Fenwick Tree

Introduction

A Fenwick tree or binary indexed tree is a data structure used for **Range Query problems**.

Fenwick tree has the following benefits over segment trees:

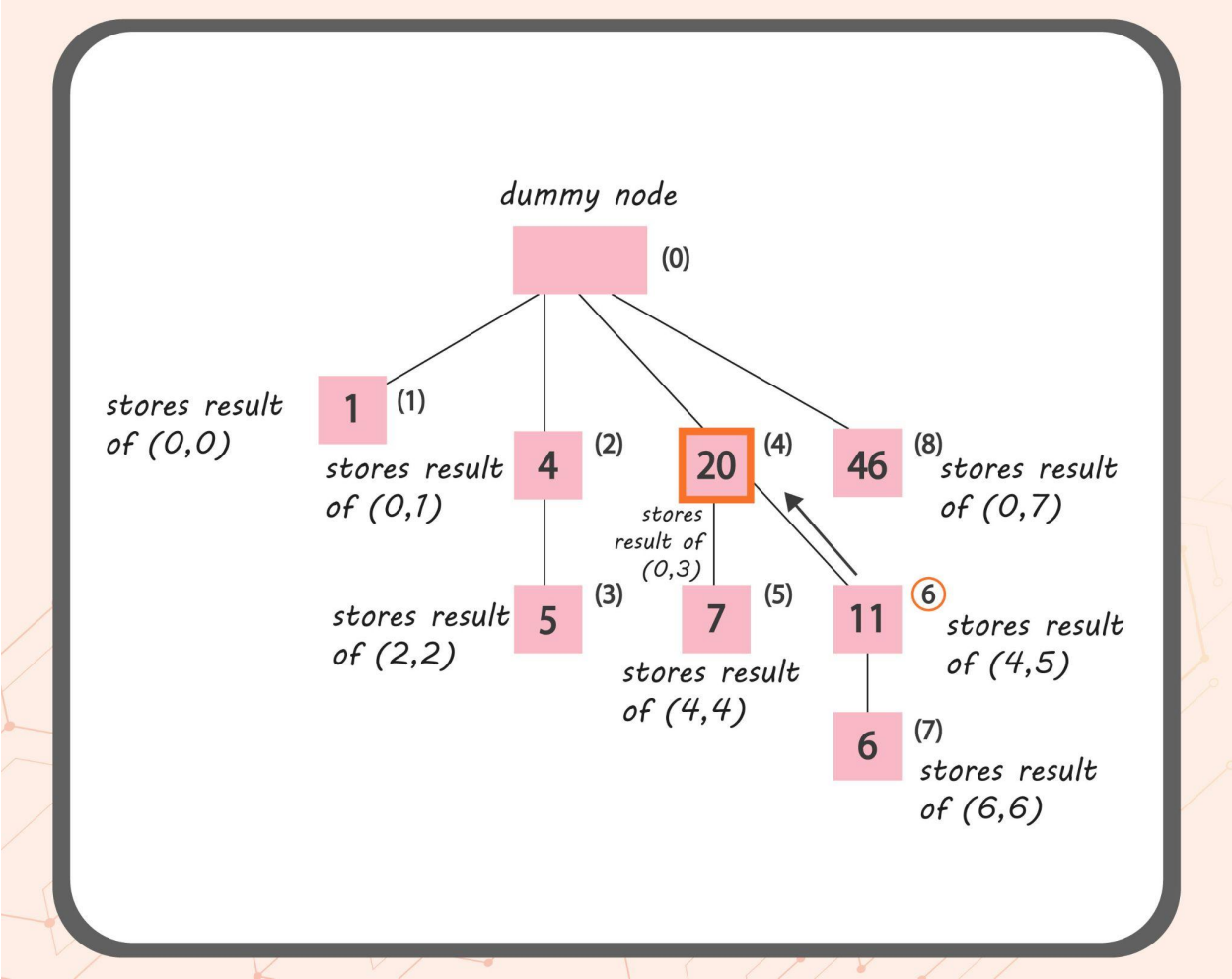
1. Code for the Fenwick tree is much simpler than that of the segment tree.
2. It takes very less space as compared to segment tree.

Let's see how a binary indexed tree is constructed.
Suppose we have an array, **Arr = [1, 3, 5, 11, 7, 4, 6, 9]** and we need to make a binary indexed tree from this.
First of all we should know that if an array has **N** elements, then the corresponding binary indexed tree will have **N + 1** nodes. The 0th node does not contain an answer to any query. In the above array there are 8 elements, so the corresponding binary indexed tree will have 9 nodes.

The first node is a dummy node which contains nothing. Thus, indexing actually starts from 1.

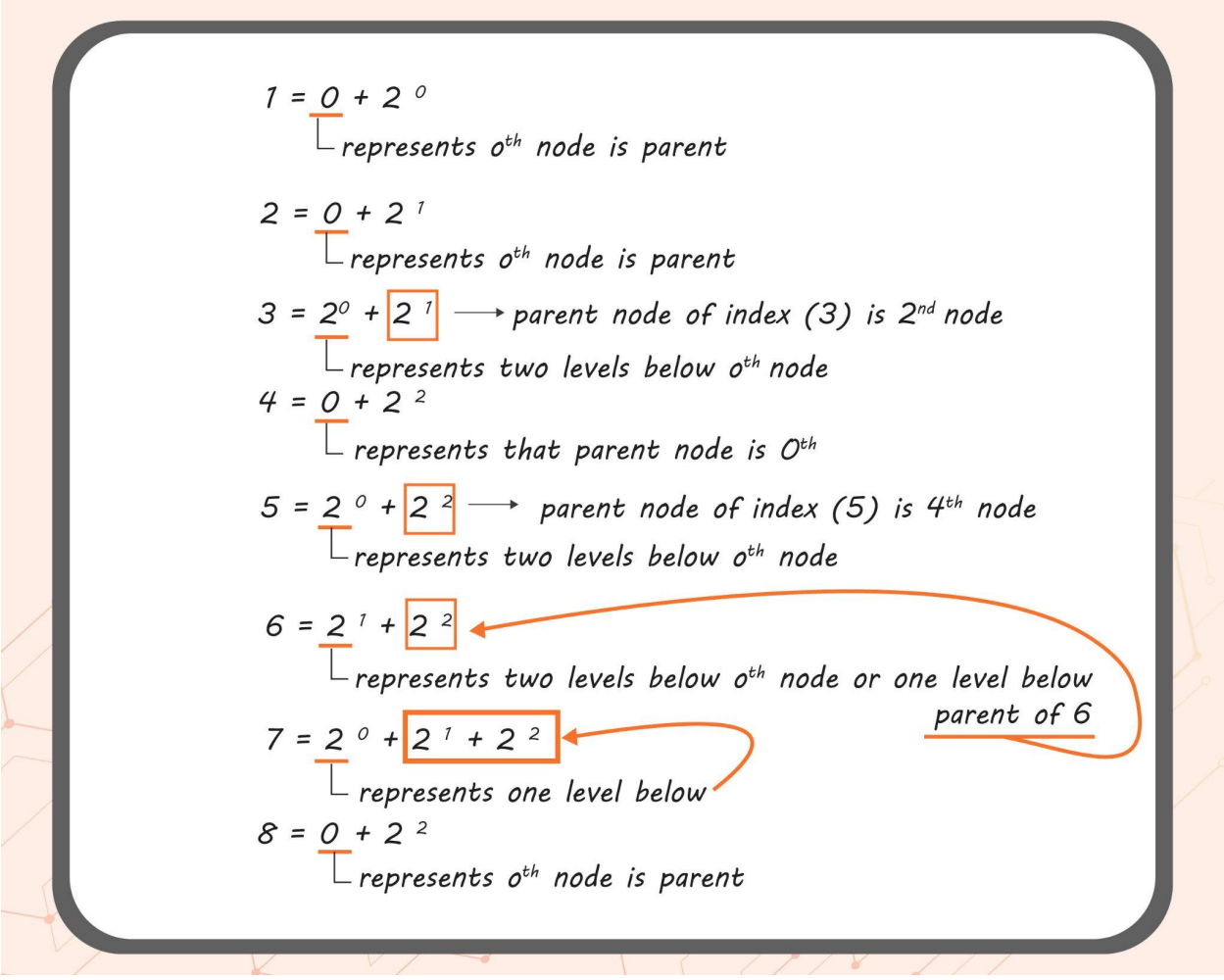
Let's see how a Fenwick Tree looks and how we query on it.

