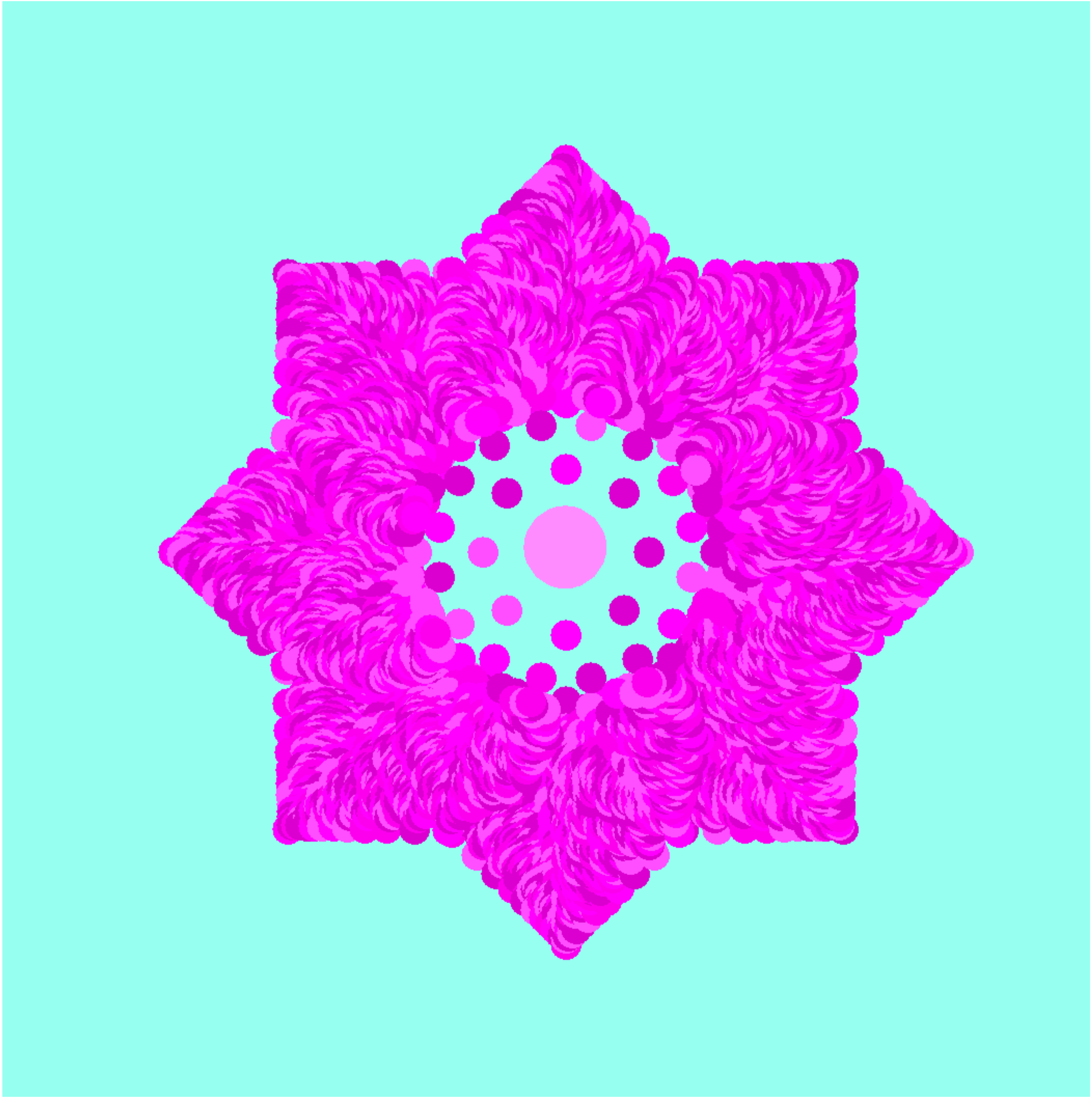
# Quick Sort

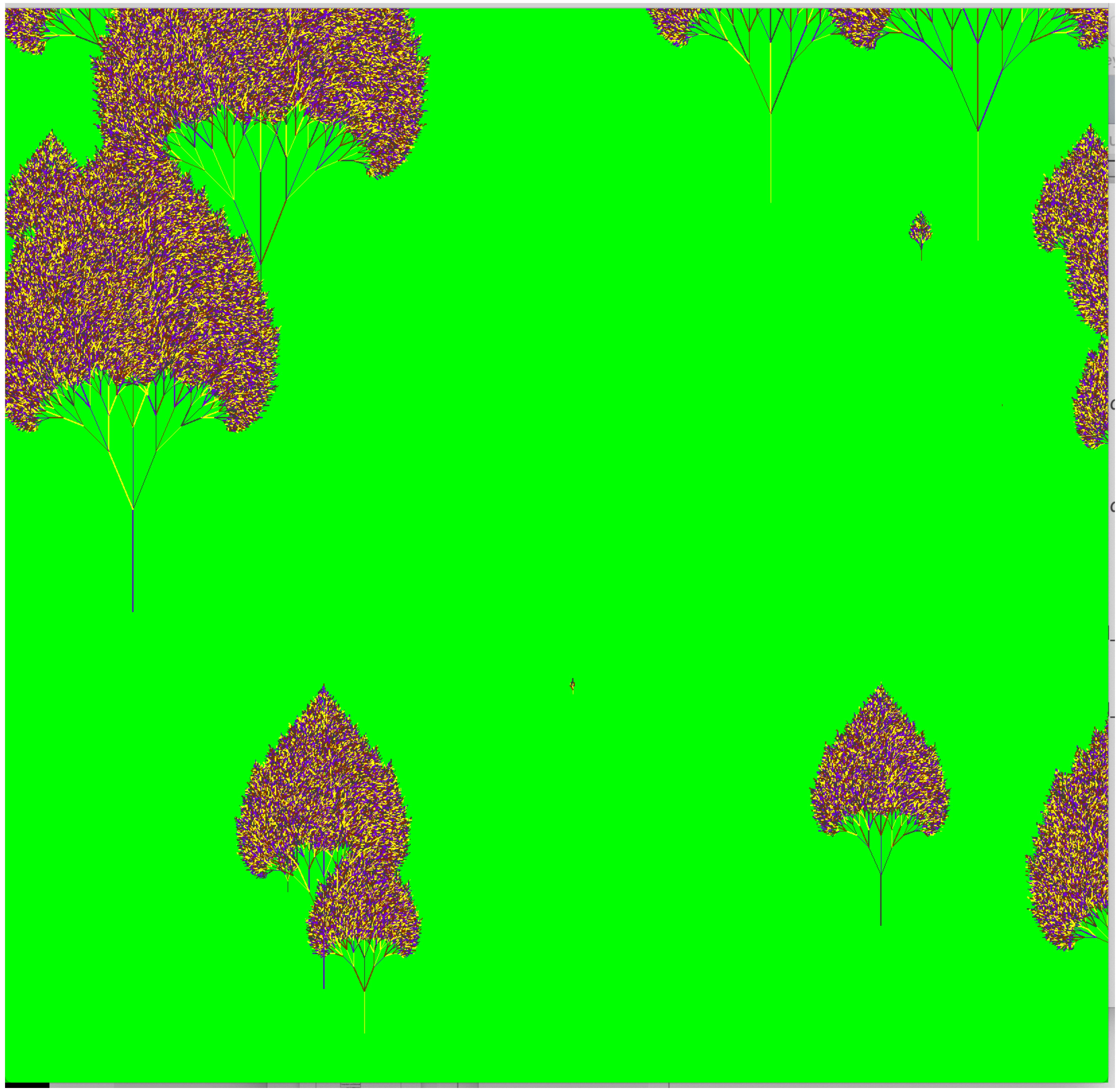
## And Trees!

