ESO207 Theoretical Assignment 1

P1 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). 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. Detailed explanation here

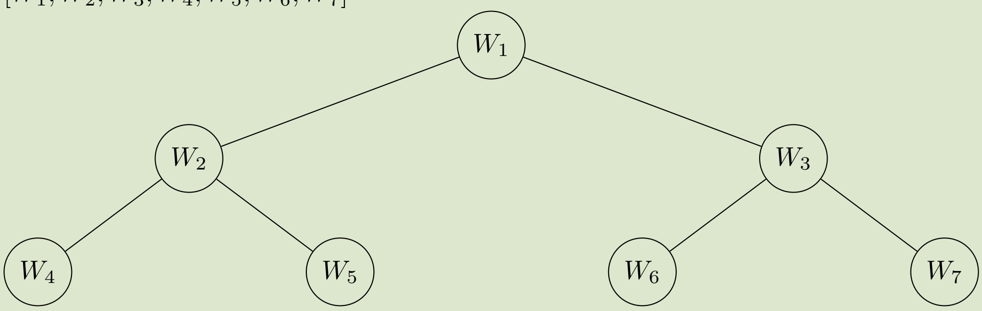
(a) Design an algorithm in $O(n^3 \log(m))$ complexity to find the final wealth of each company after m months.

`Motivations:`

- To get easier relation between parent and child company index → Level-Order-Transversal
- $n^3 log(m)$ factor suggests fast matrix exponentiation/multiplication similar to the Clever-Fib-Algo discussed in lectures
- We can easily form a recurrence relation between Wealth after $(k-1)$ years and k years, thus suggesting matrix multiplication again

---

First, we do a Level-Order-Traversal of the binary tree and store it in an array.
For example, Level-Order-Traversal of the following tree gives the array $[W_1, W_2, W_3, W_4, W_5, W_6, W_7]$



This allows us to find a relation between parent and child companies easily viz. $index(parent(j)) = \lfloor \frac{j}{2} \rfloor$ We can keep track of the wealth of all companies after k years as a $n \times 1$ matrix $W_k = [W_{1,k}\ W_{2,k}\ W_{3,k}\ \cdots\ W_{n,k}]^T$
We can represent the relation between W_k and W_{k-1} as follows:

$$\overset{W_k}{\begin{bmatrix} W_{1,k} \\ W_{2,k} \\ W_{3,k} \\ \vdots \\ W_{n,k} \end{bmatrix}} = 2^{12} \overbrace{\begin{bmatrix} \frac{1}{2} & 0 & \cdots & 0 \\ \frac{1}{4} & \frac{1}{2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}}^{A} \overset{W_{k-1}}{\begin{bmatrix} W_{1,k-1} \\ W_{2,k-1} \\ W_{3,k-1} \\ \vdots \\ W_{n,k-1} \end{bmatrix}}$$

Here, the matrix A is constructed by setting $A[i][i] = 2^{11}$ and $A[i][\lfloor \frac{i}{2} \rfloor] = 2^{10}$ since after an year $W_{j,k} = 2^{12} \left(\frac{W_{\text{parent}(j),k-1}}{4} + \frac{W_{j,k-1}}{2} \right)$

So, to find the wealth of companies after m months, We first calculate $W_{j,\lfloor \frac{m}{12} \rfloor}$ and then the final answer $W'_{j,m} = 2^{m-12\lfloor \frac{m}{12} \rfloor} W_{j,\lfloor \frac{m}{12} \rfloor}$
We accomplish this by using Matrix Exponentiation for finding powers of matrix A
Finally, $W'_{j,m} = 2^{m-12\lfloor \frac{m}{12} \rfloor} A^{\lfloor \frac{m}{12} \rfloor} W_{j,0}$

`Pseudocode:`

```
function GET-WEALTH(BT, m)
    ogwealth ← LEVEL-ORDER-TRAVERSAL(BT)
    n ← number of companies
    A ← [[0]]_{n×n}
    for i from 0 to n − 1 do
        A[i][i] ← 1/2 * 2^12
        if i ≥ 2 then
            A[i][⌊i/2⌋] ← 1/4 * 2^12
        end if
        if i > ⌊n/2⌋ then
            A[i][i] ← 1 * 2^12
        end if
    end for
    A' ←MATPOW(n, A, ⌊m/12⌋)
    H ←MATPOW(n, A', ogwealth)
    W' ← [0]_m
    for i from 0 to n − 1 do
        W'[i] ← W'[i] * 2^(m mod 12)
    end for
    return W'
```

