Range queries

การสอบถามแบบช่วง

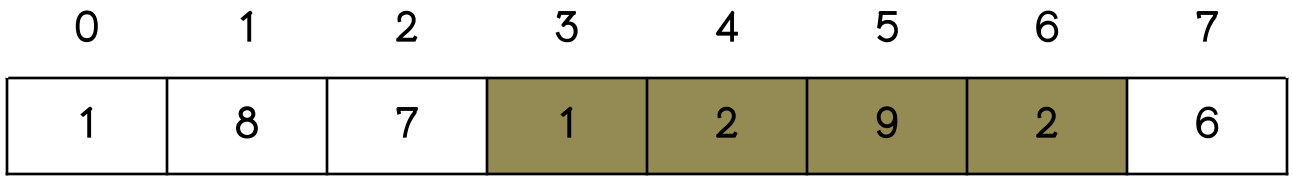
- ในบทนี้จะกล่าวถึงโครงสร้างข้อมูลที่ช่วยให้เราสอบถามคำถามแบบช่วงได้อย่างมีประสิทธิภาพ

- ในการสอบถามแบบช่วง (range query) นั้น งานของเราคือคำนวณค่าซึ่งได้มาจาก subarray ของ array

- โดยทั่วไปแล้วมีการสอบถามดังนี้
  - $\text{sum}_q(a,b)$: เป็นการคำนวณผลรวมในช่วง $[a, b]$
  - $\min_q(a,b)$: เป็นการหาค่าน้อยสุดในช่วง $[a, b]$
  - $\max_q(a,b)$: เป็นการหาค่ามากสุดในช่วง $[a, b]$

ตัวอย่างการสอบถามแบบช่วง

- ตัวอย่างเช่น
  - หากเราสอบถามในช่วง **[3,6]** ของ array ด้านล่างนี้



  - ในกรณีนี้เราจะพบว่า
    - $sum_q(3,6) = 14$
    - $min_q(3,6) = 1$
    - $max_q(3,6) = 9$

แก้ปัญหานี้แบบง่าย(ถึก)

- วิธีการที่ง่ายในการจัดการปัญหานี้คือการใช้ Loop ตรวจสอบทุก ๆ ค่าของ array ในช่วง
- ตัวอย่างต่อไปเป็น function สำหรับสอบถาม $\mathbf{sum}_q(a,b)$

```
int sum(int a, int b) {
  int s = 0;
  for (int i = a; i <= b; i++) {
    s = s + array[i];
  }
  return s;
}
```

เวลาการทำงานเป็นเท่าไร?

วิเคราะห์การทำงาน

```
int sum(int a, int b) {
  int s = 0;
  for (int i = a; i <= b; i++) {
    s = s + array[i];
  }
  return s;
}
```

- เราพบว่า ฟังก์ชันนี้ทำงานในเวลา **O(n)** เมื่อ n เป็นขนาดของ array
- ดังนั้นหากเราสอบถาม **q** ครั้งจะใช้เวลา **O(qn)** โดยใช้ฟังก์ชันนี้
- อย่างไรก็ตามเราพบว่า ถ้า **n** และ **q** มีขนาดใหญ่ ฟังก์ชันนี้จะทำงานช้า ในหัวข้อต่อ ๆ ไปเราจิจารณาวิธีที่มีประสิทธิภาพกัน

Today topics

- Static array queries
- Binary indexed tree
- Segment Tree
- Additional techniques

Static array queries

- เราจะเริ่มพิจารณากรณี Static array หรือ ค่าของ array **ไม่**เปลี่ยนแปลงระหว่างการสอบถามก่อน