Example :
Suppose we have to find the result of the query (0, 5).
Since we have started our indexing from 1, we go to index 5 + 1 = 6 **($2^2 + 2^1$)**.
Thus, the 6th node stores the answer to the query (4, 5), and its parent node which is at index 4 stores the answer to the query (0, 3). Hence our final answer / result will be the sum of results at index 6 and its parent index 4.
So the final answer of the query (0, 5) = 11 + 20 = 31.

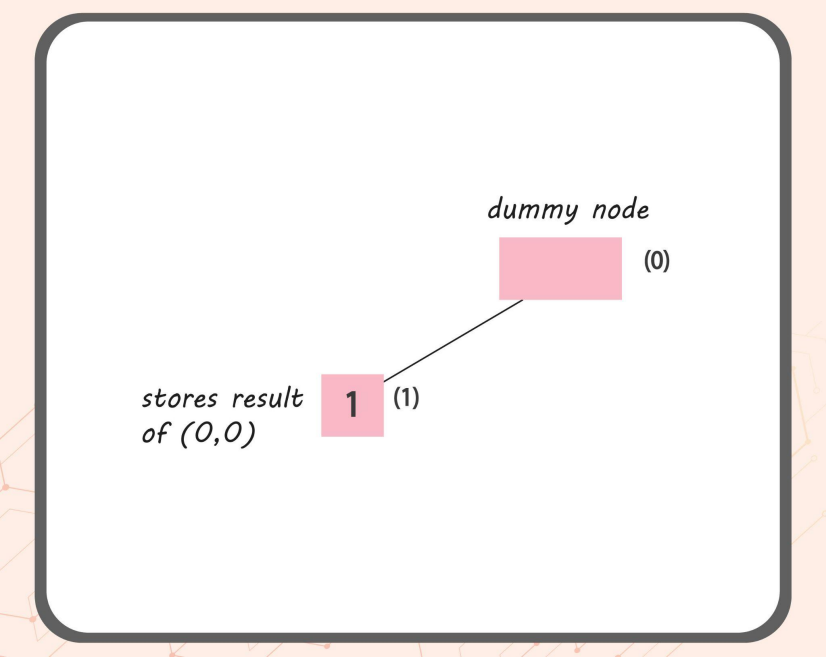

dummy node
(0)
stores result of (0,0)
1
(1)
stores result of (0,1)
4
(2)
stores result of (2,2)
5
(3)
stores result of (0,3)
20
(4)
stores result of (4,4)
7
(5)
46
(8)
stores result of (0,7)
11
6
stores result of (4,5)
6
(7)
stores result of (6,6)

Now, let's take a look behind the concept of building a Fenwick Tree.

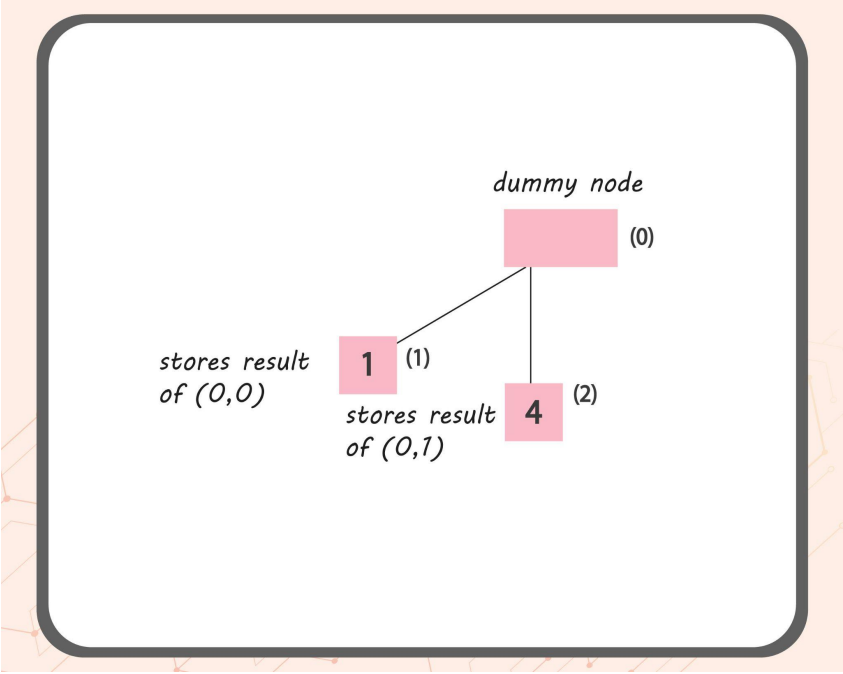
We can represent the indices in the form of powers of 2 such as -



