2895. Minimum Processing Time

Greedy Array Sorting

Problem Description

Medium

tasks. We have n processors, each with 4 cores, meaning each processor can work on up to 4 tasks simultaneously. We also have n * 4 tasks that need to be completed, with the stipulation that each core can only execute one task at a time.

We are provided with two arrays: processorTime and tasks. The processorTime array tells us when each processor will become

available to start executing tasks, with the index of the array representing each processor. The tasks array indicates the time

In this problem, we are managing a set of tasks and processors in a way that optimizes the total time required to complete all

each individual task takes to execute. The objective is to determine the minimum amount of time needed to finish all n * 4 tasks when they are distributed across the n processors and their cores. It's a scheduling problem where we must figure out an optimal assignment of tasks to processors

to minimize the overall completion time.

Intuition

To develop the solution approach, consider if we had all processors available at the start, we would likely start with the longest

However, since processors become available at different times, we want to pair the longest tasks with the soonest available

tasks to get them out of the way while all processors are available. We would then fill in the shorter tasks where possible.

processors. This way, we can ensure that a task doesn't get unnecessarily delayed waiting for a busy processor when it could be started earlier by a free processor.

Here is how we arrive at our solution approach:

1. Sort the processorTime array in ascending order, so we know the order in which processors become available.

2. Sort the tasks array in descending order, so we have the longest tasks at the beginning of the list.

Once we have these sorted lists, we begin assigning tasks to processors. Starting with the first processor in processorTime (the

one that becomes available earliest), we'll assign it the current longest task available in tasks. Since each processor has 4 cores,

tasks need to be finished.

- we can assign up to 4 tasks at a time to each processor. So, for every processor, we will assign up to 4 of our current longest tasks, then move to the next processor that becomes available.
- To compute the minimum time all tasks are completed, we track the end time for each processor assigned tasks by adding the

starting from the earliest available will take four of the remaining longest tasks.

as the problem deals with direct array manipulation.

By implementing this greedy algorithm, we ensure that we're always using the earliest available processor time optimally by pairing it with the longest remaining tasks.

Solution Approach

longest task time to the processor's available time. The minimum completion time is the maximum of these end times since all

The implementation of the solution follows the <u>Greedy</u> approach combined with <u>sorting</u>, which is a common pattern used in optimization problems. Here's how the provided solution translates our intuition into an algorithm:

1. <u>Sorting</u> - We sort both the <u>processorTime</u> and <u>tasks</u> arrays. The former is sorted in ascending order, so we know which

processor will be available the soonest. The latter is sorted in descending order, ensuring we pick the longest tasks first.

2. **Greedy Assignment** - Once sorted, we assign the longest tasks to the earliest available processors - that is, each processor

3. Calculating Completion Time - For every processor, we calculate the end time which is when the processor becomes

available (processorTime) plus the time of the longest task assigned to it. This uses the max function to ensure we are only considering the time at which the process will have completed its longest task, which is indicative of its actual availability after its assigned tasks are done.

Data Structures - We use basic Python lists to represent the arrays and sort them. No advanced data structures are needed

5. **Iteration and Indexing** - We iterate over the processorTime array, and for each processor, we keep an index i that we decrement by 4 (the number of cores per processor) to always select the next set of tasks.

Let's walk through this process with a little more detail with respect to the provided solution code snippet:

We start by <u>sorting</u> both the <u>processorTime</u> and the <u>tasks</u> arrays with the built-in <u>.sort()</u> method.
We initialize <u>ans</u>, which will keep track of the current maximum completion time of all tasks.
We iterate through the <u>processorTime</u>, for each processor, we add its available time to the time of the largest pending task and compare it with the current <u>ans</u> to update the maximum completion time.

next processor. It's also crucial to note that this calculation gives us the earliest possible end time for each processor to finish one of its four tasks, not all four. This is because once we assign the four longest available tasks to a processor (if available), the

Note that this approach works since i is started at the end of the tasks array, saying that we always add the largest task to the

processor will not finish all of them at the same time, but the completion time will be bounded by the finish time of the longest

one.

Here's the reference solution approach, further elucidated:

• We decrement the index i by 4 to move on to the next set of tasks for the following processor.

- class Solution:
- def minProcessingTime(self, processorTime: List[int], tasks: List[int]) -> int:
 processorTime.sort() # Sort the processorTime array in ascending order
 tasks.sort() # Sort the tasks array in descending order (requires reversing since Python's sort is ascending...

