Longest Increasing Subsequence Problem

In this article we'll study about the longest increasing subsequence and how to solve it.

Definitions

Def.: Given a sequence A, a *subsequence of A* is a sequence where we remove some elements of A.

Exemples
Given A = (1, 2, -3, 1 , 2, 2, 9)

The following sequences are subsequences of A:

- B = (1, 2, -3);
- C = (1, 1, 2, 9);
- D = (2, -3, 2, 9);
- E = (-3);
- F = (1, 2, -3, 1, 2, 2, 9).

But the following ones are not subsequences of A:

- G = (1, 2, 2, 2, 2);
- H = (7);
- I = (9, 2, 2, 1, -3, 2, 1).

Can you see why?

Def.: A sequence is called *increasing* if, for all elements but the first one, that element is extrictly greater than the last element. (i.e. $\forall i > 1 \; (a_i > a_{i-1})$).

Def.: A sequence is called *nondecreasing* if $\forall i > 1 \; (a_i \geq a_{i-1})$.

Exemples

The C, E, G and H are nondecreasing sequences, but only E and H era increasing sequences.
A sequence like (1, 2, 5, 7) is also a increasing sequence.

The problem

Problem: Given a sequence, find the longest increasing subsequence (i.e. the one with more elements).

Exemple
For sequence A, the LIS is (-3, 1, 2, 9).

If the problems was to find the longest nondecreasing subsequence, it could be or (1, 2, 2, 2, 9) or (-3, 1, 2, 2, 9) because both have five elements.

Solving it

The naive approach to this problem is find all the subsequences and recording the longest one. As we know, the number of subsequences (or subsets) of a sequence with n elements is 2^n. So that approach is at least $O(2^n)$.

We can find two good solutions to the problem though. The first one uses Dynamic Programming and runs in $O(n^2)$ and the second runs in $O(n \log n)$.

Dynamic Programming

Even if this is a bad solution, it's so simple to code that it deserves to be related here.

Thinking in DP, we need to think in subproblems. And there's nothing more obvious than thinking in the i first elements of the subsequence and answering the question for then.

The base case is $i = 1$ (the first element). The answer is just 1, take that element.

For all other i, we look to the previous solutions and see if we can extend then.
We can extend a solution if the i-th element is greater then the last element of the solution we're looking at.

So we declare an array dp with n elements, initializa all its elements as 1 and, for every i starting at the second element, we see if we can extend all other solutions and make $dp[i]$ to be the maximum of then.

For instance, if the sequence is (1, 2, -3, 1, 2, 9), we'll have initially
$dp = (1, 1, 1, 1, 1, 1)$.
Then we look at the second element of the sequence and compare it with the first one. If it's greater, we make $dp[2] = dp[1] + 1$ (the answer of 1 plus 1 (because we're adding this element to the sequence)).

So the implementation could be like this:

```
void LIS(int arr[N]) {
    for (int i = 0; i < N; i++) dp[i] = 1;

    for (int i = 1; i < N; i++)
        for (int j = 0; j < i; j++)
            if (arr[i] > arr[j])
                dp[i] = max(dp[i], dp[j] + 1);
}
```

In dp we have the LIS ending at all i's, so the lenght of the longest one is the maximum number of dp. We could find the maximum on the go.

To more details on that solution, see this video:
Find The Longest Increasing Subsequence - Dynamic Programming Fundamentals by Back To Back SWE.
All the thanks to the authors.

As you can see, that is $O(n^2)$ because the two nested fors.
(For all element, we check all elements before it)

The better solution

The other solution births from the following algorithm:

Supose that we're consume the elements of the sequence from the left to the right and use then following these rules:

1. Put a number on the top of a stack if the number is lower than the top;
2. Create a new stack, in the right of all others and put the number there.

Now we use a greedy algorithm to solve our problem:
If we can put the number we're considering on the first stack, put it there. If we can't, move to the next stack. If we arrive the last stack and still can't put the number there, create a new stack and leave it there.

Simulation

For the sequence (1, 2, -3, 1, 2, 9, -1), we could have this:

Consuming 1:
Stack1 = $\emptyset$

As there's no stack, we just create one and leave 1 there.

