<https://leetcode.com/problems/gas-station/description/>

There are n gas stations along a circular route, where the amount of gas at the ith station is gas[i].

You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from the ith station to its next (i + 1)th station. You begin the journey with an empty tank at one of the gas stations.

Given two integer arrays gas and cost, return the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return -1. If there exists a solution, it is guaranteed to be unique.

**Example 1:**

**Input:** gas = [1,2,3,4,5], cost = [3,4,5,1,2]

**Output:** 3

**Explanation:**

Start at station 3 (index 3) and fill up with 4 unit of gas. Your tank = 0 + 4 = 4

Travel to station 4. Your tank = 4 - 1 + 5 = 8

Travel to station 0. Your tank = 8 - 2 + 1 = 7

Travel to station 1. Your tank = 7 - 3 + 2 = 6

Travel to station 2. Your tank = 6 - 4 + 3 = 5

Travel to station 3. The cost is 5.

Your gas is just enough to travel back to station 3.Therefore, return 3 as the starting index.

**Example 2:**

**Input:** gas = [2,3,4], cost = [3,4,3]

**Output:** -1

**Explanation:**

You can't start at station 0 or 1, as there is not enough gas to travel to the next station.

Let's start at station 2 and fill up with 4 unit of gas. Your tank = 0 + 4 = 4

Travel to station 0. Your tank = 4 - 3 + 2 = 3

Travel to station 1. Your tank = 3 - 3 + 3 = 3

You cannot travel back to station 2, as it requires 4 unit of gas but you only have 3.

Therefore, you can't travel around the circuit once no matter where you start.

**Constraints:**

n == gas.length == cost.length

1 <= n <= 10^5

0 <= gas[i], cost[i] <= 10^4

**Attempt 1: 2025-02-24**

**Solution 1: Brute Force (10 min, TLE 34/39)**

class Solution {

    public int canCompleteCircuit(int[] gas, int[] cost) {

        int n = gas.length;

        for(int i = 0; i < n; i++) {

            int totalFuel = 0;

            int stopCount = 0;

            int j = i;

            while(stopCount < n) {

                totalFuel += gas[j % n] - cost[j % n];

                // Whenever we reach a negative fuel, cannot proceed

                if(totalFuel < 0) {

                    break;

                }

                stopCount++;

                j++;

            }

            // Cover all the stops & our fuel left is 0 or more than that

            if(stopCount == n && totalFuel >= 0) {

                return i;

            }

        }

        return -1;

    }

}

Time Complexity: O(N^2)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/gas-station/solutions/1706142/java-c-python-an-explanation-that-ever-exists-till-now/>

int n = gas.length;

for(int i = 0; i < n; i++){

int totalFuel = 0;

int stopCount = 0, j = i;

while(stopCount < n){

totalFuel += gas[j % n] - cost[j % n];

if(totalFuel < 0) break; // whenever we reach -ve

stopCount++;

j++;

}

if(stopCount == n && totalFuel >= 0) return i; // cover all the stops & our fuel left is 0 or more than that

}

return -1;

**Solution 2: Greedy (180 min)**

class Solution {

    public int canCompleteCircuit(int[] gas, int[] cost) {

        int n = gas.length;

// Total fuel available

        int totalFuel = 0;

// Fuel in the tank currently

        int curFuel = 0;

        int start = 0;

        for(int i = 0; i < n; i++) {

            totalFuel += gas[i] - cost[i];

            curFuel += gas[i] - cost[i];

// If curFuel is negative, reset and try starting from the next station

            if(curFuel < 0) {

                curFuel = 0;

                start = i + 1;

            }

        }

// If totalFuel is non-negative, it means there's a solution

        return (totalFuel < 0) ? -1 : start;

    }

}

Time Complexity: O(N)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/gas-station/solutions/1706142/java-c-python-an-explanation-that-ever-exists-till-now/comments/1232320>

For those who are still confused why we are not checking again at all indices (in optimized approach).

Firstly we can start at index i, only when gas[i] - cost[i]>=0 because otherwise, we won't be able to reach the next station.

Also, we will try to have the maximum value of gas in our tank at a given index because having greater fuel maximizes chances of getting success(here success means completing one circular trip) for future.

So if we try to start at index i where gas[i]-cost[i]>=0 and we get to know that we are not able to reach at index j (starting from i) , so it is not beneficial to check at indices (i+1,i+2,...,j-1) because if we start at indices (i+1,i+2,...j-1) it will result in less amount of gas in our car at station j which will definitely be a failure.

class Solution {

public int canCompleteCircuit(int[] gas, int[] cost) {

int n = gas.length;

int total\_surplus = 0;

int surplus = 0;

int start = 0;

for(int i = 0; i < n; i++){

total\_surplus += gas[i] - cost[i];

surplus += gas[i] - cost[i];

if(surplus < 0){

surplus = 0;

start = i + 1;

}

}

return (total\_surplus < 0) ? -1 : start;

}

}

**Refer to Deepseek**

Certainly! The problem "Gas Station" (LeetCode 134) is a classic greedy algorithm problem. The goal is to determine the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return -1.

