ESO207A Theoretical Assignment-1

Dhruv - 210338

September 2023

Question 1. Ideal profits
In the X world, companies have a hierarchical structure to form a large binary tree network (can be assumed to be a perfect binary tree). Thus every company has two sub companies as their children with the root as company X. The total number of companies in the structure is N. The wealth of each company follow the same general trend and doubles after every month. Also after every year, half of the wealth is distributed to the two child companies (i.e. one fourth to each) if they exist (i.e. the leaf node companies do not distribute their wealth). You can assume that at the end of the year, the month (doubling) operation happens before the year (distribution) operation. Also the sharing operation at the end of the year happens simultaneously for all the companies. Given the initial wealth of each of the N companies, you want to determine the final wealth of each company after m months. (A perfect binary tree is a special tree such that all leaf nodes are at the maximum depth of the tree, and the tree is completely filled with no gaps.)

(a) (20 points) Design an algorithm in $O(n^3 log(m))$ complexity to find the final wealth of each company after m months.

<u>*Pseudo Code*</u>

Algorithm1: *LevelOrderTraversal(Node, LevelOrder)*

Data: *Node* represents current node, *LevelOrder* array stores traversal
Result: Fills the Level Order traversal array
$queue$
$queue.push(root)$
while $!queue.empty()$
  $currNode \leftarrow queue.top()$
  $queue.pop()$

$LevelOrder.push(node.value)$
if $(currNode \rightarrow left)\,! = NULL$ **then**
$queue.push(currNode \rightarrow left)$
end if
if $(currNode \rightarrow right)\,! = NULL$ **then**
$queue.push(currNode \rightarrow right)$
end if
end while

Algorithm2: *MatrixForm(matrix, n)*

Result: Fills the matrix
for $i:\ 0 \rightarrow n-1$
if $(i >= (n-1)/2)$ **then**
$matrix[i][i] = 2^{12}$
end if
else
$matix[i][i] = 2^{11}$
end else
end for
$i \leftarrow 1$
$j \leftarrow 0$
while $i < n$
$matrix[i][j] = 2^{10}$
$matrix[i+1][j] = 2^{10}$
$i+ = 2$
$j+ = 1$
end while

Algorithm3: *MatrixMultiply($matrix_1, matrix_2$)*

Result: Return $matrix_1 * matrix_2$
$row1 = matrix_1.size()$
$col1 = matrix_1[0].size()$
$row2 = matrix_2.size()$
$col2 = matrix_2[0].size()$
$result[row1][col2]$
for $i:\ 0 \rightarrow row1-1$
for $j:\ 0 \rightarrow col2-1$
$result[i][j] = 0$
for $k:\ 0 \rightarrow row2-1$
$result[i][j]+ = result[i][k] * result[k][j]$

        end for
    end for
end for
return *result*

Algorithm4: *MatrixExponentiation(matrix, n, m)*

Result: Return $matrix^m$
$temp[n][n] \leftarrow matrix$
$MatrixExponentiation(matrix, m/2)$
$MatrixMultiply(matrix, matrix)$
if $m\%\ 2 == 1$ **then**
    $MatrixMultiply(matrix, temp)$
end if
return *matrix*

Algorithm5: *FinalWealth(X, intialWealth, n, m)*

Result: Return $matrix^n$
$levelOrder = LevelOrderTraversal(X, levelOrder)$
$matrix = MatrixForm(matrix, n)$
$finalWealth$
$finalWealth$
$years = m/12$
$months = m\%12$
$matrix = MatrixExponentiation(matrix, 12 * years)$
$finalWealth = MatrixMultiply(matrix, initialWealth)$
for $i : 0 \rightarrow n - 1$
    $finalWealth[i] = finalWealth[i] * 2^{months}$
end for
return *finalWealth*

(b) (10 points) Analyze the time complexity of your algorithm and briefly argue about the correctness of your solution.

