<https://leetcode.com/problems/longest-increasing-subsequence-ii/description/>

You are given an integer array nums and an integer k.

Find the longest subsequence of nums that meets the following requirements:

* The subsequence is **strictly increasing** and
* The difference between adjacent elements in the subsequence is **at most** k.

Return *the length of the* ***longest******subsequence*** *that meets the requirements.*

A **subsequence** is an array that can be derived from another array by deleting some or no elements without changing the order of the remaining elements.

**Example 1:**

Input: nums = [4,2,1,4,3,4,5,8,15], k = 3

Output: 5

Explanation:

The longest subsequence that meets the requirements is [1,3,4,5,8].

The subsequence has a length of 5, so we return 5.

Note that the subsequence [1,3,4,5,8,15] does not meet the requirements because 15 - 8 = 7 is larger than 3.

**Example 2:**

Input: nums = [7,4,5,1,8,12,4,7], k = 5

Output: 4

Explanation:

The longest subsequence that meets the requirements is [4,5,8,12].

The subsequence has a length of 4, so we return 4.

**Example 3:**

Input: nums = [1,5], k = 1

Output: 1

Explanation:

The longest subsequence that meets the requirements is [1].

The subsequence has a length of 1, so we return 1.

**Constraints:**

* 1 <= nums.length <= 105
* 1 <= nums[i], k <= 105

**Attempt 1: 2023-05-11**

**Solution 1: DP (10 min, TLE)**

class Solution {

public int lengthOfLIS(int[] nums, int k) {

int result = 1;

int len = nums.length;

int[] dp = new int[len];

Arrays.fill(dp, 1);

for(int i = 0; i < len; i++) {

for(int j = 0; j < i; j++) {

if(nums[j] < nums[i] && nums[i] - nums[j] <= k) {

dp[i] = Math.max(dp[i], dp[j] + 1);

}

}

result = Math.max(result, dp[i]);

}

return result;

}

}

Time Complexity : O(N^2)

Space Complexity : O(N)

**The DP solution is very similar to L300.Longest Increasing Subsequence, the only difference is add one more condition as "nums[i] - nums[j] <= k" based on new restriction from problem description, but TLE on 71/83**

**Why can't we use DP**

**Refer to**

<https://leetcode.com/problems/longest-increasing-subsequence-ii/solutions/2560316/why-can-t-we-use-dp/>

Test cases too heavy, so n^2 wouldn't pass, segment trees have Time Complexity -> O(NlogN) Space Complexity -> O(NlogN), I tries to use most optimum solution from here: <https://leetcode.com/problems/longest-increasing-subsequence/discuss/1326552/Optimization-From-Brute-Force-to-Dynamic-Programming-Explained>! but couldnt make it work for this question

**Solution 2: Segment Tree (60 min)**

**Style 1: Iterative version of Segment Trees**

**Approach 1: Query range exclusive on right as [left, right)**

class Solution {

// n = 100001 means each potential value is

// an array index based on 1 <= nums[i] <= 10^5

int n = 100001;

// n \* 2 is iterative way of segment tree length

int[] seg = new int[n \* 2];

public int lengthOfLIS(int[] nums, int k) {

int max = 0;

for(int i = 0; i < nums.length; i++) {

int l = Math.max(0, nums[i] - k);

int r = nums[i];

// Max result for the current element,

// plus 1 for adding current element on length

int cur\_max = query(l, r) + 1;

max = Math.max(max, cur\_max);

update(nums[i], cur\_max);

}

return max;

}

private int query(int left, int right) { // query max [lo, hi)

left += n;

right += n;

int result = 0;

while(left < right) {

if(left % 2 == 1) {

result = Math.max(result, seg[left]);

left++;

left /= 2;

} else {

left /= 2;

}

if(right % 2 == 1) {

right--;

result = Math.max(result, seg[right]);

right /= 2;

} else {

right /= 2;

}

}

return result;

}

private void update(int index, int val) {

index += n;

seg[index] = val;

while(index > 0) {

int left = index;

int right = index;

if(index % 2 == 0) {

right = index + 1;

} else {

left = index - 1;

}

seg[index / 2] = Math.max(seg[left], seg[right]);

index /= 2;

}

}

}

**Approach 2: Query range inclusive as [left, right]**

class Solution {

// n = 100001 means each potential value is

// an array index based on 1 <= nums[i] <= 10^5

int n = 100001;

