Dynamic Programming: Improving Space and/or Time

Lecture 6

Feb 1, 2018

Most slides are courtesy Prof. Chekuri

What is Dynamic Programming?

Every recursion can be memoized. Automatic memoization does not help us understand whether the resulting algorithm is efficient or not.

Dynamic Programming:

A recursion that when memoized leads to an *efficient* algorithm.

Key Questions:

- Given a recursive algorithm, how do we analyze the complexity when it is memoized?
- How do we recognize whether a problem admits a dynamic programming based efficient algorithm?
- How do we further optimize time and space of a dynamic programming based algorithm?

Part I

Edit Distance

Edit Distance

Definition

Edit distance between two words X and Y is the number of letter insertions, letter deletions and letter substitutions required to obtain Y from X.

Example

The edit distance between FOOD and MONEY is at most **4**:

$$\underline{\mathbf{F}}\mathbf{OOD} \rightarrow \mathbf{MO}\underline{\mathbf{O}}\mathbf{D} \rightarrow \mathbf{MON}\underline{\mathbf{O}}\mathbf{D} \rightarrow \mathbf{MONE}\underline{\mathbf{D}} \rightarrow \mathbf{MONEY}$$

Edit Distance: Alternate View

Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

$$\begin{array}{ccccc} \mathbf{F} & \mathbf{O} & \mathbf{O} & & \mathbf{D} \\ \mathbf{M} & \mathbf{O} & \mathbf{N} & \mathbf{E} & \mathbf{Y} \end{array}$$

Formally, an alignment is a sequence M of pairs (i, j) such that each index appears exactly once, and there is no "crossing": if $(i, j), \ldots, (i', j')$ then $i < i'$ and $j < j'$. One of i or j could be empty, in which case no comparision.

Edit Distance: Alternate View

Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

$$\begin{array}{ccccc} \mathbf{F} & \mathbf{O} & \mathbf{O} & & \mathbf{D} \\ \mathbf{M} & \mathbf{O} & \mathbf{N} & \mathbf{E} & \mathbf{Y} \end{array}$$

Formally, an alignment is a sequence M of pairs (i, j) such that each index appears exactly once, and there is no "crossing": if $(i, j), \ldots, (i', j')$ then $i < i'$ and $j < j'$. One of i or j could be empty, in which case no comparision. In the above example, this is $M = \{(1, 1), (2, 2), (3, 3), (\ , 4), (4, 5)\}$.

Cost of an alignment: the number of mismatched columns.

Edit Distance Problem

Problem

Given two words, find the edit distance between them, i.e., an alignment of smallest cost.

Let $X = \alpha x$ and $Y = \beta y$
α, β: strings. x and y single characters.
Possible alignments between X and Y

α	x
β	y

or

α	x
βy	

or

αx	
β	y

Observation

Prefixes must have optimal alignment!

$$EDIST(X, Y) = \min \begin{cases} EDIST(\alpha, \beta) + [x \neq y] \\ 1 + EDIST(\alpha, Y) \\ 1 + EDIST(X, \beta) \end{cases}$$

Subproblems and Recurrence

Each subproblem corresponds to a prefix of X and a prefix of Y

Optimal Costs

Let $\mathrm{Opt}(i,j)$ be optimal cost of aligning $x_1 \cdots x_i$ and $y_1 \cdots y_j$. Then

$$\mathrm{Opt}(i,j) = \min \begin{cases} [x_i \neq y_j] + \mathrm{Opt}(i-1, j-1), \\ 1 + \mathrm{Opt}(i-1, j), \\ 1 + \mathrm{Opt}(i, j-1) \end{cases}$$

Base Cases: $\mathrm{Opt}(i,0) = i$ and $\mathrm{Opt}(0,j) = j$

$X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$, we wish to compute $\mathrm{Opt}(m,n)$.