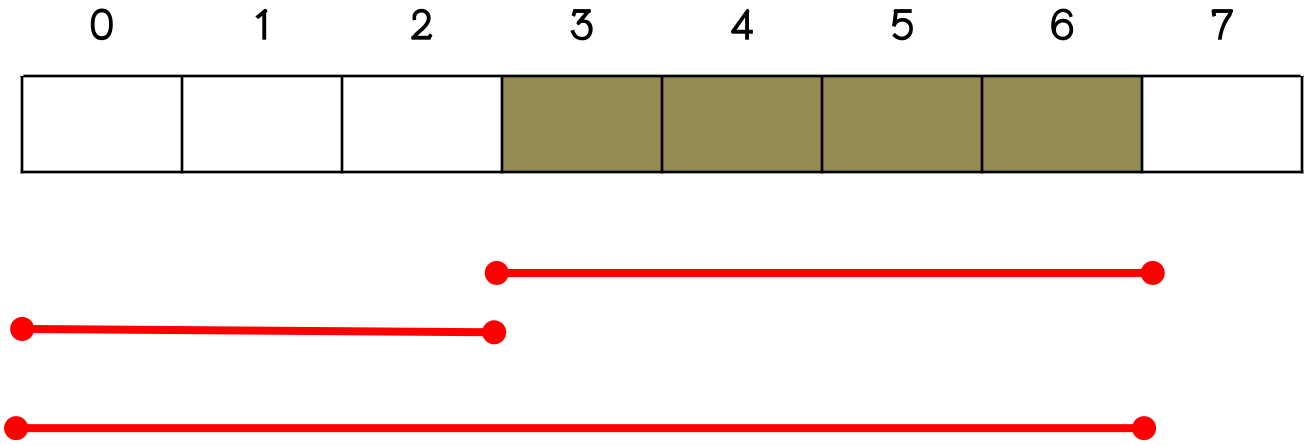
- ในกรณีนี้ เราจะสร้างโครงสร้างข้อมูลที่ไม่เปลี่ยนแปลงเพิ่ม (คือสร้างแล้วไม่ update ทีหลัง) เพื่อมาช่วยตอบคำถามของเรา

- เราจะพิจารณา sum queries จากนั้นจะพิจารณา minimum queries ทั้งนี้ maximum queries แก้ได้ในลักษณะเดียวกับ minimum queries

Sum queries

- สมมติเรามี array a ของเลขจำนวนเต็ม หากเราต้องการหาผลรวมของเลขตำแหน่งที่ 3 ถึงตำแหน่งที่ 6 หาได้อย่างไร

- sum(3,6) = a[3] + a[4] + a[5] + a[6]



- หากเรามีผลรวมของเลขช่วง [0,6] และเรามีผลรวมของเลขช่วง [0,2] เราจะหาผลรวมของเลขช่วง [3,6] ได้อย่างไร

Sum queries

- เราสามารถประมวลผล sum queries บน static array โดยการสร้าง **prefix sum array** (หรืออาจจะเคยได้ยิน **quick sum**)

- แต่ละค่าใน **prefix sum array** จะเท่ากับผลรวมของค่าใน array ต้นฉบับตั้งแต่ตำแหน่งแรกจนถึงตำแหน่งที่พิจารณา นั่นคือค่าในตำแหน่งที่ k คือ $\mathbf{sum}_q(0, k)$

- **prefix sum array** สามารถสร้างได้ในเวลา $O(n)$

- ตัวอย่างเช่นพิจารณา array ต่อไปนี้

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

- **prefix sum array** ของ array ด้านบนคือ

0	1	2	3	4	5	6	7
1	9	16	17	19	28	30	36

- เนื่องจาก prefix sum array เก็บทุกค่าของ $\mathbf{sum}_q(0, k)$ เราสามารถคำนวณผลรวมของช่วงใดๆ ได้ $\mathbf{sum}_q(a, b)$ ภายในเวลา $O(1)$ ดังนี้:

