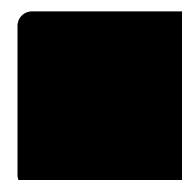
CS3383 Unit 3 Lecture 1: Longest Common Subsequence

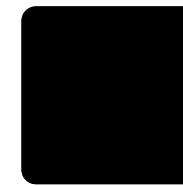
David Bremner

February 25, 2024



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 $L(i)$
- 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.

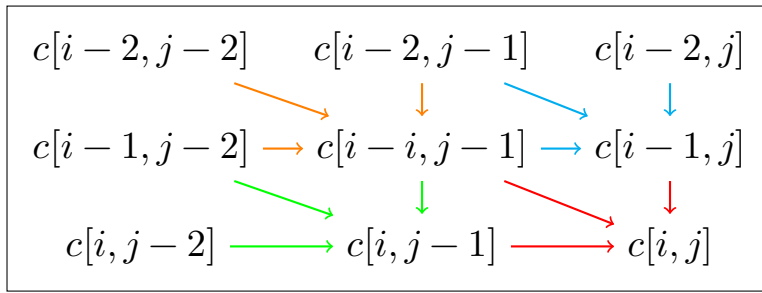
LCS definition

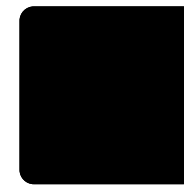
Given two strings (sequences), find a maximum length subsequence common to both?

Recursive formula for the length

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

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

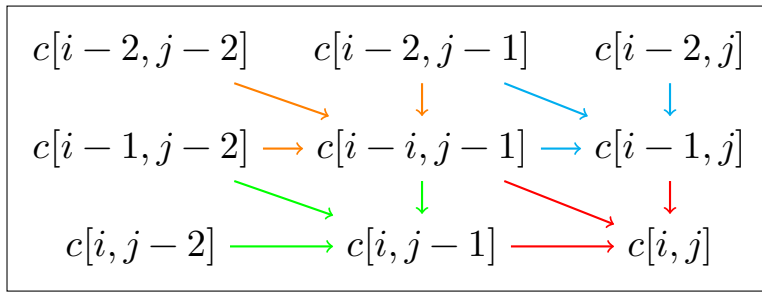


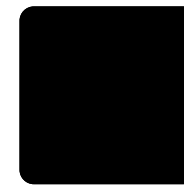


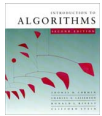
Recursive formula for the length

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

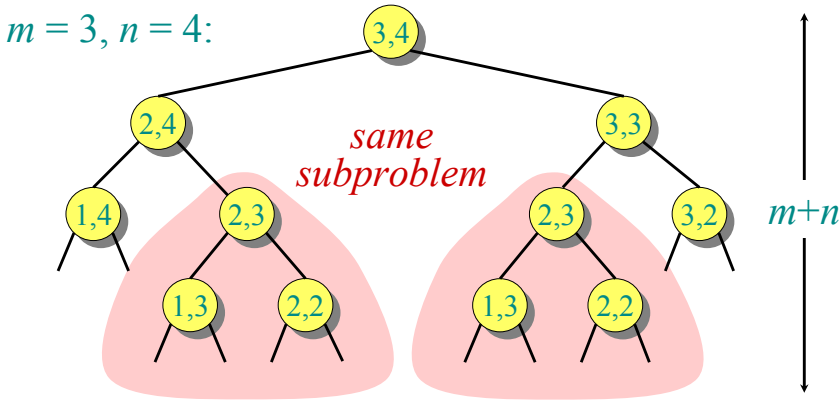
$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i-1] = y[j-1] \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$







Recursion tree



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

Memoization

Memoized version

function $\textsc{Memo}(p_1, \ldots p_k)$
  if cache$[p_1, \ldots p_k] \neq$ NIL **then**
    return cache$[p_1, \ldots p_k]$
  end if
  $\vdots$
  cache$[p_1, \ldots p_k] =$ val
  return val
end function

Recursive Version

function $\textsc{Recur}(p_1, \ldots p_k)$
  $\vdots$
  return val
end function

Memoization

Memoized version

function $\textsc{Memo}(p_1, \ldots p_k)$
  if cache$[p_1, \ldots p_k] \neq$ NIL **then**
    return cache$[p_1, \ldots p_k]$
  end if
  $\vdots$
  cache$[p_1, \ldots p_k] =$ val
  return val
end function

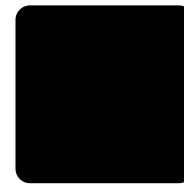
Recursive Version

function $\textsc{Recur}(p_1, \ldots p_k)$
  $\vdots$
  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.
- returned value written immediately
- charge all work to writes

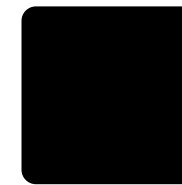


Eliminating Recursion completely

```
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
- Iterative version is typically faster
- Memoized version is easier to derive from the recursion.
- 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)
```

- time complexity?