

CSC 301: Analysis of Algorithms

Week 11: Dynamic Programming

William Rebelsky
[rebelskyw]

Grinnell College

Monday 22nd April, 2024

Table of Contents

- 1 Monday: Dynamic Programming
 - Dynamic Programming Basics
 - Rod Cutting
 - Practice
- 2 Wednesday: Dynamic Programming
 - Matrix Multiplication
 - Longest Common Subsequence
 - Optimal Binary Search Tree
- 3 Friday: Discussion
 - Practice

CSC 301: Analysis of Algorithms

Class 34: Intro to Dynamic Programming

William Rebelsky
[rebelskyw]

Grinnell College

Monday 22nd April, 2024

Reminders

- Homework 8 is due Wednesday
- Quiz 8 is due tonight
- Token opportunities: Please fill out this form:
https://grinnell.co1.qualtrics.com/jfe/form/SV_2mhIpFACg0FHYDc. If 75% of the class fills it out: 1 token, if 95% of the class fills it out: 2 tokens for everyone
- Schedule:
 - This week: Dynamic Programming, chapter 14, please read the chapter before Wednesday
 - Next week:
 - Monday: Final Project description, time to ask questions and work on it in class
 - Wednesday: No class, please work on the final project
 - Friday: More practice with topics based on the survey released this Wednesday
- Apply to be a Peer Educator: Deadline extended to Thursday

Dynamic Programming Basics

- Many ideas from other parts of class
- Like divide and conquer: solve problems by combining solutions to subproblems
- “Programming” refers to the tabular method not to writing code
- Applies when subproblems overlap (i.e. optimal substructure property)
- More efficient than divide and conquer since we don’t need to keep solving the same problems
- Optimization problems: many possible solutions, goal is min or max solution
- Solution is “an” optimal solution not “the” optimal solution
- Time-memory tradeoff: Storing values costs memory but may have significant time savings

Dynamic Programming Steps

- ➊ Characterize the structure of an optimal solution
 - ➋ Recursively define the value of an optimal solution
 - ➌ Compute the value of an optimal solution, typically in a bottom-up fashion
 - ➍ Construct an optimal solution from computed information
- Step 4 is not required if you just need the value of the optimal solution not the solution itself
 - Generally you maintain additional information during step 3 to make step 4 easier, e.g. a table with the choice made in each step/subproblem

Solving Dynamic Programming Problems

There are two main approaches with the same asymptotic runtimes

- ❶ A top down approach with memoization:
 - Write the recursive procedure as usual (Divide/conquer)
 - Save values as you compute them
 - Before recursing, check if you've solved the problem
 - Memoized algorithm: “remembers” prior computed results
 - Similar to a reverse topological sort
 - Faster if many subproblems don't need to be solved since they will never be computed
- ❷ A bottom up approach:
 - Requires that solving a subproblem relies only on solving smaller subproblems (there has to be a natural “size”)
 - Solve the smallest subproblems and work your way up
 - Similar to BFS
 - Faster in general since no recursive overhead and less overhead maintaining the table

Dynamic Programming Requirements

- Optimal substructure property
- Subproblem solutions are independent, i.e. when you make a choice to split the problem into subproblems, the solutions to the subproblems can't rely on each other

Dynamic Programming

- Significantly more efficient than Divide and Conquer when usable (exponential to polynomial)
- Runtime and space requirements are based on the number of subproblems an optimal solution uses and how many choices you have in determining which subproblems to use
 - Rod cutting: $\Theta(n)$ subproblems with at most n choices is $\Theta(n^2)$
 - Optimal Matrix Multiplication: n^2 subproblems with $n-1$ choices is $\Theta(n^3)$
- You can also generally figure out the runtime using a subproblem graph: Each subproblem is a vertex with edges pointing to subproblems it relies on, see board

Things to watch for

- The optimal substructure property does not always apply
 - Unweighted shortest path: (optimal substructure)
 - Unweighted longest simple path: (See square graph)
- If the optimal substructure property does apply, a greedy solution may be more efficient

Rod Cutting Problem

- Sterling Steel Company buys long steel rods, cuts them into shorter rods, and sells them
- Each cut is free and removes no material
- Given a table giving the price they can charge for a rod of length i (integer cuts only) e.g. 1,5,6,9,10,17,17,20,24,30
- Given a rod of length n and a table of prices $1:n$, determine the max revenue obtainable by cutting the rods

Rod Cutting Example

- Rod size 4
- There are 8 ways to cut it (in general 2^{n-1} ways)
- See board

Rod Cutting

- This exhibits the optimal substructure property (optimal solution to size n is the combination of the optimal solutions to the two sides of the cut)
- Simpler rearrangement: Think of the problem as a cut that separates a piece that is sold as is and a remainder that can be further divided.
- $r_n = \max\{p_i + r_{n-i} : 1 \leq i \leq n\}$
- Recursive implementation is very inefficient ($O(2^n)$)

