CS 341 - Algorithms, Spring 2021. University of Waterloo

Lecture 12: Dynamic Programming II

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

Longest Increasing Subsequence [DPV 6.2]

Given n numbers $a_1, \ldots, a_n$, a subsequence is a subset of these numbers taken in order of the form

$a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ where $1 \le i_1 < i_2 < \cdots < i_k \le n$, and a subsequence is increasing if $a_{i_1} < a_{i_2} < \cdots < a_{i_k}$.

Input: n numbers $a_1, a_2, \ldots, a_n$

Output: an increasing subsequence of maximum length.

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 should be relatively straightforward to solve this problem using dynamic programming.

Subproblems: Let $L(i)$ be the length of a longest increasing subsequence starting at a_i and only using the numbers in $a_i, \ldots, a_n$. So, there are only n subproblems.

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: Given we start at a_i, we try all possible numbers a_j with $j > i$ and $a_j > a_i$, and form an increasing subsequence starting from a_i by concatenating with a longest increasing subsequence starting at a_j.

$a_i < \boxed{a_j < < <}$

optimal solution starting at a_j

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. if $L(i+1), \ldots, L(n)$ are correct, then $L(i)$ is also correct.

Bottom-up implementation

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

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

the L-values are 4, 3, 3, 4, 3, 3, 2, 1, 1

Time complexity: It should be clear that it is bounded by $O(n^2)$.

Printing a longest increasing subsequence We leave it as an exercise to write out the details.

One way to do it is to keep track of the next number (e.g. parent[i] = j) when we update $L(i) \leftarrow L(j)+1$.

We can also directly trace back a longest increasing subsequence using the L-values in $O(n)$ time without using extra storage.

Longest path in DAG: An alternative way to think about this problem is to find a longest path in a DAG.

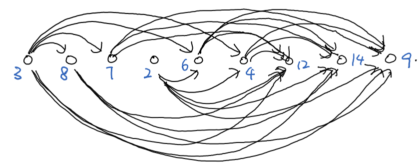
Given n numbers $a_1, \ldots, a_n$, we create a graph of n vertices, each corresponding to a number.

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 directed acyclic graph, and vice versa.

So, a longest path in the graph gives us a longest increasing subsequence.

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



In general, the longest path problem in DAG can be solved by dynamic programming efficiently, and it is a useful exercise to work out the details.

A Faster Algorithm for Longest Increasing Subsequence (Harder, maybe optional).

There is a clever algorithm to solve the problem in $O(n \log n)$ time.

The observation is that we don't need to store all the subproblems, as some subproblems are "dominated" by other subproblems.

For each length k, we will only store the "best" position to start an increasing subsequence of length k.

Then, it will turn out that these best positions satisfy a monotone property, and this allows us to use binary search to update these values in $O(\log n)$ time when we consider a new element.

This is a high level summary and now we discuss the details.

Best subproblems

Suppose we have already computed $L(i+1), L(i+2), \ldots, L(n)$ and now we want to compute $L(i)$.

For a given length k, consider the indices $i < i_1 < i_2 < \cdots < i_\ell$ so that $L(i_1) = L(i_2) = \cdots = L(i_\ell) = k$.

What is the best subproblem to keep for future computations of $L(i), L(i-1), \ldots, L(1)$?

Since we are extending these increasing subsequences using elements in $\{1, \ldots, i\}$, the starting positions $i_1, i_2, \ldots, i_\ell$ are not important.

What is important is the starting value.

If $L(i_1)=L(i_2)=k$ and $a_{i_1}>a_{i_2}$, then the subproblem $L(i_1)$ dominates the subproblem $L(i_2)$, because any increasing subsequence using numbers in $\{a_1,\ldots,a_i\}$ that can be extended by an increasing subsequence of length k starting at a_{i_2} can also be extended by an increasing subsequence of length k starting at a_{i_1}.

That is, among the increasing subsequences of length k, the one with largest starting value is easiest to be extended.

Therefore, we define $pos[k] = \arg\max_{j>i} \{ a_j \mid L(j)=k \}$, when $L(i)$ is the current subproblem to be computed.

