理する
  - 2017 に似た数 :  $N$  も  $N + 1/2$  も素数であるような数
- ヒント :  $A_i = i$  が 2017 に似ているなら 1, そうでないなら 0 とすると区間和を求める問題に帰着できる
- 回答例 C++ : <https://atcoder.jp/contests/abc084/submissions/31289418>
- 回答例 Python : <https://atcoder.jp/contests/abc084/submissions/31289440>

# 素数判定

- 整数  $N$  が与えられたときに,  $N$  が素数かどうかを判定するアルゴリズム

- 時間計算量:  $\theta(\sqrt{N})$

- 空間計算量:  $\theta(1)$

- 試し割法とも呼ばれ,  $\sqrt{N}$  以下の全ての数で割り切れるかどうかを実際に試すことで素数判定を行う

## Algorithm

1. for 文で  $i$  を 2 から  $\sqrt{N}$  まで回す
  1.  $N$  が  $i$  で割り切れるならば  $N$  は合成数
2.  $N$  が全ての  $i$  で割り切れないならば素数

## 解説

例えば 36 の約数は 1, 2, 3, 4, 6, 9, 12, 18, 36.

6 以下の数で割り切れるとき, 必ず 6 より大きい数で割り切れ, 逆に 6 以下の数で割り切れないとき, 同様に 6 より大きい数では絶対に割り切れない.

したがって,  $\sqrt{36}$  以下の数だけ試すことで判定できる

# 素数判定：実装

```
bool is_prime (int n) {  
    for (int x = 2; x*x <= n; x++) {  
        if (n % x == 0) {  
            return false;  
        }  
    }  
    return true;  
}
```

C++

```
def is_prime(n: int) -> bool:  
    x = 2  
    while x * x <= n:  
        if n % x == 0:  
            return False  
        x += 1  
    return True
```

Python



# 例題

## ABC177 C - Sum of product of pairs

- $\sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} A_i A_j$  を求める問題
- ヒント：式を変形すると  $\sum_{i=0}^{N-2} A_i \sum_{j=i+1}^{N-1} A_j$  となり累積和が見えてくる
- 回答例 C++ : <https://atcoder.jp/contests/abc177/submissions/31296937>
- 回答例 Python : <https://atcoder.jp/contests/abc177/submissions/31297000>

# 例題

## AGC023 A - Zero-Sum Ranges

- 少し難しめ
- 総和が 0 になる部分列の数を数える問題
- 部分列の総和（区間和）が 0 になるとはどういうことを考えるとよい