Consuming 2:
Stack1 = 1

As 2 is not lower then 1, we're left if no stacks and need to create a new one.

Consuming -3:
Stack1 = 1
Stack2 = 2

As -3 is lower than 1, we can put it in the top of stack1.

Consuming 1:
Stack1 = 1, -3
Stack2 = 2

1 is not lower than -3, but it's lower than 2, so we left it in the second stack.

Consuming 2:
Stack1 = 1, -3
Stack2 = 2, 1

Now we create a new stack.

Consuming 9:
Stack1 = 1, -3
Stack2 = 2, 1
Stack3 = 2

And again a new stack.

Finally consuming -1:
Stack1 = 1, -3
Stack2 = 2, 1
Stack3 = 2
Stack4 = 9

And the final configuration of the stacks is:
Stack1 = 1, -3
Stack2 = 2, 1, -1
Stack3 = 2
Stack4 = 9

Proof on correction

Maybe you see why this algorithm works and how smart it is.

Notice that in each step the tops are in increasing order.

We can prove it's increasing using induction:
The base case is when it was one element and the induction case is supposing we have a increasing sequence and one element to put somewhere, we'll put it in the first spot where it's lower then the element that was in that spot, leaving the sequence increasing. It'll be not lower then the previous element (because if it was, we would replace that element) and it'll be not greater then the next element (because the hypothesis says that the element that was in this position was lower than the next one and i'm lower than that element).

And notice that the number of stacks is the number of the LIS. That's because the numbers of each stack are in descending order (from the base to the top). Therefore, I cannot pick two element of the same stack to form a increase subsequence. (observe that if one element of a stack is upper then other, than that number comes after the bottom one).

It means that if I pick a number in the first stack, I can just form a increase subsequence if I pick a number from other stack (one in the right of that).

Another interesting to comment is that whenever we add a number, the number of the stack we'll put it onto means the maximum increasing subsequence ending at that number.
We can realize that making some observations:

1. When a number x is to be added, all the numbers in the stacks are numbers that come before x in the sequence;
2. When we're traversing the stacks, all the stack we discart contains numbers that are lower then x;
3. When we find a stack to put x (or create a new one), all the tops from x to the stack form a increasing subsequence;
4. When we create the rightest stack, all the tops form the LIS.

Observation 3 is a little more hard to see, but we can prove that by indiction:

The base is one element.
For any other x, if x is added to the top of the i-th stack, we know that the element from the (i-1)-th top is the last element of a increasing subsequence (by the induction hypothesis), so, as x is greater than the (i-1)-th top, now all the elements of the i first tops form a increasing subsequence.

Then, the fourth observation become easy.

Implementation

To have a good algorithm, we need to do some tricks on the algorithm I've showed to you. The first trick is to realize we don't need stacks, just the top elements matters (so we can use an array `tops`). The other trick is to use binary search to search for the position where we'll insert the next element we're looking. The array is sorted the whole time, so we can do that.

```
int LIS(int arr[N]) {
  // Preprocessing
  vector<int> tops(N+1, INF);
  tops[0] = -INF;
  int num_tops = 0;

  // For all element of the sequence
  for (int i = 0; i < n; i++) {
    // gets the position of the first element
    // equal or greater than arr[i]
    int j = lower_bound(tops.begin(), tops.end(), arr[i]) -
                        tops.being();
    // The minus thing is because C++

    if (tops[j] == INF) num_tops++;
    tops[j] = arr[i];
  }

  return num_tops;
}
```

An online judge problem

Here is an online problem (in Portuguese): Letras

And here is the solution:

```cpp
#include <bits/stdc++.h>
typedef long long int ll;

#define INF LLONG_MAX
#define MINF LLONG_MIN

string s;

ll LIS() {
  vector<ll> tops(s.size()+1, INF);
  tops[0] = MINF;
  ll maxi = 0;

  for (ll i = 0; i < s.size(); i++)
  {
    ll j = upper_bound(tops.begin(), tops.end(), s[i]) -
                       tops.begin();
    if (tops[j] == INF) maxi++;
    tops[j] = s[i];
  }

  return maxi;
}

int main(int argc, char const *argv[]) {
  cin >> s;
  cout << LIS() << endl;
  return 0;
}
```