Question 1. (5 points) AVL tree mechanics

1. (3 points) Only the final tree is shown for checking: fig. 1

Figure 1: After insertion

```
4
/   \
2     7
/ \   / \ 
1  3  6   9
  \   \  
   5   8
    \  
     \ 
      10
```

2. (2 points) Only the final tree is shown for checking: fig. 2

Figure 2: After deletion

```
7
/   \
2     9
/ \   / \ 
1  3  8  10
```

Question 2. (10 points) Very sparse numbers

1. (3 points) Let us try to find a recursion for $s_k$.
   We first see that any sparse number with $k$ bit length has to have the the following form
   \[10b_{k-2}b_{k-3} \ldots b_1\]
Now we can see if the first 1 after the leading digit is \( b_1 \) then there are \( s_i \) ways of filling the remaining digits. And if there are no further 1’s in the representation, the only way is to have the complete 0 string. This gives us the recursion

\[
s_k = \sum_{j=1}^{k-2} s_j + 1
\]

Now we can calculate \( s_k \) by calculating each of the previous \( s_i \) for \( i \in \{1, \ldots, k-2\} \) and keeping them in an array and also keeping track of the running sum. As we are keeping track of the sum we can calculate \( s_k \) in \( O(k) \), as shown by the algorithm in algorithm 1.

We see that as we are keeping track of a running sum we don’t need to do a lot of calculations.

**Algorithm 1** Calculate \( s_k \)

1: \( s = [1,1] \) # \( s \) has 1 based indexing
2: \( \text{sum} = 1 \)
3: for \( i = 3 \) to \( k \)
4: \( \text{sum} \leftarrow \text{sum} + s[i-2] \)
5: \( s.append(\text{sum}) \)
6: end for

Now we can calculate \( s_k \) from the previous values. Because there is only a single for loop, the complexity of the algorithm is \( O(k) \).

2. (2 points) Similar to the analysis above, we see that in the representation of \( s_k \), the last \( k - 2 \) free bits, \( b_{k-2}b_{k-3} \ldots b_1 \), can form any sparse binary string of length exactly \( k - 2 \) which is given by \( r_{k-2} - r_{k-3} \). Hence we have the recursion

\[
s_k = r_{k-2} - r_{k-3}
\]

3. (3 points) We will prove by induction that \( r_k - r_{k-1} := t_k = Fib(k + 2) \), where \( Fib(0) = 0, Fib(1) = 1, Fib(i) = Fib(i - 1) + Fib(i - 2) \). Here \( t_k \) represents all sparse string of length exactly \( k \).

For the base cases, observe that

(a) for \( i = 1 \), \( t_1 = 2 \) as the two strings 0, 1, are sparse
(b) for \( i = 2 \), \( t_2 = 3 \) as the three strings 00, 01, 10, are sparse.

**Note 2.1.** It is important to note that there are 2 base cases that we need to prove as we will be using the Fibonacci relation on \( t_k \) which needs the first two values of the numbers to be given to us.

Now suppose that the induction hypothesis is true for \( i = k - 1, k - 2, \ldots, 1 \).

Let the binary representation of a sparse string of length \( k \) be of the form \( b_kb_{k-1} \ldots b_1 \). We now have two choices for \( b_k \)

(a) \( b_k = 1 \), then \( b_{k-1} = 0 \), which leaves us free to fill the rest of the \( k - 2 \) bits with any sparse string. The total number of sparse strings in this setting is \( t_{k-2} \).
HW 1

(b) \( b_k = 1 \), then we are free to fill the rest of the \( k - 1 \) bits with any sparse string. The total options available in this case are \( t_{k-1} \).

Hence the total number of ways of filling up this string are

\[
t_k = t_{k-1} + t_{k-2}
\]

This, along with the base cases, gives us the relation to the Fibonacci numbers that we desired, thereby proving our induction hypothesis.

