Hackerrank: Larry’s array editorial

by Mourad NOUAILI

Problem’s link: https://www.hackerrank.com/challenges/larrys-array/problem

"You don't actually need to do any rotation to try and solve this; you can use a similar algorithm that is used to determine the solvability of the 15-puzzle/Tiles Game."

Boomx from hackerrenk

Formula for determining solvability

"In computer science and discrete mathematics a sequence has an **inversion** where two of its elements are out of their natural order."

Wikipédia

Given a sequence of integers (a permutation) , we gonna check the sortedness of this permutation.

We need to compute the number of inversions of the given permutation.

Why we need #inversions?

The #inversions of the identity permutation {1, 2, 3, …, n} = 0.

In the problem Larry’s array, we make a rotation of three integers once each iteration.

#inversions of these three adjacency integers, will increase or decrease by 2 in each rotation.

Proof:

for a sequence S = {a1, a2, a3} such as a1 < a2 < a3.

Rotation #1: {a2, a3, a1} → #inversions = 2

Rotation #2: {a3, a1, a2} → #inversions = 2

Rotation #3: {a1, a2, a3} → #inversions = 0

So, the entire computed #inversions will increase or decrease by 2 or 0. This implies that the total #inversions must be even, to be able reaching 0.

In the next part, we’ll see the different algorithms to compute the number of inversions of a given sequence of positif integers.

Bad solution: O(n²)

This solution consists to compare each element a_i ($0 \leq i < n$) in the array and the all other elements a_j ($i+1 \leq j < n$) , if $a_j > a_i$, increment the value of the number of inversions by 1.

$$
\begin{array}{l}
\textit{for each entry } a_i (0 \leq a_i < n) \\
\quad \textit{for each entry } a_j (i+1 \leq a_j < n) \\
\quad \textit{if } a_i < a_j \\
\qquad \text{inv\_count} += 1
\end{array}
$$

C++ function's code:

```
int number_of_inversions(vector<int> A) {
  int n = A.size();
  int inv = 1;
  for (int i = 0 ; i < n ; ++i)
    for (int j = i+1 ; j < n ; ++j)
      inv ^= (A[i] > A[j]);
  return inv;
}
```

A way to an optimized solution

Instead computing, for each integer a_i in the array, how many integers are less than a_i.

We can work at the moment of the entering of integers:

- For each entry a_i ($1 \le a_i \le n$), compute the number of integers previously entered that are greater than a_i.

In mathematics words: For each entry a_i ($1 \le a_i \le n$), search all integer x_j ($0 \le j \le i-1$), which is $x_j > a_i$.

To do that, we gonna use an array to accumulate the sum of number of integers that are greater than the current entry.s

For each entry a_i:

- Updating: affect one to it in the array of sum,
- Computing: to find the number of all previous integers greater than the current entry a_i, which is sum off all ones from a_{i+1} to a_n.

All integers greater than a_i are from a_{i+1} to a_n:

A_sum:	0	1	1	0	1	0	0	1	0
	0	*1*	*2*	….	a_i	a_{i+1}	….	a_{i+k}	a_n

we compute the sum of one from 0 to n, minus the sum of all ones from 0 to a_i in the array of sum:

$$\# \, Invserions \, of \, a_i = \sum_{i=1}^{n} 1 - \sum_{i=1}^{a_i} 1$$

The general algorithm

$$\begin{aligned}&\textit{initialize}\ \text{A\_sum to } 0\\&\textit{for each entry } a_i (1 \le a_i \le n)\\&\qquad \text{A\_sum}[a_i] \leftarrow 1\\&\qquad \text{inv\_count} += \text{sum}([0, n]) - \text{sum}([0, a_i])\end{aligned}$$

<u>proof</u>

we can prove by induction.

- ***The base case***

  for the first entry $\boldsymbol{a_i}$:

  - updating: ***A_sum[a_i] ← 1***

A_sum:

0	0	0	0	1	0	0	0
0	*1*	*2*	….	a_i	a_{i+1}	….	a_n

  - Computing the number of integers greater than $\boldsymbol{a_i}$: ***sum([0, n]) - sum([0, a_i])***

  remember that $\boldsymbol{a_i}$ is the first enrty, so sum([0, n]) = 1 and sum([0, a_i]) = 1, so

#of integers greater than $\boldsymbol{a_i}$ = 1 – 1 = 0 (logic, because $\boldsymbol{a_i}$ is the 1st entry)

- **The induction hypothesis**

  we reach the last entry $\boldsymbol{a_k}$, we suppose that the formula is correct after ***updating A_sum[a_k] ← 1***, ***#inversions of a_k = sum([0, n]) - sum([0, a_k])***

- **The induction proof**

A_sum:

0	1	1	1….1	1	1	1...1	1	1
0	*1*	*2*	…..	a_k	a_{k+1}	….	a_n	a_{n+1}

- Proof that the formula is correct for an extra.

  The extra entry must be the last entry, which mean ***n+1***, because all entries are a_i in [1, n]

#inversions of a_{n+1} = sum([0, a_{n+1}]) - sum([0, a_{n+1}]) = 0

<u>which is correct, because there is no integer in [1, n], greater than n+1.</u>

Example:

A:	1	6	7	5	2	4	3
	0	*1*	*2*	*3*	*4*	*5*	*6*

Initially, ***A_sum*** is initialized to zeros:

A_sum:	0	0	0	0	0	0	0	0
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

- For input 1: [1]
  - update ***A_sum***, by put one to A_sum[1]

A_sum:	0	1	0	0	0	0	0	0
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

  - $\# invserions\, of\ 1 = the\ \text{sum}\ of\ all\, ones \in [2,7] = \sum_{i=1}^{7} 1 - \sum_{i=1}^{1} 1 = 1 - 1 = 0$

    there is no integer greater than 1.

- For input 6: [1, 6]
  - update ***A_sum***, by put one to A_sum[6]

A_sum:	0	1	0	0	0	0	1	0
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

  - $\# invserions\, of\ 6 = the\, \text{sum}\ of\ all\, ones \in [7,7] = \sum_{i=1}^{7} 1 - \sum_{i=1}^{6} 1 = 2 - 2 = 0$

    there is no integer greater than 6.

- For input 7: [1, 6, 7]
  - update ***A_sum***, by put one to A_sum[7]

A_sum:	0	1	0	0	0	0	1	1
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

  - $\# invserions\, of\ 7 = the\, \text{sum}\ of\ all\, ones \in no\ range = \sum_{i=1}^{7} 1 - \sum_{i=1}^{7} 1 = 3 - 3 = 0$

    there is no integer greater than 7.

- For input 5: [1, 6, 7, 5]
  - update ***A_sum***, by put one to A_sum[5]

A_sum:

0	1	0	0	0	1	1	1
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

  - $\# \, invserions \, of \, 5 = the \, \mathrm{sum} \, of \, all \, ones \in [6,7] = \sum_{i=1}^{7} 1 - \sum_{i=1}^{5} 1 = 4 - 2 = 2$

    there is two integers greater than 5 (7 and 6).

