Analysis of Algorithms - Midterm

Swara Lawande : 6081-3129

October 26, 2021

Contents

1 Problem 1 - Finding change using minimum denominations

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 Pseudocode

Algorithm 1 Find change using minimum Denominations

```
Input :
D - Array of coin denominations
m - Number of denominations
N - Amount of money

Result : S - List of denominations needed with minimum number of
coins as change for the amount n

Sort array D in descending order of denominations.
for $i \leftarrow 0$ to $m - 1$ do
    while $n \geq D[i]$ do
        $S \leftarrow S \cup D[i]$
        $N \leftarrow N - D[i]$
        if $N = 0$ then
            break;
        end if
    end while
end for
if $N \neq 0$ then
    return "No Change"
end if
return S
```

1.2 Proof

- We have used greedy algorithm to solve this problem. In order to prove the correctness of our algorithm, we will be analyzing the measure of progress for this problem.
- We need to get change for the given amount of money using as less number of coins and notes as possible. In other words, we require our while loop to run as lesser number of times as possible.
- Every time we add a denomination D[i] to the list of change, we subtract that value from the amount N and our goal is to make the value of N 0 as soon as possible.
- Consequently, we identify value of N as the algorithm's measure of progress.
- We shall prove the correctness of our algorithm, by proving that it "stays ahead" of any other algorithm at every iteration.We prove this by induction.
- **Proof :**
  - Let us consider denote our algorithm as 'i' and the other algorithm as 'j'.
  - **Initialization :** In iteration 1, our algorithm will always subtract the largest denomination, $D(i_1)$, smaller than N. If j chooses a denomination $D(j_1) \leq D(i_1)$, value of $N(j_1) \geq N(i_1)$. Thus, our algorithm will always stay ahead for the base case.
  - **Induction :** We assume for that for an iteration r, our algorithm still remains ahead. We need to prove that it will still remain ahead for iteration r+1.
  - Suppose, we have at iteration r, value of $N_r = 10$ for both the algorithms and the denominations available less that 10 are [5, 2, 1].
  - Lets assume that greedy algorithm j does not produce the optimal solution at iteration r. For example $D(i_{r+1}) = 2$ and $D(j_{r+1}) = 5$. Thus, $N(i_{r+1}) = 8$ and $N(j_{r+1}) = 5$. Thus, $N(i_{r+1}) > N(j_{r+1})$
  - It means that, it has not selected the highest denomination available lesser than N. But, this is a contradiction, as our algorithm

  will always choose the highest denomination lesser than current value of N.
  - Thus, our algorithm will always stay ahead of j at all iterations.
  - **Termination :** The value of N is reducing in every iteration. The terminating condition is when either the value of N becomes 0 or if no change denomination is available than N. Thus, our algorithm will always terminate.

1.3 Analysis of Running Time

- The time required to sort the denominations in descending order is $O(mlog_2m)$
- The for loop in the pseudocode runs n times. However, the work done in every iteration is not constant.
- The work done in every iteration actually depends on the number of times it enters the while loop, that is, the number of times a denomination is chosen to be added in the list of change. For instance, if N=1000, then it will go in the while loop 10 times for denomination 100 and 0 times for other denominations. Whereas, for N=45, it will add dollar 20 twice and dollar 5 once.
- Lets say the number of times a note or coin is added to the list is k. So the constant work c inside the loop will be done ck times and time complexity of the loop is O(k).
- Thus, the total running time of the algorithm will be time required to sort the denominations array plus the time taken in the loop. So the total running time will be: $O(k + mlogm)$
- But $k >> m$, thus running time can be denoted by O(k), where k is the number of coins and nodes required to make the change.

