

6.4.6 Longest Increasing Subsequence

Consider the following problem:

Given an array of integers find the longest subsequence³⁸ which is strictly increasing.

As one might surmise, this is canonically known as the *longest increasing subsequence* problem, or *LIS*. Consider the sequence 17, 8, 22, 7, 1, 10, 3, 25, 14, 19. The LIS is 8, 10, 14, 19 for length 4.³⁹

Approaching this problem naively, we might choose each element of the output array as a candidate "last" value of an optimum sequence, and then recurse on every prior value of the array which has a lesser value, recording the longest such path:

```
//assume array 'a'

int LIS_recursion(int end){
    int ans=1;
    for(int i=0;i<end;i++) if(a[i]<a[end]) ans=max(ans,1+recurse(i));
    return ans;
}

int LIS(){
    int ans=0;
    for(int i=0;i<n;i++) ans=max(ans,LIS_recursion(i));
}
```

We can simplify this implementation by adding some absurdly large integer to the end of the array, ensuring any increasing subsequence will end there (rather than at an earlier element in the array). In either case, this implementation iterates over all possible increasing sequences in $O(2^n)$ time.⁴⁰

Using similar intuition as was used for TSP, we note that when recursing, the LIS ending at a particular index is not impacted by any future indices, and yet we compute it multiple times. This leads us to a straightforward DP:

1. Indices: The index into the array.
2. Value: The LIS ending at that index.
3. Relation: As the earlier naive code demonstrates, we examine each earlier index with a value less than this one, and take the maximum length, adding one to represent extending that sequence by one.

³⁸selection of elements from the input sequence which maintains their relative ordering

³⁹Note there are other length-4 increasing subsequences. All are equally valid.

⁴⁰In the worst case, the input array is strictly increasing. The code will traverse every possible increasing subsequence. As every subsequence is valid, we are left with 2^n possible selections of nodes.

4. Iteration Order: As the relation refers to earlier indices, the iteration order goes in index order.
5. Base Cases: The minimum length of the LIS ending at any node is 1. You can always select an individual node, and it is an increasing subsequence of length 1.

```
//assume array 'a'
int dp[n]; //LIS ending at a given index
fill(dp,dp+n,1); //base case
for(int i=0;i<n;i++) //loop over all indices
    for(int j=0;j<i;j++)if(a[j]<a[i]) //loop over all earlier indices less
        than us
        dp[i]=max(dp[i],dp[j]+1); //see if this gives us a better sequence

//LIS is the largest value in the dp array
```

As the size of the DP array is n and we perform $O(n)$ work computing each entry, the overall runtime is $O(n^2)$, significantly faster than the earlier naive solution. The solution can be readily extended to recover the actual sequence using standard path recovery techniques:

```
//assume array 'a'
int dp[n],prev[n]; //LIS ending at a given index
fill(dp,dp+n,1); //base case
for(int i=0;i<n;i++) //loop over all indices
    for(int j=0;j<i;j++)if(a[j]<a[i]) //loop over all earlier indices less
        than us
        if(dp[j]+1<dp[i])prev[i]=j,dp[i]=dp[j]+1;
```

Or to count the number of ways the longest subsequence can be found, again using standard path counting techniques.

```
//assume array 'a'
int dp[n],count[n]={0}; //LIS ending at a given index
fill(dp,dp+n,1); //base case
for(int i=0;i<n;i++) //loop over all indices
    for(int j=0;j<i;j++)if(a[j]<a[i]) //loop over all earlier indices less
        than us
        if(dp[j]+1<dp[i])dp[i]=dp[j]+1,count[i]=count[j];
        else if(dp[j]+1==dp[i])count[i]+=count[j];
```

6.4.6.1 Optimization with Segment Trees

With a slight modification to the above algorithm, we can adapt it for use with a segment tree. The inner loop of the algorithm requires iterating over all previous indices to identify the valid one with the maximum LIS. This would be a

standard maximum range query, solveable via standard range query techniques, however we have the unfortunate additional condition that the previous value must be less than the one we are examining.⁴¹

This can be tackled by a simple adjustment whereby instead of indexing our DP array on the index of the input array, we instead use the value of input array.

```
//assume array 'a'
int dp[MAX_VALUE]={}; //LIS ending at a given value
for(int i=0;i<n;i++) //loop over all indices
    for(int j=0;j<a[i];j++) //loop over all values less than us
        dp[a[i]]=max(dp[a[i]],dp[j]+1); //see if this gives us a better
        sequence

//LIS is the largest value in the dp array
```

At face value, this may make performance slower, as `MAX_VALUE` may exceed `n`. This can be adjusted trivially using range compression to bring the size of the range to at most `n`, leaving us with the same $O(n^2)$. With the above adjustment, the inner loop becomes exactly a max range query over all values less than the current value. We can replace the loop with a max segtree.

- The segments of the tree are the values of the input array.⁴²
- The segment tree stores the length of the LIS ending at any index with that particular value, and aggregates the max over all the segments.
- Lookup involves querying the max LIS ending at any value less than the current value (i.e. `a[i]`).
- If the value is better than the previous LIS ending at the given value, the segment tree is updated.

```
//assume range compressed array 'a'
//assume max segtree, st, with range-query and point query/update

for(int i=0;i<n;i++){
    int max_preceding=st.range_query(0,a[i]);
    if(max_preceding+1>st.point_query(a[i]))st.point_update(a[i],max_preceding);
}

//LIS is the largest value in the entire segtree range
```

As the inner loop is now $O(\log(n))$ with the segtree query, the overall runtime is $O(n * \log(n))$. A full discussion of segment trees and usage is covered in the data structures chapter.

