1199. Minimum Time to Build Blocks

Hard Greedy Math Heap (Priority Queue)

Leetcode Link

Problem Description

In this problem, we are given a series of tasks represented by blocks, where each block's value represents the amount of time required to complete that task (`blocks[i] = t`). Initially, there's only one worker available, capable of performing two types of actions:

1. Building a block, which completes that task.
2. Splitting into two workers, which allows for more tasks to be done in parallel.

There is a constraint on the second action: splitting a worker into two costs a fixed amount of time (`split`), regardless of how many splits occur simultaneously. The goal is to calculate the minimum time required to complete all tasks with the initial single worker, taking into account the time it takes to build each block and the possibility of splitting workers.

Intuition

The essence of the solution to this problem lies in understanding that it's more efficient to allow workers to build the longest taking blocks simultaneously while splitting workers to multiply their efficiency.

Here's the intuitive step-by-step reasoning for the algorithm:

1. **Use more workers on longer tasks first:** Prioritize assigning more workers to build the blocks that take the longest time first, because this minimizes overall waiting time.
2. **Split workers only when necessary:** It's often better to split workers early on so that they are available to work on parallel tasks, but doing so should be a calculated decision since each split incurs a fixed time cost.
3. **Utilize a min-heap to organize tasks:** By using a min-heap data structure (a binary heap where the parent node is less than or equal to its children), we can efficiently keep track of the smallest (quickest) tasks, which helps in determining when to split workers.

Here's how the algorithm is implemented:

- The `heapq` module in Python is used to convert the list of blocks into a min-heap in-place.
- Then, the process begins of repeatedly taking two elements off the top of the heap, which takes O(log n) time - the smallest element (the one that requires the least amount of time) and the second smallest.
- The smallest element (fastest task) is discarded, and the second smallest is replaced after adding the `split` time to it. This simulates a worker that's been split and then finished building the next quickest block.
- We continue this process until we're left with a single block in the heap, which now contains the minimum time needed to complete all tasks when considering split timing and building of tasks in parallel.
- Thus, the function returns the value of the sole remaining block in the heap, which is the answer to the problem.

This strategy works because it effectively simulates an optimal way of splitting workers and assigning tasks to them in a way that minimizes the overall completion time.

Solution Approach

The solution approach for this problem makes use of the heap data structure to efficiently manage the blocks while deciding the order in which to build them and when to split the workers. Here's a detailed breakdown:

1. **Heapify the blocks:** The `heapify` function from Python's `heapq` module is used to transform the list of blocks into a min-heap data structure in-place. A min-heap ensures that the smallest element is always at the root, which is important for efficiently accessing the quickest task to complete.

```
heapify(blocks)
```

2. **Processing the heap:** A while-loop runs as long as there is more than one block present in the heap. The purpose of this loop is to simulate the worker's actions in building blocks and splitting.

```
while len(blocks) > 1:
```

3. **Simulate block building and worker split:** Inside the loop, the `heappop` function is called twice. The first `heappop` removes and returns the smallest element from the heap, which represents the block that would finish the quickest. Since that worker will be split, we don't need that block's time anymore. The second `heappop` call gets the next smallest block's time which actually gets combined with the split time, simulating the time taken for a worker to split and then complete the next block.

```
heappop(blocks)  # this is the block taken by a worker who decides to split.
heapush(blocks, heappop(blocks) + split)  # Next quickest block plus the split time.
```

4. **Completing the build:** The loop continues until there is only one block (or time value) left in the heap. This remaining value represents the minimum time required to complete all tasks, considering all necessary splits and concurrent work.

5. **Return the result:** Finally, the last remaining value in the heap (the minimum time required to complete all tasks) is returned as the result.

```
return blocks[0]  # The minimum time to build all the blocks.
```

By using a heap data structure, this solution approach avoids the need to sort the blocks after each step, which keeps the overall time complexity down. Additionally, by always keeping the smallest elements (quickest tasks) at the front of the heap, it ensures that we're always making the most time-efficient decision for splitting workers or assigning them to build blocks.

Example Walkthrough

Let's consider an example with a set of tasks with block times expressed in units of time as follows: `blocks = [4, 2, 10]`, and a fixed split time of `split = 3`.

