CS101 Algorithms and Data Structures

Fall 2019

Homework 9

Due date: 23:59, November 24, 2019

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 above may result in zero score.
8. **Describe or Design an algorithm means that first you should write down how your algorithm is going on, then analyze its time complexity, and finally prove its correctness.**

Problem 1: Hash Collisions - Direct Addressing

Consider a hash table consisting of $m = 11$ slots, and suppose we want to insert integer keys $A = [43, 23, 1, 0, 15, 31, 4, 7, 11, 3]$ orderly into the table.

First, let's suppose the hash function h_1 is:

$$h_1(k) = (11k + 4) \mod 10$$

Collisions should be resolved via chaining. If there exists collision when inserting a key, the inserted key(collision) is stored at the end of a chain.

(1) In the table below is the hash table implemented by array. Draw a picture of the hash table after all keys have been inserted. If there exists a chain, please use $\rightarrow$ to represent a link between two keys.

Index	**Keys**
0	
1	**7**
2	
3	
4	**0**
5	**1** $\rightarrow$ **31** $\rightarrow$ **11**
6	
7	**43** $\rightarrow$ **23** $\rightarrow$ **3**
8	**4**
9	**15**
10	

(2) What is the load factor λ?

Solution: $\lambda = \frac{10}{11} = 0.9091$

(3) Suppose the hash function is modified into

$$h_2(k) = ((11k + 4) \mod c) \mod 10$$

for some positive integer c. Find the smallest value of c such that no collisions occur when inserting the keys from A. Show you steps in detail and draw a picture of the hash table after all keys have been inserted in the table below. If there exists a chain, please use $\rightarrow$ to represent a link between two keys.

Solution: c = 79

Index	**Keys**
0	**23**
1	**15**
2	**7**
3	**43**
4	**0**
5	**1**
6	**11**
7	**3**
8	**4**
9	**31**
10	

Steps of finding c:

```cpp
#include <iostream>
int main() {
    int keys[10] = {43, 23, 1, 0 ,15, 31, 4, 7, 11, 3};
    int final_pairs[11] = {0};
    int c = 1;
    while(true) {
        int table[11] = {0};
        bool collison = false;
        for (int i = 0; i < 10; ++i) {
            int place = ((keys[i] * 11 + 4) % c) % 10;
            if(table[place] != 0) {
                collison = true;
                ++c;
                break;
            } else {
                table[place] = keys[i];
                continue;
            }
        }
        if(!collison){
            for (int i = 0; i < 11; ++i) {
                final_pairs[i] = table[i];
            }
            break;
        }
    }
    std::cout << c <<std::endl;
    for (int i = 0; i < 11; ++i) {
        std::cout << i << "\t" << final_pairs[i] << std::endl;
    }
    return 0;
}
```

From the previous questions, we can easily find that there often exists collisions in hash table. In real world, collisions is everywhere. What we need to do is to reduce the case of collisions.

Suppose we want to store n items into a hash table with size m. Collisions resolved by chaining.

(4) In lecture, Prof. Zhao has mentioned that we often want a hash table which can support `Search()` in $O(1)$ time complexity. Assume simple uniform hashing, that is $E[h(k) = l] = \frac{n}{m}$. Therefore, is `Search()` always has expected running time $O(1)$ as the lecture says? If it always, please explain your reason. If it is not always, under what case can `Search()` for a hash table has expected running time $O(n)$?

Solution:

No. If there too many values inserted (e.g. $n = \omega(m)$), then many values have to be hashed to the same key, then we must create linked lists for each key and `Search()` suffers from $O(n)$.

After lecture, TA Yuan and TA Xu are all motivated by Prof.Zhao. For this time both of them have two interesting ideas on dealing with collisions to reduce the worst time complexity for each operation.

(5) TA Xu wants to deal with collision via **resizing the array**. He lets $m \to 2m$, but he is so lazy that he doesn't want to change the hash function. Is this method can reduce collisions? Why or Why not?

Solution:

No. Even though the array has being expanded, the hash function will still hash each value to the original key, all expanded space will not be occupied.

