Solution: Maximize Capital

Let's solve the Maximize Capital problem using the Two Heaps pattern.

We'll cover the following

Statement

A busy investor with an initial capital, `c`, needs an automated investment program. They can select `k` distinct projects from a list of `n` projects with corresponding `capitals` requirements and expected `profits`. For a given project i, its capital requirement is $capitals[i]$, and the profit it yields is $profits[i]$.

The goal is to maximize their cumulative capital by selecting a maximum of `k` distinct projects to invest in, subject to the constraint that the investor's current capital must be greater than or equal to the capital requirement of all selected projects.

When a selected project from the identified ones is finished, the pure profit from the project, along with the starting capital of that project is returned to the investor. This amount will be added to the total capital held by the investor. Now, the investor can invest in more projects with the new total capital. It is important to note that each project can only be invested once.

As a basic risk-mitigation measure, the investor wants to limit the number of projects they invest in. For example, if `k` is 2, the program should identify the two projects that maximize the investor's profits while ensuring that the investor's capital is sufficient to invest in the projects.

Overall, the program should help the investor to make informed investment decisions by picking a list of a maximum of `k` distinct projects to maximize the final profit while mitigating the risk.

Constraints:

- $1 \leq$ `k` $\leq 10^5$
- $0 \leq$ `c` $\leq 10^9$
- $1 \leq$ `n` $\leq 10^5$
- `k` $\leq$ `n`
- `n` $==$ `profits.length`
- `n` $==$ `capitals.length`
- $0 \leq$ `profits[i]` $\leq 10^4$
- $0 \leq$ `capitals[i]` $\leq 10^9$

Solution

You may have already brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and implementation constraints.

Naive approach

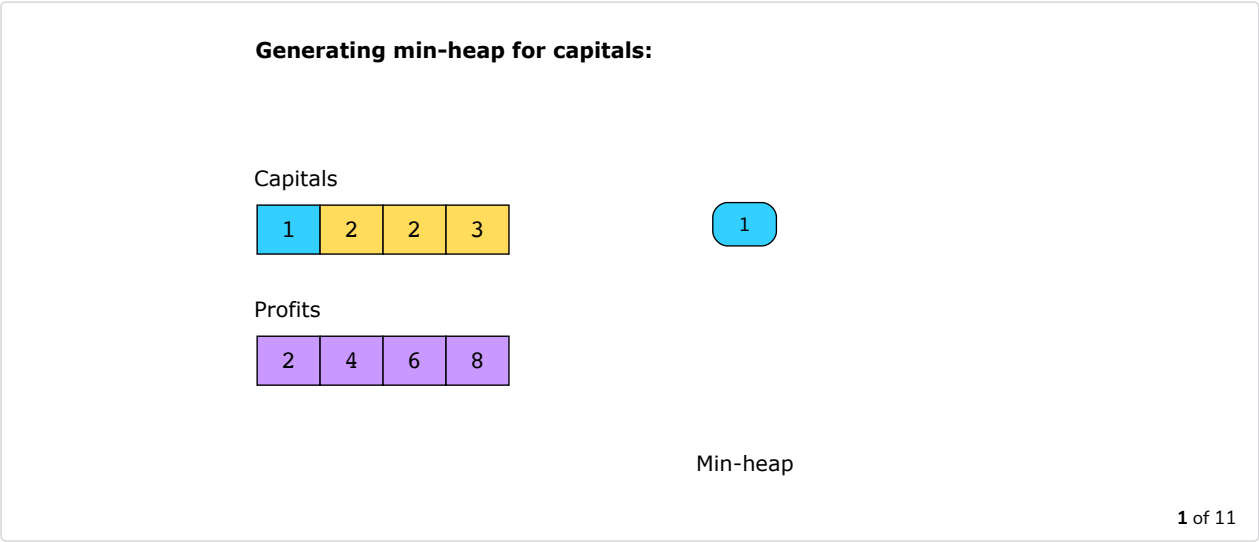
The naive approach is to traverse every value of the `capitals` array based on the available capital. If the current capital is less than or equal to the capital value in the array, then store the profit value in a new array that corresponds to the capital index. Whenever the current capital becomes less than the capital value in the array, we'll select the project with the largest profit value. The selected profit value will be added to the previous capital. Repeat this process until we get the required number of projects containing the maximum profit.

We got the solution we needed, but at what cost? The time complexity is $O(n^2)$, where n represents the number of projects. This is because we need $O(n)$ time to traverse the `capitals` array in order to find affordable projects. Additonally, for each subset of affordable projects, we would need another $O(n)$ search to narrow down the project list to the one that yields the highest profit. The space complexity for storing the profit values in a new array is $O(n)$.

Optimized approach using two heaps

This problem can be solved efficiently by using two heaps. We can minimize the time required to find affordable projects by maintaining the required `capitals` of the projects in a min-heap. Similarly, to quickly find the project with the highest profit, we can maintain `profits` of the projects in a max-heap.

The slide deck below illustrates the key steps of the solution, with `k` $= 2$ and `c` $= 1$:



> **Note:** In the following section, we will gradually build the solution. Alternatively, you can skip straight to the Just the code section.

Step-by-step solution construction

Here are the steps required to solve this problem:

We'll push all the `capitals` values in a min-heap to pick the required number of projects, starting from the one with the lowest capital requirement.