....

```
function LEVEL-ORDER-TRAVERSAL(root)
    traversed ← []
    if root is null then
        return
    end if
    queue ← Queue()                    ▷ Create an empty queue and enqueue the root node
    queue.enqueue(root)
    while not queue.isEmpty() do
        current_node ← queue.dequeue()
        traversed.APPEND(current_node.value)
        if current_node.left is not null then
            queue.enqueue(current_node.left)
        end if
        if current_node.right is not null then
            queue.enqueue(current_node.right)
        end if
    end while
    return traversed
```

....

```
function KKK1(k, m1, m2)               ▷ For matrix multiplication (k × k) · (k × 1)
    m3 ← [0 × k]
    for i from 0 to k − 1 do
        for j from 0 to k − 1 do
            m3[i] ← m3[i] + (m1[i][j] * m2[j])
        end for
    end for
```

```
    return m3
.......................................................................
function KKKK(k, m1, m2)            ▷ For matrix multiplication (k × k) · (k × k)
    m3 ← [[0]]_{k×k}
    for i from 0 to k − 1 do
        for j from 0 to k − 1 do
            m3[i][j] ← 0
            for m from 0 to k − 1 do
                m3[i][j] ← m3[i][j] + (m1[i][m] ∗ m2[m][j])
            end for
        end for
    end for
    return m3
.......................................................................
function MATPOW(k, base, pow)                    ▷ For matrix exponentiation
    mull ← [[0]]_{k×k}
    if pow = 1 then
        for i from 0 to k − 1 do
            for j from 0 to k − 1 do
                mull[i][j] ← base[i][j]
            end for
        end for
        return mull
    end if
    temp ← MATPOW(k, base, pow/2)
    mull ← KKKK(k, temp, temp)                                      ▷ O(n^3)
    if pow mod 2 = 1 then
        mull ← KKKK(k, mull, base)
    end if
    return mull
```

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

`Time Complexity Analysis :`

```
function GET-WEALTH(BT, m)
    ogwealth ← LEVEL-ORDER-TRAVERSAL(BT)    ▷ Since each node is visited exactly once,
O(n)
    n ← number of companies
    A ← [[0]]_{n×n}
    for i from 0 to n − 1 do
        A[i][i] ← 1/2 ∗ 2^12
        if i ≥ 2 then
            A[i][⌊i/2⌋] ← 1/4 ∗ 2^12
        end if
        if i > ⌊n/2⌋ then
            A[i][i] ← 1 ∗ 2^12
        end if
    end for
    A′ ← MATPOW(n, A, ⌊m/12⌋)                                  ▷ O(n^3 log(m))
    H ← MATPOW(n, A′, ogwealth)                                      ▷ O(n^3)
```

$W' \leftarrow [0]_m$
for i from 0 to $n-1$ **do**
$W'[i] \leftarrow W'[i] * 2^{(m \mod 12)}$
end for
return W'

So, the overall time complexity of GET-WEALTH(BT, m) ia $T(n) = O(n) + O(1) + O(n) + O(n^3 log(m)) + O(n^3) + O(n) = O(n^3 log(m))$ (being the dominant term)

The function MATPOW$(k, base, pow/2)$ gets called for $log(m)$ times i.e. until $pow/2$ reaches 1 and inside each call of the function, we have the function KKKK$(k, temp, temp)$ being called for matrix multiplication, and it has three nested loops contributing $O(n^3)$ to the time complexity. Thus overall time complexity contribution of MATPOW is $O(n^3 \times log(m))$