(6) TA Yuan wants to store each chain using a data structure S instead of a linked list because he think this method can reduce the worst time complexity for insertion. Assume $\Theta(m) = \Theta(n)$, so the load factor $\lambda = \frac{\Theta(n)}{\Theta(m)} = 1$. Remember that in the hash table, there is no duplications. Suppose the data structure S can be **Binary Search Tree**, **Binary Heap** or **AVL Tree**. What is the worst-case running time of `Insert()` for each data structure in new hash table? Explain your reason briefly.

Solution:

$\boldsymbol{O(n)}$ for Binary Search Tree. Only $O(1)$ to find the key but in worst case the binary search has $O(n)$ to insert. Thus the total time is $O(n)$.

$\boldsymbol{O(\lg n)}$ for Binary Heap. Only $O(1)$ to find the key but in worst case the Binary Heap has $O(\lg n)$ to insert. Thus the total time is $O(\lg n)$.

$\boldsymbol{O(\lg n)}$ for AVL Tree. Only $O(1)$ to find the key but in worst case the AVL Tree has $O(\lg n)$ to insert. Thus the total time is $O(\lg n)$.

Problem 2: Hash Collisions - Open Addressing

ShanghaiTech has recently instituted a policy of renaming students from the conventional "First-Name Last-Name" to "Student k" where k refers to the student's ID number. Keyi Yuan is a CS101 TA. Unfortunately, he isn't a very friendly TA, so the number of students in his section has dwindled to 7.

Keyi wants to maintain his students' records by hashing the student numbers in his recitation into a hash table of size 10. He is using the hash function $h(k) = k \mod 10$ to insert each "Student k" and is using linear probing to resolve collisions.

Professor Zhao has handed Keyi an ordered list of the seven students in his section. After inserting these students into his hash table one at a time, Keyi's completed hash table looks like this:

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Keys	24				14	35	54	55	98	17

(1) Keyi is supposed to return the list of students to the professor, but he accidentally spills *A Little Little Tea Milk* on it, rendering it illegible. To cover his incompetence, Keyi wants to hide his mistake by recreating the list of student numbers in the order they were given, which is the same as the order they were inserted; however, he only remembers two facts about this order:

1. 98 was the first student number to be inserted.
2. 35 was inserted before 14.

Help Keyi by figuring out what the order must have been. Write the order directly.

Solution: 98 35 14 54 55 17 24

(2) Student 98 leaves Keyi's section, so Keyi deletes that record from the hash table. The next day, Keyi sees Student 15 yawning during the lecture, so he decides to give Student 15 a zero in class participation. However, Keyi can't remember whether Student 15 is actually in his section, so he decides to look up Student 15 in his hash table. **Assume we use erasing methods in Slides 149-151.** Which cells does Keyi need to inspect to determine whether Student 15 is in the table? Write down the probe sequence directly.

Solution: 5 6 7 8 9

(3) Student 98 rejoins Keyi's section, but Keyi is forgetful. So he uses a new hash function to insert all students into a new empty hash table:

$$h(k) = ((k+7) * (k+6)/16 + k) \mod 10$$

Collisions are resolved by using quadratic probing, with the probe function:

$$(i^2 + i)/2$$

Fill in the final contents of the hash table after the key values have been inserted in the order which is the same as question (1). We need to specify that the a/b opertaion means $a/b = \lfloor \frac{a}{b} \rfloor$.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Keys	**98**	**14**	**35**	**54**	**55**	**24**		**17**		

Problem 3: Secret Love In ShanghaiTech

Bob and Alice are sophomores in ShanghaiTech. Bob loves Alice. But Alice doesn't know he loves her. And Bob is too shy to express his love. Therefore, he wants to send Alice a secret message of characters via a specially encoded sequence of integers. Each character of Bob's message corresponds to a **satisfying** triple of distinct integers from the sequence satisfying the following property: every permutation of the triple occurs consecutively within the sequence.

