Range Queries

9

In this chapter, we discuss data structures for efficiently processing range queries on arrays. Typical queries are range sum queries (calculating the sum of values) and range minimum queries (finding the minimum value).

Section 9.1 focuses on a simple situation where the array values are not modified between the queries. In this case it suffices to preprocess the array so that we can efficiently determine the answer for any possible query. We will first learn to process sum queries using a prefix sum array, and then we will discuss the sparse table algorithm for processing minimum queries.

Section 9.2 presents two tree structures that allow us to both process queries and update array values efficiently. A binary indexed tree supports sum queries and can be seen as a dynamic version of a prefix sum array. A segment tree is a more versatile structure that supports sum queries, minimum queries, and several other queries. The operations of both the structures work in logarithmic time.

9.1 Queries on Static Arrays

In this section, we focus on a situation where the array is *static*, i.e., the array values are never updated between the queries. In this case, it suffices to preprocess the array so that we can efficiently answer range queries.

First we will discuss a simple way to process sum queries using a prefix sum array, which can also be generalized to higher dimensions. After this, we will learn the sparse table algorithm for processing minimum queries, which is somewhat more difficult. Note that while we focus on processing minimum queries, we can always also process maximum queries using similar methods.

A. Laaksonen, *Guide to Competitive Programming*, Undergraduate
Topics in Computer Science, https://doi.org/10.1007/978-3-319-72547-5_9

9.1.1 Sum Queries

Let $\texttt{sum}_q(a, b)$ ("range sum query") denote the sum of array values in a range $[a, b]$. We can efficiently process any sum query by first constructing a *prefix sum array*. Each value in the prefix sum array equals the sum of values in the original array up to the corresponding position, i.e., the value at position k is $\texttt{sum}_q(0, k)$. For example, Fig. 9.1 shows an array and its prefix sum array.

The prefix sum array can be constructed in $O(n)$ time. Then, since the prefix sum array contains all values of $\texttt{sum}_q(0, k)$, we can calculate any value of $\texttt{sum}_q(a, b)$ in $O(1)$ time using the formula

$$\texttt{sum}_q(a, b) = \texttt{sum}_q(0, b) - \texttt{sum}_q(0, a - 1).$$

By defining $\texttt{sum}_q(0, -1) = 0$, the above formula also holds when $a = 0$.

As an example, Fig. 9.2 shows how to calculate the sum of values in the range $[3, 6]$ using the prefix sum array. We can see in the original array that $\texttt{sum}_q(3, 6) = 8 + 6 + 1 + 4 = 19$. Using the prefix sum array, we need to examine only two values: $\texttt{sum}_q(3, 6) = \texttt{sum}_q(0, 6) - \texttt{sum}_q(0, 2) = 27 - 8 = 19$.

Higher Dimensions It is also possible to generalize this idea to higher dimensions. For example, Fig. 9.3 shows a two-dimensional prefix sum array that can be used to calculate the sum of any rectangular subarray in $O(1)$ time. Each sum in this array

Fig. 9.1 An array and its prefix sum array

	0	1	2	3	4	5	6	7
original array	1	3	4	8	6	1	4	2

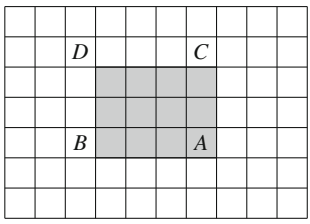
	0	1	2	3	4	5	6	7
prefix sum array	1	4	8	16	22	23	27	29

Fig. 9.2 Calculating a range sum using the prefix sum array

	0	1	2	3	4	5	6	7
original array	1	3	4	8	6	1	4	2

	0	1	2	3	4	5	6	7
prefix sum array	1	4	8	16	22	23	27	29

Fig. 9.3 Calculating a two-dimensional range sum



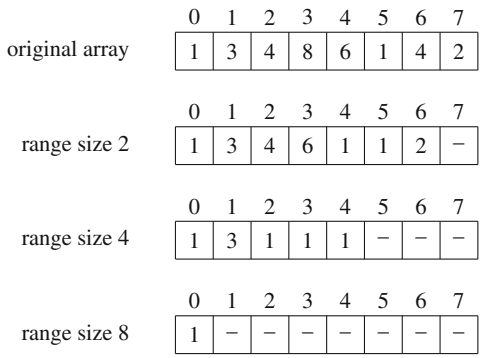


Fig. 9.4 Preprocessing for minimum queries

corresponds to a subarray that begins at the upper-left corner of the array. The sum of the gray subarray can be calculated using the formula

$$S(A) - S(B) - S(C) + S(D),$$

where $S(X)$ denotes the sum of values in a rectangular subarray from the upper-left corner to the position of X.

