Static Range Queries

Agenda

1. Frequency Arrays
2. Prefix & Suffix Arrays
    a. Prefix Sum
    b. Suffix Sum
    c. Range Sum Queries
    d. Prefix Minimum
3. Partial Sum

Static range queries are types of queries that ask for some information about some range or some elements in an array without applying any updates to the array elements. Some techniques that will help you answer these types of queries are: Frequency Arrays, Prefix and Suffix Arrays.

1. Frequency Arrays

Problem: Given an array of integers of size ***N***, count the number of occurrences of each array element.

Naive Solution:

One way is to go through all array elements and for each element ***X*** loop over array elements again to count its frequency.

Time Complexity: O(N^2)

This would be fine if this complexity fits in our constraints. But Can we do better?

Efficient Solution:

We can construct a frequency array that would have the frequency of each array element, so whenever you need to get the frequency of any element you can get it in only O(1) for each query.

So, how to construct it?

Each index in the frequency array will represent an element, and the value stored in that index will be the frequency of that element.

Array **A**:

Index	0	1	2	3	4	5	6	7	8
Element	8	5	4	6	5	8	5	4	2

Frequency Array **Freq**:

Element	0	1	2	3	4	5	6	7	8
Frequency	0	0	1	0	2	3	1	0	2

Now that we have the frequency array constructed, We can get the frequency of any element in O(1) time complexity:

- Frequency of 5 is Freq[5] = 3.
- Frequency of 8 is Freq[8] = 2.
- Frequency of 2 is Freq[2] = 1.
- Frequency of 7 is Freq[7] = 0.
- And so on..

Code:

```
const int MAX_VAL = 1e6 + 1;
int main() {
   int n, Freq[MAX_VAL] = {};
   cin>> n;
   vector<int> A(n);
   for (int i = 0; i < n; ++i) {
       cin>> A[i];
       Freq[A[i]]++;
   }
   for (int i = 0; i < n; ++i) {
       if (Freq[A[i]] > 0) {
           cout << "Frequency of " << A[i]
                         << " is " << Freq[A[i]] << ".\n";
           // to avoid printing any element more than once.
           Freq[A[i]] = 0;
       }
   }
}
```

Example:

A pangram sentence is a sentence using every letter of a given alphabet at least once. Given a string ***S*** of lowercase english letters, determine if string ***S*** is a pangram.

S = “lemrawhqcdipuhssqgcbvazjfnooyktx”
Output: pangram

S = “castzeyldbxmoop”
Output: not a pangram

Idea:

Can we use Frequency Array to count characters?
Yes, the idea is simple. We need to represent those letters using numbers, so ‘a’ = 0, ‘b’ = 1, ‘c’ = 2, ‘d’ = 3, … , ‘z’ = 25. We can get the corresponding value of each letter by subtracting character ‘a’ from each one as following:

- Let char ***C*** = ‘a’, corresponding value of ***C*** is ***C*** - ‘a’ = ‘a’ - ‘a’ = 0.
- Let char ***C*** = ‘b’, corresponding value of ***C*** is ***C*** - ‘a’ = ‘b’ - ‘a’ = 1.
- Let char ***C*** = ‘c’, corresponding value of ***C*** is ***C*** - ‘a’ = ‘c’ - ‘a’ = 2.
- …
- Let char ***C*** = ‘z’, corresponding value of ***C*** is ***C*** - ‘a’ = ‘z’ - ‘a’ = 25.

String **S**:

Index	0	1	2	3	4	5	6	7	8	9
Element	‘a’	‘b’	‘d’	‘e’	‘f’	‘d’	‘e’	‘a’	‘e’	‘g’

Frequency Array **Freq**:

Element	0 (‘a’)	1 (‘b’)	2 (‘c’)	3 (‘d’)	4 (‘e’)	5 (‘f’)	6 (‘g’)
Frequency	2	1	0	2	3	1	1

Code:

```cpp
const int MAX_VAL = 26;
int main() {
   string s;
   int Freq[MAX_VAL] = {};
   cin>> s;
   for (int i = 0; i < s.size(); ++i) Freq[s[i] - 'a']++;
   for (int i = 0; i < 26; ++i) {
       if (Freq[i] == 0) {
           cout<< "not a pangram\n";
           return 0;
       }
   }
   cout<< "pangram\n";
}
```

Although using a frequency array is efficient, it has some restrictions regarding the types of data it counts: It can be any types of data that can be represented within a positive range that fits the maximum size of the arrays, such as:

- Positive integers
- Negative integers with shifting trick
- Characters

So it would not work with:

- Large integers
- Floating-point numbers
- Any other type of data that is not numeric, such as strings.