Rod Cutting 2

- Memoized Top down approach:
 - Initialize new array to $-\infty$
 - Recursive function:
 - ① If you have a solution, return it
 - ② If $n=0$, value=0
 - ③ else compare all possible solutions recursively
 - ④ store the value in the array[n]
- Bottom up approach:
 - Initialize new array 0:n
 - Base case: $r[0]=0$
 - for $j=1:n$, initial value, q , is $-\infty$
 - for $i=1:j$, $q=\max(q, p[i]+r[j-i])$
 - $r[j]=q$

Stamp Problem

- Given Stamps with the following denominations: 1, 5, 7 determine the optimal set of stamps to pay for n cents of postage. As we saw in the greedy homework, this cannot be solved greedily.

Practice Problems

- 14.1-2 Show that the following greedy strategy does not work for the rod cutting problem: define the density of a rod of length i as p_i/i i.e. it's value per inch, in each step choose the densest rod possible.
- 14.1-3 Consider a modification to the rod-cutting problem where each cut now costs c instead of 0. Give a dynamic programming solution to this new problem.
- 14.1-6 Give an $O(n)$ time dynamic programming solution to compute the n th Fibonacci number and draw the subproblem graph. How many vertices and edges does the graph contain?

Table of Contents

- 1 Monday: Dynamic Programming
 - Dynamic Programming Basics
 - Rod Cutting
 - Practice
- 2 Wednesday: Dynamic Programming
 - Matrix Multiplication
 - Longest Common Subsequence
 - Optimal Binary Search Tree
- 3 Friday: Discussion
 - Practice

CSC 301: Analysis of Algorithms

Class 35: Dynamic Programming

William Rebelsky
[rebelskyw]

Grinnell College

Wednesday 24th April, 2024

Reminders

- Homework 8 is due Tonight
- Homework 9 is released and due next Wednesday
- Quiz 9 will be released Friday
- Fill out both surveys

Matrix Multiplication

Matrix-chain Multiplication

- Given a chain of n matrices to be multiplied, where the matrices may not be square, the goal is to compute the product efficiently
- Matrix Multiplication is associative, that is $(AB)C=A(BC)$
- Fully parenthesized: it is clear what order the multiplications happen
- Runtime is dominated by the number of scalar multiplications
- Exhaustively listing all possible parenthesizations is highly inefficient ($\Omega(2^n)$)

Matrix Multiplication Dynamic Programming

- Step 1) Structure of an optimal parenthesization: Let $A_{i:j}$ be the multiplication of matrices i through j . Splitting at matrix k gives $A_{i:j} = A_{i:k} + A_{k+1:j}$ plus the cost of multiplying them together
- Step 2) Recursive solution: $m[i,j]=0$ if $i=j$, else $\min\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\}$ where p is the input dimension
- Step 3) Fill the table: bottom up approach along the diagonal (See board)
- Step 4) Constructing an optimal solution: record the value of k in a table s

Matrix Multiplication

Matrix Multiplication Practice

- 14.2-1 Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is 5,10,3,12,5,50,6
- 14.3-4 Show greedy choice fails in Matrix multiplication

Longest Common Subsequence

Longest Common Subsequence

- Given two strings, find the longest common subsequence
- Subsequence: a string, Z , such that there exists a strictly increasing sequence of indices of X that is equal to Z
- Common subsequence: shared by both X and Y

Longest Common Subsequence Dynamic Programming

- Step 1) Characterizing a longest common subsequence: Let X and Y be sequences. Let Z be any LCS of X and Y . Then:
 - ① if $x_m = y_n$ then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}
 - ② if $x_m \neq y_n$ and $z_k \neq x_m$ then Z is an LCS of X_{m-1} and Y
 - ③ if $x_m \neq y_n$ and $z_k \neq y_n$ then Z is an LCS of X and Y_{n-1}
- Step 2) Recursive solution: $c[i,j]=$
 - 0 if i or $j=0$
 - $c[i-1,j-1]+1$ if $i,j > 0$ and $x_i = y_j$
 - $\max(c[i,j-1], c[i-1,j])$ if $i,j > 0$ and $x_i \neq y_j$
- Step 3) Fill the table: Bottom up in row major order
- Step 4) Constructing an optimal solution: Naive solution: store choice in each cell. Better solution: don't store the choice, only 3 possibilities at each step, constant time

