CS3383 Unit 3 Lecture 1: Longest Common Subsequence

David Bremner

February 25, 2024

Outline

- Dynamic Programming
  - Longest Common Subsequence

Ordering Subproblems

Ordered Subproblems

In order to solve our problem in a single pass, we need

- An ordered set of subproblems $L(i)$

Ordering Subproblems

Ordered Subproblems

In order to solve our problem in a single pass, we need

- An ordered set of subproblems $L(i)$
- Each subproblem $L(i)$ can be solved using only the answers for $L(j)$, for $j < i$.

Ordering Subproblems

Ordered Subproblems

In order to solve our problem in a single pass, we need

- An ordered set of subproblems $L(i)$
- Each subproblem $L(i)$ can be solved using only the answers for $L(j)$, for $j < i$.

- In hotel problem, (topological) ordering by time

Ordering Subproblems

Ordered Subproblems

In order to solve our problem in a single pass, we need

- An ordered set of subproblems $L(i)$
- Each subproblem $L(i)$ can be solved using only the answers for $L(j)$, for $j < i$.

- In hotel problem, (topological) ordering by time
- Often, by a recurrence relation

Ordering Subproblems

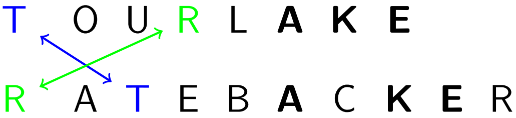
Ordered Subproblems

In order to solve our problem in a single pass, we need

- An ordered set of subproblems $L(i)$
- Each subproblem $L(i)$ can be solved using only the answers for $L(j)$, for $j < i$.

- In hotel problem, (topological) ordering by time
- Often, by a recurrence relation
- For example the Longest Common Subsequence problem.

LCS definition



Given two strings (sequences), find a maximum length subsequence common to both?

Recursive formula for the length

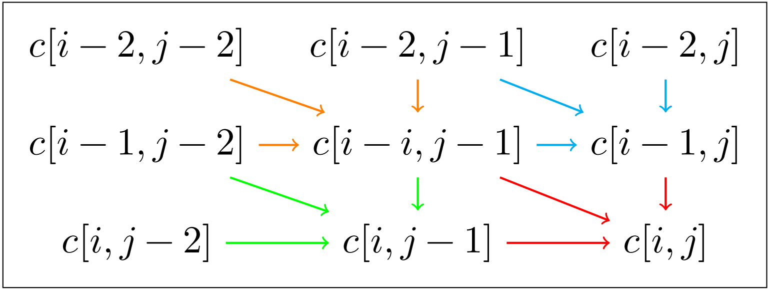
$$c[i,j] := |\,\mathrm{LCS}(x[0\ldots i-1], y[0\ldots j-1])|$$

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i-1] = y[j-1] \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

Recursive formula for the length

$$c[i,j] := |\,\mathrm{LCS}(x[0\dots i-1], y[0\dots j-1])|$$

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i-1] = y[j-1] \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$



Proof of recursion formula

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i-1] = y[j-1] \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

$x[i-1] = y[j-1] = \alpha$

If a common subsequence does not use α as its last element, it can be made longer.

Proof of recursion formula

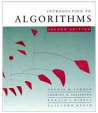
$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i-1] = y[j-1] \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

$x[i-1] \neq y[j-1]$

- LCS does not use the “last” element of x, or
- LCS does not use the “last” element of y

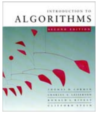
The trouble with recursion

- Although recursion is a useful step to a dynamic programming algorithm, naive recursion is often expensive because of repeated subproblems



Recursive algorithm for LCS

$$
\begin{array}{l}
\text{LCS}(x, y, i, j) \\
\quad \textbf{if } x[i] = y[j] \\
\quad\quad \textbf{then } c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1 \\
\quad\quad \textbf{else } c[i, j] \leftarrow \max\{\text{LCS}(x, y, i-1, j), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{LCS}(x, y, i, j-1)\}
\end{array}
$$



