CSE 321 - Homework 2

Due date: 13/11/2022, 23:59

1. ***20 pts.*** Solve the following recurrence relations by using Master Theorem and give Θ bound for each of them. If any of the relations cannot be solved by using Master Theorem, state that this is indeed the case together with the explanation of the reason.

   (a) $T(n) = 2 \cdot T(\frac{n}{4}) + \sqrt{nlogn}$
   (b) $T(n) = 9 \cdot T(\frac{n}{3}) + 5n^2$
   (c) $T(n) = \frac{1}{2} \cdot T(\frac{n}{2}) + n$
   (d) $T(n) = 5 \cdot T(\frac{n}{2}) + logn$
   (e) $T(n) = 4^n \cdot T(\frac{n}{5}) + 1$
   (f) $T(n) = 7 \cdot T(\frac{n}{4}) + nlogn$
   (g) $T(n) = 2 \cdot T(\frac{n}{3}) + \frac{1}{n}$
   (h) $T(n) = \frac{2}{5} \cdot T(\frac{n}{5}) + n^5$

Solution:

MASTER'S THEOREM

$T(n) = aT(\frac{n}{b}) + n^k log^p n$
where n is the input size, a is the count of subproblems in the dividing recursive function, and $\frac{n}{b}$ is the size of each subproblem. The constants a, b, k hold the following conditions:

- $a \geq 1$
- $b > 1$
- $k \geq 0$

Master's Theorem states that:
Case 1: If $a > b^k$, then
$T(n) = \Theta(n^{log_b a})$

Case 2: If $a = b^k$, then
a) If $p > -1$, then $T(n) = \Theta(n^{log_b a} log^{p+1} n)$
b) If $p = -1$, then $T(n) = \Theta(n^{log_b a} loglogn)$
c) If $p < -1$, then $T(n) = \Theta(n^{log_b a})$

Case 3: If $a < b^k$, then
a) If $p \geq 0$, then $T(n) = \Theta(n^k log^p n)$
b) If $p < 0$, then $T(n) = \Theta(n^k)$

PS: If you didn't learn the Master's theorem as shown above in the class, it is accepted if you ignored $logn$.

(a) $T(n) = 2 \cdot T(\frac{n}{4}) + \sqrt{nlogn}$

Constants:

- $a = 2 \geq 1$ ✓
- $b = 4 > 1$ ✓
- $k = \frac{1}{2} \geq 0$ ✓

Thus, we can solve this problem by using Master's theorem.

$b^k = 4^{\frac{1}{2}} = \sqrt{4} = 2 \rightarrow a = b^k \rightarrow$ Hence, we should follow Case 2.
Since Case 2 has 3 sub-cases, we look into p.
$p = \frac{1}{2} > -1$, hence we will follow Case2) a).

Case 2: a) $T(n) = \Theta(n^{log_b a} log^{p+1} n)$
$T(n) = \Theta(n^{log_4 2} log^{\frac{1}{2}+1} n)$
$T(n) = \Theta(\sqrt{n} log^{\frac{3}{2}} n)$

(b) $T(n) = 9 \cdot T(\frac{n}{3}) + 5n^2$

Constants:

- $a = 9 \geq 1$ ✓
- $b = 3 > 1$ ✓
- $k = 2 \geq 0$ ✓

Thus, we can solve this problem by using Master's theorem.

$b^k = 3^2 = 9 \rightarrow a = b^k \rightarrow$ Hence, we should follow Case 2.
Since Case 2 has 3 sub-cases, we look into p.
$p = 0 > -1$, hence we will follow Case2) a).

Case 2: a) $T(n) = \Theta(n^{log_b a} log^{p+1} n)$
$T(n) = \Theta(n^{log_3 9} logn)$
$T(n) = \Theta(n^2 logn)$

(c) $T(n) = \frac{1}{2} \cdot T(\frac{n}{2}) + n$

Constants:

- $a = \frac{1}{2} \not\geq 1$ ✗

Thus, we cannot solve this problem by using Master's theorem.

