1 Homework 3 Due April 29, 9:20 in canvas Disclose Internet Sources

Basic Dynamic Programming

Homework 3.01 (10 points)

Our function for the cost to multiply two matrices $M_1 \, * \, M2$ is $cost(c_0, \, c_1, \, c2) = c_0 \, c_1 \, c_2)$, where M_1 has c_0 rows and c_1 columns, and M_2 has c_1 rows and c_2 columns. This quantity is the number of element-wise multiplications performed by the standard matrix multiplication algorithm. It can also be thought of as performing $c_0 \, c_2$ inner products, since the output matrix has c_0 rows and c_2 columns.

Construct an example with four matrices where the multiplication order with the worst total cost is at least 100 times as large as the cost of the best order. For full credit, the greedy heuristic of lowest-cost multiplication first should not produce the optimal order.

Homework 3.02 (10 points)

Suppose the denominations of the coins in a country are $c_1 > c_2 > \ldots > c_n$ (e.g., we use 50, 25, 10, 5, 1 for the United States even though one-dollar coins exist). The *coin changing problem* is: Given a sequence of coin denominations and an amount, a cents, as input, determine the minimum number of coins needed to make a cents in change. (You may assume that $c_n = 1$, so that it is always possible to make change for any amount a.)

1. Describe a greedy algorithm for this problem. Explain how it would work to make change for \$1.43 (U.S.).
2. Explain why your greedy algorithm works for U.S. coins; that is, it will make the change using the minimum possible number of coins.
3. Make up an example of denominations for a fictitious country's coin system where your greedy algorithm will not give the minimum number of coins for certain amounts, and show a case where it fails.
4. Give a dynamic programming algorithm to solve the problem. Analyze the worst-case running time and space requirements of your algorithm as functions of n and a.

Homework 3.03 (20 points)

Consider the problem of dividing a sequence of words that forms a paragraph into lines for printing, like Exercise 6 on pages 317–318 of Kleinberg and Tardos, with a few differences.

As in the text, we assume that each line has L positions, and each position can hold one letter or one space. Words on the same line must be separated by at least one space. The input is a sequence of word lengths w_1 through w_n.

The variations from the text for this homework problem are:

1. The penalty for x extra spaces on a line is x^3 instead of x^2;
2. There is *no penalty* for extra spaces on the *last* line of the whole paragraph;
3. The x extra spaces, if any, are distributed evenly between words so that the last word on the line ends exactly at position L (uness there is only one word on that line). In other words, the lines are both left and right justified, except that the last line is only left justified.

If words i through j are placed on the same line, then

$$\sum_{m=i}^{j} (w_m + 1) \quad \leq \quad L + 1$$

is required. Suppose your format places words 1 through n on Z lines, numbered 1 through Z, and for $1 \leq k \leq Z - 1$, x_k extra spaces are left on line k. Then the total penalty to be minimized is

$$P \quad = \quad \sum_{k=1}^{Z-1} (x_k^3).$$

Note that different line breaks can produce different values for Z on the same input.

Microsoft word uses a greedy line-breaking strategy that consists of putting as many words as possible on each line in sequence, without going past position L, until the paragraph is completed.

Show (by counter-example) that this greedy algorithm for line breaking does not produce the minimum total penalty P in all cases.

Then formulate a recursive algorithm that computes an optimal way to break the given input sequence $w_1, \ldots, w_n$ into lines. The recursive function should return the **value** of the minimum penalty for its subproblem.

Your pseudo-code does **not** need to spell out how the actual solution is stored or output, but a useful implementation would need to do this. The solution might be stored an array E such that $E[k]$ tells the *last* word to go on line k; $E[0]$ would be 0; $E[Z]$ would be n, where Z would be found by the algorithm.

The primary design issue is how to break the problem into subproblems so that the subproblem graph is small and the correct minimum penalty for a problem can be computed once the minimum penalties of its child subproblems have been found.

Consider various ways that subproblems can be specified using one or more integers. For your chosen design, how many subproblems are there for a paragraph of n words? How many children can a subproblem have? These are edges in the subproblem graph. Your answers should be ranges because exact values depend on the words lengths in a particular input.

For your recursive algorithm **without** using `store` and `retrieve` for solved subproblems, Give (and justify) a lower bound on its asymptotic growth rate (i.e., big Ω) that is exponential in n. Treat L as a constant and n as the parameter. You might want to look ahead to the Asymptotic Analysis problems before working on this.

For your dynamic programming version **with** `store` and `retrieve` for solved subproblems, Give (and justify) an upper bound on its asymptotic growth rate (i.e., big O). For full credit it should be as tight as possible. Treat L as a constant and n as the parameter.

Asymptotic Analysis

For all recurrences, assume the base cases all cost 1.

Homework 3.04 (10 points)

Define $T_2(n) \ = \ 2\,T_2(n-2) \qquad$ for $n \geq 2$.

Argue that $T_2(n) \in \Theta(2^{n/2})$.

Is this the same as $\Theta(2^n)$? Why or why not?

