Minimize Max Distance to Gas Station

On a horizontal number line, we have gas stations at positions `stations[0], stations[1], ..., stations[N-1]`, where `N = stations.length`.

Now, we add `K` more gas stations so that **D**, the maximum distance between adjacent gas stations, is minimized.

Return the smallest possible value of **D**.

Example:

```
Input: stations = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], K = 9
Output: 0.500000
```

Note:

1. `stations.length` will be an integer in range `[10, 2000]`.
2. `stations[i]` will be an integer in range `[0, 10^8]`.
3. `K` will be an integer in range `[1, 10^6]`.
4. Answers within `10^-6` of the true value will be accepted as correct.

Solution 1

Why did I use s binary search?
In fact there are some similar problems on Leetcode so that is part of experience.
Secondly, I got a hint from "Answers within 10^-6 of the true value will be accepted as correct.". The first solution I tried was binary search.
Because binary search may not find exact value but it can approach the true answer.

Explanation of solution
Now we are using binary search to find the smallest possible value of D.
I initilze `left = 0` and `right = the distance between the first and the last station`
`count` is the number of gas station we need to make it possible.
`if count > K`, it means `mid` is too small to realize using only K more stations.
`if count <= K`, it means `mid` is possible and we can continue to find a bigger one.
When `left + 1e-6 >= right`, it means the answer within 10^-6 of the true value and it will be accepted.

Time complexity:
O(MlogN), where N is station length and M is st[N - 1] - st[0]

C++

```
double minmaxGasDist(vector<int>& st, int K) {
        int count, N = st.size();
        double left = 0, right = st[N - 1] - st[0], mid;
        while (left + 1e-6 < right) {
            mid = (left + right) / 2;
            count = 0;
            for (int i = 0; i < N - 1; ++i)
                count += ceil((st[i + 1] - st[i]) / mid) - 1;
            if (count > K) left = mid;
            else right = mid;
        }
        return right;
    }
```

Java:

```
public double minmaxGasDist(int[] st, int K) {
        int count, N = st.length;
        double left = 0, right = st[N - 1] - st[0], mid;

        while (left +1e-6 < right) {
            mid = (left + right) / 2;
            count = 0;
            for (int i = 0; i < N - 1; ++i)
                count += Math.ceil((st[i + 1] - st[i]) / mid) - 1;
            if (count > K) left = mid;
            else right = mid;
        }
        return right;
    }
```

Python:

```
def minmaxGasDist(self, st, K):
        left, right = 1e-6, st[-1] - st[0]
        while left + 1e-6 < right:
            mid = (left + right) / 2
            count = 0
            for a, b in zip(st, st[1:]):
                count += math.ceil((b - a) / mid) - 1
            if count > K:
                left = mid
            else:
                right = mid
        return right
```

written by lee215 original link here

Solution 2

Note this is an adaptation of my original post here with updated solution for this problem. Hopefully it can shed some light on these type of problems. First here is a list of LeetCode problems that can be solved using the trial and error algorithm (you're welcome to add more):

1. 774. Minimize Max Distance to Gas Station
2. 719. Find K-th Smallest Pair Distance
3. 668. Kth Smallest Number in Multiplication Table
4. 644. Maximum Average Subarray II

Well, normally we would refrain from using the naive trial and error algorithm for solving problems since it generally leads to bad time performance. However, there are situations where this naive algorithm may outperform other more sophisticated solutions, and LeetCode does have a few such problems (listed at the end of this post -- ironically they are all "hard" problems). So I figure it might be a good idea to bring it up and describe a general procedure for applying this algorithm.

The basic idea for the trial and error algorithm is actually very simple and summarized below:

`Step 1` : Construct a candidate solution.
`Step 2` : Verify if it meets our requirements.
`Step 3` : If it does, accept the solution; else discard it and repeat from `Step 1` .