Longest Common Subsequence Practice

- 14.4-1 Determine an LCS of $(1,0,0,1,0,1,0,1)$ and $(0,1,0,1,1,0,1,1,0)$

Optimal BST

- Given a sequence of distinct keys, K , build a search tree containing them
- For each key, k , you have the probability that a given search is for that key
- For each key, you have a dummy variable, d , representing the values not in K
- Knowing these values, we can determine the expected cost of a search in a given BST
- Goal: construct a BST whose expected search cost is smallest

Optimal BST Dynamic Programming

- Step 1) Structure of an optimal BST: Any subtree of a BST has a contiguous set of keys and has the dummy keys as leaves
- Step 2) Recursive solution: q is the probability assigned to the corresponding dummy variable, e is the expected search cost in an optimal bst. Let r be the root of a binary tree from i to j . Let $w(i,j)$ be the sum of all probabilities in the subtree from i to j . $e[i,j]=$
 - q_{i-1} if $j = i - 1$
 - $\min(e[i,r-1]+e[r+1,j]+w(i,j))$: $i \leq r \leq j$ if $i \leq j$
- Step 3) Fill the table: As with Matrix Multiplication, go along the diagonal
- Step 4) Constructing an optimal solution: As with Matrix Multiplication store the choice of r along the way

Optimal BST Practice

- 14.5-2 Determine the cost and structure of an optimal binary search tree for a set of $n=7$ keys with the following probabilities:
- $p_i = \{-, 0.04, 0.06, 0.08, 0.02, 0.1, 0.12, 0.14\}$
- $q_i = \{0.06, 0.06, 0.06, 0.06, 0.05, 0.05, 0.05, 0.05\}$

Table of Contents

- 1 Monday: Dynamic Programming
 - Dynamic Programming Basics
 - Rod Cutting
 - Practice

- 2 Wednesday: Dynamic Programming
 - Matrix Multiplication
 - Longest Common Subsequence
 - Optimal Binary Search Tree

- 3 Friday: Discussion
 - Practice

CSC 301: Analysis of Algorithms

Class 36: Discussion

William Rebelsky
[rebelskyw]

Grinnell College

Friday 26th April, 2024

Reminders

- Homework 9 is due Wednesday
- Quiz 9 is released
- Fill out both surveys
- Please turn in any redos soon so that I can grade them in a reasonable time

Discussion Problem

- Give an $O(n^2)$ time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers
- Test your solution on the sequence 50 30 0 30 30 40 20

HW

- 14.3-3 Consider the antithetical variant of the matrix-chain multiplication problem where the goal is to maximize the number of scalar multiplications. Does this problem exhibit optimal substructure?
- 14.3-5 Suppose that the rod-cutting problem also has a limit on the number of pieces of length i allowed to be produced. Show that the optimal-substructure property no longer holds.
- 14-1 Given a directed acyclic graph with real valued edge weights, find the longest weighted simple path from s to t .
- 14-2 Longest Palindrome subsequence: Given a string, find the longest palindrome subsequence
- 14.4-6: Give an $O(n \lg n)$ time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers
- No bonus

Discussion Problem Solution Sketch: Opt Substructure

- Optimal substructure: Normally we start with a solution to the original problem and show that removing a part of the solution leaves a solution to a subproblem or there is a contradiction. Here we will show the opposite. Let $A[1:n]$ be our initial array. Let $S[1:m]$ be an optimal solution to $A[1:n-1]$. Then the optimal solution to $A[1:n]$ is either $S[1:m]+A[n]$ (if $A[n] \geq S[m]$), or $S[1:m]$ otherwise. If it was not an optimal solution to $A[1:n]$ then that means there is another array S' of length either $m+2$ ($A[n] \geq S[m]$) or length $m+1$ (otherwise) that is a monotonically increasing subsequence of $A[n]$. But that would mean that our original solution S is non optimal since that would mean S' contains a subsequence of length $m+1$ that is a monotonically increasing subsequence of $A[1:n-1]$ which is a contradiction.

Discussion Problem Solution Sketch: Filling the table

- Fill a $1 \times n$ table of the value of the longest subsequence that ends in $A[i]$, That is $L[i] = \max\{1 + L[j]\}$ for all choices of j such that $j \leq i$ and $A[j] \leq A[i]$. (you don't need to explicitly have the "or 1 if all prior values are greater" if you initialize L to 0). You should also keep track of a table S such that $S[i]$ is the value j chosen in the computation for $L[i]$
- Given 50, 30, 0, 30, 30, 40, 20
- L : 1, 1, 1, 2, 3, 4, 2
- S : -, -, -, 2 or 3, 4, 5, 3
- Max L ends at 40 with value 4, tracing back through S we get 30,30,30,40 or 0,30,30,40