// n \* 2 is iterative way of segment tree length

int[] seg = new int[n \* 2];

public int lengthOfLIS(int[] nums, int k) {

int max = 0;

for(int i = 0; i < nums.length; i++) {

int l = Math.max(0, nums[i] - k);

// Query style same as L307, the right boundary must

// be an inclusive number so "-1"

//int r = nums[i];

int r = nums[i] - 1;

// Max result for the current element,

// plus 1 for adding current element on length

int cur\_max = query(l, r) + 1;

max = Math.max(max, cur\_max);

update(nums[i], cur\_max);

}

return max;

}

// Query style same as L307. Range Sum Query, require left

// and right boundary inclusively as [left, right]

// e.g To search maximum length for element 4 as seg[4],

// which based on maximum length among seg[1], seg[2], seg[3]

// the query boundary of left, right should base on [1, 3]

// rather than [1, 4)

private int query(int left, int right) {

left += n;

right += n;

int result = 0;

while(left <= right) {

if(left % 2 == 1) {

result = Math.max(result, seg[left]);

left++;

left /= 2;

} else {

left /= 2;

}

if(right % 2 == 0) {

result = Math.max(result, seg[right]);

right--;

right /= 2;

} else {

right /= 2;

}

}

return result;

}

private void update(int index, int val) {

index += n;

seg[index] = val;

while(index > 0) {

int left = index;

int right = index;

if(index % 2 == 0) {

right = index + 1;

} else {

left = index - 1;

}

seg[index / 2] = Math.max(seg[left], seg[right]);

index /= 2;

}

}

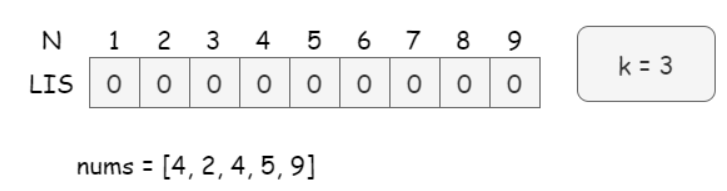
}

**Example to explain how LIS[a] = 1 + max(LIS[a - k : a]) works**

**Refer to**

<https://leetcode.com/problems/longest-increasing-subsequence-ii/solutions/2560085/python-explanation-with-pictures-segment-tree/>

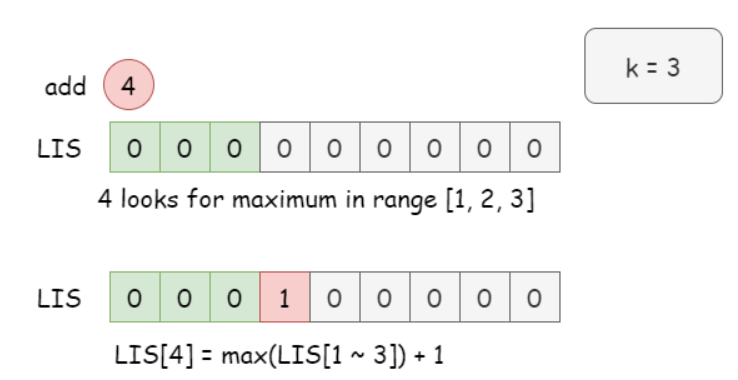
We store the longest increasing subsequence ended by each number in array LIS **(1-indexed)**. Let's say the input nums = [4,2,4,5,9]. Initially, LIS[i] = 0 as we haven't add any number yet.



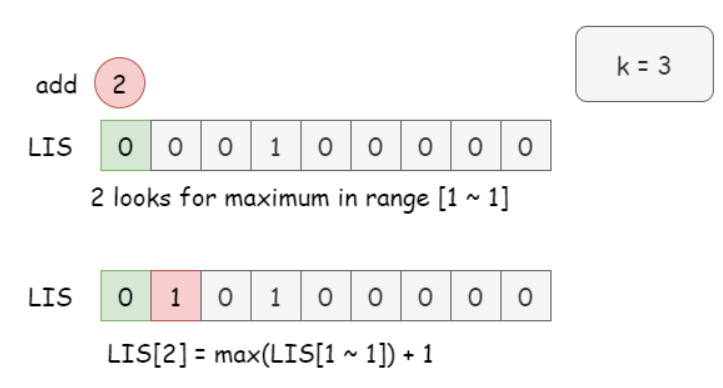
The key is for a given value a, we should find the maximum value from LIS[a - k: a] ), then LIS[a] = 1 + max(LIS[a - k : a]).