$$\mathbf{sum}_q(a, b) = \mathbf{sum}_q(0, b) - \mathbf{sum}_q(0, a-1)$$

- โดยกำหนดให้ $\mathbf{sum}_q(0,-1) = 0$

- ในกรณีนี้หากต้องการหาค่า $\mathbf{sum_q(3,6)}$ = 1+2+9+2 =14

- array

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

- prefix sum array

0	1	2	3	4	5	6	7
1	9	16	17	19	28	30	36

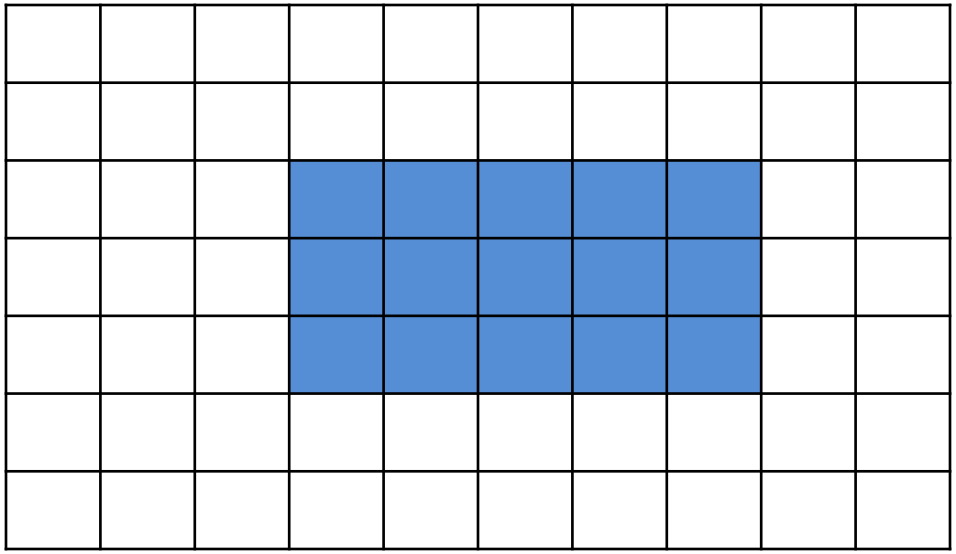
- $\mathbf{sum_q(3,6)}$ คำนวณได้จาก $\mathbf{sum_q(0,6)}$ – $\mathbf{sum_q(0,2)}$ = 30–16 = 14

- ทั้งนี้สามารถนำเอาหลักการนี้ไปใช้ในมิติที่มากกว่านี้ได้ด้วย

- ตัวอย่างเช่น เราสามารถสร้าง prefix sum array 2 มิติ สามารถใช้ในการหาผลรวมของ subarray ที่เป็นสี่เหลี่ยมใน 2 มิติได้ในเวลา $O(1)$

- โดยผลรวมแต่ละค่าใน array แทนสี่เหลี่ยมของ subarray ที่เริ่มต้นด้วยมุมซ้ายบนของ array จนถึงจุดที่เราพิจารณา

- หากต้องการหาพื้นที่สีฟ้า สามารถคำนวณได้อย่างไร



Minimum queries

- Minimum queries ยากขึ้นมาอีกหน่อย แต่สามารถจัดการได้ด้วยวิธีการ preprocess ก่อน หลังจากสามารถสอบถาม Minimum queries ได้ในเวลา *O(1)*

- ทั้งนี้ในการจัดการ Maximum queries สามารถจัดการได้คล้ายกับ Minimum queries ดังนั้นในส่วนนี้จะกล่าวถึงเพียง Minimum queries

วิธีถึก เก็บตรงๆ

- เรามาดูวิธีเก็บแบบง่ายตรงไปตรงมาก่อน
- วิธีการคือ เราสร้างตารางขนาด nxn ที่แต่ละช่องเก็บ **index** ของค่าต่ำสุดในช่วง i ถึง j

	0	1	2	3	4	5	6	7
arr	1	8	7	1	2	9	2	6
lookup	0	0	0	0	0	0	0	0
		1	2	3	3	3	3	3
			2	3	3	3	3	3
				3	3	3	3	3
					4	4	4	4
						5	6	6
							6	6
								7

เราก็เติมตารางทุกช่อง

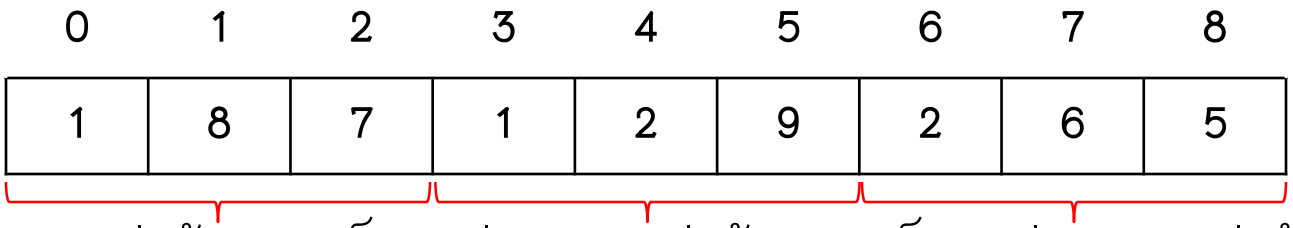
```
for (int i = 0; i < n; i++)
        lookup[i][i] = i;

for (int i=0; i<n; i++){
      for (int j = i+1; j<n; j++){
          if (arr[lookup[i][j - 1]] < arr[j])
              lookup[i][j] = lookup[i][j - 1];
          else
              lookup[i][j] = j;
      }
}
```

