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.

Problem 0: Notes and Example

Notes

1. Some problems in this homework requires you to design Divide and Conquer algorithm. When grading these problems, we will put more emphasis on how you reduce a problem to a smaller size problem and how to combine their solutions with Divide and Conquer strategy.
2. Your answer for these problems should include:
   (a) Algorithm Design
   (b) Time Complexity Analysis
   (c) Pseudocode (Optional)
3. In Algorithm Design, you should describe each step of your algorithm clearly.
4. Unless required, writing pseudocode is optional. If you write pseudocode, please give some additional descriptions if the pseudocode is not obvious.
5. You are recommended to finish the algorithm design part of this homework with LaTeX.

0: Binary Search Example

Given a sorted array a of n elements, design an algorithm to search for the index of given element x in a.

Algorithm Design: We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with the middle element, return the middle index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Otherwise (x is smaller) recur for the left half.

Pseudocode(Optional):
$left$ and $right$ are indecies of the leftmost and rightmost elements in given array a respectively.

```
function binarySearch(a, value, left, right)
    if right < left then
        return not found
    end if
    mid ← $\lfloor (right - left)/2 \rfloor + left$
    if a[mid] = value then
        return mid
    end if
    if value < a[mid] then
        return binarySearch(a, value, left, mid-1)
    else
        return binarySearch(a, value, mid+1, right)
    end if
end function
```

Time Complexity Analysis: During each recursion, the calculation of *mid* and comparison can be done in constant time, which is $O(1)$. We ignore half of the elements after each comparison, thus we need $O(\log n)$ recursions.

$$T(n) = T(n/2) + O(1)$$

Therefore, by the Master Theorem $\log_b a = 1 = d$, so $T(n) = O(\log n)$.

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?*

```
function ORDER(node)
    if node has left child then
        order(node.left)
    end if
    if node has right child then
        order(node.right)
    end if
    visit(node)
end function
```

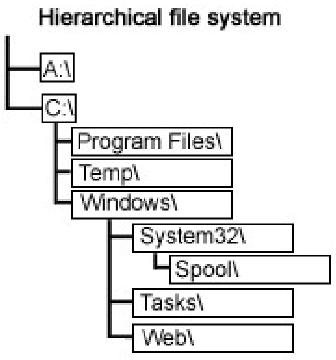
(A) Preorder depth-first traversal

(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

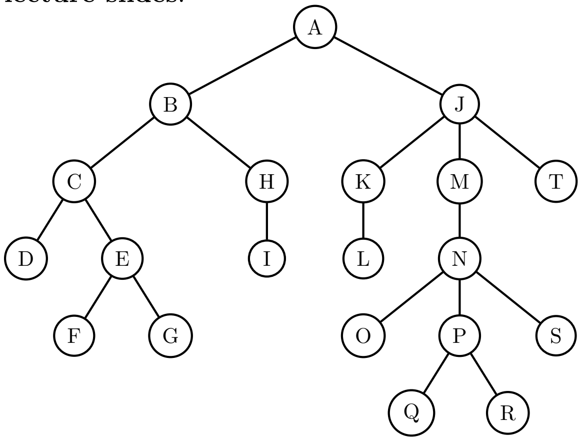
(B) Postorder depth-first traversal

(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. B(2),J(3)*
2. *All **leaf nodes** in the tree with their **depth** respectively. D(3),F(4),G(4),I(3),L(3),O(4),Q(5),R(5),S(4),T(2)*
3. *The **height** of the tree. height:5*
4. *The **ancestors** of O. O,N,M,J,A*
5. *The **descendants** of C. C,D,E,F,G*
6. *The **path** from A to S. (A,J,M,N,S)*

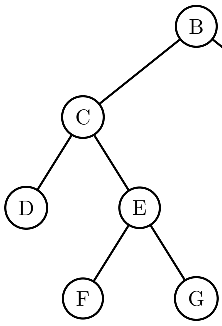
For the following two questions, traverse the **subtree** of the tree shown above with specified root.