Take the pictures below as an example:

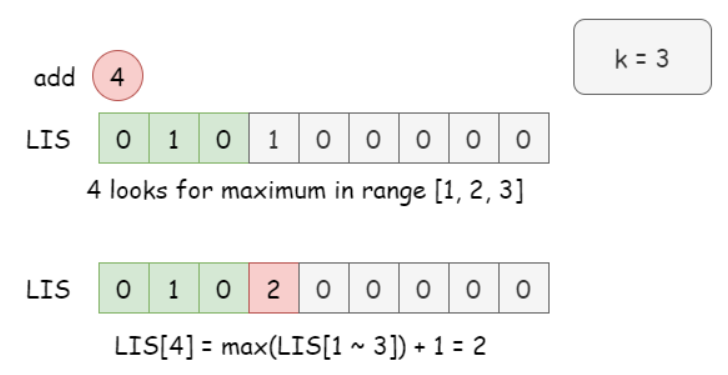
For the first number 4, the maximum length is the maximum of LIS[1], LIS[2], LIS[3] plus 1 (4 itself). Thus we shall look for the max(LIS[1:4]) . Apparently, LIS[4] = 1 which stands for 4 itself.



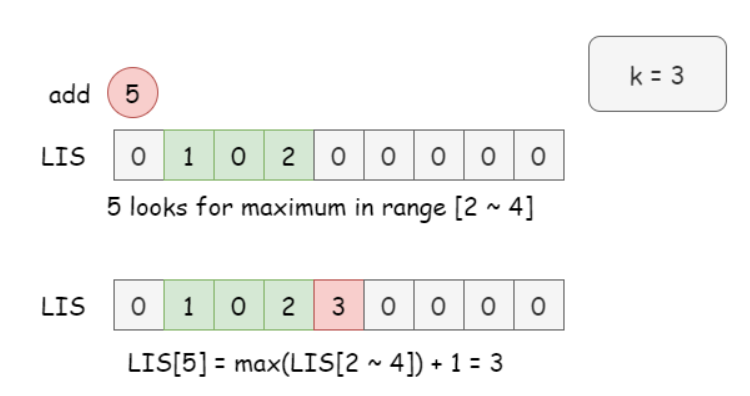
Then update LIS for 2, we shall look for max(LIS[1:2]), notice the corner case that left end always larger than 0.



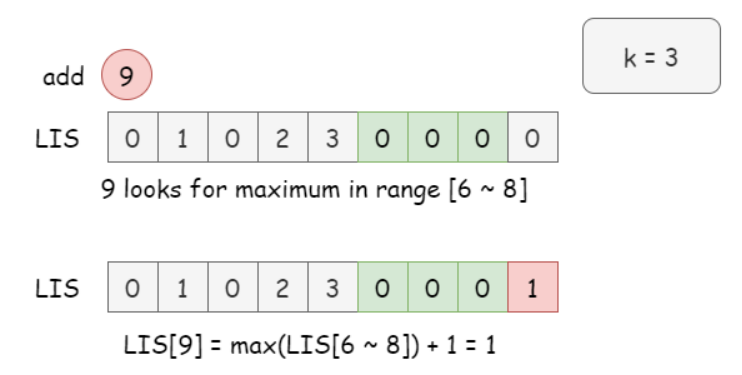
Then update LIS for 4, we look for max(LIS[1:4]), since there is an 2 updated in LIS, thus the maximum value from the same range LIS[1:4] gives us 1. Then we can update LIS[4] = 2, implying a subsequence of 2, 4.



Then update LIS for 5, we look for max(LIS[2:5]), LIS[5] = LIS[4] + 1 = 3, implying a subsequence of 2, 4, 5.



Then update LIS for 9, we look for max(LIS[6:9]).



so on so forth...

However, brute force ends up with O(n^2) time, we shall look for a better approach.

Its range queries of min/max value, thus we can use segment tree.

**Java implementation Example (Same as approach 1)**

**Refer to**

<https://leetcode.com/problems/longest-increasing-subsequence-ii/solutions/2560103/c-java-segment-tree-max-range-query/>

class Solution {

int N = 100001;

int[] seg = new int[2\*N];

void update(int pos, int val){ // update max

pos += N;

seg[pos] = val;

while (pos > 1) {

pos >>= 1;

seg[pos] = Math.max(seg[2\*pos], seg[2\*pos+1]);

}

}

/\*\*

Explain of behavior of while loop below:

I was confused by l & 1 and r & 1, and understand from the codeforces:

If l, the left interval border, is odd (which is equivalent to l&1) then l is the right child of its parent.