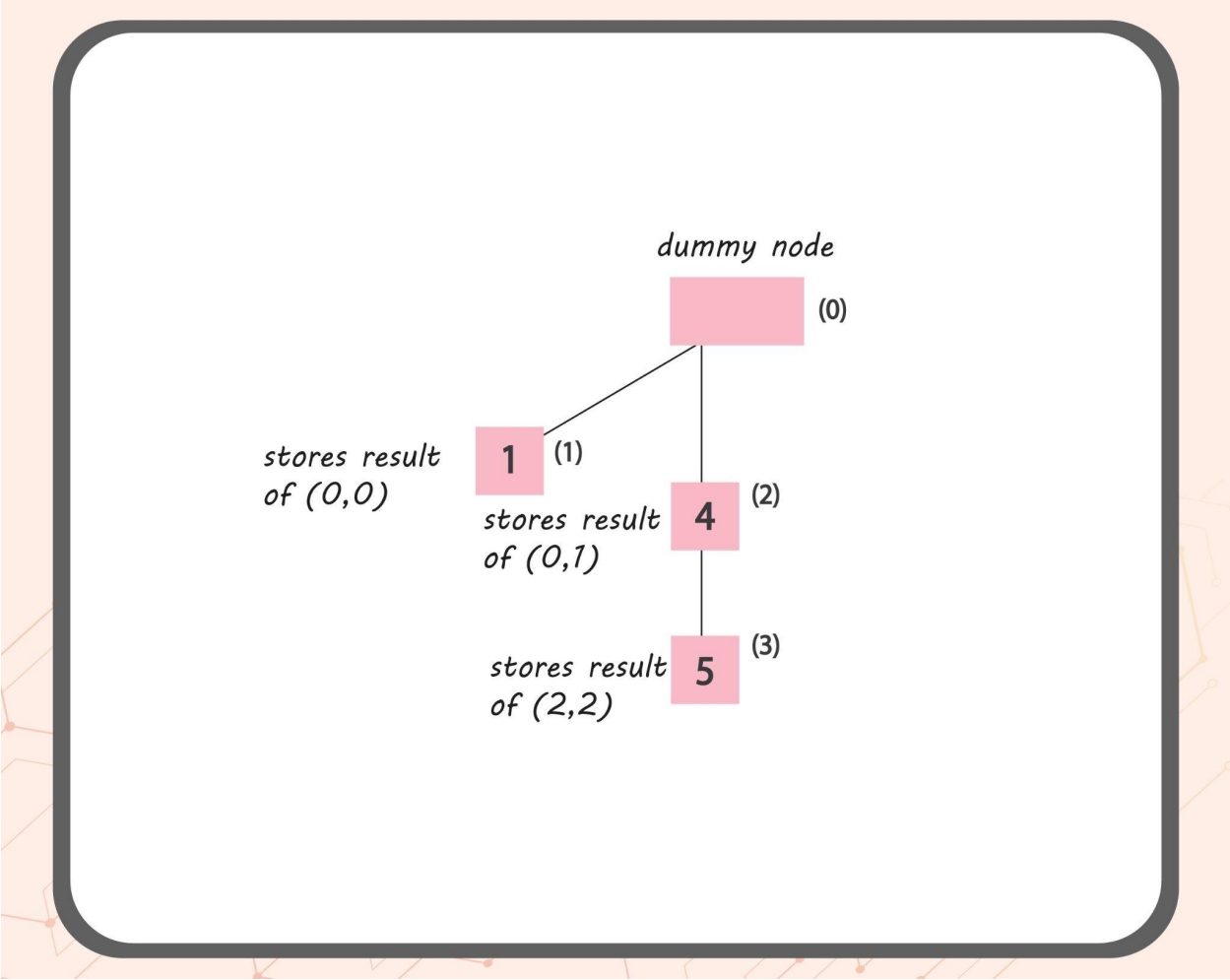
If we look at index **1 = 2^0 + 0,** the 0 represents the level or height at which the parent node of index 1 is. That is, the parent node of index 1 is the 0^{th} node (which is empty). Index 1 stores the result of (0,0).



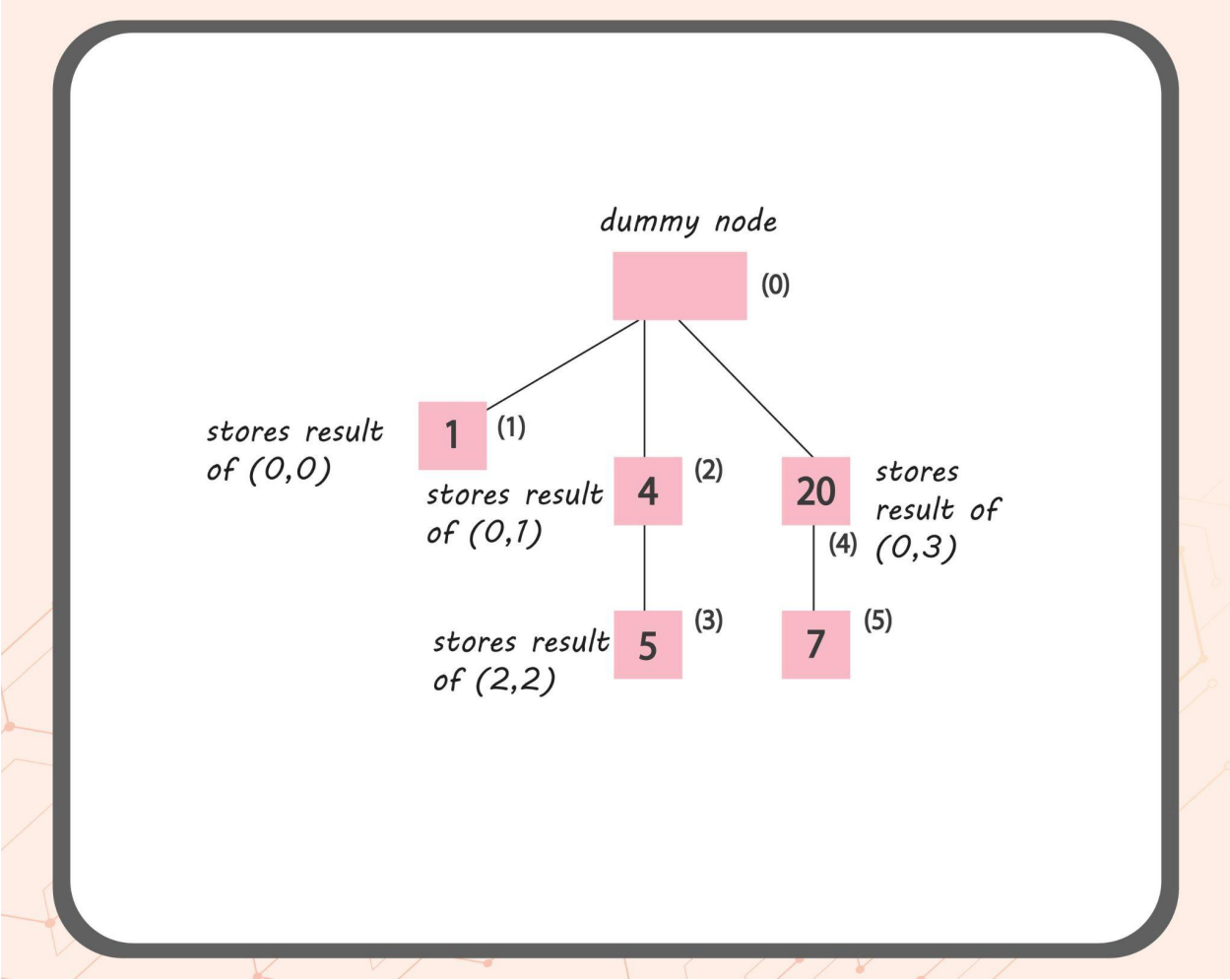
Now, index **2 = 2^1 + 0**. The 0 represents that it is the direct child of the 0^{th} node and 2^1 represents that it will store the result of (0, 1).



