502. IPO

Hard Greedy Array Sorting Heap (Priority Queue) Leetcode Link

Problem Description

LeetCode is preparing for its initial Public Offering (IPO) and wants to take on some projects to boost its overall capital before going public. However, there are constraints on how many projects it can undertake due to limited resources. Specifically, LeetCode can only complete at most k distinct projects. The task at hand is to choose a subset of these projects to maximize the total capital, given that each project has its own potential profit and a minimum required capital to start. Initially, the available capital is w. Once a project is completed, its profit increases the total capital. The goal is to find out the maximum capital that can be achieved after completing at most k projects.

The input includes several pieces of information: the integer k representing the maximum number of projects LeetCode can complete, the initial capital w, a list of integers `profits` which denotes the profit from each project, and a list `capital` showing the minimum capital required to start each project. The problem guarantees that the final answer will be small enough to fit into a 32-bit signed integer.

Intuition

The solution to this problem involves sorting projects by their capital requirements and using a greedy approach to pick projects that can be started with the current available capital and yield the highest profit.

Firstly, we need to categorize projects based on whether they are within our current capital capabilities. We use two heaps (priority queues) to do this: `h1` for projects we cannot afford yet, and `h2` for profitable projects we can do right away. `h1` is min-heaped on capital required, so projects requiring less capital are at the top. Heap `h2` is max-heaped on profits (we use negative values to achieve this in Python's min-heap), so the most profitable projects we can afford are at the top.

The algorithm proceeds as follows:

1. Pair each project's capital and profit and push them into `h1`.
2. Heapify `h1` to structure it according to the minimum capital required.
3. For up to k projects, do the following:
   - Move projects from `h1` to `h2` if they can be afforded with the current capital w.
   - If `h2` has projects, pop the project with the maximum profit, add that profit to w.
   - If `h2` is empty, it means there are no more projects that can be done with the current capital, and we break the loop.
   - Decrease k by 1 because a project has been completed.

By following this approach, we ensure that with each iteration, we are choosing the most profitable project that can be started with the available capital. Once k reaches zero or there are no projects left that can be afforded, the algorithm ends. The value of w at this point is the maximized capital we aimed to find.

Solution Approach

The given Python solution makes use of a min-heap and a max-heap to efficiently determine which project to pick next. The overall approach can be broken down into several algorithmic steps using two primary data structures — heaps (implemented via Python's priority queue).

Here's the step-by-step implementation:

1. Zip the `profits` and `capital` lists together so that each tuple `(c, p)` in the new list `h1` consists of the capital required and the profit of a project. This allows us to keep each project's financials together for easy access later on.
2. Transform the list `h1` into a min-heap in-place using Python's `heapify` function. This operation reorders the elements so that the project with the smallest capital requirement is at the root of the heap, which allows us to quickly find and extract the project with the lowest capital threshold.
3. Initialize an empty list `h2` to use as a max-heap for profits, which we'll need later to keep track of the most profitable projects that we can afford within our current capital.
4. Begin a loop that will run at most k times, where k is the maximum number of projects that can be completed.
5. Within the loop, move projects from the min-heap `h1` to the max-heap `h2` as long as the project's capital requirement is smaller than or equal to the current capital w. We use the `heappop` function to extract the project with the smallest capital requirement from `h1` and the `heappush` function to push its profit (as a negative number to facilitate max-heap behavior in Python's min-heap) into `h2`.
6. If `h2` is not empty, it means we have at least one project that can be started given the current capital. Use `heappop` again, this time on `h2`, to extract the project with the largest profit (remembering to negate the value to transform it back from the negative value we used earlier).
7. Update the current capital w by adding the profit of the chosen project to it.
8. If `h2` is empty and we can no longer afford any projects with the current capital, exit the loop as we can't do any more projects.
9. After at most k iterations or when no further projects can be done, exit the loop and return the value of w, which now represents the maximized capital.

The use of heaps allows the solution to be efficient even when dealing with a large number of projects, as projects that can't be afforded yet are quickly filtered out, and finding the most profitable of the remaining projects can be done in constant time ($O(1)$) due to the heap's properties. The heap insertions and deletions (`heappush` and `heappop`) operate in logarithmic time ($O(\log n)$), ensuring that each project insertion or extraction does not slow down the algorithm significantly even for large numbers of projects.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Consider the following scenario where LeetCode can complete at most $k = 2$ projects, the initial capital is $w = 0$, the list of profits is `profits = [1, 2, 3]`, and the list of minimum required capital for each project is `capital = [0, 1, 1]`.

We will apply the described algorithm step by step:

1. Pair each project's capital and profit: `h1 = [(0, 1), (1, 2), (1, 3)]`.
2. Heapify `h1` to structure it according to the minimum capital required. After heapifying, `h1` still looks the same since the elements were already in the correct order: `h1 = [(0, 1), (1, 2), (1, 3)]`.
3. Initialize an empty list `h2` to use as a max-heap for available profitable projects.

