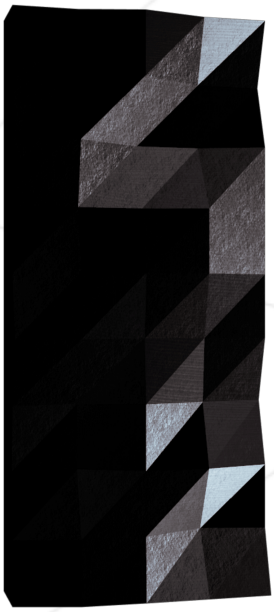

Starfleet

Interview

Staff 42 pedago@42.fr

Summary: This document is an interview question for the Starfleet Piscine.

Contents

Chapter I

General rules

- The interview should last between `45 minutes`.
- Both the interviewer and the interviewed student must be present.
- The interviewed student should write his code using a `whiteboard`, with the language of her/his choice.
- At the end of the interview, the interviewer evaluates the student based on the provided criteria.

Read carefully the interview question and solutions, and make sure you `understand` them before the interview. You can't share this document with other students, as they might be interviewed on the same question. Giving them the answer would prevent them from having to solve an unknown question during an interview.

I.1 During the interview

During the interview, we ask you to :

- Make sure the interviewed student `understands` the question.
- Give her/him any `clarification` on the subject that she/he might need.
- Let her/him come up with a solution before you guide her/him to the best solution given the constraints (time and space).
- Ask the student what is the `complexity` of her/his algorithm ? Can it be improved and how ?
- `Guide` her/him to the best solution without giving the answer. You may refer to the `hints` for that.
- You want to evaluate how the interviewed student thinks, so ask her/him to `explain everything` that she/he thinks or writes (there should be no silences).
- If you see a mistake in the code, wait untill the end and give her/him a chance to correct it by her/himself.
- Ask the student to show how the algorithm works on an `example`.
- Ask the student to explain how `limit cases` are handled.
- Bring out to the student any mistake she/he might have done.
- Give `feedback` on her/his performances after the interview.
- Be `fair` in your evaluation.

As always, stay mannerly, polite, respectful and constructive during the interview. If the interview is carried out smoothly, you will both benefit from it !

Chapter II

Smallest range

II.1 Interview question

`Given k arrays of sorted integers, find the smallest range that includes at least one number from each of the k arrays.`

Example :

```
Input :
    Array1 : {4, 10, 15, 24, 26}
    Array2 : {0, 9, 12, 20}
    Array3 : {5, 18, 22, 30}

Output : the smallest range here is [20, 24]
as it contains 24 from array 1, 20 from array 2, and 22 from array 3
```

II.2 Acceptable answers (no constraint)

II.2.1 Brute force

The brute force solution here is to find the min and max for all possible arrays containing one element of each of the k arrays, and return the smallest range.

There are (n0 * n1 * ... * nk-1) possible arrays, where ni is the number of elements in the array i.

Each possible array has k elements, so it takes O(k) time to find the min and max.

```
O(n0 * n1 * n2 * ... * nk-1 * k) time , O(k) space
where ni is the number of elements in the array i
```

`Note` : This solution does not use the fact that the k arrays are sorted. We can do better !

code:

```
struct s_range {  // container for min and max
        int min;
        int max;
};

struct s_array {  // container for the k arrays
        int *arr;     // array
        int n;        // size
};
```

```
struct s_range getRange(struct s_array **arrays, int k, int *arr) {
        struct s_range range;

        range.min = INT_MAX;
        range.max = INT_MIN;
        for (int i = 0; i < k; i++) {
                if (arr[i] < range.min)
                        range.min = arr[i];
                if (arr[i] > range.max)
                        range.max = arr[i];
        }
        return (range);
}

void smallest(struct s_array **arrays, int k, int i, int *arr, struct s_range *small) {
        struct s_range current;

        if (i == k) {
                current = getRange(arrays, k, arr);
                if (small->max == INT_MIN
                        || current.max - current.min < small->max - small->min) {
                        small->min = current.min;
                        small->max = current.max;
                }
                return ;
        }
        for (int j = 0; j < arrays[i]->n; j++) {
                arr[i] = arrays[i]->arr[j];
                smallest(arrays, k, i + 1, arr, small);
        }
}

struct s_range smallestRange(struct s_array **arrays, int k) {
        int arr[k];
        struct s_range small;

        small.min = INT_MAX;
        small.max = INT_MIN;
        smallest(arrays, k, 0, arr, &small);
        return (small);
}
```

II.2.2 With k indexes

We start with k indexes intialized to zero (one index for each of the k arrays), and find the minimum value for this set of indexes.

We know that if this minimum element is included in the smallest range that we are looking for, then it has to be with the minimum of all other arrays. Which is already the case in our set of indexes since the k arrays are sorted.

We can then move one to the next element in the corresponding array and so on.

The algorithm is described below.