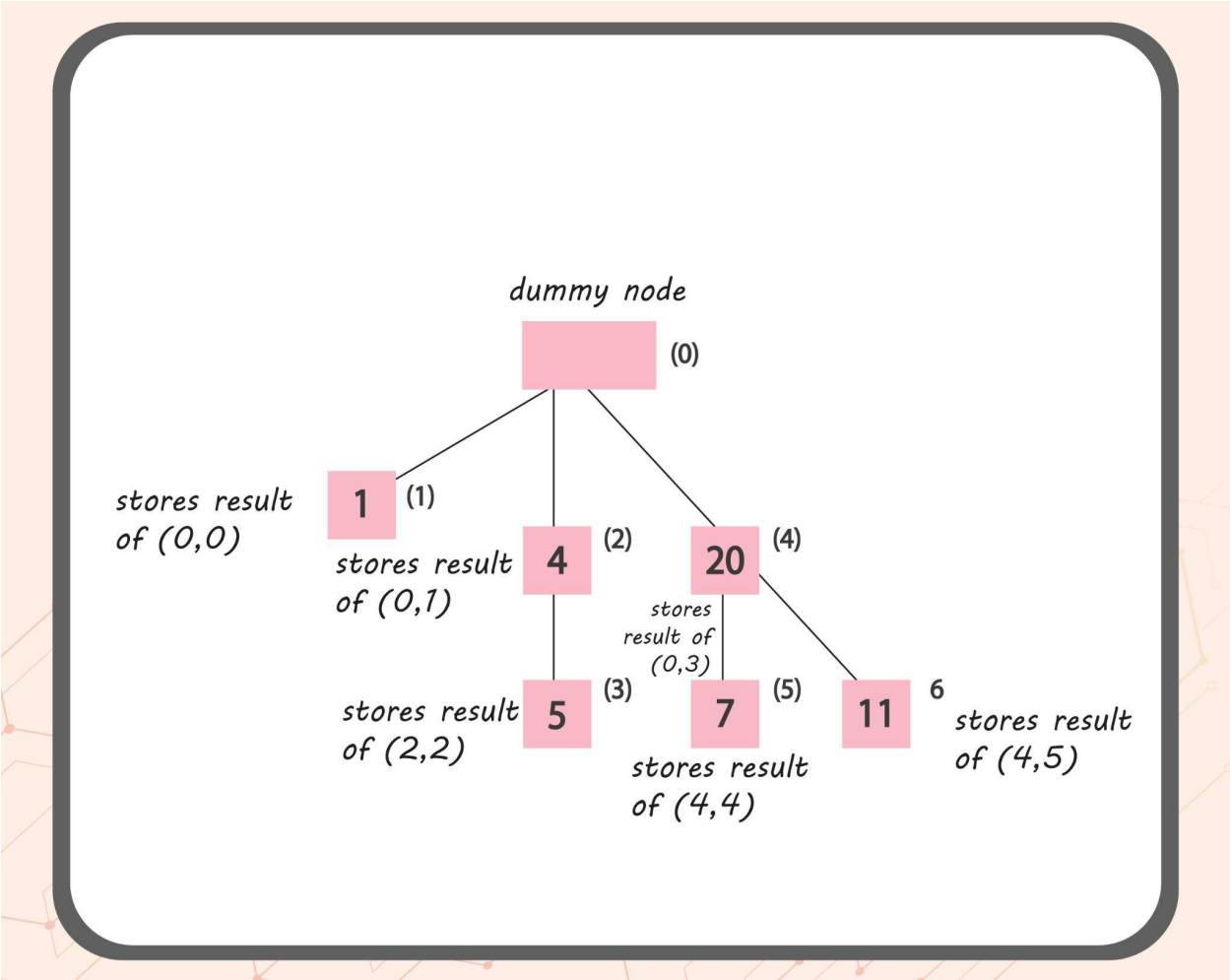
Index **3 = $2^1 + 2^0$**. Here, 2^1 represents that the parent of index 3 will be index 2 and since the result of (0,1) is stored at index 2, therefore index 3 will store the result of (2, 2).



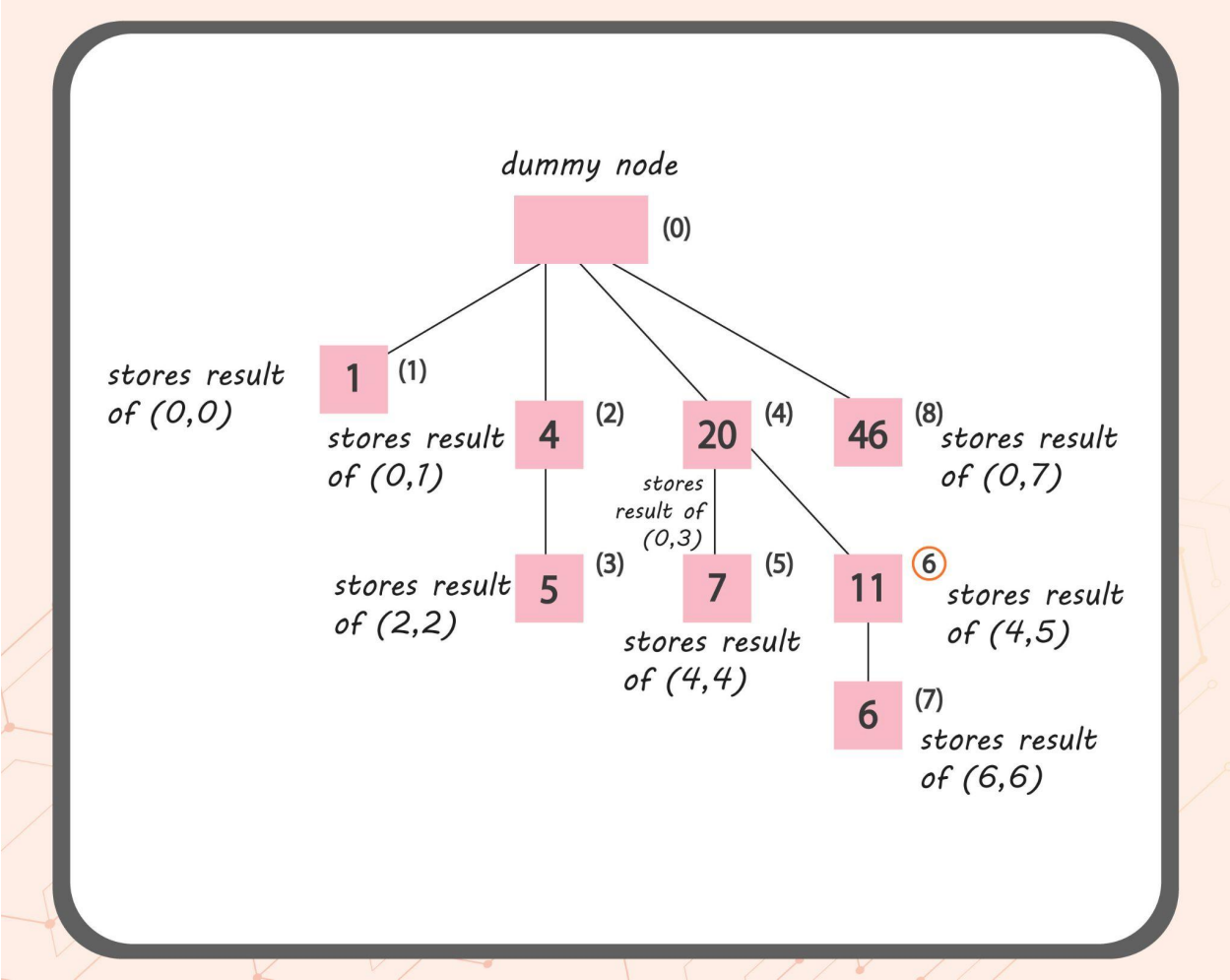
Index **4 = 2^2 + 0**, so 0 represents that index 4 will be a direct child of the 0^{th} node and it will store the result of (0, 3).