- For input 2: [1, 6, 7, 5, 2]
  - update ***A_sum***, by put one to A_sum[2]

A_sum:

0	1	1	0	0	1	1	1
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

  - $\# \, invserions \, of \, 2 = the \, \mathrm{sum} \, of \, all \, ones \in [3,7] = \sum_{i=1}^{7} 1 - \sum_{i=1}^{2} 1 = 5 - 2 = 3$

    there is three integers greater than 2 (5, 7 and 6).

- For input 4: [1, 6, 7, 5, 2, 4]
  - update ***A_sum***, by put one to A_sum[4]

A_sum:

0	1	1	0	1	1	1	1
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

  - $\# \, invserions \, of \, 4 = the \, \mathrm{sum} \, of \, all \, ones \in [5,7] = \sum_{i=1}^{7} 1 - \sum_{i=1}^{4} 1 = 6 - 3 = 3$

    there is three integers greater than 4 (5, 7 and 6).

- For input 3: [1, 6, 7, 5, 2, 4, 3]
  - update ***A_sum***, by put one to A_sum[3]

A_sum:

0	1	1	1	1	1	1	1
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

  - $\# \, invserions \, of \, 3 = the \, \mathrm{sum} \, of \, all \, ones \in [4,7] = \sum_{i=1}^{7} 1 - \sum_{i=1}^{3} 1 = 7 - 3 = 4$

    there is four integers greater than 3 (4, 5, 7 and 6).

`#inversions of the array A = 0 + 0 + 0 + 2 + 3 + 3 + 4 = 12`

Other example:

A:	7	11	8	13
	0	*1*	*2*	*3*

A_sum	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*

Instead of looking for the max of the input, can convert the original array to an array with consecutive integers of size n without changing the number of inversions.

A:	7	11	8	13
	0	*1*	*2*	*3*

Become

A:	1	3	2	4
	0	*1*	*2*	*3*

As 7 < 8 < 11 <13 (1 < 2 < 3 <4)
to do that:

- sort the array in an other temporary array temp
- put in a hash map structure the information {temp[i], i+1}
- run over the original array A, and convert the key A[i] by its value in the map.

temp:	7	8	11	13
	0	*1*	*2*	*3*

Map= {{7, 1}, {8, 2}, {11, 3}, {13, 4}}

A:	Map[7] = 1	Map[11] = 3	Map[8] = 2	Map[13] = 4
	0	*1*	*2*	*3*

A:	1	3	2	4
	0	*1*	*2*	*3*

A_sum:	0	0	0	0	0
	0	*1*	*2*	*3*	*4*

- For input 1: [1]
  - update: A_sum[1] = 1

A_sum:	0	1	0	0	0
	0	*1*	*2*	*3*	*4*

$$\# invserions\ of\ 1 = the\ \mathrm{sum}\ of\ all\ ones \in [2,4] = \sum_{i=1}^{4} 1 - \sum_{i=1}^{1} 1 = 1 - 1 = 0$$

- For input 3: [1, 3]
  - update: A_sum[3] = 1

A_sum:	0	1	0	1	0
	0	*1*	*2*	*3*	*4*

$$\# invserions\ of\ 3 = the\ \mathrm{sum}\ of\ all\ ones \in [4,4] = \sum_{i=1}^{4} 1 - \sum_{i=1}^{3} 1 = 2 - 2 = 0$$

- For input 2: [1, 3, 2]
  - update: A_sum[3] = 1

A_sum:	0	1	1	1	0
	0	*1*	*2*	*3*	*4*

$$\# invserions\ of\ 2 = the\ \mathrm{sum}\ of\ all\ ones \in [3,4] = \sum_{i=1}^{4} 1 - \sum_{i=1}^{2} 1 = 3 - 2 = 1$$

- For input 4: [1, 3, 2, 4]
  - update: A_sum[3] = 1

A_sum:	0	1	1	1	1
	0	*1*	*2*	*3*	*4*

$$\# invserions\ of\ 4 = the\ \mathrm{sum}\ of\ all\ ones \in no\ range = \sum_{i=1}^{4} 1 - \sum_{i=1}^{4} 1 = 4 - 4 = 0$$

```
#inversions of the array A = 0 + 0 + 1 + 0 = 1
```

C++11 Code: still a quadratic complexity

```
/*
Calculate the number of inversion using a array that compute the sum of all ones.
Complexity of the problem: O(t * N²)
Complexity of computing #inversions: O(N²)
*/

#include <bits/stdc++.h>

using namespace std;

int *A_sum = NULL;

int range_sum(int x, int end){
  int s = 0;
  for (int i = x+1 ; i <= end ; ++i)
    s += A_sum[i];
  return s;
}

int* convert(int *A, int n){;
  int *temp = new int[n];
  for (int i=0; i<n; i++)
    temp[i] = A[i];

  sort(temp, temp+n);

  map<int, int> A1;
  for (int i = 0; i < n; ++i)
    A1.insert({temp[i], i + 1});

  for (int i=0; i<n; i++)
    A[i] = A1[A[i]];

  return A;
}

int main(){
    int t;
    cin >> t;

    while (t--) {
        int n;
        cin >> n;

        int *A = new int[n];
        for (int i = 0; i < n; ++i) {
          cin >> A[i];
        }

        A = convert(A, n);

        A_sum = new int[n+1];
        fill_n(A_sum, n+1, 0);
        int inv_count = 0;
        for (int i = 0; i < n; ++i) {
          A_sum[A[i]] = 1;
          inv_count += range_sum(A[i], n);
```

```
        }

        inv_count % 2 != 0 ? cout<<"NO\n" : cout<<"YES\n";

        delete[] A;
        delete[] A_sum;
    }

    return 0;
}
```

Can we get better than a quadratic complexity?
The answer is yes, using a Fenwick tree.

Optimized solution

As we saw, in the array ***A_sum***, we have a ***point update (A_sum[A[i]] = 1)*** and a ***range query (sum from A_sum[i+1] to A_sum[max(A[i])]***.

We gonna use a Fenwick tree to resolve this problem in O (t * N * log N)
The complexity of computing the #of inversions = O(N log N)

If you wanna more details on Fenwick trees, see my github:
https://github.com/Mourad-NOUAILI/advanced-tutorials

How to build the Fenwick tree ?

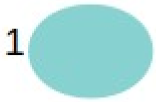
Update (point update)

In a BIT, the updating begin by a node and then all its parents until reaching or up bounding the index of greatest node.

To get the parent of a node, we add the least significant bit (LSB) to the index of that node.

