EE360C: Algorithms

Dynamic Programming (3/4)

Summer 2019

Department of Electrical and Computer Engineering
University of Texas at Austin

The Knapsack Problem

Knapsack Problem

- Given n objects and a “knapsack”
- Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$
- Knapsack has a capacity of W kilograms
- Goal: fill the knapsack so as to maximize the total value

Dynamic Programming: Adding a Variable

Definition: $OPT(i, w)$ is the max profit of items $1, \dots, i$ with weight limit w

Goal: $OPT(n, W)$

- Case 1: $OPT(i, w)$ does not select item i
  - $OPT\,(i, w)$ selects best of $\{1, 2, \dots, i-1\}$ using weight limit w
  - This could happen if $w_i > w$.
- Case 2: $OPT(i, w)$ selects item i
  - collect value v_i
  - new weight limit $w - w_i$
  - $OPT(i, w)$ selects best of $\{1, 2, \dots, i-1\}$ using this new weight limit

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max(OPT(i-1, w), v_i + OPT(i-1, w-w_i)) & \text{otherwise} \end{cases}$$

Pseudocode

```
function KNAPSACK(n, W, w_1, ..., w_n, v_1, ..., v_n)
    for w = 0 to W do
        M[0, w] ← 0
    end for
    for i = 1 to n do
        for w = 0 to W do
            if w_i > w then
                M[i, w] ← M[i − 1, w]
            else
                M[i, w] ← max(M[i − 1, w], v_i + M[i − 1, w − w_i])
            end if
        end for
    end for
    return M[n, W]
end function
```

Demo on Board

RNA Secondary Structure

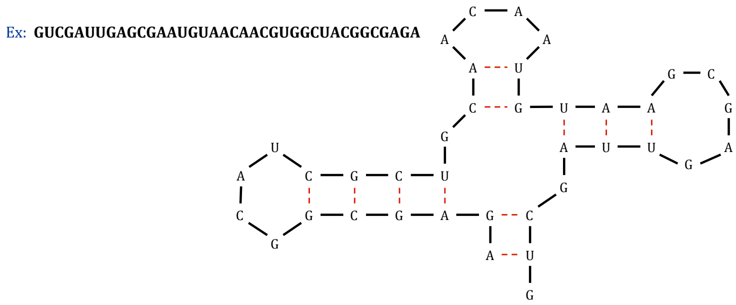
RNA Secondary Structure

RNA

String $B = b_1 b_2 \ldots b_n$ over the alphabet { A, C, G, U}

Secondary Structure

RNA is single stranded, so it tends to loop back and form *base pairs* with itself. This structure is often essential to the function of the molecule. Legal base pairs are (A, U) or (C, G).



RNA Secondary Structure

Secondary Structure

A set of pairs $S = \{(b_i, b_j)\}$ that satisfy:

- **Watson and Crick:** S is a matching and each pair in S is a Watson-Crick component (matching A to U or C to G)
- **No Sharp Turns:** The ends of each pair are separated by at least 4 intervening bases. That is, if $(b_i, b_j) \in S$, then $i < j - 4$.
- **Non-Crossing:** If (b_i, b_j) and (b_k, b_l) are two pairs in S, the we cannot have $i < k < j < l$

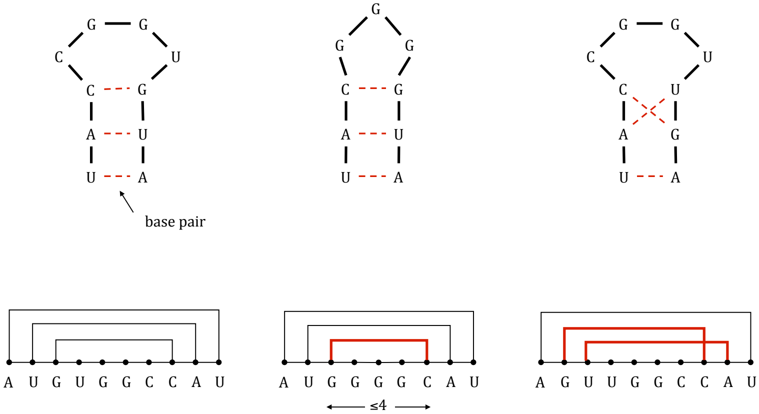
Free Energy

The usual hypothesis is that an RNA molecule will form the secondary structure that has the optimum total *free energy*, which we approximate by the number of base pairs.

Goal

Given an RNA molecule $B = b_1 b_2 \ldots b_n$, find a secondary structure S that maximizes the number of base pairs.

Examples.



ok | sharp turn | crossing

RNA Secondary Structure: subproblems

First attempt

$OPT(j)$ = maximum number of base pairs in a secondary structure of the substring $b_1 b_{i+1} \ldots b_j$

Goal

$OPT(n)$

Choice

Match bases b_t with b_j.

Difficulty

Results in two subproblems:

- Find secondary structure in $b_1 b_{i+1} \ldots b_{t-1}$
- Find secondary structure in $b_{t+1} b_{t+2} \ldots b_{j-1}$

Dynamic Programming Over Intervals

Definition

$OPT(i, j)$ is the maximum number of base pairs in a secondary structure of the substring $b_i b_{i+1} \ldots b_j$

Goal

$OPT(1, n)$

- Case 1: if $i \geq j - 4$
  - $OPT(i, j) = 0$ by the no sharp turns condition
- Case 2: Base b_j is not involved in a pair.
  - $OPT(i, j) = OPT(i, j - 1)$
- Case 3: Base b_j pairs with b_t for some $i \leq t < j - 4$
  - The non crossing constraint decouples the resulting subproblems
  - $OPT(i, j) = 1 + \max_t(OPT(i, t - 1) + OPT(t + 1, j - 1))$ such that $i \leq t < j - 4$ and (b_j, b_t) is a Watson and Crick pair

$OPT(i, j) = \max(OPT(i, j - 1), 1 + \max_t(OPT(i, t - 1) + OPT(t + 1, j - 1)))$

Bottom Up Dynamic Programming Over Intervals

Question

What order to solve the subproblems?