# 1



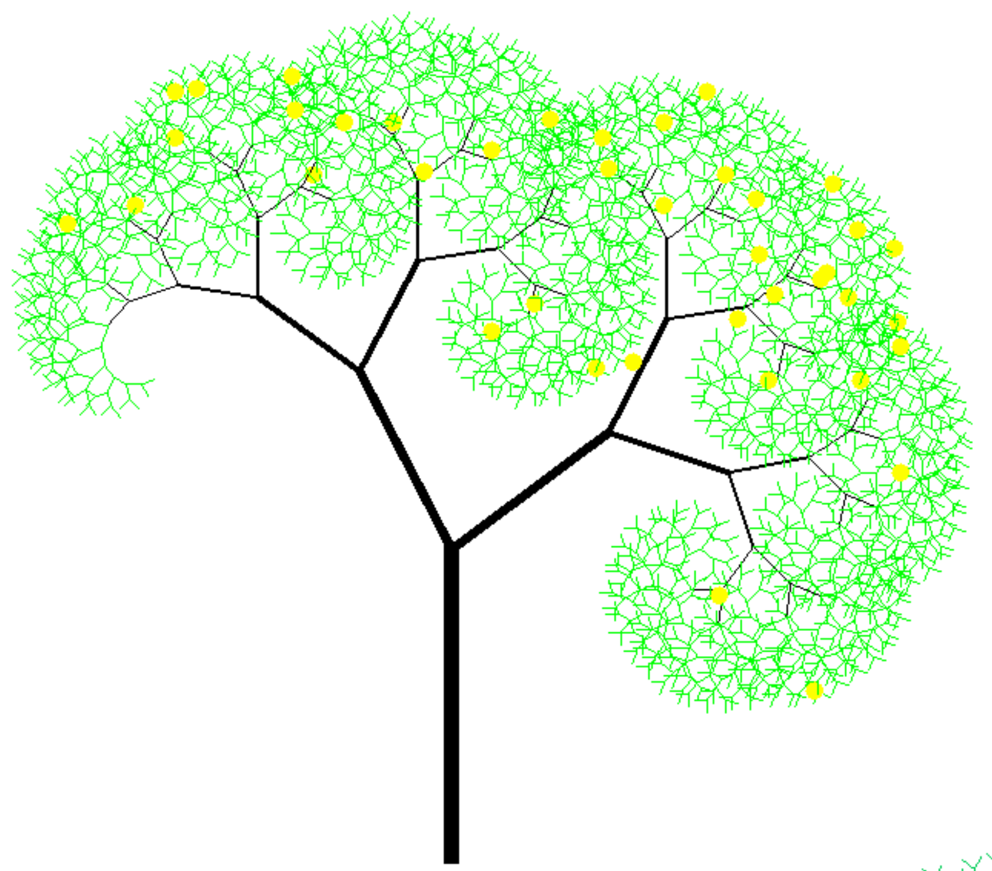
2

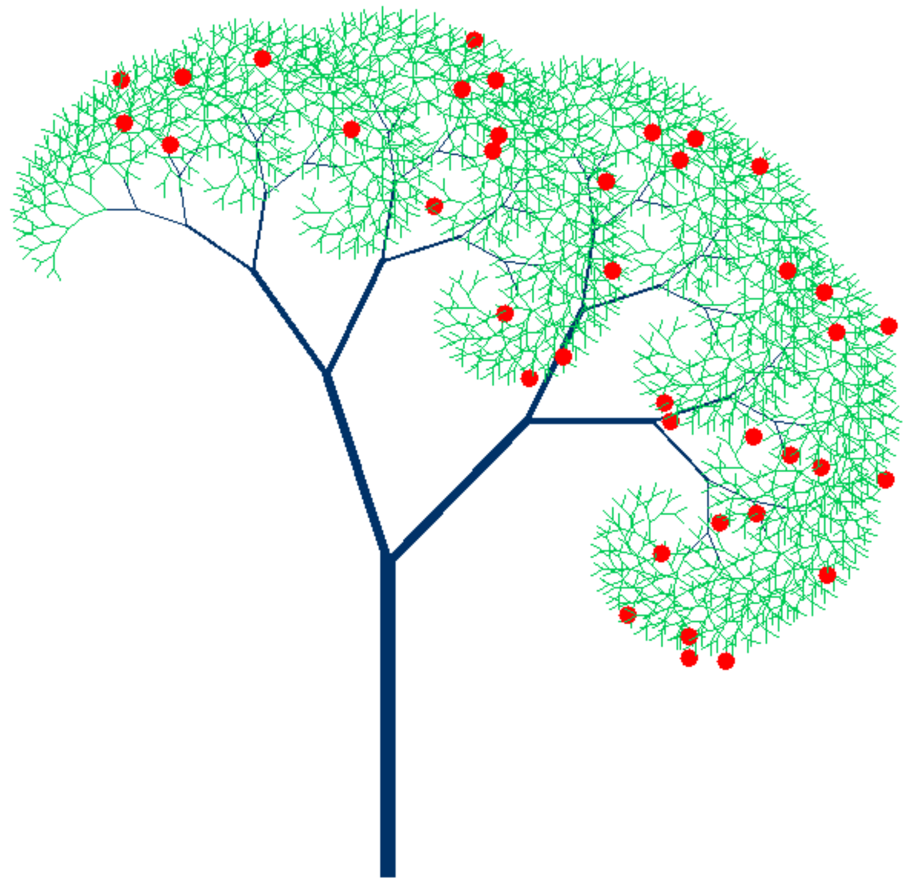
3

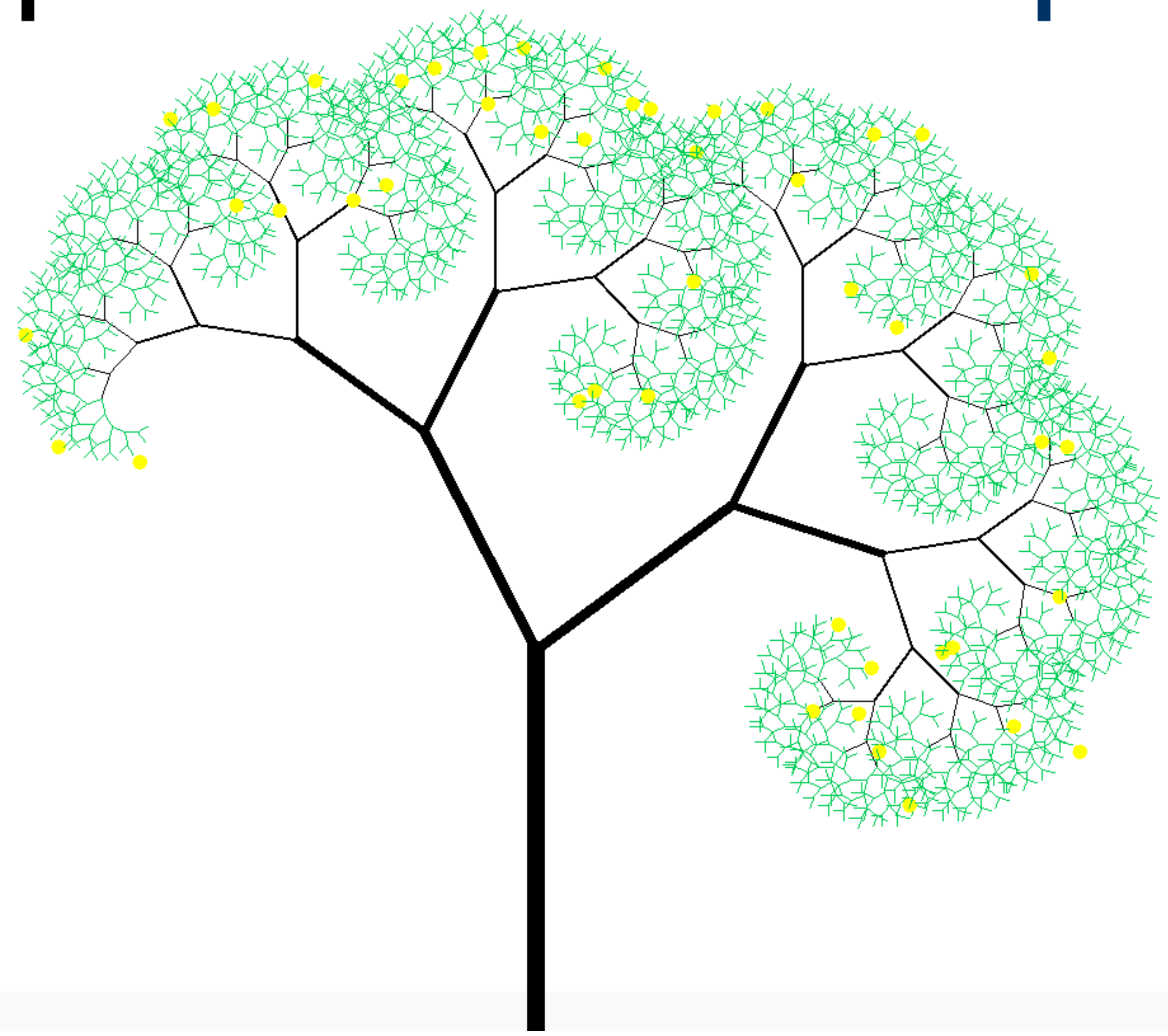


# 4

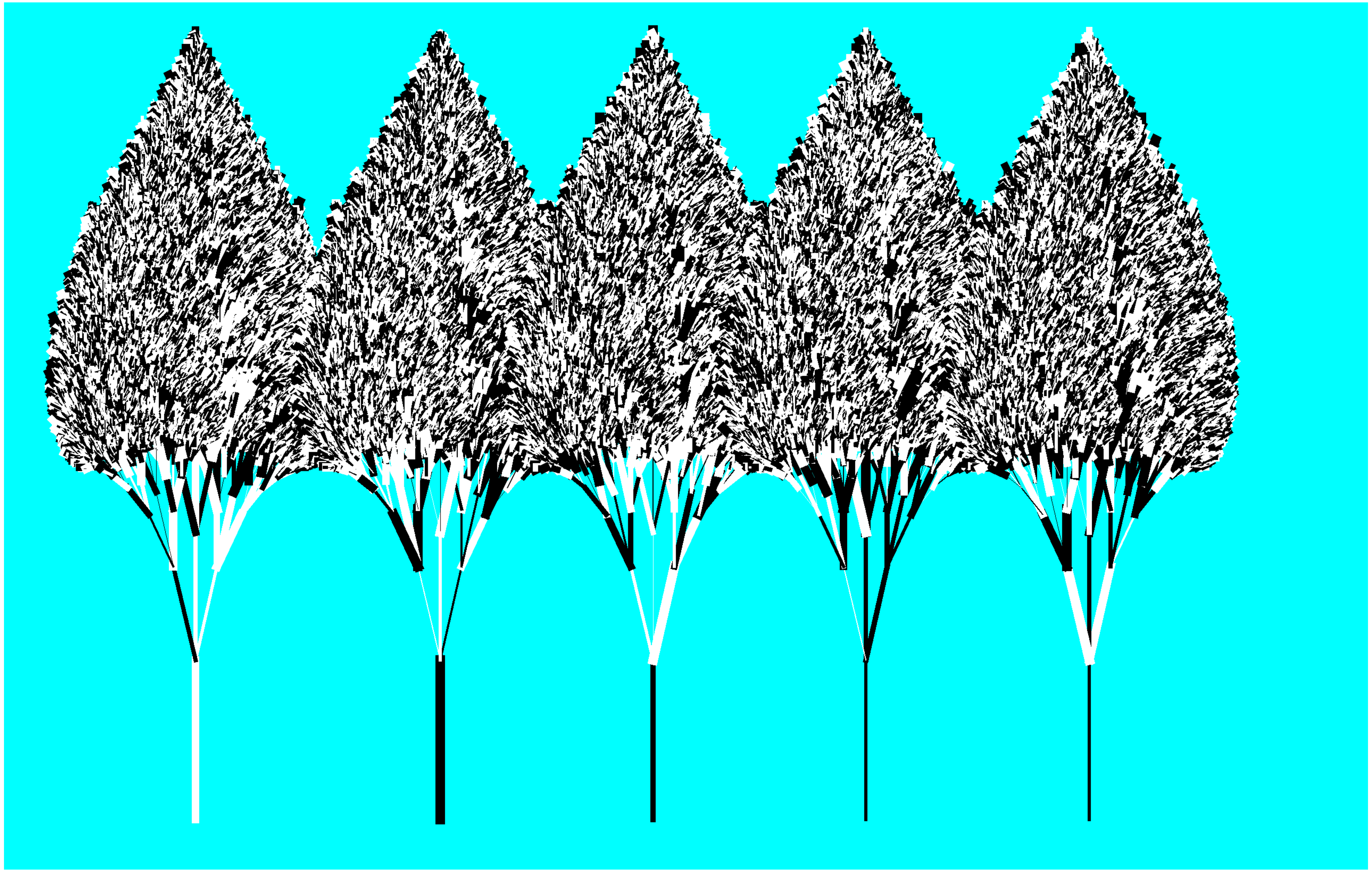
