Problem Solving Workshop #1 Tech Interviews and Competitive Programming Meetup

February 13, 2016 https://www.meetup.com/tech-interviews-and-competitive-programming/

Instructor: Eugene Yarovoi (can be contacted through the group Meetup page above under Organizers)

More practice questions: leetcode.com, glassdoor.com, geeksforgeeks.org

Books: Elements of Programming Interviews, Cracking the Coding Interview

Have questions you want answered? Contact the instructor, or ask on Quora. You can post questions and follow the instructor and other people who write about algorithms.

Solutions

(i) Give any correct solution to this problem. It can be inefficient. Difficulty level: basic

This is very simple. We can just not keep any sort of data structure, and every time there's an update to col C, we update arr[C] directly. Then, we use the following O(N) procedure to find the column number:

```
def findCol(arr, x):
        currentOffset = 0
        for i in range (0, len(arr)):
                currentOffset += arr[i]
                if currentOffset > x:
                        return i
        return -1 #out of bounds
```

(ii) You want the *Click* operation to be very fast because it's a common operation. However, you don't care if *Resize* is very slow, since resizing columns is much less common. You can also take some time to pre-process the *columnSizes* array when you load the file. Give an algorithm that achieves *Click* in less than linear time with respect to *columnSizes.length*. Difficulty: mid-tier company interview

To do column lookups faster than the naive O(n), we will need to preprocess the data in some way. We can use a cumulative array -- an array that stores the pixel offset at which each column begins. For example, if our column widths are [2, 5, 3, 6], the cumulative array would be [0, 2, 7, 10] ([0, 2, 2+5, 2+5+3]).

Once you have the cumulative array, you can perform a binary search to find the index of the greatest number that is less than or equal to your target x (this is the rightmost column that starts to the left of your click, therefore the column you want).

When a ColResize operation occurs, we must update all the entries in the cumulative array that are to the right of the updated index. This means that in the worst case, we will have to rebuild almost the entire cumulative array, because it could be that the 0-th column width is updated.

(iii) Accomplish both *Click* and *Resize* efficiently. Difficulty: elite company interview

There is a large increase in the difficulty of the problem at this point. The thought process should be that we can't build any kind of data structure where we'd have to update most of it if one of the column widths changes. This is the fatal flaw with the solution from part (ii). It has to be that changes in the column widths are very "local", in the sense that only a small amount of data in our data structure changes if a column width changes.

Idea 1: Blocks

We can consider a "block" strategy, where we subdivide the array into blocks of size K (K is some value to be determined). For each block, we store the sum of all the column widths in that block. When performing a Click operation, we first use the block sums to determine which block the click is in (using the naive algorithm from part 1, but on the block sums), and then we inspect the column widths inside that block to determine the precise column (again via a naive method).

For example, if the original array is [2, 3, 4, 5, 6, 7, 8, 9, 10], we could choose to split it into 3 blocks of 3, like this (vertical bar denotes block separation): [2, 3, 4 | 5, 6, 7 | 8, 9, 10]. Then we would store an additional array having the sum of each block: [9, 18, 27] ([2+3+4, 5+6+7, 8+9+10]).

If we now have the Click x = 30, then we look at the block sum array [9, 18, 27] to determine our click is in block #2 (0-indexed). Then we subtract the offsets 9 and 18 from 30, and look for x = 3 in [8, 9, 10] (the column widths making up block #2). x=3 corresponds to column #0 in the block. Finally, we do an arithmetic calculation to return colNum = blockNum*colsPerBlock + colNumInsideBlock. Here blockNum is 2, colsPerBlock is 3, and colNumInsideBlock is 0, so the answer is 6 -- the click is in column #6 (0-indexed).

The complexity of ColResize is just O(1) with this strategy, because when a width changes, it has to be updated in the original array, and the width of the block must also get updated. This is only 2 entries that have to change.

The complexity of Click depends on K. If the blocks are of size K, the number of blocks is N/K, and the number of column widths in each block is K. At most, we will need to inspect all the blocks, and then all the column widths in the one block where the click occurs. The time complexity will therefore be O(N / K + K). If we set K = sqrt(N), this expression is minimized to O(sqrt(N)) for Click.

We can combine this approach with the approach from (ii) by storing cumulative arrays in each block instead of raw data, and storing the block sums themselves as a cumulative array. This improves Click to O(log N), but worsens ColResize to O(sqrt(N)) because affected cumulative arrays will need to be rebuilt when ColResize operations occur.

Idea 2: Segment Tree

Another possible solution is to use a segment tree. The idea of a segment tree is similar to the approach of blocks above, except instead of trying to pick an optimal block size (which was sqrt(n) above), we instead pick a block size of 2. However, we then hierarchically define a concept of "blocks of blocks" and so on.