The implementation confirms the intuition that earlier available processors should be assigned the longest tasks in a way that

ans = max(ans, t + tasks[i]) # Update the answer with the latest finish time for the current processor

i = len(tasks) - 1 # Start with the last index of the sorted tasks (the largest task)

for t in processorTime: # Iterate through each processor's available time

return ans # Return the final calculated answer

Let's walk through a small example to illustrate the solution approach described above:

processorTime = [3, 6] and the individual tasks take times tasks = [5, 2, 3, 7, 8, 1, 4, 6].

Start with the first processor (processorTime[0] = 3) and assign the four longest tasks [8, 7, 6, 5].

Move to the second processor (processorTime[1] = 6) and assign the remaining four tasks [4, 3, 2, 1].

For the second processor, starting at time 6, the maximum task time is 4, so it will finish at time 6 + 4 = 10.

ans = max(ans, t + tasks[i]) # ans becomes max(0, 3+5), then max(11, 6+1)

i -= 4 # Move the index to the next set of 4 tasks

minimizes empty processor time and maximizes task processing overlaps.

o Sort tasks in descending order: It becomes [8, 7, 6, 5, 4, 3, 2, 1].

ans = 0 # Initialize the answer variable

1. Sorting:
Sort processorTime in ascending order: It's already [3, 6].

∘ For the first processor, which starts at time 3, the maximum task time is 8, so it will finish its longest task at time 3 + 8 = 11.

Assume we have n = 2 processors, and hence n * 4 = 8 tasks. The processors become available at times given by

4. Determine Final Answer: The largest among the completion times for the processors is 11, which corresponds to the first processor.

Example Walkthrough

Following the solution steps:

Greedy Assignment:

Calculating Completion Time:

minimize the total completion time.

processor times.sort()

if task index >= 0:

task_index -= 4

// Iterate over each processor time

if (taskIndex < 0) {</pre>

break;

taskIndex -= 4;

return minTime;

C++

TypeScript

/**

*/

for (int processorTime : processorTimes) {

// If there are no more tasks to allocate, break the loop

// Return the minimum time needed to complete all tasks

// and update minTime if this is larger than the current minTime

minTime = Math.max(minTime, processorTime + tasks.get(taskIndex));

* Calculates the minimum time required to process all tasks by assigning them to processors.

// Calculate the potential time to process the current task with the current processor

answer = Math.max(answer, processorTime + (taskIndex >= 0 ? tasks[taskIndex] : 0));

// and update the answer with the maximum value of the current answer and this potential time

// Decrement the taskIndex by 4 for the next iteration, to simulate assignment of tasks to every 4th processor

* @returns {number} The minimum time required to process all tasks.

// Initialize the answer and the index for the tasks array.

// Sort the processor times in ascending order

processorTimes.sort((a, b) => a - b);

let taskIndex: number = tasks.length - 1;

for (const processorTime of processorTimes) {

// Sort the tasks in ascending order

// Loop through each processor time

tasks.sort((a, b) => a - b);

let answer: number = 0:

taskIndex -= 4;

function minProcessingTime(processorTimes: number[], tasks: number[]): number {

tasks.sort()

min time = 0

Solution Implementation

class Solution:

Therefore, with this approach, all tasks will be finished in 11 units of time.

The solution code for this example would execute as follows:

tasks.sort(reverse=True) # tasks becomes [8, 7, 6, 5, 4, 3, 2, 1]

i = len(tasks) - 1 # i starts at 7, index of the smallest task

i -= 4 # Decrease i to select the next set of 4 tasks

return ans # Return 11 as the final calculated answer

for t in processorTime: # Iterate through processor times [3, 6]

class Solution:
 def minProcessingTime(self, processorTime: List[int], tasks: List[int]) -> int:
 processorTime.sort() # processorTime becomes [3, 6]

After running through this example, we have a clear view of how the algorithm assigns tasks to processors in a way that aims to

Importing List from the typing module for type annotations
from typing import List

Sort the processor times in ascending order

Sort the tasks in ascending order

def minProcessingTime(self, processor times: List[int], tasks: List[int]) -> int:

ans = 0 # Initialize the answer variable

Start from the last task, which has the longest processing time
task_index = len(tasks) - 1

Iterate over each processor to assign the longest task(s) they can process
for processor time in processor times:

Ensure the task index is not less than 0 to avoid IndexError

The completion time for the processor is its processing time

min time = max(min time, processor time + tasks[task index])

We select the max of the current min time or the new completion time

This represents the earliest time at which all processors have finished processing

int taskIndex = tasks.size() - 1; // Start from the last task (which is the largest due to sorting)

// Calculate the total time for current processor by adding its ready time to the task time

