Advanced Computer Contest Preparation

Lecture 24

BINARY INDEXED TREES

Problem

- We have an array of N integers
- We will do Q operations, each are either query or update
- Update:
  - Add a constant to one element in the array
- Query:
  - Get the sum of the elements in the range $[l,r]$

Solution

- We can use a segment tree, but the code can be long, and the constant factor is not very good
  - Would still pass the problem though
- Do not use PSA or DA since worst case is $O(NQ)$
  - For a PSA, every operation is update
  - For a DA, every operation is query

Binary Indexed Trees

- Also called Fenwick Trees
  - Named after Peter Fenwick
- "Binary Indexed" → binary representation of indexes is significant
- Like a segment tree, both update and query run in $O(\log N)$ time

Binary Indexed Tree

- Pros:
  - Quick, short code
  - Good constant factor
  - Low memory usage
- Cons:
  - Not as versatile
    - Can only query sum
    - Range update with range query is complex
  - Harder to understand
    - Requires knowledge of binary representation

Binary Indexed Tree

- A BIT is normally implemented using a 1-indexed array
  - We will call this array `BIT[]` and the original array `ar[]`
- BITs are based upon the concept of prefix sum arrays
- However, while `PSA[idx]` represents the sum of all elements in $[1,idx]$, BITs store the sum of only some elements in $[1,idx]$
- Tradeoff: slower query than PSA, faster update

Binary Indexed Tree

- Fenwick's idea:
  - For each idx, change it into its binary representation
  - Let n be the number of places the right-most '1' is located from the right
    - Example: `10010110100`, the right-most '1' is located at position 3
  - **`BIT[idx]`** contains the sum of the elements in $[idx - 2^{n-1} + 1, idx]$

Binary Indexed Tree

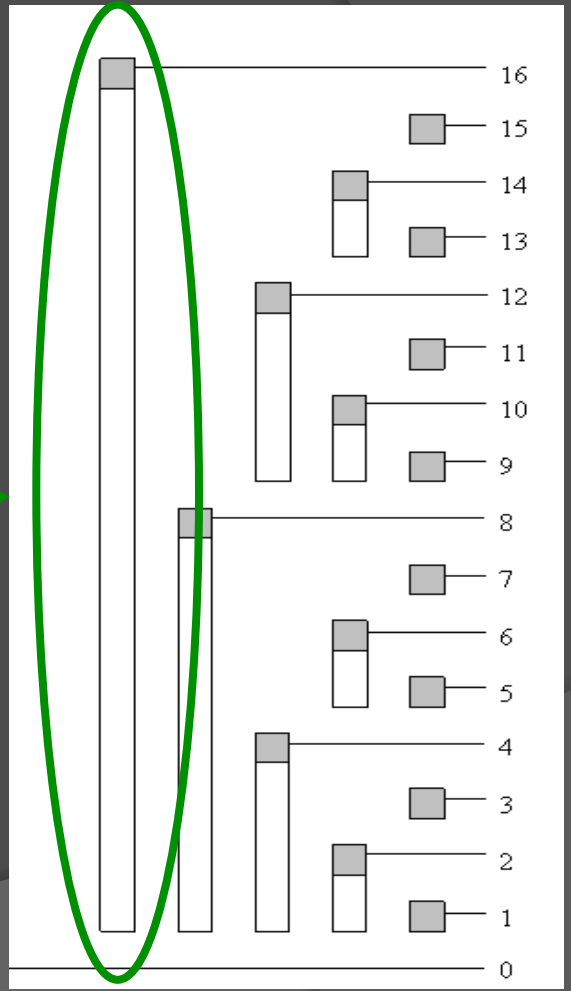
- Alternate explanation
  - Let p be the largest power of 2 that divides idx
  - **`BIT[idx]`** is the sum of the last p elements, including idx
  - Sum from $[idx-p+1, idx]$