So, if we had [2, 3, 4, 5, 6, 7, 8, 9, 10], we would create a block array like this: [5, 9, 13, 17, 10] (sum every two elements from the original array), and then a block-of-blocks array like this: [14, 30, 10] (sum every two elements from the previous array), and so on with [44, 10] and [54].

Thus the structure looks like:

```
[2, 3, 4, 5, 6, 7, 8, 9, 10]
[5,    9,    13,   17,   10]
[14,         30,         10]
[44,                     10]
[54                        ]
```

The spacing above is simply to help you understand which numbers are the sums of which other numbers -- the arrays don't leave any empty space in the implementation.

Here the *ColResize* operation proceeds similarly to what we had with the block approach, except resizes require us to update one entry in each level of the segment tree. That is, when we update a column width, we then update its corresponding block width, then the width of its block-of-blocks, and so on.

For example, if we update the 4 in the original array (column #2), changing it to 15:

```
[2, 3, 15, 5, 6, 7, 8, 9, 10]
[5,    20,    13,   17,   10]
[25,         30,         10]
[55,                     10]
```

The Click operation requires us to look at the highest level, and then based on that, to drill down to lower levels. So in the example above (the structure after the update), if we have x = 36, we first look at [65] to check bounds, then we see from [55, 10] that we want to drill into the first of those blocks (since 36 < 55). The 55 entry corresponds to [25, 30] in the next lower level, and since (36 >= 25), we should drill down into the 30 after subtracting the offset of 25. So now we're looking for x = 36-25 = 11 in the block of size 30. In other words we skipped over the first 25 pixels, and are now looking for the 11th pixel (0-indexed) in the block of 30. This block is represented by [13, 17] in the next level. 11 < 13, so now we look for x = 11 in the block of size 13, which corresponds to the raw entries [6, 7], and here we see that since 11 >= 6, x = 11 will be found in the second of those two. The final answer is therefore column #5 (0-indexed), the index of the 7 value.

Both *ColResize* and *Click* require O(1) work per level of the tree. There are O(log N) levels, so the complexity of each operation is O(log N).

You might recognize that this concept of segment trees is very general and can be used for more than just the current problem. It is useful for many problems involving finding sums, minima, etc. of subarrays in an array whose elements are frequently being updated.

Further reading

A different presentation of segment trees, using binary trees instead of arrays, is presented in the context of a somewhat similar problem here:
http://www.geeksforgeeks.org/segment-tree-set-1-sum-of-given-range/

There's another similar data structure that can be used to implement this same idea, called a Fenwick tree. I wrote a post a while back on the difference between segment trees and Fenwick trees that also explains the concept of segment trees and Fenwick trees in more detail, emphasizing the generality of the concept: http://qr.ae/RbYoS2 .

(iv) There is now a third possible operation, *Insert(index,width)*. It inserts a new column numbered *index*, causing the column that was previously numbered *index* to now be numbered *index+1*, the column previously numbered *index+1* to be numbered *index+2*, etc. The new column has the specified *width* in pixels. Difficulty: tad above (iii)

In order to be able to insert rows, we need to maintain something like a segment tree, but as a tree and not an array. Array-based structures cannot handle RowInsert operations efficiently since they are fixed-size, and simply using a growable array (list) won't help because insertions

can happen at any point in the sequence and not just at the end. Therefore, we think to use a tree (see the GeeksForGeeks presentation of segment trees as binary trees in the Further Reading section above).

Our first idea might be that each node can store, in addition to the sum of its children, an additional field indicating its column number. Unfortunately, this won't work, since the insertion of a column shifts the numbering of all the columns by 1, and we want to avoid having to edit every node when an insertion occurs.

From this line of thought, we get the idea that column numbering must also be stored in some indirect format. A natural idea is to have each tree node store the total number of descendants it has (not children, but overall descendants, which includes children of children, etc). Using this information, at the same time we determine the leaf node corresponding to the click coordinate, we can determine the column number. When inserts happen, the changes in node count need to be propagated up to the root, but that only involves editing logarithmically many nodes.

(v) Instead of being what it was in (iv), the third operation is *ResizeRange(start,end,newWidth)*. It resizes all columns numbered between *start* and *end* to have *newWidth*. This must happen very efficiently even if the range covered is very large -- you cannot simply apply the solution from (iii) to every affected column. (Difficulty: mid-level contest)

This portion requires a technique known as "lazy propagation". See this tutorial if you're interested: https://www.hackerearth.com/notes/segment-tree-and-lazy-propagation/

This is very advanced material, however.