Introduction:

This is a note to myself. But I would really appreciate feedback, because this is something that really makes me uncomfortable. Like this presents the Binary Index Tree scheme that I can prove runtimes for but feel shaky as to well how to arrive at it. It's a topic I've seen on LeetCode and cowered in fear.

Problem Statement:

2 approaches prefix sum and vanilla list each with tradeoffs. Reconcilliation, say we have equal, 50-50 split of Range and Update operations.

Premise:

Let's work in a "universe" of a list of size 15, 1-indexed. 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: Log(X) <= 4. And let the list I'll work with actually be the list : [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]. This way rather than saying the Ith element of the list I can just say the number I, since the Ith element of the special list IS I.

Preliminary Thoughts

This is just some informal commentary. TODO, really think this out, REQUEST FOR FEEDBACK!

List of size N say I have N buckets. I could give these buckets an ordering and index them as buckets 1,2,....N. Bucket I could definitely have I as a contributor in that bucket and the number I would not show up in any buckets before the Ith bucket (ordering at play)

OK so 2 angles. 1) For computing Range(X), which buckets do we use to compute this? 2) For a given number I, which buckets do we place it in? And for both we want to invole O(Log(N)) buckets, so that the Range and Update operations will be O(Log(N)).

Range LogN

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} | {A+1 - B}| {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) + (I+2) + ... + 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,...,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 + ... + 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 Log(N) approach, not all of the buckets can be singleton buckets like the vanilla list case. Because the O(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 != B, that is, A+1 <= J < B. Important: Consider Range(J). I believe it is "natural" to reuse the first bucket {1-A} to get the prefix of Range(A) and then take some other path of buckets to get to Range(J). Note that we can't use the seond bucket of {A+1 - B} becuase 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 reusues of its parents and so forth creating a happy prefix chain (of length O(Log(N))).

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 Xs will require more buckets. Largest number is 15, so hypothetically being "inspired" by binary representation of 15 as 0b1111 and as a preview of the BIT scheme, I'll use the buckets {1-8} | {9-12} | {13-14} | {15}. 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. It 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 has log(X) bits and at most all of them will be "1"s and in this scheme, the number of buckets Range(X) uses are the number of 1 bits in the binary represenation of X. And note that both Range(15) and Range(11) use the same {1-8} bucket, so this conveniently agrees with aforementioned concepts of reusing prefixes and branching.

Just to be super-redundant (TODO delete this paragraph?), compare this approach to get Range(X), that is summing Log(N) buckets, with the vanilla list approach of a simple list. They both fundamentally have the same "prefix" structure but while the vanilla list approach is linear this approach may use branching. In the vanilla list approach, to compute Range(11), I do 11 + Range(10) which in turn becomes 11 + 10 + Range(9) to 11 + 10 + 9 + Range(8) and so forth summing the first 11 singleton buckets. In this computation, I do 1 + 2 + ... + 8 to get a prefix sum that is Range(8) and then Range(9) builds on that by adding 9 and Range(10) in turn builds on Range(9) by adding 10 and finally Range(11) builds on Range(10) by adding 11. Compare this to an approach that uses Log(N) buckets. First I use bucket {1-8} to get a prefix sum Range(8) and then the next bucket {9-10} builds on that prefix to get Range(10) In this approach, buckets cannot all be singleton buckets so that the Log(N) buckets are able to cover all N numbers, 1-N. And these buckets do build on each other. Bucket {9-10}'s "parent" is bucket {1-8}.

Update LogN

Whew, that finished O(Log(N)) Range. Now consider 2): which buckets we place the index/value I in for another O(Log(N)) operation, Update. So every bucket I place I in, I will have to adjust it's value if I were to execute an Update(I) value. (lol "I" as in me or "I" as in the variable name I) So at most I can place I in 4 buckets as log(16) = 4 > log(15) so the Update(I) operation will have to touch at most 4 buckets.

Contrast this placing I in Log(N) buckets approach with the naive prefix sum approach in where insteand of Log(N) buckets, I is placed in O(N) buckets: bucket I, bucket I+1, bucket I+2, ..., bucket N.

Say I falls in 3 buckets, bucket I, bucket I+A, and bucket I+B where bucket K means the Kth bucket when the buckets are ordered. Note that I always falls as the highest contributor to the Ith bucket. That is Bucket I = Bucket{J - I} where J <= I. Now when we compute Range(X) for X>=I, we cannot double count I. So whatever chain of (Log(N)) buckets Range(X) uses it must use exactly 1 of these buckets. Moreover, from the prefix nature of the buckets (child buckets extending the prefix sum of the chain ending at their parents) none of these 3 buckets can be descendants of each other. Because that would mean double counting I. Basically, avoiding double counting mandates some constraints.

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. It 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 LogN confines. And when I do 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 >= 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 >= 4 (but < 8) use bucket{1-4}, else X < 4 and so forth.

A Binary Tree Scheme

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