Time Complexity Analysis

- *LevelOrderTraversal*, each of the n node is processed in queue once

$$LevelOrderTraversal \Rightarrow \mathbf{O(n)}$$

- *MatrixForm* have two for loops, n iteration each with each iteration having $O(1)$ operations

$$MatrixForm \Rightarrow \mathbf{O(n)}$$

- *MatrixMultiply* have three for loops with row1, col2, and row2 iterations respectively with each iteration having $O(1)$ operations

$$MatrixMultiply \Rightarrow \mathbf{O(row1*col2*row2)}$$

- *MatrixExponentiation*, m halfs every function call

$$T(m) = MatrixMultiply + T(m/2)$$

$$T(m) = 2MatrixMultiply + T(m/2^2)$$

$$\vdots$$

$$T(m) = MatrixMultiply * log_2(m)$$

$$Let\ rows\ and\ col\ of\ order\ n$$

$$MatrixExponentiation \Rightarrow \mathbf{O(}n^3 log(m)\mathbf{)}$$

- *FinalWealth*, call matrix exponentiation for $(matrix, m)$, matrix multiply with row1, col2, row2 = n. Using above functions analysis

$$FinalWealth \Rightarrow O(1) + O(n) + O(n) + O(n^3 log(m)) = \mathbf{O(}n^3 log(m)\mathbf{)}$$

Proof Of Correctness

- We use the recurrence relation $W_{i,k+1} = 2^{11} W_{i,k} + 2^{10} Wi/2, k$ where $W_{i,k}$ gives wealth of Node i after k years using this recurrence we create the matrix and and find all the wealths

(c) (10 points) Consider the case of a single company (i.e. only root) in the tree. Give a constant time solution to find the final wealth after m months.

Analysis Since there is only one company i.e. root we dont worry about distributing its wealth at the end of 12 months.

$$Wealth\ at\ end\ of\ m\ months = 2^m$$

$$For\ Constant\ time\ use\ bit\ shift\ operator \Rightarrow 1 << m$$

Question 2. Moody Friends

P friends arrive at a hotel after a long journey and want rooms for a night. This hotel has n rooms linearly arranged in form of an array from left to right where array values depict the capacities of the rooms. As these are very close friends they will only consider consecutive rooms for staying. As you are the manager of the hotel you are required to find cheapest room allocation possible for them (sum of the capacities of selected rooms should be greater than or equal to P). Cost of booking every room is same and is equal to C.

(a) (15 points) Design an algorithm in $O(n)$ time complexity for determining the minimum cost room allocation. The allocated rooms should be consecutive in the array and their capacities should sum to at least P . You will get 5 points if you design an $O(nlogn)$ time algorithm.

Algorithm Analysis

- Approach used - *two pointers* and *greedy*
- Two pointers *start* and *end* are maintained, where current room allocation is [rooms[start], rooms[end]] giving $currentprice = C(end - start + 1)$
- *start* is moved forward and while the total friends alloted are greater than P, *end* is increased reducing the cost
- Minimum cost is updated accordingly whenever total friends alloted is greater than P

Pseudo Code

Algorithm: *MinCost(Capacity, n, P, C)*

Data: *Capacity* array depicting the capacities of the n rooms, P is number of friends, and C is cost of each room

Result: Returns the *minimum* cost to allocate rooms to all the friends

$front \leftarrow 0$ `// Denotes start index of room allocation`
$back \leftarrow 0$ `// Denotes end index of room allocation`
$currCost \leftarrow 0$ `// Stores cost of current room allocation`
$friendsAlloted \leftarrow 0$ `// Current number of friends who got a room`
$minCost \leftarrow \infty$ `// Overall minimum cost of room allocation`
while $end < n$ **do**

$friendsAlloted += capacity[end]$
$currCost += C$
while $friendsAlloted - capacity[start] >= P$ **do**
  $friendsAlloted -= capacity[start]$
  $currCost -= C$
  $start++$
if $friendsAlloted >= P$ **then**
  $minCost = min(minCost, currCost)$
$end++$
return $minCost$

Time Complexity Analysis

- $Variables$ declaration $O(1)$
  Each iteration of while loop $O(1)$
  Number of iterations of *while* loop ($end\ 0 \to n-1$) $O(n)$

$$TimeComplexity = O(1) + O(n) * O(1) = \mathbf{O(n)}$$

(b) (15 points) Now suppose they don't care about the cost and total capacity anymore. But they came up with a beauty criteria for an allocation. According to them, an allocation is beautiful if $GCD(Greatest\ CommonDivisor)$ of capacities of all rooms in the allocation is at least equal to or greater than a constant K. And they want to take maximum number of contiguous rooms possible. Your task is to design an algorithm in $O(nlog(n))$ time complexity for determining the maximum number of contiguous rooms they can get which satisfy the beauty constraints. You can assume access to a blackbox GCD algorithm which can give you GCD of two numbers in constant $O(1)$ time.

