CSE 321 Homework-2

Bahadır Etka Kılınç
1901042701
[redacted]

1. $T(n) = a \cdot T\left(\frac{n}{b}\right) + \Theta(n^k \cdot \log^p n)$

a = number of subproblems in the recursion and $a \geq 1$

$\frac{n}{b}$ = size of each problem.

$b > 1$, $k \geq 0$ and p is a real number.

if $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

if $a = b^k$, then
- if $p > -1$, then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
- if $p = -1$, then $T(n) = \Theta(n^{\log_b a} \log\log n)$
- if $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

if $a < b^k$, then
- if $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$
- if $p < 0$, then $T(n) = \Theta(n^k)$

a. $a=2$, $b=4$; $(n\log n)^{\frac{1}{2}} = n^{\frac{1}{2}} \cdot \log^{\frac{1}{2}} n$; $k = \frac{1}{2}$, $p = \frac{1}{2}$

$\Rightarrow 2 = 4^{\frac{1}{2}}$, so

$$T(n) = \Theta(n^{\log_b a} \log^{p+1} n) = \Theta(n^{\frac{1}{2}} \cdot \log^{\frac{3}{2}} n)$$

b. $a=9$, $b=3$; $5n^2$; $k=2$, $p=0$

$9 = 3^2 \Rightarrow$ so

$$T(n) = \Theta(n^{\log_b a} \cdot \log^{p+1} n) = \Theta(n^2 \cdot \log n)$$

c. We can not solve this equation with master theorem because $a < 1$.

d. $a=5$, $b=2$, $\log n$; $k=0$, $p=1$

$5 > 2^0 \Rightarrow$

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 5})$$

e. We can not solve this equation with master theorem because the form of equation is wrong.

f. $a=7$, $b=4$, $k=1$, $p=1$

$\frac{a}{7}$ $\frac{b}{4^1} \Rightarrow a > b^k \Rightarrow T(n) = \Theta(n^{\log_a b}) = \Theta(n^{\log_4 7})$

g. $a=2$ $b=3$ $k=-1$, we can not solve this equation with master theorem because $k < 0$

h. $a=2/5$ we can not solve this equation with master theorem because $a < 1$

Bahadır Etka Kılınç
[redacted]

2. Implemented algorithm.

```
void insertionSort( int arr[], int n) {
    int i, key, j;
  for (i=1 ; i<n ; i++) {
        key = arr[i];
       j = i-1
    while (j >=0  && arr[j] > key) {
        arr[j+1] = arr[j] ;
        j = j-1;
      }
   }
}
```

i	j	arr[i]	arr[j]	arr[j+1]	key	arr	
1	0	6	3	6	6	{3,6,2,1,4,5} ⇒	First Iteration
2	1	2	6	2	2	{3,6,6,1,4,5}	
2	0	6	3	6	2	{3,3,6,1,4,5}	
2	1	6	3	6	2	{3,3,6,1,4,5}	Second Iteration
					arr[j+1] = key →	{2,3,6,1,4,5}	
3	2	1	6	1	1	{2,3,6,1,4,5}	
3	1	6	3	6	1	{2,3,6,6,4,5}	
3	0	6	2	3	1	{2,3,3,6,4,5}	Third Iteration
3	-1	6	null	2	1	{2,2,3,6,4,5}	
					arr[j+1] = key →	{1,2,3,6,4,5}	
4	3	4	6	4	4	{1,2,3,6,6,5}	
4	2	6	3	6	4	{1,2,3,6,6,5}	Fourth Iteration
					arr[j+1] = key →	{1,2,3,6,4,5}	
5	4	5	6	5	5	{1,2,3,4,6,6}	
5	3	6	4	6	5	{1,2,3,4,6,6}	Fifth Iteration.
					arr[j+1] = key →	{1,2,3,4,5,6}	

3. a.

Operation	Array	Linked List
Accessing the first element	In array we can use the indexing to access first element. We can use 0 index O(1)	Head will points to the first node so we can easly access the head. & in single operation O(1)
Accessing the last element	We can use the indexing here also length − 1 in index is used to access the last element O(1)	Tail will points to the last node in the linked list. So we can access the last element in single operation O(1)
Accessing any element in the middle	We can use the middle element index to access the element in middle, so it takes single operation only O(1)	Accessing element in the middle, we need to add temporarily current pointer and move to middle from head. So it takes n/2 operations at worst O(n)
Adding a new element at the end	We can use the indexing to set the last index with some value so it takes constant amortized time. O(1)	We can make the tail.Next as new node and then make the new inserted element node as tail. O(1)
Adding new element at the beginning	Adding new element at the beginning takes single operation only. But after addition we need to shift all existing element right by one position O(n)	We can access the head and then we can create the new node. Then assign new node as head O(1)
Adding new element in the middle.	Adding element at middle take single operation only. But after adding we need to shift all indexes after middle to right by one position. O(n)	We need to loop from head to middle and then we need to create a new node and change the links. It takes n operation at worst case O(n)
Deleting the first element	After deleting the first element we need to shift the all the elements from index 1 to length by one to the left side O(n)	We need to change head node only. After changing head node, we need to free the previous head node. O(1)
Deleting the last element	We need to delete or unset the last index value only. There is no shifting needed. O(1)	We need to loop the list from head to last before node of tail and then we need to free the tail node and assign the before node of tail as tail node. O(n)
Deleting any element in the middle	After deleting the element we need to shift the elements in the rightside by one position to left side O(n)	We need to loop the list from head to the middle node and then we need to remove the middle node and then we need to change the links. O(n)

Bahadır Etka Kılınç
[redacted]

3.b. The space complexity of the array is O(n) which is fixed and the space complexity of the linked list at worst case is O(n) and it will decreases during the best case. But we need to consider in worst case so it will takes O(n).