4. (2 points) In class we showed that we can calculate the \( k \)'th Fibonacci number using matrix multiplication and fast exponentiation, in \( O(\log k) \). As we showed that \( s_k = t_{k-2} = \text{Fib}(k) \), we can calculate \( s_k \) in \( O(\log k) \).

Question 3. (10 points) Recursive permutations

1. (5 points) We will write a general algorithm which takes as input \( n \) numbers \( \{a_1, \cdots, a_n\} \) and a number \( k \) and finds the \( k \)'th permutation of \( \{a_1, \cdots, a_n\} \) in \( O(n^2) \).

We first make some observations.

- As we are sorting the permutations by dictionary ordering, we have that the first set of permutations will begin in \( a_1 \), then a set of permutations beginning in \( a_2 \) and so on and so forth.
- As the total number of permutations are \( n! \) and we have an equal number of permutations starting at each index \( a_i \), there are \( \frac{n!}{n} \) permutations beginning in each index.
- Hence, if we want the \( k \)'th permutation, we first need to find the starting index of the permutation, which can found as the smallest \( i \) such that
  \[
k \leq (n-1)! \cdot i.
  \]

This gives us a recursive algorithm to calculate the \( k \)'th permutation of \( \{a_1, \cdots, a_n\} \) as follows

- Find the starting index of the \( k \)'th permutation of \( \{a_1, \cdots, a_n\} \), call it \( a_i \).
- We now have that there were \( (n-1)! \cdot (i-1) \) permutations that occur before the \( k \)'th one, in the dictionary ordering, that did not start with \( a_i \).
- Hence, given that we have found the starting index, \( a_i \), the \( k \)'th permutation of \( \{a_1, \cdots, a_n\} \) is the same as the \( k - (n-1)! \cdot (i-1) \)'th permutation of \( \{a_1, \cdots, a_n\} \setminus \{a_i\} \), concatenated with \( [a_i] \).

This gives us the following recursive algorithm in algorithm 2.

At each call of \( \text{KthPerm}(\cdot, \cdot) \) we have to do \( n \) operations to find the correct starting index, and there are \( O(n) \) recursive calls, which gives us a running time of \( O(n^2) \).

2. (5 points) We will give an algorithm which takes input \( [a_1, \ldots, a_n] \) and finds the position of this permutation in the sorted list of all permutations.

Similar to the above algorithm, we first find out the number of permutations which start with \( a_j \) such that \( a_j < a_1 \), and then we will recursively solve the problem for \( [a_2, \ldots, a_n] \). For this, notice that there are \( (n-1)! \cdot (a_1 - 1) \) permutations which begin with a number smaller than \( a_1 \), hence we get the following recursive algorithm in algorithm 3.

Here we notice that at each recursive call we have to find the index of \( a_1 \) in the sorted order which takes \( O(n) \) time, and we have to do \( O(n) \) recursive calls, which gives us a total time complexity of \( O(n^2) \).
Algorithm 2 $k$’th permutation of $\{a_1, \cdots, a_n\}$

```
function KTHPERM($\{a_1, \cdots, a_n\}$, $k$)
    if $n = 1$ then
        return $[a_1]$
    else
        for $i = 1$ to $n$:
            if $k < (n-1)! \cdot i$ then
                return $[a_i]$ ++ KTHPERM($\{a_1, \cdots, a_n\}$ \ $\{a_i\}$, $k - (n-1)! \cdot (i-1)$)
                # ++ denotes the concatenation of lists
            end if
        end for
    end if
end function
```

Algorithm 3 Position of permutation $[a_1, \ldots, a_n]$

```
function PERMPOSITION($[a_1, \ldots, a_n]$)
    if $n = 1$ then
        return 1
    else
        ind ← index of $a_1$ in the sorted order of $[a_1, \ldots, a_n]$
        return $(n-1)! \cdot (\text{ind} - 1) + \text{PermPosition}([a_2, \ldots, a_n])$
    end if
end function
```
Question 4. (15 points) Uneven tree updates