Index **5 = 2^2 + 2^0**. 2^2 represents index 4, therefore index 5 will be the child of index 4 and will store the result of (4, 4).
Index **6 = 2^2 + 2^1**. Since 2^2 = index 4, then index 6 will be a child of index 4 and will store the result of (4, 5).



Similarly we can add the nodes for the remaining indices and our final tree will look something like -



Query on Fenwick tree

Example: Suppose we have to find the result of the query (0, 6).
Since we have started our indexing from 1 we go to index 6 + 1 = 7 **($2^2 + 2^1 + 2^0$)**.
If we subtract 2^0 from the above we get $2^2 + 2^1$ (Index 6), which is the parent of index 7.
Similarly, if we subtract 2^1 from the above result, we get 2^2 (Index 4), which is the parent of index 6.

Thus, we can conclude that, by removing / subtracting the smallest power of 2 (or the rightmost set bit), we reach the parent of the current node.

Update on Fenwick tree

Suppose we have to update the value at index 2 in the array **Arr = [1, 3, 5, 11, 7, 4, 6, 9]**. This essentially means that index 3 of the fenwick tree will be updated. But there will also be updates at index 4 and index 8 because, since 4 (2^2) and 8 (2^3) are powers of 2, they will be the direct children of the 0th node and will contain results to the queries with start index 0 thus affected by a change in the index 2 of the array .

Let's take a look at the code for building, updating and query on a fenwick tree.

Code:

```
#include<bits/stdc++.h>
using namespace std;


void update(int index,int value,int* BIT,int n){

    for(;index <= n;index += index&(-index)){
        BIT[index] += value;
    }
```

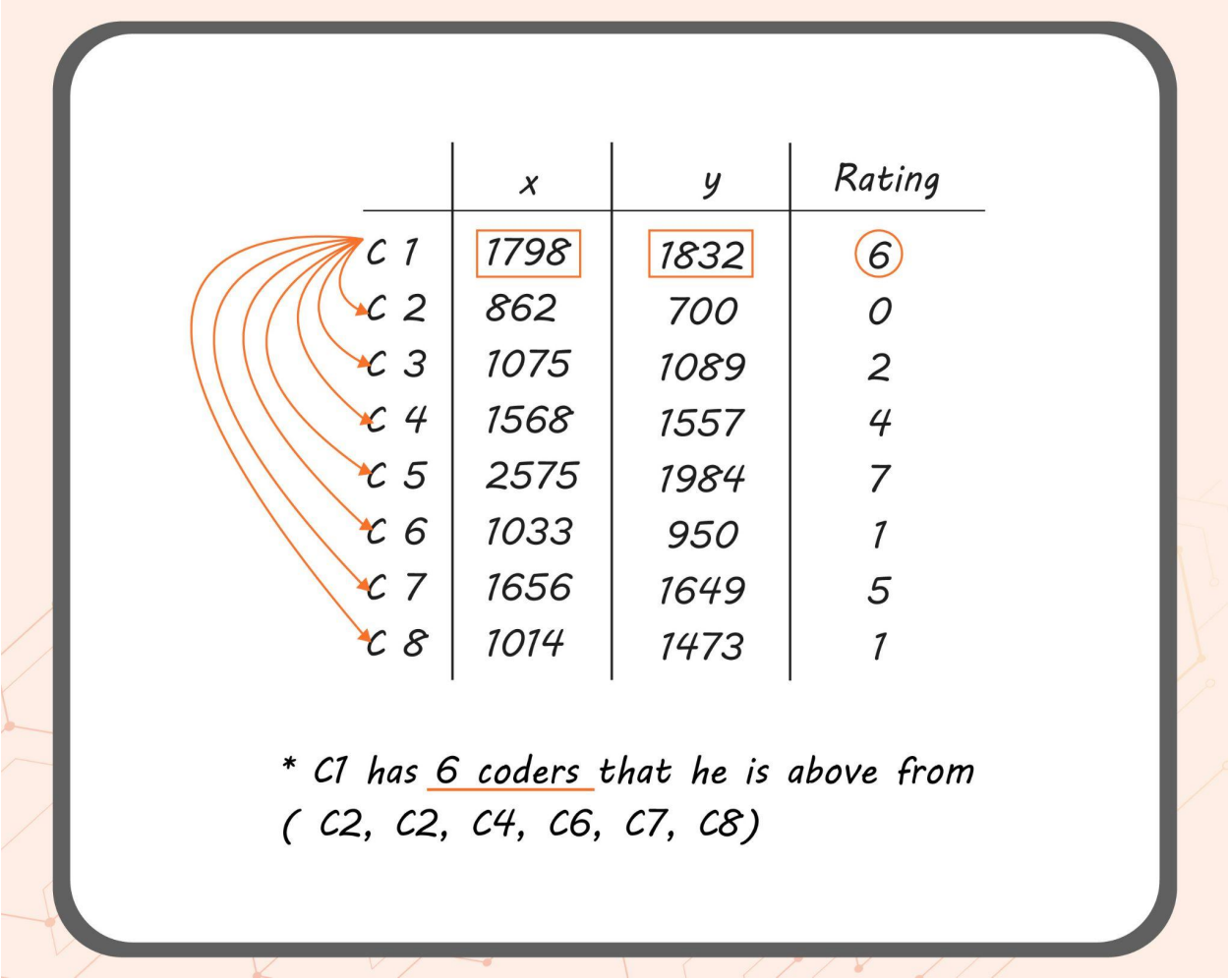
```
}

int query(int index,int* BIT){
    int sum=0;
    for(;index >0;index -= index&(-index)){
        sum += BIT[index];
    }
    return sum;
}
int main(){

    int n;
    cin >> n;

    int* arr = new int[n+1]();
    int* BIT = new int[n+1]();

    for(int i=1;i<=n;i++){
        cin >> arr[i];
        update(i,arr[i],BIT,n);
    }

    cout << "Sum of first 5 elements " << query(5,BIT) <<endl;
    cout << "Sum of elements from 2 index to 6 index " << query(6,BIT) -
query(1,BIT) <<endl;
    return 0;
}
```

Coder Rating

Problem Statement: We have different coders C1, C2, C3 and so on. Their ratings depend on two factors **x** and **y**. For every coder we have to find out how many coders he/she is better from.

