ShanghaiTech University

CS101 Algorithms and Data Structures

Fall 2021

Homework 4

Due date: 23:59, October 24, 2021

1. Please write your solutions in English.
2. Submit your solutions to gradescope.com.
3. Set your FULL NAME to your Chinese name and your STUDENT ID correctly in Account Settings.
4. If you want to submit a handwritten version, scan it clearly. Camscanner is recommended.
5. When submitting, match your solutions to the according problem numbers correctly.
6. No late submission will be accepted.
7. Violations to any of the above may result in zero grade.
8. Problem 0 gives you a template on how to organize your answer, so please read it carefully.

1: (2' + 2' + 2') Trees

Each question has **exactly one** correct answer. Please answer the following questions **according to the definition specified in the lecture slides**.

Note: Write down your answers in the table below.

Question 1	Question 2	Question 3
D	B	A

Question 1. *Which of the following statements is true?*

(A) Each node in a tree has exactly one parent pointing to it.

(B) Nodes with the same ancestor are siblings.

(C) The root node cannot be the descendant of any node.

(D) Nodes whose degree is zero are also called leaf nodes.

Question 2. *Given the following pseudo-code, what kind of traversal does it implement?*

```
1: function ORDER(node)
2:     if node has left child then
3:         order(node.left)
4:     end if
5:     if node has right child then
6:         order(node.right)
7:     end if
8:     visit(node)
9: end function
```

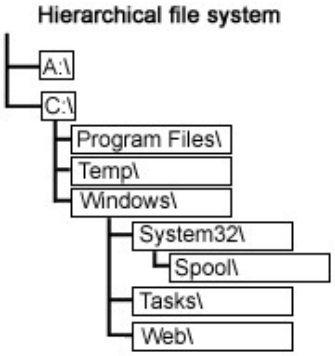
(A) Preorder depth-first traversal

(B) Postorder depth-first traversal

(C) Inorder depth-first traversal

(D) Breadth-first traversal

Question 3. *Which traversal strategy should we use if we want to print the hierachical structure ?*



(A) Preorder depth-first traversal

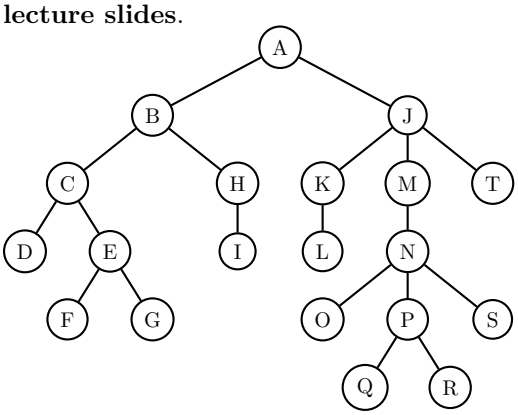
(B) Postorder depth-first traversal

(C) Inorder depth-first traversal

(D) Breadth-first traversal

2: (3+3+3pts) Tree Structure and Traversal

Answer the following questions for the tree shown below **according to the definition specified in the lecture slides**.



Question 4. *Please specify:*

1. *The **children** of the **root node** with their **degree** respectively. BJ, deg(B)=2, deg(J)=3*
2. *All **leaf nodes** in the tree with their **depth** respectively. DFGILOQRST, dep(T)=2, dep(D)=dep(I)=dep(L)=3, dep(F)=dep(G)=dep(O)=dep(S)=4, dep(Q)=dep(R)=5*
3. *The **height** of the tree. height = max{depth(nodes)} = 5*
4. *The **ancestors** of O. AJMNO*
5. *The **descendants** of C. CDEFG*
6. *The **path** from A to S. $A \to J \to M \to N \to S$*

For the following two questions, traverse the **subtree** of the tree shown above with specified root.

Note: Form your answer in the following steps.

1. Decide on an appropriate **data structure** to implement the traversal.
2. When you are pushing the children of a node into a **queue**, please push them alphabetically i.e. from left to right; when you are pushing the children of a node into a **stack**, please push them in a reverse order i.e. from right to left.
3. **Show all current elements in your data structure at each step** clearly . **Popping a node** or **pushing a sequence of children** can be considered as one single step.
4. **Write down your traversal sequence** i.e. the order that you pop elements out of the data structure.

Please refer to the examples displayed in the lecture slide for detailed implementation of traversal in a tree using the data structure.