1. **Heapify the blocks:** First, we need to transform the `blocks` into a min-heap. We end up with a heap that might look like this: `[2, 4, 10]`, where the smallest task (taking 2 time units) is at the root.

```
heapify(blocks)  # blocks becomes [2, 4, 10]
```

2. **Processing the heap:** At the start of our while-loop, our heap has more than one block, so we proceed with the following steps inside the loop.

3. **Simulate block building and worker split:** We pop the smallest element, which is 2, from the heap. This is the task that the worker who decides to split can complete.

```
heappop(blocks)  # Removes 2, blocks is now [4, 10]
```

   The worker splits (adding a split time to the next task), and finishes the next block. We pop the next smallest block, which is 4, add the split time of 3, and push the resulting 7 back onto the heap.

```
heappush(blocks, heappop(blocks) + split)  # Removes 4, adds 3, and pushes 7, blocks is now [7, 10]
```

   Now we continue with the heap `[7, 10]`.

4. **Continue loop:** The while-loop runs again because we still have more than one block in the heap.

   We pop the smallest element, which is 7. Then pop the next smallest element, which is 10, add the split time of 3, and push the result back onto the heap.

```
heappop(blocks)  # Removes 7, blocks is now [10]
heappush(blocks, heappop(blocks) + split)  # Removes 10, adds 3, and pushes 13, blocks is now [13]
```

5. **Completing the build:** The loop breaks because we now have a single element in our heap: `[13]`. This means it will take a minimum of 13 units of time to complete all tasks when considering the necessity for workers to split and the concurrent processing of tasks.

6. **Return the result:** Since we have only one element in our heap, it represents the minimum time required to complete all tasks.

```
return blocks[0]  # returns 13, which is the minimum time to build all the blocks
```

In conclusion, by following this step-by-step approach to simulating the worker splits and block builds, our single worker would be able to complete all the tasks in a minimum of 13 units of time.

Python Solution

```
from heapq import heapify, heappop, heappush  # Import required functions from heapq module

class Solution:
    def minBuildTime(self, blocks: List[int], split: int) -> int:
        """
        Calculate the minimum time to build blocks with a given split time.

        :param blocks: A list of integers representing the time needed to build each block.
        :param split: An integer representing the time needed to split a worker into two.
        :return: The minimum time needed to build all the blocks.
        """

        # Transform blocks into a min-heap in place
        heapify(blocks)

        # Continue processing until there is only one block left
        while len(blocks) > 1:
            # Pop the smallest block (done by one worker) from the heap
            heappop(blocks)
            # Pop the next smallest block and add the split time
            # This simulates the split, next job acquisition and pushing it back
            heappush(blocks, heappop(blocks) + split)

        # The last element in the total time required to build all blocks
        return blocks[0]  # Return the minimum time
```

Java Solution

```
class Solution {
    public int minBuildTime(int[] blocks, int splitTime) {
        // Create a priority queue to hold the blocks, with the natural order (i.e., min-heap)
        PriorityQueue<Integer> blockQueue = new PriorityQueue<>();

        // Add all blocks to the priority queue
        for (int block : blocks) {
            blockQueue.offer(block);
        }

        // Continue combining blocks until there's only one block left
        while (blockQueue.size() > 1) {
            // Remove the two smallest blocks from the queue
            int fasterWorkerBlock = blockQueue.poll();
            int secondFasterWorkerBlock = blockQueue.poll();

            // The two workers work on the split and build the next blocks
            // Add the combined time back to the priority queue
            // The time is the time of the second worker plus the split time.
            blockQueue.offer(secondFasterWorkerBlock + splitTime);
        }

        // The remaining block in the priority queue requires the longest time to complete.
        // This will be the total time required to build all the blocks.
        return blockQueue.poll();
    }
}
```

C++ Solution

```
#include <vector>
#include <queue>

class Solution {
public:
    // Finds the minimum time required to build all blocks with a given split time.
    int minBuildTime(vector<int>& blocks, int split) {
        // Use a min heap to always process the block that requires the least amount of time.
        priority_queue<int, vector<int>, greater<int>> minHeap;

        // Insert all the blocks into the min heap.
        for (int blockTime : blocks) {
            minHeap.push(blockTime);
        }

        // Continue processing the blocks in the min heap until only one remains.
        while (minHeap.size() > 1) {
            // Remove the first block with the least building time.
            minHeap.pop();

            // Fetch the next block with the least building time.
            int nextBlockTime = minHeap.top();
            minHeap.pop();

            // Combine the two blocks by adding the split time to the next block.
            // This represents two workers: one splitting and the other building the next block.
            // Push the result back into the heap to force to finish the next block.
            minHeap.push(nextBlockTime + split);
        }

        // The value remaining in the min heap is the total time needed to build all blocks.
        return minHeap.top();
    }
};
```