Binary Indexed Tree

- Examples:
  - Which elements does **`BIT[12]`** hold?
    - Binary representation is **`1100`**
    - Range is from $[12-2^2+1,12]$, which is $[9,12]$
  - Which elements does **`BIT[16]`** hold?
    - Binary representation is **`10000`**
    - Range is from $[16-2^4+1,16]$, which is $[1,16]$
  - Which elements does **`BIT[5]`** hold?
    - Binary representation is **`101`**
    - Range is from $[5-2^0+1,5]$, which is $[5,5]$

Table of Responsibility

White bar & grey area represent the range of each element in the Binary Indexed Tree

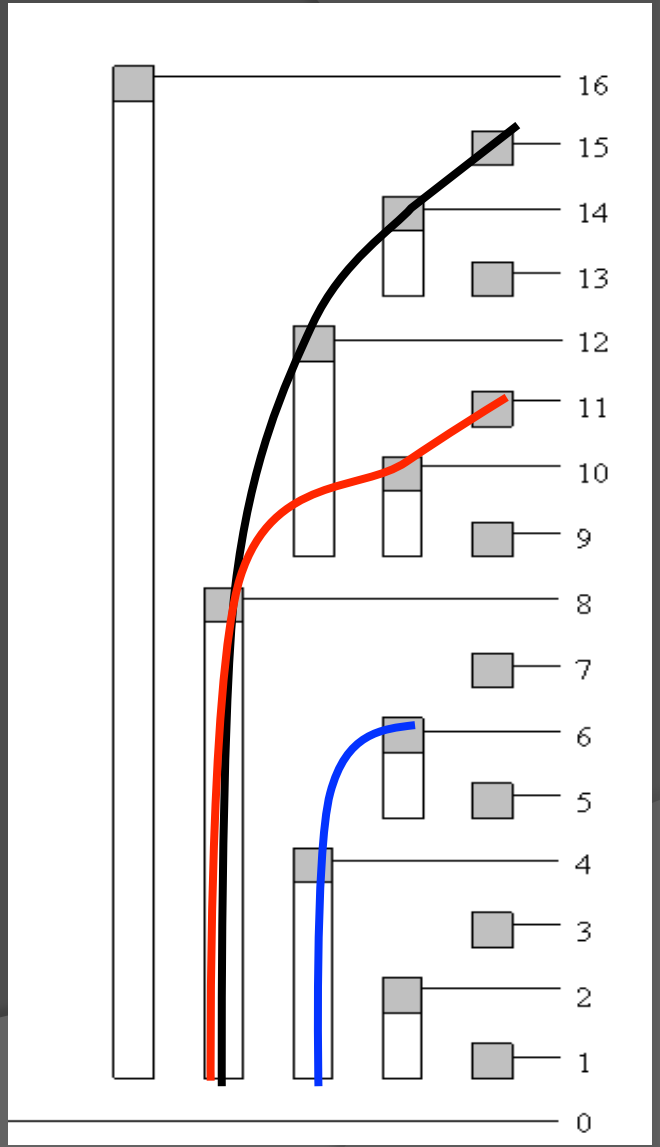


Query

- BITs, like a PSA, should return the sum of the elements in **`ar[]`** in the range $[1, idx]$
- Let this function be called **`sum(idx)`**
- **`sum(idx) = ar[1] + ar[2] + ... + ar[idx]`**
- However, **`BIT[idx]`** is not guaranteed to be equal to **`sum(idx)`**
- We must sum certain **`BIT[i]`** such that we get **`ar[1] + ar[2] + ... + ar[idx]`**

Query – Illustration

- Every index from *1* to *idx* must be covered
- From our starting index, we need to find a path to the bottom
- Note that by doing this, we will cover all indexes from *1* to *idx*
- Example of paths to bottom →
- How do we determine which indexes to add?