Then our interval includes node l but doesn't include it's parent. So we add t[l] and move to the right of

l's parent by setting l = (l + 1) / 2. If l is even, it is the left child, and the interval includes its

parent as well (unless the right border interferes), so we just move to it by setting l = l / 2.

Similar argumentation is applied to the right border. We stop once borders meet.

\*/

int query(int lo, int hi){ // query max [lo, hi)

lo += N;

hi += N;

int res = 0;

while (lo < hi) {

if ((lo & 1)==1) {

res = Math.max(res, seg[lo++]);

}

if ((hi & 1)==1) {

res = Math.max(res, seg[--hi]);

}

lo >>= 1;

hi >>= 1;

}

return res;

}

public int lengthOfLIS(int[] A, int k) {

int ans = 0;

for (int i = 0; i < A.length; ++i){

int l = Math.max(0, A[i]-k);

int r = A[i];

int res = query(l, r) + 1; // best res for the current element

ans = Math.max(res, ans);

update(A[i], res); // and update it here

}

return ans;

}

}

**Note:**

**The above article is just an example, not showing the actual segment tree resize and recursively update till top node structure, the actual segment tree implementation steps below:**

**Approach 1 Query range exclusive on right as [left, right) demo**

class Solution {

// n = 100001 means each potential value is

// an array index based on 1 <= nums[i] <= 10^5

int n = 100001;

// n \* 2 is iterative way of segment tree length

int[] seg = new int[n \* 2];

public int lengthOfLIS(int[] nums, int k) {

int max = 0;

for(int i = 0; i < nums.length; i++) {

int l = Math.max(0, nums[i] - k);

int r = nums[i];

// Max result for the current element,

// plus 1 for adding current element on length

int cur\_max = query(l, r) + 1;

max = Math.max(max, cur\_max);

update(nums[i], cur\_max);

}

return max;

}

private int query(int left, int right) { // query max [lo, hi)

left += n;

right += n;

int result = 0;

while(left < right) {

if(left % 2 == 1) {

result = Math.max(result, seg[left]);

left++;

left /= 2;

} else {

left /= 2;

}

if(right % 2 == 1) {

right--;

result = Math.max(result, seg[right]);

right /= 2;

} else {

right /= 2;

}

}

return result;

}

private void update(int index, int val) {

index += n;

seg[index] = val;

while(index > 0) {

int left = index;

int right = index;

if(index % 2 == 0) {

right = index + 1;

} else {

left = index - 1;

}

seg[index / 2] = Math.max(seg[left], seg[right]);

index /= 2;

}

}

}

================================================================

For demo purpose setup N = 10 instead of 100001

nums={4,2,4,5,9}

================================================================

1st round:

i=0, k=3, l=1, r=4

seg=[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

After query:

query(l,r)=query(1,4)=0

res=query(1,4)+1=0+1=1

After update:

update(A[i],res)=update(A[0],res)=update(4,1)

seg=[0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]

-----------------------------------------------------------------

2nd round:

i=1, k=3, l=0, r=2

seg=[0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]

After query:

query(l,r)=query(0,2)=0

res=query(0,2)+1=0+1=1

After update:

update(A[i],res)=update(A[1],res)=update(2,1)

seg=[0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0]

-----------------------------------------------------------------

3rd round:

i=2, k=3, l=1, r=4

seg=[0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]

After query:

query(l,r)=query(1,4)=1

res=query(2,4)+1=1+1=2

After update:

update(A[i],res)=update(A[2],res)=update(4,2)

seg=[0, 2, 0, 2, 0, 0, 1, 2, 0, 0, 0, 0, 1, 0, 2, 0, 0, 0, 0, 0]

-----------------------------------------------------------------

4th round:

i=3, k=3, l=2, r=5

seg=[0, 2, 0, 2, 0, 0, 1, 2, 0, 0, 0, 0, 1, 0, 2, 0, 0, 0, 0, 0]

After query:

query(l,r)=query(2,5)=2

res=query(2,4)+1=2+1=3

After update:

update(A[i],res)=update(A[3],res)=update(5,3)

seg=[0, 3, 0, 3, 0, 0, 1, 3, 0, 0, 0, 0, 1, 0, 2, 3, 0, 0, 0, 0]