Intuitively, $pos[k]$ is the best position to start an increasing subsequence of length k after the current index i.

Let $m = \max_{i<j\leq n} \{L(j)\}$ be the length of a longest increasing subsequence we have computed so far.

By the reasoning above, when we compute $L(i)$, we just need to consider $L(pos[1]), L(pos(2)), \ldots, L(pos(m))$, as the other subproblems are dominated by these subproblems.

For example, given the sequence $2,7,6,1,4,8,5,3$, when we compute $L(2)$, we have $L(3)=L(5)=2$, $L(4)=3$, and $L(6)=L(7)=L(8)=1$, then we only keep $pos[1]=6$ with $a_6=8$, $pos[2]=3$ with $a_3=6$, $pos[3]=4$ with $a_4=1$.

Monotonicity

Once we only keep the best subproblems, we have the following important monotone property.

Claim $a[pos[1]] > a[pos[2]] > \cdots > a[pos[m]]$ where $m=\max_{i<j\leq n}\{L(j)\}$ and $L(i)$ is the current subproblem.

(Intuition: A longer subsequence should be more difficult to be extended, i.e. its starting value is smaller.)

Proof Suppose, by contradiction, that there exists j such that $a[pos[j]] \geq a[pos[j-1]]$.

Let an optimal increasing subsequence of length j be $a_{p_1} < a_{p_2} < \cdots < a_{p_j}$ where $p_1 = pos[j]$.

Then $a_{p_2} < \cdots < a_{p_j}$ is an increasing subsequence of length $j-1$ with $a_{p_2} > a_{p_1} = a[pos[j]] \geq a[pos[j-1]]$, contradicting that $pos[j-1]$ is the best position to start an increasing subsequence of length $j-1$. □

Updating the best subproblems using binary search

Suppose we have found the best subproblems $pos[m], pos[m-1], \ldots, pos[1]$ after processing the numbers $a_n, \ldots, a_{i+1}$.

Now, we process the number a_i, and would like to update the best subproblems for future computations.

We consider three cases.

(1). When $a_i < a[pos[m]]$.

This is the good case, as we can extend the longest increasing subsequence so far by one, by adding a_i in front of the increasing subsequence of length m starting at $pos[m]$.

So, we can increase m by 1, and set $pos[m]=i$.

(2) When $a[pos[j]] \leq a_i < a[pos[j-1]]$.

Since $a_i > a[pos[j]]$, we cannot use a_i to form an increasing subsequence of length $j+1$.

But we can use a_i to form an increasing subsequence of length j, by adding a_i in front of the increasing subsequence of length $j-1$ starting at $pos[j-1]$.

Furthermore, this increasing subsequence of length j is better than the one starting at $pos[j]$, as $a_i > a[pos[j]]$.

So, in this case, we update $pos[j] = i$.

(3) When $a[pos[1]] \leq a_i$.

In this case, we cannot use a_i to extend any increasing subsequence because it is larger than all the starting values, but we can use it to update $pos[1] = i$.

Note that since $a[pos[m]] < a[pos[m-1]] < \cdots < a[pos[1]]$, we can use binary search to find the smallest j so that $a[pos[j]] \leq a_i$, and then we update by the above rules.

Fast Algorithm

$m = 1$, $pos[1] = n$. // base case.

for i from $n-1$ downto 1 do

if $a_i < a[pos[m]]$, then set $m \leftarrow m+1$ and $pos[m] = i$. // longer increasing subsequence

else use binary search to find the smallest j so that $a[pos[j]] \leq a_i$, then set $pos[j] = i$.

return m.

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

Time complexity $O(n \log n)$.

Exercise Write the code 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 $i_1 < i_2 < \cdots < i_k$ and $j_1 < j_2 < \cdots < j_k$ s.t. $a_{i_\ell} = b_{j_\ell}$ for $1 \leq \ell \leq k$.