Explanation:

Example:



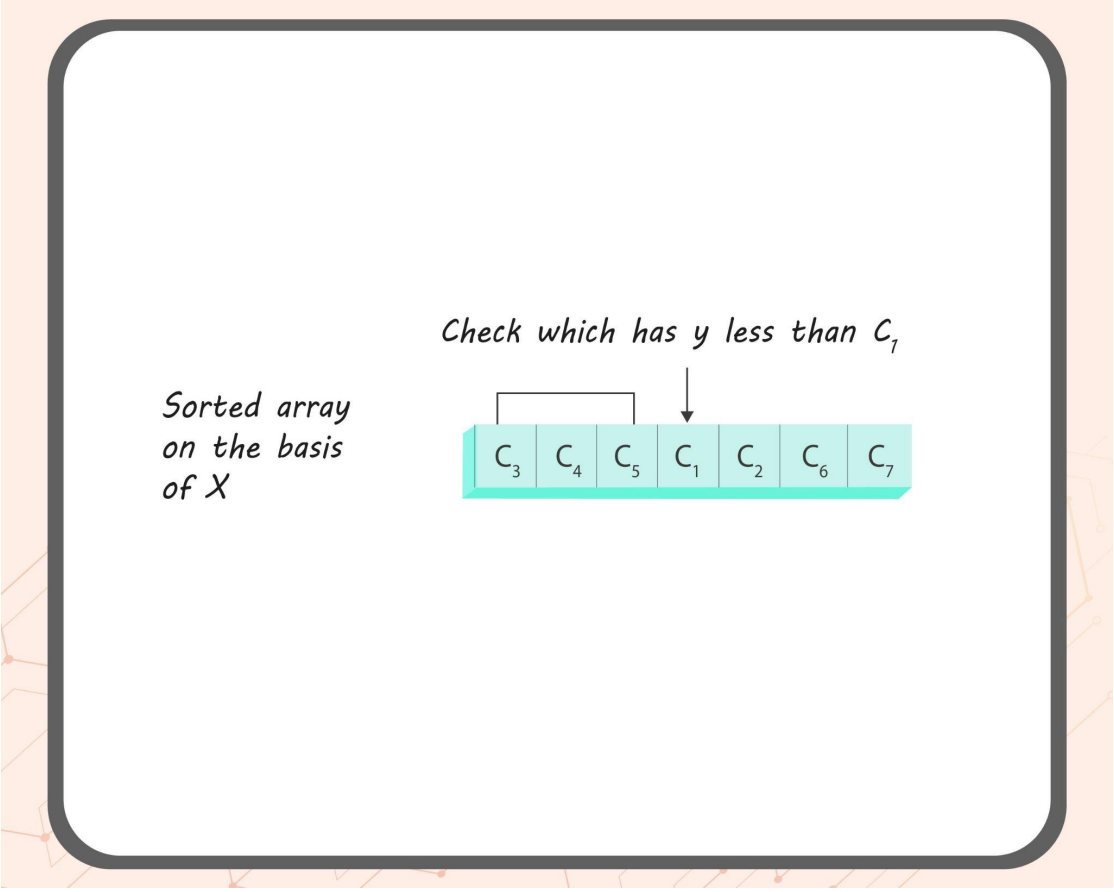
For a coder C1 to be better than C2, there can be three possibilities:

1. **C1.x > C2.x**
   C1.y > C2.y

2. **C1.x = C2.x**
   C1.y > C2.y

3. **C1.x > C2.x**
   C1.y = C2.y

If we think about sorting the array then we can do it only on the basis of one parameter. So let's say if we sort the array on the basis of **x** then we can see for every coder Ci that before him/her how many coders have **y** lesser than than Ci. For equality cases we will have to add an additional check.



Code:

```
#include<bits/stdc++.h>
using namespace std;

struct coder
{
    int x,y,index;
};
int bit[100001];

void update(int y){

    for(;y<=100000;y+= y&(-y)){
        bit[y]++;
    }
}

int query(int y){
    int value = 0;
    for(;y>0;y-= y&(-y)){
        value += bit[y];
    }
    return value;
}
bool operator < (coder A, coder B){
    if(A.x == B.x){
        return A.y < B.y;
    }
    return A.x < B.x;
}
int main(){

    int n;
    cin >> n;
    coder arr[n];

    for(int i=0;i<n;i++){
        cin>>arr[i].x >> arr[i].y;
        arr[i].index = i;
```