-----------------------------------------------------------------

5th round:

i=4, k=3, l=6, r=9

seg=[0, 3, 0, 3, 0, 0, 1, 3, 0, 0, 0, 0, 1, 0, 2, 3, 0, 0, 0, 0]

After query:

query(l,r)=query(6,9)=0

res=query(6,9)+1=0+1=1

After update:

update(A[i],res)=update(A[4],res)=update(9,1)

seg=[0, 3, 1, 3, 1, 0, 1, 3, 0, 1, 0, 0, 1, 0, 2, 3, 0, 0, 0, 1]

-----------------------------------------------------------------

We see the segment tree array is quite different than the article example

**Approach 2 Query range inclusive as [left, right] demo**

class Solution {

// n = 100001 means each potential value is

// an array index based on 1 <= nums[i] <= 10^5

int n = 100001;

// n \* 2 is iterative way of segment tree length

int[] seg = new int[n \* 2];

public int lengthOfLIS(int[] nums, int k) {

int max = 0;

for(int i = 0; i < nums.length; i++) {

int l = Math.max(0, nums[i] - k);

// Query style same as L307, the right boundary must

// be an inclusive number so "-1"

//int r = nums[i];

int r = nums[i] - 1;

// Max result for the current element,

// plus 1 for adding current element on length

int cur\_max = query(l, r) + 1;

max = Math.max(max, cur\_max);

update(nums[i], cur\_max);

}

return max;

}

// Query style same as L307. Range Sum Query, require left

// and right boundary inclusively as [left, right]

// e.g To search maximum length for element 4 as seg[4],

// which based on maximum length among seg[1], seg[2], seg[3]

// the query boundary of left, right should base on [1, 3]

// rather than [1, 4)

private int query(int left, int right) { // query max [lo, hi]

left += n;

right += n;

int result = 0;

while(left <= right) {

if(left % 2 == 1) {

result = Math.max(result, seg[left]);

left++;

left /= 2;

} else {

left /= 2;

}

if(right % 2 == 0) {

result = Math.max(result, seg[right]);

right--;

right /= 2;

} else {

right /= 2;

}

}

return result;

}

private void update(int index, int val) {

index += n;

seg[index] = val;

while(index > 0) {

int left = index;

int right = index;

if(index % 2 == 0) {

right = index + 1;

} else {

left = index - 1;

}

seg[index / 2] = Math.max(seg[left], seg[right]);

index /= 2;

}

}

public static void main(String[] args) {

int[] nums = new int[] {4,2,4,5,9};

Solution s = new Solution();

int result = s.lengthOfLIS(nums, 3);

System.out.println(result);

}

}

================================================================

For demo purpose setup N = 10 instead of 100001

nums={4,2,4,5,9}

-----------------------------------------------------------------

1st round:

i=0, k=3, l=1, r=3

seg=[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

After query:

query(l,r)=query(1,3)=0

res=query(1,3)+1=0+1=1

After update:

update(A[i],res)=update(A[0],res)=update(4,1)

seg=[1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]

compare to [left,right) approach query(1,4)

seg=[0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]

the difference is seg[0]=1

-----------------------------------------------------------------

2nd round:

i=1, k=3, l=0, r=1

seg=[1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]

After query:

query(l,r)=query(0,1)=0

res=query(0,1)+1=0+1=1

After update:

update(A[i],res)=update(A[1],res)=update(2,1)

seg=[1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0]

compare to [left,right) approach query(0,1)

seg=[0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0]

the difference is seg[0]=1

-----------------------------------------------------------------

3rd round:

i=2, k=3, l=1, r=3

seg=[1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0]

After query:

query(l,r)=query(1,3)=1

res=query(1,3)+1=1+1=2

After update:

update(A[i],res)=update(A[2],res)=update(4,2)

seg=[2, 2, 0, 2, 0, 0, 1, 2, 0, 0, 0, 0, 1, 0, 2, 0, 0, 0, 0, 0]

compare to [left,right) approach query(1,4)

seg=[0, 2, 0, 2, 0, 0, 1, 2, 0, 0, 0, 0, 1, 0, 2, 0, 0, 0, 0, 0]

the difference is seg[0]=2

-----------------------------------------------------------------

4th round:

i=3, k=3, l=2, r=4

seg=[2, 2, 0, 2, 0, 0, 1, 2, 0, 0, 0, 0, 1, 0, 2, 0, 0, 0, 0, 0]