For example, if Bob sends the sequence $A = (4, 10, 3, 4, 10, 9, 3, 10, 4, 3, 10, 4, 8, 3)$, then the triple $t = \{3, 4, 10\}$ is satisfying because every permutation of t appears consecutively in the sequence, i.e. $(3, 4, 10)$, $(3, 10, 4)$, $(4, 3, 10)$, $(4, 10, 3)$, $(10, 3, 4)$ and $(10, 4, 3)$ all appear as consecutive sub-sequences of A.

For any satisfying triple, its **initial occurrence** is the smallest index i such that all permutations of t is shown from index 0 to i in the sequence. For example, the initial occurrence of t in A is 10.

Bob's secret message will be formed by characters corresponding to each satisfying triple in the encoded sequence, increasingly ordered by initial occurrence. To convert a satisfying triple (a, b, c) into a character, Bob sends Alice an arithmatic function $f(a, b, c) = (a + b + c) \mod 27$ to decode the message. $f(\cdot) = 0$ to 25 correspond to the lower-case letters 'a' to 'z', while $f(\cdot) = 26$ corresponds to a space character. For example, the triple $\{3, 4, 10\}$ corresponds to the letter 'r'.

(1) Suppose Bob sends Alice the following sequence:

$$(10, 13, 9, 10, 13, 5, 2, 13, 5, 10, 9, 13, 10, 9, 13, 2, 5, 13, 2, 67, 23, 1, 2, 10, 1, 2, 1, 10, 2, 1)$$

Write down the list of satisfying triples contained in the sequence, as well as their associated characters, ordered by initial occurrence. What is the secret message? Fill in the table below.

You may not fill in all rows of the table, or you may need to add rows of the table.

Triple	$(a + b + c) \mod 27$	**Letter**	**Initial Occurrence**
$\{10, 13, 9\}$	5	f	13
$\{13, 5, 2\}$	20	u	18
$\{1, 2, 10\}$	13	n	29

(2) Now Bob has the courage to express his love to Alice. But Alice refuses him immediately, because she is tired of decoding his messages by hand. After that, Bob becomes very sad, knowing that Alice is very angry about the process finding all satisfying triples, which needs $O(n^3)$ time complexity. In order to help him cheer up, please design an expected $O(n)$ time algorithm to decode a sequence of n integers from Bob to output his secret message. Hint: Hash Table.

Solution:

Design a new hash function:

$$h(\{a, b, c\}) = (a + b + c) \mod 27$$

Since there's no consecutive sequence in the give message sequence that will add the occurrence times of hash result up to 6 even if it's actually not satisfied, create an array `Array[27]` initialized to all zeros to store the occurrence times of the hash result, i.e. if $h(\{a, b, c\}) = i \in [0, 26]$, then add `Array[i]` by 1.

Read start from the $0th$ entry of the message sequence and read 2 more integers consecutive to the first read integer, for example, for the first time, read $(10, 13, 9)$. Pass those three integers to the hash function defined upward and increase the value of corresponding array entry. Then read the next three integers from the $1st$ entry, for example, the second time reads $(13, 9, 10)$, invoke the hash function and increase the value of the array entry. Iterate $(n-2)$ times in total, in each time, if the value of corresponding array entry equals to 6 after increasing, push the 3 integers and corresponding information to the bottom of the table.

When the iteration ends, all characters has been pushed to the table and the order is reserved.

Analysis: $O(1)$ to create the array; for each iterate, $O(1)$ to read three integers, $O(1)$ to invoke the hash function, $O(1)$ to increase the array entry; Iterate $(n-2)$ i.e. $O(n)$ times. Therefore, the total time complexity is $O(n)$.

Problem 4: Inequal Divide

In this problem, we will analyze an alternative to divide step of merge sort. Consider an algorithm `InequalSort()`, identical to `MergeSort()` except that, instead of dividing an array of size n into two arrays of size $\frac{n}{2}$ to recursively sort, we divide into two arrays with inequal size. For simplicity, assume that n is always divisible by vivisors (i.e. you may ignore floors and ceilings).