Matrix and DAG of Computation

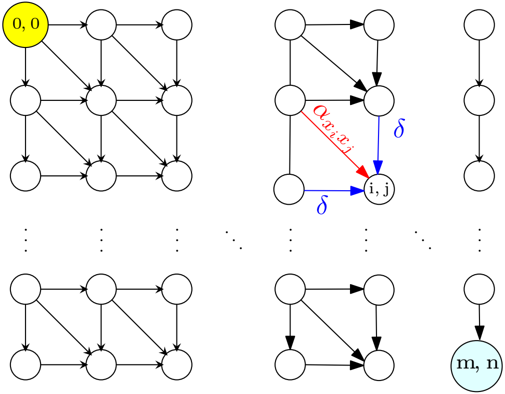

Figure: Iterative algorithm in previous slide computes values in row order.

Computing in column order to save space

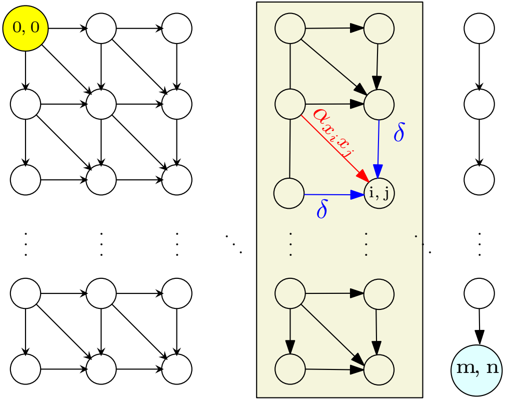

Figure: $\boldsymbol{M(i,j)}$ only depends on previous column values. Keep only two columns and compute in column order.

Optimizing Space

1. Recall

$$M(i,j) = \min \begin{cases} [x_i \neq y_j] + M(i-1, j-1), \\ 1 + M(i-1, j), \\ 1 + M(i, j-1) \end{cases}$$

2. Entries in jth column only depend on $(j-1)$st column and earlier entries in jth column
3. Only store the current column and the previous column reusing space; $N(i,0)$ stores $M(i,j-1)$ and $N(i,1)$ stores $M(i,j)$

Space Efficient Algorithm

```
for all i do N[i,0] = i
for j = 1 to n do
    N[0,1] = j (* corresponds to M(0,j) *)
    for i = 1 to m do
```

$$N[i,1] = \min \begin{cases} [x_i \neq y_j] + N[i-1,0] \\ 1 + N[i-1,1] \\ 1 + N[i,0] \end{cases}$$

```
    for i = 1 to m do
        Copy N[i,0] = N[i,1]
```

Analysis

Running time is $O(mn)$ and space used is $O(2m) = O(m)$

Finding an Optimum Solution

The DP algorithm finds the minimum edit distance in $O(nm)$ space and time.

Can find minimum edit distance in $O(m+n)$ space and $O(mn)$ time.

Previous Exercise: Find an optimum alignment in $O(mn)$ space and time.

Finding an Optimum Solution

The DP algorithm finds the minimum edit distance in $O(nm)$ space and time.

Can find minimum edit distance in $O(m+n)$ space and $O(mn)$ time.

Previous Exercise: Find an optimum alignment in $O(mn)$ space and time.

Today: Finding an optimum alignment and cost in $O(m+n)$ space and $O(mn)$ time.

Divide and Conquer Approach

Fix an optimum alignment between $X[1..m]$ and $Y[1..n]$

Divide and Conquer Approach

Fix an optimum alignment between $X[1..m]$ and $Y[1..n]$

In this optimum alignment $X[1..\frac{m}{2}]$ is aligned with $Y[1..h]$ for some h where $1 \leq h \leq n$. (Need not be unique but we can choose smallest such h). Call this $\mathrm{Half}(X, Y)$

Divide and Conquer Approach

Fix an optimum alignment between $X[1..m]$ and $Y[1..n]$

In this optimum alignment $X[1..\frac{m}{2}]$ is aligned with $Y[1..h]$ for some h where $1 \leq h \leq n$. (Need not be unique but we can choose smallest such h). Call this $\mathrm{Half}(X, Y)$

Suppose we can find $h = \mathrm{Half}(X, Y)$ in time $O(mn)$ time and $O(m + n)$ space, that is, in the same time as finding $\mathrm{Opt}(m, n)$ the optimum value of the alignment between X and Y.