Algorithm Analysis

- Contiguous room allocation is identical to a subarray, and Capacity array will be used as arr for further discussion

- Consider subarray starting at i_{th} index, and the end index j in arr. Increasing j decreases the $GCD(arr[i], arr[i+1], \ldots arr[j])$ since

$$GCD(a, b, c) <= GCD(a, b) <= a, b$$

- Since $GCD(arr[i], arr[i+1], \ldots arr[j])$ is a montonically decreasing with fixed index i_{th}, we can find maximum j with $GCD(arr[i] \rightarrow arr[j]) >= K$ using **binary search**
- To find $GCD(arr[i] \rightarrow arr[j])$ in $log(n)$ we use sparse table or range minima data structure which is precomputed as taught in class
- Finally interate over each i and find j_{max} to get continuous room allocation and store the $max(j_{max} - i + 1)$ as answer

Pseudo Code

Algorithm1: *GcdSparseTable(Capacity, n)*

Data: *Capacity* array depicting the capacities of the n rooms
Result: Returns a matrix *sparseGcd* for calculating range gcd queries

```
k ← ⌊log2 n⌋                    // Denotes start index of room allocation
back ← 0                         // Denotes end index of room allocation
sparseGcd[n][k + 1]              // Stores cost of current room allocation
for i : 0 → n − 1 do
    sparseGcd[i][0] = Capacity[i]
end for
for i : 1 → k do
    for j = 0; j + (1 << i) <= n; j + + do
        sparseGcd[j][i] = GCD(sparseGcd[j][i − 1],
                              sparseGcd[j + (1 << (i − 1))][i − 1]);
    end for
end for
return sparseGcd
```

Algorithm2: *LogTable(n)*

Data: n is the total number of rooms
Result: Returns a log_2 array

```
lg[n + 1]                        // Denotes start index of room allocation
lg[1] = 0                        // Denotes end index of room allocation
for i : 2 → n do
    lg[i] = lg[⌊i/2⌋] + 1
end for
return lg
```

Algorithm3: *RangeGcd(sparseGcd, lg, i, j)*

Data: *sparseGcd, lg* are same as discussed above, i, and j are start and end index

Result: Returns GCD of $Capacity$ array in range [i, j]
$dis \leftarrow lg[j-i+1]$ `// Denotes start index of room allocation`
$ans \leftarrow GCD(sparseGcd[i][dis],\ sparseGcd[j-(1<<dis)+1][dis])$
return ans

Algorithm4: *MaxContRooms(Capacity, n, K)*

Data: $Capacity$ stores capacity of n rooms and K is given constant
Result: Returns maximum count of continuous rooms with $GCD >= K$
$sparseGcd \leftarrow GcdSparseTable(Capacity, n)$
$lg \leftarrow LogTable(n)$
$maxCount \leftarrow 0$
for $idx : 0 \rightarrow n-1$ **do**
  $low \leftarrow idx$
  $high \leftarrow n-1$ **while** $low <= high$ **do**
    $mid \leftarrow (low + high)/2$
    if $RangeGcd(sparseGcd,\ lg,\ idx,\ mid) >= K$
      $low = mid$
    else
      $high = mid - 1$
  end while
  $maxCount = max(maxCount,\ lo - idx + 1)$
end for
return $maxCount$

Time Complexity Analysis

- *GcdSparseTable* is same as range minima problem, where $min \rightarrow GCD$. Since GCD is calculated in constant time it is no different than range minimum data structure.
  for loop 1 runs n times and nested for loop 2 runs $log_2 n$ times

$$GcdSparseTable \Rightarrow \mathbf{O(nlog(n))}$$

- *LogTable* contain array initialization oof $O(n)$ and a while loop of $O(n)$

$$LogTable \Rightarrow \mathbf{O(n)}$$

- *RangeGCd* uses GcdTable and LogTable to return range GCD in constant time

$$RangeGcd \Rightarrow \mathbf{O(1)}$$

- *MaxContRooms* precomputes the sparseGcd in $nlog(n)$ and lg in $O(n)$. for loop runs n times, nested $while$ loop runs for $log(n)$ times where each time it does constant number of operations

$$MaxContRooms \Rightarrow O(nlogn) + O(n) + O(nlog(n)) = \mathbf{O(nlog(n))}$$

(c) (10 points) Give proof of correctness and time complexity analysis of your approach for part (a).