It is possible to extend this solution:

⁴¹by definition of an increasing subsequence

⁴²potentially range compressed

- Counting LIS is accomplished by using a segtree which counts the number of maxima which occur on a range, with slight adjustments to the point call to ensure all cases are tracked.⁴³
- Recovering LIS is similar to the unoptimized version, where for each index we store the previous index in the LIS. To recover that information, we must store in the segtree which index last set any particular value, returning that value with the maximum range query.

6.4.6.2 LIS in Faster $O(n * \log(n))$

While we already have an efficient solution, the coefficient for operating on a segtree may be high enough to exceed tight time limits. We present a non-segtree based solution which typically performs better, often known as the *binary-search* based approach.

In order to apply a binary search, we have to consider how to reframe our dp. Recall the technique of dimension swapping we leveraged for certain knapsack problems. We will apply a similar technique here.

Our initial attempt used indices as a dimension, and stored the maximum appropriate subsequence length. With the swap, we will use the length of the sequence as the index, but what should be our value? Assuming we continue to iterate by adding one index at a time, what information have we lost in this swap? When we performed the inner loop to look at the previous index which could be in a LIS, the information we actually depended on was the magnitude of that value. Ultimately, we don't care where in the subsequence that value occurs, only that the value is less than the value of the current node and the LIS ends up being longer. We can re-acquire this information by storing it as the value of our array DP array.

From a high level, instead of storing the max length subsequence which terminates at a given index, we will store the values of the end of the subsequences of a given length. It may sound prohibitive to store all such values, so notice that we need only store the minimum. If there is a sequence of length x which terminates at 17, and one of the same length which terminates at 12, for all future indices, there is no value which could use the 17 as its previous sequence, but not the 12. We can greedily take the lower ending value when there is a choice.

1. Indices: The index we are on, and the length of the sequences we might have found.
2. Value: The magnitude of the number which is at the end of a sequence of the given length, after examining all indices up to here.
3. Relation: For a current node i and each subsequence length l , if the value $dp_{i-1,l}$ currently in the array is less than the value of the current index,

⁴³We must ensure case where a second instance of a LIS of length exactly x ending at a given value y is counted appropriately.

a_i , and this results in a lower value for the subsequence of length $l + 1$, $dp_{i-1,l+1}$, update $dp_{i,l+1}$ to our current value. The subsequence of length $l + 1$ ending at the current index makes it "easier" for future indices to add on. Otherwise, $dp_{i,l+1} = dp_{i-1,l+1}$.

4. Iteration Order: All relations refer to $i - 1$, so we go in order by index.
5. Base Case: The minimum value to reach any given LIS is infinite to start, However, we have a length 0 subsequence ending at any value, so can assign 0.⁴⁴

Let's walk through an example using our earlier sequence, 17, 8, 22, 7, 1, 10, 3, 25, 14, 19. In order to remove corner cases, we will prepend

Length \ index	init	0	1	2	3	4	5	6	7	8	9
0	0	?	?	?	?	?	?	?	?	?	?
1	∞	?	?	?	?	?	?	?	?	?	?
2	∞	?	?	?	?	?	?	?	?	?	?
3	∞	?	?	?	?	?	?	?	?	?	?
4	∞	?	?	?	?	?	?	?	?	?	?
5	∞	?	?	?	?	?	?	?	?	?	?
6	∞	?	?	?	?	?	?	?	?	?	?
7	∞	?	?	?	?	?	?	?	?	?	?
8	∞	?	?	?	?	?	?	?	?	?	?
9	∞	?	?	?	?	?	?	?	?	?	?
10	∞	?	?	?	?	?	?	?	?	?	?

Figure 6.21: Along the x-axis we have the dimension of the index. We will fill the array from left to right. Along the y-axis, we have the lengths of the LIS, and in the table, we will store the minimum value ending a sequence of that length. Before we begin, we have our base case of 0 for length 0, and infinite for any other length.

⁴⁴We can create a subsequence of length 1 with any value, so we encode this by indicating the minimum value which ends a subsequence of length 0 is 0.

Length \ index	init	0	1	2	3	4	5	6	7	8	9
0	0	0	?	?	?	?	?	?	?	?	?
1	∞	17	?	?	?	?	?	?	?	?	?
2	∞	∞	?	?	?	?	?	?	?	?	?
3	∞	∞	?	?	?	?	?	?	?	?	?
4	∞	∞	?	?	?	?	?	?	?	?	?
5	∞	∞	?	?	?	?	?	?	?	?	?
6	∞	∞	?	?	?	?	?	?	?	?	?
7	∞	∞	?	?	?	?	?	?	?	?	?
8	∞	∞	?	?	?	?	?	?	?	?	?
9	∞	∞	?	?	?	?	?	?	?	?	?
10	∞	∞	?	?	?	?	?	?	?	?	?

Figure 6.22: We first examine the value of 17 at index 0. The only length sequence which has an ending value of less than 17 is length 0, ending at value 0. If we extend this subsequence by appending a 17, we have a subsequence of length 1, ending at 17. The current best subsequence of length 1 ends at infinity, so we update this value. We cannot do better for sequences of any other length.

Length \ index	init	0	1	2	3	4	5	6	7	8	9
0	0	0	0	?	?	?	?	?	?	?	?
1	∞	17	8	?	?	?	?	?	?	?	?
2	∞	∞	∞	?	?	?	?	?	?	?	?
3	∞	∞	∞	?	?	?	?	?	?	?	?
4	∞	∞	∞	?	?	?	?	?	?	?	?
5	∞	∞	∞	?	?	?	?	?	?	?	?
6	∞	∞	∞	?	?	?	?	?	?	?	?
7	∞	∞	∞	?	?	?	?	?	?	?	?
8	∞	∞	∞	?	?	?	?	?	?	?	?
9	∞	∞	∞	?	?	?	?	?	?	?	?
10	∞	∞	∞	?	?	?	?	?	?	?	?

Figure 6.23: We examine the value of 8 at index 1. The only length subsequence with an ending value less than 8 is length 0, ending at value 0. If we extend this subsequence by appending an 8, we have a subsequence of length 1, ending at 8. The current best subsequence of length 1 ends at 17, so we update this value. We cannot do better for a subsequence of any other length.

Length \ index	init	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	?	?	?	?	?	?	?
1	∞	17	8	8	?	?	?	?	?	?	?
2	∞	∞	∞	22	?	?	?	?	?	?	?
3	∞	∞	∞	∞	?	?	?	?	?	?	?
4	∞	∞	∞	∞	?	?	?	?	?	?	?
5	∞	∞	∞	∞	?	?	?	?	?	?	?
6	∞	∞	∞	∞	?	?	?	?	?	?	?
7	∞	∞	∞	∞	?	?	?	?	?	?	?
8	∞	∞	∞	∞	?	?	?	?	?	?	?
9	∞	∞	∞	∞	?	?	?	?	?	?	?
10	∞	∞	∞	∞	?	?	?	?	?	?	?

Figure 6.24: We examine the value of 22 at index 2. As with the previous iterations, we could append to a subsequence of length 0 ending with 0, however if we did so, we would have a subsequence of length 1 ending with 22. This is greater than previous best ending value for a subsequence of length 1, 8, so we do not do better than previous. We can, however, append to the subsequence of length 1 ending with 8, leaving us with a subsequence of length 2 ending with 22, beating the previous best of infinity. All other values stay the same.

Length \ index	init	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	?	?	?	?	?
1	∞	17	8	8	7	1	?	?	?	?	?
2	∞	∞	∞	22	22	22	?	?	?	?	?
3	∞	∞	∞	∞	∞	∞	?	?	?	?	?
4	∞	∞	∞	∞	∞	∞	?	?	?	?	?
5	∞	∞	∞	∞	∞	∞	?	?	?	?	?
6	∞	∞	∞	∞	∞	∞	?	?	?	?	?
7	∞	∞	∞	∞	∞	∞	?	?	?	?	?
8	∞	∞	∞	∞	∞	∞	?	?	?	?	?
9	∞	∞	∞	∞	∞	∞	?	?	?	?	?
10	∞	∞	∞	∞	∞	∞	?	?	?	?	?

Figure 6.25: We skip ahead 2 steps. We can form a better subsequence of length 1, ending with 7 instead of 8. We can then do even better with the 1.

Length \ index	init	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0	0
1	∞	17	8	8	7	1	1	1	1	1	1
2	∞	∞	∞	22	22	22	10	3	3	3	3
3	∞	∞	∞	∞	∞	∞	∞	∞	25	14	14
4	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	17
5	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	inf
6	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	inf
7	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	inf
8	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	inf
9	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	inf
10	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	inf

Figure 6.26: We execute all remaining steps. The LIS which ends at a non-infinite value is 4, ending at a 17. We can further see the minimum ending value of a subsequence of length 3 is 14, length 2 is 3, and length 1 is 1.

```

//assume input array 'a'
int dp[n+1][n+1]; //index (including init) and length of sequence
fill(dp[0], dp[0]+n+1, INT32_MAX); //base cases
dp[0][0]=0;

for(int i=0; i<n; i++){ //loop through each index
    dp[i][0]=0; //base case
    for(int j=0; j<n; j++){ //loop through each length
        if(dp[i][j]<a[i]&& a[i]<dp[i][j+1]) dp[i+1][j+1]=a[i]; //did better
        else dp[i+1][j+1]=dp[i][j+1]; //didn't do better
    }
}

```

Our array has n^2 entries, and each one is updated in constant time. This leads to $O(n^2)$ runtime, but goal was $n * \log(n)$, so what gives? We make two optimizations.

First, we note that we cannot have an array of size n^2 if we ever expect to have a faster runtime. Fortunately, notice that rather than copying values from column to column, we can perform the updates in place. As all changes from column to column only impact a length of one greater, we can iterate from longer to shorter lengths without impacting the result.

```

//assume input array 'a'
int dp[n+1]; //length of sequence
fill(dp, dp+n+1, INT32_MAX); //base cases
dp[0]=0;

for(int i=0; i<n; i++){ //loop through each index
    for(int j=n-1; j>=0; j--){ //loop through each length

```

```

        if(dp[j]<a[i]&& a[i]<dp[j+1])dp[j+1]=a[i]; //did better
    }
}

```

While the runtime is still quadratic, we have set ourselves up for the second optimization. Astute readers may have noticed that from iteration to iteration, we only seem to update a single value. If that were the case, perhaps we could identify the exact element to modify in sublinear time.

Let's consider our relation a bit more closely. a_i must be strictly greater than the value we are trying to append to, and it must be strictly less than value of the sequence of length one greater. Intuitively, the end values for subsequences of a given length must be non-decreasing.⁴⁵ As the dp array increases monotonically at any point in time, there is at most one pair of consecutive values which can bound a given a_i .

