Algorithmic Strategies 2024/25

Week 4 – Dynamic Programming

· U C ·

Universidade de Coimbra

Outline

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)

Introduction

Dynamic Programming

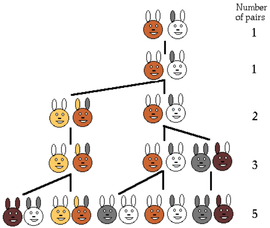
- Solve an optimization problem by caching subproblem solutions (*memoization*) rather than recomputing them
- Usually, the number of subproblems is "small" (ideally, polynomial in the input size)

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

Problem: Fibonacci numbers

A man has one pair of rabbits at a certain place entirely surrounded by a wall. We wish to know how many pairs can be bred from it in one year, if the nature of these rabbits is such that they breed every month one pair (male and female), that in turn will begin to breed in the second month after their birth.



Introduction

Recursion: $fib(n) = fib(n-1) + fib(n-2)$

Base case: $fib(0) = 0, fib(1) = 1$

Recursion: $fib(n) = fib(n-1) + fib(n-2)$

Base case: $fib(0) = 0, fib(1) = 1$

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

Bad example of recursion: Excessive recomputation since it does not take into account that $fib(n-2)$ was already computed.

Introduction

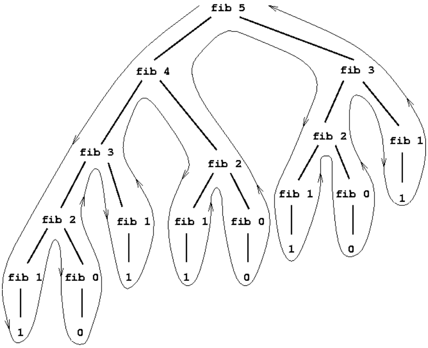
Function $fib(n)$

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**
    $T[n] = n$
  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 $|LIS(j)| > |LIS(i) \setminus \{s_i\}|$
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, $LIS(i) \setminus \{s_i\}$ must be $LIS(j)$

Longest Increasing Subsequence

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

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

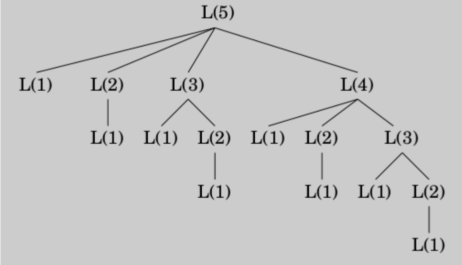
Longest Increasing Subsequence

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
    $L[1] = 0$
    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 $L[1], L[2], \ldots, L[n]$

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 $L[1], L[2], \ldots, L[n]$

- 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], ..., 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?

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

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