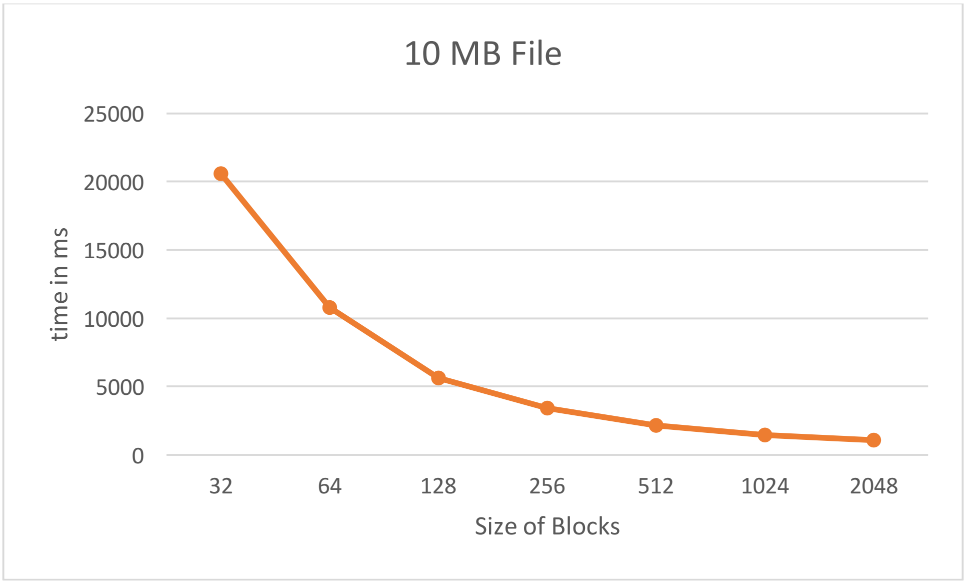
Kazi Rahma
PSO 12
Project 3 Report

For task 3, the file megabyte.txt produced a different sumhash than the one which was on the website.

Experimental Analysis

For the 10MB file:

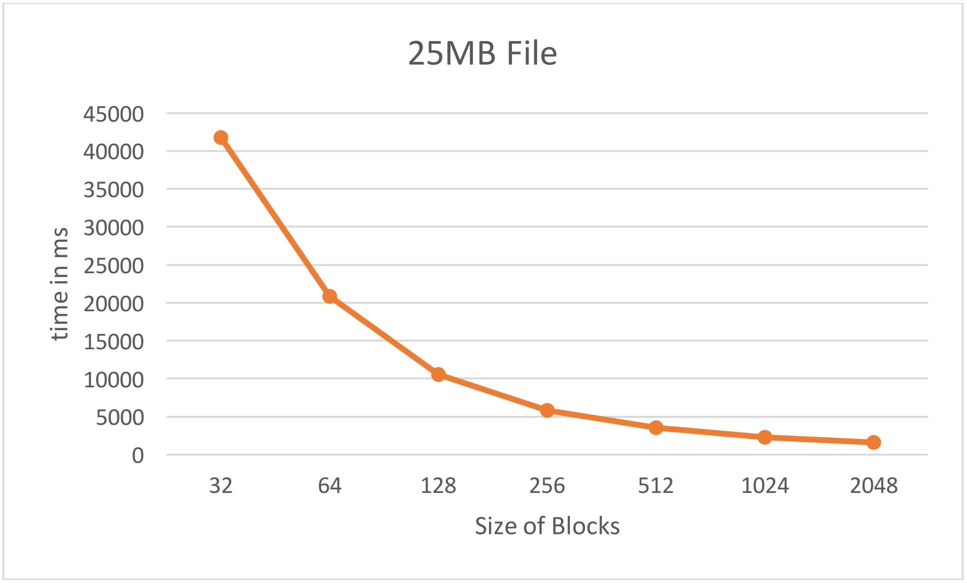
Block Size	Time in ms
32	20562
64	10759
128	5642
256	3417
512	2177
1024	1467
2048	1086



For 25MB file:

Block Size	Time in ms
32	41754
64	20859
128	10585
256	5797

512	3507
1024	2257
2048	1626



From the graphs we see that, for both the files, as we break the blocks into bytes of bigger sizes, the time taken to build the merkle tree decreases. When we first get a file, we partition it into blocks of a particular size bytes, and then hash those blocks to get the leaves. Then we hash the leaves to get the interior nodes. When the file is partitioned down to blocks of bytes of smaller sizes, the number of blocks increases which results in the increase of the number of leaves, which in turn increases the number of interior nodes; this results in higher time to hash than blocks with bytes of bigger sizes. For the 10MB file the time to build a merkle tree is lower than that for the 25MB file given that we are using the same sized bytes to partition the files. This is because the 25MB file has more data to be partitioned into. Hence, there will be more blocks and more interior nodes to be hashed. For example, if we partition both the 10MB file and the 25MB file in blocks of 128 bytes, then the 10MB file takes 5642ms to build and the the 25MB file takes 10585ms to built.