(d) $T(n) = 5 \cdot T(\frac{n}{2}) + logn$

Constants:

- $a = 5 \geq 1$ ✓
- $b = 2 > 1$ ✓
- $k = 0 \geq 0$ ✓

Thus, we can solve this problem by using Master's theorem.

$b^k = 2^0 = 1 \rightarrow a > b^k \rightarrow$ Hence, we should follow Case 1.

Case 1: $T(n) = \Theta(n^{log_b a})$
$T(n) = \Theta(n^{log_2 5})$

(e) $T(n) = 4^n \cdot T(\frac{n}{5}) + 1$

Constants:

- $a = 4^n \rightarrow a$ is not a constant value. ✗

Thus, we cannot solve this problem by using Master's theorem.

(f) $T(n) = 7 \cdot T(\frac{n}{4}) + nlogn$

Constants:

- $a = 7 \geq 1$ ✓
- $b = 4 > 1$ ✓
- $k = 1 \geq 0$ ✓

Thus, we can solve this problem by using Master's theorem.

$b^k = 4^1 = 4 \rightarrow a > b^k \rightarrow$ Hence, we should follow Case 1.
Case 1: $T(n) = \Theta(n^{log_b a})$
$T(n) = \Theta(n^{log_4 7})$

(g) $T(n) = 2 \cdot T(\frac{n}{3}) + \frac{1}{n}$

Constants:

- $a = 2 \geq 1$ ✓
- $b = 3 > 1$ ✓
- $k = -1 \not\geq 0$ ✗

Thus, we cannot solve this problem by using Master's theorem.

(h) $T(n) = \frac{2}{5} \cdot T(\frac{n}{5}) + n^5$

Constants:

- $a = \frac{2}{5} \not\geq 1$ ✗

Thus, we cannot solve this problem by using Master's theorem.

2. ***10 pts.*** Apply the insertion sort algorithm to the following array in ascending order. Explain every step in detail. What is the reasoning behind each operation? What is the updated version of the array?

$A = \{3, 6, 2, 1, 4, 5\}$

Solution:

Input:

3	6	2	1	4	5

Step 1:

3	6	2	1	4	5

The colorful part on the left side indicates the sorted part.

Step 2:

3	6	2	1	4	5

We don't change the position of 6, since it is larger than 3.

Step 3:

2	3	6	1	4	5

Since 2 is smaller than 6, we swap them, then we repeat this for 3.

Step 4:

1	2	3	6	4	5

Since 1 is smaller than all elements on the left, we move it to the beginning of the array.

Step 5:

1	2	3	4	6	5

Since 4 is smaller than 6, we swap them.

Step 6:

1	2	3	4	5	6

Since 5 is smaller than 6, we swap them.

PS: At each step, we compare the element with each element that comes before it in the array. For instance, In step 4, we compare 1 with 6 first. Since 1 is smaller, we swap them. Then we compare it with 3 and swap. Finally, we compare it with and 2 and swap again. Since there is no more element, we move into the next step.

3. ***20 pts.*** Consider an array and a linked list, both with n elements. Answer the following questions for both data structures.

   (a) Analyze the worst-case time complexity of the following operations. Explain your answer in detail.

      i. Accessing the first element.
      ii. Accessing the last element.
      iii. Accessing any element in the middle.
      iv. Adding a new element at the beginning.
      v. Adding a new element at the end.
      vi. Adding a new element in the middle.
      vii. Deleting the first element.
      viii. Deleting the last element.
      ix. Deleting any element in the middle.

   (b) Analyze the space requirements.

Solution:

(a) Analyzing the worst-time complexity.

