Maintaining partial sums in logarithmic time

Jochen Burghardt

Berlin

Abstract

We present a data structure that allows to maintain in logarithmic time all partial sums of elements of a linear array during incremental changes of element’s values.

Key words: Partial sums; Data structures; Algorithms

1 Motivation

Assume you have a linear array \( x_0, \ldots, x_{N-1} \) of numbers which are frequently updated, and you need to maintain all partial sums \( \sum_{i=j}^{k} x_i \), where \( 0 \leq j \leq k < N \). We present a data structure that allows to access each \( x_k \) and to compute any partial sum in time \( \mathcal{O}(\log(N)) \).

As an application, think of the \( x_k \) as integer numbers indicating the probabilities of certain events; by choosing a uniformly distributed random number \( r \) in the range \( 0 \leq r < \sum_{i=0}^{N-1} x_i \) and selecting the unique \( k \in \{0, \ldots, N\} \) with \( \sum_{i=k}^{N-1} x_i \leq r < \sum_{i=k-1}^{N-1} x_i \), event \( k \) is selected with probability \( \frac{x_k}{\sum_{i=0}^{N-1} x_i} \).

If the probability distribution of events changes frequently, the partial sums need to be recomputed every time, which takes time \( \mathcal{O}(N) \) using the naive algorithm.

2 Data structure and access algorithms

Our solution is to store a mix of individual values \( x_i \) and partial sums in the array, thus realizing a binary tree where each node represents the sum of all leaves below it. Figure 1 sketches an example for \( N = 16 \), the partial sums corresponding to the nodes indicated by solid circles are stored as \( s_i \).
In some respect, this idea is similar to that of heap sort [AHU74, Sect. 3.4], which also uses a mix of representations (sorted along a path and unsorted within a level) to combine the advantages of both. Our data structure combines the advantages of storing single values (easily updatable) and sums (no need to recompute them).

Formally, let \( N \) be a power of 2; let an array \( x_0, \ldots, x_{N-1} \) of size \( N \) be given. Instead of this original array, we maintain the array \( s_0, \ldots, s_{N-1} \), where

\[
s_k := \sum_{i=0}^{\gcd(N,k)-1} x_{k+i}
\]

Here, \( \gcd(N,k) \) is the greatest common divisor of \( N \) and \( k \), i.e., the largest power of 2 dividing \( k \). It corresponds to the least 1 bit in the 2–complement representation of \( k \), which can be computed as bitwise and of \( N+k \) and \( N-k \).

The following algorithms, given in C code in Fig. 2, maintain our data structure.

- int sumN(int k) returns \( \sum_{i=k}^{N-1} x_i \);
- int sum(int j,k) returns \( \sum_{i=j}^{k} x_i \);
- int get(int k) retrieves \( x_k \);
- void inc(int k,x) adds \( x \) to \( x_k \);
- void set(int k,x) assigns \( x \) to \( x_k \); and
- int find(int x) returns some \( k \) such that \( \sum_{i=k}^{N-1} x_i \leq x < \sum_{i=k+1}^{N-1} x_i \), provided \( 0 \leq x < \sum_{i=0}^{N-1} x_i \); \( k \) is unique if no \( x_i \) is negative.

Figure 3 shows some sample runs on the data in Fig. 1.

In order to deal with arrays whose size is not a power of 2, assume \( s_k = 0 \) for all \( k \geq M \), where \( N/2 < M \leq N \). At two places it is necessary to test the index boundary explicitly, using the function int S(int i).

The algorithms can immediately be generalized to deal with arbitrary (non–
int s[M];

#define S(i) (i<M ? s[i] : 0)

#define gcdN(k) ((N+k) & (N-k))

int sumN(int k) {
    int i, sm = 0;
    for (i=k; i<M; i+=gcdN(i))
        sm += s[i];
    return sm;
}

#define sum(j,k) (sumN(j) - sumN(k+1))

int get(int k) {
    int i, x = s[k];
    for (i=1; i<gcdN(k) && k+i<M; i*=2)
        x -= s[k+i];
    return x;
}

void inc(int k,x) {
    int i;
    for (i=k; i>=0; i-=gcdN(i))
        s[i] += x;
}

#define set(k,x) inc(k,x-get(k))

int find(int x) {
    int i, k = 0, pv = s[N/2];
    for (i=N/2; i>0; i/=2)
        if (x < pv) {
            pv += S(k+i*3/2) - s[k+i];
            k += i;
        } else {
            pv += S(k+i/2);
        }
    return k;
}

Fig. 2. Algorithms
abelian) group elements instead of integers; if \texttt{find} is to be used, ordered groups are necessary.

3 Complexity

All algorithms take $O(\log N)$ time due to the implicit tree structure. For \texttt{sumN} and \texttt{inc}, note that the value of $\gcd(N, i)$ grows in every loop cycle, since

$$\gcd(N, i \pm \gcd(N, i)) \geq 2 \cdot \gcd(N, i). \quad (2)$$

In the following sections 4 to 7, we give correctness proofs of the main algorithms in the Hoare calculus [Hoa69].

