COT 5405 Analysis of Algorithms Fall 2021 MidTerm #1

UFID: 96703101
Name: *Vyom Pathak*
Instructor: Professor Alin Dobra
Due Date: October 26, 2021

Problem 1

Making change problem for U.S. Coin denomination [40]

Problem Definition

Consider the problem of providing change to an arbitrary amount N using US currency denominations, i.e \$0.01, \$0.05, \$0.10, \$0.25, \$1, \$5, \$10, \$20, \$50, \$100. Find a polynomial algorithm that, when given N, finds the exact change (or indicates that such change is not possible) using the minimum number of coins/banknotes.

Solution

The algorithm is pretty straight forward, first we sort the denominations in descending order. After that, for each denomination, divide the denomination with the given amount value and select it as the number of denominations for that coin. Update the amount by setting it to the remainder of the previous division. Now, at the end, we check weather the amount is equal to zero or not (i.e. we have taken the whole amount in the form of coins or not); if not return No Solution Exists else return the list tuples of denominations with the corresponding denomination count. The algorithm is shown in Algorithm 1. **Here, we assume that the modulo and division operator will given result with a precision of 2 decimals.**

Algorithm

Algorithm 1 Function for making change from the given amount using U.S. denomination

```
Input: Amount $A$. U.S. Denomination Set $C$.
Output: List of tuples of denominations with corresponding coin count if such change exists else "No Solution Exists".
function MakeChange($A, C$):
    $S \leftarrow \phi$
    Sort $C$ in descending order
    for coin $c' \in C$ do
        $S \leftarrow S \bigcup (c', \lfloor A/c' \rfloor)$;
        $A = A \mod c'$;
    end for
    if $A \neq 0$ then
        return "No Solution Exists";
    end if
    return S;
end function
```

Analysis

Proof of Correctness

The algorithm always find the change (if possible) with the optimal solution set.

Proof: Here as we are forming optimal solution by making consecutive choices, we can prove such a operation using proof by induction where the no. of set itself is finite i.e. maximum value of #no. of denominations = 10. So, if there exists an optimal solution set with the choices made so far for k coins, then there also exists an optimal solution for further choices till k+1 [k+1 is 10 in our case] including the situation where the first selection is made. Thus, the algorithm always finds the change (if possible) with the optimal solution set.

Running Time

The algorithm finds the denomination (if it is possible to make the change) in $O(1)$.

Proof: Here, for the sort operation, it is $O(\#of denominations)$ which is constant and thus it is done in $O(1)$ time. Also, each denomination is selected only once to compute the coin count to get the same which is also constant i.e. $O(1)$. Inside the for loop, only the addition and modulo operation re performed which are also a constant time operations. Thus, the overall time-complexity of the solution is $O(1)$.

Problem 2

Longest Path in a Tree [30+10]

Problem Definition

Given a tree, provide an efficient algorithm that finds the length of and the actual sequence for the longest path starting at the root and terminating at a leaf. If we now assume that tree edges have weights, how does the algorithm need to be modified to accommodate the generalization?

Solution

Algorithm

Algorithm 2 Helper Function to find the longest path

```
1: Input: Node $root$ with attributes as $(data, left, right)$.
2: Output: Returns the list of the longest path from the root to the leaf of the tree.
3: function LongestPathHelper($root$):
4:     if $root = \phi$ then
5:         return [ ] ;
6:     end if
7:     $rightPath = LongestPathHelper(root.right)$;
8:     $leftPath = LongestPathHelper(root.left)$;
9:     if $rightPath.length > leftPath.length$ then
10:        $rightPath.add(0, root.data)$;        ▷ Adding Data at the start of the path
11:        return $rightPath$;
12:    else
13:        $leftPath.add(0, root.data)$;         ▷ Adding Data at the start of the path
14:        return $leftPath$;
15:    end if
16:    return ;
17: end function
```

Algorithm 3 Printing the longest path

```
1: Input: Node $root$ with attributes as $(data, left, right)$.
2: Output: Prints the longest path in the given tree.
3: $path \leftarrow LongestPathHelper(root)$;
4: Print the path from the list;
5: Print $path.length$;
6: return ;
```

Analysis

Proof of Correctness

The proposed algorithm always finds the longest path for any Binary Tree with n nodes.

Proof: We can prove this using induction.

Base case: Suppose (without loss of generalization) for a tree of only one or two nodes, the longest path is one or two respectively. For the third node, if we add it on the longest path, than the algorithm will pick the same path with the new added node as the longest path. If, we do not add it to the longest path, then also, the new path will be of equal length as that of the previous longest path and thus the algorithm will again pick the longest path.

