774. Minimize Max Distance to Gas Station

Jan. 27, 2018 | 12.8K views

Average Rating: 4.63 (27 votes)

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

```python
class Solution(object):
    def minmaxGasDist(self, stations, K):
        N = len(stations)
        deltas = [stations[i+1] - stations[i] for i in xrange(N-1)]
        dp = [[0.0] * (K+1) for _ in xrange(N-1)]
        #dp[i][j] = answer for deltas[:i+1] when adding j gas stations
        for i in xrange(K+1):
            dp[0][i] = deltas[0] / float(i + 1)

        for p in xrange(1, N-1):
            for k in xrange(K+1):
                dp[p][k] = min(max(deltas[p] / float(x+1), dp[p-1][k-x])
                               for x in xrange(k+1))

        return dp[-1][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

```python
class Solution(object):
    def minmaxGasDist(self, stations, K):
        N = len(stations)
        deltas = [float(stations[i+1] - stations[i]) for i in xrange(N-1)]
        count = [1] * (N - 1)

        for _ in xrange(K):
            #Find interval with largest part
            best = 0
            for i, x in enumerate(deltas):
                if x / count[i] > deltas[best] / count[best]:
                    best = i

            #Add gas station to best interval
            count[best] += 1

        return max(x / count[i] for i, x in enumerate(deltas))
```

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

```python
class Solution(object):
    def minmaxGasDist(self, stations, K):
        pq = [] #(-part_length, original_length, num_parts)
        for i in xrange(len(stations) - 1):
            x, y = stations[i], stations[i+1]
            pq.append((x-y, y-x, 1))
        heapq.heapify(pq)

        for _ in xrange(K):
            negnext, orig, parts = heapq.heappop(pq)
            parts += 1
            heapq.heappush(pq, (-(orig / float(parts)), orig, parts))

        return -pq[0][0]
```

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

```python
class Solution(object):
    def minmaxGasDist(self, stations, K):
        def possible(D):
            return sum(int((stations[i+1] - stations[i]) / D)
                       for i in xrange(len(stations) - 1)) <= K

        lo, hi = 0, 10**8
        while hi - lo > 1e-6:
            mi = (lo + hi) / 2.0
            if possible(mi):
                hi = mi
            else:
                lo = mi
        return lo
```

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.

Rate this article: