

Longest increasing subsequence

Table of Contents

- Solution in $O(n^2)$ with dynamic programming
 - Finding the length
 - Implementation
 - Restoring the subsequence
 - Implementation of restoring
 - Alternative way of restoring the subsequence
- Solution in $O(n \log n)$ with dynamic programming and binary search
 - Implementation
 - Restoring the subsequence
- Solution in $O(n \log n)$ with data structures
- Related tasks
 - Longest non-decreasing subsequence
 - Number of longest increasing subsequences
 - Smallest number of non-increasing subsequences covering a sequence
- Practice Problems

We are given an array with n numbers: $a[0 \dots n - 1]$. The task is to find the longest, strictly increasing, subsequence in a .

Formally we look for the longest sequence of indices i_1, \dots, i_k such that

$$i_1 < i_2 < \dots < i_k,$$

$$a[i_1] < a[i_2] < \dots < a[i_k]$$

In this article we discuss multiple algorithms for solving this task. Also we will discuss some other problems, that can be reduced to this problem.

Solution in $O(n^2)$ with dynamic programming

Dynamic programming is a very general technique that allows to solve a huge class of problems. Here we apply the technique for our specific task.

First we will search only for the **length** of the longest increasing subsequence, and only later learn how to restore the subsequence itself.

Finding the length

To accomplish this task, we define an array $d[0 \dots n - 1]$, where $d[i]$ is the length of the longest increasing subsequence that ends in the element at index i . We will compute this array gradually: first $d[0]$, then $d[1]$, and so on. After this array is computed, the answer to the problem will be the maximum value in the array $d[]$.

So let the current index be i . I.e. we want to compute the value $d[i]$ and all previous values $d[0], \dots, d[i - 1]$ are already known. Then there are two options:

- $d[i] = 1$: the required subsequence consists of only the element $a[i]$.
- $d[i] > 1$: then in the required subsequence is another number before the number $a[i]$. Let's focus on that number: it can be any element $a[j]$ with $j = 0 \dots i - 1$ and $a[j] < a[i]$. In this fashion we can compute $d[i]$ using the following formula: If we fixate the index j , then the longest increasing subsequence ending in the two elements $a[j]$ and $a[i]$ has the length $d[j] + 1$. All of these values $d[j]$ are already known, so we can directly compute $d[i]$ with:

$$d[i] = \max_{\substack{j=0 \dots i-1 \\ a[j] < a[i]}} (d[j] + 1)$$

If we combine these two cases we get the final answer for $d[i]$:

$$d[i] = \max \left(1, \max_{\substack{j=0 \dots i-1 \\ a[j] < a[i]}} (d[j] + 1) \right)$$

Implementation

Here is an implementation of the algorithm described above, which computes the length of the longest increasing subsequence.

```
int lis(vector<int> const& a) {
    int n = a.size();
    vector<int> d(n, 1);
```

```

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (a[j] < a[i])
                d[i] = max(d[i], d[j] + 1);
        }
    }

    int ans = d[0];
    for (int i = 1; i < n; i++) {
        ans = max(ans, d[i]);
    }
    return ans;
}

```

Restoring the subsequence

So far we only learned how to find the length of the subsequence, but not how to find the subsequence itself.

To be able to restore the subsequence we generate an additional auxiliary array $p[0 \dots n - 1]$ that we will compute alongside the array $d[]$. $p[i]$ will be the index j of the second last element in the longest increasing subsequence ending in i . In other words the index $p[i]$ is the same index j at which the highest value $d[i]$ was obtained. This auxiliary array $p[]$ points in some sense to the ancestors.

Then to derive the subsequence, we just start at the index i with the maximal $d[i]$, and follow the ancestors until we deduced the entire subsequence, i.e. until we reach the element with $d[i] = 1$.

Implementation of restoring

We will change the code from the previous sections a little bit. We will compute the array $p[]$ alongside $d[]$, and afterwards compute the subsequence.


For convenience we originally assign the ancestors with $p[i] = -1$. For elements with $d[i] = 1$, the ancestors value will remain -1 , which will be slightly more convenient for restoring the subsequence.

```
vector<int> lis(vector<int> const& a) {
    int n = a.size();
    vector<int> d(n, 1), p(n, -1);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (a[j] < a[i] && d[i] < d[j] + 1)
                d[i] = d[j] + 1;
            p[i] = j;
        }
    }

    int ans = d[0], pos = 0;
    for (int i = 1; i < n; i++) {
        if (d[i] > ans) {
            ans = d[i];
            pos = i;
        }
    }

    vector<int> subseq;
```

```
while (pos != -1) {  
    subseq.push_back(a[pos]);  
    pos = p[pos];  
}  
reverse(subseq.begin(), subseq.end());  
return subseq;  
}
```



Alternative way of restoring the subsequence

It is also possible to restore the subsequence without the auxiliary array $p[]$. We can simply recalculate the current value of $d[i]$ and also see how the maximum was reached.

This method leads to a slightly longer code, but in return we save some memory.

Solution in $O(n \log n)$ with dynamic programming and binary search

In order to obtain a faster solution for the problem, we construct a different dynamic programming solution that runs in $O(n^2)$, and then later improve it to $O(n \log n)$.

We will use the dynamic programming array $d[0 \dots n]$. This time $d[i]$ will be the element at which a subsequence of length i terminates. If there are multiple

such sequences, then we take the one that ends in the smallest element.

Initially we assume $d[0] = -\infty$ and for all other elements $d[i] = \infty$.

We will again gradually process the numbers, first $a[0]$, then $a[1]$, etc, and in each step maintain the array $d[]$ so that it is up to date.

After processing all the elements of $a[]$ the length of the desired subsequence is the largest l with $d[l] < \infty$.

```
int lis(vector<int> const& a) {
    int n = a.size();
    const int INF = 1e9;
    vector<int> d(n+1, INF);
    d[0] = -INF;

    for (int i = 0; i < n; i++) {
        for (int j = 1; j <= n; j++) {
            if (d[j-1] < a[i] && a[i] < d[j])
                d[j] = a[i];
        }
    }

    int ans = 0;
    for (int i = 0; i <= n; i++) {
        if (d[i] < INF)
            ans = i;
    }
    return ans;
}
```

We now make two important observations.

The array d will always be sorted: $d[i - 1] \leq d[i]$ for all $i = 1 \dots n$. And also the element $a[i]$ will only update at most one value $d[j]$.

Thus we can find this element in the array $d[]$ using binary search in $O(\log n)$. In fact we are simply looking in the array $d[]$ for the first number that is strictly greater than $a[i]$, and we try to update this element in the same way as the above implementation.

Implementation

```
int lis(vector<int> const& a) {
    int n = a.size();
    const int INF = 1e9;
    vector<int> d(n+1, INF);
    d[0] = -INF;

    for (int i = 0; i < n; i++) {
        int j = upper_bound(d.begin(), d.end())
            if (d[j-1] < a[i] && a[i] < d[j])
                d[j] = a[i];
    }

    int ans = 0;
    for (int i = 0; i <= n; i++) {
        if (d[i] < INF)
            ans = i;
    }
}
```



```
return ans;
```

```
}
```

Restoring the subsequence

It is also possible to restore the subsequence using this approach. This time we have to maintain two auxiliary arrays. One that tells us the index of the elements in $d[]$. And again we have to create an array of "ancestors" $p[i]$. $p[i]$ will be the index of the previous element for the optimal subsequence ending in element i .

It's easy to maintain these two arrays in the course of iteration over the array $a[]$ alongside the computations of $d[]$. And at the end it is not difficult to restore the desired subsequence using these arrays.

Solution in $O(n \log n)$ with data structures

Instead of the above method for computing the longest increasing subsequence in $O(n \log n)$ we can also solve the problem in a different way: using some simple data structures.

Let's go back to the first method. Remember that $d[i]$ is the value $d[j] + 1$ with $j < i$ and $a[j] < a[i]$.

Thus if we define an additional array $t[]$ such that

$$t[a[i]] = d[i],$$

then the problem of computing the value $d[i]$ is equivalent to finding the **maximum value in a prefix** of the array $t[]$:

$$d[i] = \max(t[0 \dots a[i] - 1] + 1)$$

The problem of finding the maximum of a prefix of an array (which changes) is a standard problem that can be solved by many different data structures. For instance we can use a [Segment tree](#) or a [Fenwick tree](#).

This method has obviously some **shortcomings**: in terms of length and complexity of the implementation this approach will be worse than the method using binary search. In addition if the input numbers $a[i]$ are especially large, then we would have to use some tricks, like compressing the numbers (i.e. renumber them from 0 to $n - 1$), or use an implicit Segment tree (only generate the branches of the tree that are important). Otherwise the memory consumption will be too high.

On the other hand this method has also some **advantages**: with this method you don't have to think about any tricky properties in the dynamic programming solution. And this approach allows us to generalize the problem very easily (see below).

Related tasks

Here are several problems that are closely related to the problem of finding the longest increasing subsequence.

Longest non-decreasing subsequence

This is in fact nearly the same problem. Only now it is allowed to use identical numbers in the subsequence.

The solution is essentially also nearly the same. We just have to change the inequality signs, and make a slightly modification to the binary search.

Number of longest increasing subsequences

We can use the first discussed method, either the $O(n^2)$ version or the version using data structures. We only have to additionally store in how many ways we can obtain longest increasing subsequences ending in the values $d[i]$.

The number of ways to form a longest increasing subsequences ending in $a[i]$ is the sum of all ways for all longest increasing subsequences ending in j where $d[j]$ is maximal. There can be multiple such j , so we need to sum all of them.

Using a Segment tree this approach can also be implemented in $O(n \log n)$.

It is not possible to use the binary search approach for this task.

Smallest number of non-increasing subsequences covering a sequence

For a given array with n numbers $a[0 \dots n - 1]$ we have to colorize the numbers in the smallest number of colors, so that each color forms a non-increasing subsequence.

To solve this, we notice that the minimum number of required colors is equal to the length of the longest increasing subsequence.

Proof: We need to prove the **duality** of these two problems.

Let's denote by x the length of the longest increasing subsequence and by y the least number of non-increasing subsequences that form a cover. We need to prove that $x = y$.

It is clear that $y < x$ is not possible, because if we have x strictly increasing elements, than no two can be part of the same non-increasing subsequence. Therefore we have $y \geq x$.

We now show that $y > x$ is not possible by contradiction. Suppose that $y > x$. Then we consider any optimal set of y non-increasing subsequences. We transform this in set in the following way: as long as there are two such subsequences such that the first begins before the second subsequence, and the first sequence start with a number greater than or equal to the second, then we unhook this starting number and attach it to the beginning of second. After a finite number of steps we have y subsequences, and their starting

numbers will form an increasing subsequence of length y . Since we assumed that $y > x$ we reached a contradiction.

Thus it follows that $y = x$.

Restoring the sequences: The desired partition of the sequence into subsequences can be done greedily. I.e. go from left to right and assign the current number or that subsequence ending with the minimal number which is greater than or equal to the current one.

Practice Problems

- Topcoder - IntegerSequence
- Topcoder - AutoMarket
- Codeforces - Tourist
- Codeforces - LCIS
- SPOJ - SUPPER
- Topcoder - BridgeArrangement
- ACMMSGURU - "North-East"