AOA Midterm

Kasiviswanathan Srikant Iyer 5222-2519

October 26, 2021

1 Question 1

[40] 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.

1.1 Solution

In the given problem, the goal is to get to the value of N using the currencies denominations given to us, while using the least number of coins/bank notes.

To achieve this, using largest possible value currencies to reach as close as possible to the given value N will ensure the lesser number of currencies used and divide the value to be reached to find the number of times it will go. The same step can be repeated with the next largest value till the goal of N is reached. If the provided denominations fail to sum up to the value of N, then the algorithm fails and needs to return "No Solution".

1.2 Algorithm

1. Store the value N in a variable n
2. Arrange the provided currencies in increasing order so that the largest value less than N can be selected.
3. Select the largest value from the arranged currencies which is smaller than the value n.
4. If there is no value smaller than n and the value of n is more than 0 this problem does not have a successful solution.
5. If there is a smaller available value in the currencies list, it needs to be divided from n and increase count by quotient. The new value of n will be the remainder of the division.
6. While the value of n is greater than 0, we need to go back to step 3 and repeat.

1.3 Proof of termination

In the given algorithm the value of n keeps decreasing each time when the largest currency of value less than it is found.

1. If no currency value of smaller than n is found the algorithm terminates.
2. Otherwise the remaining value n is run through the code again, and if the previous value does not go, the next denomination is tried till the value of n is turned to 0 when it terminates.
3. If the value has reached the smallest denomination, and still the value if n is smaller, we are taken back to the 1^st condition and the algorithm terminates with no solution.

This provides the proof of termination as in every possible condition the algorithm terminates itself.

1.4 Psudocode

Algorithm 1 Greedy algorithm to find minimum number of coins

```
Require: $N \geq 0$
Ensure: $Min(d_i) < N$                     ▷ Where $d_i$ contains all the available denominations
  Sort given denominations in increasing order such that $d_1 < d_2 < d_3 ... < d_n$
  $S \leftarrow \phi$                      ▷ Empty set where we will store the coins selected
  while $N > 0$ do
      $i \leftarrow$ largest denomination $d_i$ such that $d_i \leq N$
      if No such $i$ then return "no solution"
      else
          $temp = N/d_i$
          $N \leftarrow N\%d_i$
          while temp>0 do
              $S \leftarrow S \cup d_i$
              temp ←temp-1
        end while
    end if
end while
return S
```

1.5 Proof by Induction

Every denominating of coin will be used specific number of times. If used more times such that the sum of its value is more than or equal to the next biggest denomination, that denomination would have had been used first and thereby making the smaller denomination absolute.

The max limit of the denominations is given below.

Position of i	Value of d[i]	Optimal Solution	Max Value of Coins
i = 1	USD 0.01	d[i] ≤ 4	null
i = 2	USD 0.05	d[i] ≤ 1	0.04
i = 3	USD 0.10	d[i] ≤ 2	0.09
i = 4	USD 0.25	d[i] ≤ 3	0.24
i = 5	USD 1	d[i] ≤ 4	0.99
i = 6	USD 5	d[i] ≤ 1	4.99
i = 7	USD 10	d[i] ≤ 1	9.99
i = 8	USD 20	d[i] ≤ 2	19.99
i = 9	USD 50	d[i] ≤ 1	49.99
i = 10	USD 100	Exceeded Limit	99.99

1. Let the optimal way to change $di \leq n < di + 1$: greedy takes currency d_i.
2. Then any optimal solution must also take currency di.
   - otherwise, multiple coins of type $d1, \ldots, d(i-1)$ need to be add up to n
   - but the table above shows that this cannot be done optimally
3. Problem reduces to coin-changing n–di cents, which, by induction, is optimally solved.

1.6 Complexity

The complexity of the above algorithm will be equal to the number of denominations present which here we can take as x. This is because, in the very worst case scenario that every currency needs to be used to achieve the value, every denomination will divide the value once and then move on to the next denomination to be divided.

The complexity in this case will be O(x) where x is the number of denominations present. As x is given as 10 here we have complexity of this problem as O(1)

2 Question 2

[30+10] 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 [30]. If we now assume that tree edges have weights, how does the algorithm need to be modified to accommodate the generalization?

2.1 Solution

In the given problem, we need to recursively traverse the tree's right and left branches and we need to find the longest path. The same step is followed for every level of the tree to get the longest part of any binary tree.

2.2 Algorithm

1. select the root node
2. if it exists select left node of the root node and increase left length by 1. Make left node as root and goto step 1
3. if it exists select right node of the root node and increase right length by 1. Make right node as root and goto step 1
4. if root node is null return.
5. Compare the length of right node nad left node branches and return the greater.

2.3 Proof of termination

In the given algorithm, the tree will traverse both the left and the right branches. The traversal down a branch returns when the leaf node is reached. Similarly the whole algorithm returns once all the nodes are traversed. As we are traversing the whole tree, we are bound to finish traversing and terminate the algorithm.