9.1.2 Minimum Queries

Let $\min_q(a, b)$ ("range minimum query") denote the minimum array value in a range $[a, b]$. We will next discuss a technique using which we can process any minimum query in $O(1)$ time after an $O(n \log n)$ time preprocessing. The method is due to Bender and Farach-Colton [3] and often called the *sparse table algorithm*.

The idea is to precalculate all values of $\min_q(a, b)$ where $b - a + 1$ (the length of the range) is a power of two. For example, Fig. 9.4 shows the precalculated values for an array of eight elements.

The number of precalculated values is $O(n \log n)$, because there are $O(\log n)$ range lengths that are powers of two. The values can be calculated efficiently using the recursive formula

$$\texttt{min}_q(a, b) = \min(\texttt{min}_q(a, a + w - 1), \texttt{min}_q(a + w, b)),$$

where $b - a + 1$ is a power of two and $w = (b - a + 1)/2$. Calculating all those values takes $O(n \log n)$ time.

After this, any value of $\texttt{min}_q(a, b)$ can be calculated in $O(1)$ time as a minimum of two precalculated values. Let k be the largest power of two that does not exceed $b - a + 1$. We can calculate the value of $\texttt{min}_q(a, b)$ using the formula

$$\texttt{min}_q(a, b) = \min(\texttt{min}_q(a, a + k - 1), \texttt{min}_q(b - k + 1, b)).$$

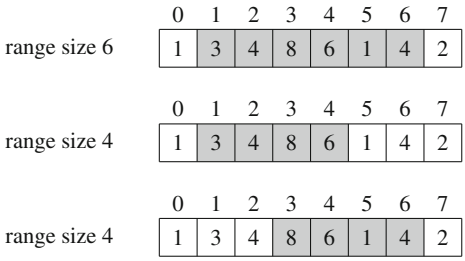


Fig. 9.5 Calculating a range minimum using two overlapping ranges

In the above formula, the range $[a, b]$ is represented as the union of the ranges $[a, a + k - 1]$ and $[b - k + 1, b]$, both of length k.

As an example, consider the range $[1, 6]$ in Fig. 9.5. The length of the range is 6, and the largest power of two that does not exceed 6 is 4. Thus the range $[1, 6]$ is the union of the ranges $[1, 4]$ and $[3, 6]$. Since $\texttt{min}_q(1, 4) = 3$ and $\texttt{min}_q(3, 6) = 1$, we conclude that $\texttt{min}_q(1, 6) = 1$.

Note that there are also sophisticated techniques using which we can process range minimum queries in $O(1)$ time after an only $O(n)$ time preprocessing (see, e.g., Fischer and Heun [10]), but they are beyond the scope of this book.

9.2 Tree Structures

This section presents two tree structures, using which we can both process range queries and update array values in logarithmic time. First, we discuss binary indexed trees that support sum queries, and after that, we focus on segment trees that also support several other queries.

9.2.1 Binary Indexed Trees

A *binary indexed tree* (or a *Fenwick tree*) [9] can be seen as a dynamic variant of a prefix sum array. It provides two $O(\log n)$ time operations: processing a range sum query and updating a value. Even if the name of the structure is a binary indexed *tree*, the structure is usually represented as an array. When discussing binary indexed trees, we assume that all arrays are one-indexed, because it makes the implementation of the structure easier.

Let $p(k)$ denote the largest power of two that divides k. We store a binary indexed tree as an array `tree` such that

$$\texttt{tree}[k] = \texttt{sum}_q(k - p(k) + 1, k),$$

i.e., each position k contains the sum of values in a range of the original array whose length is $p(k)$ and that ends at position k. For example, since $p(6) = 2$, `tree`[6]

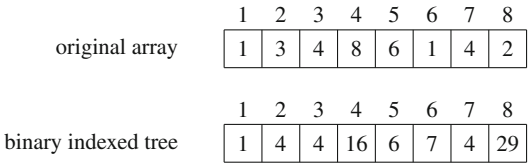


Fig. 9.6 An array and its binary indexed tree

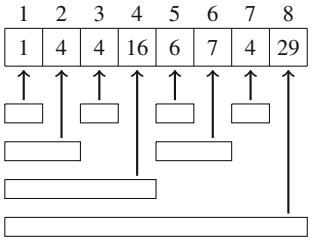


Fig. 9.7 Ranges in a binary indexed tree

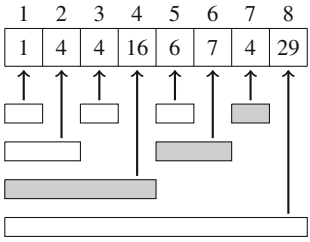


Fig. 9.8 Processing a range sum query using a binary indexed tree

contains the value of $\texttt{sum}_q(5, 6)$. Figure 9.6 shows an array and the corresponding binary indexed tree. Figure 9.7 shows more clearly how each value in the binary indexed tree corresponds to a range in the original array.