Divide and Conquer Algorithm

```
Linear-Space-Alignment(X[1..m], Y[1..n])
    If m = 1 use basic algorithm in O(n) time and O(n) space
    If n = 1 us basic algorithm in O(m) time and O(n) space

    Compute h = Half(X, Y) in O(mn) time and O(m + n) space
    Linear-Space-Alignment(X[1..m/2], Y[1..h])
    Linear-Space-Alignment(X[m/2 + 1..m], Y[h + 1..n])
    Output concatenation of the two alignments
```

Divide and Conquer Algorithm

```
Linear-Space-Alignment(X[1..m], Y[1..n])
    If m = 1 use basic algorithm in O(n) time and O(n) space
    If n = 1 us basic algorithm in O(m) time and O(n) space

    Compute h = Half(X, Y) in O(mn) time and O(m + n) space
    Linear-Space-Alignment(X[1..m/2], Y[1..h])
    Linear-Space-Alignment(X[m/2 + 1..m], Y[h + 1..n])
    Output concatenation of the two alignments
```

Correctness: Clear based on definition of $\mathrm{Half}(X, Y)$.

Divide and Conquer Algorithm

Linear-Space-Alignment$(X[1..m], Y[1..n])$
    If $m = 1$ use basic algorithm in $O(n)$ time and $O(n)$ space
    If $n = 1$ us basic algorithm in $O(m)$ time and $O(n)$ space

    Compute $h = \mathrm{Half}(X, Y)$ in $O(mn)$ time and $O(m+n)$ space
    Linear-Space-Alignment$(X[1..m/2], Y[1..h])$
    Linear-Space-Alignment$(X[m/2+1..m], Y[h+1..n])$
    Output concatenation of the two alignments

Correctness: Clear based on definition of $\mathrm{Half}(X, Y)$.

Recurrences:

Time bound $T(m, n) =$

Divide and Conquer Algorithm

Linear-Space-Alignment$(X[1..m], Y[1..n])$
    If $m = 1$ use basic algorithm in $O(n)$ time and $O(n)$ space
    If $n = 1$ us basic algorithm in $O(m)$ time and $O(n)$ space

    Compute $h = \mathrm{Half}(X, Y)$ in $O(mn)$ time and $O(m+n)$ space
    Linear-Space-Alignment$(X[1..m/2], Y[1..h])$
    Linear-Space-Alignment$(X[m/2+1..m], Y[h+1..n])$
    Output concatenation of the two alignments

Correctness: Clear based on definition of $\mathrm{Half}(X, Y)$.

Recurrences:

Time bound $T(m, n) = T(m/2, h) + T(m/2, n-h) + cmn$

Space bound $S(m, n) =$

Divide and Conquer Algorithm

Linear-Space-Alignment$(X[1..m], Y[1..n])$
  If $m = 1$ use basic algorithm in $O(n)$ time and $O(n)$ space
  If $n = 1$ us basic algorithm in $O(m)$ time and $O(n)$ space

  Compute $h = \mathrm{Half}(X, Y)$ in $O(mn)$ time and $O(m + n)$ space
  Linear-Space-Alignment$(X[1..m/2], Y[1..h])$
  Linear-Space-Alignment$(X[m/2 + 1..m], Y[h + 1..n])$
  Output concatenation of the two alignments

Correctness: Clear based on definition of $\mathrm{Half}(X, Y)$.