# 5



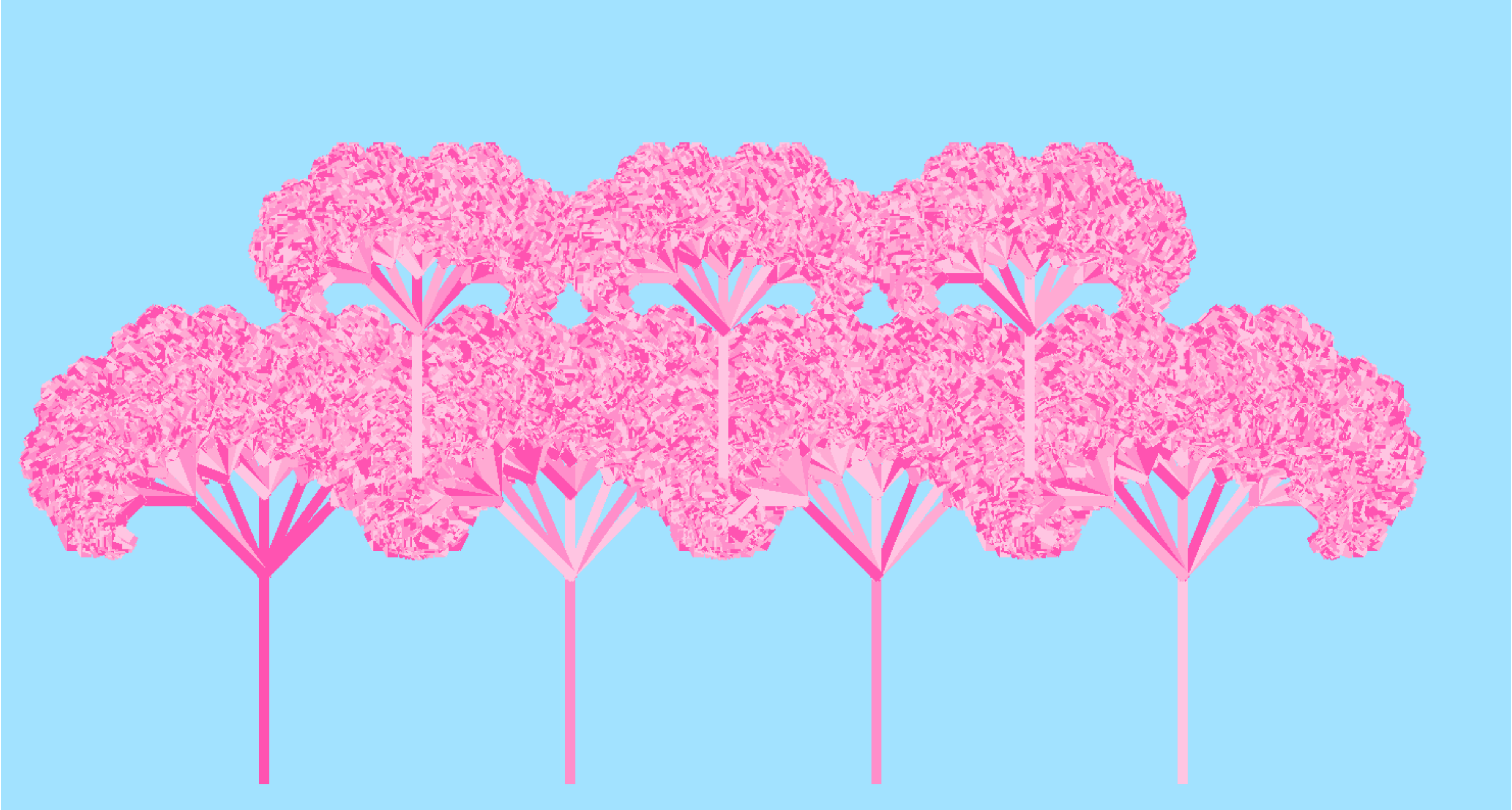




6



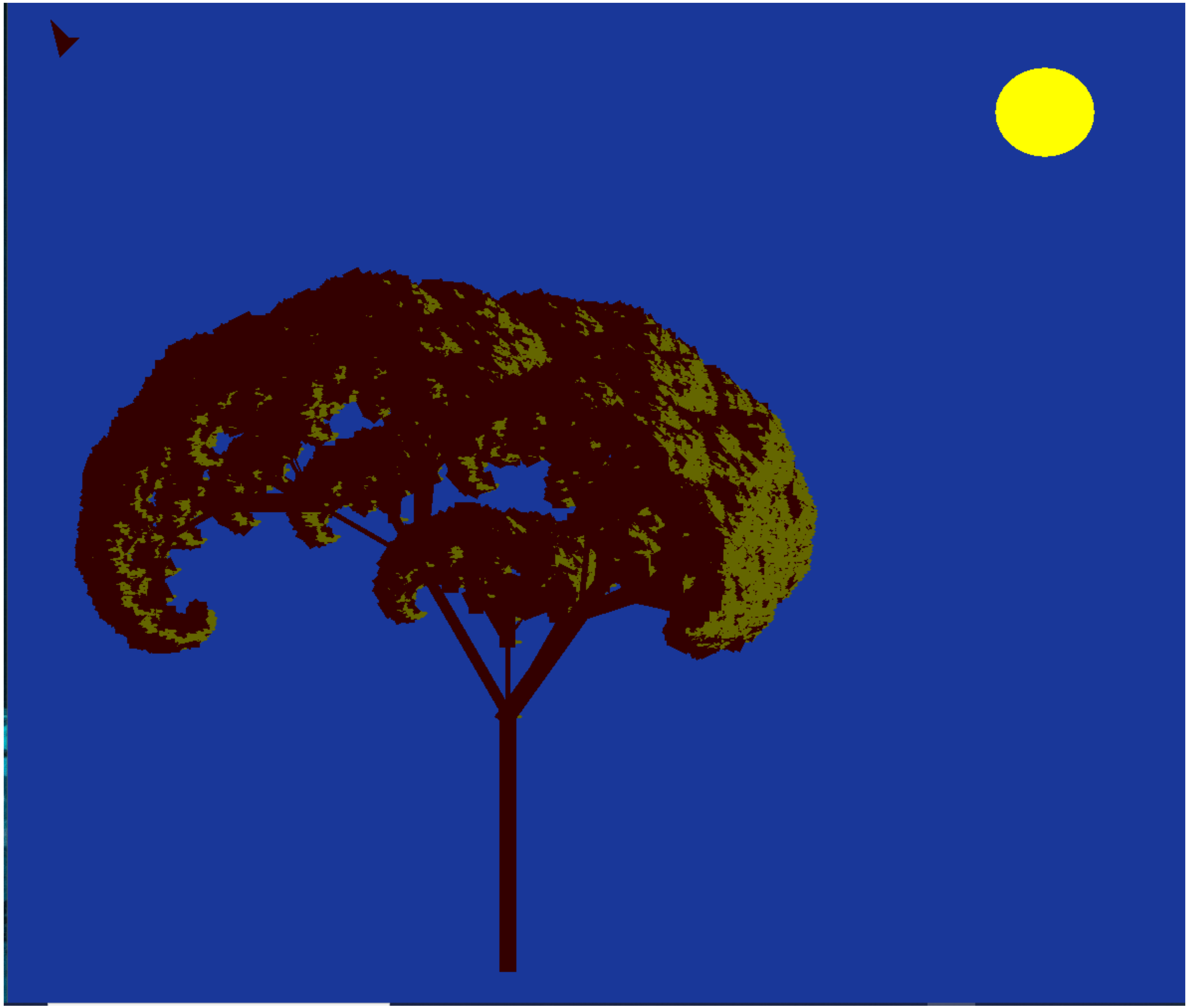
7



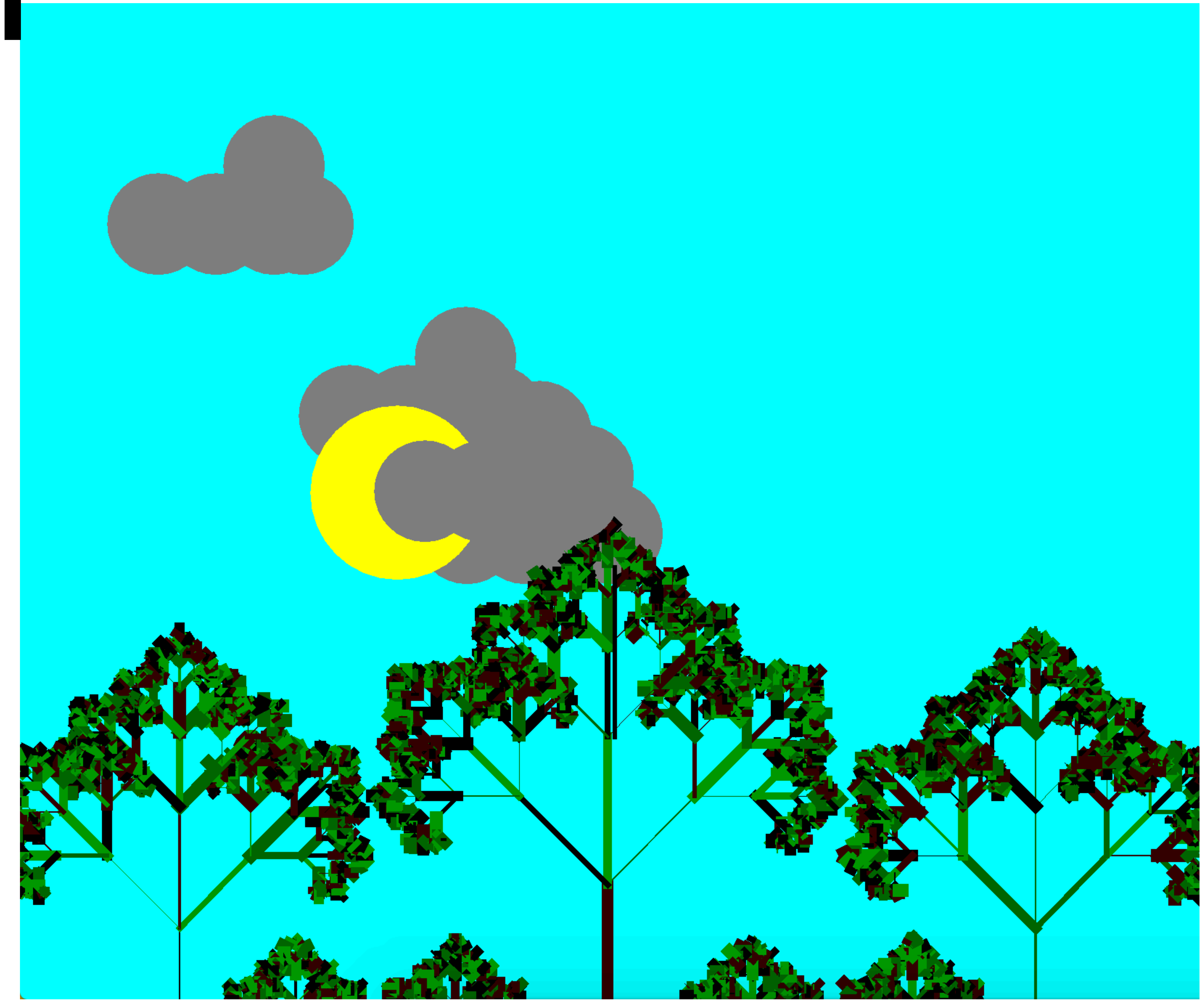