Put another way, A given entry in our input array can extend at most one sequence. Suppose it did, and we are left with sequences of length l and $l + 1$ both terminating with a_i . In this case, we could take the a_i at the end of the $l + 1$ sequence and truncate it, leaving a length l sequence ending with some value less than a_i , violating our assertion that a_i was optimal for a sequence of that length.

Ultimately, we only have to find a single location in a monotonic array, and therefore, can binary search.

```

//assume input array 'a'
int dp[n+1];
fill(dp,dp+n+1,INT32_MAX);
dp[0]=0;
for(int i=0;i<n;i++){
    int delta=1;
    while(delta<n)delta<<=1;
    int j=0;//find j with binary search
    while(delta){
        if(j+delta<n&&dp[j+delta]<a[i])j+=delta;
        delta>>=1;
    }
    dp[j+1]=a[i]; //update! (maybe stays the same)
}

```

We have reached our goal. Our DP array is size n , and we perform $O(\log(n))$ work for each entry, leaving with us of an overall runtime of $O(n * \log(n))$.

Sequence Recovery The modified and optimized algorithm increases the complication of computing the actual subsequence. As with the $O(n^2)$ solution, our output ultimately needs to be an array indicating which index represents

⁴⁵Suppose we had a length $l + 1$ subsequence ending with a value v . We trivially have a subsequence of length l ending with value v by truncating the first element of the $l + 1$ subsequence.

the previous element in the LIS ending at a given location. Unfortunately, this information is not cached in our DP array where we only have lengths and values. When we set a new value in the array, we are adding the new element to the end of a sequence which is one shorter in length, so the information we need is the index of the last element of that one shorter sequence. We can resolve this by caching for each sequence length, the index which ends that sequence. Ultimately this leads to two auxiliary structures:

1. A "previous" array, which stores the index of the previous element of the LIS ending at a given index (as with the $O(n^2)$ solution)
2. The mapping of which index in the input array last set a particular value in the DP array

```

//assume input array 'a'
int dp[n+1];
fill(dp,dp+n+1,INT32_MAX);
dp[0]=0;
int cur_end[n+1],prev[n];//index which ends a given sequence in the DP
    array, and the prev index of the LIS ending at any location of the
    input
fill(cur_end,cur_end+n+1,-1);
for(int i=0;i<n;i++){
    int delta=1;
    while(delta<n)delta<<=1;
    int j=0;//find j with binary search
    while(delta){
        if(j+delta<n&&dp[j+delta]<a[i])j+=delta;
        delta>>=1;
    }
    if(dp[i]<dp[j+1]){
        dp[j+1]=a[i]; //update DP array
        prev[i]=cur_end[j]; //previous element of sequence indending at i
            is the current end of the sequence which has length j
        cur_end[j+1]=i; //we are the new current end of length j+1
    }
}

```

Counting Subsequences As with the other methods, from a high level we will compute the count of LIS terminating at each index as we go along. Recall that at its core a given element terminates an LIS if:

1. The LIS of the previous element is one fewer than the LIS of this element
2. The value of the previous element is less than the new element
3. The previous element occurs earlier in the input than the current element

The last condition is trivially met since we process elements in input order. The first two imply that we need some way to track all elements with a given LIS and the ability to perform a range sum over all elements with a lesser value than our current element. This could, of course, be done with a segment tree, but this antithetical do attempting to use binary search in the first place. Instead we can do it with a prefix sum for each length, which we can query after we have determined the LIS for each element to obtain the count of all elements which have lower value, and thus contribute to our sum.⁴⁶

This may seem impossible, as one of the core limitations of using prefix sums for range queries is that it is not dynamic. However, consider two facts:

1. In order to use a prefix sum to query on a range of values, the elements of the sum must be ordered by value. For instance, if we want to sum the counts of all items of a given LIS length of less than some value v , the elements in the prefix sum must be ordered by value. We can then binary search the correct index into this list and use the prefixsum to perform the range query over all items of value less than v .
2. Due to the recurrence relation used when computing LIS with the binary search method, the ending value of an LIS of a given length is non-increasing over the run of the algorithm. Recall we always update a single LIS, and it is guaranteed this new value is less than or equal to the previous best end of an LIS of that length.

Given these facts, we do have a monotonic list of values (per LIS length), and each update is an append to that list. As such, the prefix sum can be updated when we perform the append. This sum ultimately represents, for a given length, the count of subsequences that terminate at a value greater than the given element in the list. We can then compute the count that terminate at some value less than a given v by subtracting the prefix sum at v from the count of all sequences of the given length, which is prefix sum of the last element.

Lets walk through the same earlier sequence 17, 8, 22, 7, 1, 10, 3, 25, 14, 19.

⁴⁶We maintain a sum for any given LIS length, and after we have determined the LIS for a new element, we query the sum for the length one fewer.

LIS Length	Elements Prefix Sum
0	$-\infty$ 1
1	∞ 0
2	∞ 0
3	∞ 0
4	∞ 0

Figure 6.27: At the start, we seed each list to avoid corner cases, indicating the only valid sequence we have is length 0, ending at negative infinity.

LIS Length	Elements Prefix Sum
0	$-\infty$ 1
1	∞ 17 0 1
2	∞ 0
3	∞ 0
4	∞ 0

Figure 6.28: Processing element 17 yields an LIS of 1. We query the length 0 prefix sum, using binary search to find the last index greater than or equal to 17, and find there is 1 LIS of length 0 ending at a value less than 17. We then update the length 1 list and sum.

