

Introduction:

This is an article primarily to myself, to clarify my understanding of the Binary Indexed Tree (BIT). Maybe some competitive programmers that want insight into how BITs actually work may also benefit from it. But I would really appreciate feedback, because this is something that really makes me uncomfortable. This article presents the BIT scheme that I do prove runtimes for, but the main goal is to provide intuition behind and motivation for the BIT scheme.

The BIT is something I have felt shaky about and cowered in fear when seeing it on LeetCode. So in this article, I'll talk myself into reinventing the BIT scheme to placate future me.

Problem Statement:

I have a list of size N . I want to support 2 operations on the list: $\text{Range}(X)$ and $\text{Update}(X, V)$. $\text{Range}(X)$ returns the sum of the first X values in the list: $\text{List}[1] + \text{List}[2] + \dots + \text{List}[X]$. And $\text{Update}(X, V)$ updates the X th entry of the list to the value V . Sitenote: $\text{Range}(X)$ supports a $\text{Range}(I, J)$ operation that returns $\text{List}[I] + \text{List}[I+1] + \dots + \text{List}[J]$ since: $\text{Range}(I, J) = \text{Range}(J) - \text{Range}(I-1)$

Premise:

Let's work in a "universe" of a list of size 15, 1-indexed. So $N = 15$. All numbers, then, will be from 1-15 and can be expressed using 4 bits from 0b0001 to 0b1111. And Log will mean Log base 2. So given I'm using 4 bits, I have the bound: $\text{Log}(X) \leq \text{Log}(N) < \text{Log}(N+1) = \text{Log}(16) = 4$.

2 Naive Approaches and Buckets

There are 2 approaches on opposite ends of runtime spectrums: vanilla list and prefix sum.

Vanilla list has $O(N)$ Range and $O(1)$ Update whereas prefix sum has $O(1)$ Range and $O(N)$ Update. So depending on the ratio of Range:Update, one scheme may be better than the other if it's skewed enough to amortize the linear operation. But, say there are equal, 50-50 split of Range and Update operations. Then both these operations on average will be $O(N)$.

Vanilla list is just the list itself. $\text{Update}(X, V)$ is simply setting $\text{List}[X] = V$ and done. And $\text{Range}(X)$ adds together the first X elements, so that is $O(N)$.

Prefix sum is building an auxiliary list, prefixsum , where the I th element is $\text{Range}[I]$. Creating this list is done in $O(N)$, single pass from left to right, sweep. Set $\text{prefixsum}[1] = \text{List}[1]$ and then, from I spanning from left to right from 2 to N , $\text{prefixsum}[I] = \text{List}[I] + \text{prefixsum}[I-1]$. This is the most elementary example of dynamic programming. So clearly prefix sum supports an $O(1)$ Range approach because $\text{Range}(X) = \text{prefixsum}[X]$. But the tradeoff is now Update is $O(N)$ because now to implement $\text{Update}(X, V)$ first compute $dV = V - \text{List}[X]$ where dV is the change in the X th value. And then increment the $O(N)$ elements $\text{prefixsum}[X], \text{prefixsum}[X+1], \dots, \text{prefixsum}[N]$ all by dV .

The goal, then, is to break these expensive Range and Update tradeoffs both approaches have by compromising and having a new approach, the BIT scheme, perform both operations in $O(\text{Log}(N))$. Here, I would like to introduce the word and the idea of "bucket".

A bucket stores the sum of certain elements in the original, input List. And these "certain elements" are completely specified by their indices. So to me, I view buckets as a set of indices. And for brevity, for the rest of this article I may refer to $\text{List}[I]$ simply as I when talking about buckets. For example a bucket containing I and J means that bucket stores the value that is $\text{List}[I] + \text{List}[J]$. Conceptually, simply imagine the list I'm working with as: $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]$, so $\text{List}[I] = I$. (Yes, updates be damned). Continuing the spirit of being imprecise, let me define the size of a bucket as how many indices the bucket contains, or more precisely, how many elements it sums. Of course the actual bucket contains a single integer that is the sum.

Ok, so both prefix sum and vanilla list approaches have N buckets. I identify 2 guiding questions: what is the size of the bucket and for a given index, which buckets does it land into?