Recurrences:

Time bound $T(m, n) = T(m/2, h) + T(m/2, n - h) + cmn$

Space bound $S(m, n) = \max\{S(\frac{m}{2}, h), S(\frac{m}{2}, n - h), c(m + n)\} + O(1)$

Divide and Conquer Algorithm

Linear-Space-Alignment($X[1..m]$, $Y[1..n]$)
    If $m = 1$ use basic algorithm in $O(n)$ time and $O(n)$ space
    If $n = 1$ us basic algorithm in $O(m)$ time and $O(n)$ space

    Compute $h = \mathrm{Half}(X, Y)$ in $O(mn)$ time and $O(m + n)$ space
    Linear-Space-Alignment($X[1..m/2]$, $Y[1..h]$)
    Linear-Space-Alignment($X[m/2 + 1..m]$, $Y[h + 1..n]$)
    Output concatenation of the two alignments

Correctness: Clear based on definition of $\mathrm{Half}(X, Y)$.

Recurrences:

Time bound $T(m, n) = T(m/2, h) + T(m/2, n - h) + cmn$

Space bound $S(m, n) = \max\{S(\frac{m}{2}, h), S(\frac{m}{2}, n - h), c(m + n)\} + O(1)$

Claim: $T(m, n) = O(mn)$ and $S(m, n) = O(m + n)$.

Proof: Time bound

$$T(m,n) \le \begin{cases} cm & \text{if } n \le 1 \\ cn & \text{if } m \le 1 \\ T(m/2, h) + T(m/2, n-h) + cmn & \text{otherwise} \end{cases}$$

Proof: Time bound

$$T(m,n) \leq \begin{cases} cm & \text{if } n \leq 1 \\ cn & \text{if } m \leq 1 \\ T(m/2,h) + T(m/2,n-h) + cmn & \text{otherwise} \end{cases}$$

Claim: $T(m,n) \leq 2cmn$ by induction on $m+n$.

Inductive step:

$$\begin{aligned} T(m,n) &\leq 2chm/2 + 2c(n-h)m/2 + cmn \\ &\leq 2cnm \end{aligned}$$

Proof: Space bound

$$S(m,n) \leq \begin{cases} cm & \text{if } n \leq 1 \\ cn & \text{if } m \leq 1 \\ \max\{S(m/2,h), S(m/2,n-h), c(m+n)\} + O(1) \end{cases}$$

We can reuse space for computing $\mathrm{Half}(X,Y)$. And storing the alignment can be accounted separatly as $O(m+n)$.

Proof: Space bound

$$S(m,n) \leq \begin{cases} cm & \text{if } n \leq 1 \\ cn & \text{if } m \leq 1 \\ \max\{S(m/2,h), S(m/2,n-h), c(m+n)\} + O(1) \end{cases}$$

We can reuse space for computing $\mathrm{Half}(X, Y)$. And storing the alignment can be accounted separatly as $O(m+n)$.

Claim: $S(m,n) \leq c(m+n) + O(\log m)$.

Computing $\mathbf{Half(X, Y)}$

Want to find h such that

$$\begin{aligned} \text{EDIST}(X, Y) &= \text{EDIST}(X[1..m/2], Y[1..h]) \\ &\quad + \text{EDIST}(X[(m/2+1)..m], Y[(h+1)..n]) \end{aligned}$$

Computing $\mathrm{Half}(\mathbf{X}, \mathbf{Y})$

Want to find h such that

$$\begin{aligned} \text{EDIST}(X, Y) &= \text{EDIST}(X[1..m/2], Y[1..h]) \\ &\quad + \text{EDIST}(X[(m/2+1)..m], Y[(h+1)..n]) \end{aligned}$$

Instead comput for all k where $1 \leq k \leq n$,
(1) $\text{EDIST}(X[1..m/2], Y[1..k])$ &
(2) $\text{EDIST}(X[(m/2+1)..m], Y[(k+1)..n]$

Computing $\text{Half}(\mathbf{X}, \mathbf{Y})$

Want to find h such that

$$\begin{aligned} \text{EDIST}(X, Y) &= \text{EDIST}(X[1..m/2], Y[1..h]) \\ &\quad + \text{EDIST}(X[(m/2+1)..m], Y[(h+1)..n]) \end{aligned}$$

Instead comput for all k where $1 \leq k \leq n$,
(1) $\text{EDIST}(X[1..m/2], Y[1..k])$ &
(2) $\text{EDIST}(X[(m/2+1)..m], Y[(k+1)..n]$