Answer

Do the shortest intervals first.

function RNA-SECONDARY-STRUCTURE($n, b_1 \ldots b_n$)
    for $k = 5$ to n **do**
        for $i = 1$ to $n - k$ **do**
            $j \leftarrow i + k$
            Compute $M[i, j]$
        end for
    end for
    return $M[1, n]$
end function

Dynamic Programming Summary

Recipe

- Characterize the structure of the problem
- Recursively define the value of an optimal solution
- Compute the value of the optimal solution
- Construct the optimal solution from computed information

Dynamic Programming Techniques

- Binary choice: weighted interval scheduling.
- Multi-way choice segmented least squares.
- Adding a new variable: knapsack
- Dynamic programming over intervals: RNA secondary structure

Top-down vs. Bottom-up?

Different people have different intuitions

An Exercise

The Coin Changing Problem

The Problem

You are given k denominations of coins, $d_1, d_2, \ldots d_k$ (all integers). Assume $d_1 = 1$ so it is always possible to make change for any amount of money. We want to find an algorithm that makes change for an amount of money n using as few coins as possible.

First Question

The coin changing problem exhibits optimal substructure. Consider any optimal solution to make change for n cents using our k denominations of coins. Consider breaking that solution into two different pieces along any coin boundary. Suppose that one half of the solution amounts to b cents and the other half to $n - b$ cents. Then the solution to the each half must be an optimal way to make change for b cents (or $n - b$ cents) using the same k denominations of coins. **Prove this.**

The Coin Changing Problem

The Problem

You are given k denominations of coins, $d_1, d_2, \ldots d_k$ (all integers). Assume $d_1 = 1$ so it is always possible to make change for any amount of money. We want to find an algorithm that makes change for an amount of money n using as few coins as possible.

Second Question

Let $C[p]$ be the minimum number of coins of the k denominations that sum to p cents. **Recursively define the value of the optimal solution.** To get you started, there must exist some "first coin" d_i, where $d_i \leq p$.

The Coin Changing Problem

The Problem

You are given k denominations of coins, $d_1, d_2, \ldots d_k$ (all integers). Assume $d_1 = 1$ so it is always possible to make change for any amount of money. We want to find an algorithm that makes change for an amount of money n using as few coins as possible.

The recursive definition

Let $C[p]$ be the minimum number of coins of the k denominations that sum to p cents. There must exist some "first coin" d_i, where $d_i \leq p$. The remaining coins in the optimal solution must be the optimal solution to making change for $p - d_i$ cents. Then $C[p] = 1 + C[p - d_i]$. But which coin is d_i? We don't know, so the optimal solution is the one that maximizes $C[p]$ over all choices of i such that $1 \leq i \leq k$ *and* $d_i \leq p$. Also, when $p = 0$, the optimal solution is clearly to have 0 coins:

$$C[p] = \begin{cases} 0 & \text{if } p = 0 \\ \min_{i:1 \leq i \leq k \wedge d_i \leq p}\{1 + C[p - d_i]\} & \text{if } p > 0 \end{cases}$$

The Coin Changing Problem

The Problem

You are given k denominations of coins, $d_1, d_2, \ldots d_k$ (all integers). Assume $d_1 = 1$ so it is always possible to make change for any amount of money. We want to find an algorithm that makes change for an amount of money n using as few coins as possible.

Third Question

Provide an algorithm (e.g., pseudocode) to compute the value of the optimal solution bottom up. Provide another algorithm to construct the optimal solution from the computed information.

The Coin Changing Problem

The Problem

You are given k denominations of coins, $d_1, d_2, \ldots d_k$ (all integers). Assume $d_1 = 1$ so it is always possible to make change for any amount of money. We want to find an algorithm that makes change for an amount of money n using as few coins as possible.

Solution

The Algorithm to Compute the Optimal Value

```
function CHANGE(d, k, n)
    C[0] ← 0
    for p ← 1 to n do
        min ← ∞
        for i ← 1 to k do
            if d[i] ≤ p then
                if 1 + C[p − d[i]] < min then
                    min ← 1 + C[p − d[i]]
                    coin ← i
                end if
            end if
        end for
        C[p] ← min
        S[p] ← coin
    end for
    return C and S
end function
```

The Coin Changing Problem

The Problem

You are given k denominations of coins, $d_1, d_2, \ldots d_k$ (all integers). Assume $d_1 = 1$ so it is always possible to make change for any amount of money. We want to find an algorithm that makes change for an amount of money n using as few coins as possible.

The Algorithm to Reconstruct the Solution

function MAKECHANGE(S, d, n)
    while $n > 0$ **do**
        PRINT($S[n]$)
        $n \leftarrow n - d[S[n]]$
    end while
end function

The Coin Changing Problem

The Problem

You are given k denominations of coins, $d_1, d_2, \ldots d_k$ (all integers). Assume $d_1 = 1$ so it is always possible to make change for any amount of money. We want to find an algorithm that makes change for an amount of money n using as few coins as possible.

Fourth Question

What is the running time of your algorithm?

Running Time

CHANGE is $\Theta(nk)$ (which is pseudo-polynomial in k, the input size and dependent on n, just like knapsack was dependent on W). MAKECHANGE is $O(n)$ since n is reduced by at least 1 in every iteration of the **while** loop. The total space requirement is $\Theta(n)$

Questions