LIS Length	Elements Prefix Sum
0	$-\infty$ 1
1	∞ 17 0 1
2	∞ 0
3	∞ 0
4	∞ 0

Figure 6.29: Processing element 17 yields an LIS of 1. We query the length 0 prefix sum, using binary search to find the last index greater than or equal to 17, and find there is 1 LIS of length 0 ending at a value less than 17. We then update the length 1 list and sum.

LIS Length	Elements Prefix Sum
0	$-\infty$ 1
1	∞ 17 8 0 1 2
2	∞ 0
3	∞ 0
4	∞ 0

Figure 6.30: Processing element 8 also yields an LIS of 1. Similar to before, we find 1 LIS of length 0, so we update the length 1 LIS list accordingly. Note the list is decreasing monotonically, and the final prefix sum value (2) indicates there are two LIS of length 1 that end with a value greater than or equal to 8.

LIS Length	Elements Prefix Sum
0	$-\infty$ 1
1	∞ 17 8 0 1 2
2	∞ 22 0 2
3	∞ 0
4	∞ 0

Figure 6.31: Processing element 22 yields an LIS of length 2. Performing a binary search against all LIS of length 1 reveals infinity to be the last value greater than or equal to 22. We subtract the count of LIS greater than or equal to 22 (0) from the total number of LIS with that count (2) to reveal 2 total LIS of length 2 ending at 22.

LIS Length	Elements Prefix Sum
0	$-\infty$ 1
1	∞ 17 8 7 1 0 1 2 3 4
2	∞ 22 0 2
3	∞ 0
4	∞ 0

Figure 6.32: Processing elements 7 and 1 update the length 1 LIS list similar to earlier elements. Note the prefix sum is trivially updateable during the append operation.

LIS Length	Elements Prefix Sum
0	$-\infty$ 1
1	∞ 17 8 7 1 0 1 2 3 4
2	∞ 22 10 0 2 4
3	∞ 0
4	∞ 0

Figure 6.33: Processing elements 10 yields and LIS of 2. Performing the binary search against length 1 LIS, we find 17 to be the last usable previous element. We subtract the count ending at 17 or greater (1) from the count of all LIS of length 1 (4) to reveal there are 3 LIS of length 2 ending at 10. We update the prefix sum accordingly.

LIS Length	Elements Prefix Sum
0	$-\infty$ 1
1	∞ 17 8 7 1 0 1 2 3 4
2	∞ 22 10 3 0 2 5 6
3	∞ 0
4	∞ 0

Figure 6.34: Processing element 3 updates similar to element 10.

LIS Length	Elements Prefix Sum
0	$-\infty$ 1
1	∞ 17 8 7 1 0 1 2 3 4
2	∞ 22 10 3 0 2 5 6
3	∞ 25 0 6
4	∞ 0

Figure 6.35: Processing element 25 yields an LIS of length 3, which can follow all 6 LIS of length 2.

LIS Length	Elements Prefix Sum
0	$-\infty$ 1
1	∞ 17 8 7 1 0 1 2 3 4
2	∞ 22 10 3 0 2 5 6
3	∞ 25 14 0 6 10
4	∞ 0

Figure 6.36: Processing element 14 yields an LIS of length 3. Binary searching we find 22 as the last element which cannot be used of the length 2 list. $6 - 2 = 4$ LIS are added to the prefix sum for length 3.

LIS Length	Elements Prefix Sum
0	$-\infty$ 1
1	∞ 17 8 7 1 0 1 2 3 4
2	∞ 22 10 3 0 2 5 6
3	∞ 25 14 0 6 10
4	∞ 19 0 4

Figure 6.37: Finally processing the 19 we find a length 4 LIS, which can only append the $10 - 6 = 4$ LIS of length 3 which terminate at 14.

At the end of the execution, the LIS is the last list which has a non-zero prefix sum, and the value of the prefix sum is the total number of LIS of that length, and thus the ultimate count. As we perform only one extra binary search per iteration, and that binary search is over a list $O(n)$ in size, the runtime is not impacted.

```

//assume array d

int dp[n+1]; //lowest value to achieve a given length
for(int i=0; i<n+1; i++) dp[i]=INT32_MAX;
dp[0]=0;
vector<pll> all[n+1]; //the final value of sequences which terminate at a
                    //given length, and the amount of sequences that terminate at that
                    //final value or lesser
for(int i=0; i<n+1; i++) all[i].push_back({INT32_MAX, 0});
all[0].push_back({INT32_MIN, 1}); //have a valid sequence of length 0,
//ending with minimum value.
for(int i=0; i<n; i++){
    //perform "standard" binary search to find the LIS at this index
    int delta=1;
    while(delta<n) delta<<=1;
    int j=0;
    while(delta){
        if(j+delta<n&&dp[j+delta]<d[i]) j+=delta;
        delta>>=1;
    }
    dp[j+1]=min(dp[j+1], d[i]);

    //perform binary search over LIS of length one fewer. find last
    //element
    //which isn't usable
    delta=1;

```

```

while(delta<all[j].size())delta<<=1;
int k=0;
while(delta){
    if(k+delta<all[j].size()&&all[j][k+delta].first>=d[i])k+=delta;
    delta>>=1;
}
//k now points to the last value larger than or equal to us. This is
//the one we DON'T want. we want all the rest, so subtract from the
//total prefix sum, and append to the right list
all[j+1].push_back({d[i],(all[j].back().second-all[j][k].second+all[j+1].back().second+MOD)%MOD});
}

```

6.4.6.3 Further Segtree Techniques

With the above tools, there are several further common operations which can be accomplished beyond counting or recovering.