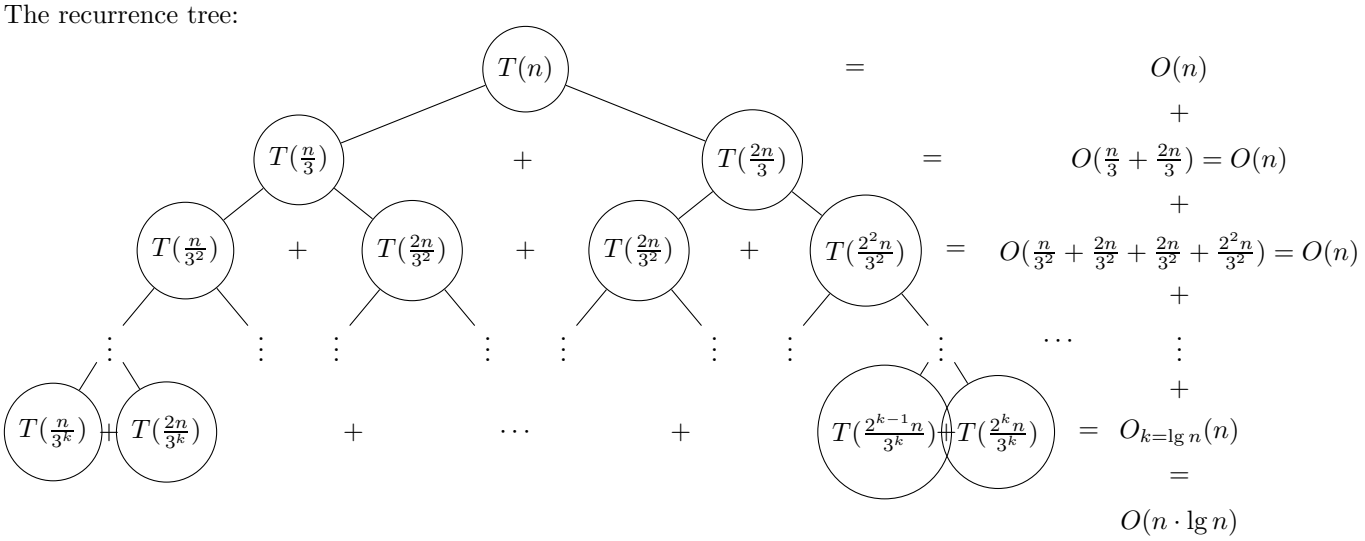
(1) **Analysis:** If we divide into two arrays with sizes roughly $\frac{n}{3}$ and $\frac{2n}{3}$. Write down a recurrence relation for `InequalSort()`. Assume that merging takes $\Theta(n)$ time. Show that the solution to your recurrence relation is $O(n \log n)$ by drawing out a recursion tree, assuming $T(1) = O(1)$. Note, you need to prove both upper and lower bounds.

Solution:

The recurrence relation is:

$$T(n) \;=\; T(\frac{n}{3}) \;+\; T(\frac{2n}{3}) \;+\; \Theta(n)$$

The recurrence tree:



Note that when the leftmost node has become $T(1)$, the rightmost node has not become $T(1)$ yet. When the leftmost node becomes $T(1)$, $k = \lg_3 n$, when the right most node becomes $T(1)$, $k = \lg_{\frac{3}{2}} n$, thus $k \sim O(\lg n)$. Since the sum of time for each level is $O(n)$, when the leftmost node has become $T(1)$, the time is $O(n \cdot \lg n)$; When the rightmost node has become $T(1)$, the time is at most $O(n \cdot \lg n)$. Therefore, the solution to the recurrence relation is $O(n \cdot \lg n)$.

(2) **Generalization:** If we divide array into two arrays of size $\frac{n}{a}$ and $\frac{(a-1)n}{a}$ for arbitrary constant $1 < a$ recursively, what is the asymptotic runtime of the algorithm? Is there any change on time complexity?

