774. Minimize Max Distance to Gas Station (/problems/minimize-max-distance-to-gas-station/)

Jan. 27, 2018 | 1.7K views

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.

Approach #1: Dynamic Programming [Memory Limit Exceeded]

Intuition

Let `dp[n][k]` be the answer for adding `k` more gas stations to the first `n` intervals of stations. We can develop a recurrence expressing `dp[n][k]` in terms of `dp[x][y]` with smaller `(x, y)`.

Algorithm

Say the `i` th interval is `deltas[i] = stations[i+1] - stations[i]`. We want to find `dp[n+1][k]` as a recursion. We can put `x` gas stations in the `n+1` th interval for a best distance of `deltas[n+1] / (x+1)`, then the rest of the intervals can be solved with an answer of `dp[n][k-x]`. The answer is the minimum of these over all `x`.

From this recursion, we can develop a dynamic programming solution.

Java Python Copy

```java
class Solution {
    public double minmaxGasDist(int[] stations, int K) {
        int N = stations.length;
        double[] deltas = new double[N-1];
        for (int i = 0; i < N-1; ++i)
            deltas[i] = stations[i+1] - stations[i];

        double[][] dp = new double[N-1][K+1];
        //dp[i][j] = answer for deltas[:i+1] when adding j gas stations
        for (int i = 0; i <= K; ++i)
            dp[0][i] = deltas[0] / (i+1);

        for (int p = 1; p < N-1; ++p)
            for (int k = 0; k <= K; ++k) {
                double bns = 999999999;
                for (int x = 0; x <= k; ++x)
                    bns = Math.min(bns, Math.max(deltas[p] / (x+1), dp[p-1][k-x]));
                dp[p][k] = bns;
            }

        return dp[N-2][K];
    }
}
```

Complexity Analysis

- Time Complexity: $O(NK^2)$, where N is the length of `stations`.
- Space Complexity: $O(NK)$, the size of `dp`.

Approach #2: Brute Force [Time Limit Exceeded]

Intuition

As in *Approach #1*, let's look at `deltas`, the distances between adjacent gas stations.

Let's repeatedly add a gas station to the current largest interval, so that we add `K` of them total. This greedy approach is correct because if we left it alone, then our answer never goes down from that point on.

Algorithm

To find the largest current interval, we keep track of how many parts `count[i]` the `i`th (original) interval has become. (For example, if we added 2 gas stations to it total, there will be 3 parts.) The new largest interval on this section of road will be `deltas[i] / count[i]`.

Java Python Copy

```java
class Solution {
    public double minmaxGasDist(int[] stations, int K) {
        int N = stations.length;
        double[] deltas = new double[N-1];
        for (int i = 0; i < N-1; ++i)
            deltas[i] = stations[i+1] - stations[i];

        int[] count = new int[N-1];
        Arrays.fill(count, 1);

        for (int k = 0; k < K; ++k) {
            // Find interval with largest part
            int best = 0;
            for (int i = 0; i < N-1; ++i)
                if (deltas[i] / count[i] > deltas[best] / count[best])
                    best = i;

            // Add gas station to best interval
            count[best]++;
        }

        double ans = 0;
        for (int i = 0; i < N-1; ++i)
            ans = Math.max(ans, deltas[i] / count[i]);

        return ans;
    }
```

Complexity Analysis

- Time Complexity: $O(NK)$, where N is the length of `stations`.
- Space Complexity: $O(N)$, the size of `deltas` and `count`.

Approach #3: Heap [Time Limit Exceeded]

Intuition

Following the intuition of *Approach #2*, if we are taking a repeated maximum, we can replace this with a heap data structure, which performs repeated maximum more efficiently.

Algorithm

As in *Approach #2*, let's repeatedly add a gas station to the next larget interval `K` times. We use a heap to know which interval is largest. In Python, we use a negative priority to simulate a max heap with a min heap.