Counting All Increasing Subsequences Computing all increasing subsequences can be solved with the a slightly different DP. Suppose we know the count of all increasing subsequences ending at every index up to $n - 1$ and want to solve for n . For every previous index $k < n$ which has a lower value, $a[k] < a[n]$, we can extend all subsequences that end at k with n . We also have a single new subsequence which starts (and ends) at n . This is a range sum which can be obtained in $O(\log(n))$ time with a segment or fenwick tree indexed by the value in the input array. In short, instead of having the data structure count the number of occurrences of the maximum, we simply count all occurrences.

Longest Non-decreasing Subsequence The longest non-decreasing subsequence can be computed identically to the computation of LIS by adjusting appropriate comparators from $>$ to \geq , or similar as appropriate. The correctness thereof can be seen by adding a small delta to identical values: 1, 2, 2, 3, 2, 3, 4 for instance could become 1, 2, $2 + \epsilon$, $3, 2 + 2 * \epsilon, 3 + \epsilon, 4$.⁴⁷ As there are now no identical values, the LIS is equal to the longest non-decreasing subsequence. Adjusting the comparators as noted above simply removes the need to add the deltas.

K-th Least Lexicographic LIS The K-th least lexicographic LIS asks us if all LIS were enumerated and sorted, which would be k-th? As with any non-enumerative indexing problem, our high level solution will involve building the output one element at the time. This is done by considering all LIS which use a given element, and identifying the least element such that the count which traverse that element and all lesser exceed the value k, at which point we know the next element must be that one.

⁴⁷assume epsilon is small enough to not reach the next highest value in the input sequence

We will solve this problem first with an input which does not have duplicate elements, and then extend it for any input sequence.

K-th Least LIS without Duplicates Suppose we can know every element which starts an increasing subsequence of length equal to the LIS. If we also knew the count of such sequence at each element, we could sort those elements and use standard non-enumerative indexing techniques to identify the first value. This process could be then be applied to all elements starting an increasing subsequence of length one fewer than the LIS whose index occurs after that of the identified first element, and whose value is greater than that of the first element, enabling identification of the second element. The process is repeated until all elements of the output are identified.

This leads to a few subproblems:

- Finding the total count of LIS which start at a given index is the mirror of finding the count of LIS which end at a given index. We can therefore find this value by reverseing the input sequence and computing the longest decreasing subsequences and counts and then reversing again. The count of longest decreasing subsequences ending at a given index in the reversed array is the same as the count of LIS starting at a given index in original array.
- In order to perform the indexing, we need to process elements in order of decreasing LIS, followed by increasing lexicographic order. We can sort all indices thusly, and jump ahead when needed, such as when:
 1. We have found the correct element for a given length, and must skip all other elements of that length.
 2. The value of a subsequent element is less than the value of the previously found element.⁴⁸

With this, we can write the code.

```
//assume we've reversed the input, computed LIS and count, then
un-reversed the results.

//Sort the indices to get the right evaluation order
int indices[n];
for(int i=0;i<n;i++)indices[i]=i;
sort(indices,indices+n,[&](int a,int b){
    if(lis[a]!=lis[b])return lis[a]>lis[b]; //highest LIS go first
```

⁴⁸While we could explicitly check if the index of the next element ($e+1$), is greater than the index of the previously identified element (e), this is not strictly necessary. There are three cases. Either the value of $e+1$ is less than the value of e , in which case it is skipped due to the value rule. If it is between the value of e and the correct value of $e+1$, then it must have an LIS one greater than the correct value of $e+1$, which is causes it to be skipped due to the length rule. Lastly, if it has value greater than the correct value of $e+1$, then the correct element in the sequence will have already been found.

```

        if(d[a]!=d[b])return d[a]<d[b]; //take thiese LIS in lexicographic
            order
        return a<b; //otherwise order by position in the sequence
    });

vector<int> out;
int to_find=lis[indices[0]]; //indicates which LIS size we're currently
    looking for
for(int i=0;i<n&&to_find;i++){
    if(lis[indices[i]]!=to_find)break; //tried all the elements with
        proper LIS...didn't get K of them, bail.
    if(!out.empty()&&d[indices[i]]<=out.back())continue; //skip if value
        didn't increase
    if(count[indices[i]]<k)k-=count[indices[i]]; //kth LIS does not start
        here. just subtract off the number that start here
    else{ //kth LIS does start here! cache value and advance to correct
        place
        out.push_back(d[indices[i]]);
        to_find--;
        while(i<n&&lis[indices[i]]!=to_find)i++;
        i--;
    }
}

//if(to_find), then no solution
//otherwise answer is in "out"

```

There is one further issue to resolve. As the count of LIS rises exponentially with the length of the input, we have the issue to resolve of how to compute the counts. In a standard problem where we are only counting, we would likely compute under some modulus, however that would break the indexing later on. To solve this, notice we only ever care about the exact value of a count if it is less than k .⁴⁹ Therefore, we do not care about counts that are greater than k , and in the process of computing the counts, if one exceeds k , we can cap that value at k .⁵⁰

This choice limits the solution techniques we can use to compute the counts. The binary search method depends on using prefix sums which would break if we could not store the exact count. Therefore, we are restricted to the segtree based approach.

K-th Least LIS with Duplicates While it may seem intriguing, running the above solution if there are duplicates fails. Consider the basic test case 1, 4, 1, 3, 5. When searching for the second LIS, the earlier code will identify

⁴⁹If the value is less than k , we have to subtract the exact value. If it is greater than k , we simply move on to the next index.