The prefix sum approach has these buckets be lined up in the auxiliary, prefixsum, list. The i th bucket stores the sum of the first i elements so the size, then, of buckets in the prefix sum approach is $O(N)$. And for a given index I , I is a member of $O(N)$ buckets that need to be updated when $List[I]$ is Update'd. These buckets are $prefixsum[I]$, $prefixsum[I+1]$, ..., $prefixsum[N]$.

As for the vanilla list approach, will the input $List$ itself has all the N buckets as elements of the list. These are singleton buckets where the i th bucket is simply contains $List[I]$. So the size of the i th bucket is $O(1)$ and literally 1 as the only member is I (or $List[I]$). And conversely, a given index I only lands into 1 bucket, the i th bucket. So also $O(1)$ for the second guiding question.

The two bucket-guiding questions give intuition for the runtimes of $Range$ and $Update$, respectively. The larger the bucket size, the less buckets need to be summed up to compute $Range(X)$. Since every bucket a given index X is a member of needs to be updated during $Update(X, V)$, it is not ideal for indices to contribute to too many buckets.

Preliminary Thoughts

Two more related guiding questions: 1) For computing $Range(X)$, which buckets do we use to compute this? 2) For computing $Update(X, V)$, for a given index X , which buckets do we place it in? And for both we want to involve $O(\log(N))$ buckets, so that the $Range$ and $Update$ operations will be $O(\log(N))$.

Range $\log N$

Consider 1) the buckets themselves. I want to introduce 2 concepts: prefixes and branching and then hint (well, blatantly, heavy-handedly) at the BIT scheme satisfying these.

Say we partition X into buckets $\{1-A\} \mid \{A+1-B\} \mid \{B+1-X\}$. To compute $Range(X)$ we sum the values of these 3 buckets, where bucket $\{I-J\}$ holds the sum of $[I + (I+1) + \dots + J]$ (again this is shorthand for sum of $[List[I], List[I+1], \dots, List[J]]$). For a given X , we want $O(\log(N))$ buckets

Compare this to the vanilla list approach to compute $Range(X)$: both use prefix sums. In this $\log(N)$ approach, to get $Range(X)$ I do: bucket $\{B+1-X\} + Range(B)$ where, again, bucket $\{B+1-X\}$ stores the value of the sum of $[B+1, B+2, \dots, X]$. And $Range(B)$ in turn I compute as bucket $\{A+1-B\} + Range(A)$ and $Range(A)$ in turn is bucket $\{1-A\}$. So there is a prefix nature of these $Range(I)$ subproblems: $Range(A)$ being a prefix to $Range(B)$ being a prefix to $Range(X)$. I view bucket $\{A+1-B\}$ as a "child" of bucket $\{1-A\}$ that extends the prefix that is $Range(A)$ to the prefix that is $Range(B)$. And lastly, bucket $\{B+1-X\}$ as a "child" of bucket $\{A+1-B\}$ that in turn extends the prefix that is $Range(B)$ to $Range(X)$. So these buckets build upon each other. Now consider the vanilla list $O(N)$ approach to computing $Range(X)$: $Range(X) = bucket\{X\} + Range(X-1)$ where bucket $\{I\}$ is the singleton bucket $\{I-I\}$, that is it simply stores the value I . Now $Range(X-1)$ in turn I compute as bucket $\{X-1\} + Range(X-2)$ and so I have $X + (X-1) + Range(X-2)$ and keep unrolling the Ranges till I get $1 + 2 + \dots + X$. This has the same prefix nature: $Range(X-2)$ is the prefix extending to $X-2$ and then the singleton bucket $\{X-1\}$ extends that to get $Range(X-1)$ and finally bucket $\{X\}$ extends $Range(X-1)$ to get $Range(X)$. Bucket $\{X\}$ is child of Bucket $\{X-1\}$ is child of Bucket $\{X-2\}$ and so forth.