After query:

query(l,r)=query(2,4)=2

res=query(2,4)+1=2+1=3

After update:

update(A[i],res)=update(A[3],res)=update(5,3)

seg=[3, 3, 0, 3, 0, 0, 1, 3, 0, 0, 0, 0, 1, 0, 2, 3, 0, 0, 0, 0]

compare to [left,right) approach query(2,4)

seg=[0, 3, 0, 3, 0, 0, 1, 3, 0, 0, 0, 0, 1, 0, 2, 3, 0, 0, 0, 0]

the difference is seg[0]=3

-----------------------------------------------------------------

5th round:

i=4, k=3, l=6, r=8

seg=[3, 3, 0, 3, 0, 0, 1, 3, 0, 0, 0, 0, 1, 0, 2, 3, 0, 0, 0, 0]

After query:

query(l,r)=query(6,8)=0

res=query(6,8)+1=0+1=1

After update:

update(A[i],res)=update(A[4],res)=update(9,1)

seg=[3, 3, 1, 3, 1, 0, 1, 3, 0, 1, 0, 0, 1, 0, 2, 3, 0, 0, 0, 1]

compare to [left,right) approach query(6,8)

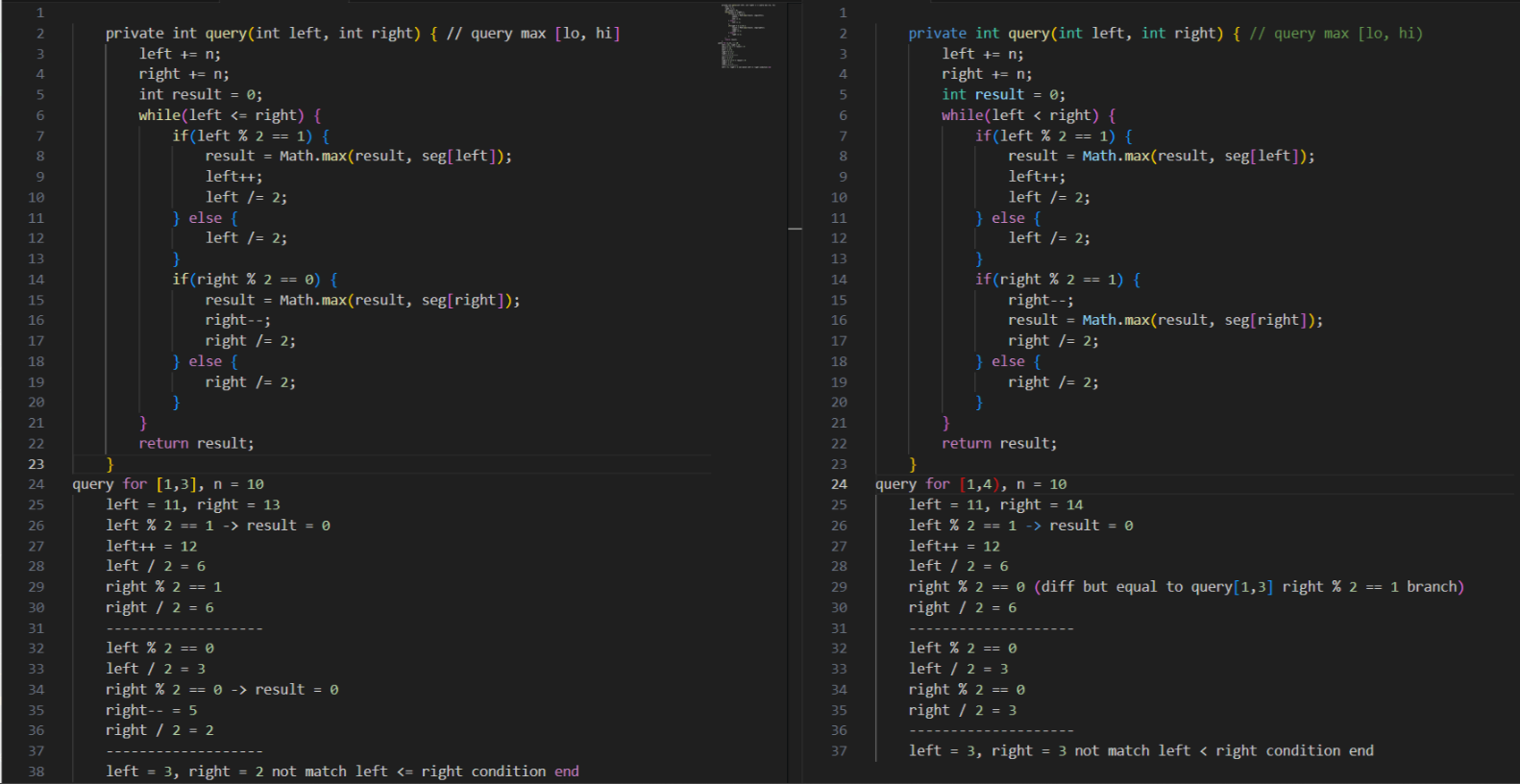
seg=[0, 3, 1, 3, 1, 0, 1, 3, 0, 1, 0, 0, 1, 0, 2, 3, 0, 0, 0, 1]

