Estratégias Algorítmicas 2020/21
Week 4 – Dynamic Programming

· U C ·

UNIVERSIDADE DE COIMBRA

Outline

1. Introduction
2. Longest Increasing Subsequence
3. Longest Common Subsequence

Reading about problem solving with dynamic programming

- J. Erickson, Algorithms, Chp 3
- T. Cormen et al., Introduction to Algorithms, Chp 15
- J. Edmonds, How to think about algorithms, Chp 18, 19
- S.S. Skiena, M.G. Revilla, Programming Challenges, Chp 11

Problem decomposition

- A problem may be decomposed in a sequence of nested subproblems
- The original problem is solved by combining the solutions to the various subproblems
- The choices made at the inner levels influence the choices to be made at the outer levels (in general)

Dynamic Programming

- Solve an optimization problem by caching subproblem solutions (*memoization*) rather than recomputing them
- Usually, the number of subproblems is “small” (ideally, polynomial in the input size)

Two properties:

1. *Optimal substructure property*: An optimal solution to a problem contains within it optimal solutions to subproblems
2. *Overlapping subproblems*: The solution to subproblems can be reused several times

Introduction

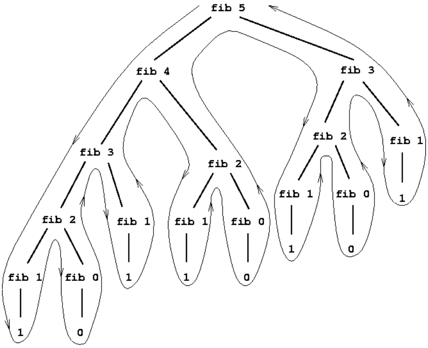
Function $fib(n)$

if $n = 0$ **or** $n = 1$ **then** {base case}

return n

else {recursive step}

return $fib(n-1) + fib(n-2)$



Top-down Dynamic Programming (with memoizing)

Function $fib(n)$

    if $T[n]$ is cached **then**
        return $T[n]$
    if $n = 0$ **or** $n = 1$ **then**
        T[n] = n
        return $T[n]$
    else
        $T[n] = fib(n-1) + fib(n-2)$
        return $T[n]$

Bottom-up Dynamic Programming

Function $fib(n)$

  $T[0] = 0$
  $T[1] = 1$
  for $i = 2$ **to** n **do**
    $T[i] = T[i-2] + T[i-1]$
  return $T[n]$

Introduction

Our approach for a given problem

1. Find a suitable notion of subproblem*
2. Define the recurrence for that notion of subproblem
3. Build a recursive algorithm
4. Build a top-down dynamic programming approach
5. Build a bottom-up dynamic programming approach

**Suitable* means that both properties hold in general (using induction). In the following examples, we only prove the *optimal substructure property*.

Rationale for proving optimal substructure (*cut & paste* proof)

An optimal solution S for a problem P contains an optimal solution for a (related) subproblem P'

1. (assumption) S is an optimal solution for problem P
2. (negation) S contains suboptimal solution S' for subproblem P'. Then, there exists an optimal solution R' for P' (i.e, R' is better than S')
3. (consequence) Then, it is possible to build a solution R to problem P that contains R' and is better than S.
4. (contradition) But, S cannot be optimal to problem P, which leads to a contradiction of 1.

Solution S must contain an optimal solution to subproblem P'!

Introduction

Problems

- Sequence prefixes: Longest Increasing Subsequence, Longest Common Subsequence, Edit Distance and Sequence Alignment
- Subset subproblems: Coin Changing, Subset Sum and Knapsack.

Longest Increasing Subsequence

- Consider this sequence of integers

  (0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 15, 7)

- What is the longest (monotonically) increasing subsequence?

Longest Increasing Subsequence

- Consider this sequence of integers

  (0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 15, 7)

- What is the longest (monotonically) increasing subsequence?

  (0, 2, 6, 9, 13, 15)

- Not unique. For instance: (0, 4, 6, 9, 11, 15)

Longest Increasing Subsequence

- Consider this sequence of integers

  (0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 15, 7)

- What is the longest (monotonically) increasing subsequence?

  (0, 2, 6, 9, 13, 15)

- Not unique. For instance: (0, 4, 6, 9, 11, 15)

Subproblem: Given a sequence $S = (s_1, \ldots, s_n)$, let $LIS(i)$ be the longest increasing subsequence (LIS) that ends with s_i.

The longest among $LIS(1), LIS(2), \ldots, LIS(n)$ gives the solution to the problem.