// Move to the task which is 4 positions earlier in the list since there are 4 processors (0-based index)

Decrement the task index by 4 because the current processor is considered to

process a task every 4 cycles/time units (as per the given decrement step)

Initialize the answer for the minimum processing time

added to the task time it will process

Return the minimum time required to process all tasks

```
return min_time
Java
import iava.util.Collections;
import java.util.List;
class Solution {
    /**
     * Calculates the minimum amount of time required to process all tasks given an array of processor times.
     * @param processorTimes A list of integers representing the times each processor requires to be ready for a task.
     * @param tasks A list of integers representing the times required to process each task.
     * @return The minimum processing time to complete all tasks.
     */
    public int minProcessingTime(List<Integer> processorTimes, List<Integer> tasks) {
        // Sort the processor times in ascending order
        Collections.sort(processorTimes);
        // Sort the tasks in ascending order
        Collections.sort(tasks);
        int minTime = 0; // Variable to store the minimum processing time required
```

#include <vector> #include <algorithm> // Include the algorithm header for using the sort function class Solution { public: int minProcessingTime(vector<int>& processorTimes, vector<int>& tasks) { // Sort the processor times in ascending order sort(processorTimes.begin(), processorTimes.end()); // Sort the tasks in ascending order sort(tasks.begin(), tasks.end()); // Initialize the answer to 0. This will track the minimum processing time. int minimumProcessingTime = 0; // Start from the last task and work backwards int taskIndex = tasks.size() - 1; // Iterate over processors and assign them the heaviest remaining task for (int processorTime : processorTimes) { if (taskIndex >= 0) { // Check if there are still tasks to process // Update the minimum processing time if it's less than the current processor's time plus task time minimumProcessingTime = max(minimumProcessingTime, processorTime + tasks[taskIndex]); // Move to the next set of tasks, assuming each processor can process 4 tasks simultaneously taskIndex -= 4; } else { // No more tasks to assign, break out of the loop. break; return minimumProcessingTime; // Return the calculated minimum processing time **}**;

* @param {number[]} processorTimes - Array of processor times, where each element represents the time a processor takes to complete

* @param {number[1} tasks - Arrav of task times, where each element represents the time a task requires for processing.

```
// Return the minimum time after all processors have been accounted for
    return answer;
# Importing List from the typing module for type annotations
from typing import List
class Solution:
    def minProcessingTime(self. processor times: List[int], tasks: List[int]) -> int:
        # Sort the processor times in ascending order
        processor times.sort()
        # Sort the tasks in ascending order
        tasks.sort()
        # Initialize the answer for the minimum processing time
        min time = 0
        # Start from the last task, which has the longest processing time
        task_index = len(tasks) - 1
        # Iterate over each processor to assign the longest task(s) they can process
        for processor time in processor times:
           # Ensure the task index is not less than 0 to avoid IndexError
            if task index >= 0:
                # The completion time for the processor is its processing time
                # added to the task time it will process
                # We select the max of the current min time or the new completion time
                # This represents the earliest time at which all processors have finished processing
                min time = max(min time, processor time + tasks[task index])
                # Decrement the task index by 4 because the current processor is considered to
                # process a task every 4 cycles/time units (as per the given decrement step)
                task_index -= 4
        # Return the minimum time required to process all tasks
        return min_time
Time and Space Complexity
```

are $O(n \log n)$ where n is the number of tasks. Each list is sorted exactly once, and therefore the time complexity remains $O(n \log n)$. After the sorting, there is a for loop that iterates through processorTime. The loop itself runs in O(m) time, where m is the number.

After the sorting, there is a for loop that iterates through processorTime. The loop itself runs in O(m) time, where m is the number of processors. However, this does not affect the overall time complexity since it is assumed that m is much less than n and because the m loop is not nested within an n loop. Thus, the for loop's complexity does not exceed $O(n \log n)$ of the sorting step.

The time complexity of the code is primarily determined by the sorting operations performed on processorTime and tasks which

The space complexity of the sort operation depends on the implementation of the sorting algorithm. Typically, the sort method in Python (Timsort) has a space complexity of O(n). However, in the reference answer, they've noted a space complexity of $O(\log n)$. This can be considered correct under the assumption that the sorting algorithm used is an in-place sort like heapsort or in-place mergesort which has an $O(\log n)$ space complexity due to the recursion stack during the sort, but Python's default sorting algorithm is not in-place and actually takes O(n) space. If the sizes of the processorTime and tasks lists are immutable, and cannot be changed in place, the space complexity could indeed be $O(\log n)$ due to the space used by the sorting algorithm's recursion stack.