CS 341: Algorithms

Douglas R. Stinson

David R. Cheriton School of Computer Science
University of Waterloo

Wintxter, 2015

Table of Contents

Computing Fibonacci Numbers Inefficiently

Algorithm: *BadFib*(n)
if $n = 0$ **then** $f \leftarrow 0$
else if $n = 1$ **then** $f \leftarrow 1$
else
$$\begin{cases} f_1 \leftarrow \textit{BadFib}(n-1) \\ f_2 \leftarrow \textit{BadFib}(n-2) \\ f \leftarrow f_1 + f_2 \end{cases}$$
return (f);

Computing Fibonacci Numbers More Efficiently

Algorithm: *BetterFib*(n)
$f[0] \leftarrow 0$
$f[1] \leftarrow 1$
for $i \leftarrow 2$ **to** n
  do $f[i] \leftarrow f[i-1] + f[i-2]$
return $(f[n])$

Designing Dynamic Programming Algorithms for Optimization Problems

Optimal Structure

Examine the structure of an optimal solution to a problem instance I, and determine if an optimal solution for I can be expressed in terms of optimal solutions to certain **subproblems** of I.

Define Subproblems

Define a set of subproblems $\mathcal{S}(I)$ of the instance I, the solution of which enables the optimal solution of I to be computed. I will be the last or largest instance in the set $\mathcal{S}(I)$.

Designing Dynamic Programming Algorithms (cont.)

Recurrence Relation

Derive a **recurrence relation** on the optimal solutions to the instances in $\mathcal{S}(I)$. This recurrence relation should be completely specified in terms of optimal solutions to (smaller) instances in $\mathcal{S}(I)$ and/or base cases.

Compute Optimal Solutions

Compute the optimal solutions to all the instances in $\mathcal{S}(I)$. Compute these solutions using the recurrence relation in a **bottom-up** fashion, filling in a table of values containing these optimal solutions. Whenever a particular table entry is filled in using the recurrence relation, the optimal solutions of relevant subproblems can be looked up in the table (they have been computed already). The final table entry is the solution to I.

0-1 Knapsack

Problem

0-1 Knapsack

Instance: **Profits** $P = [p_1, \ldots, p_n]$; **weights** $W = [w_1, \ldots, w_n]$; *and a* **capacity**, M. *These are all positive integers.*

Feasible solution: *An* n*-tuple* $X = [x_1, \ldots, x_n]$, *where* $x_i \in \{0, 1\}$ *for* $1 \leq i \leq n$, *and* $\sum_{i=1}^{n} w_i x_i \leq M$.

Find: *A feasible solution* X *that maximizes* $\sum_{i=1}^{n} p_i x_i$.

Let $P[i, m]$ denote the optimal solution to the subproblem consisting of the first i objects (having profits $p_1, \ldots, p_i$ and weights $w_1, \ldots, w_i$, respectively) and capacity m.

A Dynamic Programming Algorithm for 0-1 Knapsack

Algorithm: *0-1Knapsack*$(p_1, \ldots, p_n, w_1, \ldots, w_n, M)$

for $m \leftarrow 0$ **to** M

do { **if** $m \geq w_1$
  then $P[1, m] \leftarrow p_1$
  else $P[1, m] \leftarrow 0$

for $i \leftarrow 2$ **to** n

do { **for** $m \leftarrow 0$ **to** M
  do { **if** $m < w_i$
    then $P[i, m] \leftarrow P[i-1, m]$
    else $P[i, m] \leftarrow \max\{P[i-1, m-w_i] + p_i, P[i-1, m]\}$

return $(P[n, M])$;

Computing the Optimal Knapsack X

Algorithm: *0-1Knapsack*$(p_1, \ldots, p_n, w_1, \ldots, w_n, M, P)$

$m \leftarrow M$

$p \leftarrow P[n, M]$

for $i \leftarrow n$ **downto** 2

do
- **if** $p = P[i-1, m]$
  - **then** $x_i \leftarrow 0$
  - **else**
    - $x_i \leftarrow 1$
    - $p \leftarrow p - p_i$
    - $m \leftarrow m - w_i$

if $p = 0$

then $x_1 \leftarrow 0$

else $x_1 \leftarrow 1$

return (X);

Coin Changing

Problem

Coin Changing

Instance: *A list of* **coin denominations**, $1 = d_1, d_2, \ldots, d_n$, *and a positive integer* T, *which is called the* **target sum**.

Find: *An* n*-tuple of non-negative integers, say* $A = [a_1, \ldots, a_n]$, *such that* $T = \sum_{i=1}^{n} a_i d_i$ *and such that* $N = \sum_{i=1}^{n} a_i$ *is minimized.*

Let $N[i, t]$ denote the optimal solution to the subproblem consisting of the first i coin denominations $p_1, \ldots, p_i$ and target sum t. Let $A[i, t]$ denote the number of coins of denomination d_i used in the optimal solution to this subproblem.