Using a binary indexed tree, any value of $\texttt{sum}_q(1, k)$ can be calculated in $O(\log n)$ time, because a range $[1, k]$ can always be divided into $O(\log n)$ subranges whose sums have been stored in the tree. For example, to calculate the value of $\texttt{sum}_q(1, 7)$, we divide the range $[1, 7]$ into three subranges $[1, 4]$, $[5, 6]$, and $[7, 7]$ (Fig. 9.8). Since the sums of those subranges are available in the tree, we can calculate the sum of the entire range using the formula

$$\texttt{sum}_q(1, 7) = \texttt{sum}_q(1, 4) + \texttt{sum}_q(5, 6) + \texttt{sum}_q(7, 7) = 16 + 7 + 4 = 27.$$

Then, to calculate the value of $\texttt{sum}_q(a, b)$ where $a > 1$, we can use the same trick that we used with prefix sum arrays:

$$\texttt{sum}_q(a, b) = \texttt{sum}_q(1, b) - \texttt{sum}_q(1, a - 1)$$

We can calculate both $\texttt{sum}_q(1, b)$ and $\texttt{sum}_q(1, a - 1)$ in $O(\log n)$ time, so the total time complexity is $O(\log n)$.

After updating an array value, several values in the binary indexed tree should be updated. For example, when the value at position 3 changes, we should update

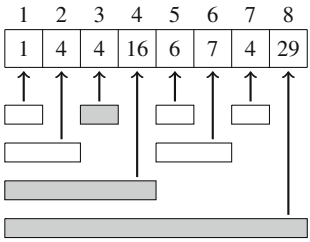


Fig. 9.9 Updating a value in a binary indexed tree

the subranges [3, 3], [1, 4], and [1, 8] (Fig. 9.9). Since each array element belongs to $O(\log n)$ subranges, it suffices to update $O(\log n)$ tree values.

Implementation The operations of a binary indexed tree can be efficiently implemented using bit operations. The key fact needed is that we can easily calculate any value of $p(k)$ using the bit formula

$$p(k) = k \,\&\, -k,$$

which isolates the least significant one bit of k.

First, the following function calculates the value of $\texttt{sum}_q(1, k)$:

```
int sum(int k) {
    int s = 0;
    while (k >= 1) {
        s += tree[k];
        k -= k&-k;
    }
    return s;
}
```

Then, the following function increases the array value at position k by x (x can be positive or negative):

```
void add(int k, int x) {
    while (k <= n) {
        tree[k] += x;
        k += k&-k;
    }
}
```

The time complexity of both the functions is $O(\log n)$, because the functions access $O(\log n)$ values in the binary indexed tree, and each move to the next position takes $O(1)$ time.

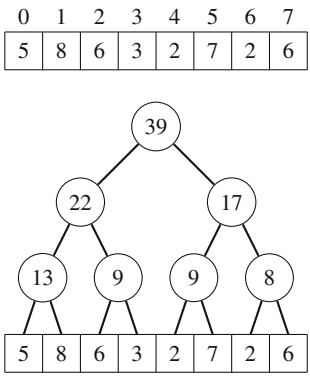


Fig. 9.10 An array and the corresponding segment tree for sum queries

9.2.2 Segment Trees

A *segment tree* is a data structure that provides two $O(\log n)$ time operations: processing a range query and updating an array value. Segment trees support sum queries, minimum queries, and many other queries. Segment trees have their origins in geometric algorithms (see, e.g., Bentley and Wood [4]), and the elegant bottom-up implementation presented in this section follows the textbook by Stańczyk [30].

A segment tree is a binary tree whose bottom level nodes correspond to the array elements, and the other nodes contain information needed for processing range queries. When discussing segment trees, we assume that the size of the array is a power of two, and zero-based indexing is used, because it is convenient to build a segment tree for such an array. If the size of the array is not a power of two, we can always append extra elements to it.