The formula is: parent(index) = index + (index & -index)

We start by node 1:

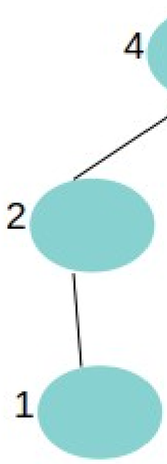


To find the parent of 1, we add to 1, the LSB of 1:

```
    0  1  (1)
 +
    0  1  (LSB of 1)
 ------------
    1  0  → 2 in decimal
```

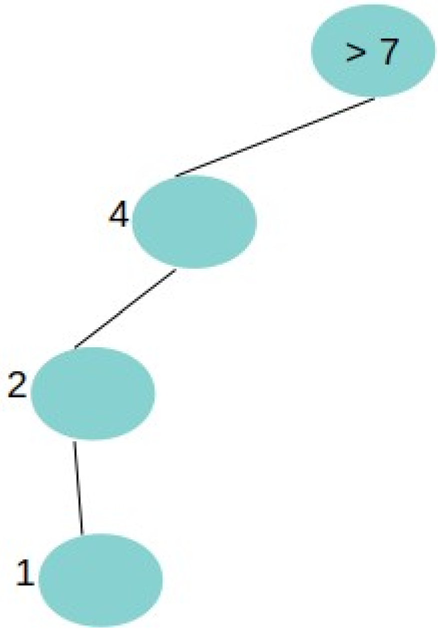
To find the parent of 2, we add to 2, the least significant bit (LSB) of 2:

```
       1  0  (2)
  +
       1  0  (LSB of 2)
 ---------------
    1  0  0  → 4 in decimal
```

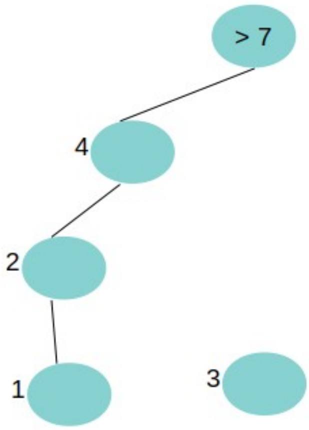


To find the parent of 4, we add to 4, the least significant bit (LSB) of 4:

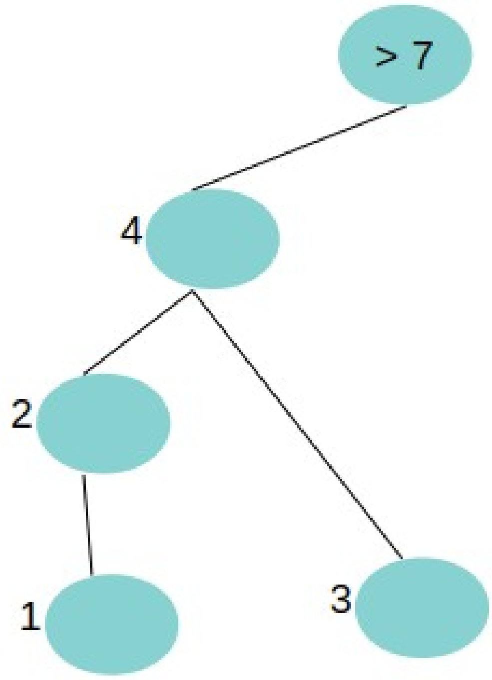
	1	0	0	(4)
+				
	1	0	0	(LSB of 4)
1	0	0	0	→ 8 in decimal > 7



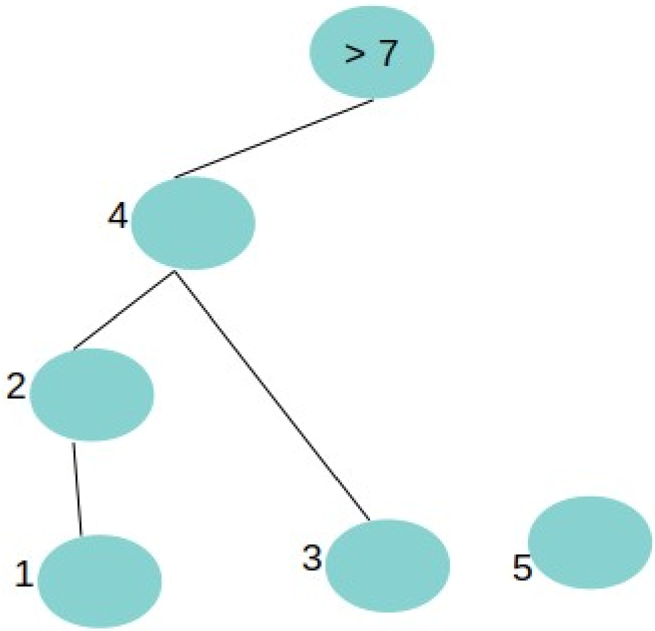
we did 1, 2, 4. it remains 3, 5, 6, 7
Let’s construct all the parents of 3:



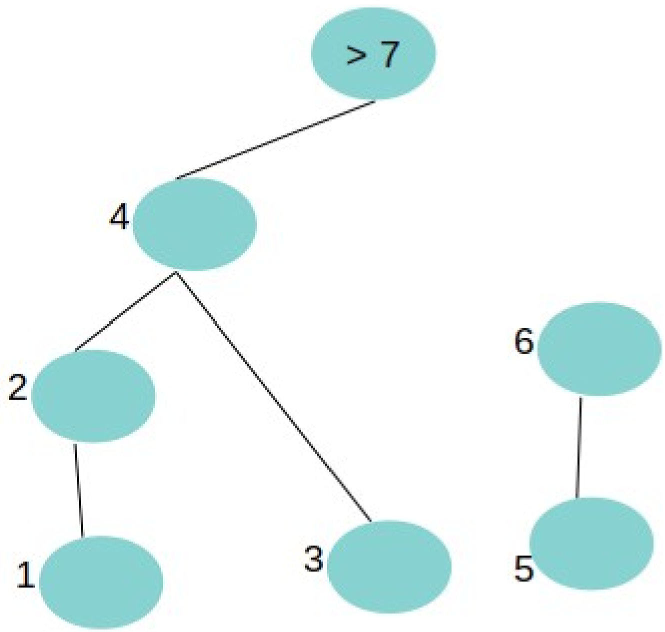
$$
\begin{array}{r@{\quad}c@{\quad}c@{\quad}c@{\quad}l}
 & 0 & 1 & 1 & (3) \\
+ & & & & \\
 & 0 & 0 & 1 & \text{(LSB of 3)} \\
\hline
 & 1 & 0 & 0 & \rightarrow \text{4 in decimal}
\end{array}
$$



