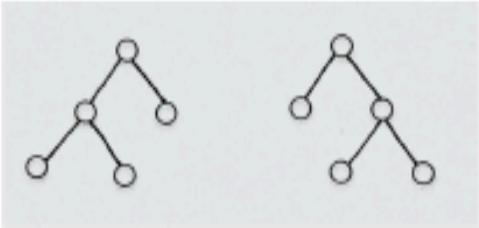
1. By Hanghang Wang

Question 1

(1) $F_5 = 2$



Schematic graph of F_5

(2) Only odd numbers of n will create full binary trees. The recurrence relation for F_n:

$$F_n = \sum_{i=1}^{\frac{n-1}{2}} F_{n-2i} F_{2i-1}$$

if $n = 4k+1$ where k is a positive integer,

$$F_n = 2(F_{n-2}F_1 + F_{n-4}F_3 + \ldots + F_{\frac{n+1}{2}} F_{\frac{n-1}{2}-1})$$

if $n = 4k+3$ where k is a positive integer,

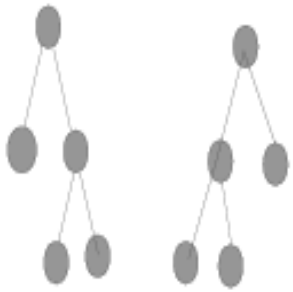
$$F_n = 2(F_{n-2}F_1 + F_{n-4}F_3 + \ldots + F_{\frac{n-1}{2}} F_{\frac{n-1}{2}}) - F_{\frac{n-1}{2}} F_{\frac{n-1}{2}}$$

F_n is a Catalan number in the form of $\frac{(2m)!}{m!(m+1)!}$ where $m = \frac{n-1}{2}$

Assume that $F_n \geq c \cdot 2^n$ for all of n, then $F_1 \geq c \cdot 2^1$, $F_3 \geq c \cdot 2^3, \ldots, F_n \geq c \cdot 2^n$
$F_{n+2} = F_nF_1 + F_{n-2}F_2 + \ldots + F_1F_n \geq c \cdot 2^n c \cdot 2^1 + c \cdot 2^{n-2} c \cdot 2^3 + \ldots + c \cdot 2^1 c \cdot 2^n = c^2 \cdot \frac{n+1}{2} \cdot 2^{n+1}$
$F_{n+2} \geq c^2(n+1) \cdot 2^n \geq c \cdot 2^{n+2}$ if $c \geq \frac{4}{n+1}$

By Rupert Freeman

1. (a) $F_5 = 2$, both possible trees are shown.



(b) First observe that we can only form a complete binary tree with an odd number of vertices. We can form a complete binary tree by taking a root node and attaching two complete binary trees as its children. For a binary tree with n vertices, the total number of vertices in the smaller binary trees must be $n-1$. We can attach a binary tree of size $n-2$ as the left child in F_{n-2} ways and a binary tree of size 1 as the right child in F_1 ways. We could also attach the tree of size 1 as the left child and of size $n-2$ as the right child. Thus there are $2 \cdot F_{n-2} \cdot F_1$ ways to do this. The same argument can be applied to child trees of size $n-3$ and 2, $n-5$ and 4 etc until we reach $\frac{n-1}{2}$. When we attach two child trees of size $\frac{n-1}{2}$ we can not obtain a new tree by reversing the left/right positions, so there are only $F^2_{\frac{n-1}{2}}$ ways to do this.
Thus we have the recurrence relation

$$F_n = 2F_{n-2} \cdot F_1 + 2F_{n-4} \cdot F_3 + \ldots + F^2_{\frac{n-1}{2}}$$

For $1 \leq n \leq n_0$, for some appropriate n_0, we can choose a $c > 0$ such that $F_n \geq c2^n$. Now suppose $F_k \geq c2^k$ for $k < n$. Then

$$\begin{aligned} F_n &\geq 2c2^{n-2}2 + 2c2^{n-4}2^3 + \ldots + c(2^{\frac{n-1}{2}})^2 \\ &= c2^n + c2^n + \ldots + c2^{n-1} \\ &\geq c2^n \end{aligned}$$

Therefore F_n is $\Omega(2^n)$.