Question 5. *Run **Depth First Traversal** in the subtree with root B.*

Stack:

□	□	*C*
B	□	*H*
□	*D*	□
□	*E*	*E*
H	*H*	*H*
□	*F*	□
□	*G*	*G*
H	*H*	*H*
H	□	*I*

Sequence:
B C D E F G H I

Question 6. *Run **Breadth First Traversal** in the subtree with root N.*

Queue:
N
□
O P S
P S
S
S Q R
Q R
R

Sequence:
N O P S Q R

3: (2+3pts) Recurrence Relations

For each question, find the asymptotic order of growth of $T(n)$ i.e. find a function g such that $T(n) = O(g(n))$. You may ignore any issue arising from whether a number is an integer. You can make use of the Master Theorem, Recursion Tree or other reasonable approaches to solve the following recurrence relations.
Note: ***Mark or circle*** *your final answer clearly.*

Question 7. $T(n) = 4T(n/2) + 42\sqrt{n}$.

$(a, b, d) = (4, 2, 1/2)$, so by the Master Theorem $\log_b a = 2 > d = 1/2$,

hence $T(n) = O(n^{\log_b a}) = O(n^2)$

Question 8. $T(n) = T(\sqrt{n}) + 1$. *You may assume that* $T(2) = T(1) = 1$.

Let $n = 2^k$, then the problem size in each level of the recursion tree will be:

$2^k \to 2^{k/2} \to 2^{k/4} \to ... \to 2^4 \to 2^2 \to 2^1$

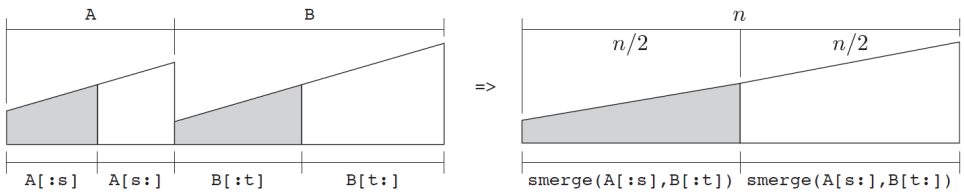
So there are $\log_2 k = \log_2(\log_2 n)$ levels in this recursion tree.

For each level, the work is $\Theta(1)$, so the total work $= \log_2(\log_2 n) \times 1 = O(\log(\log n))$

Therefore, $T(n) = O(\log(\log n))$

3: (7+6+6pts) Divide and Conquer Algorithm

Question 9. *In this problem, we will find an alternative approach to the merge step in Merge Sort named* ***Slice Merge*** *. Suppose A and B are sorted arrays with possibly different lengths, and let $n = len(A)+len(B)$. You may assume n is a power of two and all n elements have distinct value. The slice merge algorithm,* ***smerge(A,B)****, merges A and B into a single sorted arrays as follows:*



Step 1: *Find index s for subarray A and index t for subarray B ($s + t = \frac{n}{2}$) to form two prefix subarrays* `A[:s]` *and* `B[:t]`*, such that* `A[:s]` $\cup$ `B[:t]` *contains the smallest $\frac{n}{2}$ elements in all n elements of $A \cup B$.*

Step 2: *Recur for* `X = smerge(A[:s], B[:t])` *and* `Y = smerge(A[s:], B[t:])` *respectively to reorder and merge them. Return their concatenation $X + Y$, a sorted array containing all elements in $A \cup B$.*

For example, if $A = [1, 3, 4, 6, 8]$ and $B = [2, 5, 7]$, we should find $s = 3$ and $t = 1$ and then recursively compute:

```
smerge([1, 3, 4], [2]) + smerge([6, 8], [5, 7]) = [1, 2, 3, 4] + [5, 6, 7, 8]
```

1. *Describe an algorithm for Step 1 to find indices s and t in $O(n)$ time using $O(1)$ additional space. Write down your main idea briefly (or pseudocode if you would like to) and analyse the runtime complexity of your algorithm below. You may assume array starts at index 1.* ***(2pts)***

```
1:  function SLICEPARTITION(A, B)
2:      s ← 0
3:      t ← 0
4:      while s + t < n/2 do
5:          if t = length(B) or A_{s+1} < B_{t+1} then
6:              s ← s +1
7:          else if s = length(A) or A_{s+1} > B_{t+1} then
8:              t ← t +1
9:          end if
10:     end while
11:     return (s, t)
12: end function
```

This algorithm uses only constant additional storage to keep track of s and t.During each of $n/2$ iterations, the calculation of comparison and arithmetic operations can de done in constant time, which is $O(1)$. Therefore, the total runtime complexity for this step is $T(n) = O(n)$.

2. *Write down a recurrence for the runtime complexity of* `smerge(A,B)` *when* $A \cup B$ *contains a total of* n *items. Solve it using the Master Theorem and show your calculation below.* ***(2pts)***
Note: Write your answer for time complexity in asymptotic order form i.e. $T(n) = O(g(n))$.

