CS 341 - Algorithms, Winter 2016. University of Waterloo

Lecture 12: Dynamic programming II

We will use dynamic programming to design efficient algorithms for some basic sequence and string problems.

Longest increasing subsequence [DPV 6.2]

Given n numbers $a_1, a_2, \ldots, a_n$, a subsequence is any subset of these numbers taken in order, of the form $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ where $1 \le i_1 < i_2 < \ldots < i_k \le n$, and a subsequence is increasing if $a_{i_1} < a_{i_2} < \ldots < a_{i_k}$.

Input: n numbers, $a_1, \ldots, a_n$.

Output: a longest increasing subsequence.

For example, given 5,1,9,8,8,8,4,5,6,7 (recognize this?), the longest increasing subsequence is 1,4,5,6,7.

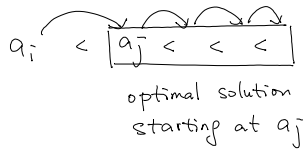
By now, it is not difficult to solve this problem using dynamic programming.

Let $L(i)$ be the length of a longest increasing subsequence starting at a_i and using the numbers in $a_i, \ldots, a_n$.

Final answer: After we compute $L(1), L(2), \ldots, L(n)$, the final answer is $\max_{1 \le i \le n} \{L(i)\}$.

Recurrence relation: If we start at a_i, then we look at the numbers in $a_{i+1}, \ldots, a_n$ that are greater than a_i, and use the subsequences starting them with a_i to form a subsequence starting at a_i:

More precisely, $L(i) = 1 + \max_{i+1 \le j \le n} \{ L(j) \mid a_j > a_i \}$.



Note that the subsequences formed must be increasing.

The correctness can be proved by induction, i.e. assuming $L(i+1), \ldots, L(n)$ are computed correctly, then $L(i)$ is also computed correctly.

Bottom up implementation

As usual, we can compute the values in backward order.

```
L(i) = 1    ∀ 1 ≤ i ≤ n    // initialization
for i from n downto 1 do
    for j from i+1 to n+1 do
        if aj > ai and L(j) + 1 > L(i)
            update L(i) ← L(j) + 1.
```

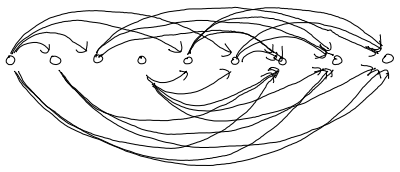
For example, given 3, 8, 7, 2, 6, 4, 12, 14, 9,
the L-values are 3, 2, 2, 3, 2, 2, 1, 1, 1.

Time complexity : It is clear that the time complexity is $O(n^2)$.

Printing a longest increasing subsequence : We leave it as an exercise to you.
One way to do it is to keep track of the next number (say in next[i]) when we update $L(i) \leftarrow L(j)+1$.
We can also directly trace back a longest increasing subsequence without using extra storage.

Longest path in a DAG : An alternative way is to think of this problem as finding a longest path in a DAG.

Given n numbers $a_1, \ldots, a_n$, we create a graph of n vertices, each corresponding to a number.
There is a directed edge from i to j if $j > i$ and $a_j > a_i$.
Then, an increasing subsequence corresponds to a directed path in this graph, and a longest increasing subsequence corresponds to a longest path in the graph

For example, given 3, 8, 7, 2, 6, 4, 12, 14, 9,
the graph is  .

The longest path problem in DAG can be solved by dynamic programming in the same way, and it is what the programming problem in HW3 asking.

An O(nlogn) algorithm (harder)

There is a clever algorithm that solves the problem in O(nlogn) time.

The observation is that: when we compute $L(i)$, we may not need to use all $L(i+1), \ldots, L(n)$. We just need to keep the "best" solution of each length.

For example, Suppose $L(j) = L(k)$ for $j < k$. then we know that $a_j > a_k$ (as otherwise $L(j)$ should be at least $L(k)+1$). When we compute $L(i)$ for $i < j < k$, we can just keep the value of $L(j)$ and forget about $L(k)$, since any increasing subsequence that can be extended using an optimal solution starting at a_k can also be extended using an optimal solution starting at a_j, because $a_j > a_k$.