Solution:

Runtime: $O(n \cdot \lg n)$; No change on time complexity. Since the sum of time for each level is $O(n)$ and when the leftmost node has become $T(1)$ the number of total levels is $\lg_a n$ and when the rightmost node has become $T(1)$ the number of total levels is $O(\lg_{\frac{a}{a-1}} n)$, the total time is at least $O(n \cdot \lg n)$ and at most $O(n \cdot \lg n)$, therefore, the time complexity is still $O(n \cdot \lg n)$.

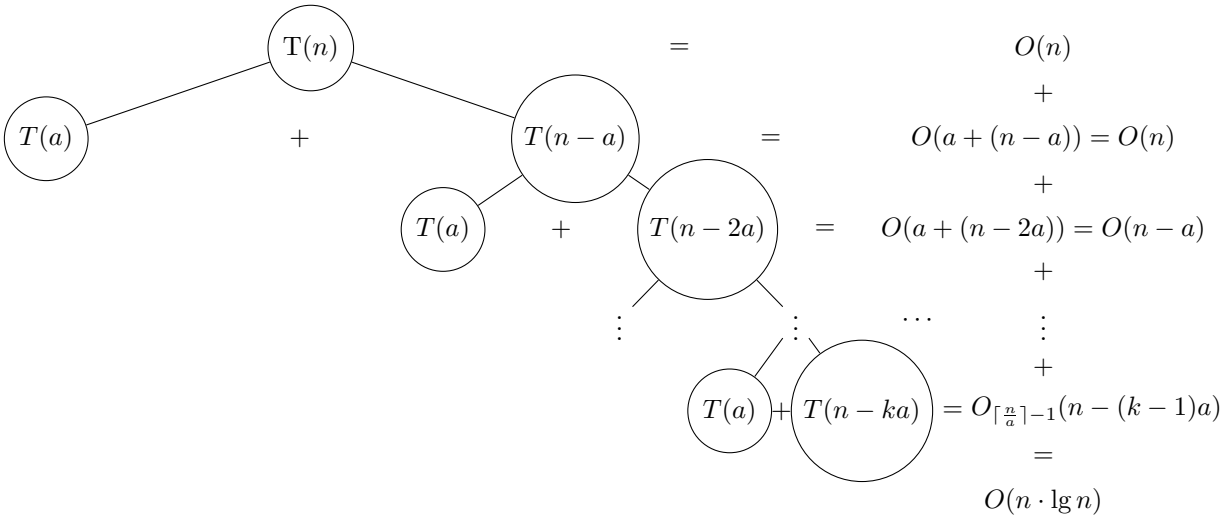
(3) **Limitation:** If we divide array into two arrays of size a and $n - a$ for some positive constant integer a recursively, what is the asymptotic runtime of the algorithm? Is there any change on time complexity? Assume that merging still takes $\Theta(n)$ time, $T(a) = O(a)$. It may help to draw a new recursion tree.

Solution:

Runtime: $O(n^2)$; Time complexity is higher.

The recurrence relation is:

$$T(n) \ = \ T(a) \ + \ T(n \ - \ a) \ + \ \Theta(n)$$

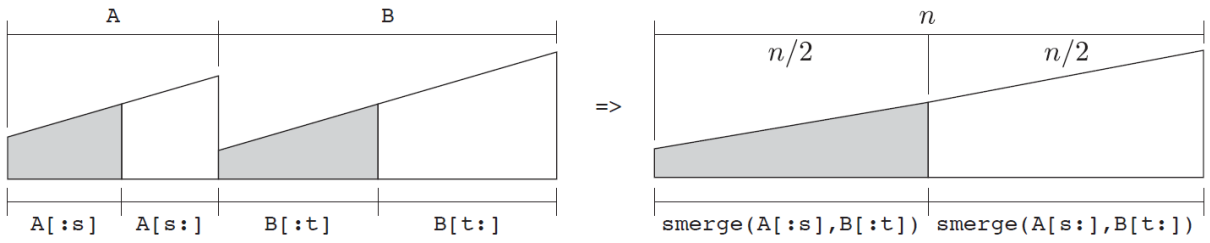


The time complexity can be calculated as:

$$T(n) \ = \ O(n) + O(\sum_{k=0}^{\lceil \frac{n}{a} \rceil - 1} (n - ka)) \ = \ O(n^2)$$

Problem 5: Slice Merge

In this problem, we will analyze an alternative to merge step of merge sort. 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 items have distinct keys. The slice merge algorithm, `smerge(A,B)`, merges A and B into a single sorted arrays as follows:



Step 1: Find indices s and t such that $s+t = \frac{n}{2}$ and prefix subarrays `A[:s]` and `B[:t]` together contain the smallest $\frac{n}{2}$ keys from A and B combined.

