**Others**

**Super Washing Machines**

**Super Washing Machines**

You have **n** super washing machines on a line. Initially, each washing machine has some dresses or is empty.

For each **move**, you could choose **any m** (1 ≤ m ≤ n) washing machines, and pass **one dress** of each washing machine to one of its adjacent washing machines **at the same time**.

Given an integer array representing the number of dresses in each washing machine from left to right on the line, you should find the **minimum number of moves** to make all the washing machines have the same number of dresses. If it is not possible to do it, return -1.

**Example1**

**Input:** [1,0,5]

**Output:** 3

**Explanation:**

1st move: 1 0 <-- 5 => 1 1 4

2nd move: 1 <-- 1 <-- 4 => 2 1 3

3rd move: 2 1 <-- 3 => 2 2 2

**Example2**

**Input:** [0,3,0]

**Output:** 2

**Explanation:**

1st move: 0 <-- 3 0 => 1 2 0

2nd move: 1 2 --> 0 => 1 1 1

**Example3**

**Input:** [0,2,0]

**Output:** -1

**Explanation:**

It's impossible to make all the three washing machines have the same number of dresses.

**Note:**

1. The range of n is [1, 10000].
2. The range of dresses number in a super washing machine is [0, 1e5].

Solution

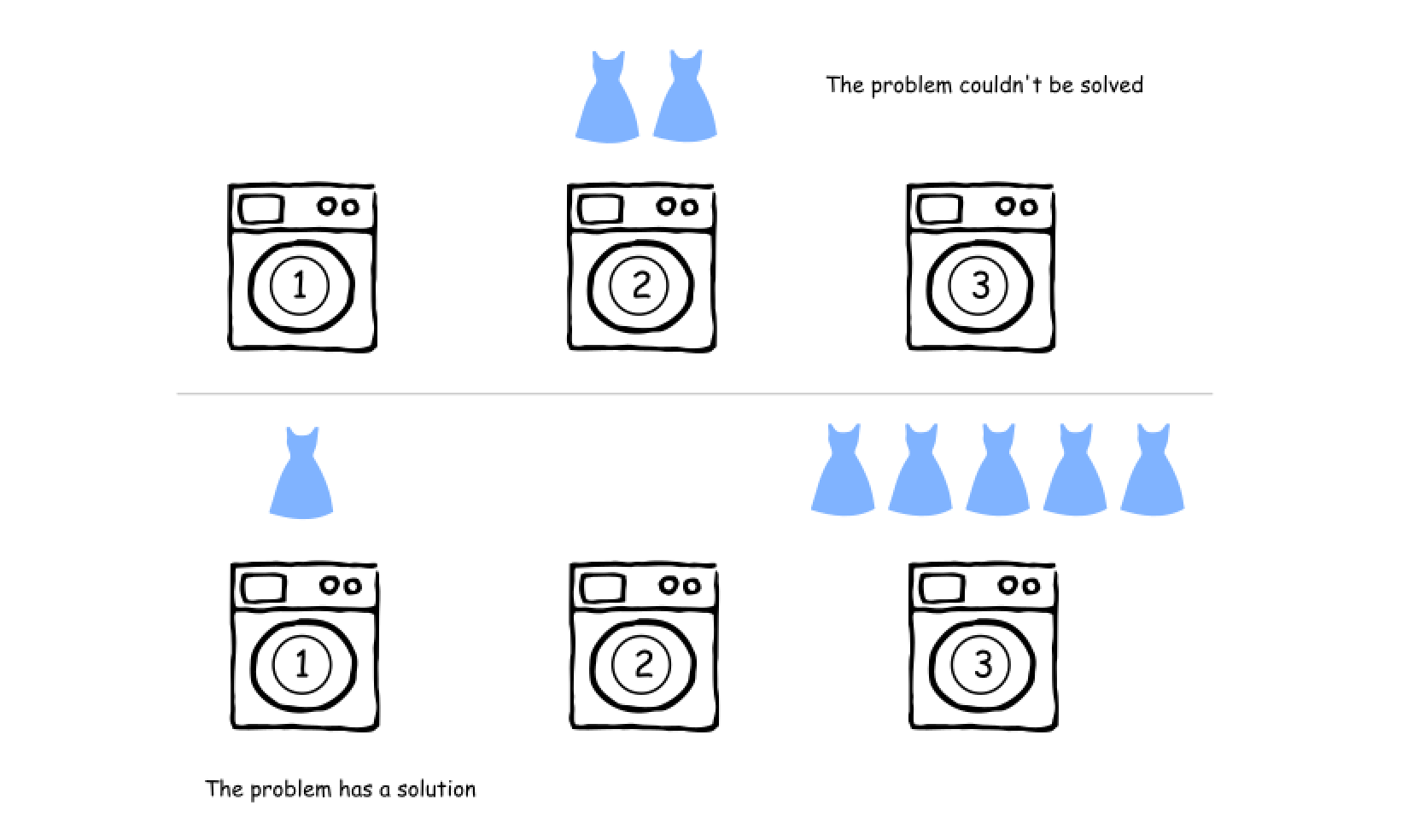
Approach 1: Greedy.

**Intuition**

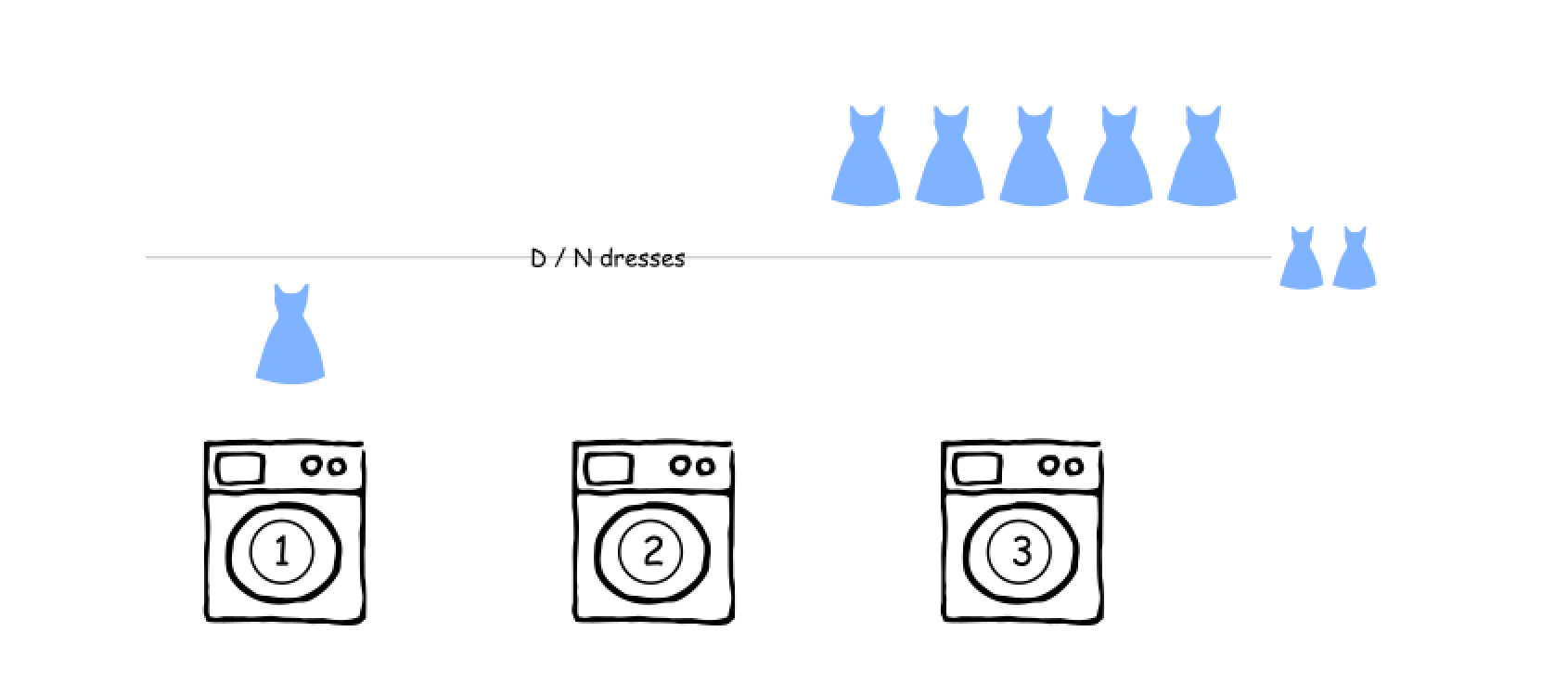
This greedy problem is very similar to [Gas station problem](https://leetcode.com/articles/gas-station/), and could be solved in linear time as well.

First of all - could the problem be solved or not?

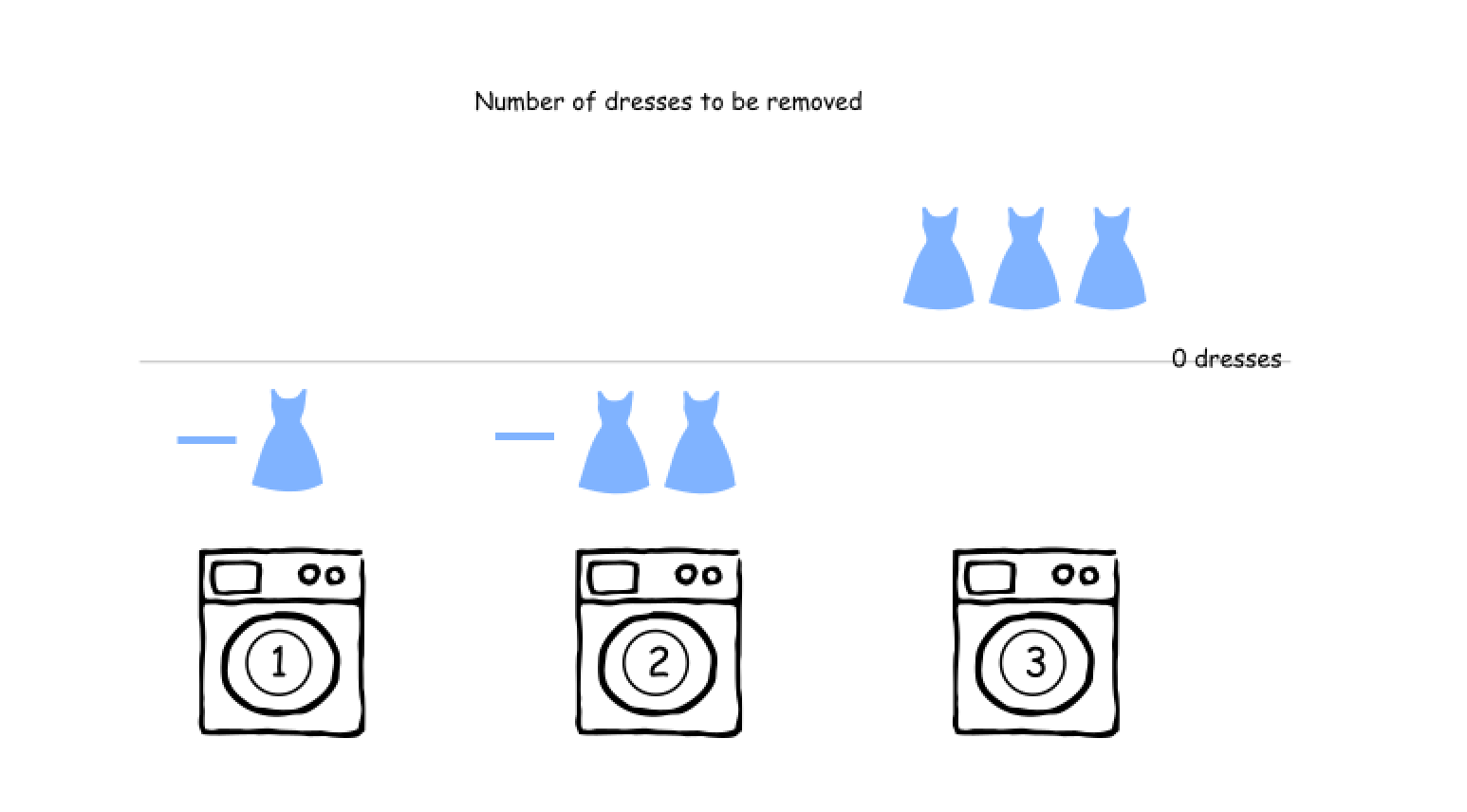
Yes, if the dresses could be divided into N equal parts where N is number of machines. In other words, N should be a divisor of the number of dresses D.



Now it's easy to compute the number of dresses that each machine should have: D / N. The starting numbers of dresses in the machines move around this D / N average value.



The standard ML trick is to normalize the data, so that the average value would be zero. For that, one could replace the actual number of dresses in the machine by the number of dresses to be removed. This number could be negative, if one actually needs *to add* the dresses into the machine.



