Laboratório de Programação Avançada 2018/19

Week 4 – Dynamic Programming

· U C ·

UNIVERSIDADE DE COIMBRA

Outline

Reading about problem solving with dynamic programming

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

Problem decomposition

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

Problem decomposition

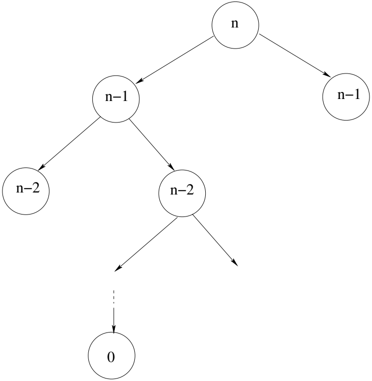
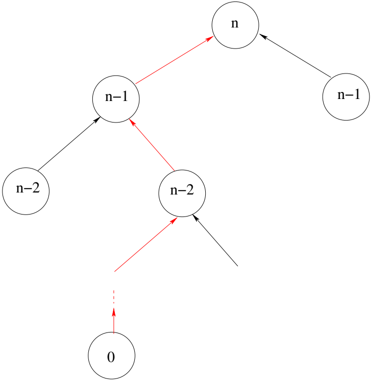


Illustration of problem decomposition in a recursion call tree.

Problem decomposition



Computation of the solution in a recursion call tree.

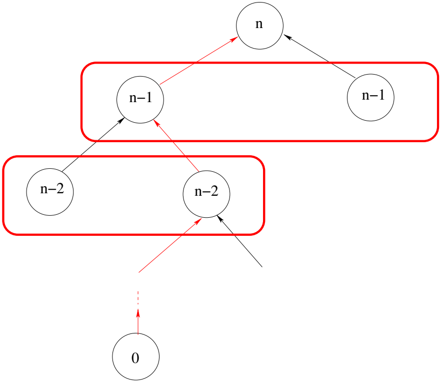
Dynamic Programming

- Solve an optimization problem by caching subproblem solutions (*memoization*) rather than recomputing them
- Usually, the number of sub-problems 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 sub-problems
2. *Overlapping sub-problems*: The solution to sub-problems can be reused several times

An example

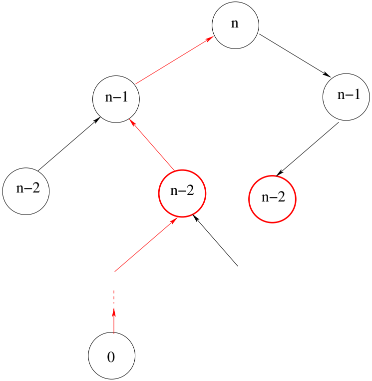


Optimal substructure: An optimal solution to a sub-problem of size k contains an optimal solution to a sub-problem of size $k-1$

Then, to obtain the optimal solution to the sub-problem of size k, select the best solution from all sub-problems of size $k-1$ and update it accordingly.

Note: In other problems, the optimal solution of size k may contain the optimal solution to a subproblem of size $j < k-1$

An example

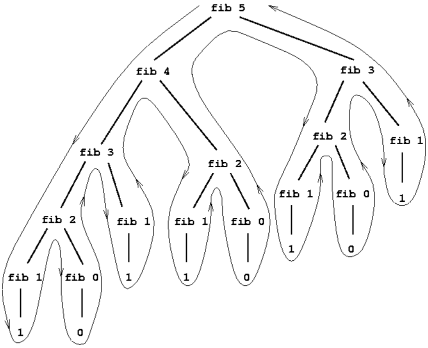


Overlapping sub-problems: The solution to a sub-problem can be reused several times.

Then, store the solutions of the sub-problems to avoid solving them again later on.

Introduction

Function $\mathit{fib}(n)$
  if $n = 0$ **or** $n = 1$ **then** {base case}
    return n
  else {recursive step}
    return $\mathit{fib}(n-1) + \mathit{fib}(n-2)$



Top-down Dynamic Programming (with memoizing)

Function $\mathit{fib}(n)$
  if $T[n]$ is cached **then**
    return $T[n]$
  if $n = 0$ **or** $n = 1$ **then**
    T[n] = n
    return $T[n]$
  else
    $T[n] = \mathit{fib}(n-1) + \mathit{fib}(n-2)$
    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 sub-problem*
2. Define the recurrence for that notion of sub-problem
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*.

Introduction

Problems

- Sequence prefixes: Longest Increasing Subsequence, Longest Common Subsequence, Edit Distance and Sequence Alignment
- Subset sub-problems: 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 by contradiction:

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 $L(i) = |LIS(i)|$.

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

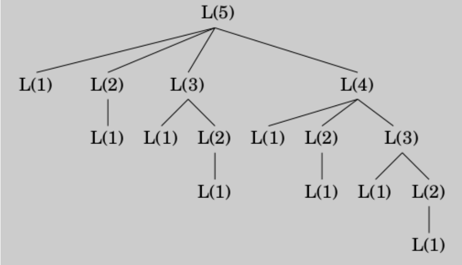
Longest Increasing Subsequence

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_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 $L_1, L_2, \ldots, L_n$

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



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

Top-down dynamic programming

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

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

- There are $O(n)$ overlapping sub-problems, 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.

- There are $O(n)$ overlapping sub-problems, 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
#LIS	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 $lis(S)$

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

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

Example

S	0	8	4	12	2	10	6	14	1	9	5	13	3	11	15	7
#LIS	1	2	2	3	2	3	3	4	2	4	3	5	3	5	6	4

How to reconstruct an optimal subsequence?

Longest Increasing Subsequence

Example

S	0	8	4	12	2	10	6	14	1	9	5	13	3	11	15	7
#LIS	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