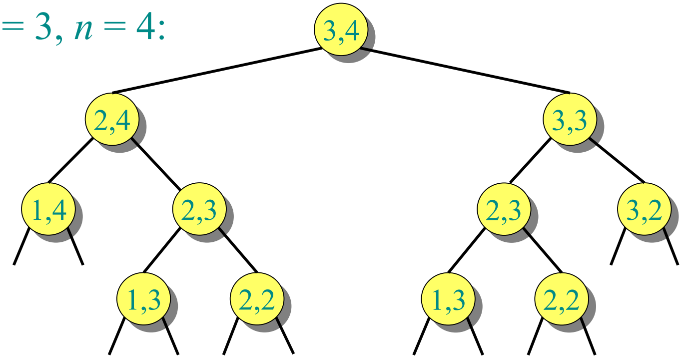
Recursive algorithm for LCS

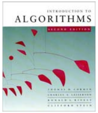
LCS(x, y, i, j)
  if $x[i] = y[j]$
    then $c[i, j] \leftarrow$ LCS$(x, y, i-1, j-1) + 1$
    else $c[i, j] \leftarrow \max\{LCS(x, y, i-1, j)$,
      LCS$(x, y, i, j-1)\}$

Worst-case: $x[i] \neq y[j]$, in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

Recursion tree

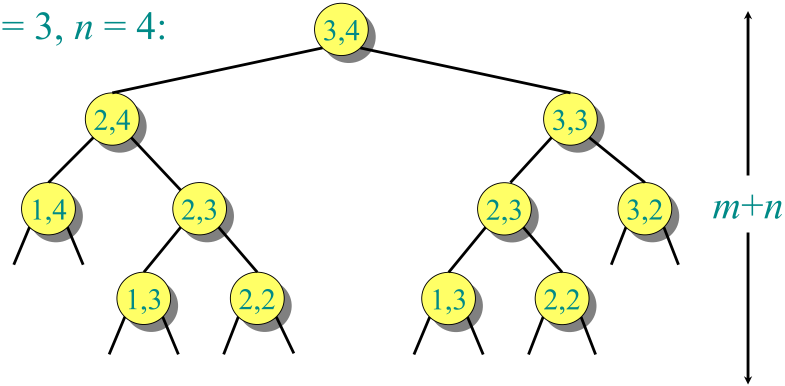
$m = 3, n = 4$:



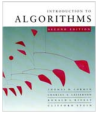


Recursion tree

$m = 3$, $n = 4$:

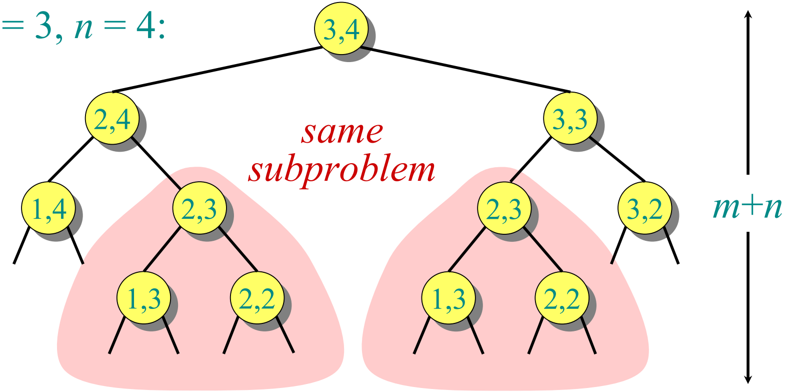


Height $= m + n \Rightarrow$ work potentially exponential.

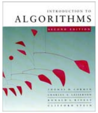


Recursion tree

$m = 3$, $n = 4$:



Height $= m + n \Rightarrow$ work potentially exponential, but we're solving subproblems already solved!



Dynamic-programming hallmark #2

Overlapping subproblems

A recursive solution contains a "small" number of distinct subproblems repeated many times.

Dynamic-programming hallmark #2

Overlapping subproblems

A recursive solution contains a "small" number of distinct subproblems repeated many times.