the difference is seg[0]=3

-----------------------------------------------------------------

We see the segment tree array is quite different than the article example

**The difference between query [left, right) and query [left, right]**



**Take example for equal query on seg[4] = Max (seg[1], seg[2], seg[3]) situation, in query [left, right) will be query [1, 4), but in query[left, right] will be query [1, 3]**

**query [left, right] => query [1, 3]**

private int query(int left, int right) { // query max [lo, hi]

left += n;

right += n;

int result = 0;

while(left <= right) {

if(left % 2 == 1) {

result = Math.max(result, seg[left]);

left++;

left /= 2;

} else {

left /= 2;

}

if(right % 2 == 0) {

result = Math.max(result, seg[right]);

right--;

right /= 2;

} else {

right /= 2;

}

}

return result;

}

query for [1,3], n = 10

left = 11, right = 13

left % 2 == 1 -> result = 0

left++ = 12

left / 2 = 6

right % 2 == 1

right / 2 = 6

-------------------

left % 2 == 0

left / 2 = 3

right % 2 == 0 -> result = 0

right-- = 5

right / 2 = 2

-------------------

left = 3, right = 2 not match left <= right condition end

**query [left, right) => query [1, 4)**

private int query(int left, int right) { // query max [lo, hi)

left += n;

right += n;

int result = 0;

while(left < right) {

if(left % 2 == 1) {

result = Math.max(result, seg[left]);

left++;

left /= 2;

} else {

left /= 2;

}

if(right % 2 == 1) {

right--;

result = Math.max(result, seg[right]);

right /= 2;

} else {

right /= 2;

}

}

return result;

}

query for [1,4), n = 10

left = 11, right = 14

left % 2 == 1 -> result = 0

left++ = 12

left / 2 = 6

right % 2 == 0 (diff but equal to query[1,3] right % 2 == 1 branch)

right / 2 = 6

--------------------

left % 2 == 0

left / 2 = 3

right % 2 == 0

right / 2 = 3

--------------------

left = 3, right = 3 not match left < right condition end

**The comparison here is in query [left, right) when right boundary identified as a right child node, we cannot use it to directly find its parent node, because "right)" is exclusive, to find a valid parent node based on given right boundary, we have to minus 1 (right--) first to find the actual valid right child inclusively as right boundary and then move on to find its parent node. That's why its logic is different than query [left, right]**

**Style 2: Recursive version of Segment Trees**

**The same style as L307. Range Sum Query Mutable recursive way**

class Solution {

int[] tree;

// Critical point: Different than L307. Range Sum Query

// The segment tree length is build on 4 \* (N = 100001) not 4 \* nums.length

// Because we create one entry for each potential value in an array index based

// on 1 <= nums[i] <= 10^5

int N = 100001;

public int lengthOfLIS(int[] nums, int k) {

int max = 0;

int n = nums.length;

//tree = new int[n \* 4]; --> This is wrong!

tree = new int[N \* 4];

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

int l = Math.max(0 , nums[i] - k);

int r = nums[i] - 1;

// Search in all the possible previous elements ([l , r]) and add '1' to the max

//int res = query(l , r , 0 , N - 1 , 0) + 1; --> Move to queryHelper(...) method

int res = query(l , r) + 1;

max = Math.max(max , res); // update max

// Update segment tree's nums[i]th index with res

//update(0 , res , 0 , N - 1 , nums[i]); --> Move to updateHelper(...) method

update(nums[i], res);

}

return max;

}

public void update(int index, int val) {

updateHelper(0, 0, N - 1, index, val);

}

private void updateHelper(int treeIndex, int lo, int hi, int index, int val) {

// Leaf node, update element

if(lo == hi) {

tree[treeIndex] = Math.max(tree[treeIndex], val);

return;

