Problem Description

Hard

required to complete that task (blocks[i] = t). Initially, there's only one worker available, capable of performing two types of actions: Building a block, which completes that task.

In this problem, we are given a series of tasks represented by blocks, where each block's value represents the amount of time

- 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.

workers.

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.

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

- 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
- 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

accessing the quickest task to complete.

to simulate the worker's actions in building blocks and splitting.

 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.

simulates a worker that's been split and then finished building the next quickest block.

- 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 heapy 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

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

the result.

fixed split time of split = 3.

inside the loop.

result back onto the heap.

processing of tasks.

class Solution:

heapify(blocks)

while len(blocks) > 1:

heappop(blocks)

8

9

10

11

12

13

14

15

16

17

19

20

21

22

23

24

10

11

12

13

14

16

17

18

19

20

22

23

24

25

26

27

28

29

30

31

32

33

35

34 };

6

9

10

11

12

1 while len(blocks) > 1:

1 heapify(blocks)

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

4. Completing the build: The loop continues until there is only one block (or time value) left in the heap. This remaining value

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

represents the minimum time required to complete all tasks, considering all necessary splits and concurrent work.

combined with the split time, simulating the time taken for a worker to split and then complete the next block.

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

we're always making the most time-efficient decision for splitting workers or assigning them to build blocks.

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

heappop(blocks) # This is the block taken by a worker who decides to split.

heappush(blocks, heappop(blocks) + split) # Next quickest block plus the split time.

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

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

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. 1 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

3. Simulate block building and worker split: We pop the smallest element, which is 2, from the heap. This is the task that the

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

heappop(blocks) # Removes 7, blocks is now [10]

able to complete all the tasks in a minimum of 13 units of time.

def minBuildTime(self, blocks: List[int], split: int) -> int:

:return: The minimum time needed to build all the blocks.

Continue processing until there is only one block left

Pop next smallest block and add the split time

heappush(blocks, heappop(blocks) + split)

public int minBuildTime(int[] blocks, int splitTime) {

while (blockQueue.size() > 1) {

Transform blocks into a min-heap in place

Calculate the minimum time to build blocks with a given split time.

Pop the smallest block (done by one worker) from the heap

The last element is the total time required to build all blocks

// Continue combining blocks until there's only one block left

// The two workers team up to split and build the next block;

// The time is the time of the second worker plus the split time.

// Remove the smallest two blocks from the queue

int secondFastestWorkerBlock = blockQueue.poll();

// add the combined time back to the priority queue.

// Fetch the next block with the least building time.

// Combine the two blocks by adding the split time to the next block.

// Once the split is done, they join forces to finish the next block.

// Function to heapify a subtree rooted with node i, which is an index in arr[]

// The value remaining in the min heap is the total time needed to build all blocks.

// This represents two workers: one splitting and the other building the next block.

int nextBlockTime = minHeap.top();

minHeap.push(nextBlockTime + split);

1 function minHeapify(arr: number[], i: number, n: number) {

// If right child is smaller than smallest so far

minHeap.pop();

return minHeap.top();

Typescript Solution

let smallest = i;

smallest = l;

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]) {</pre>

int fastestWorkerBlock = blockQueue.poll();

This simulates the split, next job acquisition and pushing it back

add the split time of 3, and push the resulting 7 back onto the heap.

worker who decides to split can complete.

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

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,

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

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

In conclusion, by following this step-by-step approach to simulating the worker splits and block builds, our single worker would be

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

: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.

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

25 return blocks[0] # Return the minimum time 26

```
// 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);
```

Java Solution

class Solution {

```
blockQueue.offer(secondFastestWorkerBlock + splitTime);
20
21
22
23
           // The remaining block in the priority queue requires the longest time to complete.
24
           // This will be the total time required to build all the blocks.
25
           return blockQueue.poll();
26
28
C++ Solution
 1 #include <vector>
2 #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;
10
           // Insert all the blocks into the min heap.
11
           for (int blockTime : blocks) {
12
               minHeap.push(blockTime);
13
14
15
16
           // Continue processing the blocks in the min heap until only one remains.
           while (minHeap.size() > 1) {
17
18
               // Remove the block with the least building time.
               minHeap.pop();
19
```

```
13
       if (r < n && arr[r] < arr[smallest]) {</pre>
14
            smallest = r;
15
16
17
       // If smallest is not root
18
       if (smallest !== i) {
19
           // Swap
20
            [arr[i], arr[smallest]] = [arr[smallest], arr[i]];
21
22
           // Recursive call to heapify the affected sub-tree
23
           minHeapify(arr, smallest, n);
24
25 }
26
   function buildMinHeap(arr: number[]) {
       // Function to build a min-heap from a given array
28
29
       let n = arr.length;
30
       // Start from the last non-leaf node and heapify each node
31
       for (let i = Math.floor(n / 2) - 1; i >= 0; i--) {
32
33
            minHeapify(arr, i, n);
34
35 }
36
   function extractMin(arr: number[]): number {
       // Function to extract the minimum element from the heap
38
       let n = arr.length;
39
40
       // The minimum element is at the root of the heap
       let minElement = arr[0];
43
44
       // Move the last element to the root and reduce heap size
45
       arr[0] = arr[n - 1];
       arr.pop();
46
47
48
       // Heapify the root node
49
       minHeapify(arr, 0, arr.length);
50
51
        return minElement;
52 }
53
54
   function insertHeap(arr: number[], value: number) {
55
       // Function to insert a new value into the heap
56
       arr.push(value);
57
58
       // Heapify from the last element up to the root
59
       let i = arr.length - 1;
60
       while (i != 0 && arr[Math.floor((i - 1) / 2)] > arr[i]) {
61
            [arr[i], arr[Math.floor((i - 1) / 2)]] = [arr[Math.floor((i - 1) / 2)], arr[i]];
62
            i = Math.floor((i - 1) / 2);
63
64 }
65
   // 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
68
69
       buildMinHeap(blocks);
70
71
       // Continue processing the blocks until only one remains
72
       while (blocks.length > 1) {
           // Remove the block with the least building time
73
74
            extractMin(blocks);
75
76
           // Fetch the next block with the least building time
77
            let nextBlockTime = extractMin(blocks);
78
```

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

Time and Space Complexity

n is the number of elements in the blocks list.

removing the smallest element.

return blocks[0];

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.

79

80

81

82

83

84

85

86

87

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

// Combine the two blocks by adding the split time to the next block

// The value remaining in the heap is the total time needed to build all blocks

insertHeap(blocks, nextBlockTime + split);

// This represents two workers — one splitting and the other building the next block

 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.

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

Since both heap operations happen each time in the loop, the total time complexity for the while loop is $0((n-1) * 2 * \log n)$, 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

- Space Complexity The space complexity of the provided code is analyzed as follows:
 - the blocks list.

complexity of the code is O(n log n).

1. The heapify(blocks) operation is done in-place, and does not require additional space, so its space complexity is 0(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

Therefore, the overall space complexity of the code is 0(1), as no additional space is needed beyond the input list.