Fenwick Tree Guide

by fbrunodr

April 17, 2023

1 Introduction

Guide to use Fenwick Tree implemented in https://github.com/fbrunodr/CompetitiveProgramming.

2 Fenwick Tree

Cumulative arrays are really useful, as they allow us to perform range queries fast. Let $A = \{a_i \in T, i \in [1, n]\}$ be an array of type T. Let $\oplus$ be a binary operator over elements of T. Lets define a query over a range:

$$Q(i, j) = a_i \oplus a_{i+1} \oplus a_{i+2} \oplus ... \oplus a_{j-1} \oplus a_j$$

If there is an inverse operation $\ominus$, we can answer those queries in constant time w.r.t. n using a cumulative array:

```
cumulative = vector<T>(n);
cumulative[0] = A[0]
for(int i = 1; i < n; i++)
    cumulative[i] = op(cumulative[i-1], A[i]);

auto query = [&cumulative](int i, int j){
    if(i == 0) return cumulative[j];
    return inv(cumulative[j], cumulative[i-1]);
}
```

Here, `op` stands for $\oplus$ and `inv` stands for $\ominus$.

Notice it takes $O(n \cdot T_{op})$ time to build the cumulative array and $O(T_{inv})$ to answer each query, where T_{op} is the time to perform $\oplus$ and T_{inv} is the time to perform $\ominus$.

This is great, but what if the data in the array A is being updated in between queries? Updating the cumulative array would take $O(n \cdot T_{op})$ time, which is way too expensive if there are a lot of updates. That is when Fenwick Trees become useful: they allow us to update in $O(log(n) \cdot T_{op})$ time while answering queries in $O(log(n) \cdot T_{op} + T_{inv})$.

https://codeforces.com/blog/entry/57292 gives an awesome explanation on how they work and prove their time complexities. The key insight I will try to bring here is **when** we can use this data structure.

Taking a closer look at the implementation, we see that this data structure only works if:

- $\oplus$ is associative. For example, to get $Q(1, 6)$ we do $Q(1, 4) \oplus Q(5, 6)$. If the order of the operations matter, we may not be able to do this.

- There is an inverse $\ominus$. Notice this is not the same as requiring there is an inverse element $y \in T$ for every element $x \in T$. For example, let $T = \mathbb{Z}$ and $\oplus$ be the usual multiplication. No element other than 1 has an inverse in $\mathbb{Z}$, yet we can still undo operations using the usual division.

- $\oplus$ is commutative. When we update the element a_3 to a_3', we update the range that saves $Q(1, 4)$ through either a left hand or right hand operation. This does not make sense, unless we can change the order of the elements and rewrite $Q(1, 4)$ as $a_1 \oplus a_2 \oplus a_4 \oplus a_3$, so we can safely perform $\ominus a_3 \oplus a_3'$ in the right hand side.

2.1 Space Complexity

Fenwick Tree takes about twice the space as A. $O\left(n \cdot \texttt{sizeof}(T)\right)$.

2.2 Time Complexity

Building the Fenwick Tree takes $O(n \cdot T_{op})$ time. Each query is $O\left(log(n) \cdot T_{op} + T_{inv}\right)$. Each update is $O\left(log(n) \cdot T_{op}\right)$. Also, this data structure is remarkable for the low multiplicative constant of each operation, as the tree transversal is done using bitwise operations, which only take a single CPU cycle to execute (http://ithare.com/infographics-operation-costs-in-cpu-clock-cycles/).