So both this vanilla list and this $\log(N)$ approach essentially operate under the same prefix extending mechanism. Main difference seeming to be vanilla list is $O(N)$ and uses X terms/buckets to compute $Range(X)$ whereas a proposed $\log(N)$ approach uses only $\log(N)$ buckets. I will now contemplate potential implications of a $\log(N)$ scheme and how it would likely be different from the vanilla case: namely, the key idea of branching as opposed to the linear, non-branching approach

that is the vanilla list. Upon careful examination of this scheme, I believe it is natural that “branching” happen (TODO think about this, REQUEST FOR FEEDBACK) First point is that under the $\text{Log}(N)$ approach, not all of the buckets can be singleton buckets like the vanilla list case. Because the $\text{Log}(N)$ buckets need to cover all X numbers, $1-X$. Let’s say the second bucket $\{A+1 - B\}$ is non-singleton and contains some $J \neq B$, that is, $A+1 \leq J < B$. Important: Consider $\text{Range}(J)$. I believe it is “natural” to reuse the first bucket $\{1-A\}$ to get the prefix of $\text{Range}(A)$ and then take some other path of buckets to get to $\text{Range}(J)$. Note that we can’t use the second bucket of $\{A+1 - B\}$ because that would overshoot J . So the idea is that if a child of a bucket B is non-empty, it cannot be an only child and bucket B branches to more than one child where, again, all the children get to reuse the contents of bucket B which in turn reuses of its parents and so forth creating a happy prefix chain (of hopefully length $\leq \text{Log}(N)$).

Actually, just as I write this, I realize this suggests why a binary recursive scheme may be natural. This may be getting a bit ahead of myself, and next paragraph I will give a tiny preview of the BIT scheme, but these prefix and branching notions gave me an idea. TODO review this as it’s stream of consciousness, evaluate content/correctness and clarity/notation/try being consistent. Say we follow some Prefix of buckets in a chain that covers $\text{Range}(A)$ and then we reach the bucket $\{A+1 - B\}$ that is non-singleton. So it’s parent bucket, P , ends in A and is the last node on the prefix bucket chain that, together, covers $\text{Range}(A)$. From last paragraph discussion on branching to cover all elements, I believe this forces branching. Taking this bucket will jump to B , but intermediate ranges $[A+1 - J]$ for $A+1 \leq J < B$ must be accounted for. Say there are K such J ’s ($K = B - 1 - A$) Then say we have some strategy, S , to build a subtree that is also a child of the parent bucket P (so this entire subtree is a sibling to the subtree rooted at bucket $\{A+1 - B\}$) And its job is to cover all these K range queries: $\text{Range}\{A+1\}, \text{Range}\{A+2\}, \dots, \text{Range}\{B-1\}$. Now consider the descendants of the bucket $\{A+1 - B\}$, or the subtree rooted at that bucket. This bucket itself answers $\text{Range}(B)$ when extending its prefix chain and its descendants would answer range queries $\text{Range}(J)$ for $J > B$ for some, but not necessarily all, J . A thought experiment is what if this subtree rooted at the bucket $\{A+1 - B\}$ would answer K more range queries: $\text{Range}\{B+1\}, \text{Range}\{B+2\}, \dots, \text{Range}\{B+K\}$. If I assume the S is an “optimal” strategy for creating a subtree to answer K range queries, it feels natural to me to reuse this same S to create the subtree rooted at $\{A+1 - B\}$. Essentially, this suggests that a recursive approach at least makes sense to me. And a binary one at that because the parent bucket, P , has 2 children: a left subtree (say with a dummy root) that handles K Range queries and a right subtree that is rooted at bucket $\{A+1 - B\}$ which handles $K+1$ Range queries (K for the descendants and $+1$ for $\text{Range}(B)$ that bucket $\{A+1 - B\}$ the root handles).