`Correctness :`

We make the assertion that at end of $k-1$ years, we know the wealth of each company as $W_{k-1} = [W_{1,k-1}\ W_{2,k-1}\ W_{3,k-1}\ \cdots\ W_{n,k-1}]^T$. Since after each month, the wealth doubles, so after an year we have $W_{j,k} = 2^{12}W_{j,k-1}$ (before distribution). Now, we distribute the wealth such that $W_{j,k} = 2^{12}\left(\frac{W_{\text{parent}(j),k-1}}{4} + \frac{W_{j,k-1}}{2}\right)$ since wealth of the company halves and then it inherits $1/4^{th}$ from its parent company. (Except for the companies at the bottom-most level of tree whose wealth doesn't halve) Now, to account for m months, we have $k = \lfloor \frac{m}{12} \rfloor$ and the wealths of companies doubles for the leftover $m \mod 12$ months, So finally we have $W'_{j,m} = 2^{(m \mod 12)}W_{j,\lfloor \frac{m}{12} \rfloor}$

(c) 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.

Considering the initial wealth to be W_0, since the root itself is the only company, it's wealth does not get distributed after every year.
$\Longrightarrow$ The company's wealth after m months, $W_m = 2^m W_0$
We can utilise binary-shift operators for getting an $O(1)$ solution

```
W_j,m = (W_j,0 << m)
```

P2 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) 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 atleast P.

`Motivations:`

- $O(n)$ suggests that we may need to process each element atmost a constant number of times
- This problems draws heavy similarity from the minimum sum subarray problem

---

We can keep two pointers to keep track of the two ends of the subarray. Now, while we traverse the original array from left to right, at each step we check whether the sum achieved is greater than or equal to P, if yes then we remove the left most element of subarray (incrementing left pointer by 1) and continue the search for an even smaller length, and if not then we add take more elements in the subarray (incrementing right pointer by 1) and repeat this again.

`Pseudocode:`

```
function GET-MINIMUM-COST(rooms)
    l ← 0                                   ⎫
    cost ← inf                              ⎬ O(1)
    cap ← 0                                 ⎭
    for i ← 0 to n − 1 do                   ⎫
        cap ← cap + rooms[i]                ⎪
        while cap >= P do                   ⎪
            cost ← min(cost, i + 1 − l)     ⎬ O(n)
            cap ← cap − rooms[l]            ⎪
            l ← l + 1                       ⎪
        end while                           ⎪
    end for                                 ⎭
    return (cost ∗ C)
```

(b) 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 Common Divisor) 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.

`Motivations:`

- We can translate this problem as finding the maximum length of subarray such that "range-GCD" satisfies the given constraint
- Thus, thinking similar on the lines of Range-Minima problem, we construct a new data structure of size $nlog(n)$
- We can effectively divide the problem in two parts:
  1. Efficiently finding the range-GCD of a given range
  2. Efficiently finding the maximal length of such valid subarray

---

We can use an $n \times log(n)$ matrix S where $S[i][j]$ stores GCD of the range $A[i], A[i+1], ..., A[i+2^j]$ where A is the original array. We can then binary search on the length of subarray such that range-GCD $>= K$ (Since if range-GCD $>= K$ for a subarray of length m then it is valid for all lengths $<= m$ while after a maximal length, it won't be valid for all lengths $>$ maximal_length)

`Pseudocode:`

function BUILD-MATRIX($rooms$)
  $n \leftarrow$ number of rooms
  $log \leftarrow [0]_{n+1}$
  for $i \leftarrow 2$ **to** $n+1$ **do**
    $log[i] \leftarrow log[\lfloor \frac{i}{2} \rfloor] + 1$
  end for
  $st \leftarrow [[0]]_{n\times(log[n]+1)}$
  for $i \leftarrow 0$ **to** n **do**
    $st[i][0] \leftarrow rooms[i]$
  end for
  for $j \leftarrow 1$ **to** $log[n]+1$ **do**
    for $i \leftarrow 0$ **to** $n - 2^j + 1$ **do**
      $st[i][j] \leftarrow$ GCD$(st[i][j-1], st[i+2^{j-1}][j-1])$
    end for
  end for
  return st

....

function GET-RANGE-GCD(st, L, R)
  $k \leftarrow$ largest power of 2 such that $2^k \leq R - L + 1$
  return GCD$(st[L][k], st[R-2^k+1][k])$

....