Java Python Copy

```
class Solution {
    public double minmaxGasDist(int[] stations, int K) {
        int N = stations.length;
        PriorityQueue<int[]> pq = new PriorityQueue<int[]>((a, b) ->
            (double)b[0]/b[1] < (double)a[0]/a[1] ? -1 : 1);
        for (int i = 0; i < N-1; ++i)
            pq.add(new int[]{stations[i+1] - stations[i], 1});

        for (int k = 0; k < K; ++k) {
            int[] node = pq.poll();
            node[1]++;
            pq.add(node);
        }

        int[] node = pq.poll();
        return (double)node[0] / node[1];
    }
}
```

Complexity Analysis

- Time Complexity: $O(K \log N)$, where N is the length of `stations`.
- Space Complexity: $O(N)$, the size of `deltas` and `count`.

Approach #4: Binary Search [Accepted]

Intuition

Let's ask `possible(D)`: with `K` (or less) gas stations, can we make every adjacent distance between gas stations at most `D`? This function is monotone, so we can apply a binary search to find D^*.

Algorithm

More specifically, there exists some `D*` (the answer) for which `possible(d) = False` when `d < D*` and `possible(d) = True` when `d > D*`. Binary searching a monotone function is a typical technique, so let's focus on the function `possible(D)`.

When we have some interval like `X = stations[i+1] - stations[i]`, we'll need to use $\lfloor \frac{X}{D} \rfloor$ gas stations to ensure every subinterval has size less than `D`. This is independent of other intervals, so in total we'll need to use $\sum_i \lfloor \frac{X_i}{D} \rfloor$ gas stations. If this is at most `K`, then it is possible to make every adjacent distance between gas stations at most `D`.

Java Python Copy

```java
class Solution {
    public double minmaxGasDist(int[] stations, int K) {
        double lo = 0, hi = 1e8;
        while (hi - lo > 1e-6) {
            double mi = (lo + hi) / 2.0;
            if (possible(mi, stations, K))
                hi = mi;
            else
                lo = mi;
        }
        return lo;
    }

    public boolean possible(double D, int[] stations, int K) {
        int used = 0;
        for (int i = 0; i < stations.length - 1; ++i)
            used += (int) ((stations[i+1] - stations[i]) / D);
        return used <= K;
    }
}
```

Complexity Analysis

- Time Complexity: $O(N \log W)$, where N is the length of `stations`, and $W = 10^{14}$ is the range of possible answers (10^8), divided by the acceptable level of precision (10^{-6}).
- Space Complexity: $O(1)$ in additional space complexity.

Analysis written by: @awice (https://leetcode.com/awice).

Rate this article: **(/ratings/107/405/?return=/articles/minimize-max-distance-to-gas-station/) (/ratings/107/405/?ret**

Previous (/articles/jewels-and-stones/) Next (/articles/swap-adjacent-in-lr-string/)

Join the conversation

Login to Reply

D (https://discuss.leetcode.com/user/dragonitedd) **dragonitedd** commented last month

@cja (https://discuss.leetcode.com/uid/331725)
great solution, but I think the complexity is O(max(n,K) * log(n))

D (https://discuss.leetcode.com/user/dragonitedd) **dragonitedd** commented last month

@qinzhiguo (https://discuss.leetcode.com/uid/289682) said in Minimize Max Distance to Gas Station (https://discuss.leetcode.com/post/244540):

> Actually we know if K>0, there will be a minmax distance exist and this minmax<max gap for this question. So for the binary search optimized solution is that we can sort the gap and set the high variable to the gap[gap.length-1] to speed up algorithm. Java code like below.

Great solution.
I think W should be min(max_gap * K, 10^8) as stations will be in range [0, 10^8]

C (https://discuss.leetcode.com/user/cacs)

cacs commented 2 months ago

Regarding the priority queue, this solution does not meet TLE. Please update if possible. Thanks. https://discuss.leetcode.com/topic/118812/simple-10-line-python-o-n-log-n-priority-queue-solution (https://discuss.leetcode.com/topic/118812/simple-10-line-python-o-n-log-n-priority-queue-solution)

Q (https://discuss.leetcode.com/user/qinzhiguo)

qinzhiguo commented 2 months ago

Actually we know if K>0, there will be a minmax distance exist and this minmax<max gap for this question. So for the binary search optimized solution is that we can sort the gap and set the high variable to the gap[gap.length-1] to speed up algorithm. Java code like below.