1. (5 points) We look at the preorder traversal of the tree. We can see that for any node, the subtree of a node will be a contiguous sublist of the preorder traversal of the tree. For the given tree we have the following preorder list.

\[
\begin{align*}
1 - (C : 5) & \quad 4 - (D : 20) \\
2 - (E : 4) & \quad 3 - (A : 10) \\
5 - (B : 2) &
\end{align*}
\]

In general, the preorder representation will have the following form, after a node,

\[
\cdots \text{Node( } \text{Child1(⋯), Child2(⋯), \cdots, ChildK(⋯) ),⋯} \]

Hence updating a subtree associated with a node, is same as updating a contiguous sublist of the preorder list of the tree.

2. (5 points) We can implement both operations in \(O(\log n)\) on the preorder list by storing the list as an AVL tree, called the PreorderTree, where nodes are inserted by position. E.g. for the given tree, store \([1 - (C : 5)], [2 - (E : 4)], [3 - (A : 10)], \cdots\) in the tree to get the representation as shown in fig. 3.

As we are also going to need to implement Increment\(,(\cdot,\cdot)\), we also need a carry variable at each node. Overall, each node will store the following information, along with the meta information needed to make it an AVL tree (e.g., height, left/right child, etc.),

- position: the position of the node in the preorder list
- key: the key associated with the node in the tree
- value: the value associated with the key in the tree
- carry: an extra variable to implement increment

3. (5 points) To implement the Increment\(,(\cdot,\cdot)\) operation on the preorder tree, we need the starting and ending indices of the subtree of each node in the preorder list. We can store these in another AVL tree, called the IndexTree, which is ordered by the keys of the original \(\cdots\).
Figure 4: Storing the endpoints of each subtree in an AVL tree

\[ B - (5 : 5) \]

\[ A - (3 : 3) \quad \rightarrow \quad D - (4 : 5) \]

\[ C - (1 : 5) \]

\[ E - (2 : 2) \]

We have already shown in class that both updateRange and find take \( O(\log n) \), when done on an AVL tree and both iTree, pTree are AVL trees.

Algorithm 4 Implementing queries on the tree: T

\[ \text{iTree} = \text{IndexTree}(T) \]
\[ \text{pTree} = \text{PreorderTree}(T) \]

\begin{verbatim}
function 
INCREMENT(key, value)
    (start, end) = iTree.find(key)
    pTree.updateRange(start, end, value)
    # Here the updateRange uses the carry method described in class
end function

function 
FIND(key)
    (start, end) = iTree.find(key)
    return pTree.find(start)
    # We return the value of node indexed with start because in the preorder of a tree the node
    # occurs before the subtree
    # Here the find method of the pTree adds up the carries on the path to the node indexed with start
end function
\end{verbatim}

Question 5. (10 points) Sweeping the plane

We keep an AVL tree of points \( \{p_i = (x_i, y_i)\}^n_{i=1} \), where we order them w.r.t their \( y \) coordinate. At the start we have an empty tree. We process the points in increasing order of \( x \) coordinates. Whenever we process a new point \( p_i \), the current tree will only have points which are to the left of \( p_i \). We now query for the number of points in the tree which are less than \( p_i \) in the tree ordering, which is the ordering by \( y \) coordinate.

Because the tree has all points to the left of \( p_i \) and we query for number of points less than \( p_i \) in \( y \) coordinate, we will get the points in the lower left quadrant, and the time take for each such query is \( O(\log n) \).

Then we insert \( p_i \) into the tree and process the next point.