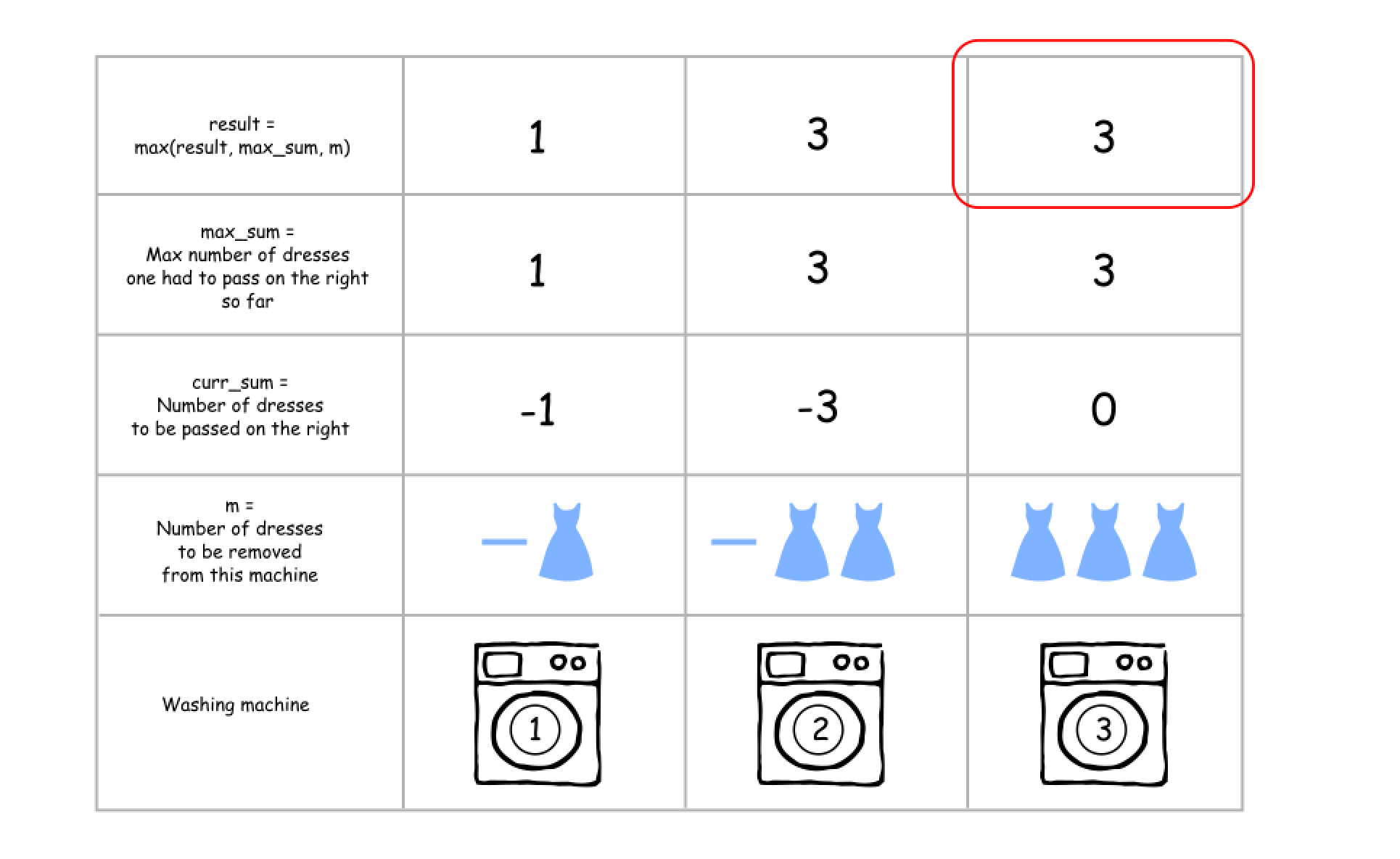
As for the [gas station problem](https://leetcode.com/articles/gas-station), one starts from the beginning and checks the standard set for such problems: the current element, the current sum, and the maximum sum seen so far :

* m. Number of dresses to be removed from the current machine.
* curr\_sum. Number of dresses to be passed on the right.
* max\_sum. Maximum number of dresses one had to pass on the right at this point or before.

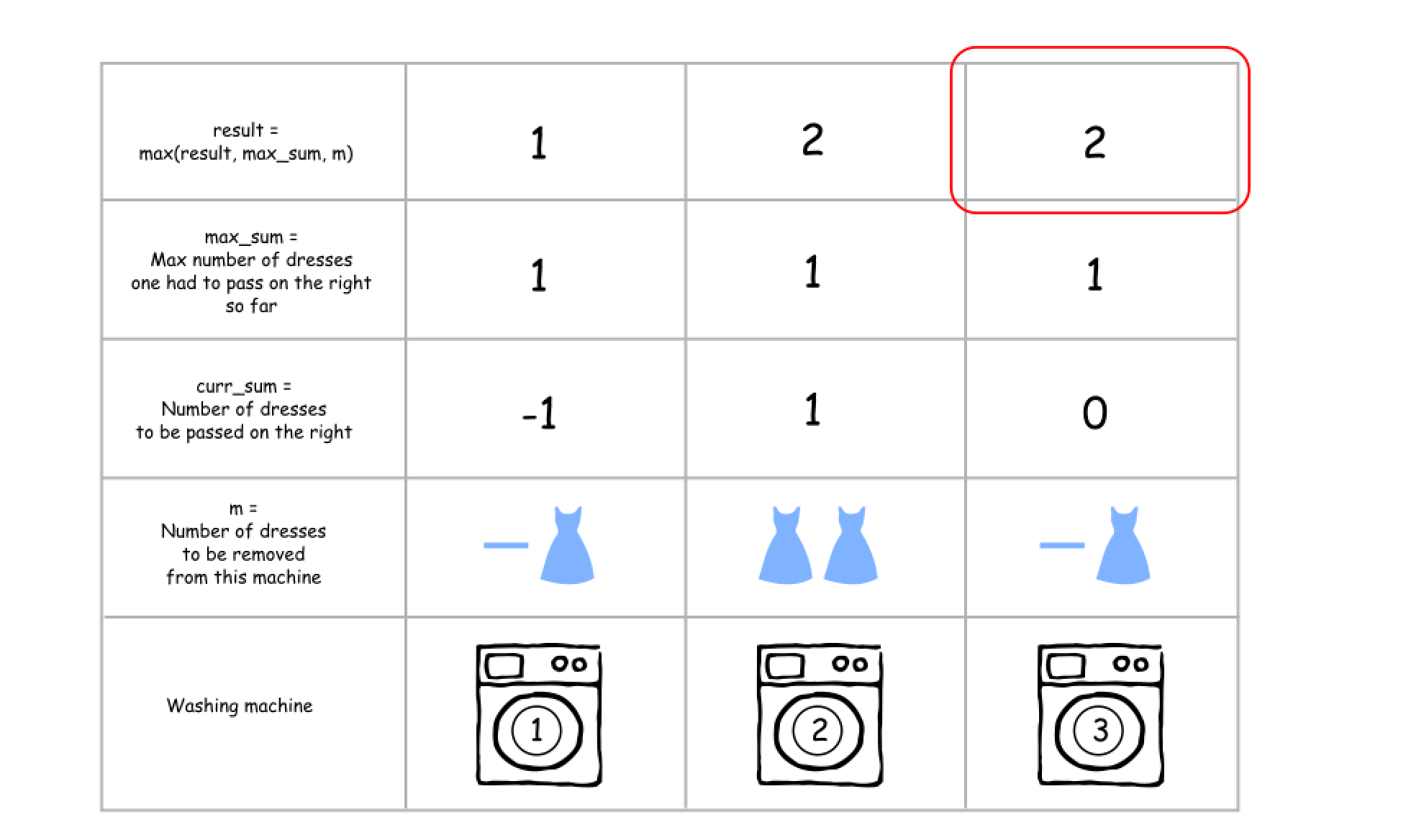
It's quite obvious that the result at each point is a maximum between max\_sum and m, i.e. one has to compare the cumulative and the local maximums.

Here are three different examples.

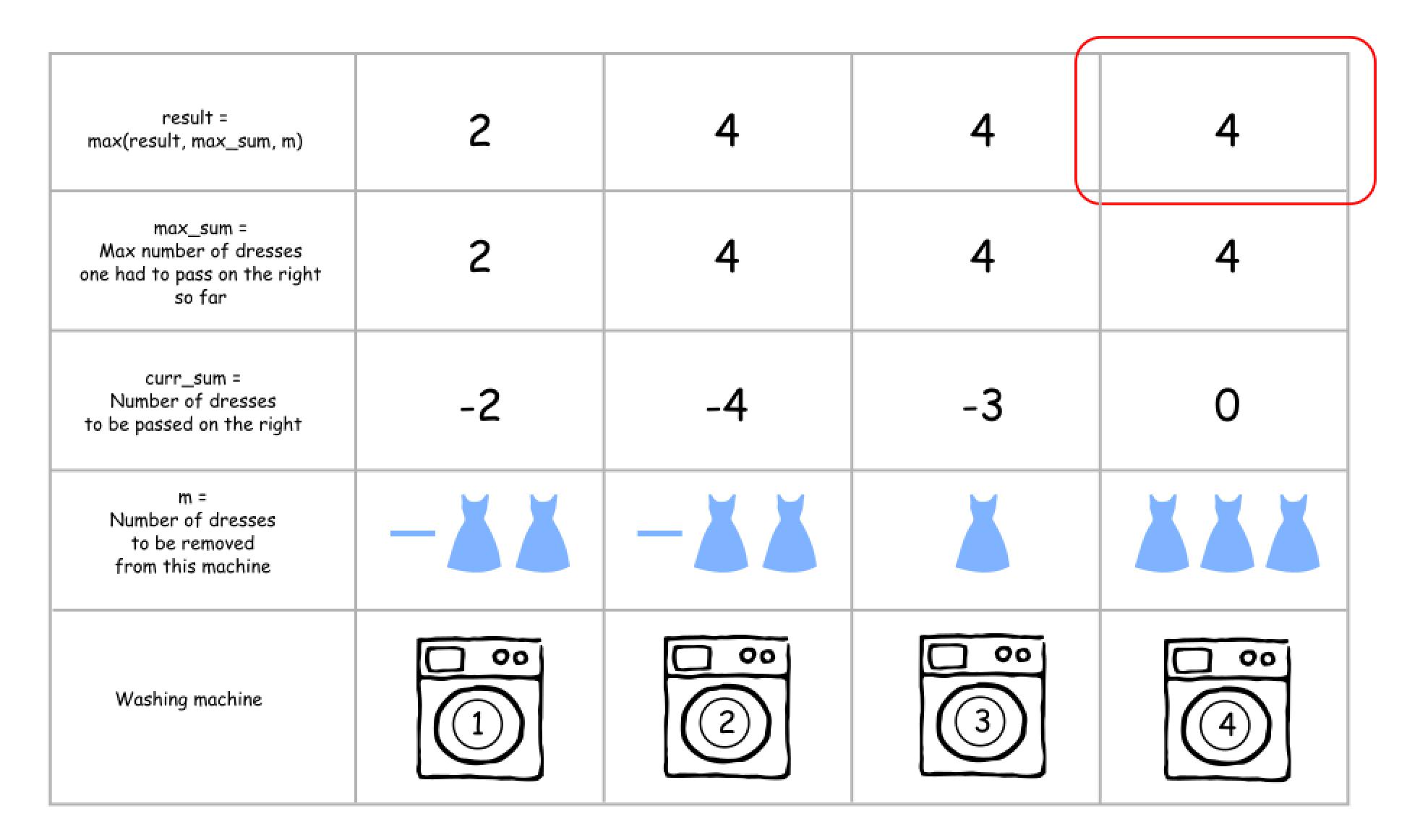
* [1, 0, 5]. The cumulative maximum is equal to the local one.



* [0, 3, 0]. The local maximum wins over the cumulative one.



* [0, 0, 3, 5]. The cumulative maximum wins over the local one.



**Algorithm**

Here is the algorithm.

1. Check if the problem could be solved: len(machines) should be a divisor of sum(machines). Otherwise the answer is -1.
2. Compute the number of dresses each machine should finally have: dresses\_per\_machine = sum(machines)/len(machines).
3. [Normalize](https://en.wikipedia.org/wiki/Normalization#Technology_and_computer_science) the problem by replacing the *number of dresses* in each machine by the *number of dresses to be removed* from this machine (could be negative).
4. Initiate curr\_sum, max\_sum, and res as zero.
5. Iterate over all machines m in machines:
   * Update curr\_sum and max\_sum at each step: curr\_sum += m, max\_sum = max(max\_sum, abs(curr\_sum)).
   * Update result res = max(res, max\_sum, m).
6. Return res.

