بِسْمِ ٱللَّهِ ٱلرَّحْمَٰنِ ٱلرَّحِيمِ

In the name of Allah, Most Gracious, Most Merciful

CSE 4303
Data Structures

Topic: Sparse Table, Fenwick Tree

Asaduzzaman Herok
Lecturer | CSE | IUT
asaduzzaman34@iut-dhaka.edu

Problem Scenario

Think of a university of 10 thousand ex-students. Their results and other informations are sorted by their student IDs. Now it is very frequent that authority want to know from roll x to roll y what is the maximum CGPA.

What to do?

- Linear Search?
- Binary Search?
- Sorting the data based on CGPA?

Is there a way to get the result without changing the relative order of the data?

These types of query / questions are called ***Range Query***.

Example: Range Maximum Query, Range Minimum Query, Range Sum Query, Range GCD / LCM Query

Range Sum Problem

Given array of n elements $A = \{ a_1, a_2, \dots \dots \}$, in each query (l, r) it is asked to give the sum of elements from a_l to a_r. That is return $\{a_l + a_{l+1} + a_{l+2} + \dots \dots + a_{r-2} + a_{r-1} + a_r\}$

Range Minimum Problem

Given array of n elements $A = \{ a_1, a_2, \dots \dots \}$, in each query (l, r) it is asked to give the minimum element among a_l to a_r. That is return minimum $\{a_l, a_{l+1}, a_{l+2}, \dots \dots , a_{r-2}, a_{r-1}, a_r\}$

Generalised Way

Given array of n elements $A = \{ a_1, a_2, \dots \dots \}$, in each query (l, r) it is asked to give the $\boldsymbol{F}(\{a_l, a_{l+1}, a_{l+2}, \dots \dots , a_{r-2}, a_{r-1}, a_r\})$.

Naive Solutions

- Linear Search on every query starting from a_l to a_r.
  - Complexity on each $O(r-l+1)$ i.e $O(n)$
- Precalculation for once for every pair of (l,r).
  - Need $O(n^2)$ space and time.
  - But can answer query in constant time i.e $O(1)$

What to do then?

- Sparse Table
- Fenwick Tree
- Segment Tree
- Square Root Decomposition
- Square Root Tree
- Mo’s Algorithm

Our Discussion will be restricted upto Segment tree

Sparse Table

A sparse table is a data structure that can answer some range query problems, such as range minimum query problem, in `O(1)` time. Range sum in `O(log n)`

Pros:

- Building time complexity `O(n log n)`.
- Space complexity `O(n log n)`
- Supported query function:
  - Element in the array supports the associative property i.e x ***o*** $(y$ ***o*** $z) = (x$ ***o*** $y)$ ***o*** z. [Here ***o*** is a operator]
  - $F(a, b, c, d) = F(a,b)$ ***o*** $F(c,d)$, from the non-overlapping segments. Time complexity `O(log n)`
  - $F(a, b, c, d) = F(a,b,c)$ ***o*** $F(b,c,d)$, from the overlapping segments. Time complexity `O(1)` i.e constant time.

Cons:

- Can't work with update i.e data sequence must be immutable
- Update operation would cause `O(n log n) time complexity to rebuild.`

Sparse Table

Intuition:

- Any non-negative number can be uniquely represented as a sum of decreasing powers of two.
  - Example: $13 = (1101)_2 = 8 + 4 + 1$
  - Number of powers of two needed is $\log_2(n)$
- So a range of *m* elements can also be represented as union of continuous sub-segments of size decreasing powers of two
  - Example: `[2,14] = [2,9] U [10,13] U [14, 14]`
    `Size:     13       8        4          1`
  - If we already know these smaller ranges value then the large range can be computed.

Idea:

- Pre-compute all the answers of range with length equal to some powers of 2
- While querying break the query segments into some segments of powers of 2 and use their value to compute for the query segment.

Sparse Table Pre-Computation

- Declare a 2D array *sparse*`[n][k+1]` where `k = floor(`$\log_2$`(n)+1)`
- *sparse*`[i][j]` keeps the answer for segment / range starting at i and ending at $i + 2^j - 1$
- The recurrence can be written as:
  - *sparse*`[i][j]` = ***F***(*sparse*`[i][j-1]`, *sparse*`[`$i + 2^{j-1}$`][j-1]`)
  - `Example:` *sparse*`[4][3] = min(`*sparse*`[4][2],` *sparse*`[4+4][2])`

    `Range:    [4, 11]      = min([4, 7], [8, 11])`
- `Base case:`
  - *Sparse*`[i][0]` = ***F***(a_i)
  - `Example:` *sparse*`[4][0]` = min(a_4) = a_4