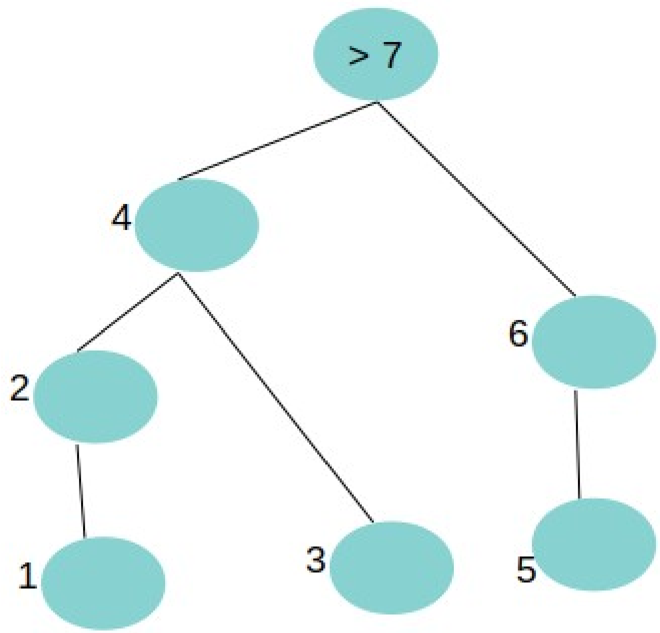
we construct 5:



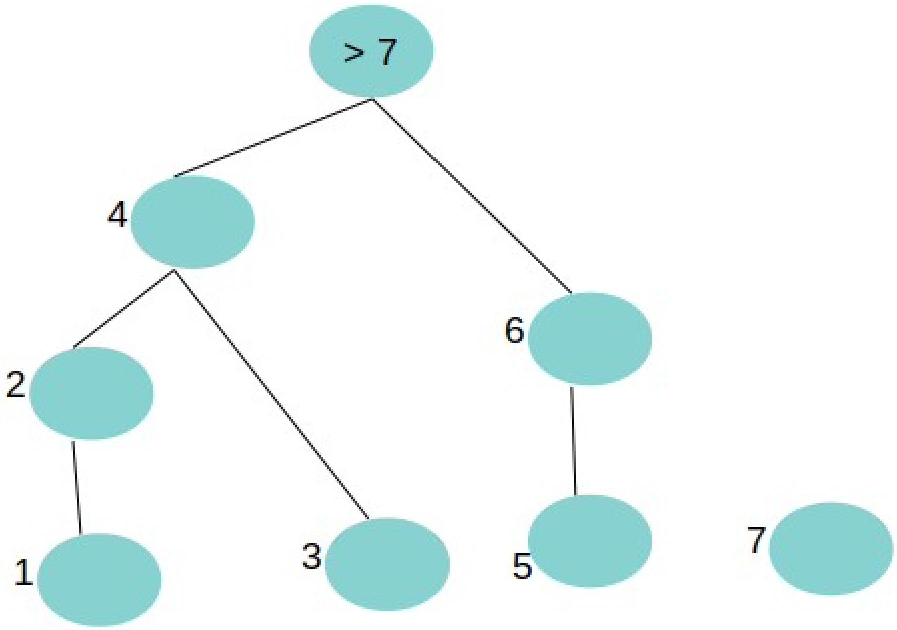
	0	1	0	1	(5)
+					
	0	0	0	1	(LSB of 5)
	0	1	1	0	→ 6 in decimal



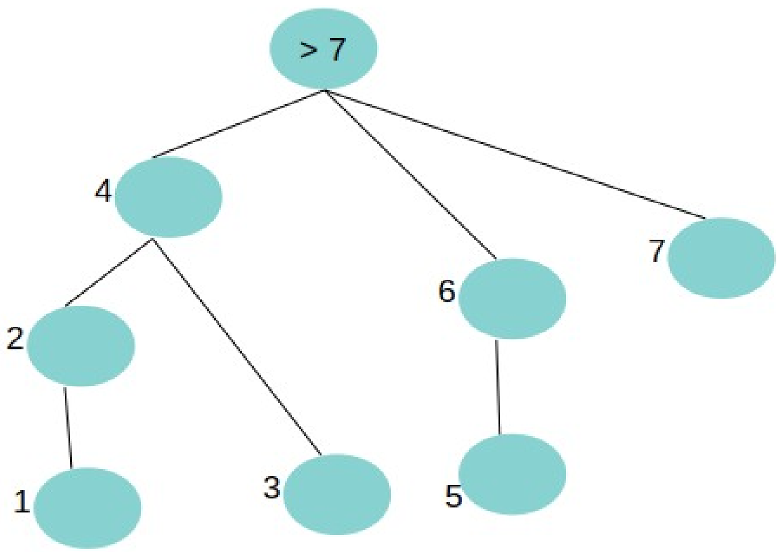
	0	1	1	0	(6)
+					
	0	0	1	0	(LSB of 6)
	1	0	0	0	→ 8 in decimal > 7



we construct 7:



	0	1	1	1	(7)
+					
	0	0	0	1	(LSB of 7)
	1	0	0	0	→ 8 in decimal > 7



This is the updating Fenwick tree.

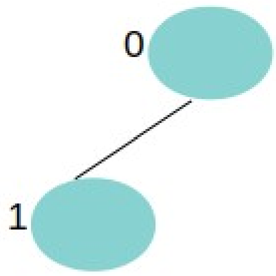
Query (point query or point get sum)

For implementing get sum, we need to visualize the BIT in a different way.

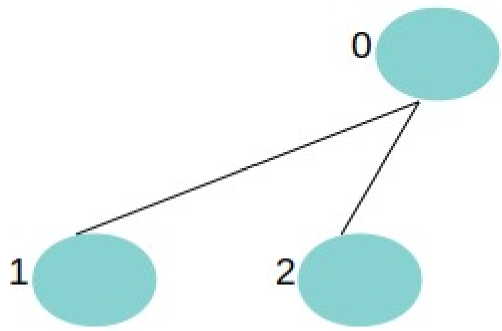
Accumulating the values of all parent of a node until reaching the index 0.
To get the parents of a node, we remove the least significant bit (LSB) from the index of that node.

The formula is: parent(index) = index - (index & -index)

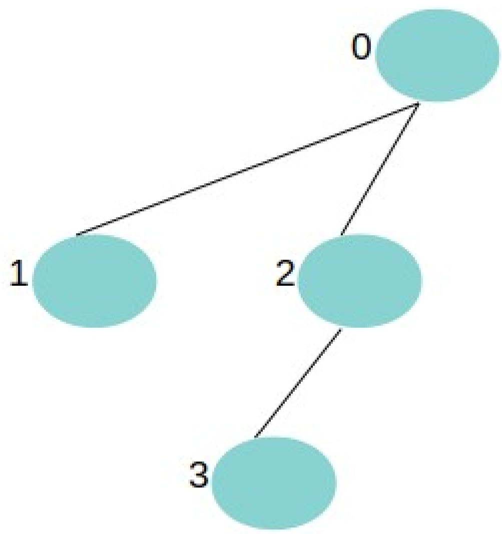
	0	1	(1)
-			
	0	1	(LSB of 1)
	0	0	→ 0 in decimal



