Dynamic prefix-sums

Rossano Venturini[1]

1 **Computer Science Department, University of Pisa, Italy**
rossano.venturini@unipi.it

Notes for the course *"Competitive Programming and Contests"* at Department of Computer Science, University of Pisa.
Web page: `https://github.com/rossanoventurini/CompetitiveProgramming`

These notes sketch the content of the 5th lecture. These are rough and non-exaustive notes that I used while lecturing. Please use them just to have a list of topics of each lecture and use the reported references to study these arguments.

1 Static prefix-sums

▶ **Problem 1** (Range sum). *Given an array $A[1, n]$ of integers, we would like to support the query* Sum(i)*, which returns the sum of the elements in $A[1, \ldots, i]$.*

The array doesn't change so the problem is trivial. Simply precompute the answers to all the possible (only n) queries. The operation Sum can be used to answer RangeSum queries. Given two indexes i and j, RangeSum(i, j) returns the sum of the elements in $A[i \ldots j]$. The latter query is very useful to solve other problems. The next one is an easy example.

▶ **Problem 2** (Ilya and Queries). *Given a string $s = s_1 s_2 \ldots s_n$ consisting only of characters a and b and m queries. Each query is described by a pair of integers l, r $(\leq l < r \leq n)$. The answer to the query l, r is the number of such integers $i \in [l, r]$ that $s_i = s_{i+1}$.*

The idea is that of computing the binary vector $B[1, n-1]$ such that $B[i] = 1$ if $s_i = s_{i+1}$, 0 otherwise. This way, the answer to the query l, r equals $\sum_{i=l}^{r} B[i]$. Thus, the problem is solved efficiently (constant time per query) by computing prefix-sums on vector B.

▶ **Problem 3** (Number of Ways). *Given an array $A[1, n]$, count the number of ways to split all the elements of the array into three contiguous parts so that the sum of elements in each part is the same.*

More formally, you need to find the number of such pairs of indices i and j $(2 \leq i \leq j \leq n-1)$ such that $\sum_{k=1}^{i-1} A[k] = \sum_{k=i}^{j} A[j] = \sum_{k=j+1}^{n} A[k]$

Let S be the sum of the values in the array. If 3 does not divide S, we conclude that the number of ways is zero. Otherwise, we compute the array C which stores in position i the number suffixes of $A[i \ldots n]$ that sum to $\frac{S}{3}$. We then compute the sum of the prefixes of A. Every time a prefix i sums to $\frac{S}{3}$, we add $C[i+2]$ to the result.

▶ **Problem 4** (Little Girl and Maximum). *We are given an array $A[1, n]$ and a set Q of q queries. Each query is a range sum query i, j which returns the sum of elements in $A[i \ldots j]$.*

The goal is to permute the element in A in order to maximize the sum of the results of the queries in Q.

We need to sort the entries of the array by their access frequencies and then assign the largest values to the most frequently accessed slots. We could use an array to keep track of the frequencies of the entries. However, let S be the sum of the length of queries size, just updating the above array would cost $\Theta(S)$. Note that S may be $\Theta(qn)$, which may be even worse than quadratic in n.

Thus, we need a cleaver way to compute those frequencies. The idea is to construct an array $F[1 \dots n]$, initially all the entries are set to 0. If we read the query $\langle l_i, r_i \rangle$, we add 1 to $F[l_i]$ and we subtract -1 to $F[r_i + 1]$. The prefix sum of F up to i equals the frequency of entry i. This algorithm costs only $O(q + n)$ time.

2 Dynamic prefix-sums: Binary indexed tree (BIT)

▶ **Problem 5** (Dynamic prefix-sums). *Given an array $A[1, n]$ of integers, we would like to support the following queries*

- Sum(i) *that returns the sum of the elements in $A[1, \dots, i]$;*
- Add(i, v) *that adds the value v to the entry $A[i]$.*

Binary indexed tree (BIT) (also known as Fenwick tree) solves the above queries in $\Theta(\log n)$ time by using linear space. Actually, BIT is an implicit data structure, which means that it uses only $O(1)$ space in addition to the space required to store the input data (the array A in our case).

Consider the following array A (note we are using one-based array).

