CPSC 320 2016W1: Assignment 4

2016-10-20 Thu

Submit this assignment via handin (see the syllabus for more information) to the target `assn4` by the deadline **Thursday 10 Nov at 10PM**. For credit, your group must make a **single** submission via one group member's account. Your group's submission **must**:

- Be on time.
- Consist of a single, clearly legible PDF file named `solution.pdf` with clearly indicated solutions to the problems. (Directly produced via LaTeX, Word, Google Docs, or other editing software is best. Scanned is fine. **High-quality** photographs are OK if we agree they're legible.)
- Include prominent numbering that corresponds to the numbering used in this assignment handout (not the individual quiz postings). Put these **in order**, ideally. If not, **very clearly** and prominently indicate which problem is answered where!
- Include at the start of the document the names and ugrad.cs.ubc.ca account IDs of each member of your team.
- Include at the start of the document the statement: "All group members have read and followed the guidelines for academic conduct in CPSC 320. As part of those rules, when collaborating with anyone outside my group, (1) I and my collaborators took no record but names away, and (2) after a suitable break, my group created the assignment I am submitting without help from anyone other than the course staff." (Go read those guidelines!)
- Include at the start of the document acknowledgment of collaborators and references (with the exceptions listed in the conduct guidelines).

1 The Price Isn't Always Right

Aconcagua.com sells storage on both an auction and fixed-price basis. They want to use historical auction data to investigate their fixed price choices.

For n seconds, they have the price point reached in each second in their auctions. They want to find the largest price-over-time stretch in their data. That is, given an array A of n price points, they want to find the largest possible value of $f(i, d) = d * \min(A[i], A[i+1], \ldots, A[i+d-1], A[i+d])$ where i is the index of the left end of a stretch of seconds, d is the duration in seconds of that stretch, and the function f computes the duration times the minimum price over that period. (Prices are positive, $d \geq 0$, and for all values of i, $f(i, 0) = 0$ and $f(i, 1) = A[i]$.)

For example, the best stretch is underlined in the following price array: $[8, 2, \underline{9, 5, 6, 5}, 3, 1]$. Using 1-based indexing, the value for this optimal stretch starting at index 3 and running for 4 seconds is $f(3, 4) = 4 * \min(9, 5, 6, 5) = 4 * 5 = 20$.

1. Give a **brute force** algorithm to solve this problem. Your algorithm must run in polynomial time.
2. Give and briefly justify a good asymptotic bound on the runtime of your algorithm.

1.1 Divide-and-Conqaguar

In this part, you will give a divide-and-conquer approach that is more efficient than the brute force approach.

1. Consider again the example array from the previous part: $[8, 2, 9, 5, 6, 5, 3, 1]$. Imagine that you are told that the solution **must** use the elements at indexes 4 and 5 (i.e., elements 5 and 6). You want to expand the stretch either to the left or right one step at a time while maintaining the invariant that the stretch you have chosen is the best that includes indexes 4 and 5 until the stretch includes the entire array.

   Finish the following table describing this expansion, thinking carefully about why you make the choice you do at each point:

i	d	minimum	$f(i, d)$
4	2	5	10
3	3	5	15
3	4	5	20
...	...	...	...
1	8	1	8

2. Give an efficient algorithm for finding the optimal price stretch.

3. Give and briefly justify a good asymptotic bound on the runtime of your algorithm.

2 CS CSI

The major credit card company DoctorCard can either check a transaction to see if it's fraudulent or skip it. Each transaction comes with an estimated value for fraud checking (based on the size and suspiciousness of the transaction). Skipping a transaction provides no value. Checking if it's fraudulent provides the estimated value. However, because of computational costs, DoctorCard **must** skip checking the next two transactions if they check the current one.

We're interested in maximizing the total fraud value checked given such an array of estimated values A.

So, for example, the optimal transactions to check are underlined in the following transaction record: $[4, \underline{9}, 1, 10, 1, \underline{7}, 2, 2]$. Note that each checked (underlined) transaction must be followed by at least two unchecked transactions and that therefore the last two entries must always be unchecked.

Design an efficient algorithm to find the maximum total fraud value. Your algorithm need not find the actual transactions to check, just the value.[1] Your algorithm may use linear memory and time.

HINT: Start by designing a recurrence $F(n)$ that describes the optimal fraud value achievable for transactions $1, \ldots, n$ in terms of the optimal value(s) for smaller stretches of the array. Then, either convert that into a recursive solution and memoize it or convert that into a dynamic programming solution.

2.1 CSI Finds the Culprit

Adapt your original algorithm so that it produces the optimal set of indexes to check for fraud rather than the maximum total value.