And compute h as
$\min_k(\text{EDIST}(X[1..\frac{m}{2}], Y[1..k]) + \text{EDIST}(X[(\frac{m}{2}+1)..m], Y[(k+1)..n])$

Computing $\mathrm{Half}(\mathbf{X}, \mathbf{Y})$

(1) Compute for all $1 \leq k \leq n$, EDIST$(X[1..\frac{m}{2}], Y[1..k])$

Computing $\mathrm{Half}(\mathbf{X}, \mathbf{Y})$

(1) Compute for all $1 \leq k \leq n$, $\text{EDIST}(X[1..\frac{m}{2}], Y[1..k])$

Claim: All values available if we compute $\text{EDIST}(X[1..\frac{m}{2}], Y[1..n])$ which we can do in $O(mn)$ time.

If M is the resulting table, what entries?

Computing $\mathrm{Half}(\mathbf{X}, \mathbf{Y})$

(1) Compute for all $1 \leq k \leq n$, EDIST$(X[1..\frac{m}{2}], Y[1..k])$

Claim: All values available if we compute EDIST$(X[1..\frac{m}{2}], Y[1..n])$ which we can do in $O(mn)$ time.

If M is the resulting table, what entries? $M(\frac{m}{2}, k)$ for all $1 \leq k \leq n$.

Computing $\mathrm{Half}(\mathbf{X}, \mathbf{Y})$

(1) Compute for all $1 \leq k \leq n$, $\mathrm{EDIST}(X[1..\frac{m}{2}], Y[1..k])$

Claim: All values available if we compute $\mathrm{EDIST}(X[1..\frac{m}{2}], Y[1..n])$ which we can do in $O(mn)$ time.

If M is the resulting table, what entries? $M(\frac{m}{2}, k)$ for all $1 \leq k \leq n$.

Can we do it in $O(m + n)$ space?

Yes! Use the space saving trick in computing edit distance and store the last row!

Computing $\mathrm{Half}(\mathbf{X}, \mathbf{Y})$

(2) Compute for all $1 \leq k \leq n$,

$$\text{EDIST}(X[(\tfrac{m}{2} + 1)..m], Y[(k + 1)..n])$$

Computing $\mathbf{Half}(\mathbf{X}, \mathbf{Y})$

(2) Compute for all $1 \leq k \leq n$,

$$\text{EDIST}(X[(\frac{m}{2}+1)..m], Y[(k+1)..n])$$

If we compute $\text{EDIST}(X[(m/2+1)..m], Y[1..n])$ we get the values $\text{EDIST}(X[(m/2+1)..m], Y[1..k])$ for $1 \leq k \leq n$ which is not what we quite want.

Computing $\mathrm{Half}(\mathbf{X}, \mathbf{Y})$

(2) Compute for all $1 \leq k \leq n$,

$$\text{EDIST}(X[(\tfrac{m}{2}+1)..m], Y[(k+1)..n])$$

If we compute $\text{EDIST}(X[(m/2+1)..m], Y[1..n])$ we get the values $\text{EDIST}(X[(m/2+1)..m], Y[1..k])$ for $1 \leq k \leq n$ which is not what we quite want.

Observation: $\text{EDIST}(X, Y) = \text{EDIST}(\text{reverse}(X), \text{reverse}(Y))$.

Computing $\mathrm{Half}(\mathbf{X}, \mathbf{Y})$

(2) Compute for all $1 \le k \le n$,

$$\mathrm{EDIST}(X[(\tfrac{m}{2}+1)..m], Y[(k+1)..n])$$

If we compute $\mathrm{EDIST}(X[(m/2+1)..m], Y[1..n])$ we get the values $\mathrm{EDIST}(X[(m/2+1)..m], Y[1..k])$ for $1 \le k \le n$ which is not what we quite want.

Observation: $\mathrm{EDIST}(X, Y) = \mathrm{EDIST}(\mathrm{reverse}(X), \mathrm{reverse}(Y))$.