2 Problem 2 - Finding longest path in a tree

Given a binary 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 Pseudocode

The nodes in the binary tree are represented by data, left and right where,
data = value of the node
left = node to the left
right = node to the right
The recursive function to find the longest path is given below:

```
function GETLONGESTPATH($currentNode$)
    if $currentNode = null$ then
        return emptyList
    end if
    $rightPath \leftarrow getLongestPath(currentNode.right)$
    $leftPath \leftarrow getLongestPath(currentNode.left)$
    if $leftPath.size > rightPath.size$ then
        $leftPath \leftarrow leftPath \cup currentNode.data$
    else
        $rightPath \leftarrow rightPath \cup currentNode.data$
    end if
    if $leftPath.size > rightPath.size$ then
        return leftPath
    else
        return rightPath
    end if
end function
```

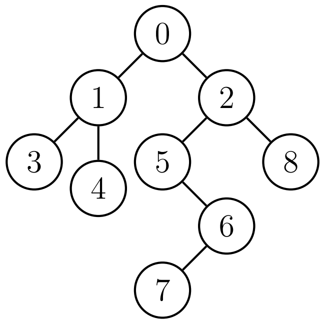
Algorithm 2 longestPath

```
Input : BT - Binary Tree with root node 'root'
$longestPath \leftarrow emptyarray$                    ▷ Initialize longestPath
$longestPath \leftarrow getLongestPath(root)$
Print nodes from longestPath
return longestPath
```

2.2 Proof

- We have followed a divide and conquer approach to solve the problem of finding the longest path in a binary tree.

- Essentially, this problem has the property of optimal substructure. That, is we need to find the longest path in the sub-trees at lower levels to find the longest path from the root.
- We prove the correctness of our algorithm using method of induction.
- **Initialization :** For the lowest level subproblem, we are dealing with leaf nodes where n = 1. As a leaf node will be the only node in the subproblem, it will always be optimal as there are no other competing paths. Thus, the base case is established.
- **Induction :** Lets assume that in the i^{th} recursion, we have the longest path from a node at level i in the tree. We need to prove that the node at level i+1 will always return the longest path as well.
- For this, consider the following Binary Tree.



- Clearly, the longest path is $0 \rightarrow 2 \rightarrow 5 \rightarrow 6 \rightarrow 7$
- Let the levels in the tree start from 1. For the induction step, we assume that the sub-problems at level 3 return the longest sub-problem. Now, we need to prove that nodes at level 2 will also return the optimal solution to its sub-problem.
- Lets consider the right part of the tree which consists of the actual longest path. At level 3, 'node 5' will return the array [7, 6, 5] and 'node 8' will return only [8] in their recursion respectively.
- Now, we need to prove that 'node 2' at level 1 will also return the optimal solution to its sub-problem. Length of array returned by 'node 5' is 3 and the one returned by 'node 8' is 1. We compare the lengths of the left and the right sub-problems, add the current node to the

array with bigger size and return that array to its parent node. Consequently, 'node 2' will be added to the array returned by 'node 5', giving us [7, 6, 5, 2], which is clearly the optimal solution of the sub-problem at 'node 2'.

- Thus, we conclude that if sub-problems at level i+1 in the tree return an optimal solution, the nodes at higher levels will always return the optimal solution as well. This proves the induction hypothesis.
- **Termination :** As we follow a bottom-up approach in solving the longest path problem from the root, the algorithm will always terminate once it reaches back to the root node and all the lower nodes are traversed once.

2.3 Analysis of Running Time

- We compute the runtime of this algorithm using the Master Theorem.
- The equation for Master Theorem is stated as follows:

$$T(n) = aT(n/b) + f(n) \tag{1}$$

  where, n = problem size
  a = number of subproblems in the recursion and a geq 1
  n/b = size of each subproblem
  f(n) = cost of work done outside the recursive calls like dividing into subproblems and cost of combining them to get the solution.
- As we are dealing with a binary tree, at every level of recursion the problem is reduced to 2 subproblems of at most n/2 size. Every recursion performs work for constant time c, so f(n) = O(1). Thus, substituting n = 2, n/b = n/2 and f(n) = 1 in equation (1).