**Implementation**

|  |
| --- |
| class Solution {  public int findMinMoves(int[] machines) {  int n = machines.length, dressTotal = 0;  for (int m : machines) dressTotal += m;  if (dressTotal % n != 0) return -1;  int dressPerMachine = dressTotal / n;  // Change the number of dresses in the machines to  // the number of dresses to be removed from this machine  // (could be negative)  for (int i = 0; i < n; i++) machines[i] -= dressPerMachine;  // currSum is a number of dresses to move at this point,  // maxSum is a max number of dresses to move at this point or before,  // m is number of dresses to move out from the current machine.  int currSum = 0, maxSum = 0, tmpRes = 0, res = 0;  for (int m : machines) {  currSum += m;  maxSum = Math.max(maxSum, Math.abs(currSum));  tmpRes = Math.max(maxSum, m);  res = Math.max(res, tmpRes);  }  return res;  }  } |

**Complexity Analysis**

* Time complexity : O(*N*) since it's a three iterations over the input array.
* Space complexity : O(1) since it's a constant space solution.

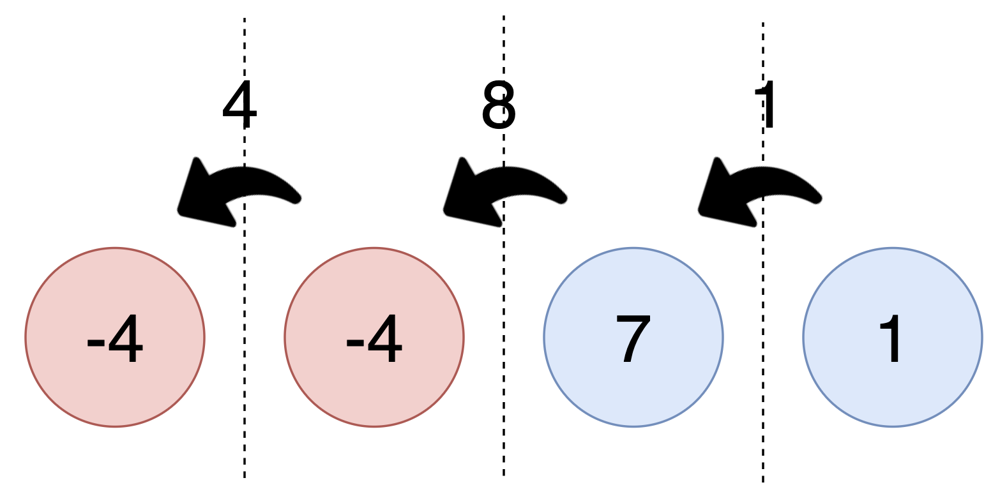
<https://leetcode.com/problems/super-washing-machines/discuss/654317/Explanation-(proof)-of-why-the-solution-works>

Although the official solution is indeed correct, it lacks reasonable explanation on the solution page and doesn't show convictively why it works. Almost all the posts in discussion only give the code implementation and/or mere explanation of **how** the code processes the input rather than **why**. There are only 2 posts that point out it: [Formal-proof-of-the-solution](https://leetcode.com/problems/super-washing-machines/discuss/591578/Formal-proof-of-the-solution), and [Follow-up:-What-if-the-question-require-detailed-operation-for-each-step](https://leetcode.com/problems/super-washing-machines/discuss/216790/Follow-up%3A-What-if-the-question-require-detailed-operation-for-each-step). (Unfortunately, the proof in the former post is incorrect.)

The solution essentially only tells the lower bound of the total number of moves, without touching on the reachability of this lower bound at all. One could ask a follow-up question like 'How do you go about a detailed movement procedure that costs the minimum possible number of moves?'. This post is going to give you the guarantee that this lower bound is in fact reachable for any input.

Let's make some assumptions first to ignore the trivial cases:

* Assume *n* >= 2, and *n* is a divisor of the total number of dresses (i.e. answer is not -1);
* Assume the list machines is already normalized by deducting avg to have an average of zero;
* Assume the values in machines are not all zeros (i.e. there exist at least one positive and one negative in machines).

There are totally *n*-1 demarcation lines that can split all the machines into two groups - group on the left and group on the right. We need a variable to denote these demarcation lines. Let D be a list of length *n*-1. D[i](i=0, ..., n-2) represents the line that stands between machines[i] and machines[i+1] thus split all machines into two groups: machines[0], ..., machines[i] on the left, and machines[i+1], ..., machines[n-1] on the right hand. The value stored in D[i] stands for the number of essential transfer(s) that must take place across this line. If the transfer(s) need to be from left to right(i.e. from D[i] to D[i+1]), we make D[i] be positve; otherwise if the transfer(s) need to be from right to left(i.e. from D[i+1] to D[i]) we make D[i] be negative. For example, for machines = [-4, -4, 7, 1], D = [-4, -8, -1]; for machines = [3, -2, -1], D= [3, 1].  


It is easy to see that max{abs(D)} is the lower bound of the final answer. However this lower bound is not tight enough. For example, for machines = [-2, 5, -3], D = [-2, 3]. The minimum number of moves for this input is actually 5 rather than 3. The reason is that each machine can only pass at most one dress at each step.

Let M = max{abs(D), machines} = max{max{abs(D)}, max{machines}}. We claim that M is the minimum number of moves, which is exactly what the official solution claims. The following part of the article is going to prove that the lower bound M is actually reachable for any valid input list machines.

We will conduct the proof by constructing a specific movement policy and showing that after each time step the updated M corresponding to the resulting machines list is one less than the former one.

One more notation: let L[i] denote the number of dresses that machine *i* need to pass to its left neighbor (can be negetive); and R[i] denote the number of dresses that machine *i* need to pass to its right neighbor. It is obvious that L[i] = -D[i-1] (i=1, ..., n-1); and R[i] = D[i] (i=0, ..., n-2).

**We design the movement policy as below:**

|  |
| --- |
| # At each time step, do the following:  machines\_updated = machines.copy()  for i in range(n):  if machines[i] >= 0:  if i > 0 and L[i] > 0:  # move one dress from machine i to machine i-1  machines\_updated[i] -= 1  machines\_updated[i-1] += 1  elif i < n-1 and R[i] > 0:  # move one dress from machine i to machine i+1  machines\_updated[i] -= 1  machines\_updated[i+1] += 1  machines = machines\_updated # used for the subsequent time step |

Although the above procedure is written as a *for* loop, it represents all operations that we do at only one single time step. (i.e. all the operations inside the *for* loop can be performed simultaneously.)

Another thing to note is that we won't come across the situation where there is not enough dress in the current machine to pass to its neighbor, because machines[i] >= 0 means machine i has at least avg dresses left, and for each machine we will move at most one dress out of it.

Then we provide several lemmas which are helpful for the final proof.

***Lemma 1.*** After each time step, for every 0 <= *i* <= n-2, the D\_updated[i] (corresponding to machines\_updated) is no more than its previous value D[i]. More precisely, D\_updated[i] is either D[i] or D[i]-1.  
(The proof of Lemma 1 is left to readers, which is fairly easy. Hint: for each demarcation line, there is at most one dress passed across this line at each time step.)

***Lemma 2.*** If machines[i] > 0, at least one of L[i] and R[i] is positive.

Following *Lemma 2*, it is easy to derive *lemma 3*:  
***Lemma 3.*** For any machines[i] >= 0(i.e. if old value no less than avg), machines\_updated[i] <= machines[i], more precisely, machines\_updated[i] is either machines[i] or machines[i]-1.

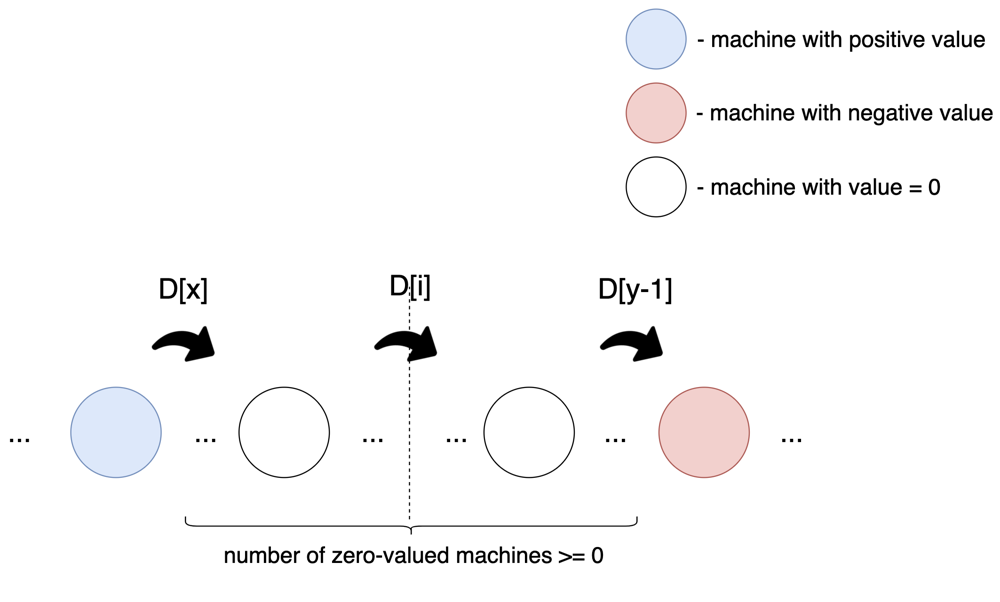
The above lemmas ensure that by following the above procedure, after each time step, M\_updated = max{max{abs(D\_updated)}, max{machines\_updated}} is no more than its previous value M (M\_updated is either M or M - 1). Then we are going to show that M\_updated is definitely equal to M - 1.

***Lemma 4*** If there exists a machine i s.t. machines[i] = M, then both L[i] and R[i] is non-negative, and at least one of L[i] and R[i] is positive.  
(*Lemma 4* can be easily proven by contradiction)

Directly following *Lemma 4*, we get the following corollary:  
***Corollary 1*** After each time step, the value max{machines} decreases exactly by one (until reaches zero).

***Lemma 5*** If abs(D[i]) = max{abs(D)}, let integer 0 <= x <= i, and integer i+1 <= y <= n-1, and machines[x] denote the nearest non-zero value to the lefthand of the demarcation line D[i] (i.e. x is the largest integer <= i s.t. machines[x] != 0), machines[y] denote the nearest non-zero value to the righthand of D[i] (i.e. y is the smallest integer >= i+1 s.t. machines[y] != 0), then machines[x] and machines[y] must be a combination of a positive and a negative (i.e. either machines[x] > 0 and machines[y] < 0, or, machines[x] < 0 and machines[y] > 0. Also, one could figure out that 'machines[x] > 0 and machines[y] < 0' corresponds to D[i] > 0; and that 'machines[x] < 0 and machines[y] > 0' corresponds to D[i] < 0).  
(*Lemma 5* is easy to prove by contradiction: assume machines[x] and machines[y] are both positive/negative...)

*Lemma 5* basically says that the demarcation line with the largest absolute transfer value lies between a positive-valued machine (# dresses > avg) and a negative-valued machine (# dresses < avg). There might be some zero-valued machines lying in between, but we will see that it won't affect our analysis.

**Now let's analyze any *i* s.t. abs(D[i]) = max{abs(D)}**. Without loss of generality, let's assume machines[x] > 0 and machines[y] < 0, as the below picture demonstrates.  
  
Obviously, for any integer j s.t. x <= j <= y-1. D[x] = D[i] = D[y-1] = D[j].  
Let's divide the problem into different cases:  
**i.** L[x] <= 0.  
That means machines[x] <= abs(D[i]) = max(abs(D)). In this case, we only care about D[i]. According to the earlier mentioned movement policy, after one time step D[i] will decrease by one, i.e. D\_updated[i] = D[i] - 1.

**ii.** L[x] > 0.  
That means machines[x] > abs(D[i]) = max(abs(D)). In such case, we don't care about D[i] at all - whether it decreases or not does not affect the update for M.

Together with *Corollary 1*, we conclude that M will decrease by one for each time step (until it reaches zero) following our movement policy. And it is easy to recognize that if the current list machines is valid, the resulting updated machines list is also valid. Therefore, the minimum number of moves to balance all dresses is M = max{max{abs(D)}, max{machines}}.

**Gas Station**

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:**

* gas.length == n
* cost.length == n
* 1 <= n <= 104
* 0 <= gas[i], cost[i] <= 104

Solution

Approach 1: One pass.