To be able to count these types of data, we can use map/unordered_map instead of frequency array.

Example:

Given **N** strings, print the string with the maximum number of occurrences.

Code:

```
int main() {
    int n, mx = 0;
    string mostOccuring;
    cin>> n;
    unordered_map<string, int> freq;
    for (int i = 0; i < n; ++i) {
        string s;
        cin>> s;
        freq[s]++;
    }
    for (auto &str: freq) {
        if (str.second > mx) {
            mx = str.second;
            mostOccuring = str.first;
        }
    }
    cout<< mostOccuring;
}
```

Problems:

- Andryusha and Socks
- Letter
- Good Array
- ZgukistringZ
- Registration system
- Conformity

2. Prefix & Suffix Arrays

There is some information that you may need to get about certain ranges in an array, such as range sum/multiplication/XOR, minimum/maximum value in a prefix or a suffix.

You can easily do this by iterating over the required ranges and compute this information. However, when you need to do this for many queries, the time complexity becomes O(#Queries * ArraySize), which is not efficient for a large number of queries. Therefore, we need a more efficient method for retrieving this information.

Prefix Array: an array holding accumulated information from the beginning of another array till each index.

Suffix Array: an array holding accumulated information from the end of another array till each index.

You can accumulate any information like sum, multiplication, minimum, maximum, xor, ..etc.

Now let's see some types of the information you may need and how to construct prefix/suffix arrays and use them to get this information efficiently.

- **Prefix Sum:**

Given an array of integers ***A*** of length ***N***, you need to answer ***Q*** queries that ask for the sum of elements from index **0** to index ***X***.

Array ***A***:

0	1	2	3	4	5	6	7	8
1	5	4	-2	10	8	6	-1	2

Queries:

- Get the sum of elements in range [0, 5].
- Get the sum of elements in range [0, 2].
- Get the sum of elements in range [0, 8].

Solution:
First construct the prefix sum array ***Prefix_Sum***:

0	1	2	3	4	5	6	7	8
1	6	10	8	18	26	32	31	33

Now we can answer any prefix sum query in only O(1) Time Complexity:

- sum of elements in range [0, 5]: Prefix_Sum[5] = 26.
- sum of elements in range [0, 2]: Prefix_Sum[2] = 10.
- sum of elements in range [0, 8]: Prefix_Sum[8] = 33.

Code:

```
const int N = 1e6 + 1;
int A[N], prefix_sum[N];
int main() {
   int n, q;
   cin>> n;
   for (int i = 0; i < n; ++i) {
       cin>> A[i];
       prefix_sum[i] = A[i];
       if (i) prefix_sum[i] += prefix_sum[i - 1];
   }
   cin>> q;
   while (q--) {
       int x; cin>> x;
       cout<< prefix_sum[x] <<"\n";
   }
}
```

- **Suffix Sum:**

Given an array of integers ***A*** of length ***N***, you need to answer ***Q*** queries that ask for the sum of elements from index **X** till the end of the array.

Array ***A***:

0	1	2	3	4	5	6	7	8
1	5	4	-2	10	8	6	-1	2

Queries:

- Get the sum of elements in range [4, 8].
- Get the sum of elements in range [7, 8].
- Get the sum of elements in range [1, 8].

Solution:
First construct the suffix sum array ***Suffix_Sum***:

0	1	2	3	4	5	6	7	8
33	32	27	23	25	15	7	1	2

Now we can answer any suffix sum query in only O(1) Time Complexity:

- sum of elements in range [4, 8]: Suffix_Sum[4] = 25.
- sum of elements in range [7, 8]: Suffix_Sum[7] = 1.
- sum of elements in range [1, 8]: Suffix_Sum[1] = 32.

Code:

```
const int N = 1e6 + 1;
int A[N], suffix_sum[N];

int main() {
   int n, q;
   cin>> n;
   for (int i = 0; i < n; ++i) cin>> A[i];
   for (int i = n - 1; i >= 0; --i) {
       suffix_sum[i] = A[i];
       if (i + 1 < n) suffix_sum[i] += suffix_sum[i + 1];
   }
   cin>> q;
   while (q--) {
       int x; cin>> x;
       cout<< suffix_sum[x] <<"\n";
   }
}
```

- **Range Sum Queries:**

What if we need the sum of a range that is neither a prefix nor a suffix?