Query

- Recall
  - For an idx, let n be the number of places the right-most '1' is located from the right
  - **`BIT[idx]`** contains the sum of the elements in $[idx - 2^{n-1} + 1, idx]$
- Therefore, the next **`BIT[]`** we use must have a right end interval of $idx - 2^{n-1}$
- **`sum(idx) = sum(idx-2`$^{n-1}$`) + BIT[idx]`**

Query

- To get the next index we need to query, we need to subtract the largest power of 2 that divides the current index
- Equivalent to changing the right-most '1' in the index's binary representation to a '0'
- Simple method: check every bit from 0 to 30 (if we are using an `int`)
  - `if (idx & (1 << pos))`
  - Not as efficient, query runtime will be $O(\log^2 N)$
- Is there a better method?

Two's Complement

- A way of representing negative numbers in binary; all computers use this method
- Negative numbers are represented by flipping the bits of the absolute value and adding 1
  - Get the complement (~, bitwise NOT) of the absolute value and add 1
- For example: represent -6 in binary
  - 6 → `0...0110`
  - -6 → ~6 + 1 → `1...1001` + 1 → `1...1010`

Finding the Last '1'

- Let N be a positive integer
- Let a be the part of the binary representation before the last/right-most '1'
- Therefore, each number is in the form **a10...0**
- Example: $N = 12$ → **000...01100**
  - a = **000...01**
  - N = **a100**

Finding the Last '1'

- Find the value of $-N$ in terms of a

```
-N
= ~N + 1
= ~(a10...0) + 1
= (~a)01...1 + 1
= (~a)10...0
```

Finding the Last '1'

- N = **a10...0**, $-N$ = **(~a)10...0**
- Use the bitwise AND operation (&) on N and $-N$

N & -N

= a10...0 & (~a)10...0

= 0...010...0

- Therefore, $N \,\&\, -N$ will give a number whose only bit set is the last '1' of N

C++ Code – Query

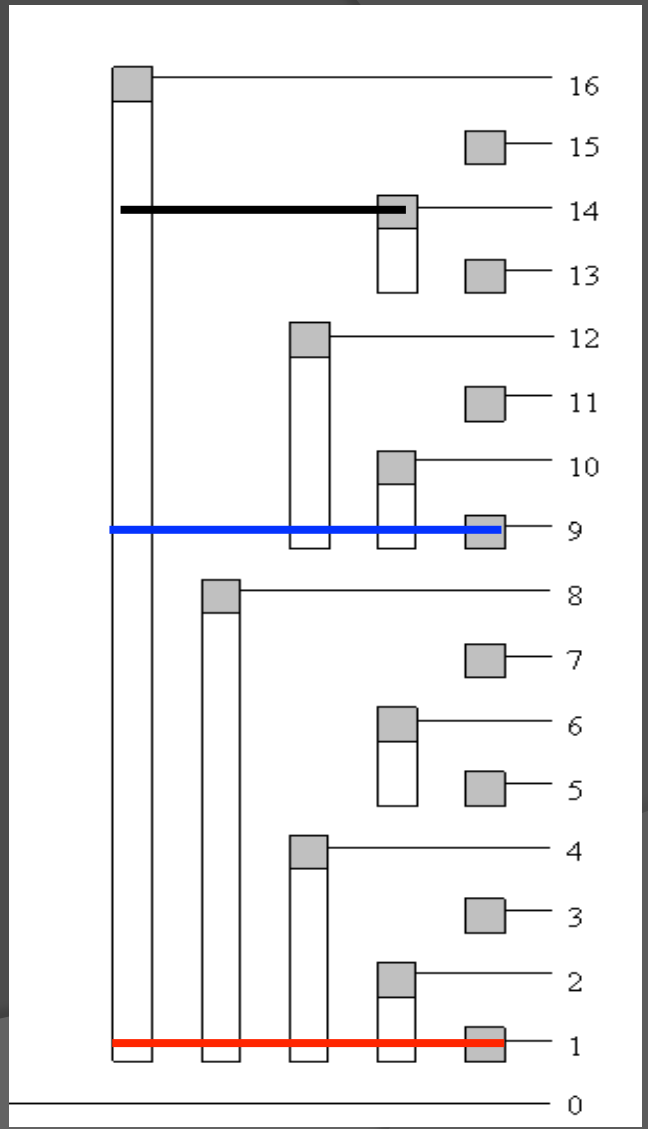
```
int BIT[];

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

Update

- We want to change the value of `ar[idx]`
- However, several `BIT[i]` might include `ar[idx]` as part of its sum
- All such `BIT[i]` must be updated as well

Update

- Suppose the largest index is 16
- Draw a line from the desired index, heading leftward
- Any `BIT[]` that this line passes must be updated



Update

- Recall
  - For an idx, let n be the number of places the right-most '1' is located from the right
  - **`BIT[idx]`** contains the sum of the elements in $[idx - 2^{n-1} + 1, idx]$
- We need to update all such **`BIT[i]`** such that idx is in **`BIT[i]`**'s range

Update

- Let x be the smallest next index to update, m be the largest integer such that 2^m divides x
- The binary representation of idx can be expressed as **a0b**, where $b = 1...10...0$
- The binary representation of x can be expressed as **c10...0**
- $x > idx \geq x - 2^m$
- **c10...0 > a0b ≥ c0...0**
- For x to be as small as possible
  - $c = a$
  - **10...0** (from **c10...0**) and **0b** (from **a0b**) have the same number of digits
- **a10...0 > a0b ≥ a0...0**
- Therefore, our next index is **a10...0**, and to get this, we add the last 1 to idx

C++ Code – Update

