Homework 9

Due Wednesday, November 11 at 11:59 PM

Homework Guidelines

Collaboration policy Collaboration on homework problems, with the exception of programming assignments and reading quizzes, is permitted, but not encouraged. If you choose to collaborate on some problems, you are allowed to discuss each problem with at most 5 other students currently enrolled in the class. Before working with others on a problem, you should think about it yourself for at least 45 minutes. Finding answers to problems on the Web or from other outside sources (these include anyone not enrolled in the class) is strictly forbidden.

You must write up each problem solution by yourself without assistance, even if you collaborate with others to solve the problem. You must also identify your collaborators. If you did not work with anyone, you should write "Collaborators: none." It is a violation of this policy to submit a problem solution that you cannot orally explain to an instructor or TA.

Solution guidelines For problems that require you to provide an algorithm, you must give the following:

1. a precise description of the algorithm in English and, if helpful, pseudocode,
2. a proof of correctness,
3. an analysis of running time and space.

You may use algorithms from class as subroutines. You may also use any facts that we proved in class.

You should be as clear and concise as possible in your write-up of solutions.

A simple, direct analysis is worth more points than a convoluted one, both because it is simpler and less prone to error and because it is easier to read and understand. Points might be subtracted for illegible handwriting and for solutions that are too long. Incorrect solutions will get from 0 to 30% of the grade, depending on how far they are from a working solution. Correct solutions with possibly minor flaws will get 70 to 100%, depending on the flaws and clarity of the write up.

Contributors: None

1. (**Recursion vs. Memoization**)

   You are playing a board game in which you move your pawn along a path of n fields. Each field has a number ℓ on it. In each move, if your field carries the number ℓ, then you can choose to take any number of steps between 1 and ℓ. You can only move forward, never back. Your goal is to reach the last field in as few moves as possible.
   Below we've written a recursive algorithm to find the optimal strategy for a given board layout. The input to the algorithms is an array `board` containing the numerical value in each field. The first field of the game corresponds to `board[0]` and the last to `board[n-1]`. Here we use Python conventions for range so `range(a,b)` consists of integers from `a` to `b-1` inclusively.

```
recursive_moves(array board, int  L, int n ):
   # board has length n and contains only positive integers.
   if L == n-1:
     return 0
   min_so_far = float('inf')
   for i in range(L + 1, min(L + board[L] +1, n) ):
      moves = recursive_moves (board, i, n)
      if (moves + 1 < min_so_far):
         min_so_far = moves + 1
   return min_so_far
```

   (a) (1 pt.) Run `recursive_moves(board,0,len(board))` with the input array $[1, 3, 6, 3, 2, 3, 9, 5]$. Write down a list of all the distinct calls to the function (making sure to note the value of input `L`) and the return values. It's ok to "memoize" as you go.

   (b) (2 pts) Show that the algorithm is correct. It will help to formulate a claim of the form: "On input `L`, the algorithm correctly finds the number of moves needed to get from position X to position Y of the board" (for some values of X and Y that depend on `L`).

   (c) (1 pts.) Show that the worst-case running time of `recursive_moves(board,0,len(board))` (as written, with no memoization) on an input array of size n is $\Omega(2^n)$.

   (d) (1 pts) For how many *distinct* values of the inputs will the algorithm make recursive calls on inputs of size n in the worst case?

   (e) (3 pts.) Write pseudocode (or working code) for a memoized version of the `recursive_moves` procedure that runs in polynomial time.

   (f) (3 pts.) Write pseudocode (or working code) for a nonrecursive "bottom-up" version of the `recursive_moves` procedure that runs in polynomial time. Note that "bottom-up" here does not necessarily mean in increasing order of `L`; it means from the simplest subproblems to the most complicated ones.[*Hint*: You might want to program your algorithms yourself to test them. They should be short and easy to code.]