Each step is \( O(\log n) \) and there are \( n \) steps, one for each point, giving a total time complexity of \( O(n \log n) \).
Question 6. (10 points) Linear equations

1. (a) \( x = 10 \)
   (b) \( x = 10 \)
   (c) \( x = 1, 5, 9, 13, 17 \)
   (d) No solution

2. \( 7^{69} = 7 \mod 100 \)

Question 7. (10 points) Cutting down costs
First let's look at the case when we only have two numbers in the array, \([a_1, a_2]\) and WLOG \( a_1 > a_2 \).
We will keep decreasing \( a_1 \) till we make it smaller than \( a_2 \), which would leave us with \([a_1 \% a_2, a_2]\) and we continue with decreasing \( a_2 \).
This is the implementation of Euclid's algorithm for GCD. Hence we see that we will reduce this array down to \([\gcd(a_1, a_2), \gcd(a_1, a_2)]\).
In general, we can see that if we have an array with \([a_1, a_2, \ldots, a_n]\), we will first reduce the first two elements to their gcd, and then the first three elements to their common gcd, and so on and so forth
\[
[a_1, a_2, a_3, \ldots, a_n] \rightarrow [\gcd(a_1, a_2), \gcd(a_1, a_2), a_3, \ldots, a_n]
\rightarrow [\gcd(a_1, a_2, a_3), \gcd(a_1, a_2, a_3), \gcd(a_1, a_2, a_3), a_4, \ldots, a_n]
\vdots
\rightarrow [\gcd(a_1, a_2, \ldots, a_n), \ldots, \gcd(a_1, a_2, \ldots, a_n)]
\]
If \( g := \gcd(a_1, \ldots, a_n) \), this gives us the sum \( n \cdot g \).
The claim is that we cannot reduce the sum any further. We prove the claim by showing that we cannot reduce any number in the array to a value smaller than \( g \).
Suppose that we have been able to reduce some index \( i \) to a value \( \alpha \). The only way we could have constructed \( \alpha \) was by performing the allowed operation on multiple elements of the list, which means that we can express \( \alpha \) as a linear combination of \( \{a_i\}_{i=1}^n \), say \( \alpha = \sum_{i=1}^n \beta_i a_i \). As we know that \( g | a_i \forall i \Rightarrow g | \sum_{i=1}^n \beta_i a_i \Rightarrow g | \alpha \Rightarrow \alpha \geq g \) or \( \alpha = 0 \). As we are only allowed to subtract smaller numbers from larger ones we cannot get any number less than or equal to 0, which means that \( \alpha \geq g \).

Question 8. (5 points) Geometric sums

1. (5 points)

\[
1 + a + a^2 + \cdots + a^{m-1} = \frac{a^m - 1}{a - 1} \mod n
= (a^m - 1) \cdot (a - 1)^{-1} \mod n
\]

We can calculate \((a^m - 1) \mod n\) in \(O(\log m)\), using the fast exponentiation technique and the inverse of \((a - 1) \mod n\) in \(O(\log n)\). Giving us an overall time complexity of \(O(\log m + \log n) = O(\log m)\).
2. (10 points) We want to calculate \( a^{m-1} \mod n \).
We will show how to calculate \( \frac{x}{y} \mod n \) given \( y|x \) and when \( x \) is too big to fit in memory but \( y \) is not.

We first calculate \( x \mod ny \). As we have that \( ny \) is small enough to fit in memory, we can calculate \( x \mod ny \), provided we have a method to calculate \( x \) in modular arithmetic.

Suppose we get the following representation of \( x \).

\[
x = q \cdot ny + r
\]

As we know that \( y|x \), we get that \( y|q \cdot ny + r \) \( \implies y|r \) \( \implies r = sy \) for some number \( s \).

Hence we get

\[
x = q \cdot ny + sy
\]

\( \implies x = sy \mod ny \)

\( \implies \frac{x}{y} = s \mod n \) we can remove common divisors from all three terms

Hence the algorithm is

(a) Compute \( (a^m - 1) \mod n(a - 1) \)

(b) Compute \( \frac{r_1}{(a - 1)} = r_2 \)

(c) Return \( r_2 \)

The only computationally heavy step is the first one, but we can calculate it using fast exponentiation, in \( O(\log m) \), which gives us the overall time complexity of \( O(\log m) \).