	1	2	3	4	5	6	7	8
A	3	2	-1	5	7	-3	2	1

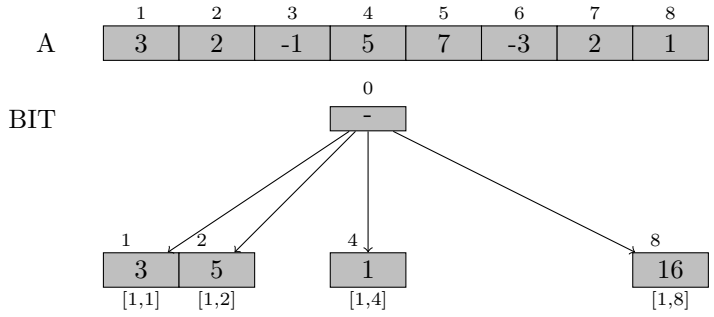
There are two possible trivial solutions to this problem.

1. Store A. Sum(i) is solved by scanning the array in $\Theta(n)$ time and Add(i, v) is solved in $O(1)$ time.
2. Store prefix-sums of A. Sum(i) is solved in $O(1)$ time and Add(i, v) is solved by modifying all the entries up to position i in $\Theta(n)$ time.

Both these solutions are clearly unsatisfactory.

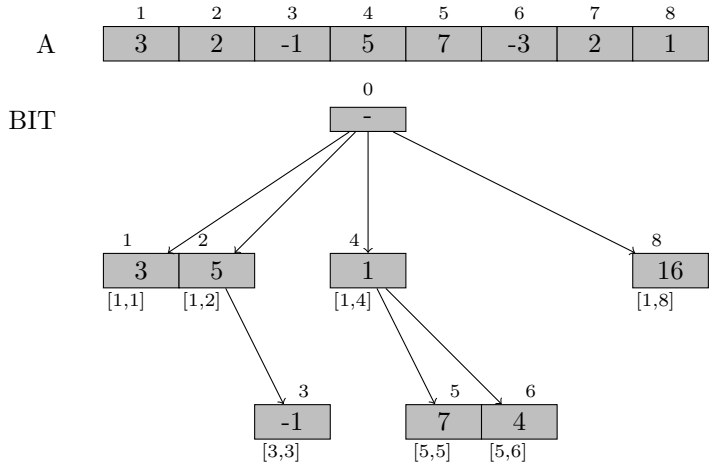
In order to approach this problem, let us assume that we only solve Sum queries on positions which are a power of 2 (i.e., positions $1, 2, 4, 8$ in our array A).

The idea for solving this relaxed version is to sparsify the latter trivial solution: we only store prefix-sums of positions that we could query. The figure below shows the solution for this particular case. For convenience we represent the solution as a tree. The root is a fake node named 0. The children of this node are nodes (named, $1, 2, 4, 8$) which store the sum up to the corresponding power of 2. Below every node we report the range covered by the corresponding value. For example, node 4 covers the range of positions in $[1, 4]$.



We solve the queries as follows. The query $\mathsf{Sum}(i)$ is easy. We simply access the node i and, thus, it takes constant time. Obviously, it works only for i which is a power of 2. The query $\mathsf{Add}(i, v)$ needs to add v to all the nodes covering ranges that contain the entry i. For example, for query $\mathsf{Add}(3, 10)$ we add the value 10 to nodes 4 and 8. In general, given i, we compute the smallest power of 2 which is larger than i, say j. Then, we add v to nodes $j, 2j, 2^2j, 2^3j, \ldots$. Thus, the query takes $\Theta(\log n)$ time.