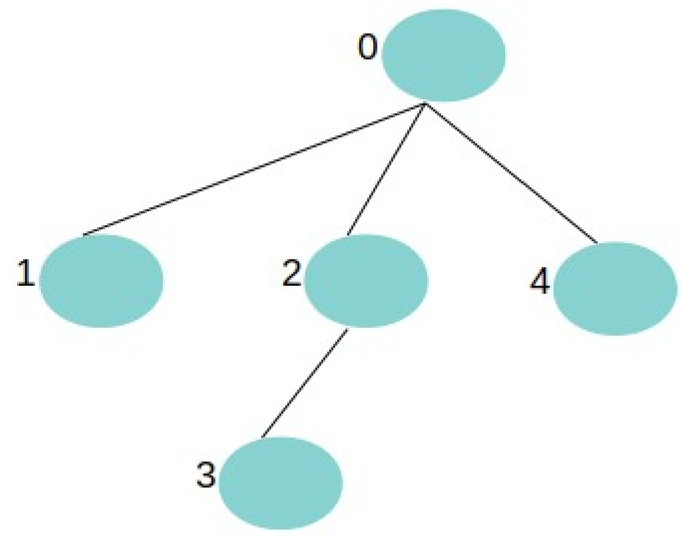
	1	0	(0)
-			
	1	0	(LSB of 2)
	0	0	→ 0 in decimal



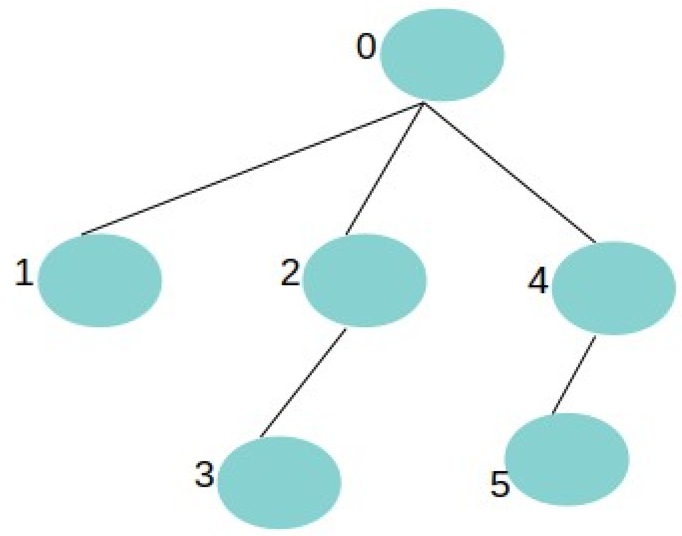
	0	1	1	(3)
-				
	0	0	1	(LSB of 3)
	0	1	0	→ 2 in decimal



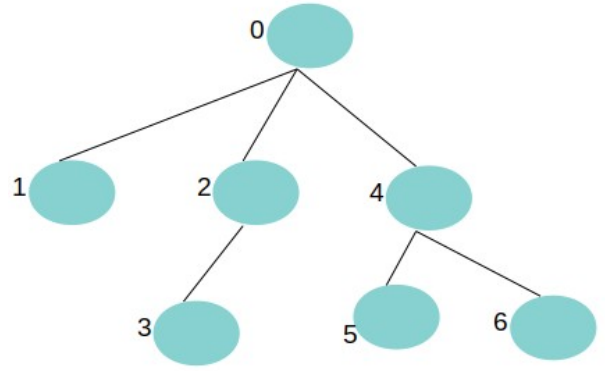
	1	0	0	(4)
-				
	1	0	0	(LSB of 4)
	0	0	0	→ 0 in decimal



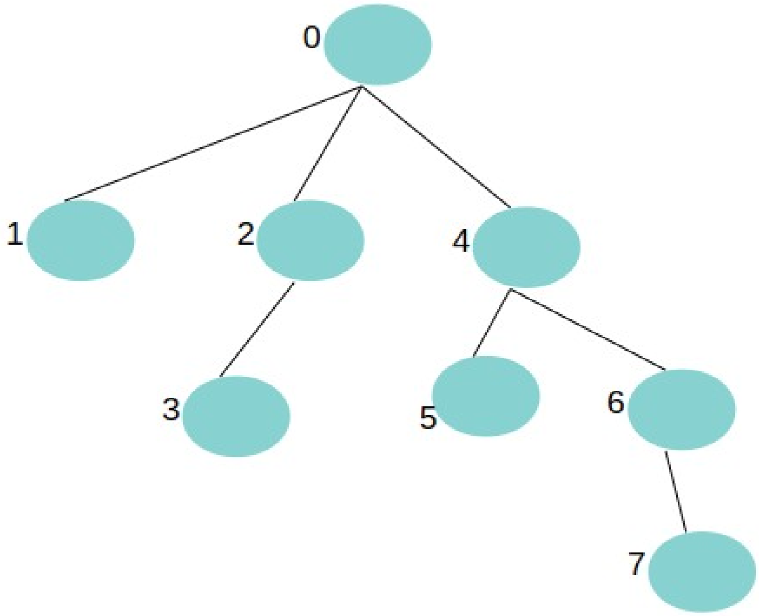
	1	0	1	(5)
-				
	0	0	1	(LSB of 5)
	1	0	0	→ 4 in decimal



	1	1	0	(6)
-				
	0	1	0	(LSB of 6)
	1	0	0	→ 7 in decimal



	1	1	1	(7)
-				
	0	0	1	(LSB of 7)
	1	1	0	→ 6 in decimal

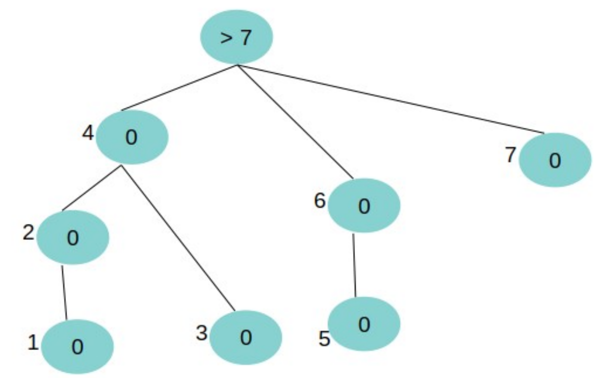
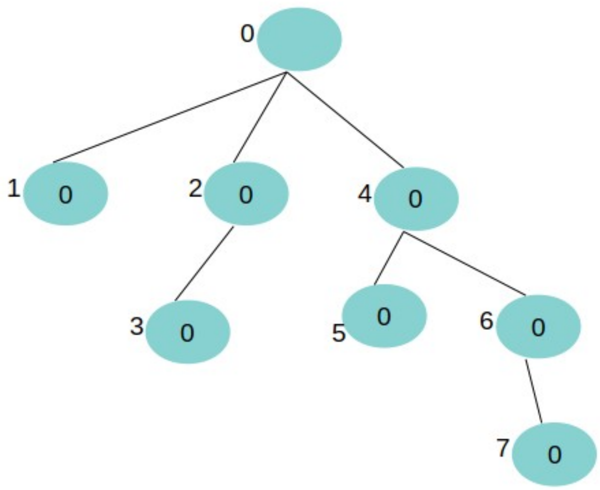


To get the sum of the first ***n*** elements: we start from the **$n+1^{th}$** element and we accumulate the values of all parents of the **$n+1^{th}$** node until reaching the dummy index (0).
For example: to compute ***the sum from 0 to 6 in the original array***, we start from the **7^{th}** node in the BIT tree, then the **6^{th}** and the **4^{th}** node.