Analysis

1) We start building a merkle tree starting from its leaves to the root. The number of leaves corresponds to the number of blocks the file is partitioned into. So if we have m blocks then we have m leaves. It takes O(1) for performing a hash function. In order to hash m blocks to get m leaves, the time taken is m. After this it will take m-1 time to hash the interior nodes. So in total the time taken to hash would be m+m-1 = 2m-1. The asymptotic worst case is O(2m-1) = O(m).

2) For finding the path sibling we are first given a leaf to start with. So we start looking at its type and if it is the right child then its sibling is the left child and vice versa. From each level of the tree we are taking one node except for the level in which the given leaf u is in. From that level we are taking 2 nodes, the leaf u itself and its sibling. In my code I start with placing the leaf u into the list of path siblings first, which takes O(1) time. Then I use a while-loop to get the sibling of the current node (leaf u in the beginning), and iterate through the array and find the parent of the current nodes we are dealing with, if I find the parent then I ignore it and take its sibling either by taking the node from the index before the parent node or the index after the parent node depending on if the parent node is the right child or the left. Every time I find a parent I update my index to the index of the parent node in order to move up a level of the tree. This can also be considered as decrementing the index of the while loop by $\left\lfloor \frac{index}{2} \right\rfloor$ after each iteration. If we try to picture a tree then this while-loop is taking us one level up after each iteration. Hence, the while loop is executed (log m) times, which can also be viewed as the height of the tree, given there are m nodes in the tree. So the worst case is when the leaf u is on the last level and in total the asymptotic worst case would be O(1 + log m) = O(log m). The addition of 1 is due to dealing with the leaf u in the very beginning of my code.

3) As given in the handout, one Challenge-Response consists of two subtasks: one in the server in which we find the path siblings and another in the client when we build the tree using the list of path siblings given by the server. In the previous question I explained the way I determined the path siblings according to my code and got the worst case to me O(1 + log m). Once, I get the path siblings from the server, I can start building my tree in the client. In my code for the client, the very first thing I do is take the first two nodes from the list of path siblings and hash them to get the parent. Then I start a while-loop from the index 2 of the list containing path siblings till I reach the end of the list. We already know that I have (log m +1) number of nodes in my list of path siblings which is returned by the server to the client, and I am already hashing the first two to get the parent, before entering the while-loop which is O(2). So when beginning the while-loop I have (log m + 1 - 2)= (log m - 1) nodes in my list. Hence, my while-loop will execute (log m - 1) times in the client and perform (log m -1) hashes. Hence, the total number of hashes performed are (2 + log m - 1) in client. The worst case for client is O(2 + log m - 1)=O(1 + log m) and for server it is O(1 + log m). The total worst case is = O(1 + log m + 1 + log m) = O(2 + 2log m) = O(log m).

4) My root is at index 1. Its left child, u, is at index 2 and right child is in index 3. The left child of u is at index 4. According to the question we do not have the hashed values of left nodes starting from the left child of u, which is starting at index 4. Given that we have m leaves, $\frac{m}{2}$ leaves belong to the left side and $\frac{m}{2}$ leaves belong to the right side. Starting the list of path siblings from any node on the left side of the tree would converge to a point where they would require the value of the node at index 4, which is the left child of u. Since, we have lost the value of the node at index 4, we cannot perform the concatenation of the values of the nodes at index 4 and index 5. Hence, we cannot perform successful Challenge-Response for leaves on the left side of the tree starting from index 4. However, there can be successful Challenge-Response for leaves on the right side of the tree. Therefore, for $\frac{m}{2}$ leaf nodes there can be successful Challenge-Response.

5) For obtaining the master hash in this approach, we will concatenate the leaves and then get the hash of those. I think it will not be as secure as hashing the nodes up the way from leaves till the root. This is because in our previous approach in which we started hashing from the bottom and moved up the levels to get to the master hash, we had to hash two nodes to get the parent in each level. I believe this way is more secure because we would need to get the hash for every node in the tree which really makes sure that the master hash is correct for the specific file. But the new method to hash would not give the hash for all the nodes which is not as secure and accurate to get the master hash. One advantage of the new method is that it is going to faster than hashing all the nodes to get the master hash. The new method will be faster and an easier approach.