However, to make this algorithm work efficiently, the following two conditions need to be true:

Condition 1: We have an efficient verification algorithm in `Step 2` ;
Condition 2: The search space formed by all candidate solutions is small or we have efficient ways to search this space if it is large.

The first condition ensures that each verification operation can be done quickly while the second condition limits the total number of such operations that need to be done. The two combined will guarantee that we have an efficient trial and error algorithm (which also means if any of them cannot be satisfied, you should probably not even consider this algorithm).

Now let's look at this problem: `Minimize Max Distance to Gas Station` , and see how we can apply the above trial and error algorithm.

`I -- Construct a candidate solution`

To construct a candidate solution, we need to understand first what the desired solution is. The problem description requires we output the **minimum value** of the maximum distance between adjacent gas stations. This value is nothing more than a non-negative double value, since the distance between any adjacent stations cannot be negative and in general it should be of double type. Therefore our candidate

solution should also be a non-negative double value.

`II -- Search space formed by all the candidate solutions`

Let `max` be the maximum value of the distances between any adjacent stations without adding those additional stations, then our desired solution should lie within the range `[0, max]`. This is because adding extra stations (and placing them at proper positions) can only reduce the maximum distance between any adjacent stations. Therefore our search space will be `[0, max]` (or the upper bound should be at least `max`).

`III -- Verify a given candidate solution`

This is the key part of this trial and error algorithm. So given a candidate double value `d`, how do we determine if it can be the minimum value of the maximum distance between any adjacent stations after adding `K` extra stations? The answer is by counting.

If `d` can be the minimum value of the maximum distance between any adjacent stations after adding `K` extra stations, then the following two conditions should be true at the same time:

1. The total number of stations we can add between all adjacent stations cannot go beyond `K`. In other words, assume we added `cnt_i` stations in between the pair of adjacent stations `(i, i+1)`, then we should have `sum(cnt_i) <= K`.
2. For each pair of adjacent stations `(i, i+1)`, the minimum value of the maximum distance between adjacent stations after adding `cnt_i` additional stations cannot go beyond `d`. If the original distance between station `i` and `i+1` is `d_i = stations[i+1] - stations[i]`, then after adding `cnt_i` additional stations, the minimum value of maximum distance between any adjacent stations we can obtain is `d_i / (cnt_i + 1)`, which is achieved by placing those stations evenly in between stations `i` and `i+1`. Therefore we require: `d_i / (cnt_i + 1) <= d`, which is equivalent to `cnt_i >= (d_i/d - 1)`.

So you can see that the two conditions are actually "contradictory" to each other: to meet condition **1**, we want `cnt_i` to be as small as possible; but to meet condition **2**, we'd like it to be as large as possible. So in practice, we can alway choose the set of smallest `cnt_i`'s that satisfy the second condition and double check if they also meet the first condition. If they do, then `d` will set an upper limit on the final minimum value obtainable; otherwise `d` will set a lower limit on this minimum value.

This verification algorithm runs at `O(n)`, where `n` is the length of the `stations` array. This is acceptable if we can walk the search space very efficiently (which can be done at the order of `O(log(max/step))`, with `step = 10^-6`). In particular, this is much faster than the straightforward `O(Klogn)` solution where we add the

stations one by one in a greedy manner (i.e., always reduce the current maximum distance first), given that `K` could be orders of magnitude larger than `n` (note this greedy algorithm can be optimized to run at `O(nlogn)`, see wannacry89's post here).

`IV -- How to walk the search space efficiently`

Up to this point, we know the search space, know how to construct the candidate solution and how to verify it by counting, we still need one last piece for the puzzle: how to walk the search space.

Of course we can do the naive linear walk by trying each double value from `0` up to `max` at a step of `10^-6` and choose the first double value `d` such that `sum(d_i/d - 1) <= K`. The time complexity will be `O(n * max/step)`. However, given that `max/step` can be much larger than `K`, this algorithm could end up being much worse than the aforementioned `O(Klogn)` solution.