**Intuition**

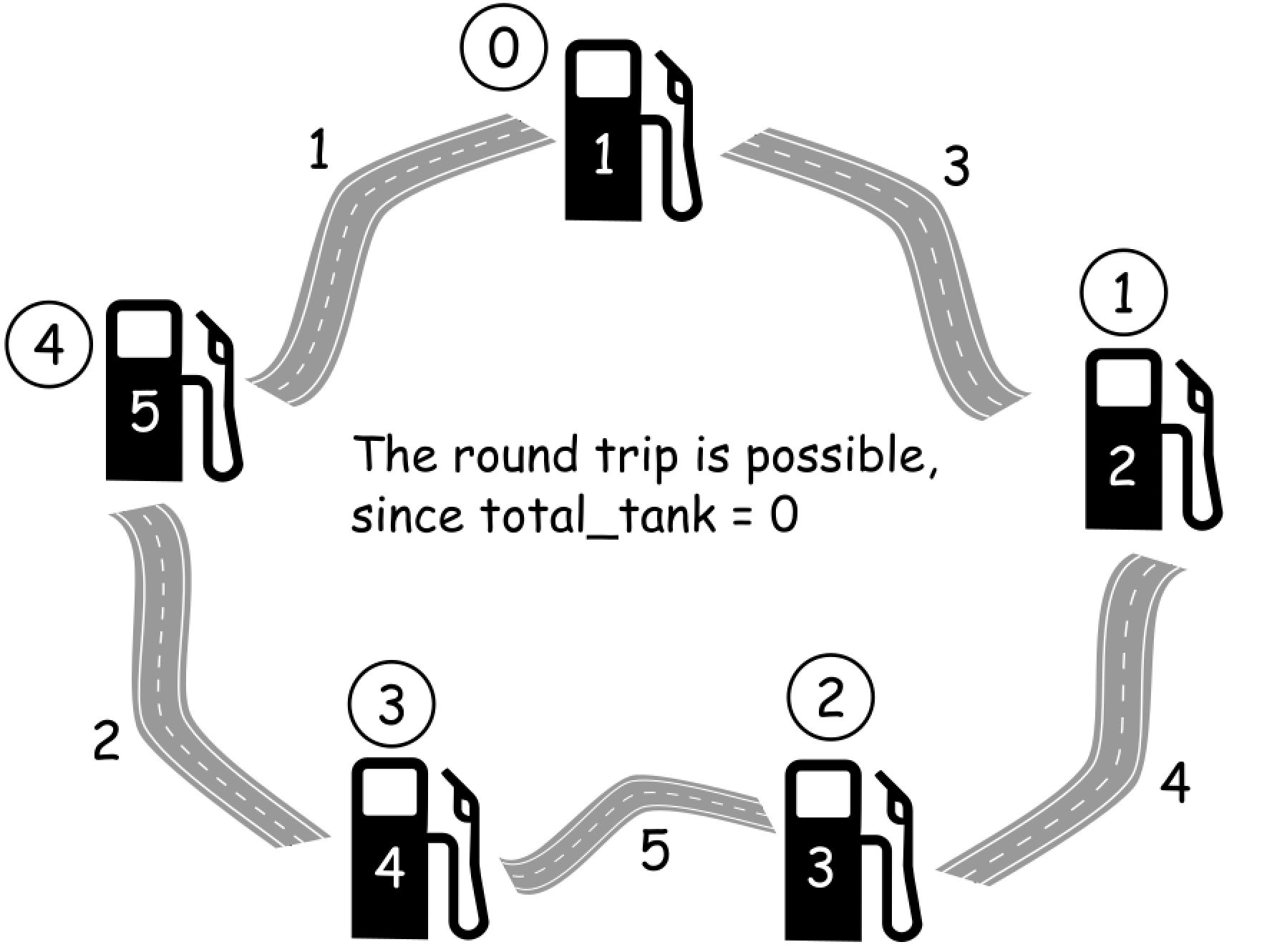
The first idea is to check every single station :

* Choose the station as starting point.
* Perform the road trip and check how much gas we have in tank at each station.

That means \mathcal{O}(N^2)O(*N*2) time complexity, and for sure one could do better.

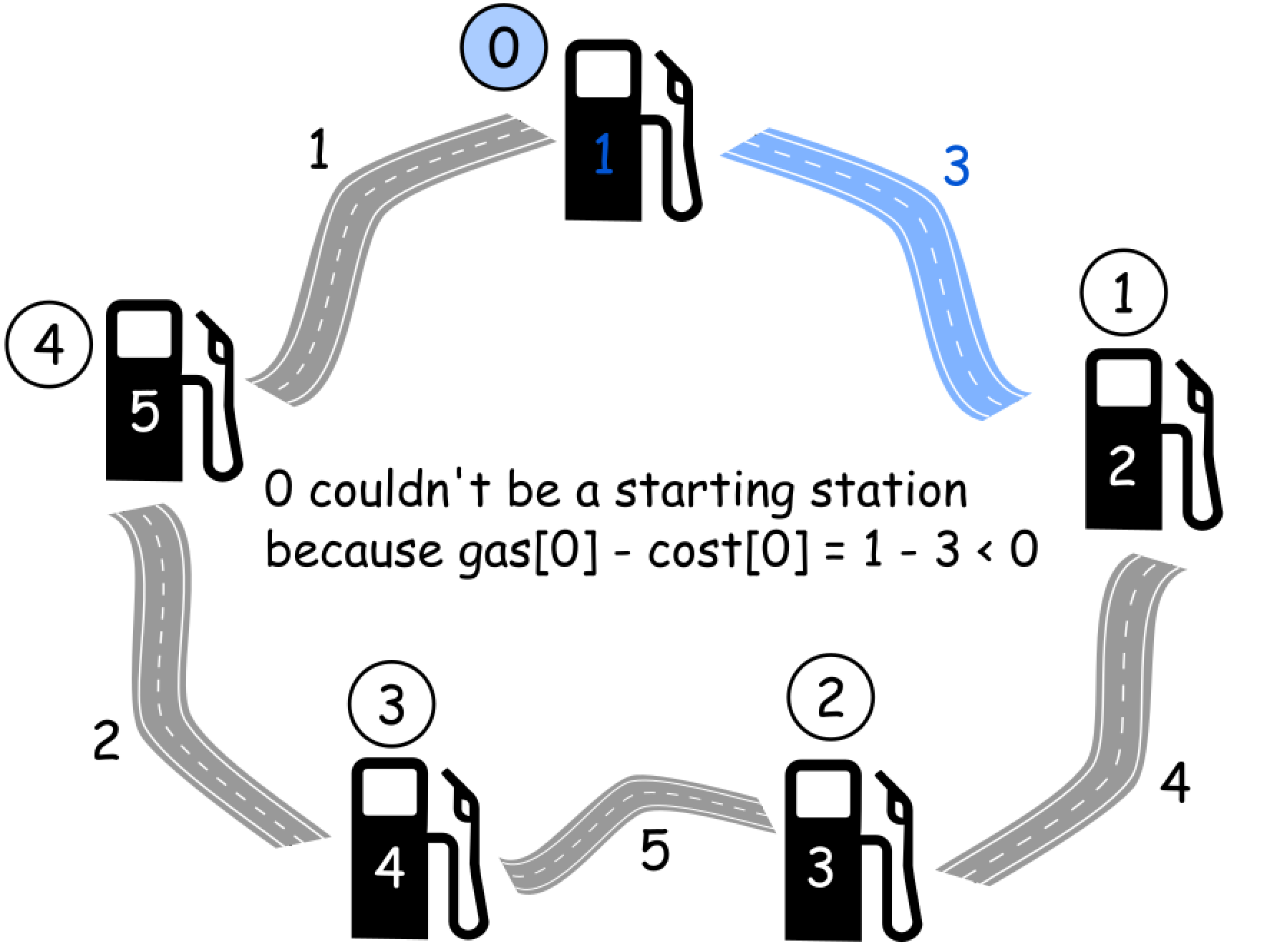
Let's notice two things.

It's impossible to perform the road trip if sum(gas) < sum(cost). In this situation the answer is -1.



One could compute total amount of gas in the tank total\_tank = sum(gas) - sum(cost) during the round trip, and then return -1 if total\_tank < 0.

It's impossible to start at a station i if gas[i] - cost[i] < 0, because then there is not enough gas in the tank to travel to i + 1 station.



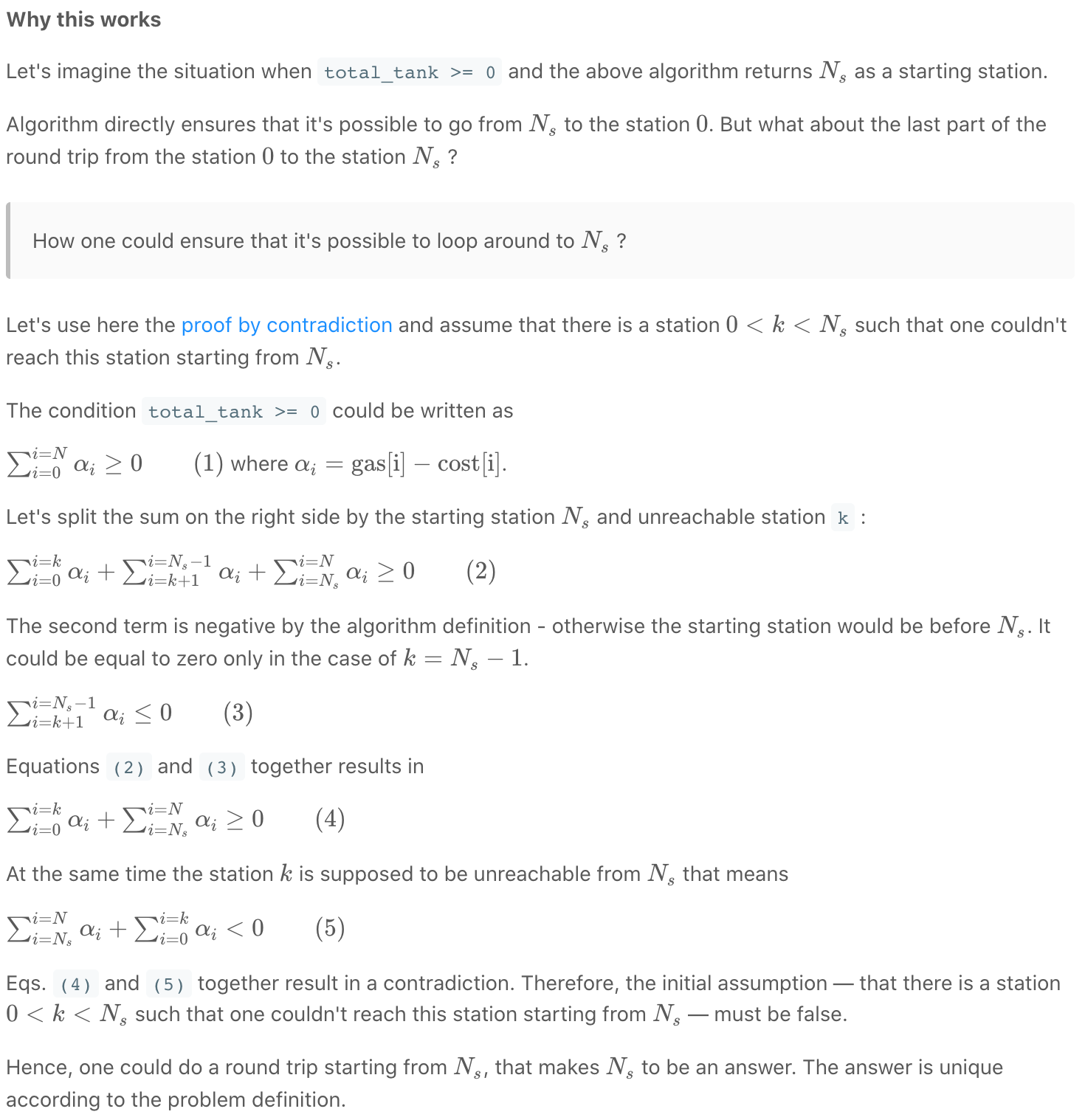
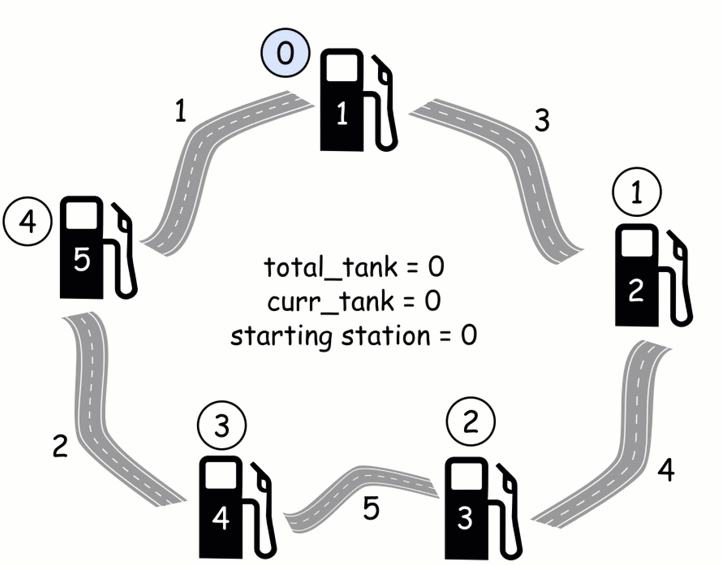
The second fact could be generalized. Let's introduce curr\_tank variable to track the current amount of gas in the tank. If at some station curr\_tank is less than 0, that means that one couldn't reach this station.

Next step is to mark this station as a new starting point, and reset curr\_tank to zero since one starts with no gas in the tank.