Hence compute $\mathrm{EDIST}(A, B)$ where A is reverse of $X[(m/2+1)..m]$ and B is reverse of $Y[1..n]$ and this will give all the desired values.

Part II

Longest Increasing Subsequence

Sequences

Definition

Sequence: an ordered list $a_1, a_2, \ldots, a_n$. **Length** of a sequence is number of elements in the list.

Definition

$a_{i_1}, \ldots, a_{i_k}$ is a **subsequence** of $a_1, \ldots, a_n$ if $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.

Definition

A sequence is **increasing** if $a_1 < a_2 < \ldots < a_n$. It is **non-decreasing** if $a_1 \leq a_2 \leq \ldots \leq a_n$. Similarly **decreasing** and **non-increasing**.

Sequences

Example...

Example

1. Sequence: $\mathbf{6, 3, 5, 2, 7, 8, 1, 9, 1}$
2. Subsequence of above sequence: $\mathbf{5, 2, 1}$
3. Increasing sequence: $\mathbf{3, 5, 9, 17, 54}$
4. Decreasing sequence: $\mathbf{34, 21, 7, 5, 1}$
5. Increasing <u>subsequence</u> of the first sequence: $\mathbf{2, 7, 9}$.

Longest Increasing Subsequence Problem

Input A sequence of numbers $a_1, a_2, \ldots, a_n$

Goal Find an **increasing subsequence** $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ of maximum length

Longest Increasing Subsequence Problem

Input A sequence of numbers $a_1, a_2, \ldots, a_n$

Goal Find an **increasing subsequence** $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ of maximum length

Example

1. Sequence: 6, 3, 5, 2, 7, 8, 1
2. Increasing subsequences: 6, 7, 8 and 3, 5, 7, 8 and 2, 7 etc
3. Longest increasing subsequence: 3, 5, 7, 8

Recursive Algorithm

Definition

LISEnding$(A[1..n])$: length of longest increasing sub-sequence that *ends* in $A[n]$.

Question: can we obtain a recursive expression?

Recursive Algorithm

Definition

$\textbf{LISEnding}(A[1..n])$: length of longest increasing sub-sequence that *ends* in $A[n]$.

Question: can we obtain a recursive expression?

$$\textbf{LISEnding}(A[1..n]) = 1 +$$

Recursive Algorithm

Definition

LISEnding$(A[1..n])$: length of longest increasing sub-sequence that *ends* in $A[n]$.

Question: can we obtain a recursive expression?

$$\textbf{LISEnding}(A[1..n]) = 1 + \max_{i:A[i]<A[n]} \textbf{LISEnding}(A[1..i])$$

Recursive Algorithm

Definition

LISEnding$(A[1..n])$: length of longest increasing sub-sequence that *ends* in $A[n]$.

Question: can we obtain a recursive expression?

$$\textbf{LISEnding}(A[1..n]) = 1 + \max_{i:A[i]<A[n]} \textbf{LISEnding}(A[1..i])$$

$$\textbf{LISEnding}(A[1..n]) = \max_{i:A[i]<A[n]} \left(1 + \textbf{LISEnding}(A[1..i])\right)$$

Example

Sequence: $A[1..8] = 6, 3, 5, 2, 7, 8, 1, 9$

Recursive Algorithm

```
LIS_ending_alg(A[1..n]):
    if (n = 0) return 0
    m = 1
    for i = 1 to n − 1 do
        if (A[i] < A[n]) then
            m = max(m, 1 + LIS_ending_alg(A[1..i]))
    return m
```

```
LIS(A[1..n]):
        return max_{i=1}^{n} LIS_ending_alg(A[1...i])
```

Recursive Algorithm

```
LIS_ending_alg(A[1..n]):
    if (n = 0) return 0
    m = 1
    for i = 1 to n − 1 do
        if (A[i] < A[n]) then
            m = max(m, 1 + LIS_ending_alg(A[1..i]))
    return m
```

```
LIS(A[1..n]):
        return max_{i=1}^{n} LIS_ending_alg(A[1 . . . i])
```

- How many distinct sub-problems will **LIS_ending_alg**$(A[1..n])$ generate?