2. By Hanghang Wang

Question 2

Algorithm

This algorithm uses MergeSort to recursively sort left and right arrays with the exception that when duplicate numbers are encountered during merging, e.g. $L[i] = R[j]$, the iteration will skip the duplicate and continue to compare $L[i+1]$ with $R[j]$.

Proof of Correctness

The Merge procedure proceeds by merging $A[p..q]$ and $A[q+1..r]$ which are in sorted order into a single sorted subarray that replaces the current subarray $A[p..r]$. Similar to the Merge algorithm, this algorithm maintains the loop invariant. Let $L[1..n_1+1]$ and $R[1..n_2+1]$ be new arrays where $n_1 = q-p+1$ and $n_2 = r-q$. Initialization: prior to the first iteration of the loop, k = p and the subarray A[p..k-1] is empty. $L[1]$ and $R[1]$ are the smallest elements of their arrays that have not been copied back into A. Maintenance: when $L[i] < R[j]$, $L[i]$ is the smallest element not yet copied back into A, which currenlty contains the $k-p$ smallest elements and after copying, the subarray $A[p..k]$ will contain the $k-p+1$ smallest elements. When $L[i] > R[j]$ it is executed similarly. And when $L[i] = R[j]$, $L[i]$ is skipped over and comparison continues between $L[i+1]$ and $R[j]$. Termination: at termination,$k = r+1$. The subarray $A[p..k-1]$ is $A[p..r]$ and contains the $k-p = r-p+1$ smallest elements of $L[1..n_1]$ and $R[1..n_2+1]$ in sorted order without duplicates. All but the two largest elements (the sentinels) have been copied back into A.

Complexity Analysis

This algorithm has the same complexity as MergeSort and therefore complexity of $O(nlogn)$.

By Rupert Freeman

2. Proceed identically to MERGE-SORT, except that when combining lists we need to check that the element being added to the combined list is not the same as the element added previously to it. This algorithm will produce a (sorted) list with no duplicates in the same running time as MERGE-SORT, $O(n \log n)$.

3. By Merve Cakir

QUESTION 3

To solve this problem, we can follow a reasoning similar to well-known BINARYSEARCH algorithm. BINARYSEARCH algorithm compares a given key with the middle element of an array and if element is not found, search space is decreased to the half of original array, focusing only on the subarray that has the possibility of containing the key. For this specific case, we do not have a particular key in our hand to search for but we can use the index of the middle element as the search key. If it is not equal to the middle element, then the algorithm will select one half of the array and continue the search in that half. Pseudocode is given below:

Algorithm 3 SEARCHINDEX(A, $start$, end)

```
1: if end < start then
2:     return null
3: else
4:     middle = ⌊(start + end)/2⌋
5:     if A[middle] < middle then
6:         SEARCHINDEX(A, middle + 1, end)
7:     else if A[middle] > middle then
8:         SEARCHINDEX(A, start, middle − 1)
9:     else
10:        return middle
```

As it can be seen from the pseudocode, if the middle element is smaller than the key, then we select the upper half of the array that contains values larger than it. Similarly, if the middle element is greater, then the lower half with smaller values becomes the new

search space. The algorithm continues its search until an equality is found or it can return an empty result if whole subarrays are exhausted and no equality is observed.

To prove that this reasoning is correct we need to analyze the choice that is done within the comparison portion. If the compared key item is smaller than the middle element, we continue with the left half of the array. This is the correct choice because we start with a sorted array and therefore the right half of the array will contain elements that are larger than the middle element. But our key is smaller than the middle element, logical choice is therefore to search for this smaller element in the left half of the array. Same logic applies to the case where middle element is greater than the searched key. In this case, we need to continue our search within the values that are greater than the middle element and only the right half of the array contains these values. This analysis shows us that the choices we make before recursive calls are correct.

We also need to prove that this algorithm runs in $O(\log n)$ time. Let's analyze the recursive nature of the algorithm to find out the recurrence relation. The recursive call is observed when the array is divided into two and then only one half is used to perform next search. We have $T(n/2)$ from this part of the algorithm. The remaining operations are trivial comparisons which run in $O(1)$ time. So the recurrence relation is as follows: $T(n) = T(n/2) + 1$.