4 Correctness of \texttt{get}

To see the correctness of \texttt{get}, show

$$\sum_{i=1}^{2^a-1} f(i) = \sum_{i=0}^{a-1} \sum_{j=0}^{2^i-1} f(2^i + j) \quad (3)$$

by induction on $a$; note that commutativity of $+$ is not required for the proof.

If $\gcd(N, k) = 2^a$, we have $\gcd(N, k + 2^i) = 2^i$ for $0 \leq i < a$, and therefore

$$s_k(1) = x_k + \sum_{i=1}^{2^a-1} x_{k+i} \quad (3) = x_k + \sum_{i=0}^{a-1} \sum_{j=0}^{2^i-1} x_{k+2^i+j} \quad (1) = x_k + \sum_{i=0}^{a-1} s_{k+2^i} \quad (4)$$
We define the abbreviation $\Sigma_{k,b} := s_k + s_{k+2b} + s_{k+4b} + \ldots + s_{k+\gcd(N,k)/2}$.

By equation (4), we obtain $s_k = x_k + \Sigma_{k,1}$, justifying the step in lines 4.–5.

We have $\Sigma_{k,b} = 0$ if $b \geq \gcd(N,k)$ or $k + b \geq M$; this justifies lines 13.–14.

We can now apply the Hoare calculus to the code of `int get(int k)`:

1. `int get(int k) {`
2. `int i, x;`
3. `x = s[k];`
4. `i = 1;`
5. `x = x_k + \Sigma_{k,1} \land i \leq \gcd(N,k)`
6. `while (i < \gcd(N,k) \&\& k+i < M) {`
7. `x = x_k + \Sigma_{k,1} \land i < \gcd(N,k)`
8. `x = x - s[k+i];`
9. `x = x_k + \Sigma_{k,1} \land i < \gcd(N,k)`
10. `i = i \ast 2;`
11. `x = x_k + \Sigma_{k,1} \land i \leq \gcd(N,k)`
12. `}`
13. `x = x_k + \Sigma_{k,1} \land (i = \gcd(N,k) \lor k+i \geq M)`
14. `return x;`
15. `}`

5 Correctness of inc

Next, we show that `inc` makes sufficiently many updates. By (1), $s_i$ depends on $x_k$, iff $i \leq k < i + \gcd(N,i)$.

Hence, if $s_i$ depends on $x_k$, then so does $s_{i - \gcd(N,i)}$, since

$$i - \gcd(N,i) \leq i \leq k \text{ and, by (2),}$$

$$(i - \gcd(N,i)) + \gcd(N,i - \gcd(N,i)) \geq i + \gcd(N,i) > k .$$

But no $s_{i'}$ for $i - \gcd(N,i) < i' < i$ depends on $x_k$:

Let $i = 2^a \cdot b$ and $i' = 2^{a'} \cdot b'$ for odd numbers $b, b'$.

Then $a' < a$ since $i - \gcd(N,i) = 2^a \cdot (b - 1)$.

And $2^{a'} \cdot b' = i' < i = 2^{a-a'} \cdot 2^{a'} \cdot b$ implies $b' + 1 \leq 2^{a-a'} \cdot b$.

Hence, $i' + \gcd(N,i') = 2^{a'} \cdot (b' + 1) \leq 2^{a'} \cdot 2^{a-a'} \cdot b = i \leq k$.
6 Correctness of sumN

The loop in sumN satisfies the invariant $s_m = \sum_{j=k}^{i-1} x_j$, since

$$s_m + s_i = \left(\sum_{j=k}^{i-1} x_j\right) + \left(\sum_{j=0}^{\gcd(N,i)-1} x_{i+j}\right) = \sum_{j=k}^{i+\gcd(N,i)-1} x_j.$$ 

This justifies the step in lines 7.–9. For lines 13.–14. note that $x_j = 0$ for $j \geq M$.

```c
int sumN(int k) {
    int i, sm;
    sm = 0;
    i = k;
    sm = \sum_{j=k}^{i-1} x_j
    while (i < M) {
        sm = \sum_{j=k}^{i-1} x_j
        sm = sm + s[i];
        sm = \sum_{j=k}^{i+\gcd(N,i)-1} x_j
        i = i + \gcdN(i);
        sm = \sum_{j=k}^{i-1} x_j
    }
    sm = \sum_{j=k}^{i-1} x_j \land i \geq M
    return sm;
}
```

7 Correctness of find

The loop in find satisfies the invariant

$$\sum_{j=k+2}^{N-1} x_j \leq x < \sum_{j=k}^{N-1} x_j \quad \text{and} \quad \left(i \geq 2 \Rightarrow pv = \sum_{j=k+1}^{N-1} x_j\right)$$

and $\gcd(N, k) \geq \gcd(N, i) = i$. \hspace{1cm} (5)