Inductive Hypothesis: Assume for a tree with k nodes, we have the optimal longest path from the proposed algorithm.

Inductive Step: Prove that, if we add a one node and make a total of (k+1) nodes, we still get the optimal longest path from the proposed approach.

There are 2 cases in general where we can add the new node:
Case 1: The node is added to the optimal longest path at the end.
For this case, if we run the algorithm, as we have already got the optimal longest path [$\because$ induction hypothesis], the added node will also selected on the same longest path and thus will generate the optimal longest path with updated length.
Case 2: The node is added to any other part of the tree but the optimal longest path at the end.
For this case, if we run the algorithm again, it doesn't matter that much because the added node will either create an equal length path or a shorter path and thus we still return the optimal largest path. [$\because$ from the Base Case]
Hence, the proposed algorithm holds $TRUE$ for a Binary Tree with n nodes.

Running Time

The running time of the proposed algorithm is $O(n)$, where n is the number of nodes in the given Binary Tree.

Proof: As the solution to the problem is recursive, and divides the problem into 2 half size sub-problems in each recursion and also performs only one operation in each recursive step; the recursive equation can be expressed as follows:

$$T(n) = 2 \cdot T(\frac{n}{2}) + 1$$

Now, we can derive the time complexity of this equation using **Master's theorem's Case 1** [2] i.e. where $[f(n)\ =\ O(n^{log_b(a-\epsilon)})\ where\ b = 2,\ a = 2\ and\ \epsilon = 1]$.

Thus, the final run time can be derived as follows:

$$T(n) = O(n^{(log_b(a))})\ (where\ b = 2\ and\ a = 2)$$
$$T(n) = O(n^{(log_2(2))})$$
$$T(n) = O(n)$$

Thus, the run-time of the proposed algorithm is $O(n)$.

Follow Up

For the weighted binary tree, the algorithm 3 will change such that instead of comparing the length of the path for each sub-tree, we will compare the weights of each edges for the particular node value. The modified algorithm is shown in Algorithm 4. Here, we assume that, longest path for a weighted binary tree is the weighted path and thus, the path with the most weight sum is the longest path. Also, we assume that the Node attribute all holds the edge weights for the right as well as left edge.

Algorithm 4 Helper Function to find the longest path in weighted graph

```
Input: Node $root$ with attributes as $(data, left, right, leftWeight, rightWeight)$.
Output: Returns the list of the longest path from the root to the leaf of the tree w.r.t to the weight-sum
along with the total weight value of the path.
function LongestPathHelper($root$):
    if $root = \{\}$ then
        return [ ],0 ;
    end if
    $rightPath, rweight = LongestPathHelper(root.right)$;
    $leftPath, lweight = LongestPathHelper(root.left)$;
    if $rweight + root.rightWeight\ >\ lweight + root.leftWeight$ then
        $rightPath.add(0, root.data)$;    ▷ Adding Data at the start of the path and updating the weight
        return $rightPath,\ rweight + root.rightWeight$;
    else
        $leftPath.add(0, root.data)$;    ▷ Adding Data at the start of the path and updating the weight
        return $leftPath,\ lweight + root.leftWeight$;
    end if
    return ;
end function
```

Problem 3

Largest element in a k shifted sorted list [20]

Problem Definition

Suppose you are given an array A[1..n] of sorted integers that have been circularly shifted k positions to the right (for an unknown k). For example, [35, 42, 5, 15, 27, 29] is a sorted array that has been circularly shifted k = 2 positions, while [27, 29, 35, 42, 5, 15] has been shifted k = 4 positions. We can obviously find the largest element in A in O(n) time. Describe an O(log n) algorithm.

Solution

Algorithm
We can use modified binary search on the the given array to find the maximum value. Here, we have to basically find the first number which is lesser the first number. Here, we check for each half of the array, if the middle element is lesser than the first element, then the maximum element lies in the left half of the list, else the maximum element lies in the right half of the list. Also, we handle the middle element i.e. if it is at the start of the list and if it is somewhere in the middle ans is the maximum; we simply return the middle value. The algorithm is shown in Algorithm 6.

