Dynamic Programming: Longest Increasing Subsequence

Class 31

The Problem

- consider a list of values

$$11, 14, 13, 7, 8, 15$$

- a subsequence is any list of the original value that maintains their original respective order, e.g.,

$$11, 7, 8$$

- an increasing subsequence is any subsequence such that the values strictly increase left-to-right, e.g.,

$$14, 15$$

- a longest increasing subsequence (LIS) is an increasing subsequence of maximum length, e.g.,

$$7, 8, 15$$

- note that an LIS is not necessarily unique, e.g.,

$$11, 14, 15$$

Note

- while superficially this seems like it might be similar to longest common subsequence, it is totally different
- in LCS, there are two sequences
- in LIS, there is only one sequence

- in LCS, characters match or not
- in LIS, values increase in order

Step 1

- given a sequence $x_0, x_1, \ldots, x_{n-1}$
- we wish to find a subsequence $x_i, x_k, \ldots, x_m$ such that
- $0 \leq i \leq m \leq n-1$ and
- $x_i < x_k < \cdots < x_m$ and the number of x_is is maximal

Step 2

- if we are at an arbitrary index i in the sequence, then a LIS from 0 to i either includes i or does not
- let $\text{opt}(i)$ denote the length of a LIS from 0 to i
- we need a recurrence relation for $\text{opt}(i)$
- there are two possibilities:
  1. x_i is an element of the LIS
  2. x_i is not an element of the LIS
- it appears that our recurrence is simply

$$\text{opt}(i) = \begin{cases} \text{opt}(i-1) + 1 & \text{if } x[i-1] < x[i] \\ \text{opt}(i-1) & \text{if } x[i-1] \not< x[i] \end{cases}$$

Step 2

$$\text{opt}(i) = \begin{cases} \text{opt}(i-1)+1 & \text{if } x[i-1] < x[i] \\ \text{opt}(i-1) & \text{if } x[i-1] \not< x[i] \end{cases}$$

- but there is a problem with this
- the LIS may skip over the $i-1$ element
- or even skip over many elements

$$\ldots, 3, 4, 5, 9, 9, 9, 6, 7, \ldots$$

Step 2

- we need to consider all possible values from here back to 0
- this gives us

$$\mathrm{opt}(i) = \max_{k=0}^{i-1} \left(\underset{x_k < x_i}{\mathrm{opt}} \, (k) \right) + 1$$

- our previous DP examples have looked at two or three specific locations; this one looks at many locations
- the many locations requires a loop
- all previous DP examples have had a 2-d memo table; this one has a 1-d memo table

```
size_t opt(size_t i, vector<size_t>& memo,
           const vector<unsigned>& values)
{
  if (memo.at(i) == SIZE_MAX)
  {
    if (i == 0)
    {
      memo.at(i) = 1;
    }
    else
    {
      size_t max_so_far = 0;
      for (size_t k = i - 1; k < SIZE_MAX; k--)
      {
        size_t max_k = opt(k, memo, values);
        if (values.at(k) < values.at(i) && max_k > max_so_far)
        {
          max_so_far = max_k;
        }
      }
      memo.at(i) = max_so_far + 1;
    }
  }
  return memo.at(i);
}
```

A Memo Table

- for this input list, what is the memo table?

3	2	5	4	6	9	6	input
0	1	2	3	4	5	6	

							memo
0	1	2	3	4	5	6	

A Memo Table

- for this input list, what is the memo table?

	3	2	5	4	6	9	6
input	0	1	2	3	4	5	6

	1	1	2	2	3	4	3
memo	0	1	2	3	4	5	6

A Memo Table

- for this input list, what is the memo table?

	3	2	5	4	6	9	6
input index	0	1	2	3	4	5	6

memo	1	1	2	2	3	4	3
index	0	1	2	3	4	5	6

- where is the "final answer"?
- in all previous DP problems, the answer was at bottom right
- here, it is not at the end
- the final answer is the largest element in the memo table
- that is where the traceback begins

Step 4

- the traceback begins at the largest element in the memo table
- where does it go from there?

0	1	2	3	4	5	6	
1	1	2	2	3	4	3	memo

- clearly the 4 value had to come from the 3 value (at index 4)
- but where did the 3 come from?
- from one of the 2's, but which one?

- in essence, traceback has to re-compute the recurrence

Traceback

- in a case such as this, we can be much more efficient if we keep track of decisions made while filling in the memo table
- we keep a helper table that records the decisions
- we will call our helper table prev

```
size_t opt(size_t i, vector<size_t>& memo, const vector<unsigned>& values,
           vector<size_t>& (prev)
  if (memo.at(i) == SIZE_MAX)
  {
    if (i == 0)
    {
      memo.at(i) = 1;
    }
    else
    {
      size_t max_so_far = 0;
      for (size_t k = i - 1; k < SIZE_MAX; k--)
      {
        size_t max_k = opt(k, memo, values, prev);
        if (values.at(k) < values.at(i) && max_k > max_so_far)
        {
          max_so_far = max_k;
          prev.at(i) = k;
        }
      }
      memo.at(i) = max_so_far + 1;
    }
  }
  return memo.at(i);
}
```

Traceback

- traceback starts with the largest element in the memo
- then proceeds with guidance from prev

3	2	5	4	6	9	6	input
0	1	2	3	4	5	6	

1	1	2	2	3	4	3	memo
0	1	2	3	4	5	6	

–	–	1	1	3	4	3	prev

```
#include <algorithm>

// find the largest entry
vector<size_t>::iterator largest_element = max_element(memo.begin(),
                                                       memo.end());
cout << "lis: " << *largest_element << endl;
size_t index = static_cast<size_t>(distance(memo.begin(), largest_element));

string lis;
while (index < memo.size())
{
  lis = to_string(values.at(index)) + " " + lis;
  index = prev.at(index);
}
cout << lis << endl;
```