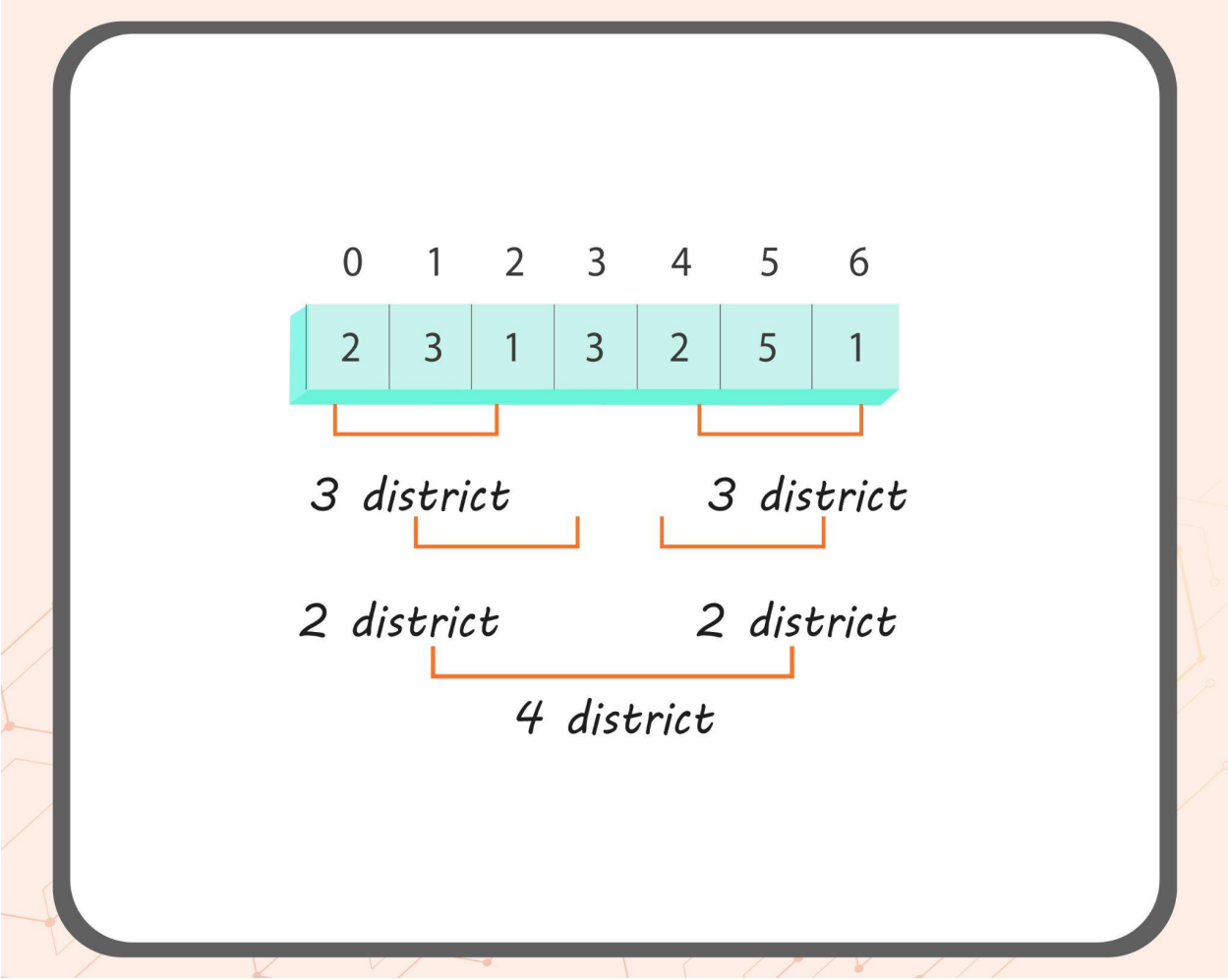
```
    }

    sort(arr,arr+n);
    int ans[n];
    for(int i=0;i<n;){
        int endIndex = i;

        while(endIndex <n && arr[endIndex].x == arr[i].x && arr[endIndex].y
== arr[i].y){
            endIndex++;
        }
        // query

        for(int j=i;j<endIndex;j++){
            ans[arr[j].index] = query(arr[j].y);
        }

        for(int j=i;j<endIndex;j++){
            update(arr[j].y);
        }

        i = endIndex;

        // update
    }

    for(int i=0;i<n;i++){
        cout << ans[i] <<endl;
    }


    return 0;
}
```

Distinct Query Problem

Problem Statement: Given a sequence of n numbers a1, a2, ..., an and a number of d-queries. A d-query is a pair (i, j) (1 ≤ i ≤ j ≤ n). For each d-query (i, j), you have to return the number of distinct elements in the subsequence ai, ai+1, ..., aj.

Explanation:

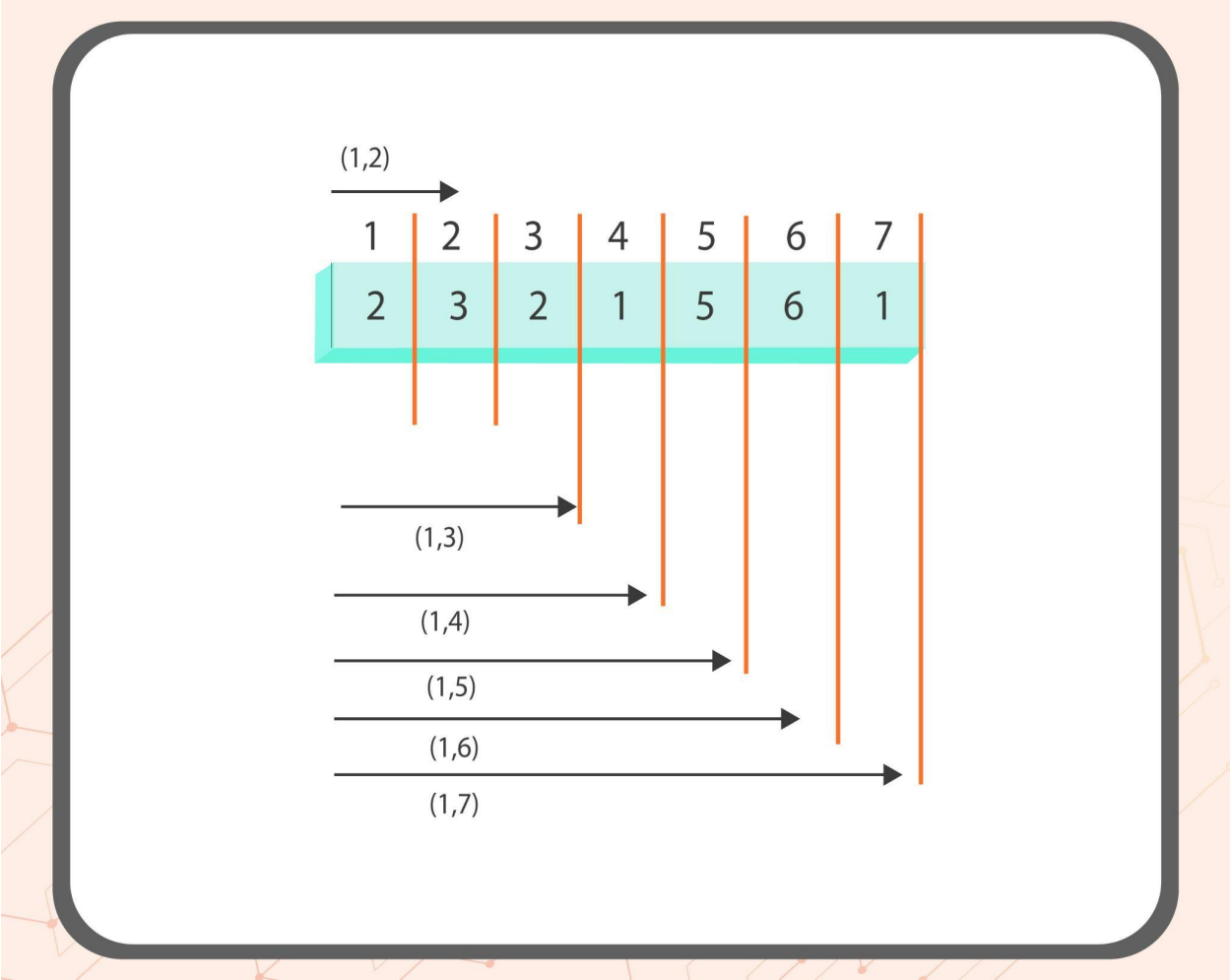
Example:



To solve this problem we will use the concept of **Offline Programming.**

Offline Programming VS Online Programming

In Online programming we calculate the result as soon as the query arrives whereas in Offline programming we store the results of the different possible queries beforehand.



We will use a Binary Indexed Tree to store the results of queries.

We will store the result of the coming queries on the basis of the end index of the query. That is, first of all we will store the result of the query (1, 1) then for (1, 2), then for (1, 3) and so on until the query (1, 7). For this we will maintain a query array and sort this on the basis of the end index of the queries.