Here's a Java solution for the problem:

class Solution {

public int canCompleteCircuit(int[] gas, int[] cost) {

int totalGas = 0; // Total gas available

int currentGas = 0; // Gas in the tank currently

int startStation = 0; // Starting station index

for (int i = 0; i < gas.length; i++) {

totalGas += gas[i] - cost[i]; // Calculate the total gas available

currentGas += gas[i] - cost[i]; // Calculate the current gas in the tank

// If currentGas is negative, reset and try starting from the next station

if (currentGas < 0) {

currentGas = 0;

startStation = i + 1;

}

}

// If totalGas is non-negative, it means there's a solution

return totalGas >= 0 ? startStation : -1;

}

}

**Explanation:**

Total Gas Calculation: We calculate the total gas available by summing up the difference between gas[i] and cost[i] for all stations. If the total gas is negative, it means it's impossible to complete the circuit, so we return -1.

Current Gas Calculation: As we iterate through the stations, we keep track of the current gas in the tank. If at any point the current gas becomes negative, it means we cannot start from any previous starting station, so we reset the current gas to 0 and try starting from the next station.

Starting Station: If we can complete the loop without the current gas becoming negative, and the total gas is non-negative, then the starting station is the one we last reset to.

**Example:**

public static void main(String[] args) {

Solution solution = new Solution();

int[] gas = {1, 2, 3, 4, 5};

int[] cost = {3, 4, 5, 1, 2};

System.out.println(solution.canCompleteCircuit(gas, cost)); // Output: 3

}

In this example, the output is 3, which means starting from station 3 allows us to complete the circuit.

This solution runs in O(n) time complexity, where n is the number of gas stations, and uses O(1) extra space.

You're absolutely correct, and the explanation from the given page is indeed a very clear and concise way to understand the logic behind the greedy algorithm for this problem. Let’s break it down further and connect it to your observation about the critical point:

**Key Insight from the Given Page**

The explanation emphasizes two main points:

Start only at stations where gas[i] - cost[i] >= 0:

If gas[i] - cost[i] < 0, starting at i is impossible because you won’t even have enough gas to reach the next station (i+1).

This ensures that we only consider \*\*valid starting candidates\*\*.

Skipping stations between i and j:

If starting at i (where gas[i] - cost[i] >= 0) fails at j, then starting at any station between i+1 and j-1 will also fail.

This is because the gas accumulated from i to j-1 is already a net loss, and starting later (e.g., at k where i < k < j) would result in even less gas at j.

**Why Starting at k (Between i and j) is Worse**

Let’s formalize this logic:

Starting at i:

You accumulate gas from i to j-1.

The net gas from i to j-1 is negative:

(gas[i] - cost[i]) + (gas[i+1] - cost[i+1]) + ... + (gas[j-1] - cost[j-1]) < 0

Starting at k (where i < k < j):

You skip the segment i to k-1, so the gas accumulated from k to j-1 is:

(gas[k] - cost[k]) + (gas[k+1] - cost[k+1]) + ... + (gas[j-1] - cost[j-1])

Since the full segment i to j-1 is a net loss, the sub-segment k to j-1 is at best equally bad or worse:

If the segment i to k-1 contributed positively, skipping it makes the situation worse.

If the segment i to k-1 contributed negatively, skipping it doesn’t help because the full segment i to j-1 is still a net loss.

**Your Critical Point**

You highlighted the most critical part of the explanation:

"Firstly we can start at index i, only when gas[i] - cost[i]>=0 because otherwise, we won't be able to reach the next station."

This is indeed the \*\*foundation\*\* of the greedy algorithm. By ensuring that we only start at stations where gas[i] - cost[i] >= 0, we guarantee that:

The starting station is \*\*locally valid\*\* (you can at least reach the next station).

If starting at i fails at j, then any station between i and j is \*\*guaranteed to fail\*\* because:

The segment i to j-1 is a net loss.

Starting later (e.g., at k) would result in \*\*less gas\*\* at j than starting at i.

**Example to Illustrate**

Let’s use an example to solidify this understanding:

gas = [1, 2, 3, 4, 5]

cost = [3, 4, 5, 1, 2]

Start at station 0:

gas[0] - cost[0] = 1 - 3 = -2 → Invalid starting point (skip).

Start at station 1:

gas[1] - cost[1] = 2 - 4 = -2 → Invalid starting point (skip).

Start at station 2:

gas[2] - cost[2] = 3 - 5 = -2 → Invalid starting point (skip).

Start at station 3:

gas[3] - cost[3] = 4 - 1 = 3 → Valid starting point.

Move to station 4: 3 + (5 - 2) = 6 → Positive.

Move to station 0: 6 + (1 - 3) = 4 → Positive.

Move to station 1: 4 + (2 - 4) = 2 → Positive.

Move to station 2: 2 + (3 - 5) = 0 → Valid.

Total Gas Check:

(1-3) + (2-4) + (3-5) + (4-1) + (5-2) = -2 - 2 - 2 + 3 + 3 = 0 → Valid.

**Why the Given Explanation is Better**

The explanation from the given page is more intuitive because:

It directly addresses the local validity of starting points (gas[i] - cost[i] >= 0).

It emphasizes the greedy nature of the algorithm: by starting at the earliest valid station and skipping invalid segments, we efficiently find the solution.

It avoids overcomplicating the logic with mathematical proofs, making it easier to understand.

**Final Thoughts**

Both explanations are correct, but the one from the given page is more concise and focuses on the \*\*key insights\*\* that make the greedy algorithm work. Your critical point about starting only at stations where gas[i] - cost[i] >= 0 is absolutely correct and is the foundation of the algorithm. By combining this with the idea of skipping invalid segments, we arrive at an efficient and elegant solution.