A Dynamic Programming Algorithm for Coin Changing

Algorithm: *Coin Changing*$(d_1, \ldots, d_n, T)$

comment: $d_1 = 1$

for $t \leftarrow 0$ **to** T

do $\begin{cases} N[1,t] \leftarrow t \\ A[1,t] \leftarrow t \end{cases}$

for $i \leftarrow 2$ **to** n

do $\begin{cases} \textbf{for } t \leftarrow 0 \textbf{ to } T \\ \quad \textbf{do} \begin{cases} N[i,t] \leftarrow N[i-1,t] \\ A[i,t] \leftarrow 0 \\ \textbf{for } j \leftarrow 1 \textbf{ to } \lfloor (t/d_i) \rfloor \\ \quad \textbf{do} \begin{cases} \textbf{if } j + N[i-1, t-jd_i] < N[i,t] \\ \quad \textbf{then} \begin{cases} N[i,t] \leftarrow j + N[i-1,t-jd_i] \\ A[i,t] \leftarrow j \end{cases} \end{cases} \end{cases} \end{cases}$

return $(N[n,T])$;

Longest Common Subsequence

Problem

Longest Common Subsequence

Instance: *Two sequences $X = (x_1, \ldots, x_m)$ and $Y = (y_1, \ldots, y_n)$ over some finite alphabet Γ.*

Find: *A maximum length sequence Z that is a subsequence of both X and Y.*

$Z = (z_1, \ldots, z_k)$ is a **subsequence** of X if there exist indices $1 \leq i_1 < \cdots < i_\ell \leq m$ such that $z_j = x_{i_j}$, $1 \leq j \leq \ell$.

Similarly, Z is a subsequence of Y if there exist (possibly different) indices $1 \leq h_1 < \cdots < h_\ell \leq n$ such that $z_j = y_{h_j}$, $1 \leq j \leq \ell$.

Computing the Length of the LCS of X and Y

Algorithm: *LCS1*$(X = (x_1, \ldots, x_m), Y = (y_1, \ldots, y_n))$

for $i \leftarrow 0$ **to** m
  do $c[i, 0] \leftarrow 0$
for $j \leftarrow 0$ **to** n
  do $c[0, j] \leftarrow 0$
for $i \leftarrow 1$ **to** m
  do **for** $j \leftarrow 1$ **to** n
    do **if** $x_i = y_j$
      then $c[i, j] \leftarrow c[i-1, j-1] + 1$
      else $c[i, j] \leftarrow \max\{c[i, j-1], c[i-1, j]\}$
return $(c[m, n])$;

Finding the LCS of X and Y

Algorithm: *LCS2*$(X = (x_1, \ldots, x_m), Y = (y_1, \ldots, y_n))$

for $i \leftarrow 0$ **to** m **do** $c[i, 0] \leftarrow 0$

for $j \leftarrow 0$ **to** n **do** $c[0, j] \leftarrow 0$

for $i \leftarrow 1$ **to** m

do { **for** $j \leftarrow 1$ **to** n

do { **if** $x_i = y_j$

then { $c[i, j] \leftarrow c[i-1, j-1] + 1$
$\pi[i, j] \leftarrow \mathbf{UL}$

else if $c[i, j-1] > c[i-1, j]$

then { $c[i, j] \leftarrow c[i, j-1]$
$\pi[i, j] \leftarrow \mathbf{L}$

else { $c[i, j] \leftarrow c[i-1, j]$
$\pi[i, j] \leftarrow \mathbf{U}$

return (c, π);

Finding the LCS

Algorithm: *FindLCS*(c, π, v)

$seq \leftarrow ()$
$i \leftarrow m$
$j \leftarrow n$
while $\min\{i, j\} > 0$
do
- **if** $\pi[i, j] = \mathbf{UL}$
  - **then**
    - $seq \leftarrow x_i \parallel seq$
    - $i \leftarrow i - 1$
    - $j \leftarrow j - 1$
- **else if** $\pi[i, j] = \mathbf{L}$ **then** $j \leftarrow j - 1$
- **else** $i \leftarrow i - 1$

return (seq)

Minimum Length Triangulation

Problem

Minimum Length Triangulation v1

Instance: *n points $q_1, \ldots, q_n$ in the Euclidean plane that form a convex n-gon P.*

Find: *A triangulation of P such that the sum S_c of the lengths of the $n-2$ chords is minimized.*

Problem

Minimum Length Triangulation v2

Instance: *n points $q_1, \ldots, q_n$ in the Euclidean plane that form a convex n-gon P.*

Find: *A triangulation of P such that the sum S_p of the perimeters of the $n-3$ triangles is minimized.*

Let L denote the perimeter of P. Then we have that $S_p = L + 2S_c$.
Hence the two versions have the same optimal solutions.

Problem Decomposition

We consider version 2 of the problem.

The edge $q_n q_1$ is in a triangle with a third vertex q_k, where $k \in \{2, \ldots, n-1\}$.

For a given k, we have:

1. the triangle $q_1 q_k q_n$,
2. the polygon with vertices $q_1, \ldots, q_k$,
3. the polygon with vertices $q_k, \ldots, q_n$.

The optimal solution will consist of optimal solutions to the two subproblems in 2 and 3, along with the triangle in 1.

Recurrence Relation

For $1 \leq i < j \leq n$, let $S[i, j]$ denote the optimal solution to the subproblem consisting of the polygon having vertices $q_i, \ldots, q_j$.

Let $\Delta(q_i, q_k, q_j)$ denote the perimeter of the triangle having vertices q_i, q_k, q_j.

The we have the recurrence relation

$$S[i, j] = \min\{\Delta(q_i, q_k, q_j) + S[i, k] + S[k, j] : i < k < j\}.$$

The base cases are given by

$$S[i, i + 1] = 0$$

for all i.

We compute all $S[i, j]$ with $j - i = c$, for $c = 2, 3, \ldots, n - 1$.