Say, given the sequence 2, 3, 4, 3, 6, 5, 8, 7, when we compute $L(2)$, the L-values so far are 3, 3, 2, 2, 1, 1. Instead of storing all these six numbers, we can as well just remember that $best[3] = 3$, $best[2] = 5$, and $best[1] = 7$, where $best[3]$ stores the best position to start an increasing subsequence of length 3.

More succint information

Suppose we would like to compute $L(i)$

Instead of directly computing it using $L(i+1), \ldots, L(n)$, we inductively maintain the following.

Let the current length of the longest increasing subsequence be ℓ, i.e. $\ell = \max_{i+1 \le j \le n} \{L(j)\}$.

For $1 \le j \le \ell$, let $S_j = \{ p \mid L(p) = j \}$ be the set of indexes with $L(p) = j$.

Let $best[j] = p$ where $p \in S_j$ and $a_p \ge a_{p'}$ for all $p' \in S_j$, so that $best[j]$ stores the best position to start an increasing subsequence of length j because the starting value is largest, so that it is easiest to be extended.

Sortedness

A key observation is that $a[best[\ell]] < a[best[\ell-1]] < \ldots < a[best[1]]$.

Suppose, for contradiction, that it is not true.

Then, there exists j such that $a[best[j]] \ge a[best[j-1]]$.

Let the optimal increasing subsequence of length j be $a_{p_1} < a_{p_2} < \ldots < a_{p_j}$ where $p_1 = best[j]$.

Then $a_{p_2} < \ldots < a_{p_j}$ is an increasing subsequence of length $j-1$, with $a_{p_2} > a_{p_1} = a_{best[j]} \ge a_{best[j-1]}$, contradicting the maximality of $best[j-1]$.

So, we have the important sortedness property.

Updating best[j] using binary search

When we consider a_i, if we see that $a_i < a[best[\ell]]$, then that's good, because we can extend the longest increasing subsequence so far by one, by adding a_i in the beginning of the increasing subsequence of length ℓ starting at $best[\ell]$. So, we can increase ℓ by 1, and set $best[\ell] = i$.

If $a[best[j]] < a_i < a[best[j-1]]$, then we can not use it to form an increasing subsequence of length $j+1$, but we can use it to form an increasing subsequence of length j, by adding a_i in the beginning of the increasing subsequence of length $j-1$ starting at $best[j-1]$. Furthermore, this increasing subsequence of length j is better than the one started at $best[j]$ because $a_i > a[best[j]]$. So, in this case, we update $best[j] = i$.

Finally, if $a[best[1]] < a_i$, then we can not use it to extend any increasing subsequence because it is larger than all starting points, but we can use it to update $best[1] = i$.

Now, since $a[best[\ell]] < a[best[\ell-1]] < \dots < a[best[1]]$, we can use binary search to find the smallest j so that $a[best[j]] \le a_i$, then we update according to the above rules.

Algorithm

$\ell = 1$. $best[1] = n$. // initialization

for i from $n-1$ downto 1 do

  if $a_i < a[best[\ell]]$, then set $j \leftarrow \ell+1$ and set $\ell \leftarrow \ell+1$. // longer increasing subsequence

  else use binary search to find the smallest index j so that $a[best[j]] \le a_i$.

  set $best[j] \leftarrow i$.

return $best[\ell]$.

The final algorithm is very simple, but may not be easy to come up with.

Time complexity: Since we can do binary search, it takes $O(\log n)$ time for each

Time complexity: Since we can do binary search, it takes $O(\log n)$ time for each iteration, and the total complexity is $O(n \log n)$.

Tracing a solution: We leave it as an exercise to print a longest increasing subsequence.

Longest Common Subsequence [CLRS 15.3]

Input: Two strings $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$ where each a_i, b_j is a symbol.

Output: the largest k such that there exist $i_1 < i_2 < \ldots < i_k$ and $j_1 < j_2 < \ldots < j_k$ so that $a_{i_\ell} = b_{j_\ell}$ for $1 \le \ell \le k$.