class Solution {
public double minmaxGasDist(int[] stations, int K) {
int[] gap = new int[stations.length-1];
for(int i=0;i<stations.length-1;i++){
gap[i]= stations[i+1] - stations[i];
}
Arrays.sort(gap);
double low= 0, high=gap[gap.length-1];

```
    while(high-low >1e-6){
        double mid = (double)(low+high)/2.0;
        if(possible(mid,gap,K)){
            high=mid;
        }else{
            low=mid;
        }
    }
    return low;
}

public boolean possible(double step, int[] gap, int K){
    int need_to_use =0;
    for(int i=0;i<gap.length;i++){
        need_to_use += (int)(gap[i]/step);
    }
    return need_to_use <= K;
}
```

}

The time complexity will be O(N logW), W should be the W = max_gap*10^6
The space complexity will be O(N), to store the gap

T (https://discuss.leetcode.com/user/thestraycat)

TheStrayCat commented 2 months ago

@ManuelP (https://discuss.leetcode.com/uid/27445) The number of pieces into which the i-th interval is divided.


(https://discuss.leetcode.com/user/cja)

cja commented 2 months ago

Simple 10-line Python O(n log(n)) priority queue solution.

We know that the minmax distance is no more than (station(n)-station(1)) / K, so let's start from there:

```
    def minmaxGasDist(self, stations, K):
        d = (stations[len(stations)-1] - stations[0]) / float(K)
        heap = []
        for i in range(len(stations)-1):
            n = max(1, int((stations[i+1]-stations[i]) / d))
            K -= (n-1)
            heapq.heappush(heap, (float(stations[i]-stations[i+1]) / n, stations[i], stations
[i+1], n))

        for i in range(K):
            (d, a, b, n) = heap[0]
            heapq.heapreplace(heap, ((a-b)/(n+1.0), a, b, n+1))
        return -heap[0][0]
```


(https://discuss.leetcode.com/user/manuelp)

ManuelP commented 2 months ago

@TheStrayCat (https://discuss.leetcode.com/uid/316376) What does your `count[i]` mean?

TheStrayCat commented 2 months ago

(https://discuss.leetcode.com/user/thestraycat)

Hi all,

I would like to show my solution which gives the exact answer and runs faster than binary search (244 ms < 1030 ms).

We have `num` intervals (`num = len(stations)-1`) with lengths stored in the `delta` list. Our goal is to subdivide them into `K+nums` subintervals by adding `K` additional breakpoints so as to make the longest subinterval as short as possible.

Intuitively, this would be true if all subintervals had approximately equal lengths, that is, `delta[i]/count[i]` were about the same for all `i`. We know that the sum of all elements in `delta` is the difference between the last and the first elements in `stations`, call it `n` (here I cheated a little bit by making an assumption that `stations` was sorted in all test cases, and by pure chance I was correct). Hence, `count[i]` should be not too far away from `1 + delta[i]*K/n`, which is the hypothetical number of parts in case breakpoints are allocated proportionally based on the distance.

But unfortunately, `delta[i]*K/n` is not always an integer. However, it is reasonable to assume that at least `count[i]>=1+int(delta[i]*K/n)`.

Now we allocate `1+int(delta[i]*K/n)` breakpoints to each interval, compute the remaining number of points (which never exceeds `num`) and finish the job by applying a priority queue as in approach #3. Here's the complete Python code:

```
from heapq import heappop, heappush

class Solution:
    def minmaxGasDist(self, stations, K):
        num = len(stations)-1
        delta = [stations[i+1]-stations[i] for i in range(num)]
        n = stations[-1]-stations[0]
        count = [1+int(r*K/n) for r in delta]
        steps = K+num-sum(count)

        Q = []
        for i in range (num):
            heappush(Q, (-delta[i]/count[i], count[i]))

        for i in range(steps):
            max_dist = heappop(Q)
            new_dist, new_count = max_dist[1]*max_dist[0]/(max_dist[1]+1), max_dist[1]+1
            heappush(Q,(new_dist,new_count))

        ints = [-heappop(Q)[0] for i in range(len(Q))]

        return max(ints)
```

ManuelP commented 2 months ago

(https://discuss.leetcode.com/user/manuelp)

@awice (https://discuss.leetcode.com/uid/71269) You forgot to sort the stations. Nothing in the problem says they're sorted, and at least your approach #4 needs that (I didn't try the others).

View original thread (https://discuss.leetcode.com/topic/118707)

Load more comments...