Code:

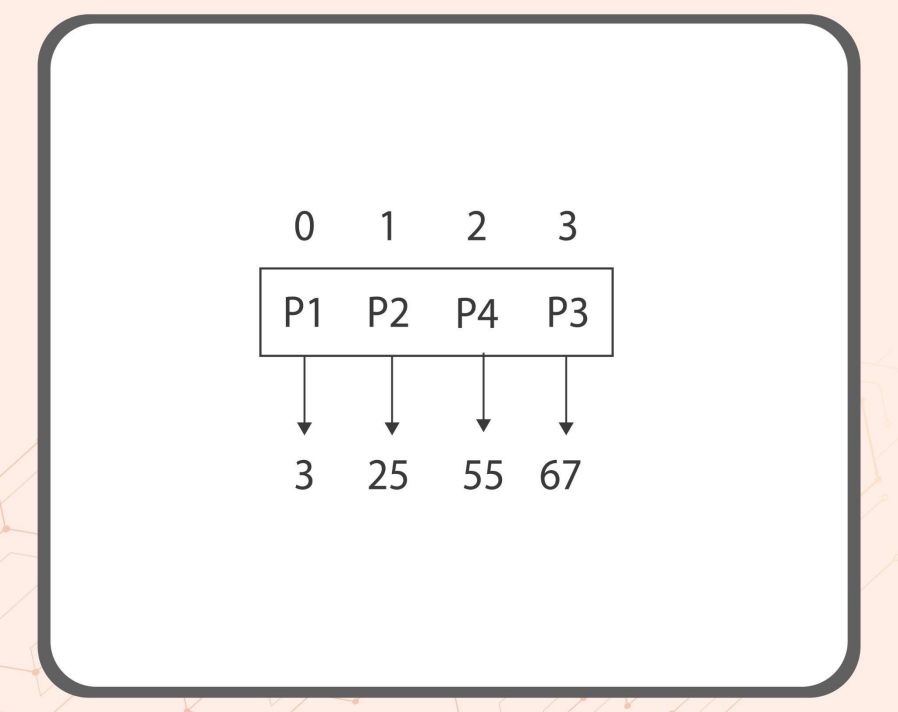
```
#include<bits/stdc++.h>
using namespace std;

struct coder
{
    int first,second,index;
}query[200000];

bool operator < (coder A, coder B){

    return A.second < B.second;
}

int bit[30001];
void update(int index,int value,int n){

    for(;index<=n;index += index &(-index)){
        bit[index] += value;
    }
}
int last[1000001];
int arr[30001];
int value(int index){

    int res= 0;
    for(;index >0 ;index -= index & (-index)){
        res += bit[index];
    }
    return res;
}

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


    for(int i=1;i<=n;i++){
        cin >> arr[i];
```

```
    }

    int q;
    cin>>q;

    for(int i=0;i<q;i++){
        cin>>query[i].first>>query[i].second;
        query[i].index = i;
    }

    sort(query,query+q);



    memset(last,-1,sizeof(last));



    int k=0;
    int ans[q];


    int total = 0;
    for(int i=1;i<=n;i++){


        if(last[arr[i]] != -1){
            update(last[arr[i]],-1,n);
        }else{
            total++;
        }

        update(i,1,n);
        last[arr[i]] = i;


        while(k<q && query[k].second == i){

            ans[query[k].index] = total - value(query[k].first-1);
            k++;
```

```
        }

    }


    for(int i=0;i<q;i++){
        cout << ans[i] <<endl;
    }
    return 0;
}
```

Coordinate Compression

Suppose we are given some points **P1 (3), P2 (25), P3 (67), P4 (55)** along with their coordinates. Now, the ordering of P1, P2, P3, P4 will be 0, 1, 3, 2 respectively.



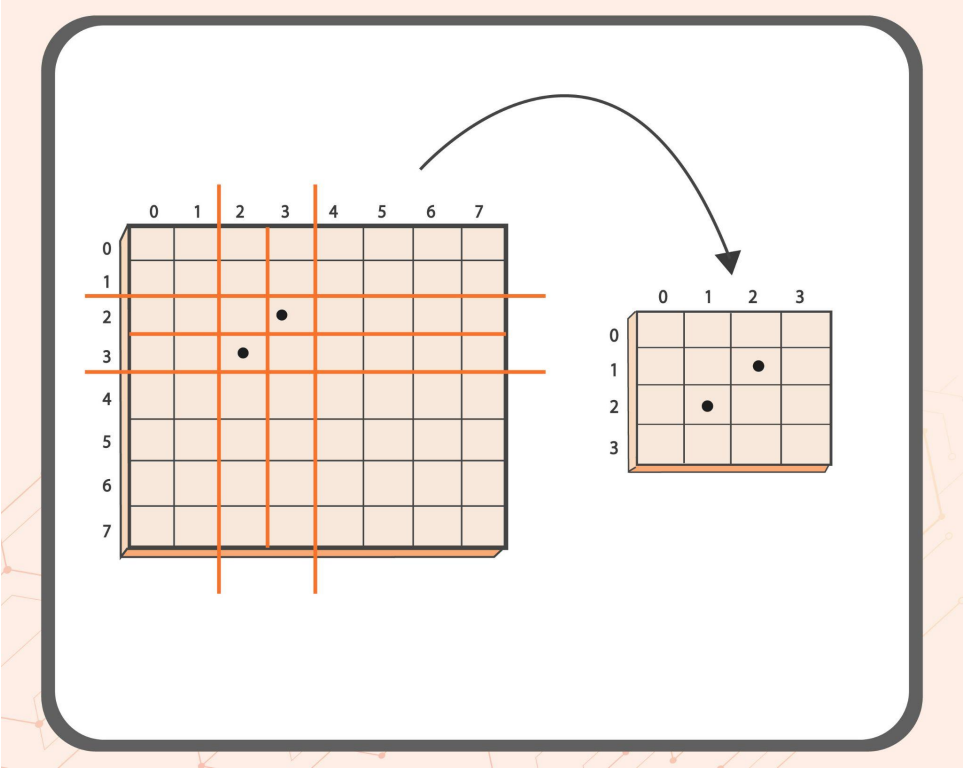
We will take input of the coordinates and then sort them and give the respective ranking to them.
We can also store the ranks in a map or an array.
Time Complexity of this will be the same as that of sorting that is **O(Nlog(N)**.

Let us see how the above concept can be used in coordinate compression.

Consider a problem in which we are given a grid with two obstacles in it and we have to somehow deal with the obstacles.

Note: This isn't an actual problem so how to deal with the obstacles are not our concern. We just have to find a way to reduce the overall effort and complexity of the solution.

The empty rectangles are kind of redundant so the above grid can be effectively reduced to



This can be achieved using the concept of mapping which we discussed above in the beginning of this section.

Mapping rows from initial grid to new grid :

- **Rows**
  - **0, 1 → 0**
  - **2 → 1**
  - **3 → 2**
  - **4, 5, 6, 7 → 3**

- **Columns**
  - **0, 1 → 0**
  - **2 → 1**
  - **3 → 2**
  - **4, 5, 6, 7 → 3**