CS 170: Efficient Algorithms and Intractable Problems

Spring 2017

•

Homework 6

Due on March 14 at 11:59pm

•

Solutions by

Michael Fan

26697596

In collaboration with

Cedric Nixon, Julie Han, Alex Gao

Problem 1: Hacking for Justice (shortcut question)

★★★★★ **Level**

In an alternate universe, the students of CS170 found a certain problem on HW6 to be extremely difficult. Initially, no one was able to find the solution. However, some subset of the students managed to download the solution PDF to their laptops. These students began to send the PDF to others via email, who then sent the PDF to others, and so on. Eventually, all of the students had the solution PDF. Uh oh!

After much effort, a TA has figured out the full history of the solution PDF sharing, and constructed a directed acyclic graph G = (V,E) to represent it. V represents the students, and an edge E from v1 to v2 represents that v1 sent the solution file to v2. All sharing is one-way, and you know that there are no cycles. If the TA could go back in time, and completely block off all communications to/from one student?s laptop, which student should be blocked to minimize the number of students who received the PDF? Assume that blocking one person will not cause anyone else to share with more people than they did before. Answer this question in part (c), or try (a) and (b) for inspiration.

Solution This problem is trashed.

Problem 2: Bridge Hop

★★★★ **Level**

You notice a bridge constructed of a single row of planks. Originally there had been n planks; unfortunately, some of them are now missing, and you're no longer sure if you can make it to the other side. For convenience, you define an array V [1..n] so that V [i] = TRUE iff the ith plank is present. You're at one side of the bridge, standing still; in other words, your hop length is 0 planks. Your bridge-hopping skills are as follows: with each hop, you can increase or decrease your hop length by 1, or keep it constant.

For example, the image above has planks at indices [1, 2, 4, 7, 8, 9, 10, 12], and you could get to the other side with the following hops: [0, 1, 2, 4, 7, 10, 12, 14].

Clarifications: You start at location 0, just before the first plank. Arriving at any location greater than n means you've successfully crossed. Due to your winged shoes, there is no maximum hop length. But you can only hop forward (hop length cannot be negative).

Devise an efficient algorithm to determine whether or not you can make it to the other side.

For this problem, you should know how to do the proof of correctness, but need not include it in your submission. You should submit the main idea, pseudocode, and runtime.

Solution

Main Idea:

Starting from spot 0, see which planks you can get to by taking a number of steps s between one fewer and one larger than the step size that was taken from a previous spot to the current plank. Enqueue this position and possible step sizes from this spot and recurse. Memoize this tuple in a set as to avoid repeated calculations. If you ever reach the nth plank or a position beyond n where n is the length of the bridge, then there is a viable path. Otherwise if all tuples are dequeued and we never reach a position $\geq n$, then return false. The recurrence relation is $planksreachablefromx = bridge[x + s]$ where s is the step size taken from x's predecessor to get to x and bridge is an array of booleans denoting the existence of a plank at position i in the array.

Pseudocode:

```
function determineValidPath(arr bridge):
    positionStepQueue = new queue
    solvedPositionSteps = new Set()
    enqueue (0, 1) positionStepQueue //starting at position 0, take steps of size 1 until you are stuck
    put (0, 1) into solvedPositionSteps
    while positionStepQueue is not empty:
        psTuple = positionStepQueue.dequeue
        startPos, stepSize = pTuple
        while bridge[startPos] is true
            if startPos ≥ bridge.length:
                return true
            if (startPos, stepSize) is not in solvedPositionSteps:
                enqueue (startPos, stepSize) into positionStepQueue
                put (startPos, stepSize) in solvedPositionSteps
            if stepSize-1 > 0 and (startPos, stepSize-1) is not in solvedPositionSteps:
                enqueue (startPos-1, stepSize) into positionStepQueue
                put (startPos-1, stepSize) in solvedPositionSteps
            if (startPos+1, stepSize) is not in solvedPositionSteps:
                enqueue (startPos+1, stepSize) into positionStepQueue
                put (startPos+1, stepSize) in solvedPositionSteps
            startPos += stepSize
    return false
```

Runtime Analysis:

Solution (cont.)