Recursive Algorithm

```
LIS_ending_alg(A[1..n]):
    if (n = 0) return 0
    m = 1
    for i = 1 to n − 1 do
        if (A[i] < A[n]) then
            m = max(m, 1 + LIS_ending_alg(A[1..i]))
    return m
```

```
LIS(A[1..n]):
        return max_{i=1}^{n} LIS_ending_alg(A[1...i])
```

- How many distinct sub-problems will **LIS_ending_alg($A[1..n]$)** generate? $O(n)$

Recursive Algorithm

```
LIS_ending_alg(A[1..n]):
    if (n = 0) return 0
    m = 1
    for i = 1 to n − 1 do
        if (A[i] < A[n]) then
            m = max(m, 1 + LIS_ending_alg(A[1..i]))
    return m
```

```
LIS(A[1..n]):
    return max_{i=1}^{n} LIS_ending_alg(A[1 . . . i])
```

- How many distinct sub-problems will **LIS_ending_alg($A[1..n]$)** generate? $O(n)$
- What is the running time if we memoize recursion?

Recursive Algorithm

```
LIS_ending_alg(A[1..n]):
    if (n = 0) return 0
    m = 1
    for i = 1 to n − 1 do
        if (A[i] < A[n]) then
            m = max(m, 1 + LIS_ending_alg(A[1..i]))
    return m
```

```
LIS(A[1..n]):
        return max_{i=1}^{n} LIS_ending_alg(A[1 ... i])
```

- How many distinct sub-problems will **LIS_ending_alg**$(A[1..n])$ generate? $O(n)$
- What is the running time if we memoize recursion? $O(n^2)$ since each call takes $O(n)$ time

Recursive Algorithm

```
LIS_ending_alg(A[1..n]):
    if (n = 0) return 0
    m = 1
    for i = 1 to n − 1 do
        if (A[i] < A[n]) then
            m = max(m, 1 + LIS_ending_alg(A[1..i]))
    return m
```

```
LIS(A[1..n]):
        return max_{i=1}^{n} LIS_ending_alg(A[1 . . . i])
```

- How many distinct sub-problems will **LIS_ending_alg**$(A[1..n])$ generate? $O(n)$
- What is the running time if we memoize recursion? $O(n^2)$ since each call takes $O(n)$ time
- How much space for memoization?

Recursive Algorithm

```
LIS_ending_alg(A[1..n]):
    if (n = 0) return 0
    m = 1
    for i = 1 to n − 1 do
        if (A[i] < A[n]) then
            m = max(m, 1 + LIS_ending_alg(A[1..i]))
    return m
```

```
LIS(A[1..n]):
    return max_{i=1}^{n} LIS_ending_alg(A[1 . . . i])
```

- How many distinct sub-problems will **LIS_ending_alg**$(A[1..n])$ generate? $O(n)$
- What is the running time if we memoize recursion? $O(n^2)$ since each call takes $O(n)$ time
- How much space for memoization? $O(n)$

Removing recursion to obtain iterative algorithm

Typically, after finding a dynamic programming recursion, we often convert the recursive algorithm into an *iterative* algorithm via *explicit memoization* and *bottom up* computation.

Why?

Removing recursion to obtain iterative algorithm

Typically, after finding a dynamic programming recursion, we often convert the recursive algorithm into an *iterative* algorithm via *explicit memoization* and *bottom up* computation.

Why? Mainly for further optimization of running time and space.

Removing recursion to obtain iterative algorithm

Typically, after finding a dynamic programming recursion, we often convert the recursive algorithm into an *iterative* algorithm via *explicit memoization* and *bottom up* computation.

Why? Mainly for further optimization of running time and space.

How?

- First, allocate a data structure (usually an array or a multi-dimensional array that can hold values for each of the subproblems)
- Figure out a way to order the computation of the sub-problems starting from the base case.