   (g) (1 pts.) Analyze the asymptotic worst-case running time of *your* algorithms on input arrays of size n. (The two algorithms will probably be the same).

ANSWER

a) If we run $recursive_moves(board, 0, len(board))$, with $board = [1, 3, 6, 3, 2, 3, 9, 5]$, we get that the minimum number of steps is 3. In addition to this, if we take a look at all the recursive calls to the function, we'll see that there are 7 distinct function calls (if we exclude the initial call), these are:

$recursive_moves([1, 3, 6, 3, 2, 3, 9, 5], 1, 8)$
$recursive_moves([1, 3, 6, 3, 2, 3, 9, 5], 2, 8)$
$recursive_moves([1, 3, 6, 3, 2, 3, 9, 5], 3, 8)$
$recursive_moves([1, 3, 6, 3, 2, 3, 9, 5], 4, 8)$
$recursive_moves([1, 3, 6, 3, 2, 3, 9, 5], 5, 8)$
$recursive_moves([1, 3, 6, 3, 2, 3, 9, 5], 6, 8)$
$recursive_moves([1, 3, 6, 3, 2, 3, 9, 5], 7, 8)$

The first and second one are only called once throughout the entire recursion, the third one is called twice, the fourth one four times, the fifth one 7 times, the sixth one 14 times, and the seventh one 22 times.

b) If we assume an input with *board* of size n, and assume that we want to get to the last field $n - 1$, from whatever field L, then let us test the base case:

The base case would be with $L = n - 1$, and in that case, if we run the algorithm, we get that the minimum number of steps we need to take is 0, since we are already at our objective.

Now if we assume for some $L = k$, so that $k > j$ and $k < n$, the algorithm will return the correct number of steps from field k to field $n - 1$, then if we have a value $j < k$, if we take $L = j$, we know that there is some shortest path between k and $n - 1$ already, so now we have two options: the shortest path between j and $n - 1$ include the path between k and $n - 1$, or it doesn't.

Now if we take the first case, then we know that we can either jump directly from j to k, which means the space between them is between 1 and $board[j]$, in which case we are done. Or we can't jump directly from j to k, however we can always jump at least 1 value in the array, and as per our assumption, we know that there is a path from a random value bigger than j and the end of the array, so even if we have to take step by step, we'll eventually find our way to the path again.

If we take our second case, where the shortest path from j to $n - 1$ does not step in k, then this can only happen if we can now directly jump from j to $n - 1$, as per our assumption, k is any generalized value bigger than j and smaller than $n - 1$, so the algorithm will still be able to find this jump.

c) In the worst case scenario, for each value it gets, it'll iterate by each other value after itself in the array. This means that, for an array of size n for example, in the worst case, it'll iterate: once for the value in index 1, twice for the value in index 2, three times through the value in index 3, 4 times through the value in index 4, etc... Since this is actually only the first level of the iteration tree, it'll end up doing the same with the values that are iterated then, and continue to do so until every branch of the recursion tree as reached $n - 1$.

If we compound this together, we'll see that, in this worst-case scenario, the tree will have 2^n leaves, and n layers, each with a different number of sub-branches depending on the level, and the value of the node in question.

d) Despite the terrible runtime in worst case scenario, there will ONLY be, at MOST n distinct values for the inputs of the recursive calls (this if you count the first call, if not it'll be n-1)

e)

```
recursive_moves(array board, int  L, int n, dictionary pastKnowledge):
   if L == n-1:
     return 0, pastKnowledge
   min_so_far = float('inf')
   for i in range(L + 1, min(L + board[L] +1, n) ):
      moves = 0
      if (i in pastKnowledge):
         moves = pastKnowledge[i] + 1
      else:
         moves, tempKnowledge = recursive_moves (board, i, n, pastKnowledge)
         pastKnowledge = tempKnowledge
         pastKnowledge[i] = moves
      if (moves + 1 < min_so_far):
         min_so_far = moves + 1
   return min_so_far, pastKnowledge
```