}

int mid = lo + (hi - lo) / 2;

if(index > mid) {

updateHelper(2 \* treeIndex + 2, mid + 1, hi, index, val);

} else {

updateHelper(2 \* treeIndex + 1, lo, mid, index, val);

}

tree[treeIndex] = Math.max(tree[2 \* treeIndex + 1], tree[2 \* treeIndex + 2]);

}

public int query(int left, int right) {

return queryHelper(0, 0, N - 1, left, right);

}

private int queryHelper(int treeIndex, int lo, int hi, int left, int right) {

if(lo > right || hi < left) {

return 0;

}

if(lo >= left && hi <= right) {

return tree[treeIndex];

}

int mid = lo + (hi - lo) / 2;

if(left > mid) {

return queryHelper(treeIndex \* 2 + 2, mid + 1, hi, left, right);

} else if(right <= mid) {

return queryHelper(treeIndex \* 2 + 1, lo, mid, left, right);

} else {

int leftMax = queryHelper(treeIndex \* 2 + 1, lo, mid, left, mid);

int rightMax = queryHelper(treeIndex \* 2 + 2, mid + 1, hi, mid + 1, right);

return Math.max(leftMax, rightMax);

}

// Same effect as below

//int leftMax = queryHelper(treeIndex \* 2 + 1, lo, mid, left, right);

//int rightMax = queryHelper(treeIndex \* 2 + 2, mid + 1, hi, left, right);

//return Math.max(leftMax, rightMax);

}

}

**Refer to**

<https://leetcode.com/problems/longest-increasing-subsequence-ii/solutions/2560747/java-easy-segment-tree/>

class Solution {

int N = 1\_00\_001;

int seg[];

void update(int idx , int x , int low , int high , int i){

if(low == high){

seg[idx] = x;

return;

}

int mid = low + (high - low) / 2;

if(i <= mid){

update(2 \* idx + 1 , x , low , mid , i);

}

else{

update(2 \* idx + 2 , x , mid + 1 , high , i);

}

seg[idx] = Math.max(seg[2 \* idx + 1], seg[2 \* idx + 2]);

}

int query(int l , int r , int low , int high , int idx){ // max query

if(l > high || r < low){

return Integer.MIN\_VALUE;

}

if(low >= l && high <= r){

return seg[idx];

}

int mid = low + (high - low) / 2;

int left = query(l , r , low , mid , 2 \* idx + 1);

int right = query(l , r , mid + 1 , high , 2 \* idx + 2);

return Math.max(left , right);

}

public int lengthOfLIS(int[] a, int k) {

int n = a.length;

int max = 0;

seg = new int[4 \* N];

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

int l = Math.max(0 , a[i] - k);

int r = a[i] - 1;

int res = query(l , r , 0 , N - 1 , 0) + 1; // search in all the possible previous elements ([l , r]) and add '1' to the max length with this previous

max = Math.max(max , res); // update max

update(0 , res , 0 , N - 1 , a[i]); // update segment tree's a[i]th index with res

}

return max;

}

}

**Why cannot use Fenwick Tree (Binary Indexed Tree) ?**

**Refer to**

<https://leetcode.com/problems/longest-increasing-subsequence-ii/solutions/2560103/c-java-segment-tree-max-range-query/>

#### **Strategy**

For every element A[i], we check the range [A[i]-k, A[i]-1] for the current best length and add 1. The problem here is that, we also need to update the max and it is not a prefix range from 0 to some number, so Fenwick tree won't work.

I figure segment tree is good for this because it supports update and query in both O(logn).

Time O(log(max(A[i])) \* N)

Space O(max(A[i]))

<https://leetcode.com/problems/longest-increasing-subsequence-ii/solutions/2578352/segment-tree/>

Can be solved by DP, where dp[nums[i]] = max(dp[1] ... dp[nums[i] - 1]).

However, this leads to a quadratic solution, and we need to do it in O(n log n) based on the problem constraints.

I wasted a lot of time trying to adapt a monotonic stack LIS solution, but only to realize that we must use DP.

We just need to make the max range query to run in O(log n) instead of O(n). This can be done using a segment tree.

**Note that, for canonical LIS, we can also use the Fenwick tree. With k, Fenwick tree would not work as it can only answer min/max queries on the [0, r] interval (min/max are not commutative operations).**

It's possible to use a pair of Fenwick trees to answer mini/max queries on [l, r] interval, but the solution would be much more complicated, compared to a segment tree.