Optimal substructure property:

Given a sequence $S = (s_1, \ldots, s_n)$, let $LIS(i)$ be the LIS that ends with s_i. Then if s_i is removed from $LIS(i)$, we obtain 1) $LIS(j)$, $s_j < s_i$, $j < i$, or 2) the empty sequence. Let's prove 1):

1. (assumption) Assume that $LIS(i)$ is the LIS that ends with s_i
2. (negation) Now, assume that $|LIS(j)| > |LIS(i) \backslash \{s_i\}|$
3. (consequence) Then, appending s_i to $LIS(j)$ generates a sequence longer than $LIS(i)$: $|LIS(j) \cup \{s_i\}| > |LIS(i)|$
4. (contradition) But, this leads to a contradiction of 1

Therefore, $LIS(i) \backslash \{s_i\}$ must be $LIS(j)$

Longest Increasing Subsequence

Recursion to compute $L(i) = |LIS(i)|$.

$$L(i) = \begin{cases} 1 & \text{if } i = 1 \\ 1 + \max\{L(j) : 1 \leq j < i \text{ and } s_j < s_i\} & \text{otherwise} \end{cases}$$

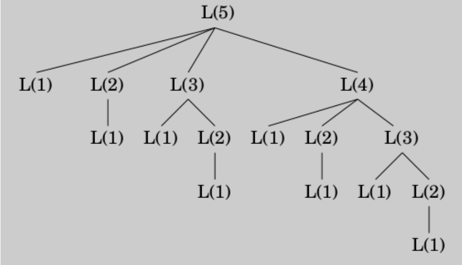
LIS can be solved recursively (only the size of the LIS of S)

```
Function lis(S, i)
  if i = 1 then
     L[1] = 1
  else
     L[1] = 0
     for j = 1 to i − 1 do
        L[j] = lis(S, j)
        if s_j < s_i and L_j > L_i  then
           L[i] = L[j]
     L[i] = L[i] + 1
  return  L[i]                          {L[i] gives the size of LIS(i)}
```

The size of the LIS is given by the maximum of $L[1], L[2], \ldots, L[n]$

Longest Increasing Subsequence

You may get exponentally many nodes in the call recursion tree:



But $L(i)$ can be cached - Top-down DP.

Longest Increasing Subsequence

Top-down dynamic programming

Function $lis(S, i)$
  if $L[i]$ is cached **then**
    return $L[i]$
  if $i = 1$ **then**
    $L[i] = 1$
  else
    $L[i] = 0$
    for $j = 1$ **to** $i - 1$ **do**
      $L[j] = lis(S, j)$
      if $s_j < s_i$ **and** $L[j] > L[i]$ **then**
        $L[i] = L[j]$
    $L[i] = L[i] + 1$
  return $L[i]$ $\{L[i]$ gives the size of $LIS(i)\}$

The size of the LIS is given by the maximum of $L[1], L[2], \ldots, L[n]$

Longest Increasing Subsequence

- There are $O(n)$ overlapping subproblems, which suggests a $O(n^2)$ (bottom up) dynamic programming algorithm:
  1. For each position $i = 1, \ldots, n$, find the largest LIS for positions $j < i$ such that $s_j < s_i$; append s_i to it.
  2. Return the largest LIS found.

Longest Increasing Subsequence

- There are $O(n)$ overlapping subproblems, which suggests a $O(n^2)$ (bottom up) dynamic programming algorithm:
  1. For each position $i = 1, \ldots, n$, find the largest LIS for positions $j < i$ such that $s_j < s_i$; append s_i to it.
  2. Return the largest LIS found.

Example

S	0	8	4	12	2	10	6	14	1	9	5	13	3	11	15	7
L[i]	1	2	2	3	2	3	3	4	2	4	3	5	3	5	6	4

The largest LIS contains 6 characters

Longest Increasing Subsequence

Bottom-up dynamic programming

Function *lis*(S)

    $L[1] = 1$
    for $i = 2$ **to** n **do**
        $L[i] = 0$
        for $j = 1$ **to** $i - 1$ **do**
            if $s_j < s_i$ **and** $L[j] > L[i]$ **then**
                $L[i] = L[j]$
        $L[i] = L[i] + 1$
    return $\max(L[1], \ldots, L[n])$

It has $O(n^2)$ time complexity.

Longest Increasing Subsequence

Example

S	0	8	4	12	2	10	6	14	1	9	5	13	3	11	15	7
L[i]	1	2	2	3	2	3	3	4	2	4	3	5	3	5	6	4

How to reconstruct an optimal subsequence?