Algorithm 5 Function for performing the binary-search

```
Input: Start index $l$. End index $r$. List of numbers $nums$.
Output: Maximum number in the given list.
function FindMaxHelper($l$, $r$, $nums$):
    if $l = r$ then
        return nums[l];
    end if
    $m \leftarrow \lfloor \frac{(l+r)}{2} \rfloor$;
    if $m = 0$ and $nums[m] > nums[m+1]$ then
        return $nums[m]$;
    else if $m < r$ and $nums[m+1] < nums[m]$ and $m > 0$ and $nums[m] > nums[m-1]$ then
        return $nums[m]$;
    end if
    if $nums[m] < nums[l]$ then
        return $FindMaxHelper(nums, l, m-1)$;
    else
        return $FindMaxHelper(nums, m+1, r)$;
    end if
end function
```

Algorithm 6 Function for finding the maximum value

```
Input: List of numbers $nums$.
Output: Prints the maximum value from the given list
function FindMax($nums$):
    $value \leftarrow FindMaxHelper(0,\ nums.length - 1,\ nums)$
    print(value);
    return ;
end function
```

Analysis
Proof of Correctness
The algorithm always finds the largest number in a z shifted sorted array where $r - l = n$ for all n.
Proof: We can prove this using induction.
Base Case: When the we have only one element ($l = r$) we just return that element only. If also in a list of size three and the value of m is the index 1, and the middle element is the maximum element we return the element.
Induction Hypothesis: Let $r - l <= k$ where the proposed algorithm always finds the correct solution.
Inductive Step: Prove that, if $l - r = k + 1$ the proposed algorithm still finds the largest number.
There are three cases in this situation:
Case 1: Suppose the middle element is the max number, then using the base case, we simply return that value.
Case 2: Suppose $nums[l] > nums[m]$, this means that we have to find the solution on the left half of the list i.e. in the range (l) *and* $(m - 1)$. We do this because of the z shift, it is the case that the shift is at-most equal to the value of m for the given list. This implies that the larger number are a part of the starting of the list rather than the end of the list. Here, the value of n for the recursion is $\lfloor (l + r)/2 \rfloor 1 - l$. Here, if $r + l$ is even, then n is $(r - l)/2$ which is also smaller than $r - l$ $(k + 1)$. And, if $l + r$ is odd, then n is $(r - l)/2 - 1$ which is smaller than $r - l$. Thus, the recursive call works in a range of a smaller range than $k + 1$ size and thus using induction it works i.e. gives the largest element.
Case 3: Suppose $nums[l] < nums[m]$, this means that we have to find the solution on the right half of the list i.e in the range $(m + 1) and (r)$. We do this because of the z shift, it is the case that the shift is at-least equal equal to the m value i.e. the larger element is present either on the right half or it is the right-most element. In this case we need to show that $r - (m + 1) \leqslant (r - l)$. So, similar to the above condition, if $r + l$ is even, we get $(r - l)/2 - 1$ which is less than $r - l$. And, if $r + l$ is odd, we get $(r - l)/2 - 1/2$ which is also less than $r - l$. Thus, this recursive call works in the smaller range of the list and thus it is assumed to be working using the induciton hypothesis.

Finally, as all the cases of the indcutive step are working, we conclude that the FindMax Algorithm always finds the largest number is correct.

Running Time
The running time of the proposed algorithm is $O(log(n))$, where n is the size of the input list.
Proof: As the solution to the problem is recursive, and divides the problem into half size in each recursion and also performs only one operation in each recursive step; the recursive equation can be expressed as follows:

$$T(n) = T(\frac{n}{2}) + 1$$

Now, we can derive the time complexity of this equation using **Master's theorem's Case 2** [2] i.e. where $[f(n) = \Theta(log_b(a)) \ and \ b = 2, \ and \ a = 1]$. Thus, the final run time can be derived as follows:

$$T(n) = O(n^{(log_b(a))} \cdot log(n) \ (where \ b = 2 \ and \ a = 1)$$
$$T(n) = O(n^{(log_2(1))} \cdot log(n))$$
$$T(n) = O(log(n))$$

Thus, the run-time of the proposed algorithm is $O(log(n))$.

Problem 4

Finding generalized set for Problem 1 [30 bonus]

Problem Definition

For problem 1, find the most general set of currency so that the algorithm you found is still correct. Your solution will be judged based on generality. Unless the solution is correct and the generalization is non-trivial, no points will be awarded.

Solution

A coin system which always gives optimal solution when the proposed greedy algorithm is applied can be considered as a canonical coin system [1]. Thus, for a 4 to 5 coin system, we can prove that many series can be a canonical series using work of Cai and Zheng with Theorem 1 and Theorem 2. This helps us to generalize the algorithm for any given coin system. i.e. check whether the given coin system is canonical or not, if it is, then we can just say that we can the greedy algorithm will always give the optimal solution. [1]

References

[1] Xuan Cai. Canonical coin systems for change-making problems. In *2009 Ninth International Conference on Hybrid Intelligent Systems*, volume 1, pages 499–504. IEEE, 2009.

[2] T Cormen, C Leiserson, R Rivest, and Clifford Stein. Introduction to algorithm the mit press. *Cambridge, Massachusetts*, 1990.