Proof of Correctness

- **Assertion1:** Each room allocation i.e subarray get covered in the algorithm
- **Assertion2:** minCost decreases
- **Proof1:** At each *end* index, after updating *start* index to $start_{end}$ and taking *cost* corresponding to this room allocation i.e $[room[start_{end}], room[end]]$ we cover minimum *cost* of all room allocation ending at index *end* beacuse *start* is increased keeping $friendsAlloted >= P$ thus decresing the cost. This gives us the minimum cost among all continuous room allocation ending at $room[end]$
- **Proof2:** Since we are iterating over index *end* and checking the cost at each index *end* and updating the minCost i.e answer accordingly, we get minimum cost among all the room allocations possible

Question 3. BST universe
You live in a BST world where people are crazy about collecting BSTs and trading them for high values. You also love Binary Search Trees and possess a BST. The number of nodes in your BST is n.

(a) (10*points*) The Rival group broke into your lab to steal your BST but you were able to stop them. But still they managed to swap exactly two of the vertices in your BST. Design an $O(n)$ algorithm to find which nodes are swapped and the list of their common ancestors.

Alogrithm Analysis

- We perform inorder traversal to find the sorted nodes but since they were swapped we will find them not in sorted manner. We just apply lesser and greater than condition to find the swapped nodes.
- To find the common ancestors we first find the path of each node from root node and then compare the path.

Pseudo Code

Algorithm1: *InorderTraversal(Node, Inorder)*

Data: *Node* represents a node of BST, Inorder array stores BST traversal
Result: Fills the Inorder traversal array
if $(Node \rightarrow left)\,! = NULL$ **then**
  $InorderTraversal(Node \rightarrow left, Inorder)$
end if
$Inorder.add(Node.value)$
if $(Node \rightarrow right)\,! = NULL$ **then**
  $InorderTraversal(Node \rightarrow right, Inorder)$
end if

Algorithm2: *ParentPath(parentNode, target, Ancestors)*

Data: $parentNode$ is current or starting node, $target$ is the node value whose path is required, $Ancestors$ array stores ancestors of $targetNode$
Result: Fills the $Ancestors$ array for $target$ starting from $parentNode$
$Ancestors.add(parentNode.value)$
if $parentNode.value == target$ **then**
  $return$
end if
if $ParentNode.value > target$ **then**
  $ParentPath(parentNode \rightarrow left, target, Ancestors)$

end if
else
$ParentPath(parentNode \to right, target, Ancestors)$
end if

Algorithm3: *SwapNodes(root, n)*

Data: *root* node of n nodes BST
Result: Returns which two nodes are swapped
$Inorder = InorderTraversal(root, Inorder)$ $node1, node2$ **for** $i : 0 \to n-2$
if $Inorder[i] > Inorder[i+1]$ **then**
$node1 = i$
$break$
end if
end for
for $i : 1 \to n-1$
if $Inorder[i] < Inorder[i-1]$ **&** $i\ != node1$ **then**
$node2 = i$
$break$
end if
end for
return $Inorder[node1], Inorder[node2]$

Algorithm4: *CommonAncestor(root, node1, node2)*

$Ancestors1 = ParentPath(root, node1, Ancestors1)$
$Ancestors2 = ParentPath(root, node2, Ancestors2)$
$n \leftarrow min(Ancestors1.size(), Ancestors2.size())$
$common$ `// Stores the common ancestors`
for $i : 0 \to n-1$
if $Ancestors1[i] == Ancestors2[i]$ **then**
$common.add(Ancestors1[i])$
end if
end for
return $common$

Time complexity analysis done in second part

(b) (20 points) Seeing you were able to easily revert the damage to your tree, they attacked again and this time managed to rearrange exactly k of your nodes in such a way that none of the k nodes remain at the same position after the rearrangement. Also all the values inside this BST are positive integers and upper bounded by a constant G. Your task is to determine the value of k and which nodes were rearranged. Design an algorithm of complexity $O(min(G + n, nlog(n)))$ for the same. (Hint : Consider two cases for $G < nlog(n)$ and $G > nlog(n)$)

Pseudo Code:
$Case1 : G > nlog(n)$

Algorithm: *RearrangedNodes(Node, n)*

Data: $Root$ is root node of BST with n nodes
Result: Find rearranged nodes
$inorder$
$inorder = InorderTraversal(Root, Inorder)$
$originalOrder \leftarrow copy\,of\,inorder$
$Sort(originalOrder)$
$knodes$
for $i : 0 \rightarrow n - 1$ **do**
  if $originalOrder[i]\,! = inorder[i]$ **then**
    $knodes.add(originalOrder[i])$
  end if
end for
return $knodes.size(), knodes$