One example is that we are given two DNA sequences and want to identify common structures.

$S_1 = AAACCGTGAGTTATTCGTTCTAGAA$
$S_2 = CACCCCTAAGGTACCTTTGGTTC$ $\Rightarrow$ $ACCTAGTACTTTG$

The longest increasing subsequence (LIS) problem is a special case of the longest common subsequence (LCS) problem

3, 8, 7, 2, 6, 4, 12, 14, 9 (for LIS) $\Rightarrow$ (reduce)

3, 8, 7, 2, 6, 4, 12, 14, 9
2, 3, 4, 6, 7, 8, 9, 12, 14 (for LCS)

By having the second sequence as sorted, we force the solution to LCS to be an increasing subsequence, and hence an optimal solution to LCS is an optimal solution to LIS, and the reduction can be carried out efficiently in $O(n \log n)$ time using sorting.

Recurrence

Let $C(i,j)$ be the length of a longest common subsequence of $a_i, \ldots, a_n$ and $b_j, \ldots, b_m$.

The answer that we are looking for is $C(1,1)$.

The boundary/base cases are $C(n+1, j) = 0 \quad \forall\, 1 \le j \le m$, and $C(i, m+1) = 0 \quad \forall\, 1 \le i \le n$.

To compute $C(i,j)$, there are three cases, depending on whether a_i and b_j are used or not.

① (Use both a_i and b_j) If $a_i = b_j$, then we can put a_i and b_j in the beginning of a common subsequence, then the remaining problem becomes finding a longest common subsequence for $a_{i+1}, \ldots, a_n$ and $b_{j+1}, \ldots, b_m$.

So, let $SOL_1 = 1 + C(i+1, j+1)$.

② (not use a_i) Then we find a longest common subsequence for $a_{i+1}, \ldots, a_n$ and $b_j, \ldots, b_m$.

So, let $SOL_2 = C(i+1, j)$.

③ (not use b_j) Then we find a longest common subsequence for $a_i, \ldots, a_n$ and $b_{j+1}, \ldots, b_m$.

So, let $SOL_3 = C(i, j+1)$.

Then, we take the best out of these three possibilities.

More succintly, $C(i,j) = \max\{ 1 + C(i+1,j+1),\ C(i+1,j),\ C(i,j+1) \}$.

↑ only include this case when $a_i = b_j$.

Correctness

All solutions for $C(i,j)$ fall into at least one of the above three cases.

We can prove correctness by induction. Assuming that the three subproblems can be solved optimally (in the base cases it is easy to verify), then we compute $C(i,j)$ correctly.

Time complexity There are $n \cdot m$ subproblems. Each subproblem looks up three values.

Using top-down memorization, the total time complexity is $O(m \cdot n)$.

Tracing out solution We can either record some "parent" information when we set $C(i,j)$, by remembering which subproblem gives the max for $C(i,j)$.

We can also compute it directly using $C(i,j)$ only, by recursively going to a subproblem that gives the max for $C(i,j)$. We leave the details as an exercise.

Bottom-up implementation

$C(i, m+1) = 0 \quad \forall\ 1 \le i \le n$, $C(n+1, j) = 0 \quad \forall\ 1 \le j \le m$. // base cases

```
for i from n downto 1 do
    for j from m downto 1 do
        if a_i = b_j, set SOL ← 1 + C(i+1, j+1); else SOL ← 0.
        C(i,j) = max{ SOL, C(i+1, j), C(i, j+1) }.
```

Remark: If we are only interested in the length (e.g. just to measure the similarity), then we can just use $O(n)$ or $O(m)$ space by storing only two rows or two columns as we did in subset-sum. We leave the details as an exercise.

But if we need to trace out the solution, then it is not known how to use less than $\Omega(mn)$ space.

Edit distance [DPV 6.3]

Input : Two strings $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$ where each a_i, b_j is a symbol.

Output : Minimum k so that we can do k add / delete / change operations to transform $a_1, \ldots, a_n$ into $b_1, \ldots, b_m$.