The number of distinct LCS subproblems for two strings of lengths m and n is only mn.

Memoization

Recursive Version

function $\textsc{Recur}(p_1, \ldots p_k)$

$\vdots$

return val

end function

Memoization

Memoized version

```
function MEMO(p_1, ... p_k)
    if cache[p_1, ... p_k] ≠ NIL then
        return cache[p_1, ... p_k]
    end if
    ⋮
    cache[p_1, ... p_k] = val
    return val
end function
```

Recursive Version

```
function RECUR(p_1, ... p_k)
    ⋮
    return val
end function
```

Memoized LCS

```
def lcs(c,x,y,i,j):
  if (i < 1) or (j<1):
    return 0
  if c[i][j] == None:
    if x[i-1] == y[j-1]:
      c[i][j]=lcs(c,x,y,i-1,j-1)+1
    else:
      c[i][j] = max(lcs(c,x,y,i-1,j),
                    lcs(c,x,y,i,j-1))
  return c[i][j]
```

- $c[i, j]$ written at most once.

Memoized LCS

```
def lcs(c,x,y,i,j):
  if (i < 1) or (j<1):
    return 0
  if c[i][j] == None:
    if x[i-1] == y[j-1]:
      c[i][j]=lcs(c,x,y,i-1,j-1)+1
    else:
      c[i][j] = max(lcs(c,x,y,i-1,j),
                    lcs(c,x,y,i,j-1))
  return c[i][j]
```

- $c[i, j]$ written at most once.
- returned value written immediately

Memoized LCS

```
def lcs(c,x,y,i,j):
  if (i < 1) or (j<1):
    return 0
  if c[i][j] == None:
    if x[i-1] == y[j-1]:
      c[i][j]=lcs(c,x,y,i-1,j-1)+1
    else:
      c[i][j] = max(lcs(c,x,y,i-1,j),
                    lcs(c,x,y,i,j-1))
  return c[i][j]
```

- $c[i, j]$ written at most once.
- returned value written immediately
- charge all work to writes

Eliminating Recursion completely

```
def lcs(x,y):
  n = len(x); m=len(y)
  c = [ [ 0 for j in range(m+1) ]
          for i in range(n+1) ]
  for i in range(1,n+1):
    for j in range(1,m+1):
      if x[i-1] == y[j-1]:
        c[i][j] = c[i-1][j-1]+1
      else:
        c[i][j] = max(c[i-1][j],
                      c[i][j-1])
  return c
```

Comparing Memoized to Iterative LCS

- Asymptotic time is the same

Comparing Memoized to Iterative LCS

- Asymptotic time is the same
- Iterative version is typically faster/more robust in practice

Comparing Memoized to Iterative LCS

- Asymptotic time is the same
- Iterative version is typically faster/more robust in practice
- memoized version is easier to derive (even automatically) from the recursive version.

Comparing Memoized to Iterative LCS

- Asymptotic time is the same
- Iterative version is typically faster/more robust in practice
- memoized version is easier to derive (even automatically) from the recursive version.
- Iterative version is easier to analyze

Comparing Memoized to Iterative LCS

- Asymptotic time is the same
- Iterative version is typically faster/more robust in practice
- memoized version is easier to derive (even automatically) from the recursive version.
- Iterative version is easier to analyze
- Both versions add extra memory use to pure recursion.

Comparing Memoized to Iterative LCS

- Asymptotic time is the same
- Iterative version is typically faster/more robust in practice
- memoized version is easier to derive (even automatically) from the recursive version.
- Iterative version is easier to analyze
- Both versions add extra memory use to pure recursion.
- Memoization never solves unneeded subproblems.

Reading back the sequence

```
def backtrace(c,x,y,i,j):
  if (i<1) or (j<1):
    return ""
  elif x[i-1] == y[j-1]:
    return backtrace(c,x,y,i-1,j-1) \
      +x[i-1]
  elif (c[i][j-1] > c[i-1][j]):
    return backtrace(c,x,y,i,j-1)
  else:
    return backtrace(c,x,y,i-1,j)
```

- What is the running time?