2.4 Psudocode

Algorithm 2 Recursive traversal to find the longest path

```
function longest(rootnode){
left=longest(root.left)
right=longest(root.right)
left.add(root)
right.add(root)
if $left > right$ then
    return left
else
    return right
end if
}
```

2.5 Weight tree

For a weighted tree, we need to consider the weights of the branches and add those to the comparison between left and right branches. The same algorithm can be used for weighted tree with minor modification.

2.6 Psudocode for weighted tree

Algorithm 3 Recursive traversal to find the longest pathin a weighted tree

```
function $longest(rootnode, sum)${
left=longest(root.left,sum)
right=longest(root.right,sum)
left.add(root)
right.add(root)
sum=Max(sum(left),sum(right))
if $sum(left) > sum(right)$ then
    return left, sum
else
    return right,sum
end if
sum
}
```

2.7 Proof by Contradiction

Let us assume that the solution S is reached using this algorithm from root to leaf, but it is not the longest path that can be travelled.

1. There must exist a solution S' which is longer to some other leaf node
2. let there be a common point P in both paths S and S'
3. our algorithm choose the path P to S instead of P to S' so it must be that P to S is longer than P to S'
4. Total length of path to S = root to P + P to S
5. Total length of path to S' = root to P + P to S'
6. as root to P is constant and P to S is longer than P to S' so
   root to S>root to S' /item But this contradicts our assumption that there exists a solution S' which has different leaf node and is the longest path.

Since our assumption was wrong, solution S given from the algorithm is the most optimal solution.

2.8 Complexity

The complexity of the above algorithm will be equal to the number nodes in the tree as each node needs to be traversed once to find longest path possible to every leaf. O(n) is the complexity where n is the number of nodes in that tree.

3 Question 3

[20] 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.

3.1 Solution

In the given problem, the goal is to get to the largest value in the given array. As the array was a sorted array before it was rotated, it can be assumed that the largest value will be smaller than the smallest value. Due to the rotation the smallest value may be in the right of the largest value. This can be used to find the largest value using linear traversal in O(n) time.

As the rest of the array is sorted too, and then rotated, we know that the value will increase till the largest value and then drop to the smallest value and then again increase till the end. In this condition, we can be sure that the first element will be larger than all the elements after the biggest element.

By using divide and conquer we can divide the array into 2 parts. This step can be repeated till the middle element will have large value to the left and smallest value to the right. In this condition the largest value can be found in O(log n) time.

3.2 Algorithm

1. Find the middle value of the array by dividing the size of array by 2 .
2. If the first element is larger than the middle element, then return to step 1 with start as start of array and end as middle value.
3. If the middle+1 element is larger than the last element, return to step 1 with start as middle+1 and end as end of array
4. When both step 2 and 3 are false the largest value is the middle element.

3.3 Proof of termination

As the value of largest element will always be present in the sorted array, even after being rotated, the maximum value will be always to the right of the array if middle is bigger than the last and always to the left if start is bigger than middle.

As the function is a recursions function till both the given conditions is false, which it will be once it reaches the max till which it will keep running, and we know for certain there is a max, the algorithm will certainly terminate.

3.4 Psudocode

Algorithm 4 Recursion to find the max value

```
Require: $Arr[n]$                ▷ Where sorted array could be rotated by k values
  $first \leftarrow 0$
  $last \leftarrow n$
  $function$ Biggest $(first, last)$
  $m \leftarrow lowerbound((first + last)/2)$
  if $first > m$ then
      $Biggest(first, m)$
  end if
  if $m + 1 > last$ then
      $Biggest(m + 1, last)$
  end if
  end $function$
  return m
```

3.5 Proof by Example

Lets take an array, Arr where values 35, 42, 5, 29, 15, 27 are stored. Once sorted, this will be 5, 15, 27, 29, 35, 42. If it was rotated for 2 values, because k=2, the values would be 35, 42, 5, 15, 27, 29.

Lets run the algorithm on our array arr:

1. We know the first element is 0 and last is 5 so mid element is 2 as per lowebound ((first+last)/2).
2. We see that $arr[i] > arr[m]$.
3. We set first as 0 and last as m and rerun the function.
4. We know the first element is 0 and last is 2 so mid element is 1 as per lowebound ((first+last)/2).
5. We see that $arr[i] < arr[m]$ and $arr[m + 1] < arr[j]$ so m is suppose to be the largest value.
6. Here m is 1 and arr[1]=42, which we can know is the largest in the array.

Because it worked on this random set of arrays, it will also work in any set of arrays which are similarly sorted and then rotated.

3.6 Complexity

The complexity of the above algorithm will be equal to the number of elements checked for largest value. T(n) is at most one half of the array so T(n)=T(n/2)+1 so the complexity of T(n) will be O(log n).

4 Question 4

[30 bonus] 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.