```
int BIT[];

void update(int num, int idx){
      while(idx <= max){
            BIT[idx] += num;
            idx += (idx & -idx);
      }
}
```

Update

- **WARNING: Never replace elements in `BIT[]`, only add or subtract**
  - `BIT[i]` can represent the sum of more than 1 element in `ar[]`
  - If we replace `BIT[i]` with a single value in `ar[]`, we lose the sum of the other elements

Range Update, Point Query

- The BIT functions like a PSA, but it can query and update in $O(\log N)$ time
- Remember that the regular array is the PSA of a difference array
- To add C to $[l,r]$, use `update(C,l)` and `update(-C,r+1)`
- To get the value at idx, use `sum(idx)`

Range Update, Range Query

- Range updates and queries are possible using BITs
- More difficult to implement; requires 2 BITs and multiplication
- Can use a segment tree, since advantages of using 2 BITs is less prominent than using a segment tree

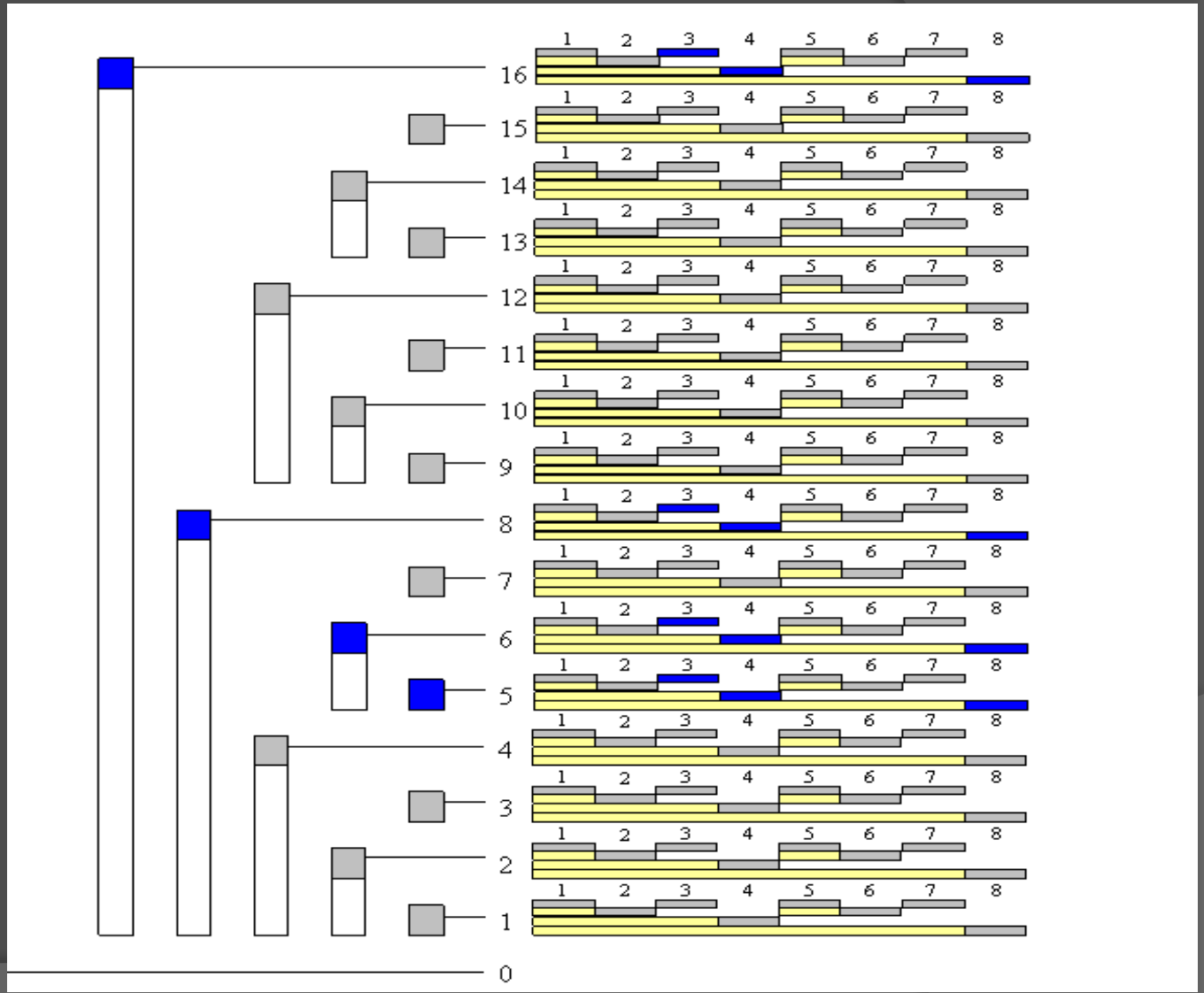
2D Binary Indexed Tree

- Each BIT in the first dimension (row) contains another BIT(column)
- Queries/updates are now in the form `(row,col)`
- When you update ANY row, you MUST update its column as well
- When you query ANY row, you MUST query its column as well

2D BIT

Update (5,3)

Blue indicates what needs to be updated



2D BIT - Update

```
int bit[][];

void update(int row, int col, int val){
	while(row <= maxR){
		int col1 = col; //we need to preserve col
		while(col1 <= maxC){
			bit[row][col1] += val;
			col1 += (col1 & -col1);
		}
		row += (row & -row);
	}
}
```

2D BIT - Query

```
int bit[][];
int sum(int row, int col){
        int val = 0;
        while(row > 0){
                int col1 = col;
                while(col1 > 0){
                        val += bit[row][col1];
                        col1 -= (col1 & -col1);
                }
                row -= (row & -row);
        }
        return val;
}
```

2D Binary Indexed Trees

- Usage is same as 2D Prefix Sum array
- Query/update is $O(\log^2 N)$ each
- Query will return sum of rectangle with corners $(1,1)$ and (R,C)
- Update can update only a single point

Higher Dimension BIT

- Higher dimension BITs work similarly
- Every dimension except the last has a BIT
- When you update/query a BIT, you must update the BIT it contains (unless it is the last dimension)
- Runtime of query/update is $O(\log^k N)$, where k is the number of dimensions

THANK YOU!