**Algorithm**

Now the algorithm is straightforward :

1. Initiate total\_tank and curr\_tank as zero, and choose station 0 as a starting station.
2. Iterate over all stations :
   * Update total\_tank and curr\_tank at each step, by adding gas[i] and subtracting cost[i].
   * If curr\_tank < 0 at i + 1 station, make i + 1 station a new starting point and reset curr\_tank = 0 to start with an empty tank.
3. Return -1 if total\_tank < 0 and starting station otherwise.

****

|  |
| --- |
| class Solution {  public int canCompleteCircuit(int[] gas, int[] cost) {  int n = gas.length;  int total\_tank = 0;  int curr\_tank = 0;  int starting\_station = 0;  for (int i = 0; i < n; ++i) {  total\_tank += gas[i] - cost[i];  curr\_tank += gas[i] - cost[i];  // If one couldn't get here,  if (curr\_tank < 0) {  // Pick up the next station as the starting one.  starting\_station = i + 1;  // Start with an empty tank.  curr\_tank = 0;  }  }  return total\_tank >= 0 ? starting\_station : -1;  }  } |

**Complexity Analysis**

* Time complexity : O(*N*) since there is only one loop over all stations here.
* Space complexity : O(1) since it's a constant space solution.

**Further reading**

There are numerous variations of gas problem, here are some examples :

[Find the cheapest path between two stations if at most Δ stops are allowed.](https://www.sciencedirect.com/science/article/pii/S002001901730203X)

[Find the cheapest path between two stations if the vehicle has a given tank capacity.](https://link.springer.com/chapter/10.1007/978-3-540-75520-3_48)

**Best Time to Buy and Sell Stock**

You are given an array prices where prices[i] is the price of a given stock on the ith day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return 0.

**Example 1:**

**Input:** prices = [7,1,5,3,6,4]

**Output:** 5

**Explanation:** Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.

Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

**Example 2:**

**Input:** prices = [7,6,4,3,1]

**Output:** 0

**Explanation:** In this case, no transactions are done and the max profit = 0.

**Constraints:**

* 1 <= prices.length <= 105
* 0 <= prices[i] <= 104

Solution

We need to find out the maximum difference (which will be the maximum profit) between two numbers in the given array. Also, the second number (selling price) must be larger than the first one (buying price).

In formal terms, we need to find  max(prices[j]−prices[i]), for every *i* and *j* such that *j*>*i*.

Approach 1: Brute Force

|  |
| --- |
| public class Solution {  public int maxProfit(int prices[]) {  int maxprofit = 0;  for (int i = 0; i < prices.length - 1; i++) {  for (int j = i + 1; j < prices.length; j++) {  int profit = prices[j] - prices[i];  if (profit > maxprofit)  maxprofit = profit;  }  }  return maxprofit;  }  } |

**Complexity Analysis**

* Time complexity : O(n^2). Loop runs n (n-1)/2​ times.
* Space complexity : *O*(1). Only two variables - maxprofit and profit are used.

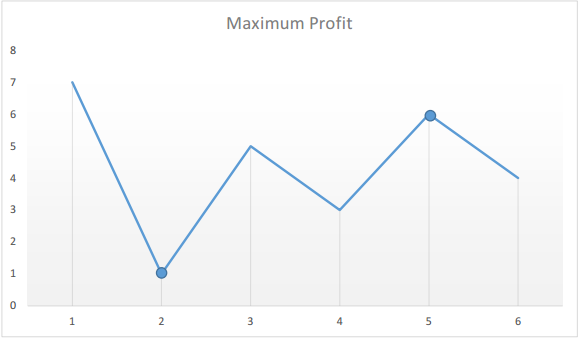
#### Approach 2: One Pass

**Algorithm**

Say the given array is:

[7, 1, 5, 3, 6, 4]

If we plot the numbers of the given array on a graph, we get:



The points of interest are the peaks and valleys in the given graph. We need to find the largest peak following the smallest valley. We can maintain two variables - minprice and maxprofit corresponding to the smallest valley and maximum profit (maximum difference between selling price and minprice) obtained so far respectively.

|  |
| --- |
| public class Solution {  public int maxProfit(int prices[]) {  int minprice = Integer.MAX\_VALUE;  int maxprofit = 0;  for (int i = 0; i < prices.length; i++) {  if (prices[i] < minprice)  minprice = prices[i];  else if (prices[i] - minprice > maxprofit)  maxprofit = prices[i] - minprice;  }  return maxprofit;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*). Only a single pass is needed.
* Space complexity : *O*(1). Only two variables are used.

**Best Time to Buy and Sell Stock II**

You are given an array prices for which the ith element is the price of a given stock on day i.

Find the maximum profit you can achieve. You may complete as many transactions as you like (i.e., buy one and sell one share of the stock multiple times).

**Note:** You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

**Example 1:**

**Input:** prices = [7,1,5,3,6,4]

**Output:** 7

**Explanation:** Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 = 4.

Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3.

**Example 2:**

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

**Output:** 4

**Explanation:** Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.

Note that you cannot buy on day 1, buy on day 2 and sell them later, as you are engaging multiple transactions at the same time. You must sell before buying again.

**Example 3:**

**Input:** prices = [7,6,4,3,1]

**Output:** 0

**Explanation:** In this case, no transaction is done, i.e., max profit = 0.

**Constraints:**

* 1 <= prices.length <= 3 \* 104
* 0 <= prices[i] <= 104

Summary

We have to determine the maximum profit that can be obtained by making the transactions (no limit on the number of transactions done). For this we need to find out those sets of buying and selling prices which together lead to the maximization of profit.

Solution

Approach 1: Brute Force

In this case, we simply calculate the profit corresponding to all the possible sets of transactions and find out the maximum profit out of them.

|  |
| --- |
| class Solution {  public int maxProfit(int[] prices) {  return calculate(prices, 0);  }  public int calculate(int prices[], int s) {  if (s >= prices.length)  return 0;  int max = 0;  for (int start = s; start < prices.length; start++) {  int maxprofit = 0;  for (int i = start + 1; i < prices.length; i++) {  if (prices[start] < prices[i]) {  int profit = calculate(prices, i + 1) + prices[i] - prices[start];  if (profit > maxprofit)  maxprofit = profit;  }  }  if (maxprofit > max)  max = maxprofit;  }  return max;  }  } |

**Complexity Analysis**

* Time complexity : O(n^n). Recursive function is called n^n times.
* Space complexity : O(n). Depth of recursion is n.

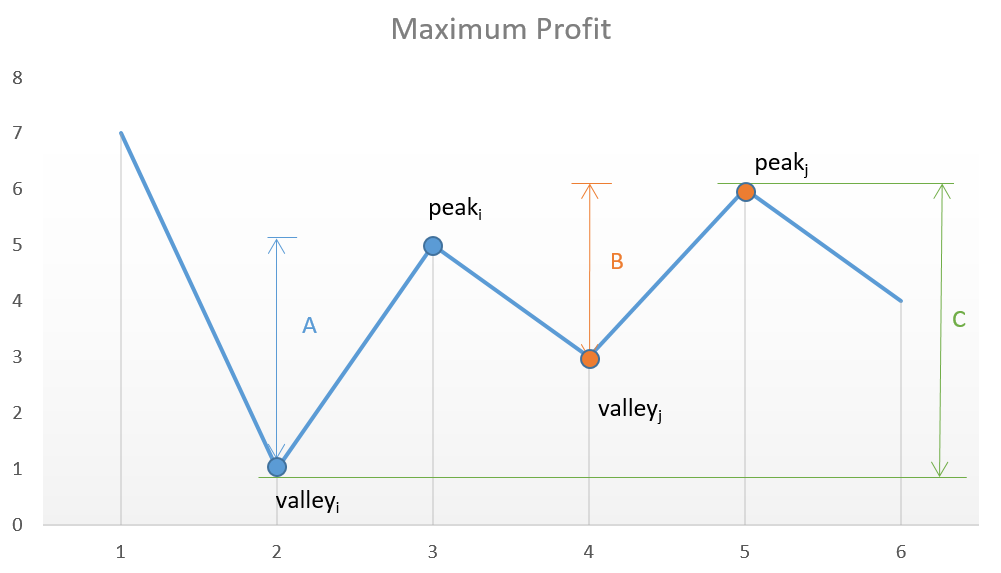
#### Approach 2: Peak Valley Approach

**Algorithm**

Say the given array is:

[7, 1, 5, 3, 6, 4].

If we plot the numbers of the given array on a graph, we get:



If we analyze the graph, we notice that the points of interest are the consecutive valleys and peaks.

Mathematically speaking: 

The key point is we need to consider every peak immediately following a valley to maximize the profit. In case we skip one of the peaks (trying to obtain more profit), we will end up losing the profit over one of the transactions leading to an overall lesser profit.

For example, in the above case, if we skip *peaki*​ and *valleyj*​ trying to obtain more profit by considering points with more difference in heights, the net profit obtained will always be lesser than the one obtained by including them, since *C* will always be lesser than *A*+*B*.

|  |
| --- |
| class Solution {  public int maxProfit(int[] prices) {  int i = 0;  int valley = prices[0];  int peak = prices[0];  int maxprofit = 0;  while (i < prices.length - 1) {  while (i < prices.length - 1 && prices[i] >= prices[i + 1])  i++;  valley = prices[i];  while (i < prices.length - 1 && prices[i] <= prices[i + 1])  i++;  peak = prices[i];  maxprofit += peak - valley;  }  return maxprofit;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*). Single pass.
* Space complexity : *O*(1). Constant space required.

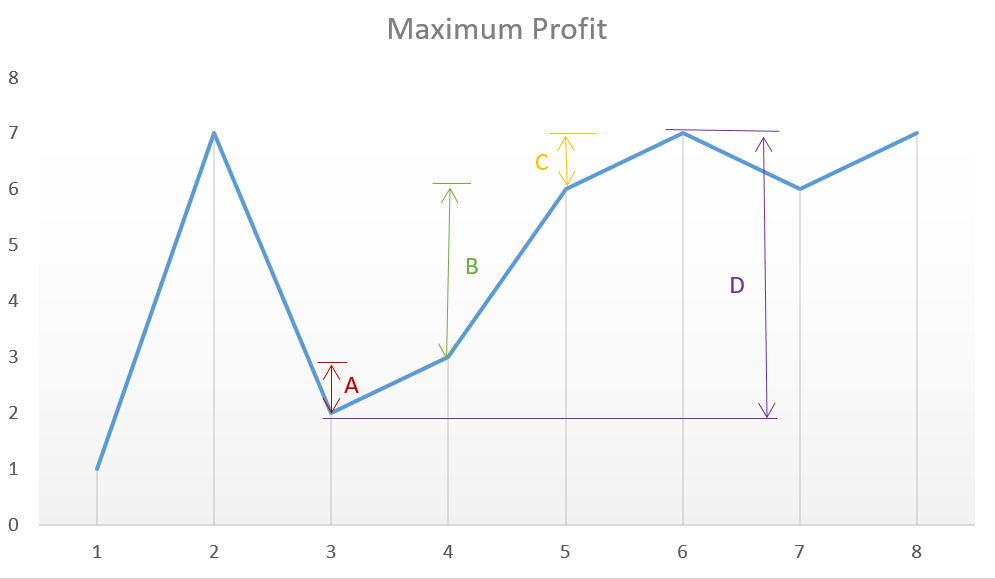
Approach 3: Simple One Pass

**Algorithm**

This solution follows the logic used in [Approach 2](https://leetcode.com/problems/best-time-to-buy-and-sell-stock-ii/solution/#approach-2-peak-valley-approach) itself, but with only a slight variation. In this case, instead of looking for every peak following a valley, we can simply go on crawling over the slope and keep on adding the profit obtained from every consecutive transaction. In the end,we will be using the peaks and valleys effectively, but we need not track the costs corresponding to the peaks and valleys along with the maximum profit, but we can directly keep on adding the difference between the consecutive numbers of the array if the second number is larger than the first one, and at the total sum we obtain will be the maximum profit. This approach will simplify the solution. This can be made clearer by taking this example:

[1, 7, 2, 3, 6, 7, 6, 7]

The graph corresponding to this array is:



From the above graph, we can observe that the sum *A*+*B*+*C* is equal to the difference *D* corresponding to the difference between the heights of the consecutive peak and valley.

|  |
| --- |
| class Solution {  public int maxProfit(int[] prices) {  int maxprofit = 0;  for (int i = 1; i < prices.length; i++) {  if (prices[i] > prices[i - 1])  maxprofit += prices[i] - prices[i - 1];  }  return maxprofit;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*). Single pass.
* Space complexity: *O*(1). Constant space needed.

**Best Time to Buy and Sell Stock III**

Say you have an array for which the *i*th element is the price of a given stock on day *i*.

Design an algorithm to find the maximum profit. You may complete at most *two* transactions.

**Note:**You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).

**Example 1:**

**Input:** prices = [3,3,5,0,0,3,1,4]

**Output:** 6

**Explanation:** Buy on day 4 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.

Then buy on day 7 (price = 1) and sell on day 8 (price = 4), profit = 4-1 = 3.