$$\begin{aligned}
T(n) &= 2T(n/2) + 1 && \text{level 1 recursion} \\
T(n/2) &= (2^2 T(n/2^2) + 1) + 1 && \text{level 2 recursion} \\
&= 4T(n/4) + 2 \\
T(n/4) &= 2^3 (T(n/2^3) + 2) + 1 && \text{level 3 recursion} \\
&= 8T(n/8) + 3. \\
&. \\
&. \\
&.
\end{aligned}$$

From the above recursions, we get, at level k,

$$T(n/2^{k-1}) = 2^k \cdot T(n/2^k) + k \qquad (2)$$

When we reach the last subproblem, it means we have reached leaf node thus n/b = 1

$$\begin{aligned}
n/2^k &= 1 \\
k &= log_2 n
\end{aligned}$$

Substituting this in equation (2)

$$T(n) = 2^{log_2 n} \cdot T(1) + log_2 n \qquad (3)$$

As $2^{log_2 n} = n^{log_2 2}$ and $log_2 2 = 1, T(1) = 1$

$$\begin{aligned}
T(n) &= n + log_2 n \\
&= O(n)
\end{aligned}$$

Thus, the asymptotic time complexity of the above algorithm is O(n).

2.4 Longest Path of Weighted Tree

- In order to find the longest path in a weighted binary tree, we just need to make a minor change in the recursion algorithm.

- Instead of adding nodes to the longest path list by comparing the length of left and right subtrees, we select the node to be added based on the weight of the subtrees.

- The function getLongestPath() will return an Object of type Longest-PLP[], weight, where LP[] is the array of nodes in the longestPath, while weight is the weight of the longestPath detected.

- Every node will have two additional parameters, weightLeft and weightRight. These weights will be added to the overall weight of the longestPath while traversing the tree branches.

- The recursive function getLongestPath(currentNode) can be modified to getLongestPathWeightTree(cuurentNode, weight) as follows to get the longest path in weighted binary trees.

```
function GETLONGESTPATHWEIGHTTREE($currentNode, weight$)
    if $currentNode = null$ then
        return emptyList
    end if
    $rightW \leftarrow currentNode.weightRight + weight$
    $leftW \leftarrow currentNode.weightLeft + weight$
    $lpRight \leftarrow getLongestPath(currentNode.right, rightW)$
    $lpLeft \leftarrow getLongestPath(currentNode.left, leftW)$
    if $lpLeft.weight > lpRight.weight$ then
        $lpLeft.LP \leftarrow lpLeft.LP \cup currentNode.data$
        return lpLeft
    else
        $lpRight.LP \leftarrow lpRight.LP \cup currentNode.data$
        return lpRight
    end if
end function
```

- The recursion will be called as follows LongestP lp $\leftarrow getLongestPathWeightTree(root, 0)$

- This algorithm will also take O(n) time to run, similar to the first algorithm.

3 Problem 3 - Finding largest element in circularly shifted sorted array

Suppose you are given an array A[1..n] of distinct 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 Pseudocode

We use a divide and conquer approach to find the largest element. The following recursive function (findMax) is used by the algorithm.

```
function findMax($A, left, right$)
    if $left = right$ then
        return A[left]
    end if
    $mid \leftarrow (right + left)/2$          ▷ If (right+left) is odd we floor mid
    if $A[mid] > A[mid + 1]$  then
        return A[mid]
    else if $A[left] > A[mid]$ then
        return findMax(A, left, mid-1)
    else
        return findMax(A, mid+1, right)
    end if
end function
```

Algorithm 3 Find largest element in array

```
Input :  A -  Sorted array, circularly shifted right k times
$size \leftarrow$ size of array A
return findMax(A, 0, size-1)
```

3.2 Proof of Correctness

- To solve this problem, we take advantage of the fact that the array is sorted. If the array is not rotated at all (k=0), the largest element is the last element.