Master Method can be used to solve this recurrence. As it can be seen $a = 1$ and $b = 2$. This means that we need to compare $n^{log_2 1}$ with $f(n) = 1$. $n^{log_2 1} = n^0 = 1$, which means that these two are equal. As a result, Case 2 of Master Method is applied to get the result of recurrence: $T(n) = T(n/2) + 1 = \log n$. So the proposed algorithm satisfies the given requirement for time bound.

4. By Rupert Freeman

4. It was shown in class that we can use counting sort to sort in $\Theta(n+k)$ time, where n is the size of the array and $A[j] \in \{1, ..., k\}$ for all $j \in \{1, ..., n\}$. Form a new array B such that $B[j] = A[j] - MIN$ for all j. Now for the new array B, we have that $k = MAX - MIN$. So we can sort B in $O(n + (MAX - MIN))$ time and we have not altered the ordering of the elements. Now re-adjust by adding MIN to each element and we have sorted A. Since the adjustments by MIN take only $\Theta(n)$ time, the overall sort is done in $O(n + (MAX - MIN))$ time.

5. By Chaoren Liu

We have two sorted array a[1:n_1] and b[1:n_2] and want to find the ith element in union list.
First we can ignore $a[i+1:n_1]$ and $b[i+1:n_2]$.
Step 1: we compare the element $a(\lfloor i/2 \rfloor)$ with $b(i-\lfloor i/2 \rfloor)$. If we have $a(\lfloor i/2 \rfloor) = b(i-\lfloor i/2 \rfloor)$, then return $a(\lfloor i/2 \rfloor)$. If $a(\lfloor i/2 \rfloor)$ is smaller than $b(i-\lfloor i/2 \rfloor)$, then we have a_{new}=a[1:i] and b_{new}=$b(1:i-\lfloor i/2 \rfloor)$.
Otherwise we have a_{new}=$a(1:\lfloor i/2 \rfloor)$ and b_{new}=b[1:i].
Step 2: If the in step 1, we have a_{new}[1:i] and $b_{new}(1:i-\lfloor i/2 \rfloor)$, we compare $b_{new}[b_{new}(middle)]$ with a[i-$b_{new}(middle)$].
If b[$b_{new}(middle)$] is bigger than a[i-$b_{new}(middle)$], then we have a_{new}=a_{new} and b_{new}=b_{new}[1:$b_{new}(middle)$].
Otherwise a_{new}=$a_{new}[1:i-b_{new}(middle)]$ and b_{new}=b_{new}.
Keep going on...
Every time compare the middle element(t^{th}) of the list, for example a_{new},

whose length is changed last time with the element $(i-t)^{th}$ of the other list, until finding they are the equal or one of a_{new} and b_{new} have the length of 1. We find the ith element.
Complexity analysis:
Because every recursion step, we shrink a_{new} or b_{new} by a half. The total complexity is $\Theta(logi+logi) = O(\log n_1 + \log n_2)$.

By Rupert Freeman

5. Call the two lists A and B. We want to find j, k such that $A[k] \leq B[j] \leq A[k+1]$ (or vice versa) such that $k + j = i$. The i-th element will then be $B[j]$. This is because there are k elements smaller than it in list A and it is the j-th largest element in list B. Thus, since $j + k = i$, it is the i-th smallest element in the combined list. This proves correctness for the algorithm I present.
   Choose $B[\frac{i}{2}]$. I ignore the possibility of non-whole numbers but if they occur we may simply take the floor or ceiling. Compare it to $A[\frac{i}{2}]$ and $A[\frac{i}{2}+1]$. If $B[\frac{i}{2}] > A[\frac{i}{2}+1]$ then consider $B[\frac{i}{4}]$ and compare it to elements of the A list such that the indices are correct for testing the stated condition. If $B[\frac{i}{2}] < A[\frac{i}{2}]$ then we will need to test $B[\frac{3i}{4}]$. We can continue like this, testing intermediate elements of the B list, until we either find an element which satisfies the condition or no intermediate elements remain in which case we may conclude that the i-th element comes from the A list instead. If this is the case, then repeat the process with the A list.
   Due to the successive halving of the entries of interest in each list, we only have to make at most $2 \log n$ comparisons per list (since for each $B[j]$ we compare it to $A[k]$ and $A[k+1]$). Thus the run time of the algorithm is $O(\log n_1 + \log n_2)$.