	Array	Linked List
i	**Regardless of the size of the array and the index of the element we are trying to access, the cost is $O(1)$. Because the elements of an array are stored at contiguous memory locations.**	**In the linked list, the elements are not stored at contiguous memory locations. But we can easily access the first element in constant time ($O(1)$) by using the root.**
ii	**As stated above, regardless of the index, cost is $O(1)$.**	**We can go to the last elements in the list step by step. This means that we have to traverse the list, which takes $O(n)$ time.**
iii	**As stated above, regardless of the index, cost is $O(1)$.**	**Similar to accessing the last element, we should traverse the list until we find what we are looking for. So, we can say that it takes $O(k)$ time where k is the index of the desired element. But in the average case, this is $O(n)$ as well.**

	Array	Linked List
iv	**To add an element at the beginning of the array, we should shift all of the elements to right, which takes $O(n)$ time.**	**We can create a new node and assign it as our new root. This will take constant time ($O(1)$) only.**
v	**If the array is not full, this operation will cost only constant time. But if the array is full, we should copy this array to a larger one, and copying n elements will take $O(n)$ time.**	**To add a new element to the end, we should go to the end of the list, which takes $O(n)$ time.**
vi	**Similar to adding an element at the beginning, we should move the elements on the right side of that element. That costs $O(n-k)$, where k is the index of the element we are adding. In general, this operation will cost $O(n)$ time.**	**First, we should go to the memory location where we want to add the new element. Since accessing an element (which is not the root) takes linear time, this operation will cost $O(n)$ as well.**
vii	**If we are deleting the first element we should shift the remaining part of the array to the left. Shifting $n-1$ elements will cost $O(n)$ time.**	**If we assign the 2^{nd} node of of the list as the root, and then delete the previous root, it will take constant time ($O(1)$).**
vii	**We can easily free the last element, There is no need to shift, because there is no element left on the right side. Therefore, this operation takes constant time ($O(1)$).**	**To delete the last element, first we should reach the node before the last. We free the last element and assign the next address pointer of the previous one as NULL. This will take $O(n)$ time, since accessing that element costs linear time.**
ix	**If we are deleting a middle element we should shift the remaining part of the array to the left. Shifting $n-k$ elements will cost $O(n-k)$, where k is the index of the element we are deleting. In general, this operation will cost $O(n)$ time.**	**First, we should access the previous of the element which we want to delete. Then we can change the address of the next element and finally, free the node. Since accessing a middle element takes linear time, this operation will cost $O(n)$ as well.**

(b) While storing n elements in an array simply requires $O(n)$ space, a linked-list has a need of storing pointers too. Therefore it is $O(2n)$ for a linked-list. Even though $O(2n)$ converges to $O(n)$, we can still say that arrays are more beneficial in terms of space complexity.

4. ***15 pts.*** Construct an algorithm that converts a given binary tree with size n to a binary search tree (BST). Make sure you preserve the structure of the tree, i.e. you should not add or delete a node. Write down the pseudo-code of the algorithm, explain your reasoning, and analyze the best-case, worst-case, and average-case time complexities.

Solution:

```
helperFunction (root, keys)
    if root is NULL then
        return
    end if
    helperFunction (root.left, keys)
    root.data ←  next element of keys
    helperFuntion  (root.right, keys)
    return

convertToBST (root):
    if root is NULL then
        return
    end if
    keys ← keys of the tree
    keys ← sort(keys)
    helperFunction (root, keys)
    return
```

Firstly, we traverse the tree and save the keys in a set, then sort them. Once we have the sorted set of keys, we start to update the data of each node (in this way, we preserve the structure of the tree). Since the size of the tree is n, we will make n operations to change the values. But the sorting operation dominates the time complexity of the algorithm. If we use Merge Sort, which costs $O(nlogn)$ both in the best and the worst case, the algorithm above will take $O(nlogn)$ as well (in all cases).

5. ***15 pts.*** Consider an integer array $A = \{a_0, a_1, ..., a_n\}$ and an integer x. You are asked to find a pair (a_i, a_j), if any, within this array such that $|a_i - a_j| = x$. Design an algorithm with $O(n)$ time complexity to solve the problem. Write down the pseudo-code of the algorithm, and explain your reasoning in detail.