Step 2: Recursively compute `X = smerge(A[:s], B[:t])` and `Y = smerge(A[s:], B[t:])`, and return their concatenation `X + Y`, a sorted array consisting of all items from A and B.

For example, if $A = [1, 3, 4, 6, 8]$ and $B = [2, 5, 7]$, we 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 to find indices s and t satisfying step (1) in $O(n)$ time, using only $O(1)$ additional space beyond array A and B themselves. Remember to argue the correctness and running time of your algorithm.

Solution:

Create two pointers pointing to the first entries of the two arrays respectively, initialize both s and t to zero. Each time compare two values of these pointed values, increase s and move the pointer of A to the next entry if the value in A which is pointed is smaller, increase t and move the pointer of B to the next entry if the value in B which is pointed is smaller. Iterate until $s + t == \frac{n}{2}$.

The next smallest value of A and B combined can be found between the smallest values of remaining entries of A and B respectively. Since the two arrays A and B are already sorted, the smallest values of remaining entries of A and B are the first entries of A and B respectively, so comparing these two entries can get the next smallest value. Since s and t are increased within that process, s and t always indicates the numbers of smallest values chosen from A and B respectively.

In each iteration, the time for increase s or t is O(1) and the time for moving the pointer is also O(1), since the whole process iterates $\frac{n}{2}$ times, thus the total time is $O(n)$.

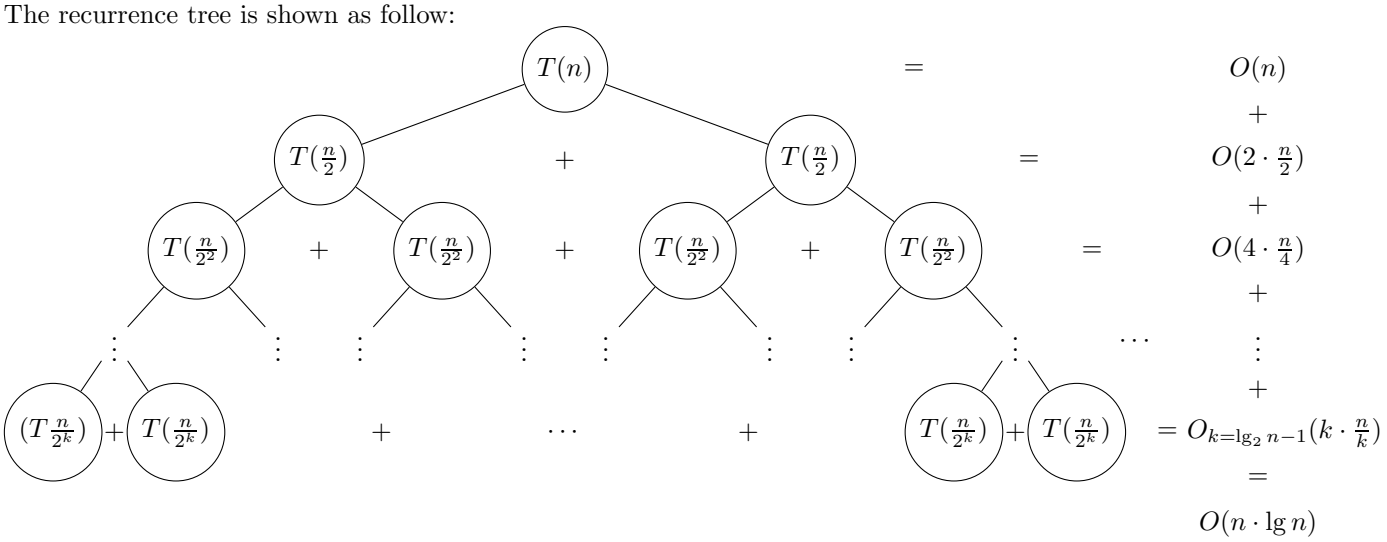
(2) Write and solve a recurrence for $T(n)$, the running time of `smerge(A,B)` when A and B contain a total of n items. Please show your steps. How does this running time compare to the `merge` step of `MergeSort()`?