function CHECK-LENGTH($arr, K, length$)
  $n \leftarrow$ number of rooms
  for $i \leftarrow 0$ **to** $n - length$ **do**
    if GET-RANGE-GCD$(st, i, i + length - 1) \geq K$ **then**
      return true
    end if
  end for
  return false

....

function GET-MAX-LENGTH($rooms$)
  $n \leftarrow$ length of arr
  $l \leftarrow 0$
  $r \leftarrow n$
  $maxLen \leftarrow -1$

while $l \leq r$ **do**
  $mid \leftarrow \lfloor \frac{l+r}{2} \rfloor$
  if CHECK-LENGTH(arr, K, mid) **then**
    $maxLen \leftarrow mid$
    $l \leftarrow mid + 1$
  else
    $r \leftarrow mid - 1$
  end if
end while
return $maxLen$

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

`Proof of Correctness :`

We have two pointers, the range contained by them are the rooms of interest. In each iteration, we check whether the target capacity has been reached or not. If it is not then we take into account the next room and continue this until we reach a point when total capacity $\geq P$, after which we start removing rooms from the left and hence decreasing the length of our subarray.
Our INVARIANT in this case is that at all times the *cost* variable stores the sum of capacities of rooms between the two pointers i.e. it considers a valid subarray at all points of time.
Also, Correctness is assured in both the cases:

- When the target capacity is not reached, we add more rooms to the subarray. This is valid since we are attempting to increase the sum.
- When the target capacity is already reached, we start removing rooms from the left to potentially reduce the subarray length and thus minimize the cost.

In all the possible cases (i.e. with $cap >= P$), our algorithm explores all possible lengths of subarray and then finds the minimum length. Hence, it correctly finds the required minimum cost.

`Time Complexity Analysis:`

Each pointer can visit an element atmost once, and thus all the elements are processed atmost twice (initially adding it to the subarray for reaching target sum and then maybe removing it to minimize the length). Thus total time complexity $T(n) = 3 + n + cn + k = O(n), c, k$ being some constants

P3 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) 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.

`Motivations:`

- In a normal BST, INORDER-TRAVERSAL gives us the sorted nodes
- While searching the nodes in the BST, we can take note of the nodes visited to know about their ancestors

---

If the vertices weren't swapped, then, after traversing the tree using INORDER-TRAVERSAL algorithm, we'd get all the n nodes in a sorted fashion. So, traversing using the same algorithm and then traversing through the sorted list, we can easily detect the two swapped vertices.

`Pseudocode:`

```
traversed ← []
function INORDER-TRAVERSAL(BST, node)    ▷ Takes O(n) time since each node is processed atmost once and processing takes constant time.
    if node <> NULL then
        INORDER-TRAVERSAL(BST, node.left)
        traversed.APPEND(node.value)
        INORDER-TRAVERSAL(BST, node.right)
    end if
    return traversed
function FIND-SWAP(sorted)
    p, q ← NULL
    for i ← 0 to n − 2 do
        if sorted[i] > sorted[i + 1] then
            p ← sorted[i]
            break
        end if
    end for
    for i ← n − 1, 1 do
        if sorted[i − 1] > sorted[i] then
            q ← sorted[i]
            break
        end if
    end for
    return m, n
```

Thus, we obtain m, n (the swapped vertices) in $O(n)$ time. (Since each node gets processed atmost once) Now, to get the list of common ancestors, we can search the vertices m, n in the (original, with vertices not swapped) BST and generate the list of nodes we visit on our way to search them respectively. We do this by searching the swapped vertices in modified BST and then swap them again. We then compare and take the common elements to get our list of common ancestors.

