The Longest Non-increasing Subsequence Problem

The longest reversely sorted (non-increasing) subsequence problem is to find a subsequence of a given sequence in which the subsequence's elements are in reversely sorted order, highest to lowest, and in which the subsequence is as long as possible. This subsequence is not necessarily contiguous, or unique. The longest non-increasing subsequence problem can be formulated as follows.

longest sorted (non-increasing) subsequence
input: a vector V of n comparable elements **output:** a vector R containing the longest reversely sorted subsequence of V

There is a complicated algorithm that solves this problem in $O(n \log n)$ time[1], but we will be implementing two simpler algorithms with slower time complexities.

The End-to-Beginning Algorithm

There is a straightforward algorithm that solves the problem and has $O(n^2)$ time complexity. The algorithm uses an additional array H of length n, of non-negative integers. The value $H[i]$ will indicate how many elements, smaller than $A[i]$, are further in the sequence A and have some special property.

Initially, the array H is set to 0 (all the elements are 0). The algorithm proceeds by attempting to increase the values of H starting with the previous to last element and going down to the first element. Then a longest subsequence can be identified by selecting elements of A in decreasing order of the H values, starting with the element in A that has the largest H value.

Algorithm End_to_Beginning

Step 1. Set all the values in the array H to 0.

Step 2. Starting with $H[n-1]$ and going down to $H[0]$ try to increase the value of $H[i]$ as follows:

Step 3. Starting with index $i+1$ and going up to $n-1$ (the last index in the array A) repeat Steps 4 and 5:

Step 4. See if any element is smaller or equal to $A[i]$ and has its H value also bigger to $H[i]$.

Step 5. If yes, then $A[i]$ can be followed by that element in a sorted subsequence, thus set $H[i]$ to be 1 plus the H value of that element.

---

[1] Schensted, C. (1961), "Longest increasing and decreasing subsequences", Canadian Journal of Mathematics 13: 179–191, doi:10.4153/CJM-1961-015-3

[2] http://www.programminglogic.com/powerset-algorithm-in-c/

Step 6. Calculate the largest (maximum) value in array *H*. By adding 1 to that value we have the length of a longest reversely sorted subsequence.

Step 7. Identify a longest subsequence by identifying elements in array *A* that have decreasing *H* values, starting with the largest (maximum) value in array *H*.

An Exhaustive Algorithm

There is an exhaustive algorithm that solves the problem and has $O(n \cdot 2^n)$ time complexity. The algorithm generates all possible subsequences of the array *A* and tests each subsequence on whether it is in reversely sorted order. The longest such subsequence is a solution to the problem.

We note that a subsequence can be uniquely identified by the set of indices in the array *A* that are part of the subsequence. For example, given the array

A = [143, 233, 55, 89, 21, 34, 8, 13, 3, 5, 1, 2, 253, 0, 1] the subsequence *R* = [143,89,21] is uniquely identified by the set of indices {0,3,4} of the elements in the array A.

To generate all possible subsequences, one may consider generating the power set of {0,1,...n-1} and consider each subset of the power set as the set of indices in A of the subsequence.

There are several ways to generate the power set. One way is to implement an iterative algorithm that uses a stack to grow and shrink the set as needed. The benefit of this approach is that it prints the subsets in lexicographic order. Below find an implementation[2] in C++ that generates the power set of the set {1, 2, …, n} (where you can specify n in the program):

```
void Powerset (int n)
// function to generate the power set of {1, .., n} and retrieve the best set
int *stack, k;   stack = new int[n+1]; // allocate space for the set
stack[0]=0; /* 0 is not considered as part of the set */   k = 0;   while(1) {
if (stack[k] < n) {
          stack[k+1] = stack[k] + 1;
          k++;
   }
else {
          stack[k-1]++;
          k--;
   }     if (k==0)
break;
  }   delete [ ] stack; // deallocate space for the
set   return; }
```

Implementation

You are provided with the following files.

1. `subsequence.hpp` is a C++ header that defines functions for the two algorithms described above. Some function definitions are incomplete skeletons; you will need to rewrite them to actually work properly.
2. `subsequence_timing.cpp` is a C++ program with a `main()` function that measures one experimental data point for each of the algorithms. You can expand upon this code to obtain several data points for each of your algorithm implementations.
3. `timer.hpp` contains a small `Timer` class that implements a precise timer using the `std::chrono` library from C++11. It is used by `subsequence_timing.cpp`.
4. `Makefile`, `subsequence_test.cpp`, `rubrictest.hpp`, and `README.md` work the same way as in Project 1.