We memoize all the position-stepsize tuples so that we never recalculate the possible paths taken from any unique position. Thus, there is a soft upper bound of k tuples for position k on the bridge, one for each possible step size up to that spot if all planks are filled in. In the worst case, this is $O(n^2)$.

Problem 3: Longest Palindrome Substring

★★★★ **Level**

A substring is palindromic if it is the same whether read left to right or right to left. For example, "bob" and "racecar" are palindromes, but "cat" is not. Devise an algorithm that takes a sequence $x[1..n]$ and returns the length of the longest palindromic substring. Its running time should be $O(n^2)$.

For this problem, you should know how to do the proof of correctness, but need not include it in your submission. You should submit the main idea, pseudocode, and runtime.

Solution

Main Idea:

Let the recurring subproblem be $P(a,l) = max(P(a,l+1)+2, P(a+1,l), P(a-1,l))$ where $P(a,l)$ denotes the length of the possible palindrome centered at a and spanning out l in either direction. The base cases are $P(a,1) = 1$ where a is any integer index, and $P(b,1) = 0$ or 2 where $b \in \{0.5, 1.5, ...k-0.5\}$ where k is the length of the input string s. $P(b,1) = 2$ if $s[b-0.5] = s[b+0.5]$ and 0 otherwise. Take the max of all the base problems for the final solution.

Pseudocode:

```
function longestPalindrome(String s):
    n = length of s
    subproblems = Hashmap of (tuple, value)
    for i = 0; i < n; i += 0.5
        if i is an integer:
            subproblems.push(((i, 1), 1))
        else if s[i-0.5] == s[i+0.5]:
            subproblems.push(((i, 1), 2))
        else:
            subproblems.push(((i, 1), 0))
    return max(evaluate(0, 1, s, subproblems), evaluate(0.5, 1, s, subproblems))

function evaluate(middle, width, s, subproblems):
    if middle + width > length of s or middle - width < 0:
        return 0
    if s[middle + width] != s[middle - width]:
        return 0
    return max(evaluate(middle, width + 1, s, subproblems) + 2, evaluate(middle - 1, width, s, subproblems),
        evaluate(middle + 1, width, s))
```

Runtime analysis:

We essentially build a 2d array of all possible substring with midpoint $0 \leq i \leq n$ where n is the length of the string input and i is incremented by 0.5 and with width $1 \leq w \leq n/2$. The overhead computation is 1 constant time comparison per entry and hence the runtime is $O(1) * O(n^2) = O(n^2)$.

Problem 4: A Sisyphean Task

★★★★ **Level**

Suppose that you have n boulders, each with a positive integer weight w_i. You'd like to determine if there is any set of boulders that together weight exactly k pounds. You may want to review the solution to the Knapsack Problem for inspiration.

For this problem, you should know how to do the proof of correctness, but need not include it in your submission. You should submit the main idea, pseudocode, and runtime.

a. Design an algorithm to do this.

Solution
Main Idea:
Let the recurring subproblem be $K(w, i) = K(w - w_i, i - 1) \vee K(w, i - 1)$ which denotes whether we include the ith boulder in our solution. The base problems are $K(w, 0) = false$ and $K(0, i)$: if we still have some nonzero weight w to account for but we have considered all i boulders, then the combination of boulders chosen so far cannot be a subset of a solution, and conversely if we have no weight left to account for and any subset of i boulders left to consider, we have already found a solution with the subset of boulders chosen so far.