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

$$\begin{aligned} S_1 &= AAACCGTGAGTTATTCGTTCTAGAA \\ S_2 &= CACCCCTAAGGTACCTTTGGTTC \end{aligned} \quad \Rightarrow \quad ACCTAGTACTTTG$$

Note that the longest subsequence problem (LIS) is a special case of the longest common subsequence problem (LCS).

$$3, 8, 7, 2, 6, 4, 12, 14, 9 \text{ (for LIS)} \xrightarrow{\text{reduce to}} \begin{matrix} 3, 8, 7, 2, 6, 4, 12, 14, 9 \\ 2, 3, 4, 6, 7, 8, 9, 12, 14 \end{matrix} \text{ (for LCS)}$$

Since the second sequence is sorted, it forces the solution of LCS to be an increasing subsequence.

Recurrence

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

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

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

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 subproblem is to find a longest common subsequence for $a_{i+1}, \dots, a_n$ and $b_{j+1}, \dots, b_m$.

So, let $SoL_1 = 1 + C(i+1, j+1)$ if $a_i = b_j$. otherwise $SoL_1 = 0$

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

So, let $SoL_2 = C(i+1, j)$.

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

So, let $SoL_3 = C(i, j+1)$.

Then, we take the best out of these three possibilities. That is, $C(i,j) = \max\{SoL_1, SoL_2, SoL_3\}$.

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

We can then prove correctness by induction.

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 computing $C(i,j)$.

We can also compute it directly using $C(i,j)$ only, by recursively going to a subproblem that gives the maximum value for $C(i,j)$.

Bottom-up implementation

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

for i from n downto 1 do

for j from m downto 1 do

if $a_i = b_j$, set $SoL \leftarrow 1 + C(i+1, j+1)$, else $SoL \leftarrow 0$.

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

Edit Distance [DPV 6.3]

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

<u>Output</u>: The minimum k so that we can do k add / delete / change operations to transform $a_1, \dots, a_n$ into $b_1, \dots, b_m$.

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.

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.

<u>Recurrence</u>

The recurrence is similar to that in LCS.

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

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. $\begin{array}{l|l} \cdots & abc \\ \cdots & def \end{array} \Rightarrow \begin{array}{l|l} \cdots - & abc \\ \cdots d & ef \end{array}$

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

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. $\begin{array}{l|l} \cdots & abc \\ \cdots & def \end{array} \Rightarrow \begin{array}{l|l} \cdots a & bc \\ \cdots - & def \end{array}$

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. $\begin{array}{l|l} \cdots & abc \\ \cdots & def \end{array} \Rightarrow \begin{array}{l|l} \cdots a & bc \\ \cdots d & ef \end{array}$

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 $a_i = b_j$, then we match and move one symbol forward in both strings.

More precisely, if $i \le n$ and $j \le m$ and $a_i = b_j$, $SoL_4 = D(i+1, j+1)$; else $SoL_4 = \infty$.

Finally, we set $D(i,j) = \min\{SoL_1, SoL_2, SoL_3, SoL_4\}$.

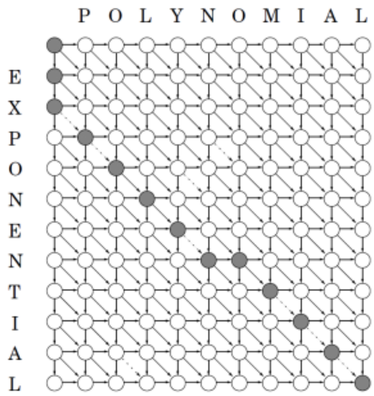
$\begin{array}{l|l} \cdots & abc \\ \cdots & aac \end{array} \Rightarrow \begin{array}{l|l} \cdots a & bc \\ \cdots a & ac \end{array}$

Correctness. Follows from the base case and an inductive argument. In the inductive step as discussed above, we have considered all the possibilities to transform one string to another string.

Time complexity There are $m \cdot n$ subproblems, each requiring a constant number of operations. Using top-down memorization, the time complexity is $O(n \cdot m)$.

Bottom-up implementation and returning a solution Similar to that in LCS. Leave as an important exercise.

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]