f)

```
nonRecursiveMoves(array board, int  L, int n):
   if L == n-1:
     return 0
   pastKnowledge[n-1] = 1
   for i in range(n-2, -1, -1):
      if (board[i] + i == n):
          moves = 1
          pastKnowledge[i] = moves
      else:
          min_so_far = float('inf')
          for j in range(i+1, min(i + board[i] + 1, n)):
              moves = pastKnowledge[j] + 1
              if (moves < min_so_far):
                  min_so_far = moves
          pastKnowledge[i] = min_so_far

    return pastKnowledge[L]
```

g) If we take that the worst case runtime for this algorithm will be given by the same set of parameters that would result in the 2^n runtime in the first algorithm, then we can assume that the worst case scenario will still require the running of every iteration/recursion before returning.

When it comes to my first algorithm, the recursion is actually only called n times, however the loop can run in the worst case, as many times as there are elements to the right of the element in question. With that in mind, the loop will NEVER surpass $n \times log_n$ runs, and if you include the recursion, you'll end up with $n \times n \times log_n$, which is just n^2, so that algorithm will run, at MOST in $O(n^2)$.

As for the second algorithm, it works a bit differently since it doesn't have recursion. With that in mind, the fact that it has an outer loop that'll iterate n times, and shares the same inner loop logic, with one difference, since it includes a fix case, so it never iterates over that, it'll actually end up being $(n-1) \times n \times log_n$, but since this isn't a proper runtime, in Big-O notation, the algorithm will run in $O(n^2)$ as well.

2. **(BST key interval)**

Your friend is given the following task: Given the root of a BST with depth h and an interval $[a, b]$ as input, write an algorithm that prints all keys in the interval in time $O(h + k)$ where k is the number of keys matching the search. (Assume $a < b$).

Your friend is unsure of whether their code works, so they ask you to take a look at it:

```
def find_keys(root, a, b):
1    if (root == NULL):
2        return
3    if (a < root.key):
4        find_keys(root.left, a, b)
5    else if (a <= root.key AND b >= root.key):
6        print(root.key)
7    if (b > root.key):
8        find_keys(root.right, a, b)
9    return
```

You decide that the code is not correct.

(a) Suggest a **single line** to change which would make the algorithm correct (you may delete what your friend has on that line). List the line number and the exact code that you would put on that line (please use the same syntax and variable names as the pseudocode e.g. `root.left, root.right, root.key, a, b`). Hint: the error is not syntactic or something else trivial.

(b) Prove the modified algorithm's correctness (with your single line change).

(c) (Choose one) Which of the following runtimes is the *lowest* upper bound on the runtime of your modified algorithm: (h is the height of the tree, k is the number of keys matching the search, and n is the number of nodes in the tree). No need to justify the answer.

i. $O(n)$

ii. $O(h + k)$

iii. $O(h \log k)$

iv. $O(k^2)$

[Hint to analyze running time: first give an upper bound on the number of nodes that this algorithm will visit *other* than the ones with keys in $[a, b]$.]

ANSWER

a) I'd remove the *else* in the *else if* in line 5, to be just an *if*:

```
def find_keys(root, a, b):
1    if (root == NULL):
2        return
3    if (a < root.key):
4        find_keys(root.left, a, b)
5    if (a <= root.key AND b >= root.key):
6        print(root.key)
7    if (b > root.key):
8        find_keys(root.right, a, b)
9    return
```

b) This might be a small thing, but it'll make a big difference, if it was an *else if* instead of just an *else*, then you'd enter the tree, recursively search the nodes until you find the first node smaller than a, and then when you would come back up, due to how *if* and *else if* statements work, then you'd be skipping the printing, and go straight into the other *if*.

This happens since the flow of control would look at the *else*, and since the recursion in question had gone into the first if, it would skip the else.

c) iv. $O(k^2)$