    `Range:    [4, 4]` = a_4

Sparse Table Pre-Computation

Algorithm 4: Sparse Table Construction for Range Minimum Query Problem

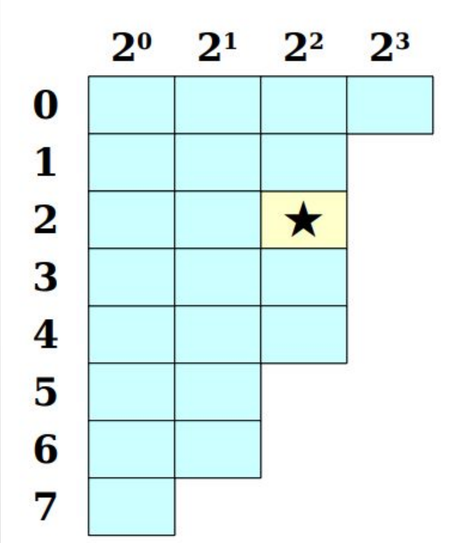
---

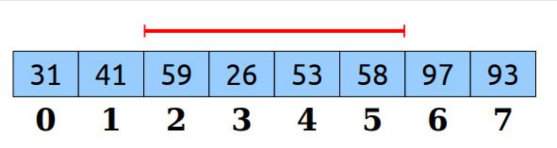
Data: A number array A with size n

Result: The minimum values of all possible power-of-two ranges of A

Function `preprocessSparse`(A, n)**:**

```
Initialize an n × (log₂ n + 1) 2D array sparse;
for i = 1 to n do
    sparse[i][0] = A[i];
end
for j = 1 to log₂ n do
    for i = 1 to n do
        sparse[i][j] = min(sparse[i][j − 1], sparse[i + 2^(j−1)][j − 1]);
    end
end
return sparse;
```





Sparse Table Query

Range Sum Query: (l, r)

Non-Overlapping Case

- Compute the power k = floor(log_2 (r-l+1))
  - Example: Range [4, 9], k = floor (log_2 (9-4+1 = 6)) = 2
- While (l<=r) do ans += *sparse*[l][k], l += 2^k, k = floor(log_2 (r-l+1))
  - Example: Range [4, 9]

    ans += *sparse*[4][2]   // [4, 7]

    l = 4 + 2^2 = 8

    k = log_2(9-8+1 = 2) = 1

    ans += *sparse*[8][1] // [8,9]

    l = 8 + 2^1 = 8

Range Min Query: (l, r)

Overlapping Case

- Compute the power k = floor(log_2 (r-l+1))
  - Example: Range [4, 9], k = floor (log_2 (9-4+1 = 6)) = 2
- ans = min(*sparse*[l][k], *sparse*[R-2^k+1][k]
  - Example: Range [4, 9]

    ans = min(*sparse*[4][2], *sparse*[9-2^2+1 = 6][2])

    [4, 7]   [6, 9]

Fenwick Tree

Fenwick tree is a data structure which:

- Calculates the value of function F in the given range $[l, r]$ (i.e. $F(\{a_l, a_{l+1}, \dots \dots, a_{r-1}, a_r\})$ in $O(\log_2 n)$ time;
- Updates the value of an element of A in $O(\log_2 n)$ time;
- Requires $O(n)$ memory, or in other words, exactly the same memory required for A
- Build time is also $O(n \log_2 n)$
- Also called Binary Indexed Tree, or just BIT abbreviated.

The most common application of Fenwick tree is calculating the sum of a range.

$$F(\{a_l, a_{l+1}, \dots \dots, a_{r-1}, a_r\}) = a_l + a_{l+1} + \dots \dots + a_{r-1} + a_r$$

F should support both $F(a, b, c, d) = F(a,b)\ o\ F(c,d)$ and $F(c,d) = F(a, b, c, d)\ o\ F(a, b)$,

o denotes some kind of operators.