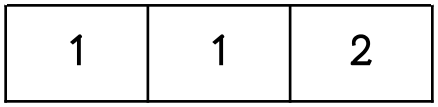
- สังเกตว่า การเติมตารางเช่นนี้ preprocess ใช้เวลา $O(n^2)$
- แต่เวลาเรียกถามเช่น ต้องการถามว่าค่าต่ำสุดในช่วง [L,R] เราก็สอบถามได้ใน $O(1)$ ด้วยคำสั่ง arr[lookup[L][R]]
- ซึ่งเวลาในการ preprocess ถือว่าสูงมากอยู่
- เราจะมาปรับปรุงเพื่อให้ preprocess ใช้เวลาน้อยลง

Square root decomposition

- เราจะใช้ Square root decomposition ในการลดเวลาจากวิธีเติมตาราง
- วิธีการคือ
  - เราจะแบ่งช่วง [0,n–1] ออกเป็นก้อน ก้อนละ $\sqrt{n}$ ตัว
  - คำนวณค่าต่ำสุดของทุกก้อน (นั่นคือมี $\sqrt{n}$ ก้อน) แล้วเก็บคำตอบ
  - ดังนั้นเวลาในการคำนวณเป็น $O(\sqrt{n} * \sqrt{n}) = O(n)$ และใช้เนื้อที่ $O(\sqrt{n})$



- ช่วง[0,2] ค่าน้อยสุดเป็น 1 ช่วง[3,5] ค่าน้อยสุดเป็น 1 ช่วง[6,8] ค่าน้อยสุดเป็น 2



- ทีนี้ในการ query ช่วง [L,R] เราก็นำเอาค่าที่น้อยสุดของทุกก้อนที่ทับช่วงที่สอบถาม

- สำหรับก้อนทางซ้ายและขวานั้น อาจจะทับแค่บางส่วน เราก็ Linear search เพื่อหาค่าน้อยสุดซึ่งมีจำนวนตัวไม่เกิน $\sqrt{n}$

- ดังนั้นเวลาในการสอบถามคือ $O(\sqrt{n})$ สังเกตว่าเรามีค่าน้อยสุดของก้อนตรงกลางที่เราสอบถามได้เลย และมีไม่เกิน $O(\sqrt{n})$ ก้อน และสองก้อนหัวท้ายที่เราอาจจะต้อง scan ไม่เกิน $2*O(\sqrt{n})$ ดังนั้นใช้เวลา $O(\sqrt{n})$

ในการ preprocess

```
void preprocess(int input[], int n)
{
    // initiating block pointer
    int blk_idx = -1;
    int minimum_block;
     // calculating size of block
    blk_sz = sqrt(n);
     // building the decomposed array
    for (int i=0; i<n; i++)
    {
        arr[i] = input[i];
        if (i%blk_sz == 0)
        {
            // entering next block incementing block pointer
            minimum_block = input[i];
            blk_idx++;
            block[blk_idx] = input[i];
        }
        if(minimum_block>input[i]){
            block[blk_idx] = input[i];
        }
    }
}
```

ในการสอบถาม

```
int query(int l, int r)
{   int m = arr[l];
    while (l<r and l%blk_sz!=0 and l!=0)
    {   // traversing first block in range
        if(m>arr[l])
            m=arr[l];
        l++;
    }
    while (l+blk_sz <= r)
    {   // traversing completely overlapped blocks in range
        if(m>block[l/blk_sz])
            m=block[l/blk_sz];
        l += blk_sz;
    }
    while (l<=r)
    {   // traversing last block in range
        if(m>arr[l])
            m=arr[l];
        l++;
    }
    return m;
}
```

- หากต้องการ update บางช่อง ต้องทำอย่างไร
- หากข้อมูลเป็น 1, 5, 2, 4, 6, 1, 3, 5, 7, 10

```
cout << "query(1,4) : " << query(1, 4) << endl;
cout << "query(1,6) : " << query(1, 6) << endl;
cout << "query(8,8) : " << query(8, 8) << endl;
update(8,1);
cout << "query(6,9) : " << query(6, 9) << endl;
```

