
Homework 9

Due Wednesday, November 11 at 11:59 PM

Student: Justin DiEmmanuele
Collaborators: Shilpen Patel, George Padavick, Matthew Gilgo

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.

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.

- The table below contains all of the distinct calls to the function and their return values.

Table 1: Distinct Calls

Call	Return Value
recursive_moves(board, 0, n)	3
recursive_moves(board, 1, n)	2
recursive_moves(board, 2, n)	1
recursive_moves(board, 3, n)	2
recursive_moves(board, 4, n)	2
recursive_moves(board, 5, n)	1
recursive_moves(board, 6, n)	1
recursive_moves(board, 7, n)	0

(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`).

- Claim: The algorithm finds the correct value for all $L \in \{0, \ldots, n-1\}$ for board with n spaces.

- Base Case: $L = n - 1$. When $L = n - 1$ the algorithm triggers the first if statement and returns the expected zero steps to the end of the board since we are already at the end.
- Assume: The algorithm is correct for all $L \in \{L + 1, \ldots, L + board[L]\}$.
- We note that all the sub problems L' where $L' > L$ are smaller than L up to the smallest problem of the base case where the algorithm simply returns zero. This proves that the algorithm will terminate as it only progressively calls smaller sub problems.
- We seek to prove correctness for position L.

  The algorithm makes calls to $\{L + 1, \ldots, L + board[L]\}$ which return correctly per the inductive hypothesis. We are interested in taking whichever number of steps that minimizes the total number of steps to the end of the board. To do this we want to use our one step from position L get to the minimum reachable cell which are covered by the induction hypothesis. The algorithm indeed takes the minimum of these possible steps and returns it with the addition of 1 for the current step correctly.

(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 $\Omega(2^n)$.

- In the worst-case, the board will have values $board[i] > n - i$ where $i \in \{0, \ldots, n-1\}$ and n is the length of the board because at each iteration the algorithm will have to look for the minimum of the max number of "min_so_far" values.
- To find how many calls are needed in this worst-case condition, we can follow the number of calls at each step from position $n - 1$ to 0. Again note that at each position i the function must be called for i to $n - 1$.
  - Position $n - 1$: Base case terminates after **1 call**.
  - Position $n - 2$: Must be called on the base (1 call) and itself (1 call) **2 calls**.
  - Position $n - 3$: The algorithm must be called on itself, $n - 2$ and $n - 1$. This gives $1 + 2 + 1 = 4$ **4 calls**.
  - Position $n - 4$: Must call $n - 3$, $n - 2$ and $n - 1$ in addition to itself. This gives $1 + 4 + 2 + 1 = 8$ **8 calls**.
- The pattern begins to become clear here: as the size increases by 1 the number of calls for that next item doubles the number of calls for the entire input $n - 1$. This is consistent with a run time of $\Omega\left(2^n\right)$.

(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?

- For an input of size n there will be at most n distinct values of the inputs to the algorithm.

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

- Below is the memoized version of the algorithm.

```
def recursive_moves(array board, int  L, int n, array memo):
    if memo is None:
        memo = [None]*n
    if memo[L] is not None:
        return memo[L]
    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, memo)
        if (moves + 1 < min_so_far):
            min_so_far = moves + 1
    memo[L] = min_so_far
    return min_so_far
```

- The array "memo" is added in this version and initialized to being null. If the algorithm is called, it first checks if memo contains a solution to the current sub problem. If there is no saved solution, the algorithm continues to calculate it and it is saved into memo as the last step of the algorithm. This guarantees that a sub problem will only be calculated once. Any call to a previously calculated problem will run in $O(1)$ 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.]

- Below is a "bottom-up" version of the algorithm.

```
def recursive_moves(board, L, n):
    OPT = [math.inf]*n
    for i in range(n - 1, -1, -1):
        if i == n - 1:
            OPT[i] = 0
        else:
            print(OPT)
            OPT[i] = min(OPT[i: min(i + board[i] + 1, n)]) + 1
    return OPT[L]
```

- The above bottom up version starts with the smallest sub problem at position $n-1$. Just the base case in previous algorithms this position is set to the correct number of zero steps since we would already be at the end of the board.
- The algorithm then iterates from $n-1$ down to position zero of the board calculating the optimum number of steps for each position along the way. The solution at each step is only dependent on the positions greater than itself so we can guarantee the algorithm will have the information needed to calculate $OPT[i]$.
  - To prove needed solution values will be available we can assume that the information at step $L+i$ is not available. If $OPT[L+i]$ is not available, we must

not have iterated through position $L+i$ yet - but since the for loop decrements from $n-1$ to 0, it is impossible that L has been reached without initializing $OPT[L+i]$ for all i.

- The algorithm uses the same principle as the original algorithm - for any position L the min number of steps to the end of the board is $min\{OPT(L+1), \ldots, OPT(L+board[L])\}+1$. In this algorithm the step L is included in the min statement but it is initialized to infinity so it will never be chosen.

(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).

- Both algorithms will consider each index in the array to find the optimum solution at that position. This first does this recursively and the second does it iteratively. This takes $O(n)$ time.
- During the call at each position the algorithm must decide which position to take a step to to minimize the overall number of steps. The algorithm will consider $n-L$ positions at each step in the worst-case. This takes $O(n)$ time.
- This makes the overall run time in both cases $O(n^2)$.

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.

- Line 5 should be changed to the following:

  ```
  if (a <= root.key AND b >= root.key):
  ```

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

- Claim: the algorithm "find_keys" will print all keys in the interval $[a, b]$.
- Base Case: node passed is null - the algorithm correctly will not print the key of this node.
- Assume: The algorithm correctly prints the keys of values within the interval $[a, b]$ for nodes in levels $root + 1$ to h.
- We will prove correctness at $root$ level to complete the induction step. There are three cases to consider.
  - Case 1: The key of root satisfies $a \leq root.key \leq b$. In this case, the algorithm is called on values less than $root$ by recursively calling on $root.left$ which will correctly print any value n where $a \leq n \leq root$ per the induction hypothesis (assumption). The algorithm will then correctly print $root.key$ as the node falls within the interval. Finally, the algorithm will call recursively on values greater than $root.key$ by passing $root.right$. This is correct per the induction hypothesis.
  - Case 2: The key of root satisfies $a > root.key$. In this case the only code triggered is a recursive call on $root.right$ or greater values than $root.key$. This

is correct by they induction hypothesis and will move the recursive calls in the correct direction of higher values.

- Case 3: the key of root satisfies $b < root.key$. In this case the only code triggered is a recursive call on $root.left$. This is correct by induction hypothesis and moves the recursive call in the correct direction of lower values.

(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 $[a, b]$.]

- **ii**