Disjoint sets with union-by-rank and path compression.

Union is $O(1)$ amortized

Find is $\alpha(n)$ amortized

$\alpha$ grows very slowly, but not in $O(1)$
Sorting

Given an array of \( n \) items that are comparable.

Permute items within array so they appear in increasing order.
Insertion Sort

c a comparison sort
c

\[ n-1 \text{ passes} \]

At start of pass \( p \in \{1, \ldots, n-1\} \)

Assume positions 0 to \( p-1 \) contain the \( p \) smallest elements
in sorted order, if \( x \) in position \( p \), move elements forward one position until you sit \( x \).
Worst-case time is $\Theta(n^2)$.

Average case:

$$1 + 2 + 3 + \cdots + (n-1) = \Theta(n^2)$$

Reverse sorted order:

Best-case: $O(1)$

Sorted order:
Selection Sort

For each $p$ from 0 to $n-2$,
find min element from positions $\{p, \ldots, n-1\}$
move to position $p$.
Always $\Theta(n^2)$ time.
Improvement:

Instead of scanning for min, store everything in a binary heap and delete Min once per element.

Heap sort $O(n)$ to build heap
$O(\log n)$ per deleteMin
so $O(n \log n)$ total

Simple implementation
uses $\Theta(n)$ extra space to store heap.

Instead:

Use a max heap.
Build heap right on input array
each deleteMax call:
put element at position no longer used by heap
Merge sort

divide-and-conquer

1) Divide input into smaller instances of same problem.

2) Recursively solve smaller instances.

3) Combine recursive
Mergesort divides into first half or second half subarrays.

Mergesort subarrays, sorting example

original | A E I G

sorted | A E I G
Now we merge...

Smallest element is first in one of the subarrays. Remove that element from its subarray and put in beginning of output.

"Recursively" merge what remains.
public static <T extends Comparable<? super T>>
void mergeSort(T[] a) {
    T[] tmpArray = (T[]) new Comparable[a.length]; // Java does not like making generic arrays.
    mergeSort(a, tmpArray, 0, a.length - 1);
}

private static <T extends Comparable<? super T>>
void mergeSort(T[] a, T[] tmpArray, int left, int right) {
    if (left < right) { // Length 1 is a base case.
        int center = (left + right) / 2;
        mergeSort(a, tmpArray, left, center);
        mergeSort(a, tmpArray, center + 1, right);
        merge(a, tmpArray, left, center + 1, right);
    }
}

// Assumes each subarray is already sorted.
private static <T extends Comparable<? super T>>
void merge(T[] a, T[] tmpArray, int leftPos, int rightPos, int rightEnd) {
    int startPos = leftPos;
    int endPos = rightEnd;
    int leftEnd = rightPos - 1;
    int tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;

    // Elements remain from both halves.
    while (leftPos <= leftEnd && rightPos <= rightEnd) {
        if (a[leftPos].compareTo(a[rightPos]) <= 0) {
            tmpArray[tmpPos++] = a[leftPos++];
        } else {
            tmpArray[tmpPos++] = a[rightPos++];
        }
    }

    while (leftPos <= leftEnd) { // Copy rest of left half.
        tmp[tmpPos++] = a[leftPos++];
    }

    while (rightPos <= rightEnd) { // Copy rest of right half.
        tmp[tmpPos++] = a[rightPos++];
    }

    // Copy tmpArray back to a.
    for (int i = startPos; i <= endPos; i++) {
        a[i] = tmpArray[i];
    }
}
\( T(n) \): time to merge sort \( n \) elements

\[
T(n) = 2 \cdot T \left( \frac{n}{2} \right) + O(n)
\]

How to solve?
Recursion Trees

Rooted tree.
Each node corresponds to a subproblem.
Root is for the original instance.

Children of a node are its recursive calls
⇒ leaves are base cases
Each node has a value: the time spent on that subproblem not counting recursive calls.

\[ T(n) = 2T\left( \frac{n}{2} \right) + \Theta(n) \]

Want an upper bound on \( T(n) \): set \( t_1 = 1 \) for simplicity.
$T(n) = \sum \text{of node values}
= \sum \text{sum at level } i
\leq \sum \text{each level}
\leq \sum n
= \sum n
= \Theta(n)
= \Theta(\log n) \text{ levels} \Rightarrow
3^n = n
$
\( T(n) = O(n \log n) \)

\( T(n) = \Omega(n \log n) \)

\( \Rightarrow T(n) = \Theta(n \log n) \)
Typical Pattern

- \( f(n) \) works
- \( r \) recursive calls
- \( n/c \) sized subproblems

\[
T(n) = r T\left(\frac{n}{c}\right) + \sum f(n)
\]

(base cases are all \( \Theta(1) \))
levels inodes have value $s(n/c_i)$

$$\Rightarrow T(n) = \sum_{i=0}^{\log c(n)} r \cdot s(n/c_i)$$

height

$$L = \log_c n$$

all leaves: $r^L = r \log_c n = n^{\log_c r}$

Time for leaves is

$$\Theta(1), \quad n^{\log_c r} = \Theta(n^{\log_c r})$$
Often, \( T(n) = \sum_{i=0}^{n} r^i \cdot f(n) \).

Falls under one of these cases.

If sum is...
- geometric & decreasing, sum dominated by root value
  \( T(n) = \Theta(f(n)) \)
- all equal terms, then
  \( T(n) = \Theta(f(n) \cdot L) = \Theta(f(n) \log n) \)
- geometric & increasing, dominated by leaves, so
  \( T(n) = \Theta(n^{\log r}) \)