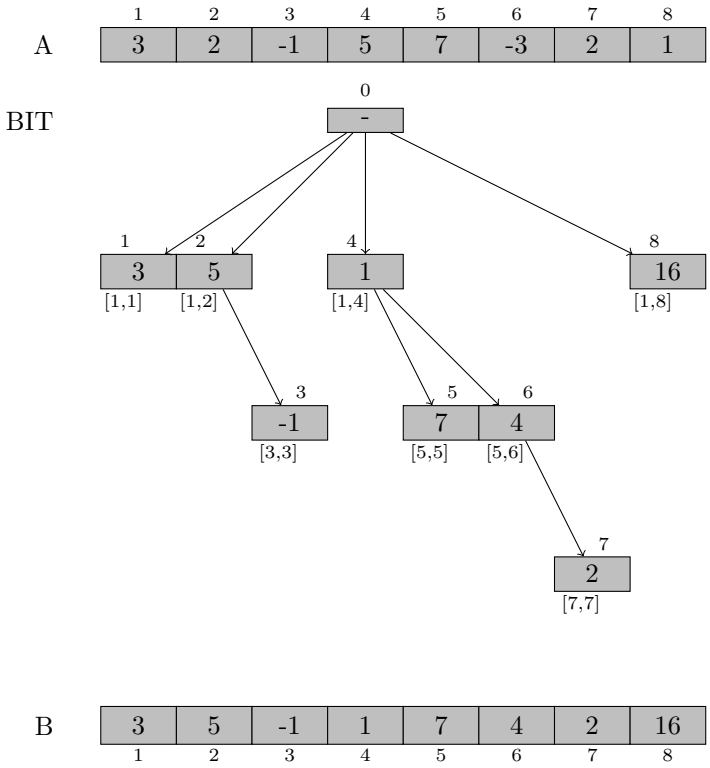
Now, is it possible to extend the above solution to support query Sum on more positions? Observe that we are not supporting the query on positions that belong to any range which is between to consecutive powers of 2. For example, positions in $[5, 7]$, which are the positions between 2^2 and 2^3. But wait, supporting the query on this subarray is just a smaller instance of our original problem. Thus, we can apply exactly the same strategy by adding a new level on our tree. If the subarray is $A[l, \ldots, r]$, the new level will support query $\mathsf{Sum}(i)$ for any i such that $i - l + 1$ is a power of 2.



Our two-level tree is now able to support $\mathsf{Sum}(i)$ queries also for any position i which is the sum of two powers of 2. Why? Well, let i be $2^{k'} + 2^k$ with $k' > k$. The range $[1, i]$ can be decomposed in two subranges $[1, 2^{k'}]$ and $[2^{k'} + 1, 2^{k'} + 2^k = i]$. Both these subranges are covered by nodes of the tree. Indeed, range $[1, 2^{k'}]$ is covered by node $2^{k'}$ at the first level, while $[2^{k'} + 1, 2^{k'} + 2^k = i]$ is covered by node i at the second level. As an example, consider the query $\mathsf{Sum}(5)$. This can be solved in our two-level tree because $5 = 2^2 + 2^0$. Thus, the range $[1, 5]$ is decomposed in $[1, 4]$ and $[5, 5]$ and the result (which is 6) is obtained by summing the values of nodes $2^2 = 4$ and $2^2 + 2^0 = 5$.

What are now the positions for which Sum is not supported? Any position i such that i is neither a power of 2 nor a sum of two powers of 2. In our example, as $n = 8$ only position

$7 = 2^2 + 2^1 + 2^0$ has such form. And now what? Let's add a new level on our tree to support the query on positions which are the sum of three powers of 2.



That's all. This is the binary indexed tree of the array A. Now some easy observations.

- Even if we gave a tree shape to our solution, it can be represented as an array B of size $n + 1$ as shown in the figure above.
- We no longer need the original array A as any of its entries i can be obtained as $A[i] = \mathsf{Sum}(i) - \mathsf{Sum}(i - 1)$.
- Let h be $\lfloor \log(n) + 1 \rfloor$, i.e., the length of the binary representation of any position in $[1, n]$. As any position is the sum of at most h powers of 2, the tree has no more than h levels. Actually, the number of levels is either h or $h - 1$ depending on the value of n.

Now let's see more precisely how to solve our queries Sum and Add on a BIT. As the solution is called binary indexed tree, it seems a good idea to consider the binary representation of the positions while solving a query. Let's start with $\mathsf{Sum}(i)$. For example, consider the case $i = 7$. The binary representation of 7 is $(\texttt{0111})_2$, so 7 is the sum of three powers of 2, and, thus, the range $[1, 7]$ is split in three subranges $[1, 4]$, $[5, 6]$, and $[7, 7]$. These subranges are covered by nodes 4, 6, and 7, one for each level of the tree. The sum of the values in these nodes is the result of the query.

Let's take a look to the binary representations of the ids of nodes involved in answering this query.

7	$(\texttt{0111})_2$
6	$(\texttt{0110})_2$
4	$(\texttt{0100})_2$

Any pattern? Well, binary representations of these nodes are obtained by removing least significant bit from the binary representation of 7. "*Coincidence? I think NOT!*" cit.