The key observation here is that the candidate solutions are ordered naturally in ascending order, so a binary search is possible. Another fact is the non-decreasing property of the `sum` function: give two double values `d1` and `d2` such that `d1 < d2`, then `sum(d_i/d1 - 1) >= sum(d_i/d2 - 1)`. So here is what a binary walk of the search space may look like:

1. Let `[l, r]` be the current search space that is initialized as `l = 0`, `r = max`.
2. As long as `r - l >= 10^-6`, compute the middle point `d = (l + r) / 2` and evaluate `sum(d_i/d - 1)`.
3. If `sum(d_i/d - 1) <= K`, we throw away the right half of current search space by setting `r = d`; else we throw away the left half by setting `l = d`. Then go to step 2.

`V -- Putting everything together, aka, solutions`

Despite the above lengthy analyses, the final solution is much simpler to write once you understand it. Here is the Java program for the trial and error algorithm. The time complexity is `O(n*log(max/step))` and space complexity is `O(1)`.

```java
public double minmaxGasDist(int[] stations, int K) {
    double l = 0, r = stations[stations.length - 1] - stations[0]; // assuming the positions are sorted, or alternatively we can go through the stations array to find the maximum distance between all adjacent stations

    while (r - l >= 1e-6) {
        double d = (r + l) / 2;

        int cnt = 0;

        for (int i = 0; i < stations.length - 1; i++) {
            cnt += Math.ceil((stations[i + 1] - stations[i]) / d) - 1;
        }

        if (cnt <= K) {
            r = d;
        } else {
            l = d;
        }
    }

    return l;
}
```

Final remarks: This post is just a reminder to you that the trial and error algorithm is worth trying if you find all other common solutions suffer severely from bad time or space performance. Also it's always recommended to perform a quick evaluation of the search space size and potential verification algorithm to estimate the complexity before you are fully committed to this algorithm.

written by fun4LeetCode original link here

Solution 3

At first I thought this was a sequential station add problem, not concurrent station add problem. Quickly you see that it is the concurrent station add problem when you try the former and it fails. The idea now is that we view each inter-station interval as having a fraction of the total ground covered. That is, we assign the extra stations to the intervals according to how much of the total ground they cover. Then, we have up to N leftover because we are always taking the floor (integer) when calculating these assignments. So we take care of the remainder greedily using priorityqueue (O(remainingmax*log (intervals)) == O(n log n)). Also I assume the stations are originally given potentially out of order, so the only way I can consider them as adjacency intervals it to sort. This also costs O(n log n).

```
class Solution {
    public double minmaxGasDist(int[] stations, int K) {

        Arrays.sort(stations);
        PriorityQueue<Interval> que = new PriorityQueue<Interval>(new Comparator<Int
erval>() {
            public int compare(Interval a, Interval b) {

                double diff = a.distance() - b.distance();
                if (diff < 0) { return +1; }
                else if (diff > 0) { return -1; }
                else { return 0; }
            }
        });

        double leftToRight = stations[stations.length-1] - stations[0];
        int remaining = K;

        for (int i = 0; i < stations.length-1; i++) {
            int numInsertions = (int)(K*(((double)(stations[i+1]-stations[i]))/leftT
oRight));
            que.add(new Interval(stations[i], stations[i+1], numInsertions));
            remaining -= numInsertions;
        }

        while (remaining > 0) {
            Interval interval = que.poll();
            interval.numInsertions++;
            que.add(interval);
            remaining--;
        }

        Interval last = que.poll();
        return last.distance();

    }

    class Interval {
        double left;
        double right;
        int numInsertions;
        double distance() { return (right - left)/  ((double)(numInsertions+1)) ; }
        Interval(double left, double right, int numInsertions) { this.left = left;
this.right = right; this.numInsertions = numInsertions; }
    }
}
```

p.s. I had originally tried using only PriorityQueue + greedy approach alone, without the pre-assignment based on proportion. This was TLE.

The binary approach should not be faster than this, and this one is easier to understand IMO.

written by wannacry89 original link here