Dynamic Programming: Longest Increasing Subsequence

Class 31

The Problem

consider a list of values

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

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

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

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

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

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

- 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 opt(i) denote the length of a LIS from 0 to i
- we need a recurrence relation for 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

$$\mathsf{opt}(i) = \begin{cases} \mathsf{opt}(i-1) + 1 & \mathsf{if } x[i-1] < x[i] \\ \mathsf{opt}(i-1) & \mathsf{if } x[i-1] < x[i] \end{cases}$$

$$\operatorname{opt}(i) = \begin{cases} \operatorname{opt}(i-1) + 1 & \text{if } x[i-1] < x[i] \\ \operatorname{opt}(i-1) & \text{if } x[i-1] \nleq x[i] \end{cases}$$

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

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

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

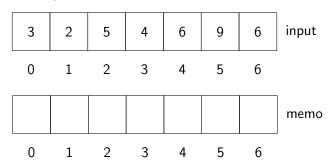
$$\operatorname{opt}(i) = \max_{k=0}^{i-1} \left(\operatorname{opt}_{x_k < x_i}(k) \right) + 1$$

- 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)
2
3
      if (memo.at(i) == SIZE_MAX)
5
         if (i == 0)
6
7
          memo.at(i) = 1;
8
9
        else
10
11
           size_t max_so_far = 0;
12
           for (size_t k = i - 1; k < SIZE_MAX; k--)</pre>
13
14
             size_t max_k = opt(k, memo, values);
15
             if (values.at(k) < values.at(i) && max_k > max_so_far)
16
17
               max_so_far = max_k;
18
19
20
          memo.at(i) = max_so_far + 1;
21
22
23
      return memo.at(i);
24
25
```

A Memo Table

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



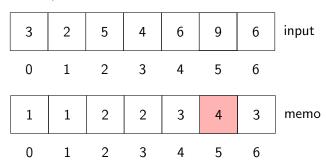
A Memo Table

• 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	•

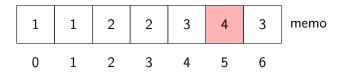
A Memo Table

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



- 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

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



- 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

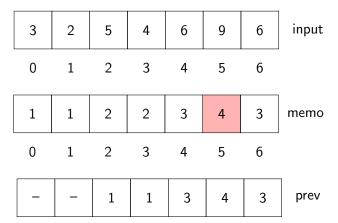
Traceback

- 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,
1
                vector<size_t>& (prev)
      if (memo.at(i) == SIZE MAX)
3
4
        if (i == 0)
5
6
          memo.at(i) = 1;
8
        else
9
        {
10
          size t max so far = 0:
11
          for (size_t k = i - 1; k < SIZE_MAX; k--)
12
13
           {
             size_t max_k = opt(k, memo, values, prev);
14
15
             if (values.at(k) < values.at(i) && max_k > max_so_far)
16
17
               max so far = max k:
               prev.at(i) = k;
18
19
           }
20
          memo.at(i) = max_so_far + 1;
21
22
      }
23
      return memo.at(i):
24
25
```

Traceback

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



lis = to_string(values.at(index)) + " " + lis;

1

9

10

11

12

13 14

15

#include <algorithm>

while (index < memo.size())</pre>

index = prev.at(index);

cout << lis << endl;</pre>

string lis;

{