ควรตอบอะไรบ้าง

- นอกจากนี้เรายังต้องการให้เร็วขึ้นอีก

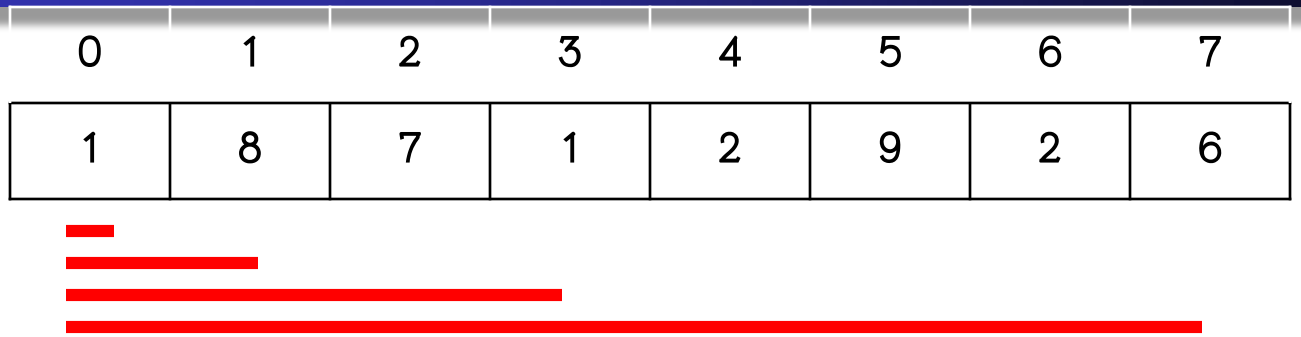
Idea

- แนวคิดเบื้องต้นคือ เมื่อพิจารณาตำแหน่ง a เราจะคำนวณค่าของทุกช่วงที่ห่างจาก a เป็นระยะสองยกกำลังต่าง ๆ (2^0, 2^1, 2^2, 2^3, 2^4, ..., $2^{\log n}$) เก็บไว้(นับ a ด้วย)

- ลองคิดดูว่า จะทำอย่างไรต่อ

- เราจะคำนวณทุกค่าของ $\min_q(a,b)$ ไว้ก่อน เมื่อ **b−a+1** (ซึ่งเป็นความยาวของช่วง) มีค่าเป็นยกกำลังของสอง

- วิธีการนี้เรียกว่า **Sparse table**



2^0

a	b	$min_q(a,b)$
0	0	1
1	1	8
2	2	7
3	3	1
4	4	2
5	5	9
6	6	2
7	7	6

2^1

a	b	$min_q(a,b)$
0	1	1
1	2	7
2	3	1
3	4	1
4	5	2
5	6	2
6	7	2

2^2

a	b	$min_q(a,b)$
0	3	1
1	4	1
2	5	1
3	6	1
4	7	2

2^3

a	b	minq(a,b)
0	7	1

- สมมติว่าข้อมูลเป็น arr[] = {7,2,3,0,5,10,3,12,18}
- สร้างตาราง lookup โดยที่ lookup[i][j] จะเก็บค่า index ของตัวน้อยสุดในช่วง arr[i] ถึง arr[i+2^k–1]

7	**2**	**3**	**0**	**5**	**10**	**3**	**12**	**18**

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

การ preprocess

```
void preprocess(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        lookup[i][0] = i;

    for (int j=1; (1<<j)<=n; j++)
    {
        // Compute minimum value for all intervals with size 2^j
        for (int i=0; (i+(1<<j)-1) < n; i++)
        {
            if (arr[lookup[i][j-1]] < arr[lookup[i + (1<<(j-1))][j-1]])
                lookup[i][j] = lookup[i][j-1];
            else
                lookup[i][j] = lookup[i + (1 << (j-1))][j-1];
        }
    }
}
```

- จำนวนของค่าที่ต้องคำนวณไว้คือ O(*nlogn*) เนื่องจากว่าช่วงของเรามีความยาวเป็นยกกำลังของสอง ดังนั้นมีไม่เกิน O(*logn*) ช่วง