**Example 2:**

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

**Output:** 4

**Explanation:** Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.

Note that you cannot buy on day 1, buy on day 2 and sell them later, as you are engaging multiple transactions at the same time. You must sell before buying again.

**Example 3:**

**Input:** prices = [7,6,4,3,1]

**Output:** 0

**Explanation:** In this case, no transaction is done, i.e. max profit = 0.

**Example 4:**

**Input:** prices = [1]

**Output:** 0

**Constraints:**

* 1 <= prices.length <= 105
* 0 <= prices[i] <= 105

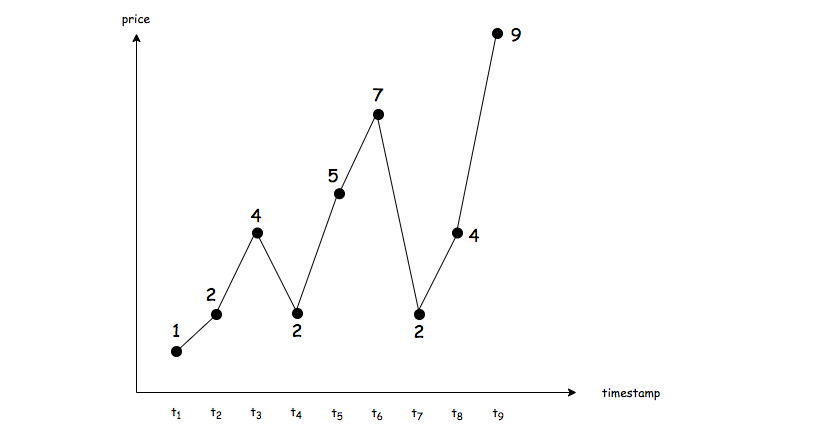
## Solution

### **Overview**

First of all, as one should know, this is one of the problems from the series of [Best Time to Buy and Sell Stock](https://leetcode.com/problems/best-time-to-buy-and-sell-stock/) problem. One could start from the first problem in the series and progress one by one from easy to hard.

If there is ever a God of the stock market, who knows the price of stock at any moment, then the strategies to gain the maximum profits from the stock market is actually surprisingly intuitive, which also depends on the number of transactions that one can make.

If one can only make one transaction (i.e. buy and sell once), then better to make this one bet count. The best strategy would be to buy at the **lowest** price and then sell at the **highest** price. To put it simple, buy low sell high.



Let us look at a concrete example as shown in the above graph, given a list of prices, the task becomes to find maximal difference between a **latter** stock price and an **earlier** one, which would be the maximal profits that we could gain, if only one transaction is allowed.

In the above example, the best moment to buy the stock would be the timestamp t1, and the best moment to sell the stock would be the timestamp t9.

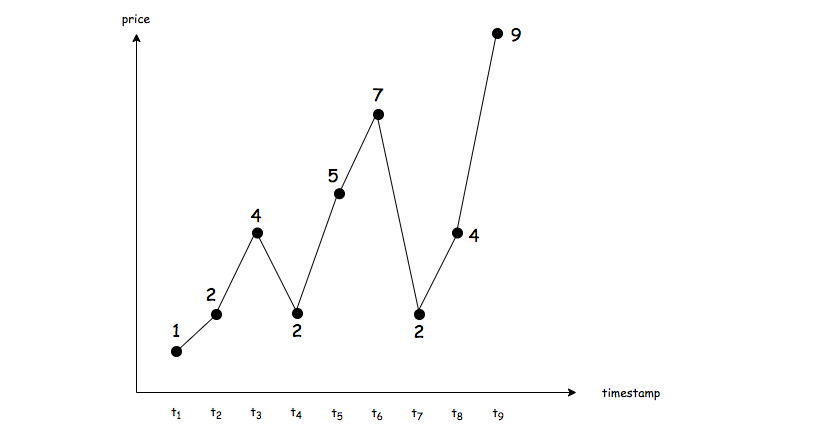
The above strategy is actually the solution to the first problem of the series, i.e. [Best Time to Buy and Sell Stock](https://leetcode.com/problems/best-time-to-buy-and-sell-stock/).

On the other hand, if one can make as many transactions as one would like, then in order to gain the maximal profits, one must capture each **augmentation** and avoid each **plunging** of stock price.

Specifically, given a list of prices, for any two adjacent time points with stock prices p1 and p2, the above best strategies can be broke down into the following two cases:

* If later the price augments, i.e. p2 > p1, then a good trader should buy at p1 and then sell at p2, seizing this moment to make profits.
* If later the price stays the same or even plunges, i.e. p2 <= p1, then a good trader should just hold the money in the pocket, neither to buy nor sell any stock.

With the above strategies, as one can see, we would perfectly capitalize at each right moment, meanwhile avoiding any loss. At the end, the accumulative profits that we gain over the time would reach the maximum.



With the same example above, we would buy at the moment of t1 and sell it at the moment of t2. Similarly, we would also buy at the moment of t2 and sell the moment of t3. As one might notice, the profits we gain from these two transactions are equivalent to the single transaction of buying at the moment of t1 and selling at the moment of t3.

The above strategy would be the solution for the second problem of series, i.e. [Best Time to Buy and Sell Stock II](https://leetcode.com/problems/best-time-to-buy-and-sell-stock-ii/) where there is no limit on the number of transactions.

#### Approach 1: Bidirectional Dynamic Programming

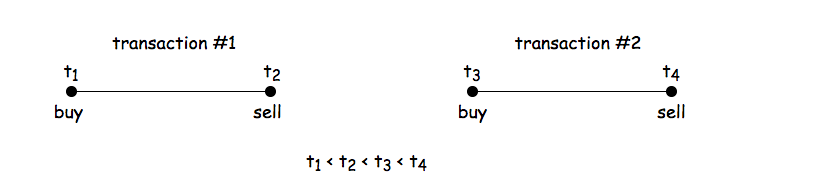
**Intuition**

The only difference between this problem and the previous two problems is that in this problem we are allowed to make **at most two** transactions.

Additionally, there is a constraint on the order of transactions stated in the problem description as follows:

You may not engage in multiple transactions at the same time, (i.e. you must sell the stock before you buy again).

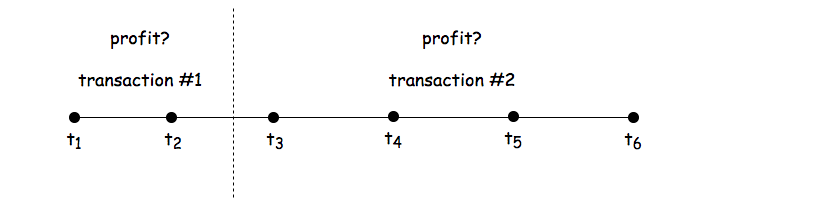
We could interpret this constraint as that there would be **no overlapping** in the sequence of transactions.



In other words, the two transactions that we should make would situate in two different subsequences of the stock prices, without any overlapping, which we illustrate in the above graph.

That being said, we can solve the problem in a **divide-and-conquer** manner, where we divide the original sequence of prices into two subsequences and then we calculate the maximum profit that we could gain from making a single transaction in each subsequence.

The total profits would be the sum of profits from each subsequence. If we enumerate all possible divisions (or we could consider them as combinations of subsequences), we could find the maximum total profits among them, which is also the desired result of the problem.



So we divide this problem into two subproblems, and each subproblem is actually of the same problem of [Best Time to Buy and Sell Stock](https://leetcode.com/problems/best-time-to-buy-and-sell-stock/) as we discussed in the overview section.

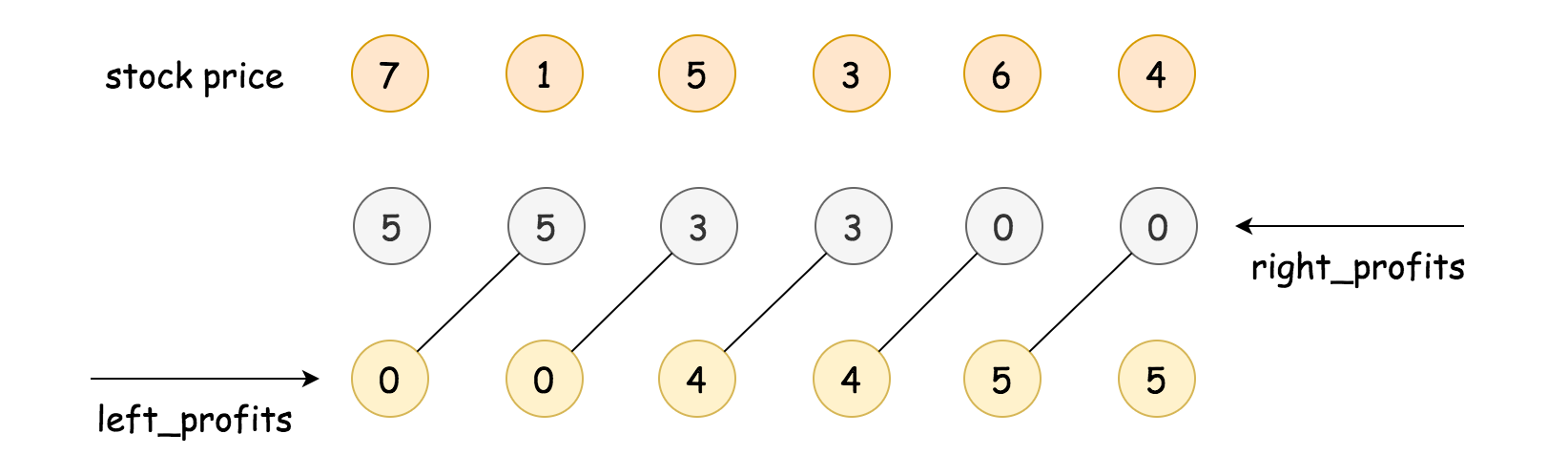
**Algorithm**

A naive implementation of the above idea would be to divide the sequences into two and then enumerate each of the subsequences, though this is definitely not the most optimized solution.

For a sequence of length N*N*, we would have N*N* possible divisions (including no division), each of the elements would be visited once in each division. As a result, the overall time complexity of this naive implementation would be O(N^2).

We could do better than the naive O(N^2) implementation. Regarding the algorithms of divide-and-conquer, one common technique that we can apply in order to optimize the time complexity is called **dynamic programming** (DP) where we trade less repetitive calculation with some extra space.

In dynamic programming algorithms, normally we create an array of one or two dimensions to keep the intermediate optimal results. In this problem though, we would use two arrays, with one array keeping the results of sequence from left to right and the other array keeping the results of sequence from right to left. For the sake of name, we could call it **bidirectional dynamic programming**.



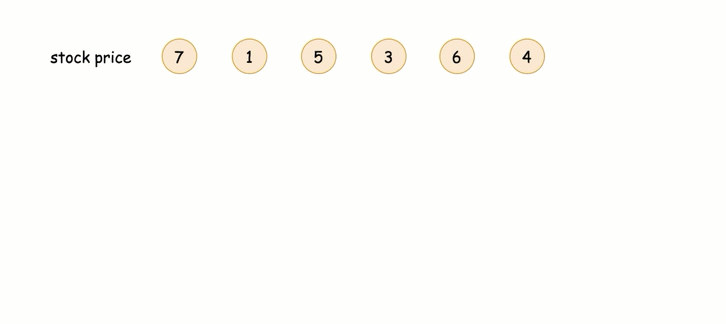
First, we denote a sequence of prices as Prices[i], with index starting from 0 to N-1. Then we define two arrays, namely left\_profits[i] and right\_profits[i].

* As suggested by the name, each element in the left\_profits[i] array would hold the maximum profits that one can gain from doing one single transaction on the left subsequence of prices from the index zero to i, (i.e. Prices[0], Prices[1], ... Prices[i]). For instance, for the subsequences of [7, 1, 5], the corresponding left\_profits[2] would be 4, which is to buy the price of 1 and sell it at the price of 5.
* And each element in the right\_profits[i] array would hold the maximum profits that one can gain from doing one single transaction on the right subsequence of the prices from the index i up to N-1, (i.e. Prices[i], Prices[i+1], ... Prices[N-1]). For example, for the right subsequence of [3, 6, 4], the corresponding right\_profits[3] would be 3, which is to buy at the price of 3 and then sell it at the price of 6.

Now, if we divide the sequence of prices around the element at the index i, into two subsequences, with left subsequences as Prices[0], Prices[1], ... Prices[i] and the right subsequence as Prices[i+1], ... Prices[N-1], then the total maximum profits that we obtain from this division (denoted as max\_profits[i]) can be expressed as follows:  

Then if we exhaust all possible divisions, i.e. we place the two transactions in all possible combinations of subsequences, we would then obtain the global maximum profits that we could gain from this sequence of stock prices, which can be expressed as follows: 

We demonstrate how the DP arrays are calculated in the following animation.