We will first discuss segment trees that support sum queries. As an example, Fig. 9.10 shows an array and the corresponding segment tree for sum queries. Each internal tree node corresponds to an array range whose size is a power of two. When a segment tree supports sum queries, the value of each internal node is the sum of the corresponding array values, and it can be calculated as the sum of the values of its left and right child node.

It turns out that any range $[a, b]$ can be divided into $O(\log n)$ subranges whose values are stored in tree nodes. For example, Fig. 9.11 shows the range $[2, 7]$ in the original array and in the segment tree. In this case, two tree nodes correspond to the range, and $\text{sum}_q(2, 7) = 9 + 17 = 26$. When the sum is calculated using nodes located as high as possible in the tree, at most two nodes on each level of the tree are needed. Hence, the total number of nodes is $O(\log n)$.

After an array update, we should update all nodes whose value depends on the updated value. This can be done by traversing the path from the updated array element to the top node and updating the nodes along the path. For example, Fig. 9.12 shows the nodes that change when the value at position 5 changes. The path from bottom to top always consists of $O(\log n)$ nodes, so each update changes $O(\log n)$ nodes in the tree.

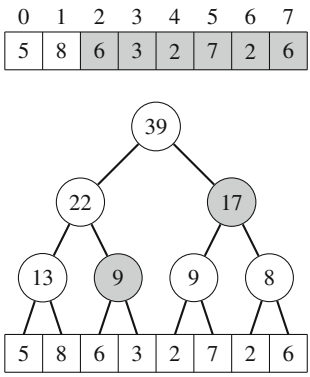


Fig. 9.11 Processing a range sum query using a segment tree

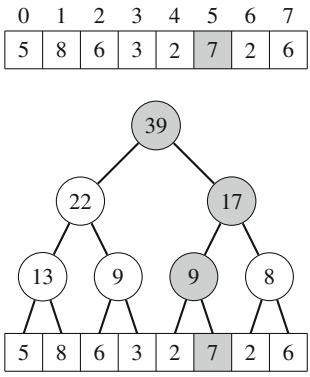


Fig. 9.12 Updating an array value in a segment tree

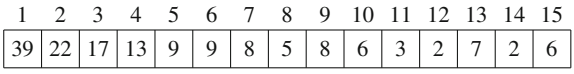


Fig. 9.13 Contents of a segment tree in an array

Implementation A convenient way to store the contents of a segment tree is to use an array of $2n$ elements where n is the size of the original array. The tree nodes are stored from top to bottom: `tree`[1] is the top node, `tree`[2] and `tree`[3] are its children, and so on. Finally, the values from `tree`[n] to `tree`[$2n-1$] correspond to the bottom level of the tree, which contains the values of the original array. Note that the element `tree`[0] is not used.

For example, Fig. 9.13 shows how our example tree is stored. Note that the parent of `tree`[k] is `tree`[$\lfloor k/2 \rfloor$], its left child is `tree`[$2k$], and its right child is `tree`[$2k+1$]. In addition, the position of a node (other than the top node) is even if it is a left child and odd if it is a right child.

The following function calculates the value of $\texttt{sum}_q(a, b)$:

```
int sum(int a, int b) {
    a += n; b += n;
    int s = 0;
    while (a <= b) {
        if (a%2 == 1) s += tree[a++];
        if (b%2 == 0) s += tree[b--];
        a /= 2; b /= 2;
    }
    return s;
}
```

The function maintains a range in the segment tree array. Initially, the range is $[a + n, b + n]$. At each step, the range is moved one level higher in the tree, and the values of the nodes that do not belong to the higher range are added to the sum.

The following function increases the array value at position k by x:

```
void add(int k, int x) {
    k += n;
    tree[k] += x;
    for (k /= 2; k >= 1; k /= 2) {
        tree[k] = tree[2*k]+tree[2*k+1];
    }
}
```

First the value at the bottom level of the tree is updated. After this, the values of all internal tree nodes are updated, until the top node of the tree is reached.

Both the above functions work in $O(\log n)$ time, because a segment tree of n elements consists of $O(\log n)$ levels and the functions move one level higher in the tree at each step.

Other Queries Segment trees can support any range queries where we can divide a range into two parts, calculate the answer separately for both parts, and then efficiently combine the answers. Examples of such queries are minimum and maximum, greatest common divisor, and bit operations and, or, and xor.

For example, the segment tree in Fig. 9.14 supports minimum queries. In this tree, every node contains the smallest value in the corresponding array range. The top node of the tree contains the smallest value in the whole array. The operations can be implemented like previously, but instead of sums, minima are calculated.