`Pseudocode:`

```
function GET-COMMON-ANCESTORS(m, n, common)
    m_ancestors ← []
    n_ancestors ← []
    common ← []
    SEARCH-N-SWAP(BST, m, n)                 ▷ Search for m and n and then swap them
    SEARCH(BST, m, m_ancestors)
    SEARCH(BST, n, n_ancestors)
    i ← 0
    while m_ancestors[i] = n_ancestors[i] do
        common.APPEND(m_ancestors[i])
function SEARCH(node, target, ancestors)
    if node is NULL then
        return NULL
    end if
    ancestors.APPEND(node.value)
    if node.value = target then
        return node
    end if
    if target < node.value then
        return SEARCH(node.left, target, ancestors)
    else
        return SEARCH(node.right, target, ancestors)
    end if
```

Thus, we obtain the list of all common ancestors in the array *common*

(b) 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 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 $G > nlog(n)$)

`Motivations:`

- We use the hints to the fullest and obviously divide the problem into two cases
- $nlog(n)$ (and even the part(a)) suggests sorting as a possible approach
- We use the same logic for second case
- For the first one, $O(G)$ part of the time complexity suggests some processing on a data structure of size of the order G

We have two cases here:

1. $G < nlog(n)$: Since, we have a cap on the range that the vertices can take, We can construct an array of length $G+1$ which contains the order in which they occur after INORDER-TRAVERSAL of the BST. Now, were the original BST preserved, the non-zero entries of our array would follow the increasing order of natural numbers $1, 2, ...$ i.e. $i+1$ at i'th index. Thus, we check for the indices where $arr[i] \neq (i+1)$ whose count gives us the number of rearranged vertices.
   In this case we have time complexity of $O(G+n)$

`Pseudocode:`

```
function FIND-REARRANGED-NODES-G(BST)
    traversed ← []
    order ← []
    rearranged_nodes ← []
    INORDER-TRAVERSAL(BST, traversed)
    for i ← 0 to n − 1 do                                   ▷ O(n)
        order[traversed[i] + 1] ← i + 1
    end for
    j ← 1, k ← 0
    for i ← 1 to G do                                       ▷ O(G)
        if order[i] <> 0 then
            if order[i] <> j then
                rearranged_nodes.APPEND(order[i])
                k ← k + 1
            end if
            j ← j + 1
        end if
    end for
    return k, rearranged_nodes
```

2. $G > nlog(n)$: In this case, we can traverse the tree using INORDER-TRAVERSAL and then copy and sort the resulting array. Now we check for differences in the sorted and unsorted array, the count of which is exactly the number of rearranged vertices.
   In this case we have time complexity of $O(nlog(n))$

`Pseudocode:`

```
function FIND-REARRANGED-NODES(BST)
    traversed ← []
    INORDER-TRAVERSAL(BST, traversed)
    sorted ← COPY-ARRAY(traversed)
    SORT(sorted)                                            ▷ O(nlog(n))
    rearranged_nodes ← []
    k, i ← 0
    n = LENGTH(traversed)
    while i < n do
        if traversed[i] <> sorted[i] then
            rearranged_nodes.APPEND(traversed[i])
            k ← k + 1
        end if
        i ← i + 1
    end while
    return k, rearranged_nodes
```

P4 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 $(a_1 < a_2 < ... < a_{n1} < a_n)$ .Then his master performed an operation on this 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 $(a_{k+1}, a_{k+2}...a_n, a_1, a_2...a_k)$ where $(k + 1, k + 2...n, 1, 2...k)$ 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) Design an algorithm of complexity $O(log(n))$ for Joker to find the value of k.

`Motivations:`

- $log(n)$ is enough to suggest Binary Search (and also the partially sorted nature of the array)

We can perform binary search on the value of k. We keep the initial range as $[0, n-1]$ and at each step we compare the values of the card at index r and the card at index $mid = \lfloor \frac{l+r}{2} \rfloor$. Now, if

1. $a[mid] > a[r]$: This means that the smallest element is in the second half $[mid, r]$. So now, we set $l = mid + 1$ and continue the search
2. $a[mid] < a[r]$: This means that the largest element lies in the first half $[l, mid]$. So we set $r = mid - 1$ and continue

We stop when the element at mid is larger than that at $mid - 1$ since we have found the smallest element. (We handle the edge case when 0 cards are shifted to the bottom of the dark separately).