Example: Sum(a, b, c, d)=Sum(a, b)+Sum(c, d) and Sum(c, d)=Sum(a, b, c, d)-Sum(a, b)

Xor(a, b, c, d)=Xor(a, b) *xor* Xor(c, d) and Xor(c, d)=Xor(a, b, c, d) *xor* Xor(a, b)

Fenwick Tree

Basic Idea:

- Each integer can be represented as a sum of powers of two.
  `13 = (1101)`$_2$`=2`^{3} `+ 2`^{2} `+ 2`^{0} `= 8+4+1`
- The answer of segment[1, 13] can be found by combining the range
  `[1,8] U [9,12] U [13,13]`
- That is the same way, a cumulative values can be represented as a sum of sets of sub segments value .
- A 1D array of size ***n*** is enough to store these sub-segment values.
  `BIT[1, 2, … , n ]`
- Let's ***i*** be an index of `BIT` and ***k*** be the position last non zero bit of the binary representation of ***i***.
  - Example: `i = 13 = (1101)`$_2$ `,` ***k*** `= 0`
    `i = 6 = (110)`$_2$ `,` ***k*** `= 1`
    `i = 8 = (1000)`$_2$ `,` ***k*** `= 3`
- `BIT[i]` will keep the segment value of segment `[i-2`^{k}`+1, i]`
  - Example: `BIT[13] = [13, 13]`
    `BIT[6] = [5, 6]`
    `BIT[8] = [1, 8]`
- So to get the segment value of `[1, 13]` we have to union
  `BIT[8] U BIT[10] U BIT[13]`
  `BIT[(1000)`$_2$`] U BIT[(1100)`$_2$`] U BIT[(1101)`$_2$`]`

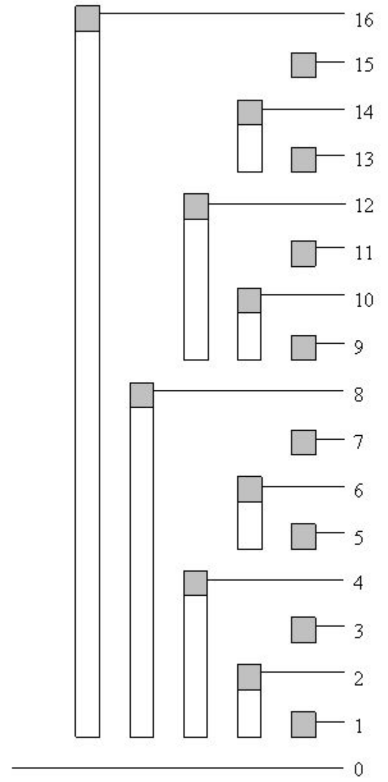


Image 1.3 – tree of responsibility for indices (bar shows range of frequencies accumulated in top element)

Asaduzzaman Herok, Lecturer
Department of Computer Science and Engineering, IUT

Fenwick Tree

Basic Idea:

- We need to find a way to jump from these segments to segments. Like from 13 to 12 to 8 by setting the least significant bit to 0.

Isolating the last BIT:

- Let ***i*** be ***a1b*** as a representation of ***i*** in binary where ***a*** is the prefix from the last significant bit and ***b*** is all zeros.
  - Example: $20 = \underbrace{(10100)_2}_{a1b}$, a = 10, b = 00
- $(a1b)^-$ is the inversion of bits.
  - $(20)^- = (10100)_2^- = (01011)_2$
- $-num = (a1b)^- + 1 = a^-0b^- + 1 = a^-0(0\ldots0)^- + 1$
  $= a^-0(1\ldots1) + 1 = a^-1(0\ldots0) = a^-1b.$
- If we do bitwise ***and*** of ***a1b*** with ***a¯1b*** we get $(0..010..0)_2$

  Example: 20 & -20 = 4

  $(10100)_2$ & $(01100)_2 = (100)_2$

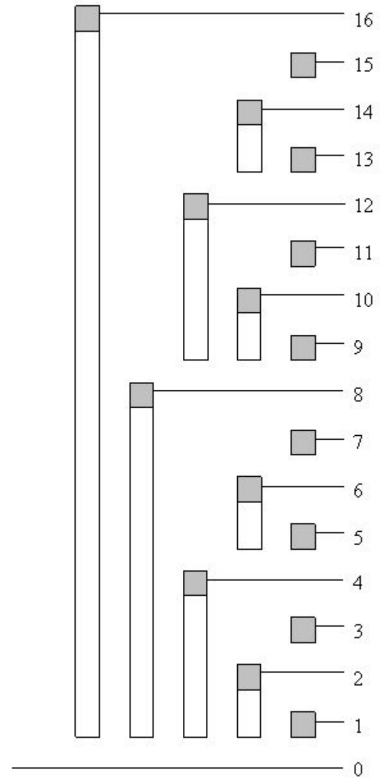
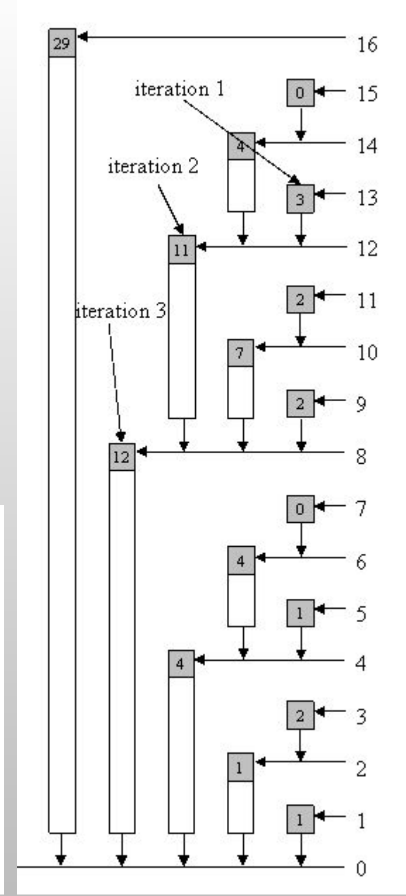