Pseudocode:

```
function findBoulders(int[] boulders, w):
    i = length of boulders
    subproblems = new boolean[w][i]
    initialize all entries of subproblems to false
    for a = 0; a < w; a++:
        subproblems[a][0] = false
    for b = 0; b < i; b++:
        subproblems[0][b] = true
    for c = 1; c < w; c++:
        for d = 1; d < i; d++:
            if c - boulders[d] ¡ 0:
                subproblems[c][d] = false
            subproblems[c][d] = subproblems[c - boulders[d]][d-1] or subproblems[c][d-1]
    return subproblems[w-1][i-1]
```

Runtime Analysis:
We build up a 2d array of dimension w by i where w is the weight that we are aiming to build to and i is the number of boulders we have to work with. Each entry in the array is solved in constant time. Thus, the runtime of this algorithm is $O(wi)$.

b. Is your algorithm polynomial in the size of the input? Remember that size is in terms of how many bits we need.

Solution The runtime of the algorithm is $O(wi)$ as shown above. Let's assume that the representation of w increases by 1 bit. Then, w is effectively twice as big as before and our table increase in size by a factor of 2. In this case, a linear increase of the size of w led to an exponential increase in the size of our table. Thus, the algorithm cannot possibly still be polynomial with respect to the size of the input.

Problem 5: Advertising network

★★★★★ **Level**

You are an advertising network tasked with making bids on Facebook's mobile ad units for your customers. On each day, you make a bid of $\$\theta, \theta \in [0, 1]$, and win that day's auction with probability θ . If you don't win, then you don't spend any money. As per your contract with your customers, you need to win at least k out of n auctions that will occur this fiscal year.

An obvious approach may be to bid \$1 for each of the first k days, then nothing for the rest of the $n - k$ days. – This strategy will cost you $\$k$. However, it may not be optimal! Consider this alternative for $n = 30, k = 4$:

- Bid \$0.50 for each of the first 26 days, or until you've won 4 auctions.
- Bid \$1.00 for each of the next 4 days, if you didn't win them yet.
- This strategy will have expected cost \$2.00, and worst case cost \$4.00 – a strictly superior strategy.

Give an optimal strategy to minimize expected cost while maintaining your contract, for any k, n. You only need to explain the main idea; no need for proof, runtime, or pseudocode. Here are some hints:

- You'll need to use probability; specifically, the linearity of expectation 2.
- Parameterize your strategy as a set of variables $\theta_{n,k}$, and notice that you need to minimize some function that can be written in terms of $\theta_{n,k}$.
- To actually solve said optimization problem, you can assume you have access to a quadratic program solver that can minimize any quadratic function of a single variable, in time that is efficient.

Solution The recurrent subproblem is $P(n, k) = \theta(P(n - 1, k - 1) + \theta) + (1 - \theta)(P(n - 1), k)$ where n is the number of days left, k is the number of bids still to be won, and θ is the amount we bid/probability of winning. The base problems are $P(n, 0) = 0$ and $P(k, k) = k$: if we have more than 1 day left but no bids left to be won, then we have already won all necessary bids and spend nothing; if we have k days left to win k bids, then we have to bid 1 dollar a day for a 100% chance of winning all k bids which leads to a cost of k. We then fill out our table of size $nk/2$ (any subproblem $P(n, q)$, $q > n$, does not make sense and thus we don't consider them). We expand our table along the diagonal below the main diagonal of base problems $P(c, c)$. The problems we now solve are $P(c, c - 1)$ from $1 \leq c \leq k$. To solve for a subproblem with dependencies of base problems, we simply plug the resulting equations of the form $P(n, k) = \theta(u + \theta) + (1 - \theta)(v) = \theta^2 + (u - v)\theta + v$ into our quadratic equation solver to find the min solutions. To get our final answer, we keep solving the subproblems by parallel diagonals until we hit the right subproblem.

Problem 6: Knightmare (extra credit question)

★★★★★★ **Level**

Give an algorithm to find the number of ways you can place knights on an N by M chessboard such that no two knights can attack each other (there can be any number of knights on the board, including zero knights). The runtime should be $O(2^{3M} \cdot N)$ (or symmetrically, switch the variables).

Note that even though this question is extra credit (and thus only worth 1 pt), the staff strongly recommends that you at least attempt it and understand the solution. It will be very helpful practice for your upcoming midterm, which we remind you will be Monday March 20, 2017.

Solution

Redemption File for Homework 5

Solution