2.2 Forgetful CSI

Instead adapt your original algorithm so that it produces only the maximum total value but does it using constant memory.

[1] If you want a justification of that: DoctorCard is comparing the value achieved by their online algorithm—i.e., algorithm that makes choices as transactions arrive—against the optimal total achievable.

3 Hitting Rock Bottom

Finding the global minimum—the single smallest element—in an unsorted array of numbers requires $\Omega(n)$ time. A **local** minimum is a number that is smaller than its neighbour(s). (For $n > 1$, the first and last elements have only one neighbour each and so it's "easier" for them to be local minima.) A local minimum can be found more quickly.

For example, the three local minima in the following array are underlined: $[8, \underline{2}, 3, 7, 9, \underline{5}, 6, 12, \underline{10}]$.

1. Give an efficient algorithm to find a **local** minimum of an array of n distinct numbers. (You may assume $n > 1$ if needed.) **HINT:** try to quickly find a portion of the larger array in which there's guaranteed to be at least one local minimum.

2. Give and briefly justify a good asymptotic bound on the runtime of your algorithm.

3. Sketch the key points in a proof that your algorithm is correct. (You may find it useful to establish and take advantage of this condition: "the leftmost and rightmost elements of the portion of A under consideration are smaller than any neighbours they have outside the portion of A under consideration" or to put it another way "I can check if the leftmost and rightmost elements are local minima without having to consider anything outside the portion of A under consideration".)

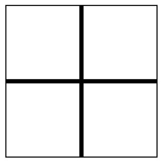
3.1 Rock Bottom Tiles

Now, consider an $n \times n$ 2-dimensional array of distinct numbers. A local minimum is still a number smaller than its neighbour(s); however, an element now neighbours the four elements above, below, to the left, and to the right. (On the edges and corners, an element has fewer neighbours and so it's "easier" for it to be a local minimum.)

1. Give an example of such a 2-dimensional array with $n = 5$ and circle all the local minima.

2. Give an efficient algorithm to find a **local** minimum in such a 2-dimensional array. (You may assume $n > 1$ if needed.)

   HINT: Consider the set of array elements down the two middle lines of the array, in this shape:

   The smallest of these elements tells you something about where a local minimum might be. It will likely help to compare against your example!

3. Give and briefly justify a good asymptotic bound on the runtime of your algorithm.

4. Sketch the key points in a proof that your algorithm is correct. (It may help to note that in the inital array of n distinct numbers, there is always at least one local minumum because the global minimum is also guaranteed to be a local minimum.)

4 No Chutes, Just Ladders

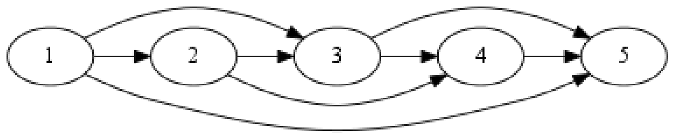
A children's game has n spaces numbered $1, \ldots, n$. A game piece can progress from one space to the next or—in certain spaces marked with ladders—it can "jump" forward a fixed number of spaces.

You are given an array A of the spaces on the board, where each entry of A is the set of spaces from which a piece can arrive at that space. So, using 1-based indexing, $A[1] = \{\}$ because pieces start at space

1 but cannot move to there from anywhere. For every other index i of a space, $i - 1 \in A[i]$ because we can always arrive at i from space $i - 1$. To illustrate ladders, if space j had a ladder of length 3 and a ladder of length 7 leading to it, its set would be $A[j] = \{j - 1, j - 3, j - 7\}$.

Your goal is to count how many different ways there are to arrive at the final space.

For example, consider this gameboard first as a graph:



and then as an array: $[\{\}, \{1\}, \{1, 2\}, \{2, 3\}, \{1, 3, 4\}]$.

There is only 1 way to reach node 2 (from node 1). There are 2 ways to reach node 3 (via 2 or directly from 1). There are 3 ways to reach node 4 (via nodes 2 and 3, via just node 2, or via just node 3). There are 6 ways to reach the final space, node 5.

Design an efficient algorithm to count how many different ways there are to arrive at the final space. Your algorithm may use linear time and "linear" memory, where we assume that the count of ways to reach a node can be stored in constant space.

HINT: Start by designing a recurrence $C(n)$ that describes the number of ways to reach space n in terms of the number of ways to reach previous spaces on the board and in terms of $A[n]$ (the set of spaces from which space n is reachable). Then, either convert that into a recursive solution and memoize it or convert that into a dynamic programming solution.