Iterative Algorithm via Memoization

Compute the values **LIS_ending_alg($A[1..i]$)** iteratively in a bottom up fashion.

```
LIS_ending_alg(A[1..n]):
    Array L[1..n]  (* L[i] = value of LIS_ending_alg(A[1..i]) *)
    for i = 1 to n do
        L[i] = 1
        for j = 1 to i − 1 do
            if (A[j] < A[i]) do
                L[i] = max(L[i], 1 + L[j])
    return L
```

```
LIS(A[1..n]):
        L = LIS_ending_alg(A[1..n])
        return the maximum value in L
```

Iterative Algorithm via Memoization

Simplifying:

```
LIS(A[1..n]):
    Array L[1..n]  (* L[i] stores the value LISEnding(A[1..i]) *)
    m = 0
    for i = 1 to n do
        L[i] = 1
        for j = 1 to i − 1 do
            if (A[j] < A[i]) do
                L[i] = max(L[i], 1 + L[j])
        m = max(m, L[i])
    return m
```

Correctness: Via induction following the recursion

Running time: $O(n^2)$

Space: $\Theta(n)$

Improving run time

Want to improve run time to $O(n \log n)$ from $O(n^2)$. How?

Idea: Use data structures to improve run-time of computing

$$\mathbf{LISEnding}(i) = \max_{j<i : A[j]<A[i]} 1 + \mathbf{LISEnding}(j)$$

Improving run time

Want to improve run time to $O(n \log n)$ from $O(n^2)$. How?

Idea: Use data structures to improve run-time of computing

$$\text{LISEnding}(i) = \max_{j<i : A[j]<A[i]} 1 + \text{LISEnding}(j)$$

- When computing **LISEnding**(i) we want to focus only on indices j such that $A[j] < A[i]$
- We need to store **LISEnding**(j) with each value $A[j]$ stored in the data structure

Augmented Balanced Binary Search Tree

Assume for simplicity that $a_1, a_2, \ldots, a_n$ are distinct numbers.

- We maintain a dynamic balanced binary search tree T which has only $a_1, \ldots, a_{i-1}$ when **LISEnding(i)** is getting considered.

Augmented Balanced Binary Search Tree

Assume for simplicity that $a_1, a_2, \ldots, a_n$ are distinct numbers.

- We maintain a dynamic balanced binary search tree T which has only $a_1, \ldots, a_{i-1}$ when **LISEnding**(i) is getting considered.
- We can search for a_i in T to obtain a set of subtrees such that each subtree has only numbers smaller than a_i. Precisely what we want, and takes $O(\log n)$ time.

Augmented Balanced Binary Search Tree

Assume for simplicity that $a_1, a_2, \ldots, a_n$ are distinct numbers.

- We maintain a dynamic balanced binary search tree T which has only $a_1, \ldots, a_{i-1}$ when **LISEnding(i)** is getting considered.
- We can search for a_i in T to obtain a set of subtrees such that each subtree has only numbers smaller than a_i. Precisely what we want, and takes $O(\log n)$ time.
- We store with the root of each subtree of T the max **LISEnding** value for all indices represented in that subtree.

Augmented Balanced Binary Search Tree

Assume for simplicity that $a_1, a_2, \ldots, a_n$ are distinct numbers.

- We maintain a dynamic balanced binary search tree T which has only $a_1, \ldots, a_{i-1}$ when **LISEnding**(i) is getting considered.
- We can search for a_i in T to obtain a set of subtrees such that each subtree has only numbers smaller than a_i. Precisely what we want, and takes $O(\log n)$ time.
- We store with the root of each subtree of T the max **LISEnding** value for all indices represented in that subtree.
- Updating tree after computing **LISEnding**(i) requires inserting a_i into the tree T and also updating the **LISEnding** values. Can be done in $O(\log n)$ time. Thus, overall $O(n \log n)$ time.

Example

A better algorithm

Using only two arrays. Elegant, fast. See Wikipedia article https://en.wikipedia.org/wiki/Longest_increasing_subsequence

Not a first-cut solution.