Typescript Solution

```
function minHeapify(arr: number[], i: number, n: number) {
    // Function to heapify a subtree rooted with node i, which is an index in arr[]
    let smallest = i;
    let l = 2 * i + 1; // left child
    let r = 2 * i + 2; // right child

    // If left child is smaller than root
    if (l < n && arr[l] < arr[smallest]) {
        smallest = l;
    }

    // If right child is smaller than smallest so far
    if (r < n && arr[r] < arr[smallest]) {
        smallest = r;
    }

    // If smallest is not root
    if (smallest !== i) {
        // Swap
        [arr[i], arr[smallest]] = [arr[smallest], arr[i]];

        // Recursive call to heapify the affected sub-tree
        minHeapify(arr, smallest, n);
    }
}

function buildMinHeap(arr: number[]) {
    // Function to build a min-heap from a given array
    let n = arr.length;

    // Start from the last non-leaf node and heapify each node
    for (let i = Math.floor(n / 2) - 1; i >= 0; i--) {
        minHeapify(arr, i, n);
    }
}

function extractMin(arr: number[]): number {
    // Function to extract the minimum element from the heap
    let n = arr.length;

    // The minimum element is at the root of the heap
    let minElement = arr[0];

    // Move the last element to the root and reduce heap size
    arr[0] = arr[n - 1];
    arr.pop();

    // Heapify the root node
    minHeapify(arr, 0, arr.length);

    return minElement;
}

function insertHeap(arr: number[], value: number) {
    // Function to insert a new value into the heap
    arr.push(value);

    // Heapify from the last element up to the root
    let i = arr.length - 1;
    while (i > 0 && arr[Math.floor((i - 1) / 2)] > arr[i]) {
        [arr[i], arr[Math.floor((i - 1) / 2)]] = [arr[Math.floor((i - 1) / 2)], arr[i]];
        i = Math.floor((i - 1) / 2);
    }
}

// Finds the minimum time required to build all blocks with a given split time
function minBuildTime(blocks: number[], split: number): number {
    // Build a min-heap from the given blocks array
    buildMinHeap(blocks);

    // Continue processing the blocks until only one remains
    while (blocks.length > 1) {
        // Extract the two blocks with the least building time
        extractMin(blocks);

        // Fetch the next block with the least building time
        let nextBlockTime = extractMin(blocks);

        // Combine the two blocks by adding the split time to the next block
        // This represents two workers - one splitting and the other building the next block
        insertHeap(blocks, nextBlockTime + split);
    }

    // The value remaining in the heap is the total time needed to build all blocks
    return blocks[0];
}
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code can be analyzed as follows:

1. The first operation is `heapify(blocks)`, which turns the list into a min-heap. This operation has a time complexity of `O(n)`, where `n` is the number of elements in the `blocks` list.
2. The while loop runs until there is only one block left, so it will run `n-1` times, where `n` is the initial number of elements in `blocks`.
3. Inside the while loop, we perform two heap operations for each iteration: `heappop(blocks)` and `heappush(blocks, heappop(blocks) + split)`.
   - `heappop(blocks)` has a time complexity of `O(log n)` for each pop operation because it maintains the heap property after removing the smallest element.
   - `heappush(blocks, heappop(blocks) + split)` is composed of another `heappop(blocks)` which is `O(log n)` and `heappush(...)` which is `O(log n)`, making the combined operations also `O(log n)` for each push operation.

Since both heap operations happen each time in the loop, the total time complexity for the while loop is `O(n-1) * 2 * log n)`, which simplifies to `O(n log n)` considering that `n-1` is approximately `n` for large `n`.

Because the `heapify()` operation is `O(n)` and dominates for smaller `n`, while the while loop dominates for larger `n`, the overall time complexity of the code is `O(n log n)`.

Space Complexity

The space complexity of the provided code is analyzed as follows:

1. The `heapify(blocks)` operation is done in-place, and does not require additional space, so its space complexity is `O(1)`.
2. The while loop does not use any extra space other than the space used for the heap operations, which is also done in-place on the `blocks` list.

Therefore, the overall space complexity of the code is `O(1)`, as no additional space is needed beyond the input list.