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.*

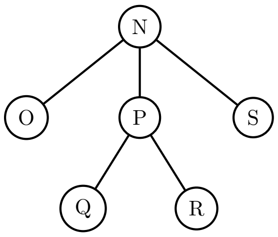


stack

		D		F			
	C	E	E	G	G		
B	H	H	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 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. $T(n) = 4T(n/2) + 42\sqrt{n}$.

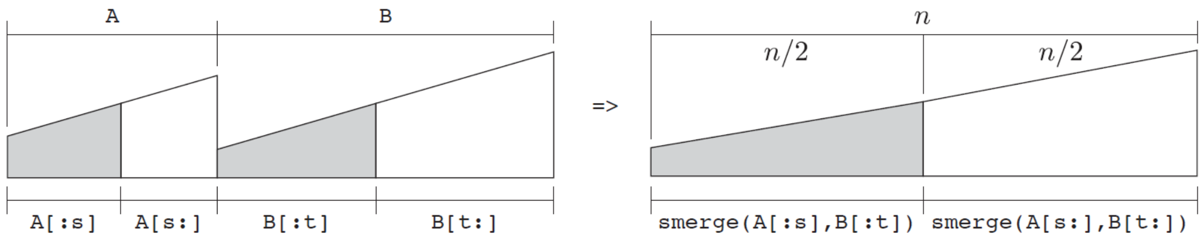
According to Master Theorem, $a = 4, b = 2, d = \frac{1}{2}$.
$\log_b a = \log_2 4 = 2 > \frac{1}{2} = d$.
So, $\boxed{T(n) = O(n^2)}$

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

$T(1) = T(2) = 1$
$T(4) = T(2) + 1 = 2$
$T(16) = T(4) + 1 = 3$
By induction, $T(2^{2^k}) = k + 1$
Let $n = 2^{2^k}$, then we have $k = \log_2 \log_2 n$.
So, $\boxed{T(n) = O(\log \log n)}$

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)***

   Algorithm Design:

   (a) *Set s and t equal to 0.*

   (b) *If $s + t$ is less than $n/2$, compare A[s] and B[t], let the smaller one's index plused by 1.*

   (c) *Repeat last step until once the condition is unsatisfied.*

   (d) *Then we get s and t.*

   Pseudocode:

```
s = t = 0
while s + t < n/2 do
    if A[s] < B[t] then
        s++
    else
        t++
    end if
end while
```

Time Complexity Analysis:
$T(n) = O(n/2) = O(n)$

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. $T(n) = O(g(n))$.

$T(n) = 2T(n/2) + O(n)$
According to Master Theorem, $a = 2, b = 2, d = 1$,
$\log_b a = \log_2 2 = 1 = d$,
so, $T(n) = O(n \log n)$

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)***

The time complexity of smerge is $O(n \log n)$.
The time complexity of merge is $O(n)$.

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. $T(n) = O(g(n))$.

$$\begin{aligned}
T(n) &= 2T(n/2) + O(n \log n) \\
&= 2(2T(n/4) + O(n/2 \log(n/2))) + O(n \log n) \\
&= 4T(n/4) + 2O(n/2(\log n - \log 2)) + O(n \log n) \\
&= 4T(n/4) + 2O(n \log n) - O(n \log 2) \\
&= \dots \\
&= \log n \times O(n \log n) \\
&= O(nlog^2 n)
\end{aligned}$$

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. Make a Student Class which contains two properties, one is A (score in Algorithms and Data Structure course), the other is P (core in Probabilty and Statistics course).
2. Use merge sort to sort all Students by their A in ascending order.
3. Then, use merge sort to sort all Students again but by their P in ascending order, while finding how many inversions there are.
4. Such as $A_i < A_j$ and $P_i > P_j$, so i goes before j after the first sort, but they should be found an inversion in the second sort, vise versa. Therefore, the number of inversions is the number of mutual-help pairs.

