Hai_dee ★ 961 Last Edit: September 21, 2019 10:32 PM 1.5K VIEWS

59

Code at end. Sorry in advance for typos, it's 2 am here in Australia, lol.

I initially though this was a dynamic programming question, and was even assuming I was aiming for O(n^2) because the max number of blocks is so low at 1000. Normally this means O(n^2). But after a little thinking, I realised this is not the best way to approach it at all. We can do it greedily!

We can model this entire question as a binary tree that we need to construct with a minimum max depth cost. Each of the blocks is a leaf node, with a cost of its face value. And then each inner node will be of cost `split`. nodes that are sitting at the same level represent work that is done in parallel. We know there will be len(blocks) - 1 of these inner nodes, so the question now is how can we construct the tree such that it has the minimum depth.

For example, a possible (not optimal) tree for the data set
[1, 2, 4, 7, 10] with split cost 3 is:
(Sorry for ascii art, 2 AM is too late at night to do this properly :))

```
........3
.....|....\
.....3.....\
../.....\....\
 3.......3....\
/..\..../..\...\
1...2...4...7...10
```

This tree has a maximum depth of 16 (3 -> 3 -> 3 -> 7).

So, how can we optimise the construction of this tree? Huffman's algorithm!

I'm going to assume you're familiar with Huffman's algorithm. If not, google it or read this. Note that it's traditionally used for building compression codes, but there is no reason we can't also use it here.
https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

Instead of actually building the whole tree, we can just keep track of the nodes the huffman algorithm still needs to consider, as the maximum depth below them. So put all the leaf node (blocks) onto a priority queue, and then repeatedly take the 2 smallest off and add `split` onto the biggest of the 2 (this is Huffman's algorithm, it's greedy) and put the result back onto the priority queue.

Once there is only one left, this is the depth of the root node, and we return it.

It is O(n-log-n) because we are making 2n - 1 insertions into a heap and n removals, and both heap insertion and removal have a cost of O(log-n). We drop the constants, giving a final cost of O(n-log-n).

And here is the code.

```
class Solution:
    def minBuildTime(self, blocks: List[int], split: int) -> int:
        heapq.heapify(blocks)
        while len(blocks) > 1:
            block_1 = heapq.heappop(blocks)
            block_2 = heapq.heappop(blocks)
            new_block = max(block_1, block_2) + split
            heapq.heappush(blocks, new_block)
        return blocks[0]
```

Small improvement: As kimS pointed out, we know that block_2 is the biggest, as the smallest came off the heap first. So we don't need that call to max, and we don't need to put block_1 into a variable. Thanks!

Type comment here... (Markdown is supported)

Post

joshualian1989 ★ 421 September 21, 2019 9:57 PM

Although the final code seems quite simple, the reverse way of thinking is not trivial.
I tried to do this in normal way using DFS + memo: each time double the worker and try to assign 0 to 2*num_worker to the largest blocks, but still got TLE.

The most valuable thing I learnt from LeetCode is to look at problems in a "reverse way", the "reverse" here means differently for different problems:

- process the data structure backwardly
- consider the complementary part
- think of the whole process in a reversed way
- using duality property

11 Show 1 reply Reply

C0R3 ★ 290 September 21, 2019 10:14 PM

You don't need max(block_1, block_2) as heappop always returns the minimum value from the heap. Therefore block_2 is always the maximum value:

```
class Solution:
    def minBuildTime(self, blocks: List[int], split: int) -> int:
        heapq.heapify(blocks)

        while len(blocks) > 1:
```