Example

S	0	8	4	12	2	10	6	14	1	9	5	13	3	11	15	7
L[i]	1	2	2	3	2	3	3	4	2	4	3	5	3	5	6	4

Start from the largest LIS and scan from right to left, choosing a smaller number with next unitary decrement in #LIS

Longest Common Subsequence

LCS = eeao LCS = steao LCS = teao

Longest Common Subsequence

- The longest common subsequence (LCS) problem is to find the longest subsequence common to all sequences in a set of sequences.
- This is the basis of the `diff` program and it has many applications in bioinformatics.
- It is NP-hard for an arbitrary number of input sequences.
- If the number of sequences is fixed, it can be solved by dynamic programming. We will only consider two.

Longest Common Subsequence

Given two sequences $A = (a_1, \dots, a_n)$ and $B = (b_1, \dots, b_m)$, let $LCS(i, j)$ denote the length of the LCS of $A[1..i]$ and $B[1..j]$.

Table representation:

	E	S	T	E	A	N	O
S	$LCS(1,1)$	$LCS(1,2)$	$LCS(1,3)$	$\cdots$			
P	$LCS(2,1)$	$LCS(2,2)$	$LCS(2,3)$	$\cdots$			
O	$LCS(3,1)$	$LCS(3,2)$	$LCS(3,3)$	$\cdots$			
R	$LCS(4,1)$	$LCS(4,2)$	$LCS(4,3)$	$\cdots$			
T	$LCS(5,1)$	$LCS(5,2)$	$LCS(5,3)$	$\cdots$			
I	$LCS(6,1)$	$LCS(6,2)$	$LCS(6,3)$	$\cdots$			
N	$\cdots$	$\cdots$	$\cdots$	$\cdots$		$\cdots$	$\cdots$
G						$\cdots$	$LCS(8,7)$

Longest Common Subsequence

Let $Z = (z_1, \ldots, z_k)$ be an LCS of A and B.

If $a_n = b_m$ then

1. $z_k = a_n = b_m$

   Proof: If $z_k \neq a_n$, then append a_n to Z and obtain $LCS(n, m) > k$, which is a contradiction.

2. $(z_1, \ldots, z_{k-1})$ is the LCS of $A[1..n-1]$ and $B[1..m-1]$

   Proof: Assume that $(z_1, \ldots, z_{k-1})$ is not LCS of $A[1..n-1]$ and $B[1..m-1]$. Then, it exists a longer LCS, say W, for the same subsequences. Then append a_n to W and obtain a subsequence longer than Z, which is a contradiction.

If $a_i = b_j$ then $LCS(i, j) = LCS(i-1, j-1) + 1$

Example

`CARNAVAL`
`NATAL`

- $LCS(8,5) = 4$, which corresponds to (N,A,A,L).
- If L is removed from (N,A,A,L), then we obtain the LCS for `CARVANA` and `NATA`, which has a length $LCS(7,4) = 3$.
- Otherwise, $LCS(7,4) > 3$ would imply that $LCS(8,5) > 4$, which is a contradiction.

Longest Common Subsequence

Let $Z = (z_1, \ldots, z_k)$ be an LCS of A and B.

If $a_n \neq b_m$

1. If $z_k \neq a_n$, then Z is an LCS of $A[1..n-1]$ and $B[1..m]$

2. If $z_k \neq b_m$, then Z is an LCS of $A[1..n]$ and $B[1..m-1]$

Proof:

1. If $z_k \neq a_n$, then Z is a common subsequence of $A[1..n-1]$ and $B[1..m]$. If it exists a longer common subsequence, it would also be longer than Z for A and B, which is a contradiction.

2. Symmetric to [1.].

Longest Common Subsequence

Let $Z = (z_1, \ldots, z_k)$ be an LCS of A and B.

If $a_n \neq b_m$

1. If $z_k \neq a_n$, then Z is an LCS of $A[1..n-1]$ and $B[1..m]$
2. If $z_k \neq b_m$, then Z is an LCS of $A[1..n]$ and $B[1..m-1]$

Proof:

1. If $z_k \neq a_n$, then Z is a common subsequence of $A[1..n-1]$ and $B[1..m]$. If it exists a longer common subsequence, it would also be longer than Z for A and B, which is a contradiction.
2. Symmetric to [1.].

If $a_i \neq b_j$ then $LCS(i,j) = \max\{LCS(i-1,j), LCS(i,j-1)\}$

Second case: when $a_n \neq b_m$.