`Pseudocode:`

function FINDK(deck, n)
▷ Here, deck[i] means a query to master to get the value of $(i+1)^{th}$ card
    $l \leftarrow 0$
    $r \leftarrow n - 1$
    if deck[l] $\leq$ deck[r] **then**
        return l
    end if
$O(1)$
    while $l \leq r$ **do**
        $mid \leftarrow \lfloor (l + r)/2 \rfloor$
        if deck[mid] < deck[$mid - 1$] **then**
            return $(n - mid)$
        end if
        if deck[mid] > deck[r] **then**
            $l \leftarrow mid + 1$
        else
            $r \leftarrow mid - 1$
$O(log(n))$

(b) Provide time complexity analysis for your strategy.

We know that the initialisation steps take constant time while Binary Search takes $O(log(n))$ time since at every iteration the length of interval reduces by half.
For binary search, we are basically cdeckying out 4-5 operations in each iteration. So the total time $T = \underbrace{5 + 5 + \cdots + 5}_{log_2(n)\text{times}} = 5log_2(n) = O(log(n))$
Why $log_2(n)$ times?: At each iteration, length halves i.e. initial length n reduces to 1 in k iterations, where k can be given by $\frac{n}{2^k} = 1$ or $2^k = n \implies k = log_2(n)$
Assuming the query time to be constant time, We have the final complexity of our algorithm as $O(1) + O(log(n)) + O(1) = O(log(n))$

P5 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) 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)

`Motivations:`

- $n(log(n))^2$ doesn't really suggest anything but once we identify the predicate function here, it leads us to yet another Binary Search problem
- The trick is to deal with odd and even length palindromes differently and then binary search on possible length of palindromes

---

We know that palindromes can be of two types; even-length and odd-length palindromes. We will handle them separately.

For example, we define the center of the palindromes $b, aba, babab$ and radii r, in $a\ \overbrace{b\ a\ \underbrace{\ b\ }_{r=0}\ a\ b}^{r=2}\ c$

Thus, palindrome of length d has radius $r = \lceil \frac{d}{2} \rceil - 1$
Now, we can have n centers of odd-length palindromes (since they can be single letters as well) and $n-1$ centers of even palindromes (counting 2 consecutive elements)
One important observation is that if we have a palindrome of radius r, then all strings with the same center and radii less than r are palindrome i.e. If we define a predicate function IS-PALINDROME(c, r) then we have a series of True's and then a series of False's. For the same reason, we can use binary search on the radii; for even and odd length palindromes differently.
So, for even-length palindromes we query on $[c-r, c+r+2]$ (Since, actually the center consists of 2 elements) with radius ranging from 0 to max possible from either end. Similarly, for odd-length, we query on $[c-r, c+r+1]$ for all possible radii.

`Pseudocode:`

▷ Here, IS-PALINDROME(i, j) means a query to oracle to know if $substring(i, j)$ is a palindrome or not

function GET-NUMBER-OF-PALINDROMES($string$)
    $pal \leftarrow 0$
    for $c \leftarrow 0$ **to** $n-1$ **do** ▷ odd-length palindromes
        $lo \leftarrow 0$
        $hi \leftarrow \min(c, n-c-1)$ ▷ To ensure that the radius lies within the string from both ends
        while $lo < hi$ **do**
            $r \leftarrow \lfloor \frac{l+r+1}{2} \rfloor$
            if IS-PALINDROME$(c-r, c+r+1)$ **then**
                $lo \leftarrow r$

else
$hi = r - 1$
end if
$pal \leftarrow pal + l + 1$ ▷ Since we assumed radius of single element to be 0
end while
end for
for $c \leftarrow 0$ **to** $n - 2$ **do** ▷ even-length palindromes
$lo \leftarrow -1$ ▷ To include the case that two consecutive letters need not be palindrome
$hi \leftarrow \min(c, n - c - 2)$
while $lo < hi$ **do**
$r \leftarrow \lfloor \frac{l+r+1}{2} \rfloor$
if IS-PALINDROME$(c - r, c + r + 2)$ **then**
$lo \leftarrow r$
else
$hi = r - 1$
end if
$pal \leftarrow pal + l + 1$ ▷ Since we assumed radius of empty string to be -1
end while
end for
return pal

Now, pal gives us the total count of palindromes in the hidden string.
Binary search requires $log(n)$ time complexity, so total complexity of the above algorithm becomes $O(nlog(n)) < O(n(log(n))^2)$ and hence the place doesn't collapse!