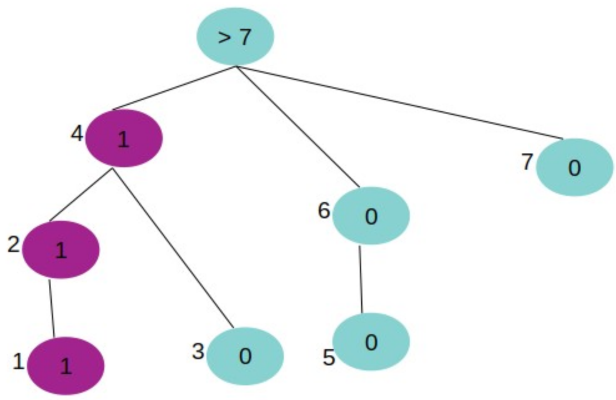
Example

A:	1	6	7	5	2	4	3
	0	*1*	*2*	*3*	*4*	*5*	*6*

tree:	0	0	0	0	0	0	0	0
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

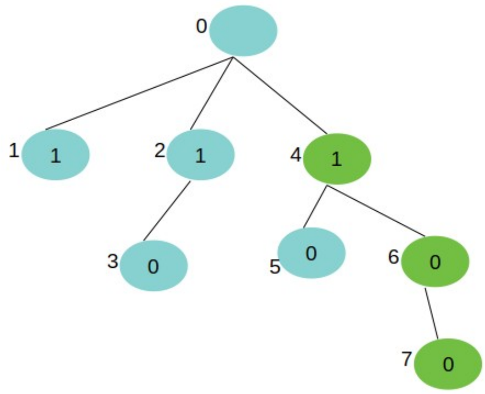
update	sum_range
	

- For input 1: [1]
  - update the tree: update(1, 1):

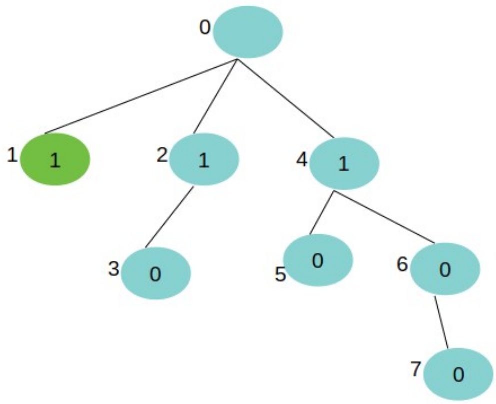


tree:	0	1	1	0	1	0	0	0
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

  - sum_range(2, 7) = sum(0, 7) – sum (0, 1)
    - sum(0, 7) = 1

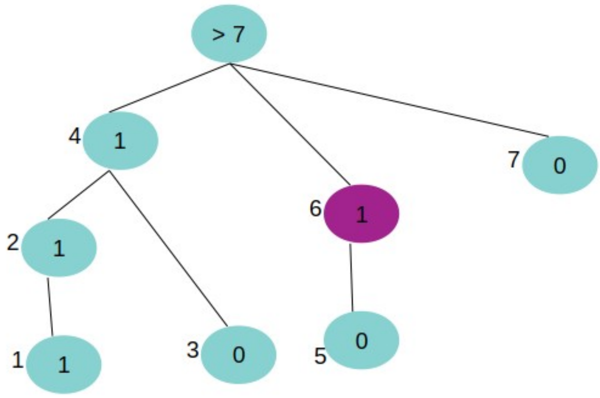


- sum(0, 1) = 1



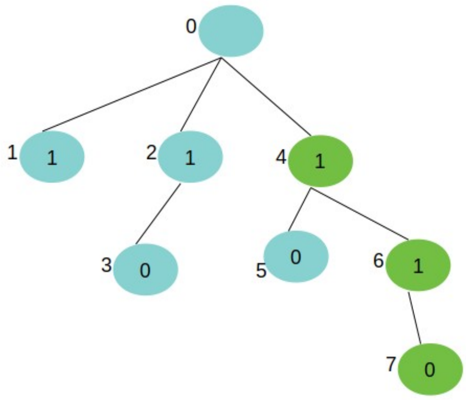
sum_range(2, 7) = 1 – 1 = 0

- For input 6: [1, 6]
    - update the tree: update(6, 1):

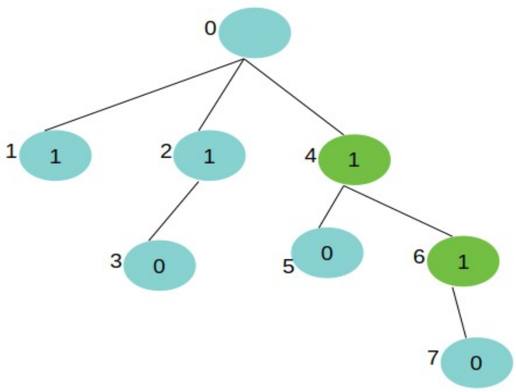


tree:	0	1	1	0	1	0	1	0
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

    - sum_range(7, 7) = sum(0, 7) – sum (0, 6)
        - sum(0, 7) = 2

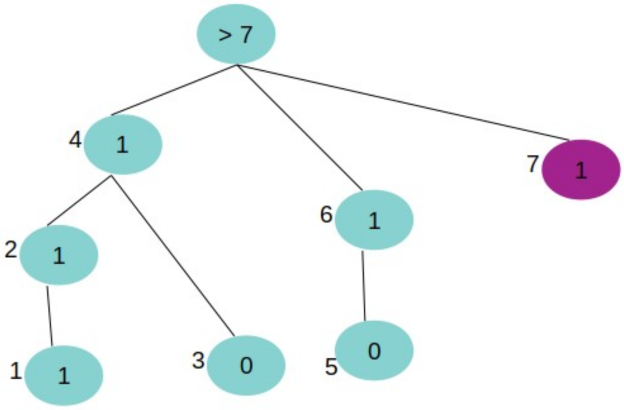


        - sum(0, 6) = 2



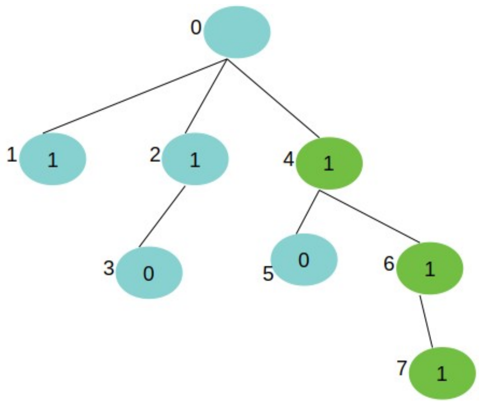
sum_range(7, 7) = 2 – 2 = 0

- For input 7: [1, 6, 7]
  - update the tree: update(7, 1):