To show this, note that for $i \geq 2$, we have

$$s_{k+i} = (1) \sum_{j=0}^{\gcd(N,k+1)-1} x_{k+1+j} = (5) \sum_{j=0}^{i-1} x_{k+1+j} = \sum_{j=k+i}^{k+2+i-1} x_j,$$
and similarly
\[ s_{k + 1/2} = \sum_{j = k + 1/2}^{k+1-1} x_j \quad \text{and} \quad s_{k + 1.3/2} = \sum_{j = k + 1.3/2}^{k+2-1} x_j, \]
hence, we get
\[ p v + s_{k+1.3/2} - s_{k+1} = \sum_{j = k + 1.3/2}^{N-1} x_j \quad \text{and} \quad p v + s_{k+1/2} = \sum_{j = k + 1/2}^{N-1} x_j, \quad (6) \]
in case of \( x < p v \) and \( x \geq p v \), respectively.

We transform the program to make the Hoare verification rules applicable and unfold the last loop cycle \((i = 1)\) to avoid confusing case distinctions. We omit the computation of the pivot element \(p v\) in the last cycle, since its value isn’t used any more.

We define the abbreviations \(\Sigma_a := \sum_{j = a}^{N-1} x_j\) and \(p(a, b) :\Leftrightarrow \gcd(N, a) \geq \gcd(N, b) = b\)

Observe that \(i \geq 2 \land p(k, i)\) implies both \(p(k + i, i)\) and \(p(k, i/2);\) this is used in lines 13.–15. and 21.–23., respectively.

Equations (6) justify the steps in lines 13.–15. and 19.–21.; equation (1) justifies step 7.–9.

1. \(0 \leq x < \Sigma_0\)
2. \textbf{int find(int x) \{}
3. \hspace{1em} \textbf{int i, k, pv;}
4. \hspace{1em} \Sigma_N \leq x < \Sigma_0
5. \hspace{1em} k = 0;
6. \hspace{1em} i = N/2;
7. \hspace{1em} \Sigma_{k+2.1} \leq x < \Sigma_{k} \land p(k, i) \land i \geq 1
8. \hspace{1em} pv = s[N/2];
9. \hspace{1em} \Sigma_{k+2.1} \leq x < \Sigma_{k} \land pv = \Sigma_{k+1} \land p(k, i) \land i \geq 1
10. \textbf{while (i >= 2) \{}
11. \hspace{1em} \Sigma_{k+2.1} \leq x < \Sigma_{k} \land pv = \Sigma_{k+1} \land p(k, i) \land i \geq 2
12. \hspace{1em} \textbf{if (x < pv) \{}
13. \hspace{2em} \Sigma_{k+2.1} \leq x < \Sigma_{k+1} \land pv = \Sigma_{k+1} \land p(k, i) \land i \geq 2
14. \hspace{2em} pv = pv + S(k+i*3/2) - s[k+i];
15. \hspace{2em} \Sigma_{k+2.1} \leq x < \Sigma_{k+1} \land pv = \Sigma_{k+3.1/2} \land p(k + i, i) \land i \geq 2
16. \hspace{2em} k = k + i;
17. \hspace{2em} \Sigma_{k+1} \leq x < \Sigma_{k} \land pv = \Sigma_{k+1/2} \land p(k, i) \land i \geq 2
18. \hspace{2em} \textbf{\}} \textbf{else \{}
19. \( \Sigma_{k+1} \leq x < \Sigma_k \land pv = \Sigma_{k+1} \land p(k,i) \land i \geq 2 \)
20. \( pv = pv + S(k+i/2); \)
21. \( \Sigma_{k+1} \leq x < \Sigma_k \land pv = \Sigma_{k+1/2} \land p(k,i) \land i \geq 2 \)
22. \}
23. \( \Sigma_{k+1} \leq x < \Sigma_k \land pv = \Sigma_{k+1/2} \land p(k,i/2) \land i \geq 2 \)
24. \( i = i/2; \)
25. \( \Sigma_{k+2} \leq x < \Sigma_k \land pv = \Sigma_{k+1} \land p(k,i) \land i \geq 1 \)
26. \}
27. \( \Sigma_{k+2} \leq x < \Sigma_k \land pv = \Sigma_{k+1} \land i = 1 \)
28. if (x < pv) {
29. \( \Sigma_{k+2} \leq x < \Sigma_{k+1} \land pv = \Sigma_{k+1} \)
30. \( k = k + 1; \)
31. \( \Sigma_{k+1} \leq x < \Sigma_k \)
32. } else {
33. \( \Sigma_{k+1} \leq x < \Sigma_k \)
34. }
35. \( \Sigma_{k+1} \leq x < \Sigma_k \)
36. return k;
37. }

This completes the verification proofs of the algorithms given in Fig. 2.
A short version of this paper (without proofs) was published in [Bur01].

References

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.

[Bur01] Jochen Burghardt. Maintaining partial sums in logarithmic time. Nordic Journal of Computing, 8(4):473–474, 2001.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. CACM, 12(10):576–583, Oct 1969.