Following the above idea, Here are some sample implementations.

|  |
| --- |
| class Solution {  public int maxProfit(int[] prices) {  int length = prices.length;  if (length <= 1) return 0;  int leftMin = prices[0];  int rightMax = prices[length - 1];  int[] leftProfits = new int[length];  // pad the right DP array with an additional zero for convenience.  int[] rightProfits = new int[length + 1];  // construct the bidirectional DP array  for (int l = 1; l < length; ++l) {  leftProfits[l] = Math.max(leftProfits[l - 1], prices[l] - leftMin);  leftMin = Math.min(leftMin, prices[l]);  int r = length - 1 - l;  rightProfits[r] = Math.max(rightProfits[r + 1], rightMax - prices[r]);  rightMax = Math.max(rightMax, prices[r]);  }  int maxProfit = 0;  for (int i = 0; i < length; ++i) {  maxProfit = Math.max(maxProfit, leftProfits[i] + rightProfits[i + 1]);  }  return maxProfit;  }  } |

In the above implementations, we refined the code a bit to make it a bit more concise and hopefully more intuitive. Here are some tweaks that we applied.

* Rather than constructing the two DP arrays in two separate loops, we do the calculation in a single loop (two birds with one stone).
* We pad the right\_profits[i] array with an additional zero, which indicates the maximum profits that we can gain from an empty right subsequence, so that we can compare the result of having only one transaction (i.e. left\_profits[N-1]) with the profits gained from doing two transactions.

By the way, one can try the above algorithm on another problem called [Sliding Window Maximum](https://leetcode.com/articles/sliding-window-maximum/).

**Complexity**

* Time Complexity: O(*N*) where *N* is the length of the input sequence, since we have two iterations of length *N*.
* Space Complexity: O(*N*) for the two arrays that we keep in the algorithm.

#### Approach 2: One-pass Simulation

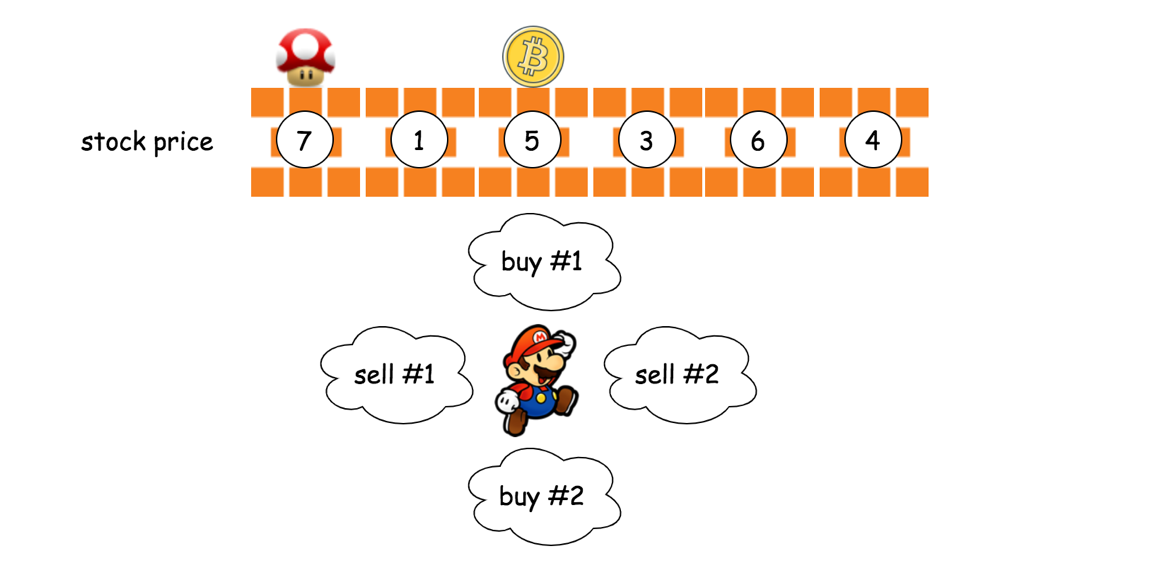
**Intuition**

Just when we think that the space complexity of O(*N*) is the best we can get for this problem, many users in the Discussion forum proposed a more optimized solution that reduced the space complexity to *O*(1), (just to name a few of them [@weijiac](https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iii/discuss/39611/Is-it-Best-Solution-with-O(n)-O(1).), [@shetty4l](https://leetcode.com/problems/best-time-to-buy-and-sell-stock-iii/discuss/149383/Easy-DP-solution-using-state-machine-O(n)-time-complexity-O(1)-space-complexity)). The idea is quite brilliant, and requires only a single iteration without the additional DP arrays.

The intuition is that we can consider the problem as a game, and we as agent could make at most two transactions in order to gain the maximum points (profits) from the game.

The two transactions be decomposed into 4 actions: "buy of transaction #1", "sell of transaction #1", "buy of transaction #2" and "sell of transaction #2".

To solve the game, we simply run a simulation along the sequence of prices, at each time step, we calculate the potential outcomes for each of our actions. At the end of the simulation, the outcome of the final action "sell of transaction #2" would be the desired output of the problem.



**Algorithm**

Overall, we run an iteration over the sequence of prices.

Over the iteration, we calculate 4 variables which correspond to the costs or the profits of each action respectively, as follows:

* t1\_cost: the minimal cost of buying the stock in transaction #1. The minimal cost to acquire a stock would be the minimal price value that we have seen so far at each step.
* t1\_profit: the maximal profit of selling the stock in transaction #1. Actually, at the end of the iteration, this value would be the answer for the first problem in the series, i.e. [Best Time to Buy and Sell Stock](https://leetcode.com/problems/best-time-to-buy-and-sell-stock/).
* t2\_cost: the minimal cost of buying the stock in transaction #2, while taking into account the profit gained from the previous transaction #1. One can consider this as the cost of reinvestment. Similar with t1\_cost, we try to find the lowest price so far, which in addition would be partially compensated by the profits gained from the first transaction.
* t2\_profit: the maximal profit of selling the stock in transaction #2. With the help of t2\_cost as we prepared so far, we would find out the maximal profits with at most two transactions at each step.

|  |
| --- |
| class Solution {  public int maxProfit(int[] prices) {  int t1Cost = Integer.MAX\_VALUE,  t2Cost = Integer.MAX\_VALUE;  int t1Profit = 0,  t2Profit = 0;  for (int price : prices) {  // the maximum profit if only one transaction is allowed  t1Cost = Math.min(t1Cost, price);  t1Profit = Math.max(t1Profit, price - t1Cost);  // reinvest the gained profit in the second transaction  t2Cost = Math.min(t2Cost, price - t1Profit);  t2Profit = Math.max(t2Profit, price - t2Cost);  }  return t2Profit;  }  } |

**Complexity**

* Time Complexity: O(*N*), where *N* is the length of the input sequence.
* Space Complexity: O(1), only constant memory is required, which is invariant from the input sequence.

**Best Time to Buy and Sell Stock IV**

You are given an integer array prices where prices[i] is the price of a given stock on the ith day.

Design an algorithm to find the maximum profit. You may complete at most k transactions.

**Notice** that you may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

**Example 1:**

**Input:** k = 2, prices = [2,4,1]

**Output:** 2

**Explanation:** Buy on day 1 (price = 2) and sell on day 2 (price = 4), profit = 4-2 = 2.

**Example 2:**

**Input:** k = 2, prices = [3,2,6,5,0,3]

**Output:** 7

**Explanation:** Buy on day 2 (price = 2) and sell on day 3 (price = 6), profit = 6-2 = 4. Then buy on day 5 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.

**Constraints:**

* 0 <= k <= 100
* 0 <= prices.length <= 1000
* 0 <= prices[i] <= 1000

## Solution

### **Overview**

You probably can guess from the problem title, this is the fourth problem in the series of [Best Time to Buy and Sell Stock](https://leetcode.com/problems/best-time-to-buy-and-sell-stock/) problem. It's strongly recommended that you should finish the previous problems before starting this one. Nevertheless, it's not necessary to finish the previous problems to understand this solution, and you can even use the methods we provide to help you solve the other problems.

Here, two approaches are introduced: Dynamic Programming approach, and Merging approach. Both are awesome, but the first method is more universal to other problems.

#### Approach 1: Dynamic Programming

**Intuition**

[Dynamic programming](https://en.wikipedia.org/wiki/Dynamic_programming) (dp) is a popular method among hard-level problems. Its basic idea is to store the previous result to reduce redundant calculations. However, it is hard for beginners to think of the dp method. Below, a step-by-step tutorial of how to think of dp is introduced. If you are already familiar with dp, you can jump to the algorithm part to check out the actual implementation.

Generally, there are two ways to come up with a dp solution. One way is to start with a brute force approach and reduce unnecessary calculations. Another way is to treat the stored results as "states", and try to jump from the starting state to the ending state.

For beginners, it is recommended to start with the brute force approach. So, how to brute force to solve this problem?

Back to (part of) the question:

Say you have an array for which the i-th element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete at most k transactions.

Cool, looks like we need to arrange at most k transactions. A natural idea is to iterate all the possible combinations of k transactions, and then find the best combination. As for those with less than k transactions, they are similar and can be considered later. A transaction consists of two parts: buying and selling. Therefore, we need to find 2k points in the stock line, k points for buying, and k points for selling.

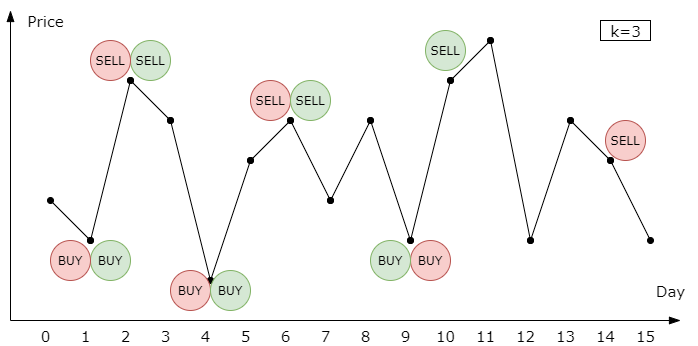
Now, we can roughly estimate the time complexity. Suppose there are n days in total, and we need to pick 2k days. The number of possible situations is about . It's not a good result because it involves factorial, which is likely to cause Time Limit Exceeded (TLE). Usually what we need is a polynomial one. However, it includes some invalid situations so the actual number is smaller.

Another problem is that, what if 2k is larger than n? In this case, we are not able to pick 2k points from n points, which means we will not reach the limit no matter how we try. Therefore, all we need to do is to iterate each day, and if the price of day i arise, buy the stock in i-1th day and sell it at ith day.

2k > n is a special case and can be addressed easily.

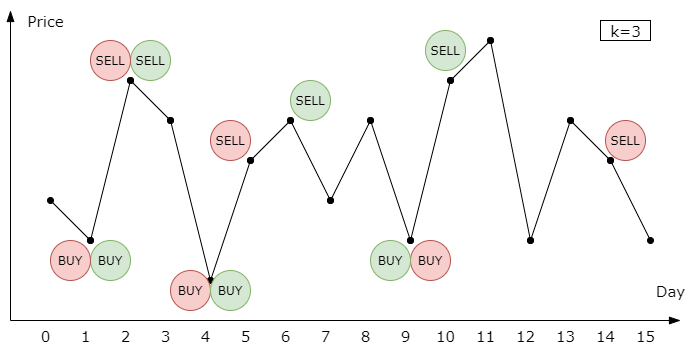
Back to our factorial number. The next step is to review our brute force approach and find out the possible redundant calculations. In our brute force approach, we need to iterate all the possible combinations and calculate the profit of each one to find the best. Can you find out where repeated calculations are?

Consider the following case, where the red color represents a possible combination, and the green represents another one:



The two combinations are the same before day 10. If we calculate the profits separately, we need to calculate the profit before day 10 twice. Here is where dp comes! We can store the current balance on day 9, and reuse it later. Therefore, we can store the result in a dict, where the key is the day number and the transactions we made before, and the value is the balance. Wait a minute, can we do better?

Consider another case:



The only difference is that the red sells stock at a lower price during the second transaction. Therefore, the red has a lower profit on day 10 than the green has. In this case, we need not calculate the rest profit of the red, since it can not beat the green in the future.

Therefore, we can compare those reds, and continue the next day with the one with the highest profit. However, we need to ensure that the best one will not be beaten by the "losers" in the future, so they should have the same "resources" at the time we store and compare the balances.

Hence, we can use three characteristics to store the profit: the day number, the transaction number used, and the stock holding status. You can use other representations of resources, such as using "the day remained" instead of "the day number". Feel free to try. Now, let's go to the algorithm part.