⁵⁰depending on implementation, this may require 64 or even 128 bit types to ensure overflow is handled before taking the `max` with k .

1, 4, 5 while the correct answer is 1, 3, 5.⁵¹ The core reason is while the code correctly identifies the first 1 initiates two LIS, and the second 1 initiates 2, it cannot infer the ordering between them. In this case, there is an LIS starting at the second 1 which orders before some of the LIS starting at the first 1.

Fundamentally, we have to consider the all the similar values at a given LIS in aggregate rather than individually. This is trivial for the first element⁵² but becomes problematic for each subsequent value. There are two high level challenges:

1. When computing the number of LIS which start at a given node, there may be multiple times a given LIS is used given the selection of previous elements. Consider the input 1, 1, 3, 2, 5. The output of the `count` array for the element 2 is 1,⁵³ However, there are two LIS which use this pattern due to the preceding multiple 1s.
2. Not every element $e + 1$ is necessarily prepended by every element e , depending on where they are in the input. Consider an input 1, 3, 1, 3, 5. The second 3 can succeed either 1, but the first 3 can only succeed the first 1. Due to the aggregation, we do not know at the time of the selection of the first element which 1 it is.⁵⁴

To resolve this, consider that non-enumerative indexing demands we count the number of times a given value is used,⁵⁵ In the 1, 1, 3, 2, 5 example, given that we might have selected 1 as our first element,⁵⁶ we must compute not how many LIS start at 2, but instead how many LIS use that value of 2 *given the first value is 1*. To accomplish this, we must query each preceding occurrence of a 1 and multiply that by the number of unique LIS which start at 2 to get the correct number.

In order to perform that query, we need to know exactly how many of the given previous value occur before each candidate next value in the sequence, and how many LIS traverse those previous values. This can be done with a segtree. Suppose we select 12 as some element. For each incidence of 12 which has the correct LIS starting at it, we can insert into a segtree at the index where that 12 occurs the count of LIS that arrive at that node using the exact elements we've already selected. Let's see an example.

Consider the example sequence 1, 3, 1, 3, 5. The LIS values are 3, 2, 3, 2,⁵⁷ and the counts are 2, 1, 1, 1, 1.⁵⁸ Suppose we are looking for the second least LIS. When sorting the indices by LIS, then value, then index, we get the ordering

⁵¹There are two occurrences of 1, 3, 5.

⁵²we can just sum up all similar values

⁵³The maximum LIS starting at this node is 2, (2,5), and there is only 1 subsequence of length 2 starting here.

⁵⁴pun intended

⁵⁵given that the previously determined value

⁵⁶when searching for $k = 1$ or $k = 2$

⁵⁷the LIS starting at index 0 is length 3, the LIS starting at index 1 is 2, etc.

⁵⁸there are 2 LIS that begin at index 0. There is 1 LIS that begins at index 1, etc

0, 2, 1, 3, 4.⁵⁹

We examine all elements with LIS 2 and find via the counts directly that 3 distinct LIS begin at some index with value 1. Therefore we know the second LIS must start with a 1. We insert the following into our segtree:

- At index 0, we have 1 LIS which arrives at 1, so we insert 1 at 0.
- At index 2, we have 1 LIS which starts with 1, so insert 1 at 2.

Note that we are not concerned with the number of LIS which start or traverse here, only the LIS which *end* here. This causes the value to differ from the count.

Now, we attempt to find the second element. We find our first element with the proper LIS (1) is 3, so we must aggregate the count of all LIS which have a 3 following the previously identified 1s. To do this we use the segtree.

- The 3 at index 1 means we query the segtree on the interval $[0, 1)$. This returns a value of 1. As the number of LIS which start at this location is also 1, we have $1 * 1 = 1$ LIS which starts with a 1 previously, and then progresses through this 3.
- The 3 at index 3 means we query the segtree on the interval $[0, 3)$. This returns a value of 2. As the number of LIS which start at this location is 1, we have $1 * 2 = 2$ LIS which start at a 1 previously, and then progress through this 3.

After summing the above results, we find there are 3 LIS which append a 3 to the previously found 1. Therefore we know the second value is 3. As with the 1, we need to cache this information in a segtree. For now, assume we use a new segtree.⁶⁰

- At index 1, we have 1 LIS which arrives at the 3, so we insert 1 at 1.
- At index 3, we have 2 LIS which arrive at the 3, so insert 2 at 3.

We then progress to the last element, and for the only element with the proper LIS,⁶¹ we query the segtree from $[0, 4)$. We find $3 * 1 = 3$ LIS which traverse this element, so can conclude our last value is 5.

From a high level, we should remember these 3 facts:

1. The count array stores the number of LIS which start at a given index and go to the end.
2. The segtree stores the number of LIS which end at a given index, using the exact elements we have determined to be in the k-th LIS.

⁵⁹the two ones come first, with an LIS of 2, following by the two 3s with an LIS of 2, then the 5

⁶⁰In theory, we could remove the entries used from before without incurring a complexity cost, however we will see shortly why this is unnecessary

⁶¹the 5

3. The product of these two values is the total number of LIS which traverse a given element, given the the LIS prefix contains the exact elements we have already determined.

One more note before we offer an implementation. Above it is suggested to use a new segtree for each element of the LIS. This is not strictly necessary. The type of segree we used when developing the counts originally counted the number of times the maximum value occurred in the range. When actually constructing the LIS, each subsequent identified element necessarily increases.⁶² Therefore, when storing the counts at given indices, we can instead store the value at that index as well as the counts. For example, when we store the second 3, we would indicate that that at index 3, there is a value of 3, and it occurs 2 times. By operating this way, each new entry effectively overrides any previous ones. The existence of the 3 overrides the count for any 1s which might exist in the queried range. We therefore must only be careful that the intended value occurs somewhere in any range we query (which it is).