# 8

# 9

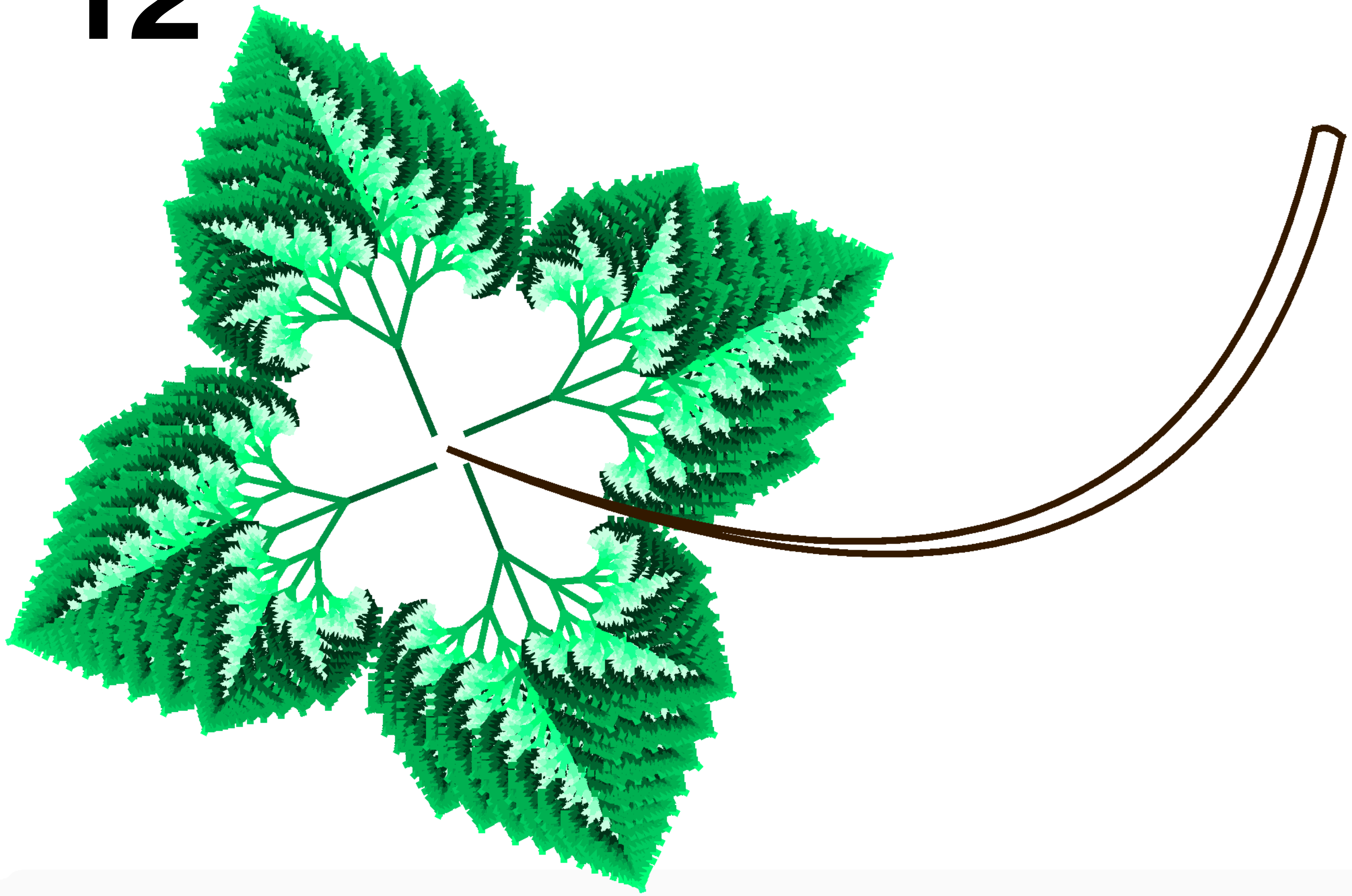
10



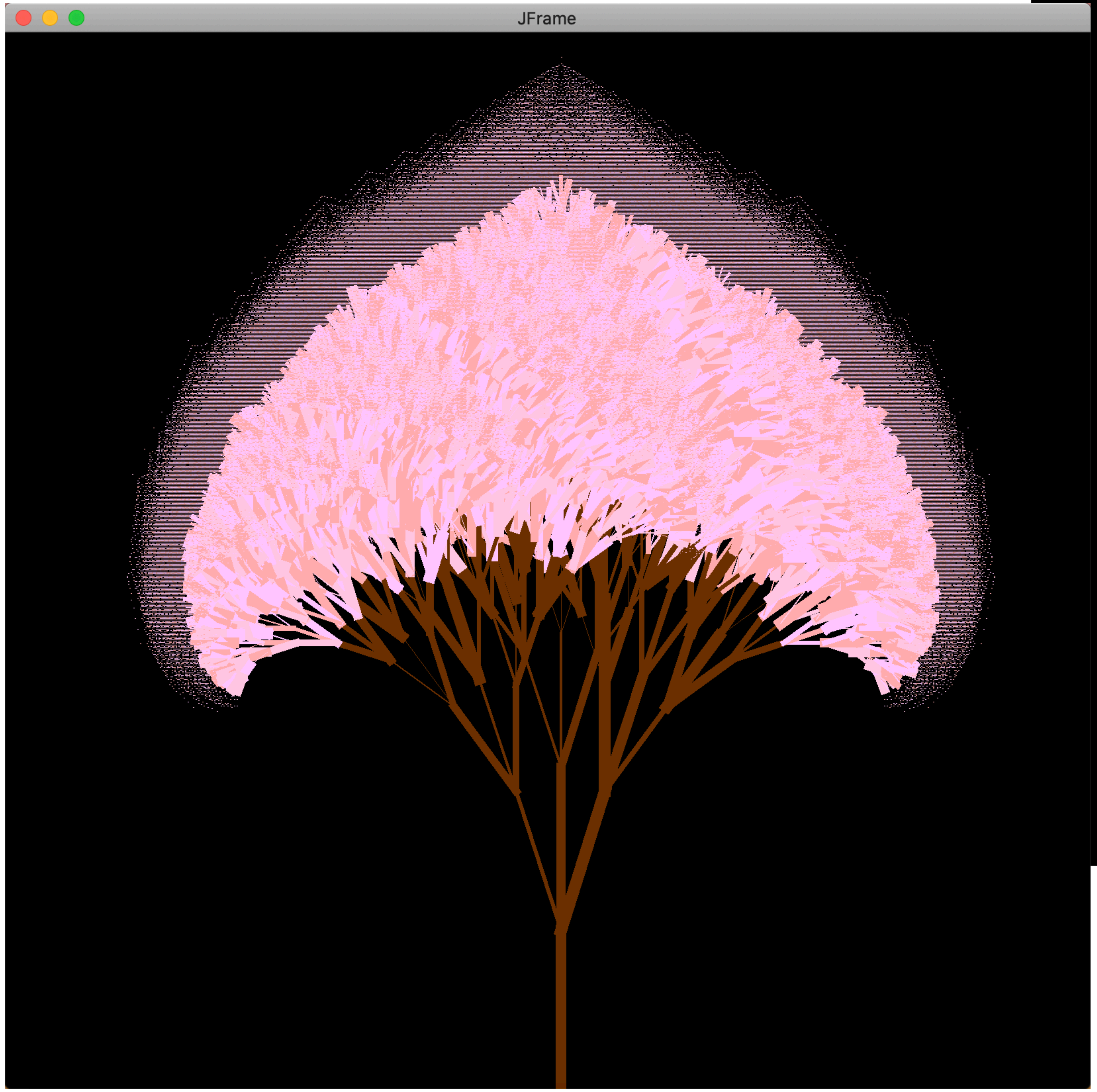
11



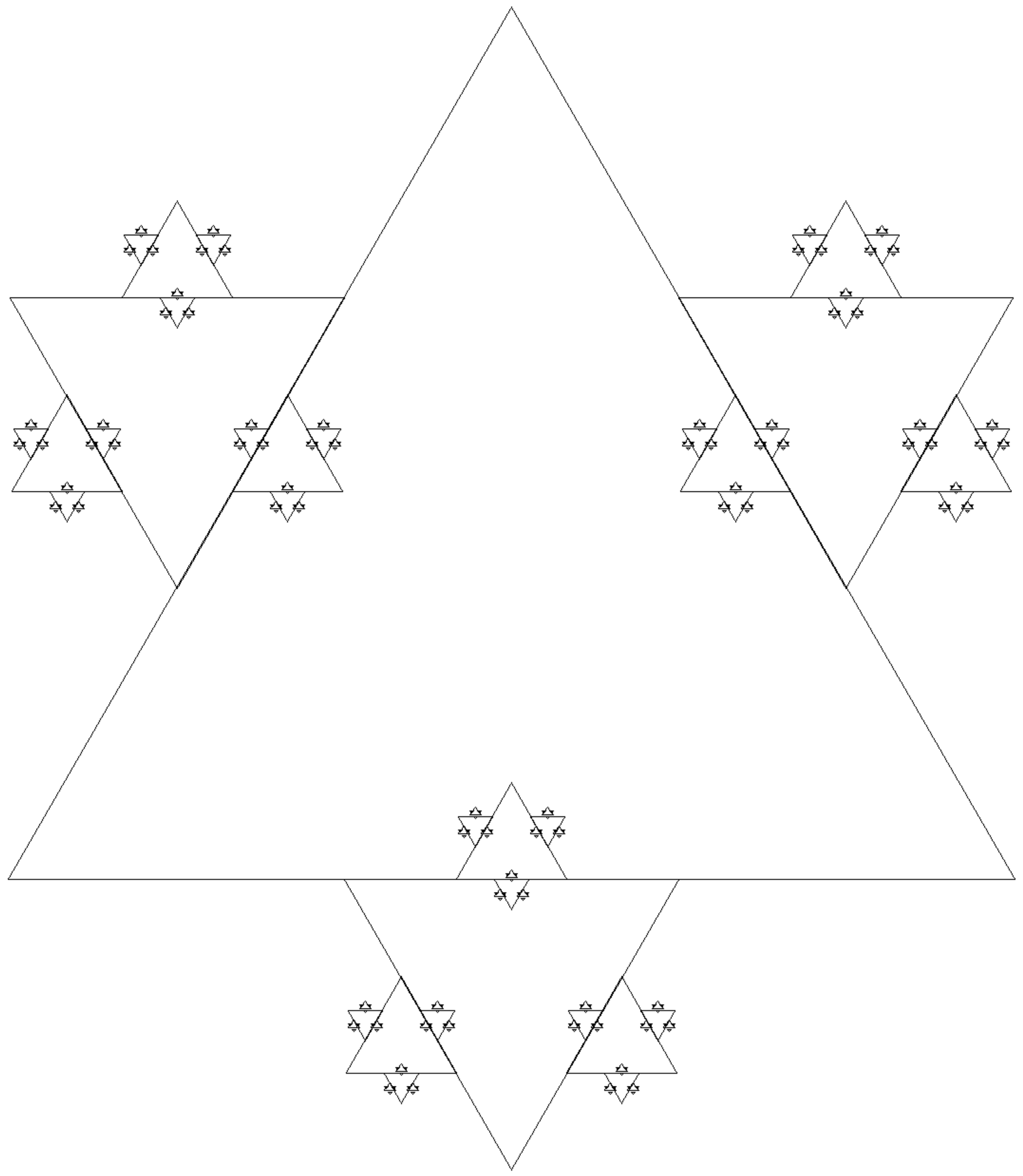
12