Image 1.3 – tree of responsibility for indices (bar shows range of frequencies accumulated in top element)

Fenwick Tree Read Operation on Segment Sum

```
int read(int idx) {
  int sum = 0;
  while (idx > 0) {
    sum += tree[idx];
    idx -= (idx & -idx);
  }
  return sum;
}
```

ITERATION	IDX	POSITION OF THE LAST BIT	IDX & -IDX	SUM
1	13 = 1101	0	1 (2 ^0)	3
2	12 = 1100	2	4 (2 ^2)	14
3	8 = 1000	3	8 (2 ^3)	26
4	0 = 0	—	—	—

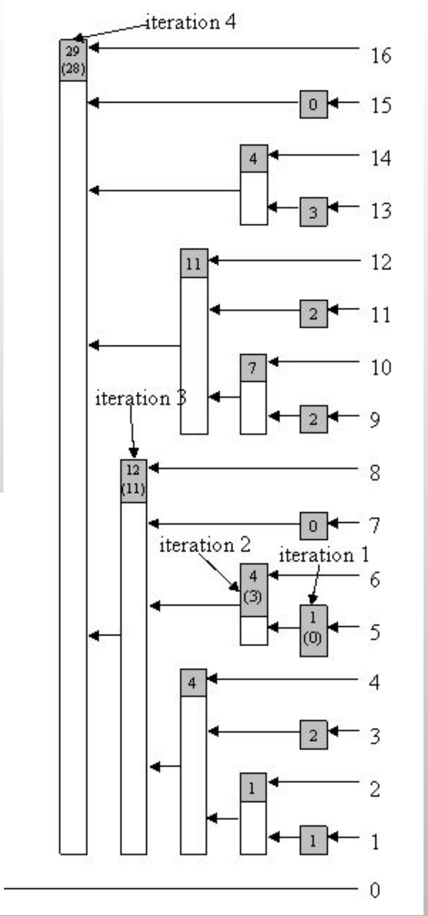


Fenwick Tree Update Operation on Segment Sum

- The update operation is to increase or decrease the certain value at index i of the given array A.
- Increase or decrease should be reflected in the affected range sums in the BIT.
- To find the affected segments we have to add the last bit of i to itself; and, repeat while i is less than or equal to n

```
void update(int idx, int val)
{
  while (idx <= MaxIdx) {
    tree[idx] += val;
    idx += (idx & -idx);
  }
}
```

ITERATION	IDX	POSITION OF THE LAST BIT	IDX & -IDX
1	5 = 101	0	1 (2 ^0)
2	6 = 110	1	2 (2 ^1)
3	8 = 1000	3	8 (2 ^3)
4	16 = 10000	4	16 (2 ^4)
5	32 = 100000	—	—



Fenwick Tree Range Query (l,r)

Recall the property:

$F(c,d) = F(a, b, c, d) \circ F(a, b)$

- $Read(r)$ gives the segment value of $[1, r]$
  $Read(l-1)$ gives the segment value of $[1, l-1]$
- $Query(l,r) = Read(r) \circ Read(l-1)$
  $[1, r] \circ [1, l-1]$

Range / Segment Sum $Query(l,r)$

- $Read(r)$ gives the segment value of $[1, r]$
  $Read(l-1)$ gives the segment value of $[1, l-1]$
- $Query(l,r) = Read(r) - Read(l-1)$
  $[1, r] - [1, l-1]$

Acknowledgements

[Sparse Table - Algorithms for Competitive Programming (cp-algorithms.com)]

[Fenwick Tree - Algorithms for Competitive Programming (cp-algorithms.com)]

[Binary Indexed Trees (topcoder.com)]