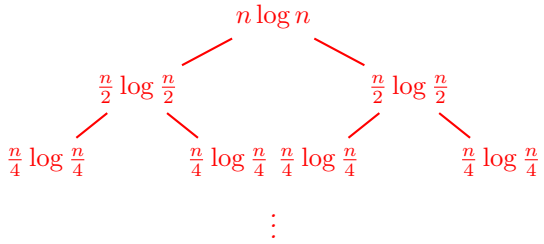
Recurrence : $T(n) = 2T(n/2) + O(n)$, *where* $O(n)$ *is the answer to question 1.*
$(a, b, d) = (2, 2, 1)$, *then by the Master Theorem* $\log_b a = 1 = d$.
Therefore, the runtime complexity of smerge is $T(n) = O(n \log n)$.

3. *Recall the merge step* `merge(A,B)` *to combine two subarrays of length* $n/2$ *in the Merge Sort algorithm covered in our lecture slides. Compare the runtime complexity of* `smerge(A,B)` *with* `merge(A,B)`. ***(1pts)***

$T_{smerge}(n) = O(n \log n)$ *while* $T_{merge}(n) = O(n)$,
so $T_{smerge}(n) > T_{merge}(n)$

4. *Replace* `merge(A,B)` *by* `smerge(A,B)` *in the merge stage of Merge Sort to develop a new sorting method namely* ***S-Merge Sort****. Write down a recurrence for the runtime complexity of S-Merge Sort. Solve it and show your calculation below.* ***(2pts)***
Note: Write your answer for time complexity in asymptotic order form i.e. $T(n) = O(g(n))$.

Recurrence : $T(n) = 2T(n/2) + O(n \log n)$, *where* $O(n \log n)$ *is the answer to question 2. The solution is* $O(n \log^2 n)$ *and the calculation is as the recursion tree shown below.*



$$
\begin{aligned}
T(n) &= n \log n + 2\frac{n}{2} \log \frac{n}{2} + 4\frac{n}{4} \log \frac{n}{4} + \cdots \\
&= n(\log n + \log \frac{n}{2} + \log \frac{n}{4} + \cdots) \\
&= n(\log n + (\log n - \log 2) + (\log n - \log 4) + \cdots) \\
&= n((\log n + \log n + \cdots) - (\log 2 + \log 4 + \cdots)) \\
&= n(\log^2 n - (1 + 2 + \cdots)) \\
&\approx n(\log^2 n - \frac{\log^2 n}{2}) \\
&= O(n \log^2 n)
\end{aligned}
$$

Question 10. *There are n students in SIST and each student i has 2 scores A_i and P_i, score in Algorithms and Data Structures course and score in Probabilty and Statistics course respectively. Students i, j could form a mutual-help pair in CS101 class if and only if $A_i < A_j$ and $P_i > P_j$. How many possible mutual-help pairs (i, j) could be formed in CS101 class?*
Design an efficient algorithm to figure out this problem. For comparison, our algorithm runs in $O(n \log n)$ time. (Hint: how to count inversions?)

Note: Your answer should be consistent with the template we provide in Problem 0 Example.

Algorithm Design:

1. Sort these students with (A_i, P_i) by their A_i scores in the ascending order, so $Ai < Aj$ implies $i < j$. Notice (A_i, P_i) pairs here are reordered. Now this problem is transformed into that two students (i, j) can form a mutual-help pair if $Pi > Pj$ and $i < j$ i.e. they form an inversion.
2. We count the inversions using divide and conquer. If the problem is reduced into size 1, then return 0 with the score itself.
3. Else we cut the score array in the middle and recur for each half. We recursively sort and merge subarrays in the same way we do in Merge Sort. Thus, we obtain sorted subarray A with its number of inversions cnt_1 and similarly B with cnt_2.
4. Now we extend the merge stage in Merge Sort to count inversions. Initially let $cnt = 0$. When comparing two scores of each half where the head pointers are located, if the head of B is smaller than head of A, then add it into the merged sorted array and let cnt $+=$ number of remaining unmerged scores in A.
5. Else, only add the head of A into the merged array.
6. Return $cnt + cnt_1 + cnt_2$ with the sorted array.

Time Comlexity Analysis: The Quick Sort takes $T_1(n) = O(n \log n)$ time. Then let's analyse the time complexity of the divide and conquer algorithm:
During each recursion, the calculation of arithmetic operations and comparision can be done in constant time, which is $O(1)$. And we divide the current score array into two subarrays of half size, computing recursively. To count the number of inversions and merge two subarrays, each iteration takes constant time and $\Theta(n)$ iterations are done. That is

$$T_2(n) = 2T_2(n/2) + \Theta(n)$$

By the Master Theorem, the complexity of this problem is $T_2(n) = O(n \log n)$.

Therefore, the complexity of this problem is $T(n) = T_1(n) + T_2(n) = O(n \log n)$ in total.