4.1 Solution

To get a set of denominations where the algorithm found will be correct and we find the general set of currency, we need to have denominations that do not often form the larger number, of if they do, help reduce the number of currencies

Countries already use the most optimal denominations that can be used to represent any value. But countries also consider the easier to calculate values for humans. Taking this into consideration any value that satisfies:

1. can represent any value with least number of currency
2. is easy to calculate mentally

will be a good candidate for being printed currency.

4.2 Possible Values

For this condition a subset of imperial values would be a good fit. Consider the set $[1, 3, 6, 24, 72]$ of which all are multiples or fractions of the number 12. This subset has less than the number of denominations as the US dollar system has and is at times more efficient than the US dollar system.

If the consideration of human calculation was to be removed there are a couple more sets that may be beneficial:

1. Values from the Fibonacci numbers where each value is more than twice of the previous number $[1, 2, 5, 13, 34, 89]$.
2. Values from the Prime numbers where each value is more than twice of the previous number $[1, 3, 7, 17, 37, 79]$.
3. Reducing the interval between initial numbers $[1, 2, 3, 5, 10, 30, 90]$.

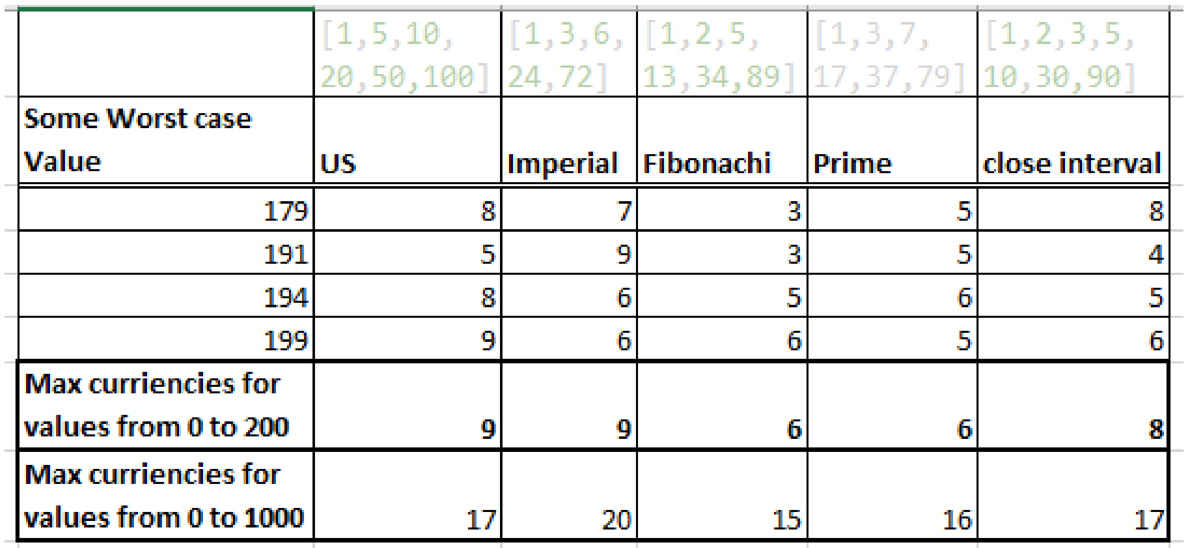
4.3 Observation

There are many more sets of currencies which would enable us to carry less currencies for the same value compared to the traditional US dollars.
The number of currency increases significantly if the largest value is smaller than other sets.
The number of currencies needed for different values can from value to value, so a set of at least 1000 values needs to be checked to get to a clear picture.

4.4 Proof by example

In the table 1 below, I have shown the use of the proposed currencies for values from 1 to 1000.

1. The Imperial values look to be doing good for values till 200, not being worse than traditional US currency, but get higher in the 1000s due to the max value being 72 in it as opposed to 100 in US. Fibonacci numbers are the best performers in both smaller numbers and larger numbers. It can get higher than US due to its max value being less than US's.
2. Prime and close interval has performed better or at par with US denominations



	[1,5,10, 20,50,100]	[1,3,6, 24,72]	[1,2,5, 13,34,89]	[1,3,7, 17,37,79]	[1,2,3,5, 10,30,90]
Some Worst case Value	**US**	**Imperial**	**Fibonachi**	**Prime**	**close interval**
179	8	7	3	5	8
191	5	9	3	5	4
194	8	6	5	6	5
199	9	6	6	5	6
Max curriencies for values from 0 to 200	**9**	**9**	**6**	**6**	**8**
Max curriencies for values from 0 to 1000	17	20	15	16	17

Figure 1: The number of currencies needed

4.5 Conclusion

Any of the given other sets can be used as currencies. The imperial and Fibonacci are the most promising sets for quick solving and efficient storing respectively. The only shortcoming that needs to be fixed for them to be more efficient than US currencies is to add a value more than 100 which would effectively make them always better at having less currency than the US system