Dynamic Programming

Question: Suppose we have a recursive program $\textit{foo}(x)$ that takes an input $x$.
- On input of size $n$ the number of distinct sub-problems that $\textit{foo}(x)$ generates is at most $A(n)$
- $\textit{foo}(x)$ spends at most $B(n)$ time not counting the time for its recursive calls.

Suppose we \textit{memoize} the recursion.

Assumption: Storing and retrieving solutions to pre-computed problems takes $O(1)$ time.

Question: What is an upper bound on the running time of memoized version of $\textit{foo}(x)$ if $|x| = n$? $O(A(n)B(n))$.

Part I

Checking if string is in $L^*$

Problem

Input A string $w \in \Sigma^*$ and access to a language $L \subseteq \Sigma^*$ via function $\text{IsStrInL}(string \ x)$ that decides whether $x$ is in $L$.

Goal Decide if $w \in L^*$ using $\text{IsStrInL}(string \ x)$ as a black box sub-routine

Example

Suppose $L$ is \textit{English} and we have a procedure to check whether a string/word is in the \textit{English} dictionary.
- Is the string “isthisanenglishsentence” in \textit{English}*?
- Is “stampstamp” in \textit{English}*?
- Is “zibzzzad” in \textit{English}*?
Recursive Solution

When is $w \in L^*$?

A $w \in L^*$ if $w \in L$ or if $w = uv$ where $u \in L$ and $v \in L^*$, $|u| \geq 1$.

Assume $w$ is stored in array $A[1..n]$.

**IsStringInLstar($A[1..n]$):**

- If ($n = 0$) Output YES
- If ($IsStrInL(A[1..n])$) Output YES
- Else
  - For ($i = 1$ to $n - 1$) do
    - If ($IsStrInL(A[1..i])$ and $IsStrInLstar(A[i+1..n])$) Output YES
  - Output NO

Output: $ISL(1)$

Question: How many distinct sub-problems does $IsStrInLstar(A[1..n])$ generate? $O(n)$

Example

Consider string *samiam*

Naming subproblems and recursive equation

After seeing that number of subproblems is $O(n)$ we name them to help us understand the structure better.

$ISL(i)$: a boolean which is 1 if $A[i..n]$ is in $L^*$, 0 otherwise

Base case: $ISL(n + 1) = 1$ interpreting $A[n+1..n]$ as $\epsilon$

Recursive relation:

- $ISL(i) = 1$ if
  - $\exists i < j \leq n + 1$ s.t. $ISL(j)$ and $IsStrInL(A[i..(j-1)]$
- $ISL(i) = 0$ otherwise

Output: $ISL(1)$
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.

Caveat: Dynamic programming is not about filling tables. It is about finding a smart recursion. First, find the correct recursion.

Example

Consider string samiam

Iterative Algorithm

```
Iterative Algorithm

IsStringInLstar-Iterative(A[1..n]):

boolean ISL[1..(n + 1)]
ISL[n + 1] = TRUE
for (i = n down to 1)
    ISL[i] = FALSE
    for (j = i + 1 to n + 1)
        if (ISL[j] and IsStrInL(A[i..j - 1]))
            ISL[i] = TRUE
            Break

If (ISL[1] = 1) Output YES
Else Output NO
```

- Running time: $O(n^2)$ (assuming call to IsStrInL is $O(1)$ time)
- Space: $O(n)$

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**.

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

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

Recursive Approach: Take 1

: Longest increasing subsequence
Can we find a recursive algorithm for LIS?

LIS($A[1..n]$):

- **Case 1**: Does not contain $A[n]$ in which case $\text{LIS}(A[1..n]) = \text{LIS}(A[1..(n-1)])$
- **Case 2**: contains $A[n]$ in which case $\text{LIS}(A[1..n])$ is not so clear.

Observation

For second case we want to find a subsequence in $A[1..(n-1)]$ that is restricted to numbers less than $A[n]$. This suggests that a more general problem is $\text{LIS}_{\text{smaller}}(A[1..n], x)$ which gives the longest increasing subsequence in $A$ where each number in the sequence is less than $x$. 


