COMP 3761: Algorithm Analysis and Design

Class 7

Shidong Shan

BCIT

Overview

- The design strategy: Dynamic Programming
- Difference between dynamic programming and divide-and-conquer
- Use dynamic programming to compute binomial coefficient
- Solve the knapsack problem by dynamic programming

Dynamic programming

- A general algorithm design technique
- Solving problems defined by recurrences with **overlapping** subproblems
- Invented in 1950s by Richard Bellman for optimizing multistage decision processes
- "programming" means "planning" here instead of "computer programming"
- Dynamic programming can also be used for solving non-optimization problems.

Main idea of dynamic programming

- set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
- solve smaller instances once
- record solutions in a table
- extract solution to the initial instance from the table

Main characteristics (1)

1. Overlapping subproblems
   - The subproblems of the original problem are reused for multiple times
   - Closely related to recursion
   - Dynamic programming wants to avoid solving the overlapping subproblems over and over again.

Main characteristics (2)

2. Principle of optimality
   - *Optimal substructure*: the globally optimal solution can be constructed from locally optimal solutions to subproblems.
   - An optimal solution to any instance of an optimization problem is made up of optimal solutions to its subinstances.

Divide-and-conquer vs. Dynamic Programming

- Divide-and-conquer: no overlapping subproblems
- Dynamic Programming: overlapping subproblems exist

- Divide-and-conquer: do not explicitly store solutions to smaller instances
- Dynamic programming: explicitly store solutions to smaller instances.

Example of recursive algorithm

Problem: Compute $n!$

Algorithm $recursiveFactorial(n)$
    if $n = 0$, return 1
    else return $n * recursiveFactorial(n - 1)$

- $recursiveFactorial()$ function is called exactly once for each positive integer less than n
- No overlapping subproblems.

Example: Fibonacci numbers (revisited)

- $F(n) = F(n-1) + F(n-2)$
  $F(0) = 0, F(1) = 1$
- Computing the nth Fibonacci number recursively (top-down):

$$
\begin{aligned}
&F(n) \\
&F(n-1) + F(n-2) \\
&F(n-2) + F(n-3) + F(n-3) + F(n-4) \\
&\dots
\end{aligned}
$$

- naive **top-down** recursive algorithm: $fib(n)$:
  if $n = 0$: return 0
  if $n = 1$: return 1
  return $fib(n-1) + fib(n-2)$.

Example of overlapping subproblems: $fib(5)$

Top-down recursive approach:

$$fib(5)$$
$$fib(4) + fib(3)$$
$$fib(3) + fib(2) + fib(2) + fib(1)$$
$$fib(2) + fib(1) + fib(1) + fib(0) + fib(1) + fib(0) + fib(1)$$
$$fib(1) + fib(0) + fib(1) + fib(1) + fib(0) + fib(1) + fib(0) + fib(1)$$

- Notice the massive redundancy of subproblem function calls

Dynamic Programming Approach

To compute the *n*th Fibonacci number:

- At the *k*th stage we only need to know the values of $fib(k-1)$ and $fib(k-2)$
- Compute the *n*th Fibonacci number using the **bottom-up** approach
- Record results of each iteration in a 1D array
- Remove the massive redundancy

$$F(0) = 0, \quad F(1) = 1, \quad F(2) = 1 + 0 = 1, \quad \ldots$$
$$F(n) = F(n-1) + F(n-2)$$

0	1	1	...	F(n-2)	F(n-1)	F(n)

Bottom-up Fibonacci Algorithm: *DynamicProgFib*(n)

Algorithm *DynamicProgFib*(n)

$F[0] \leftarrow 0;$

$F[1] \leftarrow 1;$

for $i \leftarrow 2..n$

$\quad F[i] = F[i-1] + F[i-2];$

return $F[n]$

Efficiency of Bottom-up Fibonacci Algorithm

- Time efficiency: $\Theta(n)$
- Space efficiency: $\Theta(n)$
- The extra array storage can be avoided by only storing the last two values

Note: Read Section 2.5 for a review of Fibonacci numbers.

Computing a binomial coefficient

- A standard example of applying dynamic programming to a **non-optimization** problem
- Binomial coefficients $C(n, k)$: the number of combinations (subsets) of k elements from an n-element set $(0 \leq k \leq n)$
- Coefficients of the binomial formula:

$$(a+b)^n = C(n,0)a^n b^0 + \ldots + C(n,k)a^{n-k}b^k + \ldots + C(n,n)a^0 b^n.$$

Properties of binomial coefficients

- Important properties:

$$C(n,k) = C(n-1,k) + C(n-1,k-1) \quad \text{for } n > k > 0$$

$$C(n,0) = 1, \quad C(n,n) = 1 \quad \text{for } n \geq 0$$

- $C(n,k)$ can be computed by smaller and overlapping subproblems
- Dynamic programming: filling a table with $n+1$ rows and $k+1$ columns.