Let's see why it works this way. A range $[1, i]$ should be split into subranges. The first subrange is $[1, j]$ where j is the largest power of 2 smaller than i, or, if you prefer, $[1, j]$ is the longest prefix of $[1, i]$ which length is a power of 2. Given i, what is j? Just keep the most significant bit in the binary representation of i (node 4 in our example). Nice! That's consistent with our example. Now, the subrange $[j + 1, i]$ should be further split. How? Find the longest prefix $[j + 1, j']$ of $[j + 1, i]$ which length is a power of 2. Ok, that's exactly what we have done before but on the subrange $[j + 1, i]$. Given i, what is j'? Just keep the first 2 most significant bit in the binary representation of i (node 6 in our example). We continue this way until we completely cover the range $[1, i]$. How many subranges do we need? One for each bit set to 1 in the binary representation of i.

Now we need a bittrick to easily obtain the ids of nodes involved in any query $\mathsf{Sum}(i)$. By discussion above, it is clear that we need a way to remove the least significant bit from the binary representation of i. Iterating we will obtain the ids of all the required nodes.

The least significant bit in the binary representation of i can be isolated by computing $k = i\& - i$. Thus, $i - k$ is the binary representation of i without its least significant bit.

For example,

7	$(\texttt{0111})_2$	
-7	$(\texttt{1001})_2$	two's complement
7 & -7	$(\texttt{0110})_2$	

Let's now consider the query $\mathsf{Add}(i)$. We need to modify the values of nodes which range contains position i. For sure, node i is one of such nodes. Indeed, its range ends at i. Also right siblings of node i contain position i in their ranges. This is because ranges of siblings have the same starting position and right siblings have increasing sizes. Also the right siblings of the parent of node i contain position i. And the right siblings of the grand-grandparent of node i. And so on. Apparently, it seems that we have to change the value of a lot of nodes. However, a simple observation shows that this number is at most $\log n$. Indeed, every time we move from a node to its right sibling or to the right sibling of its parent, the size of the covered range at least double.

It remains to show how to find the list of nodes to modify given a position i. As we said above, node i is the first one. The next node can be obtained with a couple of easy bittricks.

Consider $i = 5$. The next node is its closest right sibling 6. Let's take a look to their binary representation.

5	$(\texttt{0101})_2$
6	$(\texttt{0110})_2$

Any pattern? Isolate the least significant bit in 5 ($(\texttt{0001})_2 = 1$) and sum it to 5 to get 6. *It could work!* cit. Let's try with another one. Right sibling of the parent of 6 (and, thus, of 5) is 8.

6	$(\texttt{0110})_2$
8	$(\texttt{1000})_2$

The least significant bit in 6 is $(0010)_2 = 2$ and $6 + 2 = 8$. Cool!

Why is this correct? It is easy to see that the position of least significant bit distinguishes among the siblings of a node. This means that we can move to the next sibling by moving the least significant bit one position to the left. That's exactly what happen when we sum 5 and its least significant bit. The least and the second least significant bits in the last children of a node are consecutive. In this case, if we sum the least significant bit, the least significant bit disappear and the second least significant bit is moved one position to the left. Thus, we are moving to the right sibling of the parent.

3 Problems

▶ **Problem 6** (Inversions count). *We are given an array $A[0 \dots n-1]$ of n distinct positive integers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an inversion of A. The goal is to count the number of inversions of A.*

Assume the integers are smaller than $n + 1$. If not, sort A and replace each element with its rank in the sorted array. Then, we use a BIT to index an array B of n elements, initially all set to 0. Now, we scan the array from left to right. When processing $A[i]$, we set $B[i]$ to 1. The number of element larger than $A[i]$ that we have already processed is $i - \mathsf{Sum}(i)$.

▶ **Problem 7** (Update the array). *We are given an array $A[1, n]$, initially all the entries are set to 0. we need to perform a sequence of updates and accesses on it. Each update specifies l, r and v which are the starting index, ending index and value to be added. The update adds v to all the entries in $A[l, r]$. An access i requires to return the current value of $A[i]$.*

We use a BIT B. For the update l, r, v we add v to $B[l]$ and we subtract v to $B[r + 1]$. This way, the value at position i is simply the prefix sum up to $B[i]$.

Note: the updates of the previous problem are different than the more difficult range updates. In the latter case we are also asked to add a value in a range but the goal is to support prefix-sums queries. BIT can be extender to support this operation as described here `http://petr-mitrichev.blogspot.com/2013/05/fenwick-tree-range-updates.html`.