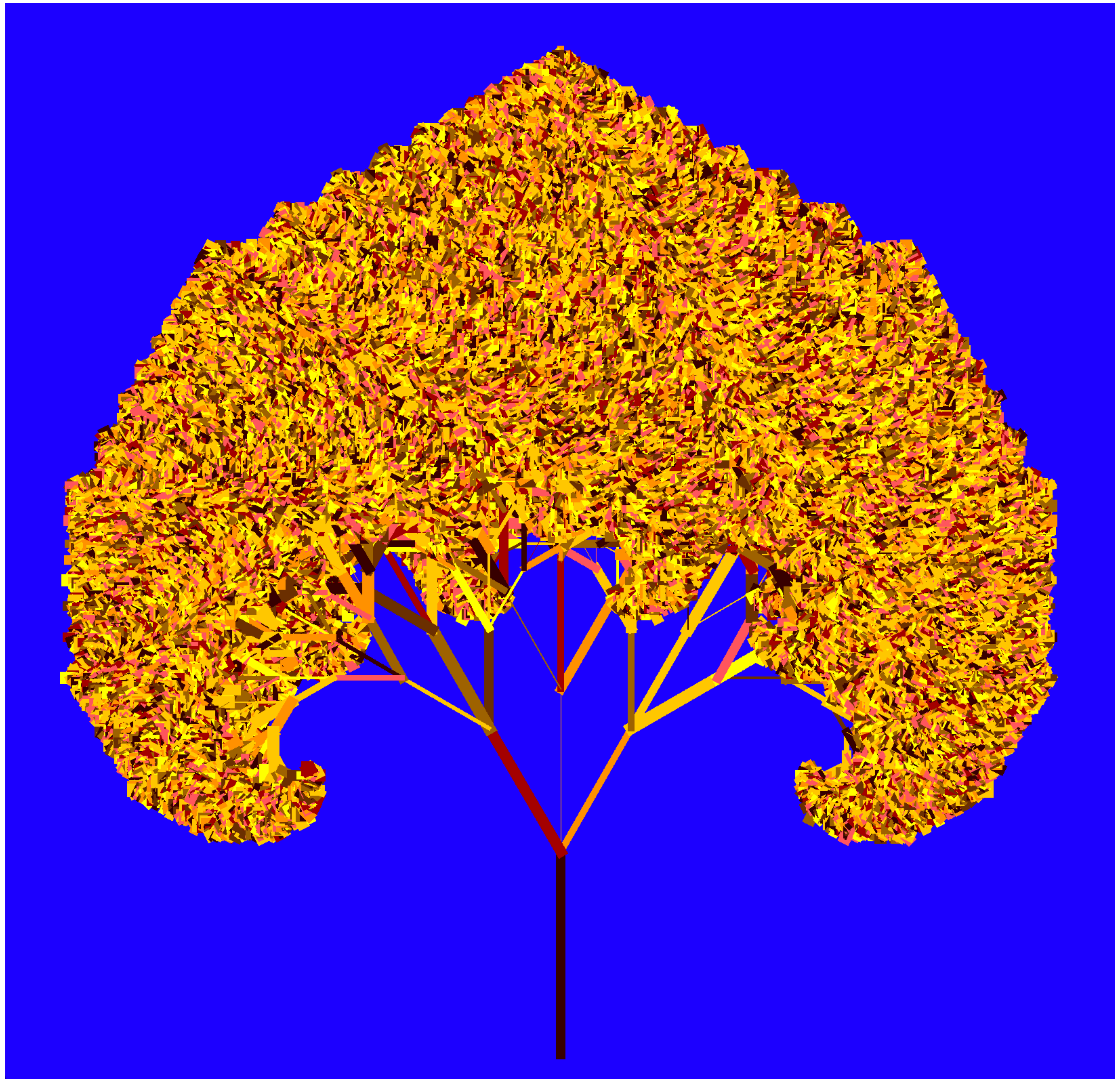
# 13




14



**15**



# Quick Sort

- Another divide-and conqueror sort
  - divide: pick a random element $x$ (pivot) and partition into
    - $L:\ < x$
    - $E: = x$
    - $G: > x$
  - conquer: sort L and $G$
  - combine: join L, $E$ and $G$

# Pseudo Code

```
quickSort(S):
  if (S.size()<2) return
  p = S.last()                     // first as pivot
  L = E = G = new list()
  partition(S, p)
  quickSort(L)
  quickSort(G)
  S = L+E+G
```