Recursive Approach

\(LIS(A[1..n])\): the length of longest increasing subsequence in \(A\)

\(LIS_{\text{smaller}}(A[1..n], x)\): length of longest increasing subsequence in \(A[1..n]\) with all numbers in subsequence less than \(x\)

\[
\begin{align*}
LIS_{\text{smaller}}(A[1..n], x) : & \\
\text{if } (n = 0) \text{ then return } 0 \\
& m = LIS_{\text{smaller}}(A[1..(n - 1)], x) \\
\text{if } (A[n] < x) \text{ then } & m = \max(m, 1 + LIS_{\text{smaller}}(A[1..(n - 1)], A[n])) \\
& \text{Output } m \\
\end{align*}
\]

\[
\begin{align*}
LIS(A[1..n]) : & \\
& \text{return } LIS_{\text{smaller}}(A[1..n], \infty) \\
\end{align*}
\]

Example

Sequence: \(A[1..7] = 6, 3, 5, 2, 7, 8, 1\)

Naming subproblems and recursive equation

After seeing that number of subproblems is \(O(n^2)\) we name them to help us understand the structure better. For notational ease we add \(\infty\) at end of array (in position \(n + 1\))

\(LIS(i, j)\): length of longest increasing sequence in \(A[1..i]\) among numbers less than \(A[j]\) (defined only for \(i < j\))

Base case: \(LIS(0, j) = 0\) for \(1 \leq j \leq n + 1\)

Recursive relation:

- \(LIS(i, j) = LIS(i - 1, j)\) if \(A[i] > A[j]\)
- \(LIS(i, j) = \max\{LIS(i - 1, j), 1 + LIS(i - 1, i)\}\) if \(A[i] \leq A[j]\)

Output: \(LIS(n, n + 1)\)
Iterative algorithm

\[
\text{LIS-Iterative}(A[1..n]):
\begin{align*}
A[n+1] &= \infty \\
\text{int } LIS[0..n,1..n+1] \\
\text{for } (j = 1 \text{ to } n+1) \text{ do} \\
LIS[0,j] &= 0 \\
\text{for } (i = 1 \text{ to } n) \text{ do} \\
& \quad \text{for } (j = i+1 \text{ to } n) \\
& \quad \quad \text{If } (A[i] > A[j]) \quad LIS[i,j] = LIS[i-1,j] \\
& \quad \quad \text{Else } LIS[i,j] = \max\{LIS[i-1,j], 1 + LIS[i-1,i]\}
\end{align*}
\]

Return \( LIS[n,n+1] \)

Running time: \( O(n^2) \)
Space: \( O(n^2) \)

How to order bottom up computation?

Sequence: \( A[1..7] = 6, 3, 5, 2, 7, 8, 1 \)

Base case: \( LIS(0,j) = 0 \) for \( 1 \leq j \leq n+1 \)

Recursive relation:
- \( LIS(i,j) = LIS(i-1,j) \) if \( A[i] > A[j] \)
- \( LIS(i,j) = \max\{LIS(i-1,j), 1 + LIS(i-1,i)\} \) if \( A[i] \leq A[j] \)

Two comments

Question: Can we compute an optimum solution and not just its value?
Yes! See notes.

Question: Is there a faster algorithm for \( LIS \)? Yes! Using a different recursion and optimizing one can obtain an \( O(n \log n) \) time and \( O(n) \) space algorithm. \( O(n \log n) \) time is not obvious. Depends on improving time by using data structures on top of dynamic programming.
Dynamic Programming

1. Find a “smart” recursion for the problem in which the number of distinct subproblems is small; polynomial in the original problem size.
2. Estimate the number of subproblems, the time to evaluate each subproblem and the space needed to store the value. This gives an upper bound on the total running time if we use automatic memoization.
3. Eliminate recursion and find an iterative algorithm to compute the problems bottom up by storing the intermediate values in an appropriate data structure; need to find the right way or order the subproblem evaluation. This leads to an explicit algorithm.
4. Optimize the resulting algorithm further