**Algorithm**

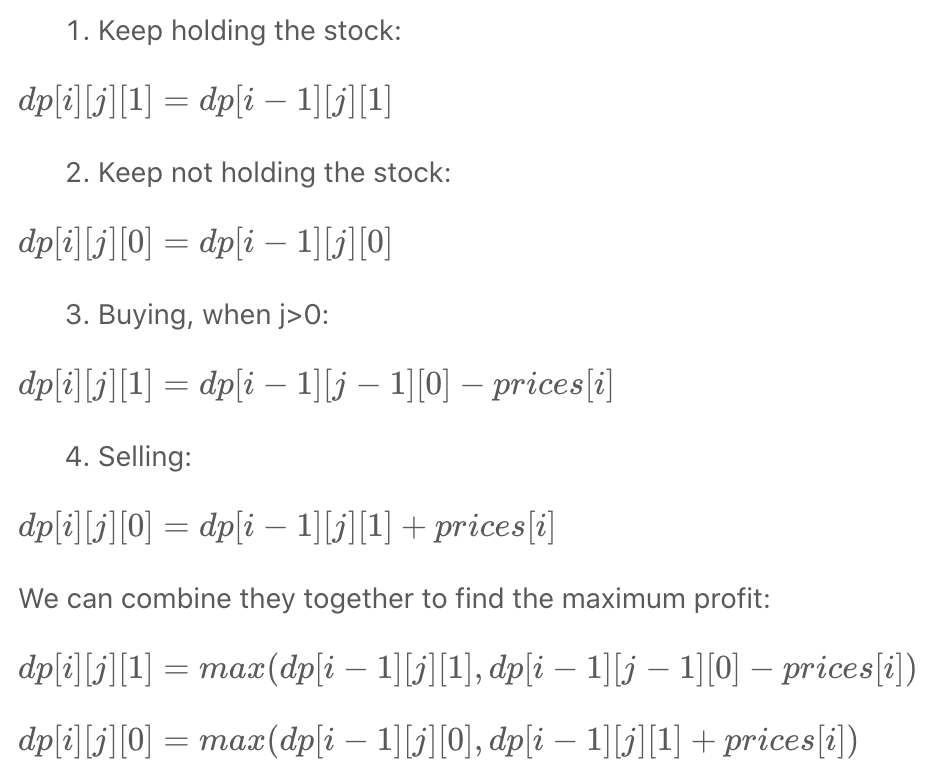
In the previous part, we introduced an intuitive idea from brute force to dp method, and here we need to decide the details of the algorithm.

We can either store the dp results in a dict or an array. Array costs less time for accessing and updating than dict, so we always prefer an array when possible. Because of three needed characteristics (day number, transaction number used, stock holding status), a three-dimensional array is our choice. We can use dp[day\_number][used\_transaction\_number][stock\_holding\_status] to represent our states, where stock\_holding\_status is a 0/1 number representing whether you hold the stock or not.

The value of dp[i][j][l] represents the best profit we can have at the end of the i-th day, with j remaining transactions to make and l stocks.

The next step is finding out the so-called "transition equation", which is a method that tells you how to jump from one state to another.

We start with dp[0][0][0] = 0 and dp[0][0][1]=-prices[0], and our final aim is max of dp[n-1][j][0] from j=0 to j=k. Now, we need to fill out the entire array to find out the result. Assume we have gotten the results before day i, and we need to calculate the profit of day i. There are only four possible actions we can do on day i: 1. keep holding the stock, 2. keep not holding the stock, 3. buy the stock, or 4. sell the stock. The profit is easy to calculate.



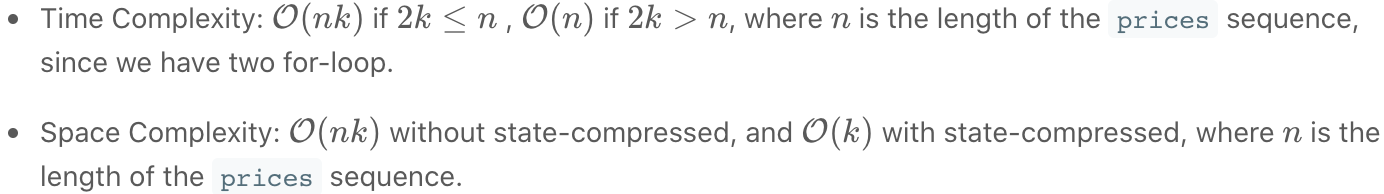
Awesome! Now we can use for-loop to calculate the whole dp array and achieve our final result. Remember to solve the special cases when 2k > n.

|  |
| --- |
| public class Solution {  public int maxProfit(int k, int[] prices) {  int n = prices.length;  // solve special cases  if (n <= 0 || k <= 0) {  return 0;  }  if (2 \* k > n) {  int res = 0;  for (int i = 1; i < n; i++) {  res += Math.max(0, prices[i] - prices[i - 1]);  }  return res;  }  // dp[i][used\_k][ishold] = balance  // ishold: 0 nothold, 1 hold  int[][][] dp = new int[n][k + 1][2];  // initialize the array with -inf  // we use -1e9 here to prevent overflow  for (int i = 0; i < n; i++) {  for (int j = 0; j <= k; j++) {  dp[i][j][0] = -1000000000;  dp[i][j][1] = -1000000000;  }  }  // set starting value  dp[0][0][0] = 0;  dp[0][1][1] = -prices[0];  // fill the array  for (int i = 1; i < n; i++) {  for (int j = 0; j <= k; j++) {  // transition equation  dp[i][j][0] = Math.max(dp[i - 1][j][0], dp[i - 1][j][1] + prices[i]);  // you can't hold stock without any transaction  if (j > 0) {  dp[i][j][1] = Math.max(dp[i - 1][j][1], dp[i - 1][j - 1][0] - prices[i]);  }  }  }  int res = 0;  for (int j = 0; j <= k; j++) {  res = Math.max(res, dp[n - 1][j][0]);  }  return res;  }  } |

There a few points you should notice from the code above:

1. Take care of the initial values in dp array. Generally, it's ok to initialize them to zero. However, in this case, we need to make them -inf to mark impossible situations, such as dp[0][0][1].
2. You can reverse the order of filling the dp array, with some modifications in the transition equation. For example, decreasing j instead of increasing it.
3. Some state-compressed method can be applied if you want. For example, we only need dp[i-1], when calculating dp[i], therefore we can delete other useless dp to save memory. Just using two arrays to storing dp[i-1] and dp[i] and refreshing them every iteration will do.
4. The code above is not the fastest because we prioritize the readability. It would be faster if you put the larger dimension in the inner array since it uses CPU cache more efficiently.

**Complexity**

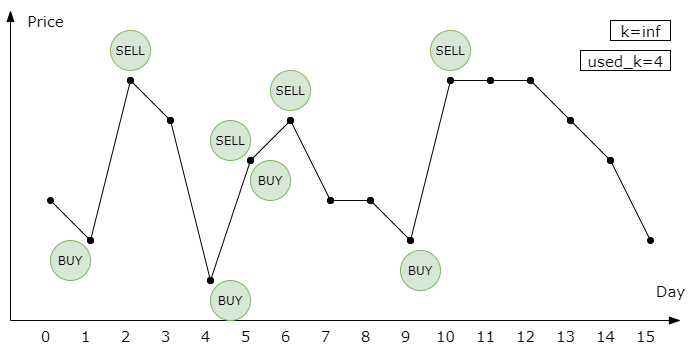


#### Approach 2: Merging

**Intuition**

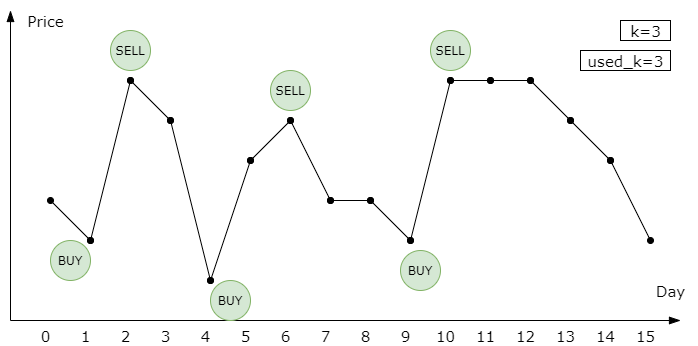
This approach starts from a simple situation with k=infinity, and drecrease k one by one.

Consider a weakened problem when k=infinity. Since we already know the prices of tomorrow, our solution is to trade whenever prices[i-1] < prices[i]. Below is an example.



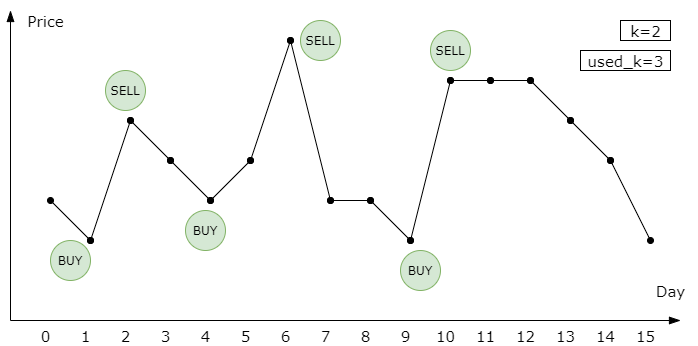
We only used 4 transactions! However, what we need to solve is the case with an actual k. Let's decrease k from inf and see what happens. Our solution can handle all the k >=4, since we only used 4 transactions. But what if k=3?

Notice that at day 5, we buy and sell the stock at the same time. We can cancel the redundant transaction without impact the final profit!



We can conclude that for the consecutively increasing subsequence, we only need to buy once at the start and sell once at the end.

How about k=2? Maybe we need to delete one transaction. We can iterate all the transactions and delete the one with least revenue. However, deleting can not always achieve our best solution. Consider the following example:

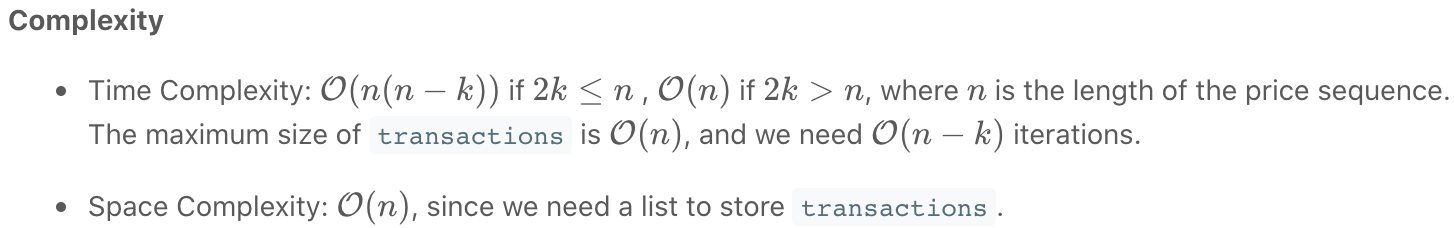


When k=2, the best solution is to buy at day 1 and day 9, and to sell on day 6 and day 10. Deleting any transactions cannot reach this solution. However, we can merge the previous two transactions to get to this. A naive approach is iterating all the near transactions and find out the pair with the lowest impact on the revenue. Since we decrease k one by one, reducing one transaction is enough. Ok, let's go to the algorithm part to check the detail.

**Algorithm**

The general idea is to store all consecutively increasing subsequence as the initial solution. Then delete or merge transactions until the number of transactions less than or equal to k.

|  |
| --- |
| public class Solution {  public int maxProfit(int k, int[] prices) {  int n = prices.length;  // solve special cases  if (n <= 0 || k <= 0) {  return 0;  }  // find all consecutively increasing subsequence  ArrayList<int[]> transactions = new ArrayList<>();  int start = 0;  int end = 0;  for (int i = 1; i < n; i++) {  if (prices[i] >= prices[i - 1]) {  end = i;  } else {  if (end > start) {  int[] t = { start, end };  transactions.add(t);  }  start = i;  }  }  if (end > start) {  int[] t = { start, end };  transactions.add(t);  }  while (transactions.size() > k) {  // check delete loss  int delete\_index = 0;  int min\_delete\_loss = Integer.MAX\_VALUE;  for (int i = 0; i < transactions.size(); i++) {  int[] t = transactions.get(i);  int profit\_loss = prices[t[1]] - prices[t[0]];  if (profit\_loss < min\_delete\_loss) {  min\_delete\_loss = profit\_loss;  delete\_index = i;  }  }  // check merge loss  int merge\_index = 0;  int min\_merge\_loss = Integer.MAX\_VALUE;  for (int i = 1; i < transactions.size(); i++) {  int[] t1 = transactions.get(i - 1);  int[] t2 = transactions.get(i);  int profit\_loss = prices[t1[1]] - prices[t2[0]];  if (profit\_loss < min\_merge\_loss) {  min\_merge\_loss = profit\_loss;  merge\_index = i;  }  }  // delete or merge  if (min\_delete\_loss <= min\_merge\_loss) {  transactions.remove(delete\_index);  } else {  int[] t1 = transactions.get(merge\_index - 1);  int[] t2 = transactions.get(merge\_index);  t1[1] = t2[1];  transactions.remove(merge\_index);  }  }  int res = 0;  for (int[] t : transactions) {  res += prices[t[1]] - prices[t[0]];  }  return res;  }  } |

****