So for the BIT scheme for $N=15$, we have a bucket $\{1-8\}$ so $\text{Range}(J)$ for $1 \leq J \leq 7$ have some strategy and this same strategy applies for $\text{Range}(K)$ for $9 \leq K \leq 15$. Basically, recursive nature of this arrangement. I’m definitely getting ahead of myself, but I can almost declare completion here itself. $\text{Range}(X)$ is $\text{Log}(N)$ because 3 cases: simplest is $X = 8$ so $\text{Range}(8) = \text{bucket}\{1-8\}$ and done, $1 \leq X < 8$ case recurse the left subtree so instead of universe $[1,15]$ the search is now in universe $[1,7]$ and we have eliminated half the buckets so given there are $O(N)$ buckets total, this will be $O(\text{Log}(N))$ if the recursion keeps eliminating half the buckets. Which is does because the last case is $8 < X \leq 15$ so universe has halved to $[9, 15]$ (after incorporating bucket $\{1-8\}$ into the sum what is left is $[9,15]$) and $\text{Range}(X)$ is computed in the same manner as would $\text{Range}(X-8)$ except there is an offset of bucket $\{1-8\}$ and recurse on the right subtree rooted at the bucket $\{1-8\}$. So the recursion on the right subtree for $\text{Range}(X)$ is mechanistically the same as recursion for $\text{Range}(X-8)$ on the left subtree except I have to do $O(1)$ extra work that is adding bucket $\{1-8\}$ to the answer. Thus $\text{Range}(X)$ in this potential binary bucket tree approach will be $O(\text{Log}(N))$. Quick note: the crucial idea here is a symmetry where bucket $\{1-8\}$ has a sibling subtree that follows the same strategy as the descendants of bucket $\{1-8\}$, meaning by symmetry, the number of buckets in the $[1,7]$ “left universe” and the

same as the number of buckets in the “right universe”, so at most there are $\log(N)$ steps given that each step halves the number of buckets that can be considered on the bucket chain that computes $\text{Range}(X)$.

Let me restate to explain the binary search, halving, logarithmic nature. $\text{Range}(X)$ is computed by traversing a chain of buckets and while in the middle of the chain, there is some prefix already computed upto $\text{Range}(P)$ and what’s left is $\text{Range}(P+1, X)$ or the remaining range. The elements the Range sums for $\text{Range}(S, E)$ is $E - S + 1$ and let me call this number the range “width”. And until $\text{Range}(X)$ is completely formed, there is at least 1 more bucket to take. and taking a bucket means adding its value to the sum and recursing right as this bucket is at the root of the right subtree. Buckets are designed such that they cover half(1 more than half) the indices and their descendant buckets cover the remaining half. Then each iteration after adding a bucket to the answer, the remaining “width” contracts by half.

I’m done with 1) but I just want to throw in a preview of the BIT scheme that respects a potential $\log(N)$ scheme that uses $\log(N)$ buckets to partition X . We want to use $\log(N)$ buckets. Presumably larger X s will require more buckets. Largest number is 15, so hypothetically being “inspired” by binary representation of 15 as 0b1111 and to demo the BIT scheme, I’ll use the buckets {1-8} | {9-12} | {13-14} | {15} to compute $\text{Range}(15)$ which is exactly what BIT does. Where each 1 in the binary representation has a corresponding bucket with size commensurate with the place of that 1. So take the number 11 or 0b1011. $\text{Range}(11)$ will use buckets {1-8} + {9-10} + {11}. Since these are just preliminary thoughts, the take away is that whatever scheme I use, I need at most $\log(N)$ buckets which this binary scheme complies with. Why? Well $X \leq N$ has $\log(X)$ bits and at most all of them will be “1”s and in this scheme, the number of buckets $\text{Range}(X)$ uses are the number of 1 bits in the binary representation of X . And note that both $\text{Range}(15)$ and $\text{Range}(11)$ use the same {1-8} bucket and extend it (bucket {1-8} is a parent to bucket {9-12} and {9-10} in the BIT scheme), so this conveniently agrees with aforementioned concepts of reusing prefixes and branching. And also note that going down the chain of buckets, the size of the buckets at least halves, as does the remaining range width, again consistent with a proposed binary, $\log(N)$ scheme.

Update $\log N$

Whew, that finished $O(\log(N))$ Range. Now consider 2): which buckets I place the index/value X in for the other $O(\log(N))$ operation, Update. So every bucket I place X in, I will have to adjust its value if I were to execute an $\text{Update}(X)$ value. So at most I can place X in 4 buckets as $\log(16) = 4 > \log(X)$ so the $\text{Update}(X)$ operation will have to touch at most 4 buckets.

Contrast this placing X in $\log(N)$ buckets approach with the naive prefix sum approach in where instead of $\log(N)$ buckets, X is placed in $O(N)$ buckets: bucket X , bucket $X+1, \dots$, bucket N .