We now show code to implement this indexing. TODO can startlis be eliminated, just query from 0? yeh? TODO define the term "proper prefix" or something

```

//assume counts and lis. also assume list of indices sorted by
//-lis in decreasing order (MSB)
//-values in increasing order
//-index (LSB)
//as seen in the case with no repeats

segtree<pll> st2(n+2); //max-counting segtree
vector<int> out;
int to_find=lis[indices[0]]; //which LIS values we're currently looking
    for
int start_lis=0; //the earliest identical value at a given LIS size
for(int i=0;i<n&&to_find;i++){
    if(lis[indices[i]]!=to_find)break;//we exhausted all the lengths and
        didn't hit k. bail.
    if(!out.empty()&&d[indices[i]]<=out.back())continue;//can't use this
        one, since it's value is lower than previously found element

    //figure out how many LIS use this char as the next
    ll final_same; //This will just track the last index with this value
    __int128 cnt=0;
    vector<ll>incoming; //number of LIS coming into this index.
        ultimately a cache of the segtree lookup

    for(final_same=i;final_same<n&&lis[indices[final_same]]==lis[indices[i]]&&d[indices[final_same]]==d
        incoming.push_back(out.empty()?1:st2.rq(start_lis,indices[final_same]).second);//number
            of ways this sequence can be prepended
        cnt += (__int128)incoming.back()*count[indices[final_same]]; //The

```

⁶²by definition of LIS

```

        magic multiplication to account for all LIS which traverse
        this node using the proper prefix
    cnt=min(cnt,(__int128)MOD);
}
if(cnt<k){
    k-=cnt;//kth LIS does not start here. just subtract off the number
        that start here
    i=final_same-1;//set to next number after all the ones we examined
        here
}
else{//kth LIS does start here. add value and advance
    out.push_back(d[indices[i]]);
    start_lis=indices[i];//earliest instance of this value. helpful
        for the range query
    to_find--;
    for(int
        l=i;l<final_same;l++)st2.pu(indices[l],{d[indices[l]],incoming[l-i]});
        //update the segtree indicating the number of proper LIS
        coming into this index
    while(i<n&&lis[indices[i]]!=to_find)i++; //set to next number with
        next LIS value
    i--;
}
}
}

```

Increasing Subsequence Cover The increasing subsequence cover problem asks us to find the minimum number of disjoint increasing subsequences which cover all values in an input. It may seem counterintuitive, but this turns out to be exactly the length of the longest non-increasing subsequence.⁶³

We offer the following rough proof:

- It should be clear that the size of the increasing subsequence cover must be at least as large as the length of the longest non-increasing subsequence. No two items in any non-increasing subsequence can be in the same increasing subsequence.⁶⁴
- We can show the size of the increasing subsequence cover can be as small as the longest non-increasing subsequence by construction. Consider the binary-search based algorithm for counting LIS. An output of this technique is a list per LIS length of values which could have terminated an LIS of that length. Each of those lists is non-increasing. Using the properties we articulated earlier,⁶⁵ we can surmise that if we attempted to compute the length of the longest non-increasing subsequence, we could find those

⁶³Elements of the subsequence may decrease or stay the same. This is just the reverse of the longest non-decreasing subsequence, discussed earlier

⁶⁴by definition of increasing vs non-increasing

⁶⁵namely that we can trivially move between computing increasing, non-increasing, decreasing, and non-decreasing by only toggling a few comparators

lists stringly increasing. As each list is strictly increasing, and each element is stored in some list, and the number of total lists equals the length of the longest non-increasing subsequence, we have successfully covered all elements with some increasing subsequence.

As we know the size of the cover must be at least as large as, but no greater than the length of the longest non-increasing subsequence, we have sufficiently proven the supposition.⁶⁶

LIS on Trees The LIS on a tree is the LIS on any path from a leaf to the root. To solve this problem using standard DP-on-a-tree techniques, we would solve both subtrees, and then combine the solutions. Unfortunately, as each subtree carries at least $O(n)$ elements of information,⁶⁷ combining the solution at each interior node would take at least $O(n)$ time, leading to quadratic overall performance.

Instead, we will use a different technique where we DFS from the root toward the leaves. Consider how a typical DFS works. We evaluate some subtree and then "pop" the node off the stack before trying the next subtree. This pop represents the undoing of some data structure to leave that structure in the same state it was before processing the subtree. There is no fundamental reason why that undoing must be limited to a single stack.

Consider a tree which is just a single branch from a root to a leaf. DFSing along this tree would be tantamount to iterating through an array, in which case we could compute the LIS while doing so. Now suppose there is a branch somewhere. In order to compute the LIS on the second branch, we would need all data structures in the same state they were before the first branch was computed. To achieve this, when returning from the recursion in the DFS, we also undo whatever changes we made to the LIS data structures. This may mean:

- reverting a change to the segtree, by replacing the proper index with the previous value
- reverting changes to the array used for binary search, by remembering which index was updated and what its previous value was

As we only have to undo exactly as many operations as we performed at any given node, and all updates are reversible in time no worse than the time to make them, this undoing does not incur a complexity cost.

⁶⁶This duality is known as *Dilworth's Theorem*.

⁶⁷the LIS DP array or segtree is at least $O(n)$ big