Pseudocode:
A pair counter:

```
count = 0;
```

A normal mergesort, ascending by A:

```
function MERGESORT1(Student arr[], int head, int tail)
    mid = (head + tail) / 2;
    mergesort1(arr,head,mid);
    mergesort1(arr,mid,tail);
    i1 = i2 = k = 0;
    while i1 < n1 and i2 < n2 do
        if arr1[i1].A < arr2[i2].A then
            arr[k] = arr1[i1];
            i1++;
        else
            arr[k] = arr1[i2];
            i2++;
        end if
        k++;
    end while
end function
```

A mergesort ascending by P, counting inversions:

```
function MERGESORT2(Student arr[], int head, int tail)
    int mid = (head + tail) / 2;
    mergesort1(arr,head,mid);
    mergesort1(arr,mid,tail);
    i1 = i2 = k = 0;
    while i1 < n1 and i2 < n2 do
        if arr1[i1].P < arr2[i2].P then
            arr[k] = arr1[i1];
            i1++;
        else
            arr[k] = arr1[i2];
            i2++;
            count++;
        end if
        k++;
    end while
end function
```

Function finding pairs:

```
function FINDPAIRS(Student arr[],int n)
    mergesort1(arr,0,n);
    mergesort2(arr,0,n);
    return count;
end function
```

Complexity Analysis:
Each mergesort in the function is $O(n\log n)$.($T(n) = 2T(n/2) + O(n), T(1) = \Theta(1)$, by Master Theorem, $T(n) = O(n\log n)$.)
Totally two mergesort in this function, so $T(n) = O(n\log n) + O(n\log n) = O(n\log n)$.

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.*

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

Algorithm Design:

1. Make use of mergesort, divide the sequence into two equal pieces each turn, until the size of the subsequence is 1.
2. For each subsequence, there can be a major value, and the number of times it occurs. Record both of them as **m** and **t**, respectively. If there is none, set $m = none$ and $t = 0$.
3. When merging two sequences, following these rules:
   (a) If both sequences have their $m = none$ and $t = 0$, we can simply set $m = none$ and $t = 0$ for the merged sequence.
   (b) If one of the sequence have its $m = none$ and $t = 0$, while the other does not, we can traverse the sequence whose $m = none$, counting how many numbers is equivalent to the other's m, add it to the other's t to form a new pair of m and t for the merged sequence.
   (c) If both sequences have an equal m coincidentally (which is not none), just set the sum of two t as a new t for the merged sequence, and inherit the m.
   (d) If the two sequences have different m, saying m1 and m2 respectively, corresponding to t1 and t2, we would traverse both sequence and count how many opponent's m occurs in this sequence.(i.e. traverse sequence one to find how many times m2 occors, and add it to t2; traverse sequence two to find how many times m1 occors, and add it to t1). Then compare t1 with t2, set the t for the merged sequence as the lager one of them, and its corresponding m as the new m.(i.e. if t1>t2, t=t1, m=m1).
   (e) After all these steps, judge whether the t for the merged sequence is greater than n/2 (half size of the sequence). If it's true, we can keep the m and t, else, just reset it as $m = none, t = 0$.

Pseudocode:

```
function findMajority(Student arr[], int head, int tail)
    if head == tail then
        return;
    end if
    mid = (head + tail) / 2;
    (m1,t1) = findMajority(arr[],head,mid);
    (m2,t2) = findMajority(arr[],mid,tail);
    if m1==none and m2==none then
        (m,t) = (none,0);
    end if
    if m1==none and m2!=none then
        t1 += traverse sequence two to find how many m1
        (m,t) = (m1,t1);
    end if
    if m1!=none and m2==none then
        t2 += traverse sequence one to find how many m2
        (m,t) = (m2,t2);
    end if
    if m1!=none and m2!=none then
        t1 += traverse sequence two to find how many m1
        t2 += traverse sequence one to find how many m2
        t = max(t1,t2);
        m = corresponding to t
    end if
    if t<=n/2 then
        (m,t) = (none,0);
    end if
    return (m,t);
end function
```

Complexity Analysis:
The same as mergesort, we solve the problem by Divide and Conquer:
$T(n) = 2T(n/2) + O(n)$, and $T(1) = \Theta(1)$, according to Master Theorem, $a = 2, b = 2, d = 1$,
so $T(n) = O(n \log n)$.