Now we begin our main loop which runs at most 2 times:

- For the first iteration, $w = 0$. We move projects from `h1` to `h2` as long as the project's capital requirement is smaller than or equal to w. Since the first project requires 0 capital, which is equal to our current capital, we add it to `h2` as `(-1, 0)` (we use negative profit for max-heap behavior). Now `h1 = [(1, 2), (1, 3)]` and `h2 = [(-1, 0)]`.
- Pop the most profitable project from `h2`, which gives us `(-1, 0)`. This means we earn a profit of 1 (negating the popped value). Our capital is now $w = 1$.
- We have completed one project. Decrease k by 1, now $k = 1$.
- For the second iteration, $w = 1$. We transfer all projects which fit into our current capital w from `h1` to `h2`. Now we can afford the remaining projects as they both require a capital of 1. Add them to `h2`: `h2 = [-2, 1), (-3, 1)` (again using negative profits). `h1` is now empty and `h2` is `[-3, -2]` after re-heapifying for max-profit.
- Pop the most profitable project from `h2`, which is `(-3, 1)`. We earn a profit of 3, so now our capital is $w = 4$.
- We have completed the second project. Decrease k by 1, now $k = 0$.
- The loop ends as k is now 0.

With no more projects that can be completed due to the limit k, we have finished our iterations. The maximized capital is now $w = 4$, which is the value we return.

By processing the projects in this manner, LeetCode chose the most profitable projects available within their capital limits, thus maximizing their capital by the end of the process.

Python Solution

```python
from heapq import heappop, heappush, heapify

class Solution:
    def findMaximizedCapital(self, k: int, startingCapital: int, profits: List[int], capital: List[int]) -> int:
        # Create a list of tuples, where each tuple consists of (project capital, project profit)
        # and then transform this list into a min-heap based on required capital to start projects.
        min_heap_by_capital = [(c, p) for c, p in zip(capital, profits)]
        heapify(min_heap_by_capital)

        # Create a max heap for available projects to track the profits that we can achieve.
        max_heap_by_profit = []

        # Loop until we have completed k projects or there are no profitable projects available.
        while k:
            # Move projects we can afford from the min heap to the max heap.
            while min_heap_by_capital and min_heap_by_capital[0][0] <= startingCapital:
                # We push the negative profit to simulate the max heap using the min heap property.
                heappush(max_heap_by_profit, -heappop(min_heap_by_capital)[1])

            # If there are no projects available in the max heap, we break out of the loop.
            if not max_heap_by_profit:
                break

            # Pop the project with the highest profit from the max heap and add it to our current capital.
            startingCapital -= heappop(max_heap_by_profit)

            # Decrement the counter for how many projects we have completed.
            k -= 1

        # Return the maximized capital after completing up to k projects.
        return startingCapital
```

Java Solution

```java
class Solution {
    public int findMaximizedCapital(int k, int w, int[] profits, int[] capital) {
        int n = capital.length; // Number of projects

        // Create a min-heap (priority queue) to keep track of projects based on required capital
        PriorityQueue<int[]> minCapitalHeap = new PriorityQueue<>((a, b) -> a[0] - b[0]);

        // Populate the min-heap with project information - each item is an array where
        // the first element is the capital required and the second element is the profit.
        for (int i = 0; i < n; ++i) {
            minCapitalHeap.offer(new int[] {capital[i], profits[i]});
        }

        // Create a max-heap (priority queue) to keep track of profitable projects that we can afford
        PriorityQueue<Integer> maxProfitHeap = new PriorityQueue<>(n, (a, b) -> b - a);

        // Iterate k times, which represents the maximum number of projects we can select
        while (k-- > 0) {
            // Move all the projects that we can afford (w >= required capital) to the max profit heap
            while (!minCapitalHeap.isEmpty() && minCapitalHeap.peek()[0] <= w) {
                maxProfitHeap.offer(minCapitalHeap.poll()[1]);
            }
            // If the max profit heap is empty, it means there are no projects we can afford, so we break
            if (maxProfitHeap.isEmpty()) {
                break;
            }
            // Otherwise, take the most profitable project from the max profit heap and add its profit to our total capital
            w += maxProfitHeap.poll();
        }
        return w; // Return the maximized capital after picking up to k projects
    }
}
```

C++ Solution

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

using std::pair;
using std::priority_queue;
using std::vector;

// A pair of integers alias to represent a pair of capital and profit.
using ProfitCapitalPair = pair<int, int>;

class Solution {
public:
    // This function maximizes the capital by doing at most 'k' projects starting
    // with initial capital 'w'. Projects have associated profits and capital requirements.
    int findMaximizedCapital(int k, int w, vector<int>& profits, vector<int>& capital) {
        // A min-heap that stores pairs of capital requirements and profits
        // Such that the project with the least capital requirement is on top.
        priority_queue<ProfitCapitalPair, vector<ProfitCapitalPair>, std::greater<ProfitCapitalPair>> minCapitalHeap;