`COROA`
`PORTO`

- $LCS(5,5) = 3$, which corresponds to (O,R,O).
- As A $\notin$ (O,R,O), then LCS for `CORO` and `PORTO` is also $LCS(4,5) = 3$
- Otherwise if $LCS(4,5) > 3$ would imply that $LCS(5,5) > 3$, which is a contradiction.

Longest Common Subsequence

The computation of $LCS(i,j)$ can be written as a recurrence:

$$LCS(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \max(LCS(i-1,j), LCS(i,j-1)) & \text{if } a_i \neq b_j \\ LCS(i-1,j-1)+1 & \text{if } a_i = b_j \end{cases}$$

Note that when $i = 0$ ($j = 0$), the first (second) sequence is empty.

Function $lcs(A[1..i], B[1..j])$
  if $i = 0$ **or** $j = 0$ **then** {base case}
    return 0
  if $a_i = b_j$ **then**
    return $lcs(A[1..i-1], B[1..j-1]) + 1$
  else
    $LCS_1 = lcs(A[1..i-1], B[1..j])$
    $LCS_2 = lcs(A[1..i], B[1..j-1])$
    return $\max(LCS_1, LCS_2)$

- This agorithm is slow since we are performing the same operations several times.

Longest Common Subsequence

Function $lcs(A[1..i], B[1..j])$
  if $LCS[i,j]$ is cached **then**
    return $LCS[i,j]$
  if $i = 0$ **or** $j = 0$ **then** {base case}
    $LCS[i,j] = 0$
    return $LCS[i,j]$
  if $a_i = b_j$ **then**
    $LCS[i,j] = lcs(A[1..i-1], B[1..j-1]) + 1$ {Memoizing}
  else
    $LCS_1 = lcs(A[1..i-1], B[1..j])$
    $LCS_2 = lcs(A[1..i], B[1..j-1])$
    $LCS[i,j] = \max(LCS_1, LCS_2)$ {Memoizing}
  return $LCS[i,j]$

- Store LCS values of computed subsequences in table LCS

Longest Common Subsequence

		S	P	O	R	T	I	N	G
	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0	0	0
S	0	1	1	1	1	1	1	1	1
T	0	1	1	1	1	2	2	2	2
E	0	1	1	1	1	2	2	2	2
A	0	1	1	1	1	2	2	2	2
N	0	1	1	1	1	2	2	3	3
O	0	1	1	2	2	2	2	3	3

How to construct the matrix with bottom-up DP?

Longest Common Subsequence

Function $lcs(A, B)$

for $i = 0$ **to** n **do** {1st base case}
  $LCS[i, 0] = 0$
for $j = 0$ **to** m **do** {2nd base case}
  $LCS[0, j] = 0$
for $i = 1$ **to** n **do**
  for $j = 1$ **to** m **do**
    if $a_i = b_j$ **then**
      $LCS[i, j] = LCS[i - 1, j - 1] + 1$
    else
      $LCS[i, j] = \max(LCS[i - 1, j], LCS[i, j - 1])$
return $LCS[n, m]$

- Bottom-up approach in $O(mn)$ time.

Longest Common Subsequence

		S	P	O	R	T	I	N	G
	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0	0	0
S	0	1	1	1	1	1	1	1	1
T	0	1	1	1	1	2	2	2	2
E	0	1	1	1	1	2	2	2	2
A	0	1	1	1	1	2	2	2	2
N	0	1	1	1	1	2	2	3	3
O	0	1	1	2	2	2	2	3	3

- How to reconstruct the subsequence from the table?

Longest Common Subsequence

		S	P	O	R	T	I	N	G
	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0	0	0
S	0	1	1	1	1	1	1	1	1
T	0	1	1	1	1	2	2	2	2
E	0	1	1	1	1	2	2	2	2
A	0	1	1	1	1	2	2	2	2
N	0	1	1	1	1	2	2	3	3
O	0	1	1	2	2	2	2	3	3

The LCS is `STN`

Longest Common Subsequence

Function $backtrack(LCS, i, j)$
  if $i = 0$ **or** $j = 0$ **then** {base case}
    return “ ”
  if $a_i = b_j$ **then**
    return $backtrack(LCS, i-1, j-1) + a_i$
  else
    if $LCS[i, j-1] \geq LCS[i-1, j]$ **then**
      return $backtrack(LCS, i, j-1) + a_i$
    else
      return $backtrack(LCS, i-1, j) + b_i$