- But, when the array is circularly shifted right k times, we notice that the only element with next element smaller than itself is the max element.
- We take advantage of this observation to solve the problem in O(log n).
- At every iteration we reduce the search space by half, by selecting the portion of the array where the maximum element is. In order to identify the portion of array with the maximum element, we analyze the left side. If the first element on the left side is larger than the mid element, it means that the max element is in this side. If not, it means the left part is properly sorted and max element is on the right side.
- We prove the algorithm using the method of induction. As we are using divide and conquer to solve this algorithm, the size of search space reduces by half every iteration. Thus, we use the problem size as the variable in our proof by induction. We try to prove that if our algorithm works for a problem size n, then it also works for size n+1.
- **Proof:**
  - Let left be the index of the first element and right be the index of the last element in the search array. Let mid = $\lfloor (left+right)/2 \rfloor$ be the index of the middle element.
  - Let n be the size of the problem such that n = right - left. Let x be the largest element in the array that we have to find.
  - **Base Case :** The problem size = 1, that is n = 0 and right = left = mid. As the array has only one element, it will always be the largest one and we will always get the solution.
  - **Induction :** We assume that we are able to find x in an array of size (right - left) $\leq$ n. We have to prove that our algorithm will work for a problem size of n+1 as well. We come across 3 cases as follows:
    * **Case 1 :** $array[mid] > array[mid + 1]$
      As the array is sorted and rotated, it is obvious that x is the only element whose next element is smaller than itself. So, if the above condition is met, we have found a solution.

    - **Case 2 :** $array[left] > array[mid]$
      The left portion must be in increasing order of elements for a sorted array. But, the first element being larger than the middle element indicates that x was shifted to the left of the middle element by circular rotation and must be present in this part. So, we explore only this part of the array by reducing the problem size to $\lfloor(right - left)/2\rfloor$-1- left *leq* n. According to the assumption made in the induction step, we are able to find the solution to problem sizes $\leq$ n.
    - **Case 3 :**
      If the above two cases are not met, it means that x is present to the right of the middle element in the array. Thus, we have to explore only that part of the array and we reduce the problem size as $right - \lfloor(right - left)/2\rfloor + 1 \leq n$. As we can find the solution in search space of this size, according to the induction hypothesis, our algorithm is correct.
  - **Termination :** The algorithm will run recursively until it meets either of the termination conditions. If the search space left is of size 1, or if the next element is lesser than the current mid element. As a solution is found for every case in the induction step, the algorithm will always terminate.

3.3 Analysis of Running Time

- The algorithm we used is a divide and conquer approach, where constant work c is performed in every recursion. If the solution is not found in that recursion, the problem is further reduced to a size of at most n/2. Thus, the recursion will end when n=2.

- Let T(n) represent the running time of the algorithm on input of size n.

$$T(n) \leq T(n/2) + c \qquad (2)$$

  where, $n > 2$ and $T(2) \leq c$

- We prove the running time of the algorithm by unrolling recursions. We do this in the following 3 steps:

  Analyzing first few recursions

- At the first recursion, the problem size is n. It runs in a constant time c.
- At level 2, the problem size is reduced to at most n/2 and it again runs for a constant time c.
- Level 3 has at most n/4 input size and it runs for another c time.

Identify a pattern

- From the above recursions, we derive that at every level j, the problem size is reduced to at most $n/2^j$ and every recursion contributes c to the overall running time.

Summing runtime of all recursions

- There are a total of $log_2 n$ levels of recursion till n reaches 2.
- Each recursion contributes a constant c to the runtime irrespective of its level j.
- Thus the total time complexity is denoted by $clog_2 n$ = O(log n)

4 References

1. https://www.cs.cornell.edu/courses/cs211/2006sp/Lectures/L06-Induction/binary_search.ht

2. https://www.geeksforgeeks.org/advanced-master-theorem-for-divide-and-conquer-recurrences/

3. Lecture Notes by Prof. Alin Dobra

4. Algorithm Design - Jon Kleinberg, Eva Tardos

5. https://graal.ens-lyon.fr/ abenoit/algo09/coins2.pdf