        // Populate the min-heap with the capital and profits of the projects.
        int numProjects = profits.size();
        for (int i = 0; i < numProjects; ++i) {
            minCapitalHeap.push({capital[i], profits[i]});
        }

        // A max-heap to store profits of projects we can afford to invest in,
        // so we can always choose the most profitable one next.
        priority_queue<int> maxProfitHeap;

        // Loop to perform up to 'k' investments.
        while (k--) {
            // Move all projects we can afford (with current capital 'w') to a max-heap.
            // This way we will only be left with projects that we can't afford.
            while (!minCapitalHeap.empty() && minCapitalHeap.top().first <= w) {
                maxProfitHeap.push(minCapitalHeap.top().second);
                minCapitalHeap.pop();
            }

            // If we cannot afford any project, break out of the loop.
            if (maxProfitHeap.empty()) {
                break;
            }

            // Pick the most profitable project and increase our capital.
            w += maxProfitHeap.top();
            maxProfitHeap.pop();
        }

        // After completing 'k' or the maximum number of profitable investments,
        // 'w' is our maximized capital.
        return w;
    }
};
```

Typescript Solution

```typescript
type ProfitCapitalPair = { capital: number, profit: number };

// A function that maximizes the capital by doing at most 'k' projects
// starting with initial capital 'W'. Projects have associated profits
// and capital requirements.
function findMaximizedCapital(k: number, W: number, Profits: number[], Capital: number[]): number {
    // Initialize a min-heap based on capital requirements so the project
    // with the least capital requirement is at the beginning.
    let minCapitalHeap: ProfitCapitalPair[] = [];

    // Populate the min-heap with capital and profits of the projects.
    for (let i = 0; i < Profits.length; ++i) {
        minCapitalHeap.push({ capital: Capital[i], profit: Profits[i] });
    }
    // Sort the heap to ensure the smallest capital requirement is at the start.
    minCapitalHeap.sort((a, b) => a.capital - b.capital);

    // Initialize an array to use as a max-heap to store profits.
    // If the projects we can afford, we'll add to choose the most profitable one.
    let maxProfitHeap: number[] = [];

    // Define a helper function to turn the array into a max-heap,
    // using simple push-pop since TypeScript/JavaScript does not have a native priority queue structure.
    const pushMaxHeap = (value: number) => {
        maxProfitHeap.push(value);
        maxProfitHeap.sort((a, b) => b - a); // Sort in descending order to keep max element at the start
    };

    const popMaxHeap = (): number => {
        return maxProfitHeap.shift() || 0; // Remove and return the first element (max element)
    };

    // Loop to perform up to 'k' investments.
    for (let i = 0; i < k; ++i) {
        // Move all projects we can afford (with current capital 'W') to the max-heap.
        // This way we will only be left with projects that we can't afford.
        while (minCapitalHeap.length && minCapitalHeap[0].capital <= W) {
            const project = minCapitalHeap.shift()!;
            pushMaxHeap(project.profit);
        }

        // If we cannot afford any project, exit the loop.
        if (!maxProfitHeap.length) {
            break;
        }

        // Pick the most profitable project we can afford and increase our capital.
        W += popMaxHeap();
    }

    // After completing 'k' or the maximum number of profitable investments,
    // 'W' is our maximized capital.
    return W;
}
```

Time and Space Complexity

Time Complexity

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

1. Preparing the heap `h1` involves pairing each element from `capital` with `profits`. This takes $O(N)$ time where N is the number of projects.
2. The `heapify` function on `h1` runs in $O(N)$ time.
3. In the worst case, the loop runs k times. In each iteration, the inner while loop can push up to N elements to `h2` (over all iterations), this gives us $O(N\log(N))$ for all `heappush` operations since each `heappush` operation takes $O(\log(N))$.
4. Each `heappop` operation on `h1` takes $O(\log(N))$ time, resulting in $O(N\log(N))$ for all `heappop` operations from `h1`.
5. Each `heappop` from `h2` takes $O(\log(K))$. Since there are at most k such operations, the total time taken by all `heappop` operations from `h2` is $O(k\log(K))$.

Summing up all the operations, the resulting time complexity would be the sum of the heap operations across both heaps, which in big-O notation simplifies to the largest term. This would be $O(N\log(N) + k\log(K))$.

Space Complexity

The space complexity can be analyzed as follows:

1. Two extra heaps `h1` and `h2` are used, of which `h1` can contain up to N elements and `h2` can contain up to k elements.
2. There are no other significant uses of space beyond the input storage.

This gives us a space complexity of $O(N + k)$. Since heaps use space proportionate to the number of elements they contain.