Let L and R be the starting and ending indices of the range. We know that sum[0, R] can be obtained as follows:

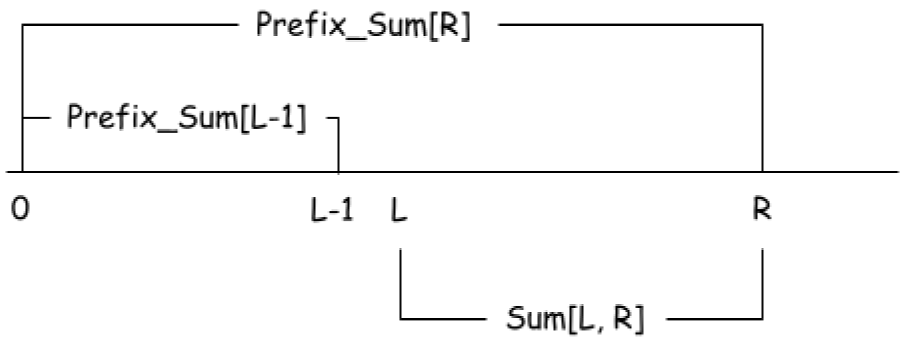
$$\sum_{i=0}^{R} A[i] = \sum_{i=0}^{L-1} A[i] + \sum_{i=L}^{R} A[i]$$

Thus, we can get sum[L, R] using this formula:

$$\sum_{i=L}^{R} A[i] = \sum_{i=0}^{R} A[i] - \sum_{i=0}^{L-1} A[i]$$

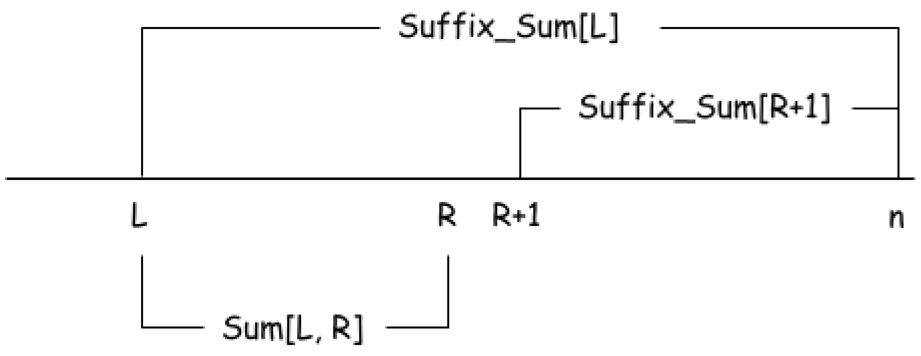
Since $\sum_{i=0}^{R} A[i]$ and $\sum_{i=0}^{L-1} A[i]$ both are prefix sums, then the prefix sum array can be used to obtain sum[L, R] as follows:

Sum[L, R] = Prefix_Sum[R] - Prefix_Sum[L - 1]



Similarly, the suffix sum array can be used to obtain sum[L, R] as follows:

Sum[L, R] = Suffix_Sum[L] - Suffix_Sum[R + 1]



These formulas are valid because summation is an invertible operation, an invertible operation is an operation that has a corresponding inverse operation that can undo its effect.

Queries:

- Get the sum of elements in range [4, 7].
- Get the sum of elements in range [1, 5].
- Get the sum of elements in range [0, 8].

Prefix_Sum:

0	1	2	3	4	5	6	7	8
1	6	10	8	18	26	32	31	33

Answer to queries:

- sum of elements in range [4, 7] = Prefix_Sum[7] - Prefix_Sum[3] = 31 - 8 = 23.
- sum of elements in range [1, 5] = Prefix_Sum[5] - Prefix_Sum[0] = 26 - 1 = 25.
- sum of elements in range [0, 8] = Prefix_Sum[8] = 33.

Code:

```
const int N = 1e6 + 1;
int A[N], prefix_sum[N];

int main() {
   int n, q;
   cin>> n;
   for (int i = 0; i < n; ++i) {
       cin>> A[i];
       prefix_sum[i] = A[i];
       if (i) prefix_sum[i] += prefix_sum[i - 1];
   }
   cin>> q;
   while (q--) {
       int l, r, sum;
       cin>> l >> r;
       sum = prefix_sum[r];
       if (l) sum -= prefix_sum[l - 1];
       cout<< sum <<"\n";
   }
}
```

- In exactly the same way we can construct multiplication, XOR (and any invertible operation) prefix and suffix arrays and use them to get the required information about any range we want.