Homework 3.05 (10 points)

Let $k \geq 2$ be some constant. Define

$$T(n) \quad = \quad T(n-1) + T(n-2) + \cdots + T(n-k) \qquad \text{for } n \geq k.$$

Give (and justify) an easy asymptotic **lower bound** for $T(n)$ (i.e., big Ω). It need not be as tight as possible, but it should be exponential in n. You may use the result of problem 3.04 even if you did not solve it. *Hint*: Derive a recursive **inequality** for $T(n)$ that allows you to exploit problem 3.04.

Is your lower bound the same as $\Omega(2^n)$? Why or why not?

Advanced Dynamic Programming – Do What You Can Correctly

Homework 3.06 (20 points)

Let E be an array of n distinct integers. Give an algorithm to find the length of a **longest increasing subsequence** of entries in E. The subsequence is not required to be contiguous in the original sequence. For example, if the entries are 11, 17, 5, 8, 6, 4, 7, 12, 3, a longest increasing subsequence is 5, 6, 7, 12.

First describe the recursive procedure and justify its correctness, being **very careful** about how subproblems are specified. Be **forewarned** that a PhD student came up with a buggy solution in a previous year.

Execute your procedure on numerous examples to try to expose bugs. Turn in only **one** such example that you consider a severe test.

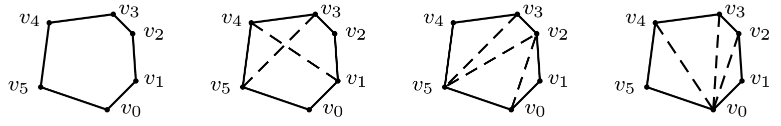
Then explain how to modify it with `store` and `retrieve`, to create a dynamic programming algorithm. Give (and justify) the worst-case asymptotic order with these modifications in place. Assume the integers in E are small enough to do arithmetic in $O(1)$.

Give (and justify) an exponential worst-case running time for the recursive version without the `store` and `retrieve` that you added to make the dynamic programming version. Use techniques from problems 3.04 and 3.05 to easily get a suitable bound.

Homework 3.07 (20 points)

An n-sided *polygon* is an undirected graph with n vertices and n edges that form a simple cycle, v_0, v_1, …, v_{n-1}, v_0. (It is conventional to index polygon vertices from 0.) A *chord* of a polygon is an edge (undirected) between any two nonadjacent vertices of the polygon. Two distinct chords, say wx and yz, are *nonintersecting* if there is a path of polygon edges from w to x that does not contain either y or z as an intermediate vertex. If two chords share exactly one vertex, they are nonintersecting. A *triangulation* of a polygon is a maximal set of mutually nonintersecting chords. A *triangulated polygon* is the graph consisting of the original polygon and a set of chords that comprise a triangulation of it.

The definitions are motivated by thinking of the vertices of the polygon as being positioned in counterclockwise sequence around a circle; however, this positioning is not required by the definitions. The figure below shows a polygon, an example of intersecting chords, then two possible triangulations.


v_4 v_3 v_2 v_1 v_5 v_0
v_4 v_3 v_2 v_1 v_5 v_0
v_4 v_3 v_2 v_1 v_5 v_0
v_4 v_3 v_2 v_1 v_5 v_0

Suppose floating-point weights are associated with all the possible chords and are stored in a matrix so they can be looked up in $O(1)$ each. For example, if the polygon vertices are points in space, the weight of a chord might be the distance between its two vertices, but other weighting schemes are possible, also.

The goal for this problem is: Given a polygon of n vertices, named and indexed as described above, and given a set of weights for its $n(n-3)/2$ chords as input, find a *minimum-weight triangulation*, that is, one whose sum of chord weights is minimized.

1. Use the definitions of "nonintersecting" and "triangulation" to show that each polygon edge is in exactly one triangle of a triangulated polygon.
2. Describe a plausible greedy algorithm for polygon triangulation. Analyze its worst-case running time as a function of n. Show an example polygon and set of chord weights for which your greedy algorithm does not produce the minimum-weight triangulation.
3. Design a recursive algorithm to solve the minimum-weight polygon triangulation problem (without using `store` and `retrieve` — that comes in part 4). Be specific about the identifiers and definitions for subproblems. Show that its worst-case running time is exponential in n. (Exact analysis is not necessary; show an exponential lower bound, using techniques in earlier problems.)
4. Convert your recursive algorithm into its dynamic programming version by appropriately inserting `retrieve` and `store`. Analyze the asymptotic order of its worst-case running time as a function of n.

Disclose Internet Sources

These are all very standard dynamic programming problems. Copying solutions from the Internet is not allowed. **Looking at** solutions on the Internet is OK, but if you do, you must include the full URLs of all sources you look at in your answers. Failure to disclose such sources is academic misconduct and warnings in the syllabus will apply.

In any case your answers should pay attention to the details of what is asked in the homework. Straight copying from the Internet is unlikely to cover these details, and will likely get little credit.