Java

```java
class MaximizeCapital {

    public static int maximumCapital(int c, int k, int[] capitals, int[] profits) {
        PriorityQueue<Integer> CapitalminHeap = new PriorityQueue<Integer>();

        int i = 0;
        // Insert all capitals values to a min-heap
        while (i<capitals.length) {
            CapitalminHeap.add(capitals[i]);

            i++;
        }
        printCapitalsMinHeap(CapitalminHeap);
        return 0;
    }
    static void printCapitalsMinHeap(PriorityQueue<Integer> capitals)
    {
        List<Integer> l = new ArrayList<>();
        while(!capitals.isEmpty()){
            l.add(capitals.peek());
            capitals.poll();
        }
        System.out.println("\t"+ l.toString());
    }

    public static void main(String[] args) {
        int[] c = { 1, 2, 1, 7, 2 };
        int[] k = { 2, 3, 3, 2, 4 };
```

Maximize Capital

Let's identify all the projects whose capital requirements are less than or equal to our current capital and push the profits yielded by these projects onto the max-heap.

Assume the current capital is fixed as 2.

Java

```java
class MaximizeCapital {

    public static int maximumCapital(int c, int k, int[] capitals, int[] profits) {
        int currentCapital = 2;
        PriorityQueue<Integer> ProfitsmaxHeap = new PriorityQueue<Integer>(Collections.reverseOrder());

        System.out.println("\tIdentifying projects we can afford with our current capital of " + currentC

        int i = 0;
        // Select projects (in the range of the current capital)
        // then push them onto the max-heap
        while (i<profits.length) {

            if(capitals[i] <= currentCapital){
                ProfitsmaxHeap.add(profits[i]);
                System.out.println("\t\tProfit of project (with capital requirement of " +
                currentCapital + " pushed on to max-heap = " + profits[i]);
            }

            i++;
        }

        return 0;
    }
```

```
    public static void main(String[] args) {
        int[] c = {1, 2, 1, 7, 2};
```

Maximize Capital

Now, we can select the most affordable project that yields the highest profit. We'll add the profit from every selected project to the previous capital to calculate the maximum capital we can accumulate. We need to do this after each project selection as more capital will typically allow us to choose from a larger subset of projects. In case no profit is pushed onto the max-heap, we break out of the loop.

Here's the complete solution to calculate the maximum capital:

Java

```
class MaximizeCapital {

    public static int maximumCapital(int c, int k, int[] capitals, int[] profits) {
        int n = capitals.length;
        int currentCapital = c;
        // Insert all capitals values to a min-heap
        PriorityQueue<int[]> CapitalminHeap = new PriorityQueue<>((a, b) -> a[0] - b[0]);
        for (int i = 0; i < n; ++i) {
            CapitalminHeap.offer(new int[] {capitals[i], i});
        }
        PriorityQueue<int[]> ProfitsmaxHeap = new PriorityQueue<>((a, b) -> b[0] - a[0]);
        int i = 0;
        // Calculate capital of all the required number of projects
        // containing maximum profit
        while (i < k) {
            // Select projects (in the range of the current capital)
            // then push them onto the max-heap
            while (!CapitalminHeap.isEmpty() && CapitalminHeap.peek()[0] <= currentCapital) {
                int[] j = CapitalminHeap.poll();
                ProfitsmaxHeap.offer(new int[]{profits[j[1]], j[1]});
            }
            // check if the max-heap is empty
            if (ProfitsmaxHeap.isEmpty()) {
                break;
            }
            // Select those projects from the max-heap that contain the maximum profit
            int x = ProfitsmaxHeap.poll()[0];
            System.out.println("\t\tUpdated capital = "+ currentCapital + " + "+ x);
```

Maximize Capital

Just the code

Here's the complete solution to this problem:

Java

```
class MaximizeCapital {

    public static int maximumCapital(int c, int k, int[] capitals, int[] profits) {
        int n = capitals.length;
        int currentCapital = c;
        PriorityQueue<int[]> CapitalminHeap = new PriorityQueue<>((a, b) -> a[0] - b[0]);
        for (int i = 0; i < n; ++i) {
            CapitalminHeap.offer(new int[] {capitals[i], i});
        }
        PriorityQueue<int[]> ProfitsmaxHeap = new PriorityQueue<>((a, b) -> b[0] - a[0]);
        int i = 0;
        while (i < k) {
            while (!CapitalminHeap.isEmpty() && CapitalminHeap.peek()[0] <= currentCapital) {
                int[] j = CapitalminHeap.poll();
                ProfitsmaxHeap.offer(new int[]{profits[j[1]], j[1]});
```

```
            }
            if (ProfitsmaxHeap.isEmpty()) {
                break;
            }
            int x = ProfitsmaxHeap.poll()[0];
            currentCapital += x;
            i++;
        }
        return currentCapital;
    }

    public static void main(String[] args) {
```

Maximize Capital

Solution summary

To recap, the solution to this problem can be divided into the following four steps:

1. Push all `capitals` values in a min-heap.
2. Select projects that can be invested in the range of the current capital and push their profits in a max-
3. Select the project from the max-heap that yields the maximum profit.
4. Add the profit from the selected project to the current capital.

Repeat steps 2–4 until `k` projects have been selected.

Time complexity

The time complexity to push the required `capitals` values in the min-heap is $O(n \log n)$, where n represents the number of projects. The time complexity to select the projects with the maximum profits from the heaps is $O(k \log n)$, where k represents the number of selected projects. Therefore, the total time complexity comes out to be $O(n \log n + k \log n)$, that is, $O((n + k) \log n)$. Since the upper bound on k is n, the worst case is when $k = n$. We can express the worst-case time complexity as $O((n + n) \log n)$, which is $O(n \log n)$.

Space complexity

We're using two heaps, one to store `capitals` and one to store `profits`. In the worst case, where we meet the capital requirements of all the projects right from the start, we populate both heaps with n elements each. Hence, the space complexity of this solution is $O(n)$.