$Case2 : G < nlog(n)$

Algorithm: *RearrangedNodes(Node, n)*

Data: $Root$ is root node of BST with n nodes
Result: Find rearranged nodes
$inorder$
$inorder = InorderTraversal(Root, Inorder)$
$index[G + 1] = \{-1\}$
$knodes$
$idx \leftarrow 0$
for $i : 0 \rightarrow n - 1$ **do**
  $index[inorder[i]] = i$
end for
for $i : 1 \rightarrow G$ **do**
  $index[inorder[i]] = i$
  if $index[i]\,! = -1$ **&** $index[i]! = idx$ **then**
    $knodes.add(i)$

end if
if $index[i]\,! = -1$ **then**
$idx++$
end if
end for
return $knodes.size(), knodes$

<u>*Time Complexity Analysis*</u>

- *InorderTraversal:* Let number of operation at i^{th} node, T(i)

$$T(i) = a + T(i \rightarrow left) + T(i \rightarrow right)$$

where a is constant.
Let root $node = n$, where is total number of nodes in BST

$$T(n) = a + T(n \rightarrow left) + T(n \rightarrow right)$$

$$T(n) = a+(a+T(n-1 \rightarrow left)+T(n-1 \rightarrow right))+(a+T(n-2 \rightarrow left)+T(n-2 \rightarrow right))$$

$$\vdots$$

$$T(n) = a + a + \ldots + a + a = a * n$$

$$InorderTraversal \Rightarrow \mathbf{O(n)}$$

- *ParentPath:* Same as InorderTraversal but we stop when we reach the required node

$$T(x) = a + a + \ldots + a + a = depth(x) * a$$

$$ParentPath(x) \Rightarrow \mathbf{O(d)}$$

- *Case1: G > nlog(n)*

$Inorder$ array fill	$O(n)$
$originalOrder$ array initialization	$O(n)$
$sort(originalOrder)$	$O(nlog(n))$
for loop $(0 \rightarrow n-1)$	$O(n)$

$$Case1 :\Rightarrow \mathbf{O(nlogn)}$$

- *Case2: G < nlog(n)*
  $Inorder$ array fill $O(n)$
  $index$ array initialization $O(G)$
  $index$ array update accordingly (for loop) $O(n)$
  find $knodes$ array using $index$ and $inorder$ (for loop) $O(G)$

$$Case2 :\Rightarrow 2O(n) + 2O(G) = \mathbf{O(G + n)}$$

$$min(Case1, Case2) = min(G + n, nlog(n))$$

$$Time\,Complexity \Rightarrow \mathbf{O(min(G + n,\ nlog(n)))}$$

Question 4. Helping Joker

Joker was challenged by his master to solve a puzzle. His master showed him a deck of n cards. Each card has value written on it. Master announced that all the cards are indexed from 1 to n from top to bottom such that ($a_1 < a_2 < a_3 < a_4 < \ldots < a_n$) .Then his master performed an operation on this deck invisible to Joker (Joker was not able to see what he did), he picked a random number k between 0 and n and shifted the top k cards to the bottom of the deck. So after the operation arrangement of cards from top to bottom looks like $(a_{k+1}, a_{k+2}, \ldots a_n, a_1, a_2, \ldots a_k)$ where $(k+1, k+2 \ldots n, 1, 2 \ldots k)$ are original indices in the sorted deck. Joker's task is to determine the value of k. Joker can make a query to his master. In a query, joker can ask to look at the value of any card in the deck. Joker asked you for help because he knew you were taking an algorithms course this semester.

(a) (15 points) Design an algorithm of complexity $O(log(n))$ for Joker to find the value of k.

Algorithm Analysis

- Deck is increasing everywhere else at a index where we see decrease in value. Problem converts to finding index i such that $deck[i] > deck[i+1]$
- We use binary search to find the kth index

Pseudo Code

Algorithm1: *findK(Deck)*

Data: *Deck* array having value of all cards (0 based indexing)
Result: *R*eturns the value of k about which *Deck* is rotated

$low \leftarrow 0$ `// Start index for palindrome check`
$high \leftarrow n - 1$ `// End index for palindrome check`
while $low < high$ **do**
  $mid \leftarrow (low + high)/2$ `// Mid index for binary search`
  if $Deck[mid] >= Deck[0]$ **then**
    $low = mid$
  else
    $high = mid - 1$
end while
return $n - low$

(b) (5 points) Provide time complexity analysis for your strategy.