Solution:

```
findPair(array, x):
    n  ←  size of the array
    if n == 0 then
        return
    dictionary  ←  {}
    for each element of the array
        if element is not in dictionary then
            append element to dictionary with value 1
        else
            increment the value of the element in the dictionary
        end if
    for i from 0 to n:
        if x - array[i] in dictionary.keys then
            return (array[i], x-array[i])
        else if array[i] - x in dictionary.keys then
            return (array[i], array[i]-x)
    return -1
```

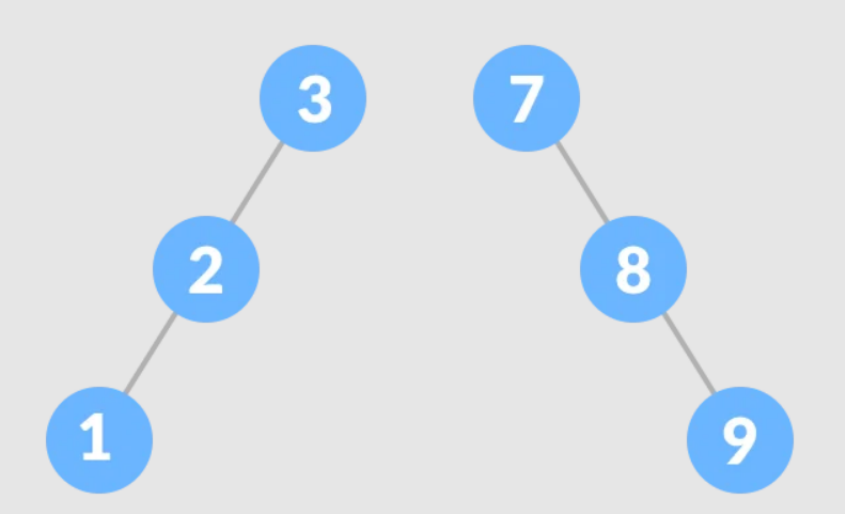
We keep the unique elements of the array in a dictionary. Then we traverse the array and check if the dictionary has any element that generates the given difference when we differentiate it from the current element. If so, the algorithm returns it. As seen, the algorithm uses a single loop with n iterations and there is no other operation that takes larger than $O(n)$. Thus, the time complexity is $O(n)$.

PS: You may think that checking if an element is in a set or not, we should traverse it which causes nested loops. But the point of using hashing is the containment operation ("in", which we use to check if an element is in a list, set, etc.) over a set that takes constant time instead of linear time. For further information, you may check this page.

6. ***20 pts.*** For each of the statements below, indicate true or false with the explanation of the reason (explain the reason for each of them, not only for false ones).

   (a) Shape of a BST (full, balanced, etc.) depends on the insertion order.
   TRUE → Because the first node we add to the BST will be the root and it won't be changed after the addition of other nodes. This might cause storing most of the elements on one side only. Therefore, the order of the insertion has a great impact on the shape of the tree.

   (b) The time complexity of accessing an element of a BST might be linear in some cases.
   TRUE → If the tree is left or right skewed as seen below, it will take $O(n)$ time to access the element at the bottom.



   (c) Finding an array's maximum or minimum element can be done in constant time.
   FALSE → Because we have to traverse the array, to make sure. This costs linear time. (But if we know that the given array is sorted, it can be done in constant time.)

   (d) The worst-case time complexity of binary search on a linked list is $O(log(n))$ where n is the length of the list.
   FALSE → The efficiency of binary search comes from the easiness of reaching an element in an array in constant time. But in a linked-list, as seen in Question 3, the worst time complexity of accessing an element is $O(n)$. Therefore the worst-time complexity of binary search on a linked list is $O(n)$ (which is the same as linear search).

   (e) Worst-case time complexity of the insertion sort algorithm is $O(n)$ if the given array is reversely sorted.
   FALSE → If the array is reversely sorted, it means that the algorithm has to simply sort it, and no algorithm does this operation in $O(n)$ time.