- อีกทั้งค่าต่าง ๆ ที่จริงแล้วสามารถถูกคำนวณได้อย่างมีประสิทธิภาพโดยใช้ recursive formula

$$\min_q(a,b) = \min(\min_q(a, a+w-1), \min_q(a+w, b))$$

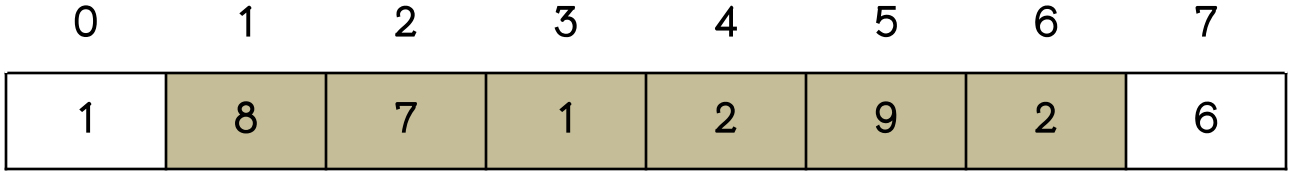
เมื่อ b−a+1 มีค่าเป็นยกกำลังของสองและ w = (b−a+1)/2

การคำนวณทุกค่าจึงใช้เวลาทั้งสิ้น O(nlogn)

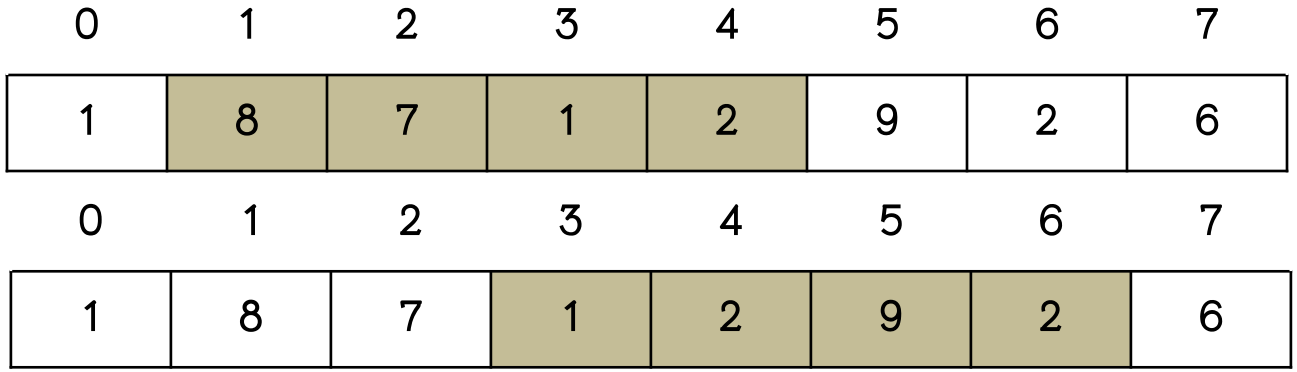
การสืบค้น

- หลังจาก preprocess ค่าของ $\min_q(a,b)$ สามารถคำนวณได้ใน O(1)
- วิธีการ เนื่องจากแต่ละตัวจะมีการคำนวณช่วงยกกำลังของสองไว้ ดังนั้นเราจะเลือกค่ายกกำลังของสองที่มากที่สุดที่ไม่เกินช่วงมาใช้ สมมติให้ k เป็นยกกำลังของสองที่มากที่สุดที่ไม่เกิน b–a+1
- ด้านหน้าจะเริ่มจาก a ไป k ช่อง ยังเหลือส่วนด้านหลัง คำถามถ้าช่วงซ้อนทับกันยังได้ค่าน้อยสุดอยู่ไหม
- นั่นคือ $\min_q(a,b) = \min(\min_q(a, a+k-1), \min_q(b-k+1, b))$

- ตัวอย่างพิจารณาช่วง [1,6]



- ความยาวของช่วงเป็น 6 และค่าสองยกกำลังที่มากที่สุดที่ไม่เกิน 6 คือ 4 ดังนั้นช่วง [1,6] จะเกิดจากการ union กันของ [1,4] และ [3,6]



- ดังนั้น $\min_q(1,6) = \min(\min_q(1, 4), \min_q(3, 6)) = \min(1,1) = 1$

- ลอง implement ดู
- หากข้อมูลเป็น int a[] = {7, 2, 3, 0, 5, 10, 3, 12, 18};
- เมื่อเรียก
- cout<<query(0, 4)<<endl;
- cout<<query(4, 7)<<endl;
- cout<<query(7, 8)<<endl;
- ควรได้ผลลัพธ์เท่าไร