Question 1

a) What are the possible solutions for the problem?
   1. Using *stringstream* and *getline* to handle the input, *vector* to store the data, *unordered_set* to check the distinct value. Check the distinct value and sum the data by traverse the entire array when the command is 2 or 3.
   2. Using *cin* to handle the input, dynamic array to store the data and check the distinct value. Check the distinct value and sum once handle the input, each time the array is updated, simply update the distinct value and sum. When the command is 2 or 3 simply return the value.

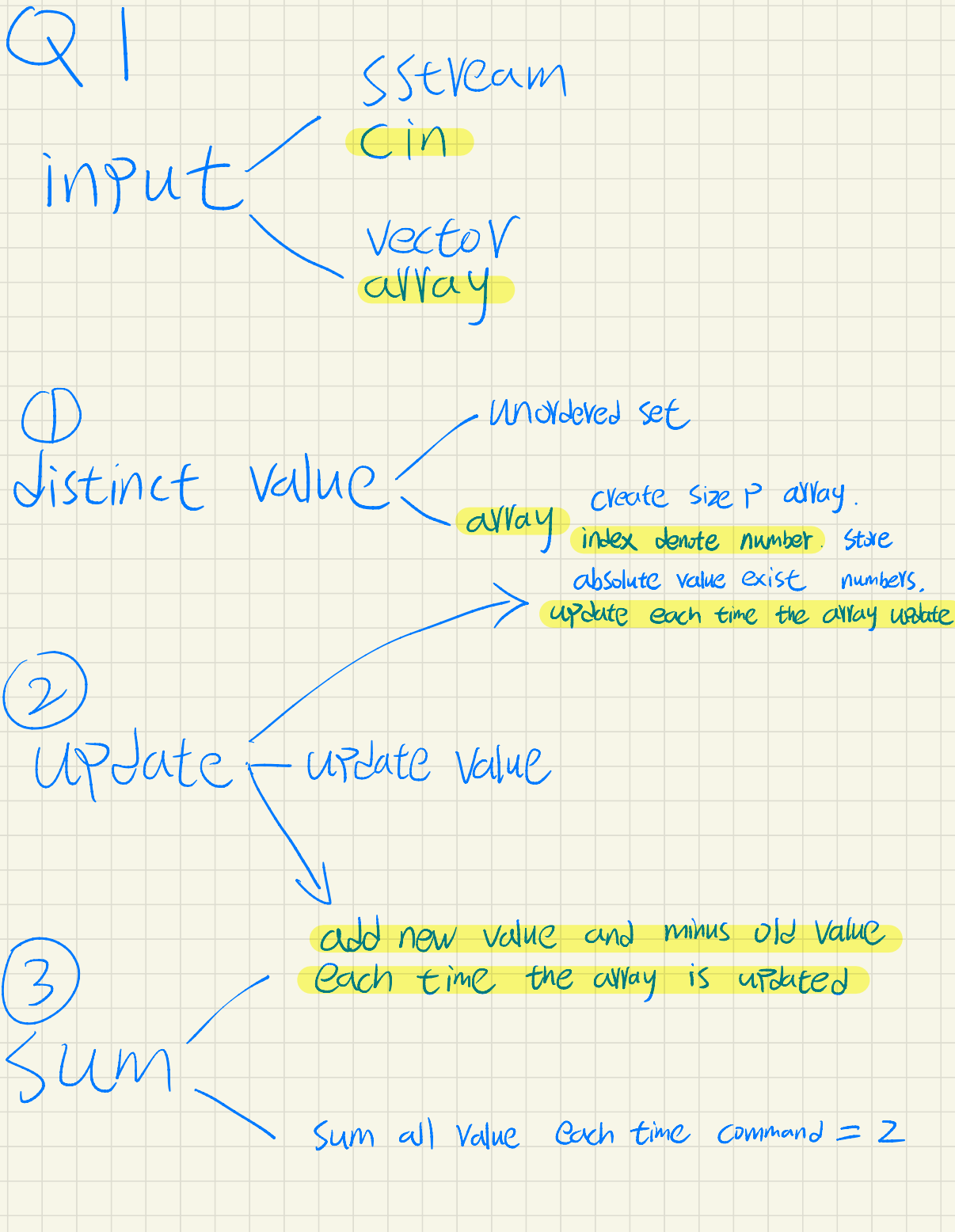
b) How do you solve this problem?
   Input: At first, I tried to figure out how to handle the input. I used *stringstream* and *getline* to handle the input and store the array in an vector, later on I found out that it is lack of efficiency and change to use only *cin* and dynamic array to handle the input.