Pseudocode(Optional):

```
1: function SOLUTION(A, P)
2:     P ← QUICKSORTBYA(A, P)
3:     (cnt, sorted) ←COUNTINVERSIONS(P, 1, n)
4:     return cnt
5: end function
6:
7: function COUNTINVERSIONS(P, left, right)
8:     if left = right then
9:         return (0, P_left)
10:    end if
11:    mid ← ⌊(right − left)/2⌋ + left
12:    (cnt_1, A) ← COUNTINVERSIONS(P, left, mid)
13:    (cnt_2, B) ← COUNTINVERSIONS(P, mid + 1, right)
14:    cnt ← cnt_1 + cnt_2
15:    sorted ← {}
16:    while there exists remaining element in A or B do
17:        if the head of B < the head of A then
18:            cnt ← cnt + number of remaining elements in A
19:            remove the head of B and add it into sorted
20:        else
21:            remove the head of A and add it into sorted
22:        end if
23:    end while
24:    return (cnt, sorted)
25: end function
```

Question 11. *Suppose you are a teaching assistant for CS101, Fall 2077. The TA group has a collection of n suspected code solutions from n students for the programming assignment, suspecting them of academic plagiarism. It is easy to judge whether two code solutions are equivalent with the help of "plagiarism detection machine", which takes two code solutions* `(A,B)` *as input and outputs* `isEquivalent(A, B)` $\in \{$`True`, `False`$\}$ *i.e. whether they are equivalent to each other.*
TAs are curious about whether there exists a ***majority*** *i.e. an* ***equivalent class of size*** $> \frac{n}{2}$ *among all subsets of the code solution collection. That means, in such a subset containing more than* $\frac{n}{2}$ *code solutions, any two of them are equivalent to each other.*
Assume that the only operation you can do with these solutions is to pick two of them and plug them into the plagiarism detection machine. Please show TAs' problem can be sloved using $O(n \log n)$ invocations of the plagiarism detection machine.

Algorithm Design:

1. If the problem is reduced into size 1 with exactly one solution, then the majority is the solution itself.
2. Else we divide current solution collection into two subsets (cut in the middle), recur for a candidate majority of each half, denoted by m_1 and m_2.
3. Count the size of each equivalent class of m_1 and m_2 in the whole collection, denoted by $size_1$ and $size_2$, by plugging m_i and each solution in the collection into the machine and determining whether they are equivalent.
4. If any of $size_1$ or $size_2$ exceeds $n/2$, then return the corresponding candidate as majority.
5. Else there is no majority in current problem.

(The correctness of this algorithm follows from that if there is a majority equivalence class, then there must be a majority equivalence class for at least one of its two halves, which can be proved by contradiction)

Pseudocode(Optional):

```
1:  function MAJORITY(a, left, right)
2:      if right = left then
3:          return a_left
4:      end if
5:      mid ← ⌊(right − left)/2⌋ + left
6:      m_1 ← MAJORITY(a, left, mid)
7:      m_2 ← MAJORITY(a, mid+1, right)
8:      size_1 ← the size of equivalence class of m_1 in a
9:      size_2 ← the size of equivalence class of m_2 in a
10:     if size_1 > n/2 then
11:         return m_1
12:     else if size_2 > n/2 then
13:         return m_2
14:     else
15:         return None
16:     end if
17: end function
```

Time Complexity Analysis: During each recursion, the calculation of arithmetic operations can be done in constant time, which is $O(1)$. And we divide the current set into two subsets of half size. To merge two solutions to the subproblems, we scan the whole collection and count how many solutions are equivalent to candidates m_1 and m_2, which is $\Theta(n)$. That is

$$T(n) = 2T(n/2) + \Theta(n)$$

By the Master Theorem, the complexity of this problem is $T(n) = O(n \log n)$.

Alternative solution in linear time:
Pair up all solutions, and test all pairs for equivalence. If n was odd, one student is unmatched. For each pair that is not equivalent, discard both. For pairs that are equivalent, keep one of the two. Keep also the unmatched students, if n is odd. We can call this subroutine ELIMINATE.
The observation that leads to the linear time algorithm is as follows. If there is an equivalence class with more than $n/2$ solutions, then the same equivalence class must also have more than half of the solutions after calling ELIMINATE. This is true, as when we discard both solutions in a pair, then at most one of them can be from the majority equivalence class. One call to ELIMINATE on a set of n students takes $n/2$ tests, and as a result, we have only $\leq \lceil n/2 \rceil$ students left. When we are down to a single solution, then its equivalence is the only candidate for having a majority. We test this solution against all others to check if its equivalence class has more than $n/2$ elements.
This method takes $n/2 + n/4 + \cdots$ tests for all the eliminates, plus $n - 1$ tests for the final counting, for a total of less than $2n$ tests. Therefore the total time complexity is $O(n)$.