Repeat those steps :

- Find the minimum element for the given set of indexes.
- Move on to the next element in the corresponding array (increment the index). Break the loop if we have reached the end of the array.
- Find the new `min` and `max` for the new set of indexes.
- Update the smallest range if `(max - min) < smallest range`.

```
O(n * k) time , O(k) space
where n is the total number of elements in the k arrays
```

code:

```c
struct s_range getRange(struct s_array **arrays, int k, int *indexes) {
	struct s_range range;
	int value;

	range.min = INT_MAX;
	range.max = INT_MIN;
	for (int i = 0; i < k; i++) {
		value = arrays[i]->arr[indexes[i]];
		if (value < range.min)
			range.min = value;
		if (value > range.max)
			range.max = value;
	}
	return (range);
}

int getSmallest(struct s_array **arrays, int k, int *indexes) {
	int idx = -1;
	int small = INT_MAX;
	int value;

	for (int i = 0; i < k; i++) {
		value = arrays[i]->arr[indexes[i]];
		if (value < small) {
			small = value;
			idx = i;
		}
	}
	if (indexes[idx] == arrays[idx]->n - 1)
		return (-1);
	return (idx);
}

struct s_range smallestRange(struct s_array **arrays, int k) {
	int indexes[k];
	struct s_range current;
	struct s_range small;
	int idx;

	// initialize all indexes to 0
	memset(indexes, 0, sizeof(int) * k);

	// initialize the smallest range
	small = getRange(arrays, k, indexes);

	while ((idx = getSmallest(arrays, k, indexes)) != -1) {

		// move on to the next element in the array which contains the minimum element
		indexes[idx]++;

		// get the new range
		current = getRange(arrays, k, indexes);

		// update the smallest range if necessary
		if (current.max - current.min < small.max - small.min) {
			small.min = current.min;
			small.max = current.max;
		}
	}
	return (small);
}
```

`Note 1` : Here `getSmallest()` and `getSmallest()` are two seperate function with a runtime of O(k). It could be done in only one function, but the runtime would still be O(n).

`Note 2` : Keeping track of the maximum here could be improved since the k arrays are sorted. Each time that we increment an array, we could check if the maximum needs to be updated. The real issue is keeping track of the minimum.

II.3 Follow up question

Your algorithm must run in `O(n * logk) time`.

II.3.1 Hints

- What is the data structure that will allow you to keep track of the minimum with an optimized runtime ?
- Have you tried with a `min heap` ?

II.4 Best solutions

II.4.1 With a Min Heap

This solution is the same as the one above, except that this time we will use a `min heap` to keep track of the minimum. In fact, getting the range will then be done in O(logk) time instead of O(k).

The algorithm is described below.

Build a `min heap` with the first element of each the k arrays (including these information: index of array, index of element in array, value of element), then repeats those steps :

- The minimum element of the heap is at index 0.
- Move on to the next element in the corresponding array (update the `min heap`, which takes O(logk) time). Break the loop if we have reached the end of the array.
- Update the smallest range if necessary.

```
O(n * logk) time , O(k) space
where n is the total number of elements in the k arrays
```

`Note` : The final product can be obtained directly in the second step (as shown by the code below).

code for the `min heap` : (these functions do not need to be coded during the interview)

```c
struct s_item {
        int i;                  // index of array between 0 and k-1
        int idx;                // index of current element in the array
        int value;              // value of the current element
};

struct s_heap {             // container for the heap
        struct s_item **items;  // actual heap
        int min;                // min of the heap
        int max;                // max of the heap
};

void swapStruct(struct s_heap *heap, int i, int j) {
        struct s_item *tmp;

        tmp = heap->items[i];
        heap->items[i] = heap->items[j];
        heap->items[j] = tmp;
}

void heapifyMin(struct s_heap *heap, int k, int i) {
        int small;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < k && heap->items[left]->value < heap->items[i]->value) {
                small = left;
        } else {
                small = i;
        }
        if (right < k && heap->items[right]->value < heap->items[small]->value) {
                small = right;
        }
        if (small != i) {
                swapStruct(heap, i, small);
                heapifyMin(heap, k, small);
        }
}

void buildMinHeap(struct s_heap *heap, int k) {
        int i = k / 2 - 1;
        while (i >= 0) {
                heapifyMin(heap, k, i);
                i--;
        }
        heap->min = heap->items[0]->value;
        heap->max = INT_MIN;
        for (int i = 0; i < k; i++) {
                if (heap->items[i]->value > heap->max)
                        heap->max = heap->items[i]->value;
        }
}
```

code:

```
void initHeap(struct s_heap *heap, struct s_array **arrays, int k) {
        heap = malloc(sizeof(struct s_heap));
        heap->items = malloc(sizeof(struct s_item *) * k);
        for (int i = 0; i < k; i++) {
                heap->items[i] = malloc(sizeof(struct s_item));
                heap->items[i]->i = i;
                heap->items[i]->idx = 0;
                heap->items[i]->value = arrays[i]->arr[0];
        }
}

int updateHeap(struct s_heap *heap, struct s_array **arrays) {
        if (heap->items[0]->idx == arrays[heap->items[0]->i]->n - 1)
                return (0);
        (heap->items[0]->idx)++;
        heap->items[0]->value = arrays[heap->items[0]->i]->arr[heap->items[0]->idx];

        // set maximum of heap
        if (heap->items[0]->value > heap->max)
                heap->max = heap->items[0]->value;

        return (1);
}

struct s_range smallestRange(struct s_array **arrays, int k) {
        struct s_heap *heap;    // min heap
        struct s_range small;  // smallest range

        // initialize min heap with first element of each of the k arrays
        initHeap();

        // rearrange element to fit min heap criteria
        buildMinHeap(heap, k);

        // set smallest range to current heap range
        small.min = heap->min;
        small.max = heap->max;

        while (updateHeap(heap, arrays)) { // Move on to the next element

                // rearrange element to fit min heap criteria : O(logk) time
                heapifyMin(heap, k, 0);

                // set minimum of heap
                heap->min = heap->items[0]->value;

                //update smallest range
                if (heap->max - heap->min < small.max - small.min) {
                        small.min = heap->min;
                        small.max = heap->max;
                }
        }
        return (small);
}
```