The structure of a segment tree also allows us to use a binary search style method for locating array elements. For example, if the tree supports minimum queries, we can find the position of an element with the smallest value in $O(\log n)$ time. For example, Fig. 9.15 shows how the element with the smallest value 1 can be found by traversing a path downwards from the top node.

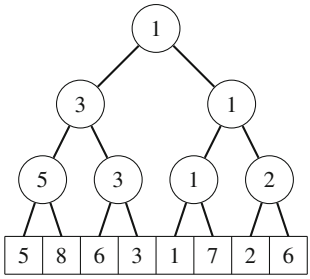


Fig. 9.14 A segment tree for processing range minimum queries

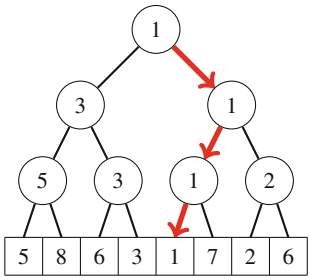


Fig. 9.15 Using binary search to find the minimum element

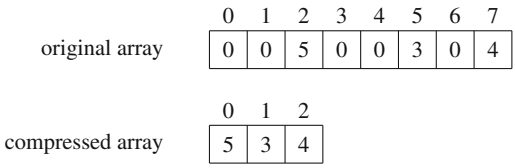


Fig. 9.16 Compressing an array using index compression

9.2.3 Additional Techniques

Index Compression A limitation in data structures that are built upon arrays is that the elements are indexed using consecutive integers. Difficulties arise when large indices are needed. For example, if we want to use the index 10^9, the array should contain 10^9 elements which would require too much memory.

However, if we know all the indices needed during the algorithm beforehand, we can bypass this limitation by using *index compression*. The idea is to replace the original indices with consecutive integers 0, 1, 2, and so on. To do this, we define a function c that compresses the indices. The function gives each original index i a compressed index $c(i)$ in such a way that if a and b are two indices and $a < b$, then $c(a) < c(b)$. After compressing the indices, we can conveniently perform queries using them.

Figure 9.16 shows a simple example of index compression. Here only indices 2, 5, and 7 are actually used, and all other array values are zeros. The compressed indices are $c(2) = 0$, $c(5) = 1$, and $c(7) = 2$, which allows us to create a compressed array that only contains three elements.

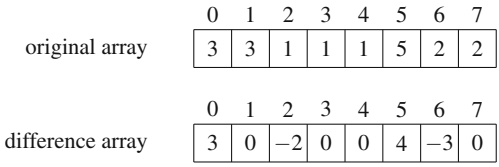


Fig. 9.17 An array and its difference array

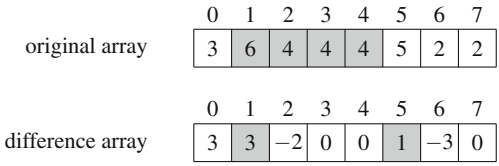


Fig. 9.18 Updating an array range using the difference array

After index compression, we can, for example, build a segment tree for the compressed array and perform queries. The only modification needed is that we have to compress the indices before queries: a range $[a, b]$ in the original array corresponds to the range $[c(a), c(b)]$ in the compressed array.

Range Updates So far, we have implemented data structures that support range queries and updates of single values. Let us now consider an opposite situation, where we should update ranges and retrieve single values. We focus on an operation that increases all elements in a range $[a, b]$ by x.

It turns out that we can use the data structures presented in this chapter also in this situation. To do this, we build a *difference array* whose values indicate the differences between consecutive values in the original array. The original array is the prefix sum array of the difference array. Figure 9.17 shows an array and its difference array. For example, the value 2 at position 6 in the original array corresponds to the sum $3 - 2 + 4 - 3 = 2$ in the difference array.

The advantage of the difference array is that we can update a range in the original array by changing just two elements in the difference array. More precisely, to increase the values in range $[a, b]$ by x, we increase the value at position a by x and decrease the value at position $b + 1$ by x. For example, to increase the original array values between positions 1 and 4 by 3, we increase the difference array value at position 1 by 3 and decrease the value at position 5 by 3 (Fig. 9.18).

Thus, we only update single values and process sum queries in the difference array, so we can use a binary indexed tree or a segment tree. A more difficult task is to create a data structure that supports *both* range queries and range updates. In Sect. 15.2.1, we will see that also this is possible using a lazy segment tree.