Bahadır Etka
Kılınç
190104701

4.

1. Creating the inorder traversal array of a given binary tree.
2. It sorts the created inorder array
3. It again traverses given binary tree in inorder fashion and simultaneously it traverses sorted inorder array. At each node visit of binary tree, the value of that node is changed to corresponding element in the inorder array. That is value of first node visited during inorder traversal is changed to value of element at 0'th index in sorted inorder array, value of second node visited during inorder traversal is changed to value of element at 1st index so on.

Algorithm

```
void changeTreeToBST() {
    int[] inorder = new int[treeSize];
    int[] index = new int[1];
    createInorderArray(root, inorder, index);
    Arrays.sort(inorder);
    index[0] = 0;
    changeNodeValues(root, inorder, index);
}
void changeNodeValues(TreeNode currentNode, int[] inorder, int[] index) {
    if (currentNode == null) return;
    changeNodeValues(currentNode.left, inorder, index);
    currentNode.value = inorder[index[0]];
    index[0] += 1;
    changeNodeValues(currentNode.right, inorder, index);
}
void createInorderArray(TreeNode currentNode, int[] inorder, int[] index) {
    if(currentNode == null) return;
    createInorderArray(currentNode.left, inorder, index);
    inorder[index[0]] = currentNode.value;
    index[0] += 1;
    createInorderArray(currentNode.right, inorder, index);
}
```

The time complexity of algorithm is O(nlogn) where n is the number of nodes in the tree. This is because we need to do an in-order traversel of the tree, which takes O(n) time, and then we need to sort the array which takes O(nlogn) time.

Best Case:

The best time complexity of this algorithm is O(n), where n is the number of nodes in the tree. This is because if the tree is already a BST, then we just need to do an in-order traversal of the tree, which takes O(n) time.

Worst Case:

The worst case time complexity of this algorithm is O(nlogn), where n is the number of nodes in the tree. This is because if the tree is not a BST then we need to do an inorder traversal of the tree, which takes O(n) time, and then we need to sort the array which tales O(nlogn) time.

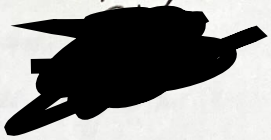
Average Case:

The average case time complexity of this algorithm is O(nlogn), where n is the number of nodes in the tree. This is because we need to do an inorder traversel of the tree, which takes O(n) time, and then we need to sort array, which takes O(nlogn) time.

Bahadur Ethee Kulung
190104701
[redacted]

5.

Bahadır Etka Kılınç
1901042+01



```
findPairs (array, x):
    dictionary ← {}
    n ← length of array
    for i from 0 to n:
        if x - array[i] is in dictionary then
            return (array[i], i + array[i])
        else
            append array[i] to dictionary
        end if
    end for
    return -1
```

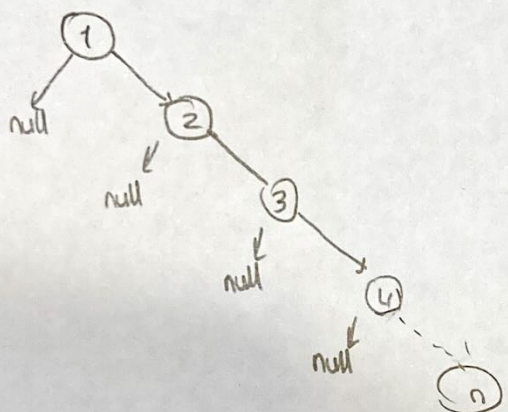
Analyze

In best case, the algorithm returns the firstly compared pair. That means the loop will iterate only once. Therefor the best case complexity is $\Omega(1)$

In worst case, there might not be such a pair in the array, or the pair might be the last one to compared. Since we go over the array once, we will make at most n comparisons and then return. Therefore, the worst case complexity is $O(n)$.

6. a. False. The shape of BST depends on the order in which elements are added. A new element is added. A new element is added to the BST as a new leaf node, potentially increasing the depth of the tree. It is possible for the BST with n nodes to be chain of nodes of height n. This happens, for example, if all items are added in a sequential order. In general, it is preferable for a BST to be as shallow as possible. This keeps the average cost of a BST transaction low.

b. True. The binary search tree is a balanced binary search tree. Height of the binary search tree becomes log(n). So time complexity of BST operations = O(log n)
But in some cases the complexity is linear, time complexity of BST operations = O(n)
when the tree is skewed:



c. True. If the array is ordered, it takes constant time to find the largest and smallest element. If the array is not sorted we may need to iterate the whole array to find the largest or smallest element of this array then the time complexity is O(n).

d. False. In case of arrays, the middle element can be accessed in O(1) time, but in linked lists, there is no concepts of indices as the element is allocated in a non-contiguous manner. So, the time complexity to find the middle of a linked list is O(n) Hence, binary search on arrays proves to be more efficient than in the linked list.

e. False. The insertion sort algorithm has two loops. One which iterates from 2 to array length and inner loop which iterates from the picked up element to all of the sorted array elements. The first loop executed n-1 time irrespective of input order of elements so the differentiation factor is inner loop execution. If the input array is reversely sorted then inner loop will execute maximum times.

Bahadır Etka Kılınç
1901042701

[redacted]