Solution:

Since `smerge` needs to find s and t in $O(n)$, then invoke another two `smerge`s, the recurrence relation is:

$$T(n) \;=\; 2T(\frac{n}{2}) \;+\; O(n),\; T(2) \;=\; O(1)$$

The recurrence tree is shown as follow:



The runtime of `smerge` is $O(n \cdot \lg n)$, which is larger than the asymptotic runtime of `merge`.

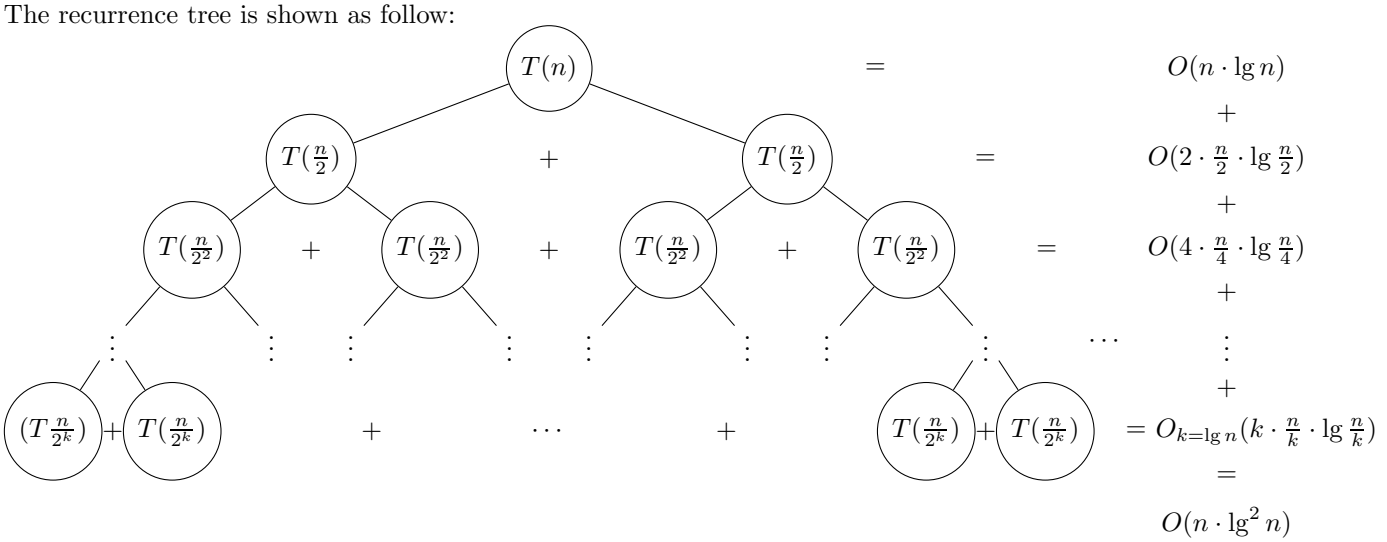
(3) Let `smerge_sort(A)` be a variant of `MergeSort(A)` that uses `smerge` in place of `merge`. Write and solve a recurrence for the running time of `smerge_sort(A)`. Please show your steps.

Solution:

The recurrence relation is:

$$T(n) \;=\; 2T(\frac{n}{2}) \;+\; O(n \cdot \lg n),\; T(1) = O(1)$$

The recurrence tree is shown as follow:



Thus the runtime complexity of `smerge_sort` is $O(n \cdot \lg^2 n)$.