For example, if the two input strings are SNOWY and SUNNY, the following are two ways:

S	–	N	O	W	Y	
S	U	N	N	–	Y	

Cost: 3

–	S	N	O	W	–	Y
S	U	N	–	–	N	Y

Cost: 5

In the first way, we match S, add U, match N, change O to N, delete W, and match Y

This takes three add/delete/change operations to transform SNOWY to SUNNY.

The second way requires five add/delete/change operations to transform SNOWY to SUNNY.

The first way is an optimal solution to the above example. Do you see why?

We call the minimum number of operations to transform one string to another string the "edit distance" between the two strings.

It is a useful measure of the similarity of two strings, e.g. in a word processor.

Recurrence

The recurrence is similar to that in the longest common subsequence problem.

We just need to be careful in the boundary cases.

Let $D(i,j)$ be the edit distance of the strings $a_i, \ldots, a_n$ and $b_j, \ldots, b_m$.

The answer that we want is $D(1,1)$.

The base case is $D(n+1, m+1) = 0$.

To compute $D(i,j)$, there are four possible operations to perform:

(add) We add b_j to the current string, when $j \le m$. e.g. $\begin{matrix}\ldots|abc\\ \ldots|def\end{matrix} \Rightarrow \begin{matrix}\ldots -|abc\\ \ldots d|ef\end{matrix}$

Then we match one more symbol of the target string and move on to the remaining.

More precisely, if $j \le m$, $SOL_1 = 1 + D(i, j+1)$; else $SOL_1 = \infty$.

(delete) We delete a_i from the current string, when $i \le n$. e.g. $\begin{matrix}\ldots|abc\\ \ldots|def\end{matrix} \Rightarrow \begin{matrix}\ldots a|bc\\ \ldots -|def\end{matrix}$

Then we move one symbol forward in the current string.

More precisely, if $i \le n$, $SOL_2 = 1 + D(i+1, j)$; else $SOL_2 = \infty$.

(change) We change a_i to b_j, when $i \le n$ and $j \le m$.

Then we move one symbol forward in both strings. e.g. $\begin{array}{c|c}\cdots & abc \\ \cdots & def\end{array} \Rightarrow \begin{array}{c|c}\cdots a & bc \\ \cdots d & ef\end{array}$

More precisely, if $i \le n$ and $j \le m$, $SOL_3 = 1 + D(i+1, j+1)$; else $SOL_3 = \infty$.

(match) If $i \le n$ and $j \le m$ and $a_i = b_j$, we match this symbol and move forward

More precisely, if $i \le n$ and $j \le m$ and $a_i = b_j$, $SOL_4 = D(i+1, j+1)$; else $SOL_4 = \infty$.

$\begin{array}{c|c}\cdots & abc \\ \cdots & aac\end{array} \Rightarrow \begin{array}{c|c}\cdots a & bc \\ \cdots a & ac\end{array}$

Finally, we set $D(i,j) = \min\{SOL_1, SOL_2, SOL_3, SOL_4\}$.

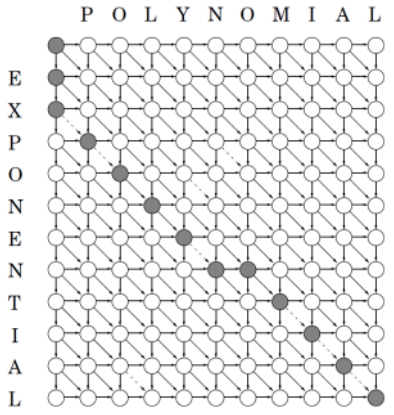
Correctness Follow from the base case and an inductive argument. In the inductive step we have considered all the possibilities to transform one string to another.

We leave it as an exercise to do the formal proof.

Time complexity There are mn subproblems, each requires constant number of operations. Using top-down memorization, the total time complexity is $O(mn)$.

Bottom-up implementation and returning a solution: Given the similarity to the longest common subsequence problem, we leave it as an important exercise for you to complete the details.

Graph searching Once again, we would like to point out that dynamic programming can be thought of as finding a (shortest) path from the starting state to the target state in the state graph. This connection is even more transparent when we are using the state table to trace out a solution.



[DPV 6.3]