# QS in action

Partition (pink) | Divide

{7, 2, 9, 4, 3, 7, 1, 6}

Partition on 6

{2, 4, 3, 1} 6 {7, 7, 9}

Partition on 1 | Partition on 9

{} 1 {2, 4, 3} | {7, 7} 9 {}

Partition on 3 | Partition on 7

{2} 3 {4} | {7} 7 {}

Assemble (aqua) | Conquer

{2,3,4} {7,7}

{1,2,3,4} {7,7,9}

{1,2,3,4,6,7,7,9}

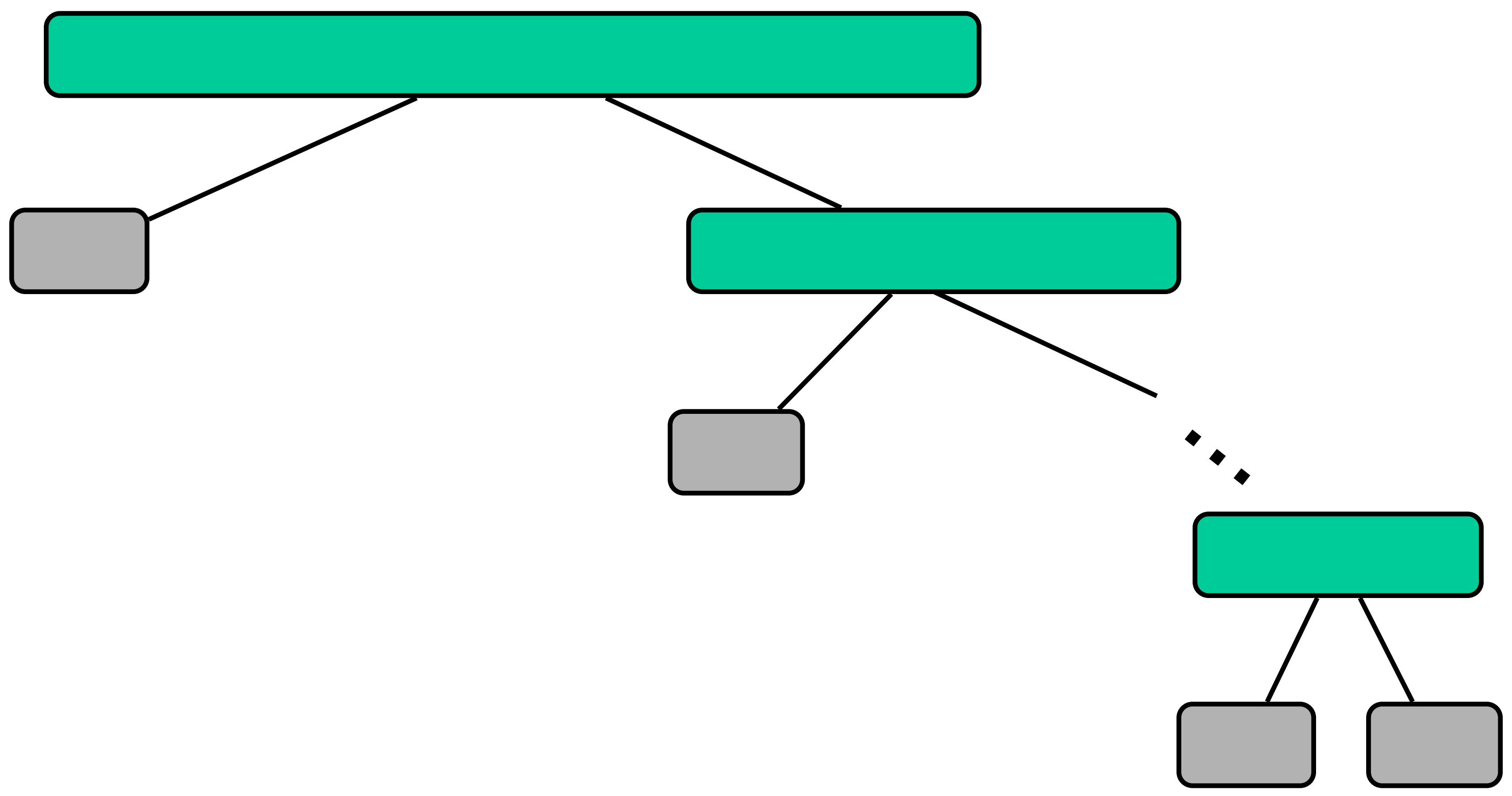
# MergeSort, Quicksort, etc

- Quicksort does work on way down in recursion
    - Mergesort does work on way up
- Insertion sort does work on way down
    - Selection sort on way up
- Which one in faster Quick or Merge?

# Worst-case Running Time

- When the pivot is the min or max
  - one of $L$ or $G$ has size $n - 1$
  - $T(n) = n + (n - 1) + \dots + 2 + 1 = O(n^2)$

| depth | time |
|---|---|
| 0 | $\boldsymbol{n}$ |
| 1 | $\boldsymbol{n} - 1$ |
| … | … |
| $\boldsymbol{n} - 1$ | 1 |



# In-place Quick Sort

- instead of three lists partition rearranges the input list
  - $L: [0, l-1]$
  - $E: [l, r]$
  - $G: [r+1, n-1]$
- Recursive calls on $[0, l-1]$ and $[r+1, n-1]$

# Partition

```
public int partition(int arr[], int begin, int end)  {
    int pivot = arr[end];
    int insertLoc = (begin-1);
    for (int j = begin; j < end; j++) {
        if (arr[j] <= pivot) {
            insertLoc++;
            int swapTemp = arr[insertLoc];
            arr[insertLoc] = arr[j];
            arr[j] = swapTemp;
        }
    }
    int swapTemp = arr[insertLoc+1];
    arr[insertLoc+1] = arr[end];
    arr[end] = swapTemp;
    return insertLoc+1;
}
```

# QuickSort

```
    private void quickSort(int arr[], int begin, int end)
    {
        if (begin < end) {
            int partitionIndex = partition(arr, begin, end);

            quickSort(arr, begin, partitionIndex-1);
            quickSort(arr, partitionIndex+1, end);
        }
    }
```

# Speed

Table 1

| size | Insertion | Heap | merge (improved) | Quick |
|---|---|---|---|---|
| **1000** | 11 | 2 | 2 | 1 |
| **2000** | 26 | 3 | 3 | 1 |
| **4000** | 20 | 5 | 7 | 2 |
| **8000** | 81 | 10 | 9 | 5 |
| **16000** | 315 | 17 | 16 | 13 |
| **32000** | 1218 | 36 | 32 | 30 |
| **64000** | 4605 | 77 | 69 | 59 |
| **128000** | 19849 | 161 | 143 | 108 |
| **256000** | | 345 | 294 | 219 |
| **512000** | | 1128 | 563 | 464 |
| **1024000** | | 1973 | 1191 | 955 |
| **2048000** | | 3225 | 2412 | 1989 |
| **4096000** | | 7577 | 5191 | 4148 |
| **8192000** | | 18586 | 10282 | 10101 |
| **16384000** | | | | 17614 |
| **32768000** | | | | 37291 |

# Quick and Merge

- Quicksort is reliably quicker than merge
- Quicksort does not need extra memory for auxiliary array

# Mini Homework

14, 6, 18, 2, 13, 7, 8, 9, 3, 17, 5, 10, 11, 12, 15, 19, 16, 0, 1, 4

For the data above, show all the steps of a quick sort, following the pattern of slide 19. Always choose the last element as the partitioning element.