Time Complexity Analysis

1. *low* and *high* variable initialization $O(1)$
2. Each iteration of $whileloop$ $O(1)$
3. Number of iterations of *while* loop -
   After each iteration size of search space becomes half
   Size after first iteration $n/2$
   Size after second iteration $n/4$
   $\vdots$
   Size after k iterations $n/2^k$
   Let size after k iteration be 1 i.e when low = high
   $n/2^k = 1 \Rightarrow k = log_2 n$
   Total number of iterations = k
4. Total Time Complexity = $O(1) + O(1) * O(log_2 n) = O(log_2 n)$

Question 5. One Piece Treasure

Strawhat Luffy and his crew got lost while searching for One piece (worlds largest known treasure). It turns out he is trapped by his rival Blackbeard. In order to get out he just needs to solve a simple problem. You being the smartest on his crew are summoned to help. On the gate out, you get to know about a hidden string of lowercase english alphabets of length n. Also, an oracle is provided which accepts an input query of format (i, j), and returns true if the $substring(i, j)$ of the hidden string is a palindrome and false otherwise in $O(1)$ time. But there is a catch, the place will collapse killing all the crew members, if you ask any more than $nlog^2(n)$ queries to the oracle. The string is hidden and you can't access it. (Assume the string is very big i.e. n is a large number)

(a) (30 points) You need to design a strategy to find number of palindromic substrings in the hidden string so your crew can safely escape from this region. Please state your algorithm clearly with pseudocode. (A contiguous portion of the string is called a substring)

Algorithm Analysis

- Palindromes of odd length with middle element at index i will have a len_{max} such that $string[i-len_{max}] \rightarrow string[i+len_{max}]$ is palindrome. Every len between $[0, len_{max}]$ will form palindrome ($string[i-len] \rightarrow string[i+len]$) with i^{th} index as middle element. Any $len > len_{max}$ will not yeild palindrome with i which means effect of len on palindrome check is monotonic or predicate function. Therefore **Binary Search** can be used to find len_{max}

- Similar approach can be used for palindromes of even length. Palindrome for len_{max} with i^{th} index as first of the two middle element will be $string[i - len_{max} + 1] \rightarrow string[i + len_{max}]$. Binary Search can be used to find the len_{max}

Pseudo Code

Algorithm1: *PalindromeCheck(i, j)*

Data: *Substring* from index $[i, j]$
if $substring[i, j]$ *is palindrome* **return** *true* `/* Blackbox does`
else return $false$ `palindrome check*/`

Algorithm2: *totalPalindromeCount(n)*

Data: n is length of hidden string

Result: Returns the count of palindromic substrings in hidden string

$idx \leftarrow 0$ `// Index of hidden string`
$countOdd \leftarrow 0$ `// Count of palindromic substrings`
for $idx: 0 \rightarrow n-1; idx++$ **do**
  $low \leftarrow 0$ `// Start index for palindrome check`
  $high \leftarrow n - idx - 1$ `// End index for palindrome check`
  while $low < high$ **do**
    $mid \leftarrow (low + high)/2$ `// Mid index for binary search`
    if $idx >= mid$ **&** $PalindromeCheck(idx - mid, idx + mid) == true$
      $low = mid$
    else
      $high = mid - 1$
  end while
  $countOdd += low + 1$
end for
$idx = 0$ `// Index of hidden string`
$countEven \leftarrow 0$ `// Count of palindromic substrings`
for $idx: 0 \rightarrow n-1; idx++$ **do**
  $low \leftarrow 0$
  $high \leftarrow idx$
  while $low < high$ **do**
    $mid \leftarrow (low + high)/2$ `// Check substring[idx-mid, idx+mid+1]`
    if $(idx+mid+1) < n$ **&** $PalindromeCheck(idx-mid, idx+mid+1)$
      $low = mid$
    else
      $high = mid - 1$
  end while
  $countEven += low$
end for
return $countOdd + countEven$

Time Complexity Analysis

- for loop run for n times, where each iteration have $2 + 3log(n)$ operation. This process is done twice (odd and even length palindrome)

$$Time\,Complexity \Rightarrow \mathbf{O(nlog(n))}$$

$$Number\,of\,queires\,asked \Rightarrow \mathbf{2log(n)} < nlog^2(n)$$