Say X falls in multiple buckets. Now when I compute $\text{Range}(K)$ for $K \geq X$, I cannot double count X . So whatever chain of $(\log(N))$ buckets $\text{Range}(K)$ uses, it must use exactly 1 of these buckets. I also observe the following related constraint: to avoid double counting any indices, whenever an index X is a member of a bucket, it may not be a member of any descendant buckets. Buckets only extend prefixes of the parent chain of buckets leading to a given bucket, extension with new indices only, no double counting already covered indices.

Now if I contemplate the aforementioned binary recursive partitioning scheme that I showed as being $\log(N)$ Range, this constraint of having no descendants of a bucket contain any of the indices that bucket contains also enforces Update to be $\log(N)$ as well in a very similar manner. Starting from the top of the binary bucket tree at dummy node that is an empty range prefix, there are decisions to make at each node before going down a level, and there are $\log(N)$ levels. So at each node (and the fact that we reached a given node means X is greater than the prefix covered thus far.

Let me say that so far, the amount A has been covered), there is a left subtree of handling K range queries and a right subtree rooted at a bucket $\{A+1-B\}$ with size $K+1$ and whose descendants cover an additional K queries.

If $X > B$, then bucket $\{A+1-B\}$ and the entire left subtree will be ruled out so over half the candidate buckets are precluded from containing X , because X is too large and must then be in the descendants of bucket $\{A+1-B\}$. If $X = B$, then X is by definition in bucket $\{A-B\}$ and like the previous case, it cannot be in the left subtree because the maximum that can cover is up to $\text{Range}(B-1)$. And X cannot be in any of the descendants of bucket $\{A+1-B\}$ to avoid double counting it. So this is in fact the terminal case, the last bucket X will be a part of. Finally, if $X < B$ it will show up in bucket $\{A-B\}$ because, following this making-a-decision-at-each-node-and-descending-down-the-tree procedure, X is greater than the prefix so far, meaning $X > A$ so X lies in bucket $\{A+1-B\}$ again by definition and it may also show up multiple times in the left sub tree. However, the key insight again is this double counting constraint and since X lies in bucket $\{A+1 - B\}$, this precludes it from being in any of its descendants, so the rest of the right subtree is ruled out, +1 inclusion count of X in bucket $\{A+1 - B\}$ and then, boom, recurse this downward procedure but on the left subtree again where the size of the left subtree matches the size of an entire subtree sans its bucket $\{A+1 - B\}$ root that was ruled out. So level by level down we go at least ruling out half the candidate buckets X could lie in each time and at most placing X in a single bucket per step and by virtue of halving, the cap of steps is $\text{Log}(N)$.

Just like past Range section where I gave a preview of the Range computation for the actual BIT scheme where I used the extreme, highest, number 15, I'll do the same for the Update operation just like how the BIT scheme does it. But this time, the number that appears in the maximum amount of buckets is the extreme, smallest, number that is 1. Which is intuitive because smaller numbers appear in Range sums of all larger numbers so the smallest number will be used the most for all other numbers. The number 1 appears in bucket $\{1\}$, bucket $\{1-2\}$, bucket $\{1-4\}$, bucket $\{1-8\}$. So again, there's this power of 2 deal going on where 1 appears in 4 buckets staying within $\text{Log}N$ confines. And when I do $\text{Range}(X)$ pursuant to the BIT scheme, I must either get the 1 from bucket $\{1-8\}$ and potentially continue down that bucket path for all $X \geq 8$ or get the 1 from the remaining buckets. This means $X < 8$ so it's basically binary search, next decision node is if $X \geq 4$ (but < 8) use bucket $\{1-4\}$, else $X < 4$ and so forth.

A Binary Tree Scheme

This is the section that essentially should spell everything out exactly, at least for the 4 bit, $N=2^4=16$ case and hopefully for all powers of 2 N . The actual BIT, I maintain, can be viewed as a "compressed" version of this tree where we only focus on non-empty buckets. Consider this guiding picture: start TODO TODO paste this handrawn picture maybe instead of flat files, organize each note into it's own directory to hold the .tpy, .pdf, and image assets together. Maybe also gives room for other file types like animations, TODO/plan files, etc. end TODO