Pseudocode and analysis

ALGORITHM *Binomial*(n, k)

//Computes $C(n, k)$ by the dynamic programming algorithm
//Input: A pair of nonnegative integers $n \geq k \geq 0$
//Output: The value of $C(n, k)$
for $i \leftarrow 0$ **to** n **do**
    for $j \leftarrow 0$ **to** $\min(i, k)$ **do**
        if $j = 0$ **or** $j = i$
            $C[i, j] \leftarrow 1$
        else $C[i, j] \leftarrow C[i-1, j-1] + C[i-1, j]$
return $C[n, k]$

- Time efficiency: $\Theta(nk)$
- Space efficiency: $\Theta(nk)$

Knapsack problem

- Given n items:
  - weights: $w_1, w_2, \dots, w_n$
  - values: $v_1, v_2, \dots, v_n$
  - a knapsack of capacity W
- Output: find most valuable subset of the items that fit into the knapsack.
- Assume that both weights and capacity are positive integers
- In reality, weights and capacity do not need to be integers.

Approaches to Knapsack problem

- Exhaustive search (Brute-force):
  1. Generate all subsets of the set of n items
  2. Compute the total weight and the total value of each subset
  3. Find the subset with the largest value.
  4. Time complexity: $\Omega(2^n)$
- Dynamic programming can be used to solve such difficult combinatorial problems more efficiently.

Knapsack problem by Dynamic Programming (1)

- Express a solution to an instance of the knapsack problem in terms of solutions to its smaller subinstances.
- Consider instance defined by first i $(1 \leq i \leq n)$ items and capacity j $(1 \leq j \leq W)$
- Let $V[i,j]$ be optimal value of this instance
- $V[i,j]$ is the value of the most valuable subset of the first i items that fit into the knapsack of capacity j.

Knapsack problem by Dynamic Programming (2)

- If $w_i > j$, then $V(i, j) = V(i-1, j)$, because we cannot include the ith item in the capacity of j.
- If $w_i \leq j$, then we have a choice: either include the ith item or do not include it.
  1. If we do **not** include item i , then the value will be $V(i-1, j)$
  2. If we do include item i , the value will be $v_i + V(i-1, j-w_i)$.
- Which choice should we make?
- By the principle of optimality, we choose the maximum value of the above two.

Recurrence relation of the Knapsack problem

- Recurrence relation

$$V[i,j] = \max\{V[i-1,j], \quad v_i + V[i-1,j-w_i]\} \quad \text{if } j - w_i \geq 0$$
$$V[i,j] = V[i-1,j] \qquad\qquad\qquad\qquad \text{if } j - w_i < 0$$

- Initial conditions:
  $V[0,j] = 0 \quad \text{and } V[i,0] = 0, \quad 0 \leq i \leq n, \quad 0 \leq j \leq W$
- Overlapping subproblems: at any stage (i,j), we may need to calculate several $V(k,l)$ for $k < i$ and $l < j$.

Knapsack problem by Dynamic Programming (3)

- Goal: to compute $V[n, W]$ and find an optimal subset.
- set up a dynamic programming table with $n+1$ rows and $W+1$ columns
- compute and record the entry $V[i, j]$ in ith row and jth column, until the table is filled.
- Time and space efficiency: $\Theta(nW)$.
- To find the composition of an optimal set, trace back the computations of $V[n, W]$ in the table.
  1. if $V[i, j] \neq V[i-1, j]$, item i is included in the optimal set
  2. if $V[i, j] = V[i-1, j]$, item i is not included in the optimal set
- Time needed to find the composition of an optimal subset is $O(n+W)$.

Example: knapsack problem by Dynamic Programming

Example: Capacity $W = 5$

item (i)	weight (w_i)	value(v_i)
1	2	12
2	1	10
3	3	20
4	2	15

Fill up the dynamic programming table and find the composition of an optimal set of the given problem.

Top-down recursive pseudocode for Knapsack problem

Algorithm *TopDownV*(i, j):

if $i = 0$ or $j = 0$, return 0

if $w[i] > j$, return $TopDownV(i-1, j)$

if $w[i] \leq j$, return

$\max\{TopDownV(i-1, j), \quad v[i] + TopDownV(i-1, j-w[i])\}$

- Solves common subproblems more than once
- not very efficient

Remarks on Dynamic Programming

- Classical dynamic programming – **bottom-up** approach:
- fills a table with solutions to **all** smaller subproblems
- each subproblem is solved only once.

- **drawback**: **not necessary** needed to have solutions to all of the subproblems.
- Improvement: Only need to solve the subproblems that are necessary and does it only once.

Memory Function

- Combines the strengths of the top-down and bottom-up approaches
- Initially, all the entries in the table are initialized with a "null" symbol
- The first time we calculate the value of $V(i,j)$, we store it in the table at the appropriate location.
- When a new value needs to be calculated, the method checks the table first, if the entry has a value, then retrieve it; otherwise, it is computed by the recursive call.

- Use a global variable table $V[0..n, 0..W]$
- All table entries are initialized with -1 except for row 0 and column 0 initialized with 0.

Pseudocode for MFKnapsack(i,j)

Algorithm $MFKnapsack(i, j)$

if $V[i, j] < 0$

  if $j < w[i]$, value $\leftarrow MFKnapsack(i-1, j)$

  else value $\leftarrow \max\{MFKnapsack(i-1, j),$

    $v[i] + MFKnapsack(i-1, j-w[i])\}$

  $V[i, j] \leftarrow$ value

return $V[i, j]$

- Time and space efficiency: $\Theta(nW)$
- Same as the bottom-up algorithm

Exercise: apply *MFKnapsack* to the previous example of the knapsack problem.

Lab Exercises

Section 8.1: #4, 7

Section 8.4: #1, 5