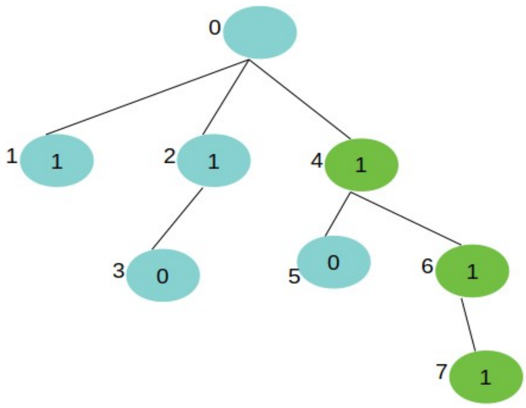


tree:	0	1	1	0	1	0	1	1
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

  - sum_range(8, 7) = sum(0, 7) – sum (0, 7)
    - sum(0, 7) = 3

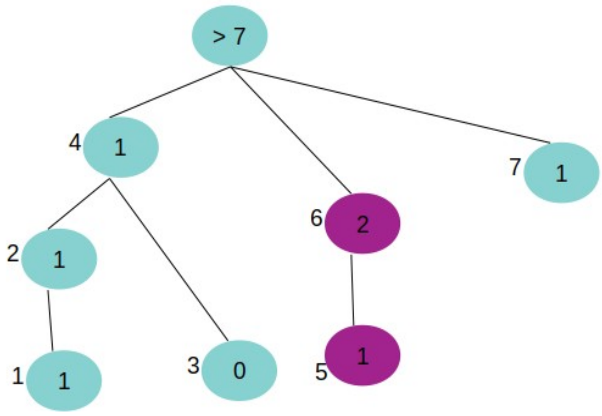


    - sum(0, 7) = 3



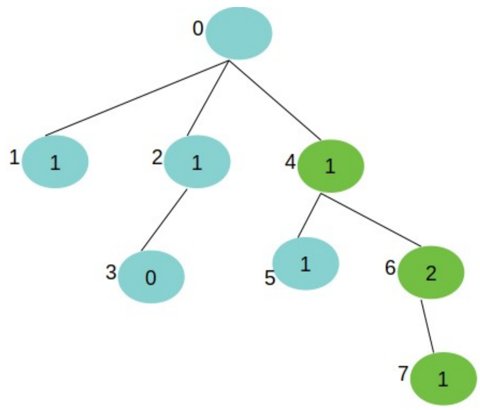
sum_range(8, 7) = 3 – 3 = 0

- For input 5: [1, 6, 7, 5]
    - update the tree: update(5, 1):

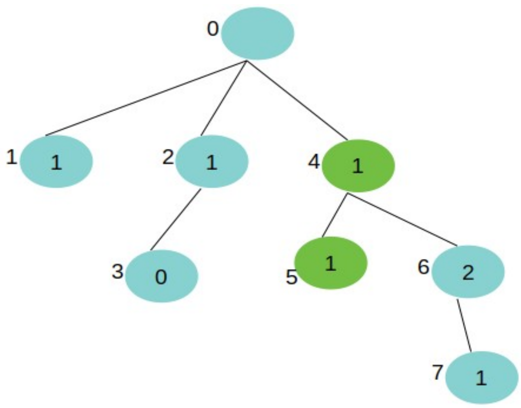


tree:	0	1	1	0	1	1	2	1
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

- sum_range(6, 7) = sum(0, 7) – sum (0, 5)
    - sum(0, 7) = 4

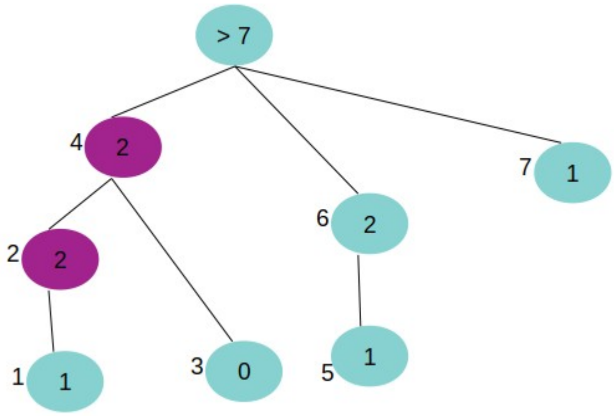


- sum(0, 5) = 2



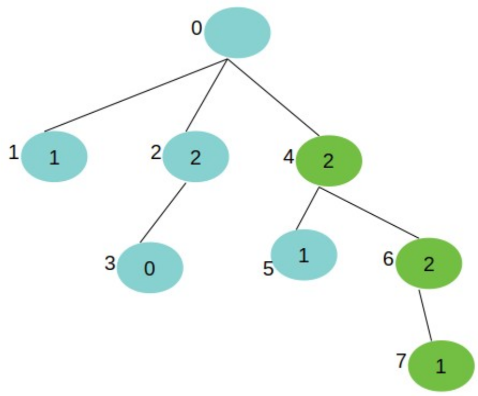
sum_range(5, 7) = 4 – 2 = 2

- For input 2: [1, 6, 7, 5, 2]
  - update the tree: update(2, 1):