- **Prefix Minimum:**

Given an array of integers ***A*** of length ***N***, you need to answer ***Q*** queries that ask for the minimum value among elements from index **0** to index ***X***.

Array ***A***:

0	1	2	3	4	5	6	7	8
1	5	4	-2	10	8	6	-5	2

Queries:
- Get the minimum value in range [0, 2].
- Get the minimum value in range [0, 7].
- Get the minimum value in range [0, 5].

Solution:
First construct the prefix minimum array ***Prefix_Min***:

0	1	2	3	4	5	6	7	8
1	1	1	-2	-2	-2	-2	-5	-5

Now we can answer any prefix min query in only O(1) Time Complexity:

- minimum value in range [0, 2]: Prefix_Min[2] = 1.
- minimum value in range [0, 7]: Prefix_Min[7] = -5.
- minimum value in range [0, 5]: Prefix_Min[5] = -2.

Code:

```
const int N = 1e6 + 1;
int A[N], prefix_min[N];


int main() {
   int n, q;
   cin>> n;
   for (int i = 0; i < n; ++i)
       cin>> A[i];
   prefix_min[0] = A[0];
   for (int i = 1; i < n; ++i)
       prefix_min[i] = min(prefix_min[i - 1], A[i]);
   cin>> q;
   while (q--) {
       int x; cin>> x;
       cout<< prefix_min[x] <<"\n";
   }
}
```

- Using the same concept, we can construct suffix minimum arrays and prefix/suffix maximum arrays.

- We can’t get the minimum/maximum value in a range that is neither a prefix or a suffix using prefix/suffix arrays because they are not invertible operations.

Problems:

- CSUMQ
- Kuriyama Mirai's Stones
- Program
- Lecture Sleep
- Ilya and Queries
- Book

3. Partial Sum

Given an array ***A*** of ***N*** integers, you need to apply ***Q*** queries in the following form:

- **L R VAL**: for each index ***i*** **(L <= i <= R)** update **A[i]** to **A[i] + VAL.**

Solution

Again, we can simulate this process and have O(Q * N) time complexity, which we need to optimize.

So what can we do?

1. Create a new array **D** of size **N.**
2. For each update query, do 2 steps:
    a. Update **D[L]** to **D[L] + VAL.**
    b. Update **D[R + 1]** to **D[R + 1] - VAL.**
3. In one go through array **D**, do a prefix sum. After applying this step, each index in **D** will be holding the final update value that needs to be applied to the original array **A**.
4. Add each value in **D** to its corresponding value in **A**.

Array ***A***:

0	1	2	3	4	5	6	7	8
2	6	-3	4	9	15	8	2	5

Queries:

- 0 4 3: add 3 to all values in range [0, 4].
- 4 8 -2: add -2 to all values in range [4, 8].
- 2 3 5: add 5 to all values in range [2, 3].

Construct array ***D***:

0	1	2	3	4	5	6	7	8	9
+3	0	+5	0	-7	-3	0	0	0	+2

Do prefix sum in array ***D***:

0	1	2	3	4	5	6	7	8	9
+3	+3	+8	+8	+1	-2	-2	-2	-2	0

These values are the final updates that you need to apply to array ***A***.

Add each value in ***D*** to its corresponding value in ***A:***

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

This way we applied all the update queries in **O(Q + N) time complexity**.

Code:

```
const int N = 1e6 + 1;
int A[N], D[N];

int main() {
   int n, q;
   cin >> n;
   for (int i = 0; i < n; ++i) cin >> A[i];
   cin >> q;
   while (q--) {
       int l, r, val;
       cin >> l >> r >> val;
       D[l] += val;
       D[r + 1] -= val;
   }
   for (int i = 0; i < n; ++i) {
       if (i) D[i] += D[i - 1];
       A[i] += D[i];
   }
   for (int i = 0; i < n; ++i)
       cout << A[i] << " ";
   cout << "\n";
   return 0;
}
```

- This solution is valid only when all range update queries are consecutive and you need to get the final array only after applying all the updates. So if the problem has a mix of range update queries with other types of queries asking for some information about the updated array elements, this would not help.

Problems:

- Karen and Coffee
- Greg and Array