   Operation checks distinct value: First I used the unordered set to check out the distinct value, add the array element to the set, if meet the same value it multiplies minus one and try to add again. At last, it returns the size of the set which denote the number of distinct values. It run each time the checks distinct value operation is pull up. Later I found out that it reports TLE on OJ, so I use only array to check the value. I create a dynamic array of size P with all elements 0. Since -P < a[i] < P, we can use the index to denote the absolute value of a[i], and the element of index abs(a[i]) denote the times that it appears. For elements 1 and 2, it adds to distinct value 1 and 2, for element larger than 2 it still adds 2. For a[i] = 0, it adds to distinct value 1 if element is larger than 0. Also, it only run at the first time we get the whole array from the input, then we only update it when the array value is updated.

   Operation sum: First I simply add up the numbers together each time the operation is pull up. This is the main reason that my code always get a TLE on OJ, because the time complexity is O(n * m) (m denote the number of operation sum up). So I then define a global variable to store the sum, it only sum all numbers up at the beginning, later it only add or minus the old and new number each time the array is updated, which improve the efficiency.

   Operation update: very simply, skipped.

c) Why is your solution better than others?
   My solution using only array, and update the distinct value and sum when updating the data, no need to run the function every time to traverse the entire array and it saves the time. Also using the dynamic array can save the memory.


Q1
input
sstream
cin
vector
array
①
distinct value
unordered set
array
create size P array.
index denote number. store
absolute value exist numbers.
update each time the array update
②
update
update value
add new value and minus old value
each time the array is updated
③
sum
sum all value each time command = 2

Question 2

a) What are the possible solutions for the problem?
   1. Using *cin* to handle the input, merge the left and right array, and delete the repeat element, we get a merged array that may have contents fit the permutation. Read the permutation from left to right, and match the elements in the merged array, if all elements in the merged array can be found in order in the permutation array, and the first and last elements of both arrays match each other, we can consider it is a valid permutation and return 1, else 0.
   2. Following the hint, we process from the middle of the permutation array. Using *cin* to handle the input, and we first using a *map* to store the index of the permutation array's elements. Then get the last element of left and right array, and find the index of the two elements at the map, then we update the left and right boundary by the index and repeat the process until all elements in the left and right array have been read. Finally we check out the left and right boundary is equal to 0 and n – 1 or not to return 1 or 0.

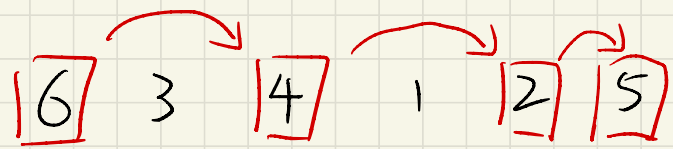
b) How do you solve this problem?
   Input: Simply using *cin* to handle the input, get three array, permutation array, left array and right array.

   Check valid: Define a map first to store the index of permutation elements, then initialize the left and right boundary which denote the index of the permutation array. Then traverse the entire left and right array from the last element to the first element, check each element's index in the map of the permutation array index, and compare the index whether the left index is larger than right index or not, if yes, return 0 since left index should be larger than right index. At last, we check if the left boundary and right boundary equals to 0 and n – 1 or not, if yes it denotes that the permutation is valid and return 1.

c) Why is your solution better than others?
   Starting the process from bottom to top make the code work efficiently to track the left and right boundaries without repeatedly splitting the array. By using a map to store index of permutation array reduced the overall complexity. The first solution I write that match the array from left to right will cause unknown problem, since it doesn't follow the principle of pushing from the result to the original array.

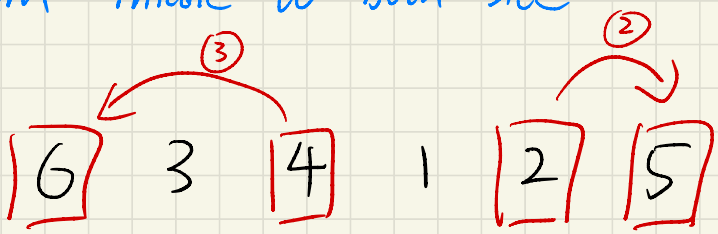
Q2

① from left to right



L = 6, 4, 4
R = 5, 5, 2
→ merge = 6 4 2 5

② from middle to both side.



③ ② ①
L = 6, 4, 4
R = 5, 5, 2
→ 6 4 2 5