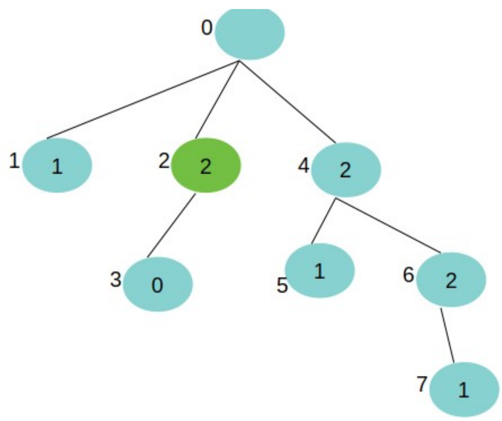


tree:	0	1	2	0	2	1	2	1
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

  - sum_range(3, 7) = sum(0, 7) – sum (0, 2)
    - sum(0, 7) = 5

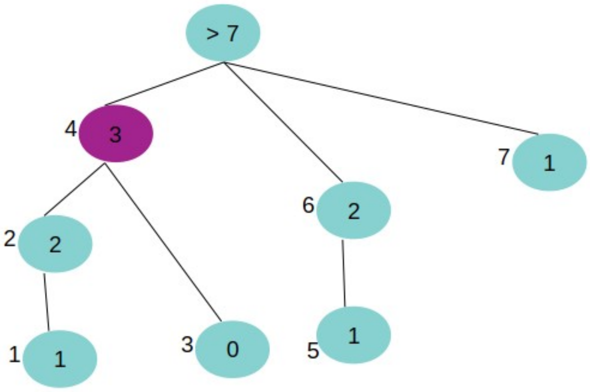


    - sum(0, 2) = 2



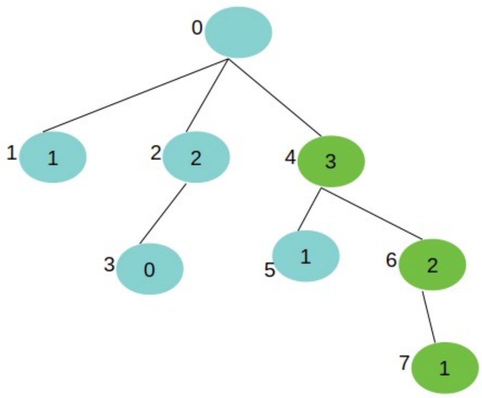
sum_range(3, 7) = 5 – 2 = 3

- For input 4: [1, 6, 7, 5, 2, 4]
    - update the tree: update(4, 1):

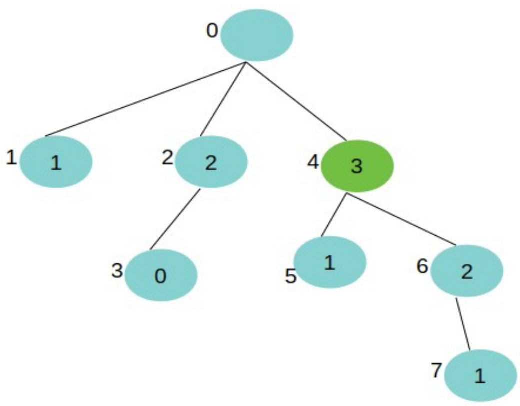


tree:	0	1	2	0	3	1	2	1
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

- sum_range(5, 7) = sum(0, 7) – sum (0, 4)
    - sum(0, 7) = 6

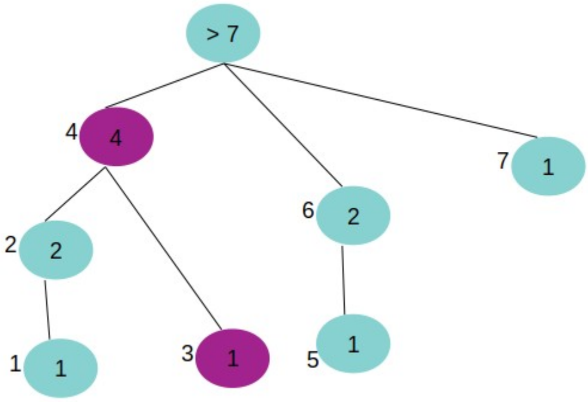


- sum(0, 4) = 3



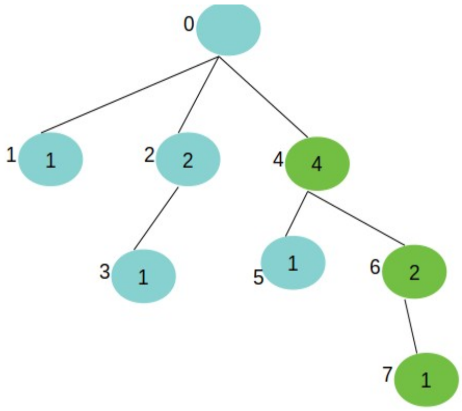
sum_range(5, 7) = 6 – 3 = 3

- For input 3: [1, 6, 7, 5, 2, 4, 3]
    - update the tree: update(3, 1):

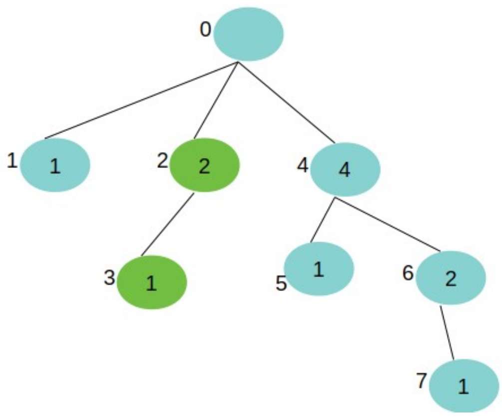


tree:								
	0	1	2	1	4	1	2	1
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*

- sum_range(4, 7) = sum(0, 7) – sum (0, 3)
    - sum(0, 7) = 7



- sum(0, 3) = 3



sum_range(4, 7) = 7 – 3 = 4
#inversions of the array = 0 + 0 + 0 + 2 + 3 + 3 + 4 = 12

C++11 Code:

```
/*
Calculate the number of inversion using a Fenwick tree.
Complexity of the problem: O(t * N * log N) (logarithmic)
Complexity of computing the #inversions: O(N log N)
*/

#include <bits/stdc++.h>

using namespace std;

int *tree = NULL;
int n;

void update(int i, int value){
  while(i<=n){
    tree[i]+=value;
    i+=(i&-i);
  }
}

int sum(int i){
  int sm=0;
  while(i>0){
    sm+=tree[i];
    i-=(i&-i);
  }
  return sm;
}

int range_sum(int i, int j){
  return sum(j)-sum(i);
}


int* convert(int *A){;
  int *temp = new int[n];
  for (int i=0; i<n; i++)
    temp[i] = A[i];

  sort(temp, temp+n);

  map<int, int> A1;
  for (int i = 0; i < n; ++i)
    A1.insert({temp[i], i + 1});

  for (int i=0; i<n; i++)
    A[i] = A1[A[i]];

  return A;
}
```

```
int main(){
   int t;
   cin >> t;

   while (t--) {
      cin >> n;

      int *A = new int[n];
      for (int i = 0; i < n; ++i) {
        cin >> A[i];
      }

      A = convert(A);

      tree = new int[n+1];
      fill_n(tree, n+1, 0);
      int inv_count = 0;

      // O(N * log N)
      for (int i = 0; i < n; ++i) {
        update(A[i], 1);
        inv_count += range_sum(A[i], n);
      }

      inv_count&1?cout<<"NO\n":cout<<"YES\n";

      delete[] A;
      delete[] tree;
   }

   return 0;
}
```

References

- http://www.cs.bham.ac.uk/~mdr/teaching/modules04/java2/TilesSolvability.html
- https://en.wikipedia.org/wiki/Parity_of_a_permutation
- https://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree-2/
- https://en.wikipedia.org/wiki/Inversion_(discrete_mathematics)